SOFTWARE REFERENCE MANUAL

Power PMAC

Power PMAC Software Reference Manual

050-PRPMAC-0S0

O015-E-01

October 25, 2016

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

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

Copyright Information

To report errors or inconsistencies, call or email:

Delta Tau Data Systems, Inc. Technical Support

Phone: (818) 717-5656

Fax: (818) 998-7807

Email: support@deltatau.com

Website: http://www.deltatau.com

Operating Conditions

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

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

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.

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

Warning

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

Caution

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

Note

REVISION HISTORY				
REV.	**DESCRIPTION**	**DATE**	**CHG**	**APPVD**
1	New Manual Generated	11/27/2011	SS	Curt Wilson
2	Manual updated for 1.5 version of firmware	08/10/2012	SS	Curt Wilson
3	Manual updated for 1.6 version of firmware	03/17/2014	SS	Curt Wilson
4	Manual updated for 2.0 version of firmware	12/05/2014	SS	Curt Wilson
5	Manual updated for 2.1 version of firmware	4/11/2016	SS	Curt Wilson
6	Added Omron Part Number	7/15/2016	Sgm	Sgm
7	Manual update for 2.2 version of firmware	10/25/2016	SS	Curt Wilson

Table of Contents

POWER PMAC COMMAND SYNTAX SUMMARY

This section provides an overview of the Power PMAC command syntax, both for "on-line" (immediately executed) commands, and for buffered program commands. Detailed descriptions of each command and mathematical feature are given in separate chapters.

Notes

- Power PMAC script-language syntax is not case sensitive.
- Spaces are not important in Power PMAC syntax, except where noted
- Characters and symbols shown in italics in the syntax description are not to be included in the actual command; the user must substitute something the items in italics represent
- ***{}*** – Item in italics inside ***{}*** can be replaced by anything fitting definition
- ***[]***– Item in italics inside ***[]*** is optional to syntax
- ***[{item}…]*** – Ellipses in syntax description indicate previous item may be repeated
- ***[..{item}]*** – The explicit periods are to be included in the syntax to specify a range
- **()** – Parentheses not in italics are to be included in syntax as they appear
- **[]** – Square brackets not in italics are to be included in syntax as they appear
- **{}** – Curly brackets not in italics are to be included in syntax as they appear

Definitions

- ***constant*** – Non-changeable numeric value
- ***variable*** – Entity that holds a changeable numeric value
- ***function*** – Mathematical process that operates on one or more arguments inside following parentheses and returns a numeric value
- ***operator*** – Performs mathematical or logical operation on preceding and following values
- ***expression*** – Coherent combination of constants, variables, functions, and operators that evaluates to a numerical value
- ***assignment operator*** – Mathematical symbol that causes an assignment of a value to the preceding variable, with either "standard" or "synchronous" assignment
- ***standard assignment operator*** – Assignment operator that causes assignment at the time the statement is evaluated
- ***synchronous assignment operator*** – Assignment operator that causes the actual assignment to be delayed until the beginning of execution of the next programmed move
- ***data*** – Syntax specification that represents a constant without parentheses or an expression inside parentheses
- ***comparator*** – Evaluates relative values of preceding and following expressions
- ***condition*** – Evaluates as true or false based on comparison of expression values
- ***list*** – Set of integer constants that can represent multiple numbered entities as a group. Can be motor list, coordinate system list, or variable list
- ***motor list*** – Set of multiple integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, representing the motors to be acted on by the following command
- ***C.S. list*** – Set of multiple integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, representing the coordinate systems to be acted on by the following command

- ***variable list*** – Set of three integer constants separated by commas and representing a group of evenly spaced variable numbers, or a pair of integer constants separated by two periods representing a consecutive range of variable numbers
- ***axis*** – Letter or double letter denoting a programming axis name
- ***vector*** – Letter or double letter denoting a vector component name

Mathematical Elements

Mathematical operators and functions can be used in both on-line and buffered program commands.

Arithmetic Operators

+ – Addition operator
- – Subtraction or negation operator
***** – Multiplication operator
/ – Division operator
% – Modulo operator

Bit-by-Bit Logical Operators

& – Bit-by-bit "and" operator
| – Bit-by-bit "or" operator
^ – Bit-by-bit "exclusive-or" operator
~ – Bit-by-bit "invert" unary operator
<< – Shift-left operator
>> – Shift-right operator

Standard Assignment Operators

= – Standard value assignment operator
+= – Assignment with addition operator
-= – Assignment with subtraction operator
***=** – Assignment with multiplication operator
/= – Assignment with division operator
%= – Assignment with modulo operator
&= – Assignment with bit-by-bit "and" operator
|= – Assignment with bit-by-bit "or" operator
^= – Assignment with bit-by-bit "exclusive" operator
>>= – Assignment with shift-right operator
<<= – Assignment with shift-left operator
++ – Increment assignment operator
-- – Decrement assignment operator

Synchronous Assignment Operators

== – Synchronous value assignment operator
+== – Synchronous assignment with addition operator
-== – Synchronous assignment with subtraction operator

***==** – Synchronous assignment with multiplication operator
/== – Synchronous assignment with division operator
&== – Synchronous assignment with bit-by-bit "and" operator
|== – Synchronous assignment with bit-by-bit "or" operator
^== – Synchronous assignment with bit-by-bit "exclusive" operator
++= – Synchronous increment assignment operator
--= – Synchronous decrement assignment operator

Conditional Comparators

== – Equality comparator
!= – Inequality comparator
< – Less-than comparator
> – Greater-than comparator
<= – Less-than-or-equal-to comparator
>= – Greater-than-or-equal-to comparator
~ – Approximate equality comparator
!~ – Approximate inequality comparator
<> – Alternate inequality comparator
!> – Alternate less-than-or-equal-to comparator
!< – Alternate greater-than-or-equal-to comparator

Conditional Combinatorial Operators

&& – Logical "and" operator
|| – Logical "or" operator

Scalar Mathematical Functions

abs – Absolute value
acos – Trigonometric arc-cosine, result in radians
acosd – Trigonometric arc-cosine, result in degrees
acosh – Hyperbolic arc-cosine
asin – Trigonometric arc-sine, result in radians
asind – Trigonometric arc-sine, result in degrees
asinh – Hyperbolic arc-sine
atan – Trigonometric arc-tangent, result in radians
atand – Trigonometric arc-tangent, result in degrees
atan2 – Trigonometric 2-argument arc-tangent, result in radians
atan2d – Trigonometric 2-argument arc-tangent, result in degrees
atanh – Hyperbolic arc-tangent
cbrt – Cube root
ceil – Round up to integer value
cos – Trigonometric cosine, argument in radians
cosd – Trigonometric cosine, argument in degrees
cosh – Hyperbolic cosine
exp – Exponentiation base e (e^x)

exp2 – Exponentiation base 2 (2^x)
floor – Round down to integer value
int – Truncation to 32-bit integer
isnan – Check for "not a number"
ln – Natural logarithm (alternate syntax)
log – Natural logarithm
log10 –Logarithm base 10
log2 –Logarithm base 2
madd – Combined multiply and add
pow – Generalized exponentiation (x^y)
qnrt – Quintic (5^{th}) root
qrrt – Quartic (4^{th}) root
rem – Remainder
randx – 64-bit random number generator
rint – Round to nearest integer
rnd – 32-bit random number generator
seed – Random number seed generator
sin – Trigonometric sine, argument in radians
sind – Trigonometric sine, argument in degrees
sinh – Hyperbolic sine
sincos – Combined trigonometric sine and cosine, argument in radians
sincosd – Combined trigonometric sine and cosine, argument in degrees
sgn – Algebraic sign
sqrt – Square root
tan – Trigonometric tangent, argument in radians
tand – Trigonometric tangent, argument in degrees
tanh – Hyperbolic tangent

Vector Mathematical Functions

sum – Sum of vector elements
sumprod – Sum of products of vector elements
vcopy – Vector copy
vmadd – Vector multiply and add
vscale – Vector scale

Matrix Mathematical Functions

mdet – Matrix determinant
minv – Matrix inverse
minv – Matrix minor determinant
mmadd – Matrix multiply and add
mmul – Matrix multiply
msolve – Matrix solve
mtrans – Matrix transpose

Transformation Matrix Functions

tinit – Initialize selected axis transformation to identity matrix
tprop – Propagate transformation by matrix multiplication and addition

Character (Byte) Buffer Functions

memcpy – Copy character buffer memory contents
memset – Set character buffer memory to specified character value

String Manipulation Functions

sprintf – Create formatted string variable
strcpy – Copy string variable (from other variable or literal)
strncpy – Copy limited-length string variable (from other variable)
strtolower – Copy limited-length string variable, converting upper case to lower
strtoupper – Copy limited-length string variable, converting lower case to upper
strcat – Add to end of string variable (from other variable or literal)
strncat – Add limited-length text to end of string variable (from other variable)
strchr – Search for first instance of character in string variable
strrchr – Search for last instance of character in string variable
strcmp – Compare string variable (to other variable or literal)
strncmp – Compare limited-length string variable (to other variable)
strspn – Find longest substring of specified characters in string variable
strcspn – Find longest substring without specified characters in string variable
strlen – Find number of characters in string variable
strpbrk – Find first incidence of any of specified characters in string variable
strstr – Find first incidence of another string in string variable
strtod – Decode numerical value in string variable into double-precision value

On-Line Commands

"On-line" commands are executed immediately by the Power PMAC, then discarded. It is not possible to list back a sequence of on-line commands that have been sent to the Power PMAC.

On-Line Global Commands

The action of on-line global commands is not dependent on the presently addressed motor or coordinate system, or the immediately preceding motor or coordinate-system list.

Addressing Mode Commands

Addressing mode commands are thread-specific, only affecting the individual communications thread in which they were issued.

#*{constant}* – Make specified motor the currently modally addressed motor
– Report current modally addressed motor number to host
&*{constant}* – Make specified coordinate system the currently addressed coordinate system
& – Report current modally addressed coordinate system to host

Addressing Non-Modal Commands

#*{list}* – Make the immediately following motor-specific command act on all motors in list
#* – Make the immediately following motor-specific command act on all motors in Power PMAC
&*{list}* – Make the immediately following coordinate-system-specific command act on all coordinate systems in list
&* – Make the immediately following coordinate-system-specific command act on all coordinate systems in Power PMAC

General Global Commands

$$$ – Reset Power PMAC control application, restoring saved setup values
$$$*** – Reset and re-initialize Power PMAC control application, using factory default setup values
reboot – Restart Power PMAC computer and control application
save – Copy active memory settings into non-volatile flash memory
fsave – Copy limited set of active memory settings into non-volatile flash memory
fload – Reload limited set of active memory settings from non-volatile flash memory
backup – Report present active values of saved setup elements
#*{constant}*-> – Report axis definition of specified motor
undefine all – Erase axis definition of all coordinate systems
cx – Execute immediately following command(s) as single-line PLC program
echo*{constant}* – Set command echo mode
echo – Report present command echo mode
string*{constant}* – Report contents of string variable starting at specified location in user shared memory buffer

Global Configuration Query Commands

cpu – Report processor type
date – Report release date of firmware version used
vers – Report revision number of firmware version used
size – Report total memory reserved for several user buffers
free – Report available (unused) memory reserved for several user buffers
brickacver – Report revision number of Power Brick AC amplifier firmware version used
bricklvver – Report revision number of Power Brick LV amplifier firmware version used

Global Variable Commands

D*{data}* – Report value of D-variable to host
D*{data}*=*{expression}* – Set value of D-variable to value of expression
D*{list}* – Report value of D-variable(s) to host
D*{list}*=*{expression}* – Set value of D-variable(s) to value of expression
I*{data}* – Report value of I-variable to host
I*{data}*-> – Report definition of I-variable to host
I*{data}*=*{expression}* – Set value of I-variable to value of expression
I*{list}* – Report value of I-variable(s) to host
I*{list}*-> – Report definition of I-variable(s) to host
I*{list}*=*{expression}* – Set value of I-variable(s) to value of expression
L*{data}* – Report value of L-variable to host
L*{data}*=*{expression}* – Set value of L-variable to value of expression
L*{list}* – Report value of L-variable(s) to host
L*{list}*=*{expression}* – Set value of L-variable(s) to value of expression
M*{data}* – Report value of M-variable to host
M*{data}*-> – Report definition of M-variable to host
M*{data}*=*{expression}* – Set value of M-variable to value of expression
M*{list}* – Report value of M-variable(s) to host
M*{list}*-> – Report definition of M-variable(s) to host
M*{list}*=*{expression}* – Set value of M-variable(s) to value of expression
P*{data}* – Report value of P-variable to host
P*{data}*=*{expression}* – Set value of P-variable to value of expression
P*{list}* – Report value of P-variable(s) to host
P*{list}*=*{expression}* – Set value of P-variable(s) to value of expression
R*{data}* – Report value of R-variable to host
R*{data}*=*{expression}* – Set value of R-variable to value of expression
R*{list}* – Report value of R-variable(s) to host
R*{list}*=*{expression}* – Set value of R-variable(s) to value of expression
{data structure element} – Report value of specified data structure element to host
{data structure element}.a – Report address of specified data structure element to host
{data structure element}={expression} – Set value of specified data structure element to value of expression

Buffer Control Commands

open prog*{constant}[,[{constant}]][,{constant}]* – Open specified motion program buffer in active memory for entry, optionally specifying non-default stack offset, line-jump label table size

open plc*{constant}[,[{constant}]][,{constant}]* – Open specified PLC program buffer in active memory for entry, optionally specifying non-default stack offset, line-jump label table size

open subprog*{constant}[,[{constant}]][,{constant}]* – Open specified subprogram buffer in active memory for entry, optionally specifying non-default stack offset, line-jump label table size

close – Close currently opened program buffer

close all buffers – Close opened program buffer, even if it had been opened on another communications thread

delete all lookahead – Delete lookahead buffers for all coordinate systems

delete all rotary – Delete rotary motion program buffers for all coordinate systems

clear gather – Erase contents of servo data gathering buffer

clear phase gather – Erase contents of phase data gathering buffer

Buffer Query Commands

list prog*{constant}[,[{constant}]][,{constant}]* – Report contents of specified motion program buffer in active memory, optionally specifying starting line number and number of lines to report

list plc*{constant}[,[{constant}]][,{constant}]* – Report contents of specified PLC program buffer in active memory, optionally specifying starting line number and number of lines to report

list subprog*{constant}[,[{constant}]][,{constant}]* – Report contents of specified subprogram buffer in active memory, optionally specifying starting line number and number of lines to report

Script PLC Program-Control Commands

enable plc*{list}* – Start execution of specified PLC program(s) at beginning of scan

disable plc*{list}* – Stop execution of specified PLC program(s) at end of scan

pause plc*{list}* – Stop execution of specified PLC program(s) at present execution point

resume plc*{list}* – Start execution of specified PLC program(s) at paused point

C PLC Program-Control Commands

enable bgcplc*{list}* – Start execution of specified background C PLC program(s)

enable rticplc – Start execution of foreground C PLC program

disable bgcplc*{list}* – Stop execution of specified background C PLC program(s)

disable rticplc – Stop execution of foreground C PLC program

On-Line Coordinate-System Commands

Except where specifically noted, coordinate-system commands act on the (single) addressed coordinate system unless immediately preceded by a list of multiple coordinate systems (e.g. **&1,2,4r**). Note that a list of multiple coordinate systems does not affect the modally addressed coordinate system. To address all coordinate systems simultaneously, the **&*** syntax can be used (e.g. **&*a** will abort programs in all coordinate systems).

Axis-Definition Commands

Note that axis-definition commands cannot be preceded by a list of multiple coordinate systems

#*{constant}*->*{axis definition}* – Assign specified motor to one or more axes with possible scale factors and offset

Examples:

```
#1->X
#1->3333.333C
#1->8660X-5000Y
#2->5000X+8660Y+5000
```

#*{constant}*->I– Assign specified motor as inverse kinematic axis
#*{constant}*->S*[{constant}]* – Assign specified motor as spindle axis
#*{constant}*->0 – Declare "null" definition of specified motor in this C. S.
undefine – Erase axis definition of all motors in this C. S.

Time-Base Commands

%*{constant}* – Set commanded time base value for addressed [or listed] coordinate system[s]
%– Report present time base value for addressed [or listed] coordinate system[s]

Motor-Control Commands

enable – Enable servo control of all motors assigned to axes in this coordinate system (closed-loop, amplifier enabled)
disable – Disable servo control of all motors assigned to this coordinate system (open-loop, amplifier disabled)
ddisable – Delayed disable of servo control for all motors assigned to this coordinate system (open-loop, amplifier disabled), providing time for brakes to engage fully
adisable – Controlled stop of all motors in coordinate system (equivalent of abort) followed by delayed disable of servo control (open-loop, amplifier disabled), providing time for brakes to engage fully

Program-Control Commands

r – Run motion program[s] in addressed [or listed] coordinate system[s] from presently selected point (long-form syntax: **run**)
q – Quit motion program calculation in addressed [or listed] coordinate system[s] at presently selected point, ready to resume at this point on **r** or **s** command (long-form syntax: **pause**)
s*[{constant}]* – (Step) Start motion program[s] in addressed [or listed] coordinate system[s] from presently selected point and pause after one [or specified number of] line[s] (long-form syntax: **step**)

b*[{constant}]* – Set program counter to beginning of specified motion program (or to specified line jump label if constant has fractional component), ready to run (long-form syntax: **begin**)
stop – Stop calculation of motion program[s] in addressed [or listed] coordinate system[s] from presently selected point and make the selected point the beginning of the present program
start*[{constant}]* – Start running present [or specified] motion program in addressed [or listed] coordinate system[s] from beginning
a – (Abort) Stop calculation of motion program[s] in addressed [or listed] coordinate system[s] from presently selected point, commencing immediate deceleration of all motors in the coordinate system[s] (long-form syntax: **abort**)
h – (Hold) Execute feed hold in addressed [or listed] coordinate system[s] by ramping coordinate system time base value to 0, ready to resume on **r** or **s** command (long-form syntax: **hold**)
**** – (Quick stop) Execute fastest stop in lookahead buffer that does not violate acceleration constraints for addressed [or listed] coordinate system[s]
< – (Reverse) Start reverse execution in lookahead buffer for addressed [or listed] coordinate system[s]
> – (Forward) Resume forward execution in lookahead buffer for addressed [or listed] coordinate system[s]
cpx – Execute immediately following command(s) as single-line motion program

Reporting Commands

Note: For the following coordinate-system reporting commands, the command must be immediately preceded by a coordinate-system specifier. If no coordinate system is specified, the command is treated as a motor reporting command.

&*{list}*p – Report actual positions of all active axes of coordinate systems in list in user units
&*{list}*d – Report desired positions of all active axes of coordinate systems in list in user units
&*{list}*v – Report actual velocities of all active axes of coordinate systems in list in user units
&*{list}*f – Report following errors of all active axes of coordinate systems in list in user units
&*{list}*t – Report target positions of all buffered axes of coordinate systems in list in user units
&*{list}*g – Report distance-to-go in move of all buffered axes of coordinate systems in list in user units
&*{list}*? – Report value of status words of coordinate systems

Coordinate-System Variable Commands

Q*{data}* – Report value of Q-variable for addressed C.S. to host
Q*{data}={expression}* – Set value of Q-variable for addressed C.S. using expression and assignment operator
Q*{list}* – Report value of Q-variable(s) for addressed C.S. to host
Q*{list}={expression}* – Set value of Q-variable(s) for addressed C.S. to value of expression

Buffer Control Commands

open forward*[,[{constant}]][,{constant}]* – Open forward-kinematic buffer for addressed C.S. in active memory for entry, optionally specifying non-default stack offset, line-jump label table size
open inverse*[,[{constant}]][,{constant}]* – Open inverse-kinematic buffer for addressed C.S. in active memory for entry, optionally specifying non-default stack offset, line-jump label table size

define rotary ***{constant}[,{constant}]*** – Define rotary motion program buffer for addressed C.S. of byte size specified by first constant, optionally specifying non-default byte size of buffer for processing each incoming line
open rotary – Open already-defined rotary motion-program buffer for addressed C.S for entry
close – Close currently opened fixed program buffer
close rotary – Close currently opened rotary motion-program buffer
clear rotary – Erase contents of rotary motion-program buffer for addressed C.S, but maintain buffer itself
clear prog ***{constant}*** – Erase fixed motion program buffer
clear plc ***{constant}*** – Erase PLC program buffer
rotfree – Report number of unexecuted lines in rotary motion-program buffer for addressed C.S
delete rotary – Eliminate rotary motion-program buffer for addressed C.S.
define lookahead ***{constant}*** – Define lookahead buffer for addressed C.S. of specified number of motion segments
delete lookahead – Eliminate lookahead buffer for addressed C.S.

Buffer Query Commands

list forward*[,[{constant}]][,{constant}]* – Report contents of forward-kinematic buffer for addressed C.S. in active memory, optionally specifying starting line number and number of lines
list inverse*[,[{constant}]][,{constant}]* – Report contents of inverse-kinematic buffer for addressed C.S. in active memory, optionally specifying starting line number and number of lines
list rotary – Report unexecuted contents of rotary motion-program buffer for addressed C.S
list pc – Report contents of present motion program from point of program counter
list apc – Report contents of aborted motion program from point of program counter at time of abort (commanded or fault)
rotfree – Report size of largest continuous block of free memory in rotary motion program buffer
rotfreeall – Report size of blocks of free memory in rotary motion program buffer

Axis Commands

pmatch – Calculate starting axis positions for next move from present motor positions
pstore – Save present set of **pset** axis/motor offsets for later recovery with **pload**
pload – Restore set of **pset** axis/motor offsets that were saved with latest **pstore**

On-Line Motor Commands

Motor commands act on the (single) addressed motor unless immediately preceded by a list of multiple motors (e.g. **#1..5hm**). Note that a list of multiple motors does not affect the modally addressed motor. To address all motors simultaneously, the **#*** syntax can be used (e.g. **#*k** will kill all motors).

Reporting commands

p – Report actual positions of addressed [or listed] motor[s] in motor units
d – Report desired positions of addressed [or listed] motor[s] in motor units
v – Report actual velocities of addressed [or listed] motor[s] in motor units/msec
f – Report following errors of addressed [or listed] motor[s] in motor units
#***{list}*****?** – Report value of status word for listed motor[s]

Jogging Commands

j+ – Jog addressed [or listed] motor[s] indefinitely in positive direction (long-form syntax: **jog+**)
j- – Jog addressed [or listed] motor[s] indefinitely in negative direction (long-form syntax: **jog-**)
j/ – Stop jogging addressed [or listed] motor[s]; also restore to position control (long-form syntax: **jog/**)
j= – Jog addressed [or listed] motor[s] to last programmed (pre-jog) position (long-form syntax: **jog=**)
j=*{constant}* – Jog addressed [or listed] motor[s] to specified position (long-form syntax: **jog=*{constant}***)
j==*{constant}* – Jog addressed [or listed] motor[s] to specified position, making that position the new "pre-jog" position position (long-form syntax: **jog==*{constant}***)
j=* – Jog addressed [or listed] motor[s] to position specified in **ProgJogPos** element position (long-form syntax: **jog=***)
j:*{constant}* – Jog addressed [or listed] motor[s] specified distance from current commanded position position (long-form syntax: **jog:*{constant}***)
j:* – Jog addressed [or listed] motor[s] the distance specified in **ProgJogPos** element from current commanded position position (long-form syntax: **jog:***)
j^*{constant}* – Jog addressed [or listed] motor[s] specified distance from current actual position position (long-form syntax: **jog^*{constant}***)
j^* – Jog addressed [or listed] motor[s] the distance specified in **ProgJogPos** element from current actual position position (long-form syntax: **jog^***)
jog – Start indefinite jog move for addressed [or listed] motor[s] at present actual velocity (usually from open loop move)

Jog-Until-Trigger Commands

j=^*{constant}* – Jog addressed [or listed] motor[s] to last programmed (pre-jog) position; if trigger occurs, jog instead to specified distance from trigger position position (long-form syntax: **jog=^*{constant}***)
j=*{constant}*^*{constant}* – Jog addressed [or listed] motor[s] to specified position; if trigger occurs, jog instead to specified distance from trigger position position (long-form syntax: **jog=*{constant}*^*{constant}***)
j=*^*{constant}* – Jog addressed [or listed] motor[s] to position specified in **ProgJogPos** element; if trigger occurs, jog instead to specified distance from trigger position (long-form syntax: **jog=*^*{constant}***)
j:*{constant}*^*{constant}* – Jog addressed [or listed] motor[s] specified distance from current commanded position; if trigger occurs, jog instead to specified distance from trigger position (long-form syntax: **jog:*{constant}*^*{constant}***)
j:*^*{constant}* – Jog addressed [or listed] motor[s] the distance specified in **ProgJogPos** element from current commanded position; if trigger occurs, jog instead to specified distance from trigger position (long-form syntax: **jog:*^*{constant}***)
j^*{constant}*^*{constant}* – Jog addressed [or listed] motor[s] specified distance from current actual position; if trigger occurs, jog instead to specified distance from trigger position (long-form syntax: **jog^*{constant}*^*{constant}***)
j^*^*{constant}* – Jog addressed [or listed] motor[s] the distance specified in **ProgJogPos** element from current actual position; if trigger occurs, jog instead to specified distance from trigger position (long-form syntax: **jog^*^*{constant}***)

General Motor Commands

$ – Establish phase reference and optionally close loop for listed motor[s]
hm – Perform homing search move for addressed [or listed] motor[s]
hmz – Set present commanded position for addressed [or listed] motor[s] to zero, or read absolute position sensor to establish position reference
k – Kill output for addressed [or listed] motor[s]
dkill – Delayed kill of servo control for addressed [or listed] motor[s] (open-loop, amplifier disabled), providing time for brakes to engage fully
out*{constant}* – Set open-loop servo output of specified magnitude for addressed [or listed] motor[s]

Buffered Script Program Commands

These buffered commands can be used in both motion programs and PLC programs

Move Commands

{axis}{data}[{axis}{data}…] – Simple position movement command; can be used in LINEAR, RAPID, or SPLINE modes
.....Example: **X1000 Y(P1) Z(P2*P3)**
{axis}{data}:{data}[{axis}{data}:{data}…] – Position/velocity move command; to be used only in PVT mode
.....Example: **X5000:750 Y3500:(P3) A(P5+P6):100**
{axis}{data}^{data}[{axis}{data}^{data}…] – Move-until-trigger command, to be used only in RAPID mode
{axis}{data}[{axis}{data}…][{vector}{data}...] – Arc move command; to be used only in CIRCLE mode; vector is to circle center
.....Examples: **X2000 Y3000 Z1000 I500 J300 K500**
..... **XX20 YY15 II10 JJ5**
{axis}{data}[{axis}{data}…] **R*{data}*** -- Arc move command; to be used only in CIRCLE mode on main Cartesian axes; R-value is radius magnitude
.....Example: **X2000 Y3000 Z1000 R500**
dwell*{data}* – Zero-distance command; fixed time base
delay*{data}* – Zero-distance command; variable time base

Move Mode Commands

linear – Set blended linear interpolation move mode
rapid – Set minimum-time point-to-point move mode
circle1 – Set clockwise circular interpolation move mode for main Cartesian axes (X/Y/Z)
circle2 – Set counterclockwise circular interpolation move mode for main Cartesian axes (X/Y/Z)
circle3 – Set clockwise circular interpolation move mode for alternate Cartesian axes (XX/YY/ZZ)
circle4 – Set counterclockwise circular interpolation move mode for alternate Cartesian axes (XX/YY/ZZ)
pvt*{data}* – Set position/velocity/time move mode (parabolic velocity profiles), specifying move time
spline*{data}*[spline*{data}*[spline*{data}*]] – Set cubic B-spline move mode, specifying spline segment time(s)

Axis Attribute Commands

abs[({axis}[,{axis},…])] – Set absolute move mode for all [or specified] axes
inc[({axis}[,{axis},…])] – Set incremental move mode for all [or specified] axes
abs({vector list}) – Set absolute circle radius mode for vector components
inc({vector list}) – Set incremental circle radius mode for vector components
frax[({axis}[,{axis}…])] – Set default [or specified] axes as vector feedrate axes
frax2[({axis}[,{axis}…])] – Set standard [or specified] axes as secondary vector feedrate axes
normal*{vector}{data}*[*{vector}{data}*…] – Specify normal vector to plane for circular moves and cutter compensation
pset*{axis}{data}[{axis}{data}…]* – Set axis position(s) to specified value(s)
pstore – Save present axis/motor offsets for later restoring
pload – Recover last saved axis/motor offsets for use
pclr – Set all axis positions in coordinate system to zero

pmatch – Calculate starting axis positions for next programmed move from present motor positions

Move Attribute Commands

F*{data}* – Specify move speed for linear and circle moves
tm*{data}* – Specify move time for linear and circle moves
ta*{data}* – Specify move acceleration/deceleration time for linear and circle moves
td*{data}* – Specify separate final deceleration time for linear and circle moves
ts*{data}* – Specify acceleration/deceleration S-curve time for linear and circle moves
nxyz*{vector}{data}[{vector}{data}…]* – Specify surface-normal vector for 3D cutter compensation
txyz*{vector}{data}[{vector}{data}…]* – Specify tool-orientation vector for 3D cutter compensation

Cutter (Tool) Radius Compensation Commands

ccmode0 – Turn off (2D or 3D) cutter radius compensation
ccmode1 – Turn on 2D cutter radius compensation to the left
ccmode2 – Turn on 2D cutter radius compensation to the right
ccmode3 – Turn on 3D cutter radius compensation
ccr*{data}* – Specify 2D cutter compensation radius
txyzscale*{data}* – Specify transformation rescaling factor for CC radius and feedrate

Variable Assignment Commands

I*{data}{assignment operator}{expression}* – Assign expression value to specified I-variable using assignment operator
P*{data}{assignment operator}{expression}* – Assign expression value to specified P-variable using assignment operator
Q*{data}{assignment operator}{expression}* – Assign expression value to specified Q-variable using assignment operator
M*{data}{assignment operator}{expression}* – Assign expression value to specified M-variable using assignment operator
L*{data}{assignment operator}{expression}* – Assign expression value to specified L-variable using assignment operator
R*{data}{assignment operator}{expression}* – Assign expression value to specified R-variable using assignment operator
C*{data}{assignment operator}{expression}* – Assign expression value to specified C-variable using assignment operator
D*{data}{assignment operator}{expression}* – Assign expression value to specified D-variable using assignment operator
{data structure element}{assignment operator}{expression} – Set value of specified data structure element using expression and assignment operator
N*{data}* – Assign value of ***{data}*** to **Coord[*x*].Ncalc** at program calculation time, to **Coord[*x*].Nsync** at subsequent move execution time

Program Logic Control

N*{constant}*: – Numeric line label for destination of program jump commands
goto*{data}* – Jump to specified numeric line label in same program, no return

gosub*{data}* – Jump to specified numeric line label in same program, with jump back on return; no passing of local variables to subroutine

callsub*{data}* – Jump to specified numeric line label in same program, with jump back on return; passing of local variables to subroutine possible

call*{data}* – Jump to specified subprogram [and numeric line label], with jump back on return; passing of local variables to subroutine possible

return – Jump back to program line that called this subroutine and continue program execution

read(*{letter}[{letter}][,{letter}[{letter}…]***)** – Read argument (value following specified letter or double letter) into subroutine/subprogram from calling line

G*{data}* – **G***nnn[.mmm]* interpreted as **CALL***(Coord[x].Gprog).nnnmmm* (the program specified by **Coord***x***].Gprog** provides subroutines for desired G-Code actions)

M*{data}* – **M***nnn[.mmm]* interpreted as **CALL***(Coord[x].Mprog).nnnmmm* (the program specified by **Coord[***x***].Mprog** provides subroutines for desired M-Code actions)

T*{data}* – **T***nnn[.mmm]* interpreted as **CALL***(Coord[x].Tprog).nnnmmm* (the program specified by **Coord[***x***].Tprog** provides subroutines for desired T-Code actions)

D*{data}* – **D***nnn[.mmm]* interpreted as **CALL***(Coord[x].Dprog).nnnmmm* (the program specified by **Coord[***x***].Dprog** provides subroutines for desired D-Code actions)

if(*{condition}***)***{command}[{command}…]* – Conditionally execute single-line action

```
if({condition})
{
  {command}
  [{command}…]
}
```
– Conditionally execute multiple-line action

else*{command}[{command}…]* – Execute single-line action on preceding false IF condition

```
else
{
  {command}
  [{command}…]
}
```
– Execute multiple-line action on preceding false IF condition

while(*{condition}***)***{command}[{command}…]* – Conditionally repeat single-line action

```
while({condition})
{
  {command}
  [{command}…]
}
```
– Conditionally repeat multiple-line action

do *{command}[{command}…]* **while(***{condition}***)** – Conditionally repeat single-line action, always executing once

```
do
{
  {command}
  [{command}…]
}
```

while(*{condition}*) – Conditionally repeat multiple-line action, always executing once

```
switch({expression})
{
  case{constant}:{command}[{command}…][break}]
  [case{constant}:{command}[{command}…][break}]
  [default:{command}[{command}…]
}
```
– Multi-branch conditional execution structure

cset*{constant}* – Set specified conditional execution flag
cclr*{constant}* – Clear specified conditional execution flag
cexec*{constant}* – Conditionally execute rest of line if specified flag is set
cskip*{constant}* – Conditionally skip rest of line if specified flag is set
ccall*{constant}* – Conditionally call *n*th defined subprogram
cdef*{constant}{subprogram call}* – Define *n*th conditional subprogram call
cundef*{constant}* – Undefine *n*th conditional subprogram call

Script PLC Execution Control

enable plc*{list}* – Start execution of specified PLC program(s) at beginning of scan
disable plc*{list}* – Stop execution of specified PLC program(s) at end of scan
pause plc*{list}* – Stop execution of specified PLC program(s) at present execution point
resume plc*{list}* – Start execution of specified PLC program(s) at paused point

C PLC Execution Control

enable bgcplc*{list}* – Start execution of specified background C PLC program(s)
enable rticplc – Start execution of foreground C PLC program
disable bgcplc*{list}* – Stop execution of specified background C PLC program(s)
disable rticplc – Stop execution of foreground C PLC program

Port Communications

send*{constant}*,"*{string}*"[,*{expression}*…] – Send formatted text message to specified port

Direct Motor Commands

cout*{data}[{list}]* – Set open-loop command output of specified magnitude on addressed [or listed] motor[s]
home*[{list}]* – Do homing-search move on addressed [or listed] motor[s]
homez*[{list}]* – Do zero-move home on addressed [or listed] motor[s], or read absolute position sensor to establish position reference
jog+*[{list}]* – Jog addressed [or listed] motor[s] indefinitely in positive direction
jog-*[{list}]* – Jog addressed [or listed] motor[s] indefinitely in negative direction
jog/*[{list}]* – Stop jogging addressed [or listed] motor[s]; also restore to position control
jog*[{list}]={data}* – Jog addressed [or listed] motor[s] to specified position
jog*[{list}]:{data}* – Jog addressed [or listed] motor[s] specified distance from present commanded position
jog*[{list}]^{data}* – Jog addressed [or listed] motor[s] specified distance from present actual position

jog*[{list}]={data}^{data}* – Jog addressed [or listed] motor[s] to specified position; if trigger occurs, jog instead to specified distance from trigger position

jog*[{list}]:{data}^{data}* – Jog addressed [or listed] motor[s] specified distance from present commanded position; if trigger occurs, jog instead to specified distance from trigger position

jog*[{list}]^{data}^{data}* – Jog addressed [or listed] motor[s] specified distance from present actual position; if trigger occurs, jog instead to specified distance from trigger position

jogret*[{list}]* – Jog addressed [or listed] motor[s] to last programmed (pre-jog) position

jogret*[{list}]={data}* – Jog addressed [or listed] motor[s] to specified position, making that position the new "pre-jog" position

kill*[{list}]* – Kill (disable servo control on) addressed [or listed] motor[s] immediately

dkill*[{list}]* – Delayed kill (disable servo control on) of addressed [or listed] motor[s], providing time for brakes to engage fully

Direct Coordinate-System Commands

enable*[{list}]* – Enable servo control of all motors assigned to addressed [or listed] coordinate system

disable*[{list}]* – Disable (kill) servo control of all motors assigned to addressed [or listed] coordinate system

ddisable*[{list}]* – Delayed disable (kill) of servo control for all motors assigned to addressed [or listed] coordinate system, providing time for brakes to engage fully

abort*[{list}]* – Stop calculation of motion program[s] in addressed [or listed] coordinate system[s] from presently selected point, commencing immediate deceleration of all axes in the coordinate system[s].

adisable*[{list}]* – Stop calculation of motion program[s] in addressed [or listed] coordinate system[s] from presently selected point, commencing immediate deceleration of all axes in the coordinate system[s], followed by delayed disable of each motor.

hold*[{list}]* – Execute feed hold in addressed [or listed] coordinate system[s] by ramping coordinate system time base value to 0, ready to resume on **R** or **S** command

pause*[{list}]* – Stop calculation of motion program[s] in addressed [or listed] coordinate system[s] at presently selected point, ready to resume at that point

resume*[{list}]* – Restart continuous of motion program[s] in addressed [or listed] coordinate system[s] from paused point

run*[{list}]* – Begin continuous of motion program[s] in addressed [or listed] coordinate system[s] from presently selected point

start*[{list}][:{data}]* – Start running present [or specified] motion program in addressed [or listed] coordinate system[s] from beginning

step*[{list}]* – Execute single command of motion program[s] in addressed [or listed] coordinate system[s] from presently selected point

stop*[{list}]* – Stop calculation of motion program[s] in addressed [or listed] coordinate system[s] at presently selected point and make the selected point the beginning of the present program

lh – (Quick stop) Execute fastest stop in lookahead buffer that does not violate acceleration constraints for addressed [or listed] coordinate system[s]

lh< – (Reverse) Start reverse execution in lookahead buffer for addressed [or listed] coordinate system[s]

lh> – (Forward) Resume forward execution in lookahead buffer for addressed [or listed] coordinate system[s]

Program Query Commands

dread – Report desired positions of all active axes of addressed coordinate system in user units
pread – Report actual positions of all active axes of addressed coordinate system in user units
vread – Report filtered velocities of all active axes of addressed coordinate system in user units
fread – Report following errors of all active axes of addressed coordinate system in user units
tread – Report target positions of all buffered axes of addressed coordinate system in user units
dtogread – Report distance-to-go of all buffered axes of addressed coordinate system in user units

Reported Errors for Illegal Commands

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

Error ID	Error Message
0..2	*Reserved*
3	SYSTEM FILE NOT AVAILABLE
4..19	*Reserved*
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	*Reserved*
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	*Reserved*
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	*Reserved*
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	*Reserved*
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	*Reserved*
91	MACRO STATION: ILLEGAL(I,M,P,Q) DATA_TYPE
92	MACRO STATION: ILLEGAL(I,M,P,Q) DATA_NUMBER
93	*Reserved*
94	MACRO STATION: REMOTE COM TIMEOUT
95	MACRO STATION: ANOTHER STATION AT THIS ADDRESS
96	UNKNOWN # & ERROR

POWER PMAC SAVED DATA STRUCTURE ELEMENTS

This chapter documents the setup data structure elements whose present values are stored in flash memory on a **save** command and automatically restored to active memory on a power-up or reset. These elements comprise the fundamental configuration of the Power PMAC for a given application.

Some other setup data structure elements are not copied to flash memory on a **save** command; these "non-saved setup elements" are documented in a separate chapter.

Still other data structure elements provide status information to the user, and are not meant to be written to from the user's application. The values of these are not copied to flash memory on a **save** command; these "status elements" are documented in a separate chapter.

Acc5E[*i*]. Saved Data Structure Elements

The **Acc5E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate2[*i*]** data structure. The data structure elements for the ACC-5E MACRO interface board are listed under the **Gate2[*i*]** data structure, below.

Acc5E3[*i*]. Saved Data Structure Elements

The **Acc5E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-5E3 UMAC MACRO interface board are listed under the **Gate3[*i*]** data structure, below.

Acc5EP3[*i*]. Saved Data Structure Elements

The **Acc5EP3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-5EP3 Etherlite MACRO interface board are listed under the **Gate3[*i*]** data structure, below.

Acc11C[*i*]. Saved Data Structure Elements

The **Acc11C[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-11C digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc11E[*i*]. Saved Data Structure Elements

The **Acc11E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-11E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Note that the ACC-11E cannot be auto-identified by the Power UMAC CPU, so to use this data structure, the user must manually set **GateIo[*i*].PartNum** to 603307, **GateIo[*i*].PartType** to 8, issue a **save** command, and reset the controller, before being able to use this structure. This structure name is new in V2.0 firmware, released 1st quarter 2015.

Acc14E[*i*]. Saved Data Structure Elements

The **Acc14E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-14E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc24C2[*i*]. Saved Data Structure Elements

The **Acc24C2[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24C2 digital servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24C2A[*i*]. Saved Data Structure Elements

The **Acc24C2A[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24C2A analog servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2[*i*]. Saved Data Structure Elements

The **Acc24E2[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2 digital servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2A[*i*]. Saved Data Structure Elements

The **Acc24E2A[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2A analog servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2S[*i*]. Saved Data Structure Elements

The **Acc24E2S[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2S stepper interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E3[*i*]. Saved Data Structure Elements

The **Acc24E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-24E3 servo interface board are listed under the **Gate3[*i*]** data structure, below.

Acc51C[*i*]. Saved Data Structure Elements

The **Acc51C[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-51C sine-encoder interface board are listed under the **Gate1[*i*]** data structure, below.

Acc51E[*i*]. Saved Data Structure Elements

The **Acc51E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-51E sine-encoder interface board are listed under the **Gate1[*i*]** data structure, below.

Acc58E[*i*]. Saved Data Structure Elements

The **Acc58E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-58E resolver-to-digital converter board are listed under the **Gate1[*i*]** data structure, below.

Acc59E3[*i*]. Saved Data Structure Elements

The **Acc59E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-59E3 A/D and D/A-converter board are listed under the **Gate3[*i*]** data structure, below.

Acc65E[*i*]. Saved Data Structure Elements

The **Acc65E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-65E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc66E[*i*]. Saved Data Structure Elements

The **Acc66E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-66E digital input board are listed under the **GateIo[*i*]** data structure, below.

Acc67E[*i*]. Saved Data Structure Elements

The **Acc67E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-67E digital output board are listed under the **GateIo[*i*]** data structure, below.

Acc68E[*i*]. Saved Data Structure Elements

The **Acc68E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-68E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc84B[*i*]. Saved Data Structure Elements

The **Acc84B[*i*]** data structure for the serial-encoder add-in board for the Power Brick product line is equivalent to the **Acc84E[*i*]** data structure that is used for the rack-mounted Power UMAC serial-encoder interface board that is documented below. It is *not* an alias for the **Acc84E[*i*]** data structure name, as it can only be used for the Brick products.

Acc84C[*i*]. Saved Data Structure Elements

The **Acc84C[*i*]** data structure for the serial-encoder board for the Compact UMAC product line is equivalent to the **Acc84E[*i*]** data structure that is used for the rack-mounted Power UMAC serial-encoder interface board that is documented below. It is *not* an alias for the **Acc84E[*i*]** data structure name, as it can only be used for the Compact UMAC products.

Acc84E[*i*]. Saved Data Structure Elements

The ACC-84E is a 4-channel serial-encoder interface board that can support a variety of serial-encoder protocols. Its registers can be accessed through the **Acc84E[*i*]** data structure. Index values ***i*** can range from 0 to 15, as determined by the hardware address DIP switch setting on the card. This section documents the saved setup elements for the ACC-84E. The same elements are present in the **Acc84B[*i*]**, **Acc84C[*i*]**, and **Acc84S[*i*]** data structures.

Acc84E[*i*]. Multi-Channel Setup Elements

Some aspects of the serial-encoder interface are common to all channels on the board. The saved setup elements in this section affect all channels.

Acc84E[*i*].SerialEncCtrl

Description: Accessory control word for serial encoder configuration

Range: $0 .. $FFFFFF

Units: Bit field

Default: Protocol specific

Acc84E[*i*].SerialEncCtrl is the full-word element that comprises the multi-channel setup for serial encoder interfaces for the ACC-84E. Further individual channel setup is implemented in **Acc84E[*i*].Chan[*j*].SerialEncCmd**.

Acc84E[*i*].SerialEncCtrl is comprised of the following components (which cannot be accessed as independent elements):

Component	Script Bits	Hex Digit #	C Bits	Functionality
SerialClockMDiv	23 – 16	1 – 2	31 – 24	Serial clock linear division factor
SerialClockNDiv	15 – 12	3	23 – 20	Serial clock exponent division factor
(Reserved)	11 – 10	4	19 – 18	(*Reserved for future use*)
SerialTrigClockSel	09	4	17	Serial trigger source select
SerialTrigEdgeSel	08	4	16	Serial trigger source edge select
SerialTrigDelay	07 – 04	5	15 – 12	Serial trigger delay from source edge
SerialProtocol	03 – 00	6	11 – 08	Serial encoder protocol select (*read-only*)

The component *SerialClockMDiv* 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 component *SerialClockNDiv*, described below.

The equation for this intermediate clock frequency f_{int} is:

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

where *M* is short for *SerialClockMDiv*. 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 component *SerialClockNDiv* controls how the final serial-encoder clock frequency is generated from the intermediate clock frequency set by *SerialClockMDiv*. The equation for this final frequency f_{ser} is:

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

where *N* is short for *SerialClockNDiv*. 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, the resulting frequency is the frequency of the clock signal that is output from the ACC-84E's IC to the encoder. For "self-clocking" protocols without an explicit clock signal, this frequency is the input sampling frequency, and will be 20 to 25 times higher than the input bit rate f_{bit}. Refer to the instructions for the particular protocol for details.

The component *SerialTrigClockSel* controls which Power PMAC clock signal causes the encoder to be triggered. If this single-bit component is set to 0, the encoder will be triggered on the phase clock; if it is set to 1, the encoder will be triggered on the servo clock. If the encoder feedback is required for commutation rotor angle feedback, it should be triggered on the phase clock; otherwise it can be triggered on the servo clock.

The component *SerialTrigEdgeSel* controls which edge of the clock signal (phase or servo) selected by *SerialTrigClockSel* initiates the triggering process. If this single-bit component is set to 0, the triggering process starts on the rising edge; if it is set to 1, the triggering process starts on the falling edge.

Power PMAC software expects to have the resulting encoder data available to it immediately after the falling edge of the relevant phase or servo clock signal, which interrupts the processor to initiate the activity that reads this data. Since minimum delay from trigger to use is desirable, it is better to start the triggering on rising clock edge if the data can be fully transferred before the falling edge. If this is not possible, the falling edge should be used to start the triggering process.

The component *SerialTrigDelay* specifies the delay from the specified clock edge to the actual start of the output signal that will trigger the encoder response, in units of the serial encoder clock. A non-zero value can be used to minimize the delay between triggering the encoder and its resulting use by the Power PMAC.

The component *SerialProtocol* controls which serial-encoder protocol is selected for all channels of the IC. This 4-bit component can take a value from 0 to 15. This component is read-only, as it reflects the protocol interface that was installed in the board at the factory.

The following table shows the protocol selected for each value of this component (more protocols may be added):

Value	Protocol	Value	Protocol	Value	Protocol	Value	Protocol
0	(*Reserved*)	4	(*Reserved*)	8	Panasonic	12	(*Reserved*)
1	(*Reserved*)	5	(*Reserved*)	9	Mitutoyo	13	(*Reserved*)
2	SSI	6	Sigma II/III	10	Kawasaki	14	(*Reserved*)
3	EnDat	7	Tamagawa	11	BiSS-B/C	15	(*Reserved*)

When used in the Script environment, **Acc84E[*i*].SerialEncCtrl** is a 24-bit element. When used in the C environment, it is a 32-bit element, with real data in the high 24 bits, so its value in the C environment is 256 times greater than its value in the Script environment.

BiSS B/C (Unidirectional) Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for a BiSS B or BiSS C (unidirectional) encoder.

SerialClockMDiv: = (100 / f_{bit}) - 1 // Serial clock frequency = bit transmission frequency
SerialClockNDiv: = 0 // No further division unless f < 400 kHz
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $B // Shows BiSS protocol is programmed into IC

For example, for a 1.0 MHz bit transmission rate, ***SerialClockMDiv*** = (100 / 1.0) - 1 = 99 ($63) and **Acc84E[*i*].SerialEncCtrl** is set to $63000B for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	6				3				0				0				0				B				-	-
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	0	1	1	0	0	0	1	1	0	0	0	0	-	-	0	0	0	0	0	0	1	0	1	1	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

EnDat 2.1/2.2 Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for an EnDat encoder. The serial clock frequency is set 25 times higher than the external clock frequency, which is the bit transmission frequency f_{bit}, to permit oversampling of the input signal.

SerialClockMDiv: = (4 / f_{bit}) - 1 // Serial clock freq. = 25x bit transmission freq.
SerialClockNDiv: = 0 // No further division
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $3 // Shows EnDat protocol is programmed into IC

For example, for a 2.0 MHz bit transmission rate, ***SerialClockMDiv*** = (4 / 2) - 1 = 1 ($01) and **Acc84E[*i*].SerialEncCtrl** is set to $010003 for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	0				1				0				0				0				3				-	-
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	0	0	0	0	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	0	0	1	1	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				*--*	*--*	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

Matsushita (Nikon D) Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for a Matsushita serial encoder. The serial clock frequency is set 20 times higher than the external clock frequency, which is the bit transmission frequency f_{bit}, to permit oversampling of the input signal.

SerialClockMDiv:	= (5 / f_{bit}) - 1	// Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv:	= 0	// No further division
SerialTrigClockSel:	= 0	// Use phase clock if possible
SerialTrigEdgeSel:	= 0	// Use rising clock edge if possible
SerialTrigDelay:	= 0	// Can increase from 0 if possible to reduce latency
SerialProtocol:	= $C	// Shows Matsushita protocol is programmed into IC

For example, for the required 2.5 MHz bit transmission rate, ***SerialClockMDiv*** = (5 / 2.5) - 1 = 1 ($01) and **Acc84E[*i*].SerialEncCtrl** is set to $01000C for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	0				1				0				0				0				C				-	-
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	0	0	0	0	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	1	1	0	0	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				*--*	*--*	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

Mitsubishi Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for a Mitsubishi serial encoder. The serial clock frequency is set 20 times higher than the external clock frequency, which is the bit transmission frequency f_{bit}, to permit oversampling of the input signal.

SerialClockMDiv:	= (5 / f_{bit}) - 1	// Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv:	= 0	// No further division
SerialTrigClockSel:	= 0	// Use phase clock if possible
SerialTrigEdgeSel:	= 0	// Use rising clock edge if possible
SerialTrigDelay:	= 0	// Can increase from 0 if possible to reduce latency
SerialProtocol:	= $D	// Shows Mitsubishi protocol is programmed into IC

For example, for the required 2.5 MHz bit transmission rate, ***SerialClockMDiv*** = (5 / 2.5) - 1 = 1 ($01) and **Acc84E[*i*].SerialEncCtrl** is set to $01000D for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	0				1				0				0				0				D				-	-
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	0	0	0	0	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	1	1	0	1	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				*--*	*--*	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

Note

Mitsubishi Serial Encoder on HG-☐ type servo motors can only be queried at 55.5µsec±1.0µsec (18 kHz), 111µsec±1.0µsec (9 kHz) and 222µsec±1.0µsec (4.5 kHz). If the request cycle is other than the above cycles the data will not be latched properly.

Mitutoyo Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for a Mitutoyo serial encoder. The serial clock frequency is set 20 times higher than the external clock frequency, which is the bit transmission frequency f_{bit}, to permit oversampling of the input signal.

SerialClockMDiv: = (5 / f_{bit}) - 1 // Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv: = 0 // No further division
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $9 // Shows Mitutoyo protocol is programmed into IC

For example, for a 2.5 MHz bit transmission rate, ***SerialClockMDiv*** = (5 / 2.5) - 1 = 1 ($01) and **Acc84E[*i*].SerialEncCtrl** is set to $010009 for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	0				1				0				0				0				9				-	-
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	0	0	0	0	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	1	0	0	1	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

Panasonic Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for a Panasonic encoder. The serial clock frequency is set 20 times higher than the external clock frequency, which is the bit transmission frequency f_{bit}, to permit oversampling of the input signal.

SerialClockMDiv: = (5 / f_{bit}) - 1 // Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv: = 0 // No further division
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $8 // Shows Panasonic protocol is programmed into IC

For example, for a 2.5 MHz bit transmission rate, ***SerialClockMDiv*** = (5 / 2.5) - 1 = 1 ($01) and **Acc84E[*i*].SerialEncCtrl** is set to $010008 for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	0				1				0				0				0				8				-	-
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	0	0	0	0	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	1	0	0	0	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

SSI Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for an SSI encoder.

SerialClockMDiv: = (100 / f_{bit}) - 1 // Serial clock frequency = bit transmission frequency
SerialClockNDiv: = 0 // No further division unless f < 400 kHz
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $2 // Shows SSI protocol is programmed into IC

For example, for a 2.5 MHz bit transmission rate, ***SerialClockMDiv*** = (100 / 2.5) - 1 = 39 ($27) and **Acc84E[*i*].SerialEncCtrl** is set to $230002 for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	2				3				0				0				0				2				-	-
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	0	0	1	0	0	0	1	1	0	0	0	0	-	-	0	0	0	0	0	0	0	0	1	0	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

Tamagawa Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for a Tamagawa FA-Coder serial encoder. The serial clock frequency is set 20 times higher than the external clock frequency, which is the bit transmission frequency f_{bit}, to permit oversampling of the input signal.

SerialClockMDiv: = (5 / f_{bit}) - 1 // Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv: = 0 // No further division
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $7 // Shows Tamagawa protocol is programmed into IC

For example, for a 2.5 MHz bit transmission rate, ***SerialClockMDiv*** = (5 / 2.5) - 1 = 1 ($01) and **Acc84E[*i*].SerialEncCtrl** is set to $010007 for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	0				1				0				0				0				7				-	-
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	0	0	0	0	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	0	1	1	1	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

Yaskawa Sigma II/III/V Protocol

The following list shows typical settings of **Acc84E[*i*].SerialEncCtrl** for a Yaskawa II/III/V encoder.

SerialClockMDiv: = 0 // 100 MHz serial clock freq. = 25x bit transmission freq.
SerialClockNDiv: = 0 // No further division
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $6 // Shows Yaskawa II/III/V protocol is programmed

For example, for the standard 4.0 MHz bit transmission rate, a 100 MHz serial clock frequency is used, and **Acc84E[*i*].SerialEncCtrl** is set to $000006 for triggering on the rising edge of phase clock without delay.

Hex Digit ($)	0				0				0				0				0				6				-	-
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	0	0	0	0	0	0	0	0	0	0	0	0	-	-	0	0	0	0	0	0	0	1	1	0	-	-
Component:	*SerialClockMDiv*								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*				*SerialProtocol*					

Acc84E[*i*]. Single-Channel Setup Elements

Some aspects of the serial-encoder interface can be set up individually for each channel. The channel index ***j*** has a range from 0 to 3, corresponding to hardware channel 1 to 4 on the board. The channel index is one less than the corresponding hardware channel number.

Acc84E[*i*].Chan[*j*].SerialEncCmd

Description: Accessory channel control word for serial encoder commands

Range: 0 .. $FFFFFF

Units: Bit field

Default: (Protocol specific)

Acc84E[*i*].Chan[*j*].SerialEncCmd is the full-word element that comprises the command information for the serial encoder interface of the channel on the accessory. Note that the specific protocol, trigger timing, and clock frequency are determined by the multi-channel element **Acc84E[*i*].SerialEncCtrl**.

Acc84E[*i*].Chan[*j*].SerialEncCmd is comprised of the following components (which cannot be accessed as independent elements):

Component	Script Bits	Hex Digit #	C Bits	Functionality
SerialEncCmdWord	23 – 16	1 – 2	31 – 24	Serial encoder output command
SerialEncParity	15 – 14	3	23 – 22	Serial encoder parity type
SerialEncTrigMode	13	3	21	Serial trigger mode: continuous or one-shot
SerialEncTrigEna	12	3	20	Serial trigger enable
SerialEncGtoB	11	4	19	Serial SSI data Gray-to-binary convert control
SerialEncEna/ SerialEncDataReady	10	4	18	Serial encoder circuitry enable (write) Serial encoder received data ready (read)
SerialEncStatusBits	09 – 06	4 – 5	17 – 14	Serial encoder SPI number of status bits
SerialEncNumBits	05 – 00	5 – 6	13 – 08	Serial encoder bit length control

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.

When used in the Script environment, **Acc84E[*i*].Chan[*j*].SerialEncCmd** is a 24-bit element. When used in the C environment, it is a 32-bit element, with real data in the high 24 bits, so its value in the C environment is 256 times greater than its value in the Script environment.

BiSS-B/C (Unidirectional) Protocol

For the BiSS-B and BiSS-C (unidirectional) protocols, the *SerialEncCmdWord* component of **Acc84E[*i*].Chan[*j*].SerialEncCmd** specifies the CRC polynomial used for error detection when the position and status data are reported. This is an 8-bit mask value "M" that can define any 4-bit to 8-bit CRC polynomial. It must be set up to match the polynomial used for the particular BiSS encoder.

The mask bits M_7 to M_0 represent the coefficients for the terms x^8 to x^1, respectively, in the CRC polynomial:

$$M_7x^8 + M_6x^7 + M_5x^6 + M_4x^5 + M_3x^4 + M_2x^3 + M_1x^2 + M_0x^1 + 1$$

The coefficient for x^0 in a CRC polynomial is always 1, and so is not included in the mask. A mask with all zeros is used to indicate that no CRC bits are included with the encoder data.

For example, if the encoder uses a CRC polynomial of $x^6 + x^1 + 1$ (as with the Renishaw Resolute™ encoders), the CRC mask value M should be set to 00100001 (bits M_5 and M_0 set to 1), or $21.

For the BiSS protocol, the *SerialEncParity* component of **Acc84E[*i*].Chan[*j*].SerialEncCmd** is used to distinguish between the BiSS-B and BiSS-C protocol variants. Bit 1 of the component (bit 15 of the 24-bit element) is set to 0 for BiSS-C, and to 1 for BiSS-B. (BiSS-C provides a zero bit between the start bit and the position data; BiSS-B does not.)

Bit 0 of the component (bit 14 of the 24-bit element) is only used for BiSS-B. If it is set to 1, it permits the acceptance of a "Multi-Cycle Data" (MCD) bit from the encoder by providing an extra clock cycle output. The MCD bit is not captured or used.

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for a BiSS C encoder.

SerialEncCmdWord:	= $21	// CRC polynomial of $x^6 + x^1 + 1$
SerialEncParity:	= 0	// BiSS-C protocol variant
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray code supported for EnDat protocol
SerialEncEna:	= 1	// Enable driver circuitry
SerialEncStatusBits:	= *{enc spec}*	// Encoder-specific number of status bits returned
SerialEncNumBits:	= *{enc spec}*	// Encoder-specific number of position bits returned

For example, for a BiSS-C encoder with 36 position bits, 2 status bits, and a CRC polynomial of $x^6 + x^1 + 1$, **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $2114A4 for continuous position reporting. (It may report back as $2110A4 if the data-ready status bit is not set.)

Hex Digit ($)	2				1				1				4				A				4				-	-
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	0	0	1	0	0	0	0	1	0	0	0	1	-	1	-	0	1	0	1	0	0	1	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*				*Status*				*NumBits*					

For a BiSS-B encoder with 32 position bits, 4 status bits, an MCD bit, and a CRC polynomial of $x^4 + x^1 + 1$, **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $09D520 for continuous position reporting. (It may report back as $09D120 if the data-ready status bit is not set.)

Hex Digit ($)	0				9				D				5				2				0				-	-
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	0	0	0	0	1	0	0	1	1	1	0	1	-	1	-	1	0	0	1	0	0	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*				*Status*				*NumBits*					

EnDat2.1/2.2 Protocol

The EnDat interface in the ACC-84E supports four 6-bit command codes that are sent directly to the encoder:

- 000111 ($07) for reporting position (EnDat2.1)
- 101010 ($2A) for resetting the encoder (EnDat2.1)
- 111000 ($38) for reporting position with possible additional information (EnDat2.2)
- 101101 ($2D) for resetting the encoder (EnDat2.2)

These 6 bits fit at the low end of the 8-bit *SerialEncCmdWord* command field of **Acc84E[*i*].Chan[*j*].SerialEncCmd**.

Note

By the EnDat standard, EnDat2.2 encoders should be able to accept and process EnDat2.1 command codes as well. However, not all encoders sold as meeting the EnDat2.2 standard can do this.

For EnDat2.2 encoders, the ACC-84E (starting 1st quarter 2014) also supports controller requests for additional information from the encoder through the use of Memory Range Select (MRS) codes. To implement these, the *SerialEncCmdWord* command field contains the MRS code. (In this mode, the ACC-84E sends the 111000 command code – report position with additional information – to the encoder.)

The following MRS codes are supported in the EnDat2.2 standard:

- $40 – Send additional info 1 w/o data content (NOP)
- $42 – Send position value 2 word 1 LSB
- $43 – Send position value 2 word 2
- $44 – Send position value 2 word 3 MSB
- $47 – Acknowledge MRS code
- $49 – Send test value word 1 LSB
- $4A – Send test value word 2
- $4B – Send test value word 3 MSB
- $4C – Send temperature 1
- $4D – Send temperature 2
- $4F – Stop sending additional information 1
- $50 – Send additional info 2 w/o data contents (NOP)
- $51 – Send commutation
- $52 – Send acceleration
- $53 – Send commutation & acceleration
- $54 – Send limit position signals
- $55 – Send limit position signals & acceleration
- $56 – Currently not assigned
- $5F – Stop sending additional information 2

The response from the encoder to specific MRS code data requests from the encoder depends on the availability of that data in the encoder. The additional information provided from supported

MRS codes will be found in status elements **Acc84E[*i*].Chan[*j*].SerialEncDataC** and **SerialEncDataD**.

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for position reporting from an EnDat encoder.

SerialEncCmdWord: = *{cmd/MRS code}* // Command code
SerialEncParity: = 0 // No parity check supported for EnDat protocol
SerialEncTrigMode: = 0 // Continuous triggering (EnDat2.2)
SerialEncTrigEna: = 1 // Enable triggering
SerialEncGtoB: = 0 // No Gray code supported for EnDat protocol
SerialEncEna: = 1 // Enable driver circuitry
SerialEncStatusBits: = 0 // No status bits supported for EnDat protocol
SerialEncNumBits: = *{enc spec}* // Encoder-specific number of position bits returned

For example, for an EnDat2.2 encoder with 37 position bits, **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $381425 for continuous position reporting. (It may report back as $381025 if the data-ready status bit is not set.)

Hex Digit ($)	3				8				1				4				2				5				-	-
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	-	0	1	1	1	0	0	0	-	-	0	1	-	1	-	-	-	-	1	0	0	1	0	1	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB*		*Ena*		*Status*				*NumBits*					

This same encoder can be reset with a command word value of 45 ($2D) sent in one-shot mode with **Acc84E[*i*].Chan[*j*].SerialEncCmd** set to $2D3425.

Hex Digit ($)	2				D				3				4				2				5				-	-
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	-	0	1	0	1	1	0	1	-	-	1	1	-	1	-	-	-	-	1	0	0	1	0	1	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB*		*Ena*		*Status*				*NumBits*					

For an EnDat2.1 encoder with 24 position bits, **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $073418 for one-shot position reporting (at power-up). It will report back as $073018 until the data is received.

Hex Digit ($)	0				7				3				4				1				8				-	-
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	-	-	0	0	0	1	1	1	-	-	1	1	-	1	-	-	-	-	0	1	1	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB*		*Ena*		*Status*				*NumBits*					

For an EnDat2.2 incremental encoder with 24 position bits, **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $421418 for continuous position reporting with additional information of position 2 word 1. (It may report back as $421018 if the data-ready status bit is not set.)

Hex Digit ($)	4				2				1				4				1				8				-	-
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	-	1	0	0	0	0	1	0	-	-	0	1	-	1	-	-	-	-	0	1	1	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB*		*Ena*		*Status*				*NumBits*					

Matsushita (Nikon D) Protocol

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for a Matsushita serial encoder.

SerialEncCmdWord:	= $CA	// Command word for multi-turn position in Matsushita
SerialEncParity:	= 0	// No parity check supported for Matsushita protocol
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray code supported for Matsushita protocol
SerialEncEna:	= 1	// Enable driver circuitry
SerialEncStatusBits:	= 0	// No status bits supported for Matsushita protocol
SerialEncNumBits:	= 0	// first 4 bits represent the Encoder ID

Acc84E[*i*].Chan[*j*].SerialEncCmd would be set to $CA1400 for continuous position reporting. (It may report back as $CA1000 if the ready status bit is not set.)

Hex Digit ($)	C				A				1				4				0				0				-	-
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	1	1	0	0	1	0	1	0	0	0	0	1	-	1	-	-	-	-	-	-	0	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*		*Reserved*						*EncoderID*					

Matsushita protocol supports two modes of communication: independent and continuous. In Delta Tau's implementation of this protocol, only the independent communication is supported where the ID of the request packet should match the ID of the encoder.

If the *SerialEncCmdWord* component is set to $9A for single-turn position reporting (first 16-bits of single turn data), **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $9A1400. (It may report back as $9A1000 if the data-ready status bit is not set.)

If the *SerialEncCmdWord* component is set to $A2 for multi-turn position plus MSB of single turn data reporting (15 bits of multiple turn data + bit 17 of single turn data), **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $A21400. (It may report back as $A21000 if the data-ready status bit is not set.)

If the *SerialEncCmdWord* component is set to $AA for Alarm code data reporting, **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $AA1400. (It may report back as $AA1000 if the data-ready status bit is not set.)

If the *SerialEncCmdWord* component is set to $CA for single-turn + multi-turn + alarm reporting (17 bits of single-turn data + 15 bits of multi-turn data + 8 bits of alarm data), **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $CA1400. (It may report back as $CA1000 if the data-ready status bit is not set.)

If the *SerialEncCmdWord* component is set to $E2 (Reset I) and triggered for 10 consecutive cycles with 7 microseconds interval or more battery alarm, system down and over speed flags are cleared. This should be done in "one-shot" mode, making the element equal to $E23400 and triggered for 10 consecutive cycles with 7 microseconds interval or more. The register should report as $E22000 after completion of a single trigger.

If the *SerialEncCmdWord* component is set to $EA (Reset II) and triggered for 10 consecutive cycles with 7 microseconds interval or more, multi-turn data and counter overflow are reset. This

reset mode should only be called when the motor speed is less than 300 RPM. This should be done in "one-shot" mode, making the element equal to $EA3400 and triggered for 10 consecutive cycles with 7 microseconds interval or more. The register should report as $EA2000 after completion of a single trigger.

Note

For both $E2 and $EA reset modes:

- To check whether the reset is made correctly, send a request signal $CA to the encoder and check the multi-turn data and ALC on the output signal data field.
- Single-turn absolute data is prohibited from resetting.

If the *SerialEncCmdWord* component is set to $B2 and triggered for 10 consecutive cycles with 7 microseconds interval or more single revolution data will be reset to 0° and shaft position will be written to EEPROM. This should be done in "one-shot" mode, making the element equal to $B23400 and triggered for 10 consecutive cycles with 7 microseconds interval or more. Notice that this reset command (all 10) should only be sent when the encoder is at rest with no movement. Once reset, the single turn zero location is maintained regardless of connection of external battery after main power source is turned off. The register should report as $B22000 after completion of a single trigger.

If the *SerialEncCmdWord* component is set to $BA and triggered for 10 consecutive cycles with 7 microseconds interval or more single revolution data will be reset to its initial data and EEPROM data will be reset. This should be done in "one-shot" mode, making the element equal to $BA3400 and triggered for 10 consecutive cycles with 7 microseconds interval or more. Notice that this reset command (all 10) should only be sent when the encoder is at rest with no movement. Once reset, the single turn zero location is maintained regardless of connection of external battery after main power source is turned off. The register should report as $BA2000 after completion of a single trigger.

If the *SerialEncCmdWord* component is set to $F2/$FA and triggered for 10 consecutive cycles with 7 microseconds interval or more, The Encoder ID present in lower 4 bits of **Acc84E[*i*].Chan[*j*].SerialEncCmd** is written to EEPROM on the encoder. This should be done in "one-shot" mode, making the element equal to $F23400/$FA3400 and triggered for 10 consecutive cycles with 7 microseconds interval or more. The register should report as $F22000/$FA2000 after completion of a single trigger.

If the *SerialEncCmdWord* component is set to $D2/$DA and triggered for 10 consecutive cycles with 7 microseconds interval or more, The Encoder ID stored in EEPROM of the encoder is fored into "S8-shaft" (111). This should be done in "one-shot" mode, making the element equal to $D23400/$DA3400 and triggered for 10 consecutive cycles with 7 microseconds interval or more. The register should report as $D22000/$DA2000 after completion of a single trigger.

Note

Only the encoders that are written with an encoder ID "S8-shaft" permit to change the setting of the encoder ID.

If an encoder has an ID, other than S8 (111), it is necessary to assign it the ID S8 using $D2/DA mode before assigning it its final ID.

The difference between $F2 and $FA is similar to the difference between $D2 and $DA command modes. The command modes $F2 and D2 despite different functionality, cause the overflow flag to be reflected in the **ea0** status bit. The command modes $FA and DA, prevent the reflection of overflow flag in the **ea0** status bit.

Mitsubishi Protocol

The Mitsubishi encoder has 8 request codes defined for the Request Field transmitted from the controller to the encoder. To transmit a specific ID code to the encoder, program the *SerialEncCmdWord* register with the appropriate Command Code listed below:

- $02 – for reporting single-turn data (single-turn data in lower 18 bits of 24-bit word. 18-bit single-turn data in bits 0 to 17 and bits [18:23] report 0)
- $8A – for reporting multi-turn data
- $92 – for reporting Encoder-ID
- $A2 – for reporting single-turn and multi-turn data (single-turn data in lower 20 bits of 24-bit word. 18-bit single-turn data in bits 2 to 19 and bits [0:1] and [18:23] report 0)
- $2A – for reporting single-turn and multi-turn data (single-turn data in lower 18 bits of 24-bit word. 18-bit single-turn data in bits 0 to 17 and bits [18:23] report 0)
- $32 – for reporting single-turn and multi-turn data (single-turn data in upper 20 bits of 24-bit word. 18-bit single-turn data in bits 6 to 23 and bits [0:5] report 0)
- $BA – for clearing alarms and reporting single-turn data(single-turn data in lower 18 bits of 24-bit word. 18-bit single-turn data in bits 0 to 17 and bits [18:23] report 0)
- $7A – for reporting encoder/motor ID

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for a Mitsubishi HG type servo motor's serial encoder.

SerialEncCmdWord:	= $32	// Command word for position reporting in Mitsubishi
SerialEncParity:	= 0	// No parity check supported for Mitsubishi protocol
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray code supported for Mitsubishi protocol
SerialEncEna:	= 1	// Enable driver circuitry
SerialEncStatusBits:	= 0	// No status bits supported for Mitsubishi protocol
SerialEncNumBits:	= 0	// Fixed number of position bits returned

Acc84E[*i*].Chan[*j*].SerialEncCmd would be set to $321400 for continuous position reporting. (It may report back as $321000 if the ready status bit is not set.)

Hex Digit ($)	3				2				1				4				0				0				-	-
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	0	0	1	1	0	0	1	0	0	0	0	1	-	1	-	-	-	-	0	0	0	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*		*Status*						*NumBits*					

If the *SerialEncCmdWord* component is set to $BA and triggered for 10 consecutive cycles with 55.5 microseconds interval or more, the ABS lost alarm is cleared. This should be done in "one-shot" mode, making the element equal to $BA3400 and triggered for 10 consecutive cycles with 55.5 microseconds interval or more.

If the *SerialEncCmdWord* component is set to $7A and triggered for 10 consecutive cycles with 222±1.0 µsec interval, the encoder/motor ID is sent back from encoder to controller. In order to exit from this mode, any other mode command ($02, $8A, $92, $A2, $2A, $32) should be triggered for 10 consecutive cycles with 222±1.0 µsec interval after which normal cyclic position reporting could be resumed. This should be done in "one-shot" mode, making the element equal to $7A3400 and triggered for 10 consecutive cycles with 222 microseconds interval. Once encoder ID is retrieved, any other mode command (for example $323400) should be triggered for 10 consecutive cycles with 222±1.0 µsec interval to exit the encoder ID reporting mode.

When the reset operation is done, the component should report as $BA2000, $7A2000 respectively.

Mitutoyo Protocol

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for a Mitutoyo serial encoder.

SerialEncCmdWord:	= $01	// Command word for position reporting in Mitutoyo
SerialEncParity:	= 0	// No parity check supported for Mitutoyo protocol
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray code supported for Mitutoyo protocol
SerialEncEna:	= 1	// Enable driver circuitry
SerialEncStatusBits:	= 0	// No status bits supported for Mitutoyo protocol
SerialEncNumBits:	= 0	// Fixed number of position bits returned

Acc84E[*i*].Chan[*j*].SerialEncCmd would be set to $011400 for continuous position reporting. (It may report back as $011000 if the ready status bit is not set.)

Hex Digit ($)	0				1				1				4				0				0				-	-
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	0	0	0	0	0	0	0	1	0	0	0	1	-	1	-	-	-	-	0	0	0	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*		*Status*						*NumBits*					

If the *SerialEncCmdWord* component is set to $89 and sent 8 times, the multi-turn position value in the encoder is reset to 0. This should be done in "one-shot" mode repeated 8 times, making the element equal to $893400. When the reset operation is done, the component should report as $892000. If this component is set to $9D, the encoder will report its ID value. This should be done in "one-shot" mode, and the IC will hold this value in status element **Acc84E[*i*].Chan[*j*].SerialEncDataC**.

Panasonic Protocol

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for a Panasonic serial encoder.

SerialEncCmdWord:	= $2A	// Command word for multi-turn position in Panasonic
SerialEncParity:	= 0	// No parity check supported for Panasonic protocol
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray code supported for Panasonic protocol
SerialEncEna:	= 1	// Enable driver circuitry
SerialEncStatusBits:	= 0	// No status bits supported for Panasonic protocol
SerialEncNumBits:	= 0	// Fixed number of position bits returned

Acc84E[*i*].Chan[*j*].SerialEncCmd would be set to $2A1400 for continuous position reporting. (It may report back as $2A1000 if the ready status bit is not set.)

Hex Digit ($)	2				A				1				4				0				0				-	-
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	0	0	1	0	1	0	1	0	0	0	0	1	-	1	-	-	-	-	0	0	0	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*		*Status*						*NumBits*					

If the *SerialEncCmdWord* component is set to $52 for single-turn position reporting with alarm code, **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $521400. (It may report back as $521000 if the data-ready status bit is not set.) If the *SerialEncCmdWord* component is set to $52, the encoder ID value is also reported.

If the *SerialEncCmdWord* component is set to $4A, $7A, $DA, or $F2, the multi-turn position value in the encoder is reset to 0. This should be done in "one-shot" mode, making the element equal to $4A3400, $7A3400, $DA3400, or $F23400, respectively. When the reset operation is done, the component should report as $4A2000, $7A2000, $DA2000, or $F22000, respectively.

SSI Protocol

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for an SSI encoder.

SerialEncCmdWord:	= 0	// No command word supported for SSI protocol
SerialEncParity:	= ??	// Encoder-specific parity check
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= ??	// Encoder-specific data format
SerialEncEna:	= 1	// Enable driver circuitry
SerialEncStatusBits:	= 0	// No status bits supported for SSI protocol
SerialEncNumBits:	= ??	// Encoder-specific number of position bits returned

For example, for an SSI encoder with 25 position bits in Gray-code format with odd parity, **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $005C19. (It may report back as $005819 if the data-ready status bit is not set.)

Hex Digit ($)	0				0				5				C				1				9				-	-
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	-	-	-	-	-	-	-	-	0	1	0	1	1	1	-	-	-	-	0	1	1	0	0	1	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*		*Status*						*NumBits*					

Tamagawa FA-Coder Protocol

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for a Tamagawa FA-Coder serial encoder.

```
SerialEncCmdWord:    = $1A    // Command word for position reporting in Tamagawa
SerialEncParity:     = 0      // No parity check supported for Tamagawa protocol
SerialEncTrigMode:   = 0      // Continuous triggering
SerialEncTrigEna:    = 1      // Enable triggering
SerialEncGtoB:       = 0      // No Gray code supported for Tamagawa protocol
SerialEncEna:        = 1      // Enable driver circuitry
SerialEncStatusBits: = 0      // No status bits supported for Tamagawa protocol
SerialEncNumBits:    = 0      // Fixed number of position bits returned
```

Acc84E[*i*].Chan[*j*].SerialEncCmd would be set to $1A1400 for continuous position reporting. (It may report back as $1A1000 if the ready status bit is not set.)

Hex Digit ($)	1				A				1				4				0				0				-	-
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	0	0	0	1	1	0	1	0	0	0	0	1	-	1	-	-	-	-	0	0	0	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*		*Status*						*NumBits*					

If the *SerialEncCmdWord* component is set to $BA, $C2, or $62, the multi-turn position value in the encoder is reset to 0. This should be done in "one-shot" mode, making the element equal to $BA3400, $C23400, or $623400, respectively. When the reset operation is done, the component should report as $BA2000, $C22000, or $622000, respectively.

Yaskawa Sigma II/III/V Protocol

The Yaskawa Sigma II/III/V interface supports position reporting and fault-reset modes. The command code for position reporting is $00; the command code for fault reset is $04.

The following list shows typical settings of **Acc84E[*i*].Chan[*j*].SerialEncCmd** for position reporting from a Yaskawa Sigma II/III/V encoder.

```
SerialEncCmdWord:    = 0      // No command word for position reporting in Yaskawa
SerialEncParity:     = 0      // No parity check supported for Yaskawa protocol
SerialEncTrigMode:   = 0      // Continuous triggering
SerialEncTrigEna:    = 1      // Enable triggering
SerialEncGtoB:       = 0      // No Gray code supported for Yaskawa protocol
SerialEncEna:        = 1      // Enable driver circuitry
SerialEncStatusBits: = 0      // No status bits supported for Yaskawa protocol
SerialEncNumBits:    = 0      // Fixed number of position bits returned
```

Acc84E[*i*].Chan[*j*].SerialEncCmd would be set to $001400 for continuous position reporting. (It may report back as $001000 if the ready status bit is not set.)

Hex Digit ($)	0				0				1				4				0				0				-	-
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	-	-	0	0	0	0	0	0	-	-	0	1	-	1	0	0	0	0	0	0	0	0	0	0	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*				*Status*				*NumBits*					

To reset the encoder, the components must be set up as follows:

SerialEncCmdWord:	= $04	// Fault-reset command code
SerialEncParity:	= 0	// No parity check supported for Yaskawa protocol
SerialEncTrigMode:	= 1	// Single-shot triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray code supported for Yaskawa protocol
SerialEncEna:	= 1	// Enable driver circuitry
SerialEncStatusBits:	= 4	// Special reset command for Yaskawa protocol
SerialEncNumBits:	= $01	// "Encoder address" for reset

This means that **Acc84E[*i*].Chan[*j*].SerialEncCmd** would be set to $043501 to start a fault reset. (It may report back as $043101 if the ready status bit is not set.)

Hex Digit ($)	0				4				3				5				0				1				-	-
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	-	-	0	0	0	1	0	0	-	-	1	1	-	1	0	1	0	0	0	0	0	0	0	1	-	-
Component:	*SerialEncCmdWord*								*Parity*		*TM*	*TE*	*GB Ena*				*Status*				*NumBits*					

After writing this value to the element to start the fault reset of the encoder, the user should:

1. Wait 100 milliseconds.
2. Wait for the trigger-enable component (bit 12) of this element to clear.
3. Wait for the busy signal (bit 8) of **Acc84E[*i*].Chan[*j*].SerialEncDataB** to clear.
4. Clear the command code of this element to 0 by writing $003501 to the element.
5. Repeat steps 1 to 3.
6. Resume continuous position requests by writing $001400 to the element.

Acc84S[*i*]. Saved Data Structure Elements

The **Acc84S[*i*]** data structure for the serial-encoder board for the Power Clipper product line is equivalent to the **Acc84E[*i*]** data structure that is used for the rack-mounted Power UMAC serial-encoder interface board that is documented below. It is *not* an alias for the **Acc84E[*i*]** data structure name, as it can only be used for the Power Clipper products.

Note that the ACC-84S cannot be auto-identified by the Power Clipper CPU, so to use this data structure, the user must manually set **GateIo[*i*].PartNum** to 603936, **GateIo[*i*].PartType** to 8, issue a **`save`** command, and reset the controller, before being able to use this structure. This structure name is new in V2.0 firmware, released 1st quarter 2015.

AdcDemux. Saved Data Structure Elements

The **AdcDemux.** data structure provides a convenient means for "de-multiplexing" the multiplexed analog-to-digital converters on boards such as the ACC-36E and ACC-59E.

AdcDemux.Address[*i*]

Description: ADC De-multiplexing source addresses

Range: Valid I/O address offsets

Units: Byte offset from I/O base address

Default: 0

AdcDemux.Address[*i*] elements specify the source addresses of the A/D converter pairs to be de-multiplexed. The index value for these elements can take a value from 0 to 15, so 16 possible sources can be specified.

Note that an ACC-36E has 8 pairs of A/D converters accessed at a single address, and an ACC-59E has 8 single A/D converters accessed at a single address, so in many cases, 8 of these elements will specify the same address.

The addresses are specified in terms of an offset from the I/O base address. (This base address does not need to be known.) The value is dependent on the DIP-switch settings for the board to be accessed. The following table shows the proper value for each of the possible DIP-switch settings on ACC-36E and ACC-59E boards:

SW1-1	SW1-2	SW1-3	SW1-4	AdcDemux. Address[*i*]
ON	ON	ON	ON	$A00000
OFF	ON	ON	ON	$B00000
ON	OFF	ON	ON	$C00000
OFF	OFF	ON	ON	$D00000
ON	ON	OFF	ON	$A08000
OFF	ON	OFF	ON	$B08000
ON	OFF	OFF	ON	$C08000
OFF	OFF	OFF	ON	$D08000
ON	ON	ON	OFF	$A10000
OFF	ON	ON	OFF	$B10000
ON	OFF	ON	OFF	$C10000
OFF	OFF	ON	OFF	$D10000
ON	ON	OFF	OFF	$A18000
OFF	ON	OFF	OFF	$B18000
ON	OFF	OFF	OFF	$C18000
OFF	OFF	OFF	OFF	$D18000

Note: SW1-5 and SW1-6 must be ON to enable this addressing.

Only those addresses permitted by **AdcDemux.Enable** will actually be used. For example, if **AdcDemux.Enable** is set to 8, **AdcDemux.Address[0]** through **AdcDemux.Address[7]** will be used. For each **AdcDemux.Address[*i*]** enabled, the **AdcDemux.ConvertCode[*i*]** of the same

index value must be properly specified. The results of the demultiplexing can be found in **AdcDemux.ResultLow[*i*]** and **AdcDemux.ResultHigh[*i*]** of the same index value.

This function was performed by I5061 – I5076 in Turbo PMAC, but these cannot be used as "legacy I-variables" for these elements in Power PMAC.

AdcDemux.ConvertCode[*i*]

Description: ADC De-multiplexing conversion codes

Range: $000000 .. $F00F00

Units: ADC conversion codes

Default: 0

AdcDemux.ConvertCode[*i*] elements contain the convert codes written to the multiplexed A/D converters on ACC-36E or ACC-59E that are read in the A/D ring table, as enabled by **AdcDemux.Enable**. The convert codes control which of the multiplexed ADCs at the address is to be read, and the range of the analog input for that ADC.

AdcDemux.ConvertCode[*i*] elements take a value of $*m*00*n*00. Each one specifies the value to be written to the multiplexed ADCs at the address specified by **AdcDemux.Address[*i*]** of the same index values. The conversion code specifies two things. First, it selects which pair of ADCs is to be used for this slot in the de-multiplexing ring. Second, it selects whether the ADCs sample a unipolar voltage input and return an unsigned number, or sample a bipolar input and return a signed number. Both ADCs of a pair will typically sample and return in the same format.

The ACC-36E has two 12-bit ADCs in each register – a "high" ADC and a "low" ADC. The high ADC is controlled by the first three hex digits of **ConvertCode[*i*]**, and the low ADC by the last three digits. The ACC-59E has only one 12-bit ADC in each register – the "low" ADC, which is controlled by the last three hex digits of **ConvertCode[*i*]**. The setting of the first three hex digits does not matter for the ACC-59E.

The following table shows which ADCs are sampled at the specified address, and in which format they are sampled, for each of the standard values of **AdcDemux.ConvertCode[*i*]**, for which both ADCs are converted in the same format.

ConvertCode	ResultLow ADC	ResultHigh ADC	Format
$000000	ADC1	ADC9	Unipolar, unsigned
$100100	ADC2	ADC10	Unipolar, unsigned
$200200	ADC3	ADC11	Unipolar, unsigned
$300300	ADC4	ADC12	Unipolar, unsigned
$400400	ADC5	ADC13	Unipolar, unsigned
$500500	ADC6	ADC14	Unipolar, unsigned
$600600	ADC7	ADC15	Unipolar, unsigned
$700700	ADC8	ADC16	Unipolar, unsigned
$800800	ADC1	ADC9	Bipolar, signed
$900900	ADC2	ADC10	Bipolar, signed
$A00A00	ADC3	ADC11	Bipolar, signed

ConvertCode	ResultLow ADC	ResultHigh ADC	Format
$B00B00	ADC4	ADC12	Bipolar, signed
$C0C00	ADC5	ADC13	Bipolar, signed
$D00D00	ADC6	ADC14	Bipolar, signed
$E00E00	ADC7	ADC15	Bipolar, signed
$F00F00	ADC8	ADC16	Bipolar, signed

Only those conversion codes permitted by **AdcDemux.Enable** will actually be used. For example, if **AdcDemux.Enable** is set to 8, **AdcDemux.ConvertCode[0]** through **AdcDemux.ConvertCode[7]** will be used. For each **AdcDemux.ConvertCode[*i*]** enabled, the **AdcDemux.Address[*i*]** of the same index value must be properly specified. The results of the demultiplexing can be found in **AdcDemux.ResultLow[*i*]** and **AdcDemux.ResultHigh[*i*]** of the same index value. (For the ACC-59E, only the **ResultLow[*i*]** values are used.)

This function was performed by I5081 – I5096 in Turbo PMAC, but these cannot be used as "legacy I-variables" for these elements in Power PMAC.

AdcDemux.Enable

Description: ADC De-multiplexing enable/ring size

Range: 0 – 16

Units: Number of ADC pairs

Default: 0

AdcDemux.Enable controls the number of pairs of multiplexed A/D converters, typically from ACC-36E or ACC-59E boards, that are processed and "de-multiplexed" into individual registers. If **AdcDemux.Enable** is set to 0, none of these A/D converters is processed automatically.

If **AdcDemux.Enable** is set to a value greater than 0, it specifies the number of pairs of ADCs in the automatic processing ring. Each phase clock cycle, one pair is processed, and the values copied into data structure elements **AdcDemux.ResultLow[*i*]** and **AdcDemux.ResultHigh[*i*]** in RAM.

For each pair enabled, one of the A/D ring slot pointer setup elements **AdcDemux.Address[*i*]**and one of the A/D ring convert code setup elements **AdcDemux.ConvertCode[*i*]** must be set properly. If **AdcDemux.Enable** is set to 1, then **AdcDemux.Address[0]** and **AdcDemux.ConvertCode[0]** must be set properly; if **AdcDemux.Enable** is set to 2, then **AdcDemux.Address[1]** and **AdcDemux.ConvertCode[1]** must also be set properly.

This function was performed by I5060 in Turbo PMAC, but I5060 cannot be used as a "legacy I-variable" for this element in Power PMAC.

BrickAC. Saved Data Structure Elements

The Power Brick AC is one class of the Power Brick family of intelligent amplifiers. It can support multiple types of brush, brushless and AC-induction motors, operating from an AC line input. Its registers can be accessed through **BrickAC.** data structure.

Note

The **BrickAC.** data structure elements documented in this section are software elements that are distinct from the **PowerBrick[*i*].** hardware data structure elements that form the control and status registers for the ASIC(s) in the Power Brick AC. (The **PowerBrick[*i*].** data structure is an "alias" for the **Gate3[*i*].** hardware data structure.)

BrickAC. Multi-Channel Saved Setup Elements

Some aspects of the Power Brick AC amplifier are common to all channels on the board. The saved setup elements in this section affect all channels.

BrickAC.MonitorPeriod

Description: Time interval for updating status registers

Range: 0 .. 2,147,483,647 (2^{31}-1)

Units: Milliseconds

Default: 0 (50 msec)

BrickAC.MonitorPeriod tells Power PMAC software how much time there is between consecutive requests for the value of all Brick AC status registers. It is expressed in milliseconds as an integer value.

If **BrickAC.MonitorPeriod** is set to the default value of 0 or any value up to 50, all Brick AC status elements are updated every 50 milliseconds. Setting the value higher will reduce the update frequency and reduces the background time which monitor process takes from the Power PMAC CPU.

Note

The value of **BrickAC.MonitorPeriod** does not affect how often the amplifier stage checks the status conditions internally. It only controls how frequently the Power PMAC CPU requests this information.

While the value of **BrickAC.MonitorPeriod** is saved, the element that starts the monitoring process itself, **BrickAC.Monitor**, is not a saved setup element. It must explicitly be set to 1 by the user application in order to start the monitoring process. Also, when either the configuration

process or the fault-clearing reset process is started with **BrickAC.Config** or **BrickAC.Reset**, respectively, the monitoring process is stopped, and it is not automatically restarted. The user application must explicitly restart the monitoring process.

Note

The monitored data in the Power Brick AC is provided to the controller on the lower 10 bits of the **Gate3[*i*].Chan[*j*].AdcAmp[*k*]** registers and it is essential that **Gate3[*i*].Chan[*j*].PackInData** and **Gate3[*i*].Chan[*j*].PackOutData** are set to 0, disabling "packed" register access and allowing all ADC register bits to be read by the CPU.

BrickAC.SinglePhaseIn

Description: Expected line input type

Range: 0 .. 1

Units: none

Default: 0 (three-phase)

BrickAC.SinglePhaseIn tells the amplifier whether to expect a single-phase line input or a 3-phase line input. If set to the default value of 0, the Power Brick AC expects a 3-phase AC line input and the **BrickAC.PhaseInMissing** monitor is active, so the amplifier will issue a warning on the loss of any of the three phases, setting status bit **BrickAC.PhaseInMissing** to 1.

If **BrickAC.SinglePhaseIn** is set to 1, the 3-phase line input monitor process is disabled and the Power Brick AC runs in single-phase input mode with no phase-loss detection. In this case, a single-phase AC line input, or a DC line input can be connected across any two of the three line inputs on the amplifier. (Loss of this input would result in a power fault condtion.)

The **BrickAC.SinglePhaseIn** value is sent to the active amplifier-control circuit upon setting one of the non-saved setup elements **BrickAC.Reset** or **BrickAC.Config** equal to 1 in a Script command. It does not take effect until then.

BrickAC.UnderVoltageDisplay

Description: Undervoltage condition display control

Range: 0 .. 1

Units: Boolean

Default: 0

BrickAC.UnderVoltageDisplay tells Power Brick AC whether to display the undervoltage status on its 7-segment display or not. If it is set to 0, no undervoltage condition status is displayed. If it is set to 1, an error code "U" is displayed in the event of a bus undervoltage condition.

This setting is purely for display-control purposes and does not affect what action the Power Brick AC takes when an undervoltage condition is detected. The separate **BrickAC.UnderVoltageWarnOnly** element controls whether an undervoltage condition should be treated as a warning or as a fault. An undervoltage condition occurs when the internal DC bus voltage drops below 100 VDC, which corresponds to about 70 VAC(rms).

The **BrickAC.UnderVoltageDisplay** value is sent to the active amplifier-control circuit upon setting one of the non-saved setup elements **BrickAC.Reset** or **BrickAC.Config** equal to 1 in a Script command. It does not take effect until then.

BrickAC.UnderVoltageWarnOnly

Description: Undervoltage condition warning/fault control

Range: 0 .. 1

Units: Boolean

Default: 0

BrickAC.UnderVoltageWarnOnly controls whether an undervoltage condition should be treated as a warning or as an error. If it is set to the default value of 0, then Power Brick AC will treat it as a fault and stop all outputs with a fault on all channels if an undervoltage condition is detected. The amplifier stage must be reset by setting **BrickAC.Reset** to 1 in order to clear the fault and permit continued operation.

If **BrickAC.UnderVoltageWarnOnly** is set to 1, an undervoltage condition is treated only as a warning, with no automatic action taken. In either case, the **BrickAC.BusUnderVoltage** status bit reflects the current state of the undervoltage condition.

An undervoltage condition occurs when the internal DC bus voltage drops below 100 VDC, which corresponds to about 70 VAC(rms). The undervoltage status bit **BrickAC.BusUnderVoltage** is automatically cleared once the bus voltage is restored. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

The **BrickAC.UnderVoltageWarnOnly** value is sent to the active amplifier-control circuit upon setting one of the non-saved setup elements **BrickAC.Reset** or **BrickAC.Config** equal to 1 in a Script command. It does not take effect until then.

BrickAC. Single-Channel Saved Setup Elements

Some aspects of the Power Brick AC can be set up individually for each channel. The channel index ***j*** has a range from 0 to 7, corresponding to hardware channels 1 to 8, respectively, in the amplifier. The channel index value is one less than the corresponding hardware channel number.

BrickAC.Chan[*j*].I2tWarnOnly

Description: Channel I^2T warning/fault control

Range: 0 .. 1

Units: Boolean

Default: 0

BrickAC.Chan[*j*].I2tWarnOnly determines the course of action the amplifier hardware takes upon detection of an excess integrated current (I^2T) condition on the channel. If **BrickAC.Chan[*j*].I2tWarnOnly** is set to the default value of 0, then upon detection of a I^2T excess condition, an amplifier fault is generated, the motor is killed, the corresponding status bit **BrickAC.Chan[*j*].I2tExcess** is set, and the corresponding error code is displayed on the amplifier (Error Code `n.L`).

If **BrickAC.Chan[*j*].I2tWarnOnly** is set to a value of 1, the I^2T excess condition will be reported as a warning with the status bit, but it will not generate a fault on amplifier.

The **BrickAC.Chan[*j*].I2tWarnOnly** value is sent to the active amplifier-control circuit upon setting one of the non-saved setup elements **BrickAC.Reset** or **BrickAC.Config** equal to 1 in a Script command. It does not take effect until then.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The integrated current (I^2T) calculations accessed by this element are performed in the amplifier stage of the Power Brick AC. These calculations are separate from those done by the Power PMAC software.

BrickLV. Saved Data Structure Elements

The Power Brick LV is one class of the Power Brick family of intelligent amplifiers. It can support multiple types of brush, brushless and stepper motors, operating from a low-voltage DC input. Its registers can be accessed through **BrickLV.** data structure.

Note

The **BrickLV.** data structure elements documented in this section are software elements that are distinct from the **PowerBrick[*i*].** hardware data structure elements that form the control and status registers for the ASIC(s) in the Power Brick LV. (The **PowerBrick[*i*].** data structure is an "alias" for the **Gate3[*i*].** hardware data structure.)

BrickLV. Multi-Channel Saved Setup Elements

Some aspects of the Brick LV amplifier are common to all channels on the board. The setup elements in this section affect all channels.

BrickLV.MonitorPeriod

Description: Time interval for updating status registers

Range: 0 .. 2,147,483,647 (2^{31}-1)

Units: Milliseconds

Default: 0 (50 msec)

BrickLV.MonitorPeriod tells Power PMAC software how much time there is between consecutive requests for the value of all Brick LV status registers. It is expressed in milliseconds as an integer value.

If **BrickLV.MonitorPeriod** is set to the default value of 0 or any value up to 50, all Brick LV status elements are updated every 50 milliseconds. Setting the value higher will reduce the update frequency and reduces the background time which monitor process takes from the Power PMAC CPU.

Note

The value of **BrickLV.MonitorPeriod** does not affect how often the amplifier stage checks the status conditions internally. It only controls how frequently the Power PMAC CPU requests this information.

While the value of **BrickLV.MonitorPeriod** is saved, the element that starts the monitoring process itself, **BrickLV.Monitor**, is not a saved setup element. It must explicitly be set to 1 by the user application in order to start the monitoring process. Also, when either the configuration

process or the fault-clearing reset process is started with **BrickLV.Config** or **BrickLV.Reset**, respectively, the monitoring process is stopped, and it is not automatically restarted. The user application must explicitly restart the monitoring process.

Note

The monitored data in the Power Brick LV is provided to the controller on the lower 10 bits of the **Gate3[*i*].Chan[*j*].AdcAmp[*k*]** registers and it is essential that **Gate3[*i*].Chan[*j*].PackInData** and **Gate3[*i*].Chan[*j*].PackOutData** are set to 0, disabling "packed" register access and allowing all ADC register bits to be read by the CPU.

BrickLV. Single-Channel Saved Setup Elements

Some aspects of the Brick LV can be set up individually for each channel. The channel index ***j*** has a range from 0 to 7, corresponding to hardware channel 1 to 8 on the board. The channel index is one less than the corresponding hardware channel number.

BrickLV.Chan[*j*].I2tWarnOnly

Description: I^2T protection-level control

Range: 0 .. 1

Units: Boolean

Default: 0

BrickLV.Chan[*j*].I2tWarnOnly determines the course of action the amplifier hardware takes upon detection of an excess integrated current (I^2T) condition on the channel. If **BrickLV.Chan[*j*].I2tWarnOnly** is set to the default value of 0, then upon detection of a I^2T excess condition, an amplifier fault is generated, the motor is killed, the corresponding status bit is set, and the corresponding error code is displayed on the amplifier (Error Code `n.L`).

If **BrickLV.Chan[*j*].I2tWarnOnly** is set to a value of 1, the I^2T excess condition will be reported as a warning in the status register, but it will not generate a fault on amplifier.

The **BrickLV.Chan[*j*].I2tWarnOnly** value is sent to the active amplifier-control circuit upon setting one of the non-saved setup elements **BrickAC.Reset** or **BrickAC.Config** equal to 1 in a Script command. It does not take effect until then.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The integrated current (I^2T) calculations accessed by this element are performed in the amplifier stage of the Power Brick AC. These calculations are separate from those done by the Power PMAC software.

BrickLV.Chan[*j*].TwoPhaseMode

Description: Channel motor phase count control

Range: 0 .. 1

Units: Boolean

Default: 0

BrickLV.Chan[*j*].TwoPhaseMode selects the operational output mode of the amplifier channel. If set to its default value of 0, the amplifier is set to 3-phase operational mode, using the U, V, and W output lines. This operational mode is mainly used with Y-wound or delta-wound brushless servo motors (but 3-phase stepper motors do exist).

If the **BrickLV.Chan[*j*].TwoPhaseMode** is set to a value of 1, the amplifier channel is placed in 2-phase operational mode, using the U and W output lines to drive the first phase, and the V and X output lines to drive the second phase. This operational mode is mainly used with 2-phase stepper motors (but 2-phase brushless servo motors do exist).

Note

If the channel is put in 2-phase output mode with **BrickLV.Chan[*j*].TwoPhaseMode**, the Power PMAC motor commanding the channel should also be put in two-phase mode by setting bit 0 (value 1) of **Motor[*x*].PhaseMode** to 1.

Note

DC brush motors, voice-coil motors, and other similar "two-lead" motors that do not require electronic commutation can be driven between the U and W output lines with either setting of this element. However, it is recommended in this case to leave **BrickLV.Chan[*j*].TwoPhaseMode** at its default value of 0, so less processing of the commanded PWM signals is required.

The **BrickLV.Chan[*j*].TwoPhaseMode** value is sent to the active amplifier control circuits upon setting **BrickLV.Config** to 1 in a Script command. The user can check the operational mode of each channel by setting the **BrickLV.Monitor** equal to 1 in a Script command and reading the **BrickLV.Chan[*j*].ActivePhaseMode** value.

BufIo[*i*]. Buffered Input/Output Saved Data Structure Elements

The **BufIo[*i*]** data structure contains saved setup elements for the buffered input/output functionality.

BufIo[*i*].InScans

Description: Buffered input additional number of scans to confirm bit change

Range: 0 .. 3

Units: Input scans

Default: 0

BufIo[*i*].InScans specifies the number of additional scans reading the input specified by **BufIo[*i*].pIn** required to confirm a change in any bit of the input word. That is, the input bit must be in the new state for (**InScans** + 1) consecutive scans for the corresponding bit in the holding register **BufIo[6].In** to change and for the corresponding bit in **BufIo[6].RiseIn** or **BufIo[6].FallIn** to be set to mark the change for the user algorithms.

This ability to require multiple scans in the new state before registering the change permits automatic protection against electrical noise and contact "bounce". Of course, this filtering does cause a delay in responding to an actual change in an input.

All bits in the same input word are subject to the same filter scan length. Different input words can have different filter scan lengths.

BufIo[*i*].InScans is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].pIn

Description: Buffered PLC-style input register address

Range: 0, Legitimate addresses

Units: Data structure element addresses

Default: 0 (no input)

BufIo[*i*].pIn specifies the address of the "***i***th" 32-bit input register that Power PMAC will read each scan and copy into a memory holding register if **Sys.BufIoEnable** is set to 1. The index value "***i***" can take a value of 0 to 63.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Buffered reads of inputs can be performed at the start of a real-time interrupt (RTI) cycle and/or a background cycle. Input registers addressed by **BufIo[*i*].pIn** with index values ***i*** from 0 through (**Sys.MaxRtBufIn**-1) can be (but are not necessarily) read at the beginning of an RTI cycle. Input registers addressed by **BufIo[*i*].pIn** with index values ***i*** from **Sys.MaxRtBufIn** through 63 can be (but are not necessarily) read at the beginning of a background cycle.

If **BufIo[*i*].pIn** is set to the default value of 0, no register is read for this index or any higher index for that cycle (RTI or background), so the first **BufIo[*i*].pIn** that is set to 0 prevents any use of elements with a higher index number for that cycle, even if those elements are not set to 0.

If **BufIo[*i*].pIn** is set to the address of a data structure element (and no **BufIo[*i*].pIn** in that cycle with a lower index number is set to 0), then each scan Power PMAC will copy the 32-bit value at that address into memory holding element **BufIo[*i*].pIn**. This is a full 32-bit copy operation even if there is not real hardware in all 32 bits of the source address.

If **BufIo[*i*].InScans** is greater than 0, changes in a bit of the source input register will not be reflected in the holding register **BufIo[*i*].In** until they have been seen in (**InScans** + 1) consecutive scans.

Power PMAC will also compare the new value of **BufIo[*i*].In** to the previous scan's value in a bit-by-bit fashion, and set bits of **BufIo[*i*].RiseIn** to 1 if the corresponding bits changed from 0 to 1 in the most recent scan, and bits of **BufIo[*i*].FallIn** to 1 if the corresponding bits changed from 1 to 0 in the most recent scan.

While the specified registers are typically hardware input registers, there are a few elements in memory that can be used as well:

- **Coord[*x*].Status[*i*]**
- **CtrlPanel[*i*].Input[*j*]**
- **ECAT[*i*].IO[*j*].Data**
- **ECAT[*i*].LPIO[*j*].Data**
- **Motor[*x*].Status[*i*]**
- **MuxIo.Port*α*[*i*]**
- **Sys.Status**
- **Sys.Udata[*i*]**

Examples of settings are:

- **BufIo[0].pIn** = **Acc68E[1].Data[2].a**
- **BufIo[1].pIn** = **PowerBrick[0].GpioData[0].a**
- **BufIo[2].pIn** = **Clipper[1].GpioData[0].a**
- **BufIo[3].pIn = Acc5E3[0].MacroInA[6][0].a**
- **BufIo[4].pIn** = **ECAT[0].IO[24].Data.a**
- **BufIo[5].pIn** = **Acc72EX[1].Udata32[301].a**
- **BufIo[6].pIn** = **Motor[5].Status[0].a**
- **BufIo[7].pIn** = **Coord[1].Status[0].a**
- **BufIo[8].pIn** = **CtrlPanel[0].Input[3].a**
- **BufIo[9].pIn** = **MuxIo.PortA[4].Data.a**
- **BufIo[10].pIn** = **Sys.Udata[347].a**

BufIo[*i*].pIn is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].pOut

Description: Buffered PLC-style output register address

Range: 0, Legitimate addresses

Units: Data structure element addresses

Default: 0 (no write)

BufIo[*i*].pOut specifies the address of the "***i***th" 32-bit output register that Power PMAC will write to each scan and copy from a memory holding register if **Sys.BufIoEnable** is set to 1. The index value "***i***" can take a value of 0 to 63.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Buffered writes of outputs can be performed at the end of a real-time interrupt (RTI) cycle and/or a background cycle. Output registers addressed by **BufIo[*i*].pOut** with index values ***i*** from 0 through (**Sys.MaxRtBufOut**-1) can be (but are not necessarily) written to at the end of an RTI cycle. Output registers addressed by **BufIo[*i*].pOut** with index values ***i*** from **Sys.MaxRtBufOut** through 63 can be (but are not necessarily) written at the end of a background cycle.

If **BufIo[*i*].pOut** is set to the default value of 0, no register is read for this index or any higher index for that cycle (RTI or background), so the first **BufIo[*i*].pOut** that is set to 0 prevents any use of elements with a higher index number for that cycle, even if those elements are not set to 0.

If **BufIo[*i*].pOut** is set to the address of a data structure element (and no **BufIo[*i*].pOut** in that cycle with a lower index number is set to 0), then each scan Power PMAC will copy the 32-bit value from memory holding element **BufIo[*i*].Out** to the register at that address. This is a full 32-bit copy operation even if there is not real hardware in all 32 bits of the target address.

The copying operation will only occur in a given scan if the value of this holding register has changed from that of the previous scan. This prevents unneeded slow I/O accesses from occurring.

While the specified registers are typically hardware output registers, there are a few elements in memory that can be used as well:

- **CtrlPanel[*i*].Output[*j*]**
- **ECAT[*i*].IO[*j*].Data**
- **ECAT[*i*].LPIO[*j*].Data**
- **MuxIo.Port*α*[*i*]**
- **Sys.Udata[*i*]**

Examples of settings are:

- **BufIo[0].pOut** = **Acc68E[1].Data[5].a**
- **BufIo[1].pOut** = **PowerBrick[0].GpioData[0].a**
- **BufIo[2].pOut** = **Clipper[1].GpioData[0].a**
- **BufIo[3].pOut = Acc5E3[0].MacroOutA[7][1].a**
- **BufIo[4].pOut** = **ECAT[0].IO[48].Data.a**
- **BufIo[5].pOut** = **Acc72EX[1].Udata32[301].a**
- **BufIo[6].pOut** = **CtrlPanel[0].Output[1].a**
- **BufIo[7].pOut** = **MuxIo.PortB[12].Data.a**
- **BufIo[8].pOut** = **Sys.Udata[426].a**

BufIo[*i*].pOut is new in V2.1 firmware, released 1st quarter 2016.

CamTable[*m*]. Saved Data Structure Elements

The **CamTable[*m*]** data structure implements Power PMAC's cam table following functionality. Power PMAC can store 256 cam tables at one time (index ***m*** values from 0 to 255).

CamTable[*m*].DacData[*i*]

Description: Cam table torque-offset command entry

Range: Floating-point

Units: User-defined torque position units

Default: nan (not-a-number)

CamTable[*m*].DacData[*i*] specifies the value of a specific torque-offset command entry of the cam table. Torque-offset commands from the table are computed by interpolation between adjacent entries based on the present position of the source motor if this function is enabled by **CamTable[*m*].DacEnable**. The command is then multiplied by the scale factor term **CamTable[*m*].DacSf** before being written to the **Motor[*x*].CompDac** register for the target motor, where it will be automatically added to the servo-loop output of the target motor. If the scale factor is at its default value of 1.0, the **DacData[*i*]** entries will be in the 16-bit output units of the target motor.

Note that while the resulting torque command is often written to a D/A converter – the reason for the name of this element – there is no requirement that this be true. This feature will work equally well with other forms of command signals, and when the torque command is used as an input to Power PMAC's commutation and current-loop algorithms.

CamTable[*m*].DacData[0] specifies the torque-offset command when the source motor is at the position specified by **CamTable[*m*].X0**. **CamTable[*m*].DacData[1]** specifies the torque-offset command when the source motor is at the position specified by (**X0** + **Dx**/**Nx**). **CamTable[*m*].DacData[2]** specifies the torque-offset command when the source motor is at the position specified by (**X0** + 2***Dx**/**Nx**), and so on. **CamTable[*m*].DacData[*Nx-1*]** specifies the torque-offset command when the source motor is at the position specified by (**X0** + [**Nx**-1]***Dx**/**Nx**).

The operation of cam tables always rolls over when the source motor position is outside of the range (**X0** to **X0** + **Dx**). This rollover functionality is always "returning", so the torque offsets are always in the same range each cycle. To support this rollover, at the end of the table, Power PMAC interpolates between table points **CamTable[*m*].DacData[*Nx-1*]** and **CamTable[*m*].DacData[0]**. While there exists an entry for **CamTable[*m*].DacData[*Nx*]**, it is not used in any table computations.

CamTable[*m*].DacEnable

Description: Cam table torque output enable control

Range: 0 .. 1

Units: Boolean

Default: 0

CamTable[*m*].DacEnable specifies whether the "torque" offset command for the table is computed from the **DacData[*i*]** entries for the table, and written to the **CompDac** register of the target motor or not. If it is set to the default value of 0, this command is not computed or written. This saves processor time and leaves the target motor's **CompDac** register available for torque compensation tables using that motor's position.

If **CamTable[*m*].DacEnable** is set to 1 when the table is enabled, the "torque" offset command will be computed and written to the target motor's **CompDac** register. In this case, a torque compensation table should not be used on the same target motor.

CamTable[*m*].DacGain

Description: Cam table iterative learning control correction gain

Range: Non-negative floating-point.

Units: 16-bit DAC equivalent per motor position unit per cam cycle

Default: 0.0

CamTable[*m*].DacGain specifies the corrective gain of the automatic "iterative learning control" (ILC) algorithm for the table. The higher the gain value, the quicker the algorithm will try to converge on a minimum error condition, but the greater the chance of overshooting the correction or getting limit cycling around the optimal correction. The ILC algorithm will only execute if **DacGain** is greater than 0.0.

It is recommended to start with a very small value of **DacGain** (~1.0) and gradually increase until desired performance is achieved.

CamTable[*m*].DacGain is new in V2.0 firmware, released 1st quarter 2015.

CamTable[*m*].DacSf

Description: Cam table torque scale factor

Range: Floating-point

Units: Target motor 16-bit output units per table torque unit

Default: 0.0

CamTable[*m*].DacSf specifies the commanded scale factor that multiplies the "torque" offset command computed by the table before it is written to the desired servo-output compensation register (**Motor[*x*].CompDac**) for the target motor. If it is set to its default value of 1.0, the table torque offset entries are in the units of a signed 16-bit output.

CamTable[*m*].Dx

Description: Cam table source position span

Range: Non-negative floating-point

Units: Motor units of the source

Default: 0.0

CamTable[*m*].Dx specifies the span distance of the source motor for the table, in units of the source motor. The minimum position used directly from the source motor is specified by **CamTable[*m*].X0**. The maximum position used directly from the source motor is (**CamTable[*m*].X0** + **CamTable[*m*].Dx**). Source motor positions from outside this range are automatically "rolled over" to within this range. The spacing between table points is (**CamTable[*m*].Dx** / **CamTable[*m*]/Nx**), where **Nx** is the number of zones in the table.

CamTable[*m*].MaxDac

Description: Cam table maximum automatically set torque offset magnitude

Range: 0.0 .. 32,767.999

Units: 16-bit DAC equivalent

Default: 0.0

CamTable[*m*].MaxDac specifies the magnitude of the maximum torque *offset* value that will be applied to a **CamTable[*m*].DacData[*i*]** element by the automatic "iterative learning control" (ILC) algorithm for the table. It does *not* limit what a user can directly write to one of these elements.

The ILC algorithm works to minimize position errors that are repeated in multiple cycles of the table by adjusting the **DacData[*i*]** values in each zone of the table to provide appropriate torque offsets for the zone.

CamTable[*m*].MaxDac is new in V2.0 firmware, released 1st quarter 2015.

CamTable[*m*].MinPosError

Description: Cam table minimum following error for automatic torque adjustment

Range: Non-negative floating-point

Units: Target motor position units

Default: 0.0

CamTable[*m*].MinPosError specifies the magnitude of the minimum following error for which the "iterative learning control" (ILC) algorithm will automatically adjust the value of a **CamTable[*m*].DacData[*i*]** element to try to reduce the error in subsequent table cycles.

The ILC algorithm works to minimize position errors that are repeated in multiple cycles of the table by adjusting the **DacData[*i*]** values in each zone of the table to provide appropriate torque offsets for the zone.

CamTable[*m*].MinPosError is new in V2.0 firmware, released 1st quarter 2015.

CamTable[*m*].Nx

Description: Cam table number of zones

Range: Non-negative integers

Units: Table zones

Default: 0

CamTable[*m*].Nx specifies the number of zones in the cam table. A zone is the space between two adjacent points in the table. Data points **CamTable[*m*].PosData[*i*]**, **DacData[*i*]**, and **OutData[*i*]** should be entered for the table with index values ***i*** ranging from 0 through **Nx**, inclusive, for a total of **Nx** + 1 data points of each type for the table.

CamTable[*m*].OutBits

Description: Cam table general-purpose output number of bits

Range: 0 .. 32

Units: Number of bits

Default: 0

CamTable[*m*].OutBits specifies the number of consecutive bits, starting at bit 0 (the LSB), in the general-purpose output word in the **CamTable[*m*].OutData[*i*]** element for the present zone of the table will actually be written to the specified register. Any higher bits in **OutData[*i*]** are masked out before the resulting value is shifted left by the number of bits specified in **CamTable[*m*].OutLeftShift** before the masked write to the register specified by **CamTable[*m*].pOut** or **CamTable[*m*].pOutMask** is performed.

For example, if **OutBits** is set to 4, and **OutLeftShift** is set to 12, the values in bits 0 – 3 of **OutData[*i*]** are written to bits 12 – 15 of the 32-bit output register. No other bits of the output register will be affected by this table.

CamTable[*m*].OutData[*i*]

Description: Cam table general-purpose output command entry

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: nan (not-a-number)

CamTable[*m*].OutData[*i*] specifies the value of the general-purpose output word for the specified zone of the table. This output word will be used when the position of the source motor is in the range specified by

(**X0** + ***i******Nx*****Dx**) <= Pos < (**X0** + [***i***+1]***Nx*****Dx**).

CamTable[*m*].OutData[*i*] is a 32-bit word, so up to 32 discrete output bits or a single value of up to 32 bits can be specified. Only the lowest *N* bits of the active **OutData[*i*]** element are used, where *N* is specified by **CamTable[*m*].OutBits**; the other bits are masked out. The resulting value is shifted left by the number of bits specified by **CamTable[*m*].OutLeftShift**, and the specified bits are then written to the output register specified by **CamTable[*m*].pOut**.

CamTable[*m*].OutLeftShift

Description: Cam table general-purpose output word number of bits to left shift

Range: 0 .. 31

Units: Bits

Default: 0

CamTable[*m*].OutLeftShift specifies the number of bits that the general-purpose output word in the **CamTable[*m*].OutData[*i*]** element for the present zone of the table is shifted left before being written to the specified output register. This permits the utilization of hardware mapped into high bits of the 32-bit bus without the table's **OutData[*i*]** values each needing to be shifted to reflect this.

For example, the outputs of the UMAC general-purpose digital I/O boards such as the ACC-14E, ACC-65E, and ACC-68E are mapped into bits 8 – 15 of the 32-bit bus. By setting **OutLeftShift** to 8, the values in each **OutData[*i*]** can use bits 0 – 7, which is easier for most users. Alternately, if the general-purpose outputs are used to write to a 16-bit D/A converter mapped into bits 16 – 31 of the 32-bit bus, **OutLeftShift** could be set to 16 to let the user write the 16-bit DAC values in the low 16 bits of each **OutData[*i*]** table entry.

CamTable[*m*].PosBias

Description: Cam table desired slave position bias

Range: Floating-point

Units: Target motor position units

Default: 0.0

CamTable[*m*].PosBias specifies the commanded target-motor user bias for the cam table. This is the steady-state value to be added to the position value calculated from the table **PosData[*i*]** entries each servo cycle after being multiplied by **CamTable[*m*].PosSf** scale factor.

The functionality of **CamTable[*m*].PosBias** is very similar to that of **CamTable[*m*].PosOffset**, which also specifies a value to be added into the value calculated from the table entries. However, Power PMAC automatically writes to the **PosOffset** element each time the table is enabled in order to provide a smooth transition for the target motor to lock in on the table-commanded position. No automatic functions write to the **PosBias** element, so it is more useful for user-specified offsets.

When the value of **CamTable[*m*].PosBias** is changed, the actual bias value used in a given servo cycle is incremented by **CamTable[*m*].SlewPosOffset** until the new desired value is reached.

CamTable[*m*].PosBias is new in V2.0 firmware, released 1st quarter 2015.

CamTable[*m*].PosData[*i*]

Description: Cam table position command entry

Range: Floating-point

Units: User-defined table position units

Default: nan (not-a-number)

CamTable[*m*].PosData[*i*] specifies the value of a specific position-command entry of the cam table. Net position commands from the table are computed by interpolation between adjacent entries based on the present position of the source motor. The command is then multiplied by the scale factor term **CamTable[*m*].ActivePosSf** before being written to the **Motor[*x*].CompDesPos** register for the target motor. If the scale factor is at its default value of 1.0, the **PosData[*i*]** entries will be in the position units of the target motor.

CamTable[*m*].PosData[0] specifies the position command when the source motor is at the position specified by **CamTable[*m*].X0**. **CamTable[*m*].PosData[1]** specifies the position command when the source motor is at the position specified by (**X0** + **Dx**/**Nx**). **CamTable[*m*].PosData[2]** specifies the position command when the source motor is at the position specified by (**X0** + 2*__Dx__/__Nx__), and so on. **CamTable[*m*].PosData[*Nx-1*]** specifies the position command when the source motor is at the position specified by (**X0** + [**Nx**-1]*__Dx__/__Nx__).

The operation of cam tables always rolls over when the source motor position is outside of the range (**X0** to **X0** + **Dx**). This rollover functionality can be "returning" or "non-returning". With "returning" functionality, which is typically used for linear actuation, the end point of each cycle must be the same as the start point. In this case, **CamTable[*m*].PosData[*Nx*]** must have the same value as **CamTable[*m*].PosData[0]**.

With "non-returning" functionality, which is typically used for rotary actuation, as with printing and cutting wheels, the end point of each cycle must be different from the start point, usually by one full revolution of the rotary device. In this case, **CamTable[*m*].PosData[*Nx*]** must have a different value from **CamTable[*m*].PosData[0]**. This difference will be the size of the cycle – usually a revolution of the rotary device.

CamTable[*m*].PosSf

Description: Cam table position scale factor

Range: Floating-point

Units: Target motor units per table position unit

Default: 0.0

CamTable[*m*].PosSf specifies the commanded scale factor that multiplies the interpolated position command computed by the table before it is written to the desired position compensation register (**Motor[*x*].CompDesPos**) for the target motor. If it is set to the default value of 1.0, the table position entries are in the position units of the target motor.

CamTable[*m*].pOut

Description: Cam table general-purpose output register address

Range: Valid Power PMAC addresses

Units: Power PMAC addresses

Default: 0 (discrete outputs disabled)

CamTable[*m*].pOut specifies the address of the register to which the general-purpose outputs for the present zone of the table are written after masking and shifting. The value of this variable is usually set by assigning to the address of the data structure element representing the register used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address. However, it is also possible to specify the address numerically, which is useful if the register does not have a defined data structure element (as with the old ACC-11E I/O board).

For example,

```
CamTable[0].pOut = Acc68E[0].DataReg[4].a          // GateIo[0]

CamTable[1].pOut = PowerBrick[1].GpioData[0].a     // Gate3[1]
```

CamTable[2].pOut = **Acc24E3[2].Chan[3].Dac[1].a** // **Gate3[2]**

CamTable[3].pOut = **Sys.piom** + $A0000C // for ACC-11E

CamTable[4].pOut = **Acc5E[0].Macro[2][1].a** // **Gate2[0]**

CamTable[5].pOut = **Acc5EP3[1].MacroOutA[6][0].a** // **Gate3[1]**

In some applications, the direct-output function of a cam table may be used for a purpose other than setting physical outputs, such as driving a software state machine. In this case, a memory register, often in the user buffer, will typically be used as the "output register". In this case, the setting would be something like:

CamTable[6].pOut = **Sys.Udata[475].a**

If the register specified by **CamTable[6].pOut** has the property that values written to it cannot be read back (i.e. it is a "write-only" register), then it is not possible for the table to modify particular bits of the register. This is true for PMAC2-style MACRO output registers, as with the **Acc5E[*i*].Macro[*j*][*k*]** registers, and the PMAC3-style output registers, as with the **PowerBrick[*i*].GpioData[*j*]** registers.

In this case, the table must be set up to write to a buffering register in which it can modify particular bits by setting **CamTable[6].pOutBuf** to the address of this buffering register.

When this is done, the specified bits of the buffering register can be modified individually, and then the entire 32-bit value in the buffering register is copied to the actual output register specified by **pOut** each servo cycle. This means that no other task, whether another cam table or a user program, should write directly to the output register, because this table action will overwrite the entire output register every servo cycle.

CamTable[*m*].pOutBuf

Description: Cam table general-purpose output holding register address

Range: Valid Power PMAC addresses

Units: Power PMAC addresses

Default: 0 (no holding register)

CamTable[*m*].pOutBuf specifies the address of the holding register for general-purpose outputs from the table. It is used if multiple tables are setting outputs in the same physical output register. Without the use of a holding register, higher-numbered tables would overwrite the values set by lower-numbered tables, even if specifying different output bits in this register.

If **CamTable[*m*].pOutBuf** is set to 0, this table does not use a holding register for its general-purpose outputs. If it is set to a non-zero value, the table will set and clear the bits in the holding register as specified by **CamTable[*m*].OutMask** and **CamTable[*m*].OutLeftShift** using read/modify/write operations on the specified register. These operations will not change other bits in the register, provided that a read/write register (such as any memory register) is specified. This

means that multiple tables can control bits in the same register without conflict, as long as no two tables try to control any of the same bits.

In this case, the table does not write directly to the output register specified by **CamTable[*m*].pOut**; in this case, after all of the tables have updated in a servo cycle, the value from the holding register is copied to the output register.

The general-purpose input/output registers of the DSPGATE3 ICs, such as those used for the I/O in a Power Brick, do not always permit you to read back bit values you have written to the register, so if **CamTable[*m*].pOut** is set to **Gate3[*i*].GpioData[*j*].a**, a buffered holding register should be used. The **Gate3[*i*]** data structure provides a special holding register for each of its I/O registers, so in this case, **CamTable[*m*].pOutBuf** should be set to **Gate3[*i*].GpioOutData[*j*].a**, the address of this holding register.

The MACRO output registers of the DSPGATE2 ICs, such as those used in the ACC-5E, do not permit you to read back bit values you have written to the register, so if **CamTable[*m*].pOut** is set to **Gate2[*i*].Macro[*j*][*k*].a**, a buffered holding register should be used. In this case, a memory register should be used, such as one in the user buffer, so **CamTable[*m*].pOutBuf** is usually set to something like **Sys.Udata[1297].a**.

CamTable[*m*].SlewPosOffset

Description: Cam table position offset rate of change

Range: Floating-point

Units: Target motor units per servo cycle

Default: 0.0

CamTable[*m*].SlewPosOffset specifies the rate of change of the actual position offset (found in status element **CamTable[*m*].ActivePosOffset**) that is used to bring the target motor position into synchronization with the source motor when the cam table is enabled. It is effectively the target motor velocity used to establish synchronization. If the user changes **CamTable[*m*].PosOffset** or **CamTable[*m*].PosBias** while the table is enabled, this element specifies the rate of change of the actual position offset used to reach this new value.

If **CamTable[*m*].SlewPosOffset** is at its default value of 0.0, then when the table is enabled, the target motor position will not be able to pull into synchronization with the table commanded position.

CamTable[*m*].SlewX0

Description: Cam table starting source position rate of change

Range: Floating-point

Units: Motor units of the source per servo cycle

Default: 0.0 (slew control disabled)

CamTable[*m*].SlewX0 specifies the rate of change of the reference position of the source motor that is used each servo cycle in the table calculations (**CamTable[*m*].ActiveX0**) when the commanded reference position **CamTable[*m*].X0** is changed. It is expressed in units of the source motor per servo cycle. If **SlewX0** is set to its default value of 0.0, the value of **ActiveX0** will jump immediately to a new value of **X0**.

This element allows for controlled changes in the source motor reference position so the table can be "phased in", even when active, without the potential for violent reactions.

CamTable[*m*].Source

Description: Cam table source motor number

Range: 0 .. 255

Units: Motor number

Default: 0

CamTable[*m*].Source specifies the number of the Power PMAC motor whose desired position is used as the "source" of the position in the cam table. The cam motion of the "target" motor, and the state of any table-driven outputs, will be a function of this source motor's position when the table is enabled.

Note that even if there is not a physical motor attached to the position sensor used as the master, or the physical motor is not under Power PMAC control, the sensor must be used as the position for an active Power PMAC "motor". In these cases, the Power PMAC motor should be left in the disabled (killed) state, in which case Power PMAC copies the actual position value to the desired position register every servo cycle.

In some cases, a virtual Power PMAC motor with no physical actuator or sensor may be used as the source motor for one or more cam tables, providing a mathematical substitute for a mechanical line shaft in a mechanical cam system.

CamTable[*m*].Target

Description: Cam table target motor number

Range: 0 .. 255

Units: Motor number

Default: 0

CamTable[*m*].Target specifies the number of the motor whose action is controlled by the table. This motor's **Motor[*x*].CompDesPos** desired position offset register will be set by the position

data in the table, and its **Motor[*x*].CompDac** torque offset register will be set by the torque data in the table.

CamTable[*m*].X0

Description: Cam table starting source position

Range: Floating-point

Units: Motor units of the source

Default: 0.0

CamTable[*m*].X0 specifies the commanded starting (minimum) position of the source motor for the table. Data points **CamTable[*m*].PosData[0]**, **DacData[0]**, and **OutData[0]** represent the table's values at this source motor position. The table is directly specified for the range of positions from **CamTable[*m*].X0** to (**CamTable[*m*].X0** + **CamTable[*m*].Dx**). The table automatically rolls over for positions outside of this declared range.

This position is expressed in the motor units for the source, relative to the motor's zero position. The value of **CamTable[*m*].X0** can be changed at any time. If changed, the reference position used each servo cycle in the table calculations, status element **CamTable[*m*].ActiveX0**, will approach the new commanded **X0** value at the rate specified by saved setup element **CamTable[*m*].SlewX0**.

Clipper[*i*]. Saved Data Structure Elements

The **Clipper[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the Power Clipper embedded controller are listed under the **Gate3[*i*]** data structure, below.

CompTable[*m*]. Saved Data Structure Elements

The **CompTable[*m*].** data structure implements the Power PMAC compensation tables, which can automatically apply corrections to the measured positions or commanded servo outputs of motors. Power PMAC has 256 compensation tables, so the table index value ***m*** has a range of 0 to 255. Global saved setup variable **Sys.CompEnable** specifies how many of these tables are active; the tables with index ***m*** from 0 to **Sys.CompEnable** – 1 are active.

CompTable[*m*].Ctrl

Description: Compensation table correction mode control

Range: $00 .. $FF (0 – 255)

Units: Bit field

Default: $03

CompTable[*m*].Ctrl specifies several aspects of the mode of operation of the compensation table. It is an 8-bit value, organized as four 2-bit controls.

Bits 0 and 1 determine the order of interpolation in the dimension from each source. Together they can take the following values:

= 0 (00): 1st-order interpolation from all sources

= 1 (01): 3rd-order interpolation from **Source[0]**, 1st-order from others

= 2 (10): 3rd-order interpolation from **Source[0]** and **Source[1]**, 1st-order from **Source[2]**

= 3 (11): 3rd-order interpolation from all sources

With first-order interpolation, the correction in the dimension of the source is calculated as a linear fit between the point on either side of the present position. It can have sudden changes in slope as it passes a point in the table, which may result in noticeably rough motion.

With third-order interpolation, the correction in the dimension of the source is calculated as a cubic fit using two points on either side of the present position. The slope of the correction is always continuous, even as a table point is passed. This interpolation takes about twice the calculation time of first-order interpolation.

The order of interpolation specified for an unused source (**Nx[*n*]**=0) does not matter. However, for a "0D" table (all **Nx[*n*]** values = 0), if this 2-bit value is greater than 0, then Power PMAC will take the servo-output command value from the motor specified by **Source[0]** and place it in the single data point for the table (**Data[0]**). From there, it can be used as a position-compensation value, permitting the cascading of servo loops.

Bits 2 and 3 determine the "boundary mode" of the table at the edges of the table in the dimension from **Source[0]**.

Bits 4 and 5 determine the "boundary mode" of the table at the edges of the table in the dimension from **Source[1]**.

Bits 6 and 7 determine the "boundary mode" of the table at the edges of the table in the dimension from **Source[2]**.

In each dimension, the two bits can take the following values:

=0 (00): Rollover at table boundary in this dimension

=1 (01): Maintain last correction past the table boundary in this dimension

=2 (02): Mirror the table correction at the boundary in this dimension

=3 (03): (Reserved for future use)

CompTable[*m*].DacEnable

Description: Compensation table torque learning enable control

Range: 0 .. 1

Units: Boolean

Default: 0 (disabled)

CompTable[*m*].DacEnable specifies whether the "torque" learning algorithm for the table is enabled or not. If it is set to the default value of 0, this algorithm is not enabled, and the compensation table type (dimension and target) is unconstrained.

If **CompTable[*m*].DacEnable** is set to 1, the automatic "iterative learning control" (ILC) algorithm for the table is enabled. In this case, the value of **CompTable[*m*].DacTarget** specifies the number of the "target" motor, and the computed correction value from the table is automatically written to the **CompDac** register for this motor, making the table a torque compensation table.

CompTable[*m*].DacEnable should only be set to 1 0 for a one-dimensional (1D) table (**Nx[1]** = 0, **Nx[2]** = 0), because the internal algorithms used will treat the table as a 1D table.

In operation, the ILC algorithm will continually evaluate position errors in the target motor in each zone of the table, and automatically modify table values to try to reduce those errors. The algorithm uses saved setup elements **CompTable[*m*].DacGain**, **MaxDac**, and **MinPosError** to compute how it will modify the table values.

CompTable[*m*].DacEnable is new in V2.1 firmware, released 1st quarter 2016. At its default value of 0, operation is compatible with older firmware versions.

CompTable[*m*].DacGain

Description: Compensation table iterative learning control correction gain

Range: Non-negative floating-point.

Units: 16-bit DAC equivalent per motor position unit per table cycle

Default: 0.0

CompTable[*m*].DacGain specifies the corrective gain of the automatic "iterative learning control" (ILC) algorithm for the table. It is only used if **CompTable[*m*].DacEnable** is set to 1 to enable the ILC algorithm. The higher the gain value, the quicker the algorithm will try to converge on a minimum error condition, but the greater the chance of overshooting the correction or getting limit cycling around the optimal correction.

The ILC algorithm will only be able to compute corrections if **DacGain** is greater than 0.0. Some users will set **DacGain** greater than 0.0 for a calibration period, then to 0.0 to "freeze" the table values that have minimized the errors during the calibration period.

Initially, it is recommended to start with a very small value of **DacGain** (~1.0) and gradually increase until desired performance is achieved.

CompTable[*m*].DacGain is new in V2.1 firmware, released 1st quarter 2016.

CompTable[*m*].DacTarget

Description: Compensation table torque learning target motor

Range: 0 .. 255

Units: Motor number

Default: 0

CompTable[*m*].DacTarget specifies the target motor number for the compensation table when the automatic "iterative learning control" (ILC) algorithm for the table is enabled by setting **CompTable[*m*].DacEnable** to 1. In this case, the value of **CompTable[*m*].DacTarget** specifies the number of the "target" motor, and the computed correction value from the table is automatically written to the **CompDac** register for this motor, making the table a torque compensation table. (When **DacEnable** is 1, none of the **CompTable[*m*].Target[*q*]** values is used, but **CompTable[*m*].Sf[0]** is used to scale the table output value.)

CompTable[*m*].DacEnable should only be set to 1for a one-dimensional (1D) table (**Nx[1]** = 0, **Nx[2]** = 0), because the internal algorithms used will treat the table as a 1D table.

In operation, the ILC algorithm will continually evaluate position errors in the target motor in each zone of the table, and automatically modify table values to try to reduce those errors. The algorithm uses saved setup elements **CompTable[*m*].DacGain**, **MaxDac**, and **MinPosError** to compute how it will modify the table values.

CompTable[*m*].DacTarget is new in V2.1 firmware, released 1st quarter 2016. At its default value of 0, operation is compatible with older firmware versions.

CompTable[*m*].Data[*i*]

Description: Compensation table single-dimensional entry

Range: Floating-point

Units: User-defined table units

Default: nan (not-a-number)

CompTable[*m*].Data[*i*] specifies the value of a specific entry (correction) of a one-dimensional table. Corrections are computed by interpolation between adjacent entries. The correction is then multiplied by a scale factor before being written to the target register. Usually the scale factor is 1.0, so these entries are specified in units of the target register (position or torque).

CompTable[*m*].Data[0] specifies the correction when the motor specified by **Source[0]** is at the position specified by **X0[0]**. **CompTable[*m*].Data[1]** specifies the correction when this motor is at the position (**X0[0]** + **Dx[0]**/**Nx[0]**). **CompTable[*m*].Data[2]** specifies the correction when this motor is at the position (**X0[0]** + 2*__Dx[0]__/__Nx[0]__), and so on.

If the table is not specified for rollover in this dimension, the final table entry **CompTable[*m*].Data[*Nx[0]*]** specifies the correction when this motor is at the position (**X0[0]** + **Dx[0]**). If the table is specified for rollover in this dimension, only the points **CompTable[*m*].Data[0]** through **CompTable[*m*].Data[*Nx[0]-1*]** are used.

For one-dimensional data points, the point index value ***i*** can take any integer constant value from 0 to 16,777,215 (within the range declared by **Nx[*n*]**) or any local L-variable. No fractional constants or mathematical expressions are permitted.

CompTable[*m*].Data[*j*][*i*]

Description: Compensation table two-dimensional entry

Range: Floating-point

Units: User-defined table units

Default: nan (not-a-number)

CompTable[*m*].Data[*j*][*i*] specifies the value of a specific entry (correction) of a two-dimensional table. Corrections are computed by interpolation between adjacent entries in both dimensions. The correction is then multiplied by a scale factor before being written to the target register. Usually the scale factor is 1.0, so these entries are specified in units of the target register (position or torque).

CompTable[*m*].Data[0][0] specifies the correction when the motor specified by **Source[0]** is at the position specified by **X0[0]** and the motor specified by **Source[1]** is at the position specified by **X0[1]**.

CompTable[*m*].Data[0][1] specifies the correction when motor specified by **Source[0]** is at the position (**X0[0]** + **Dx[0]**/**Nx[0]**) and the motor specified by **Source[1]** is at the position specified by **X0[1]**.

CompTable[*m*].Data[1][0] specifies the correction when motor specified by **Source[0]** is at the position **X0[0]** and the motor specified by **Source[1]** is at the position specified by (**X0[1]** + **Dx[1]**/**Nx[1]**).

If the table is not specified for rollover in the dimension of the motor specified by **Source[0]**, the boundary table entry **CompTable[*m*].Data[0][*Nx[0]*]** specifies the correction when motor specified by **Source[0]** is at the position (**X0[0]** + **Dx[0]**) and the motor specified by **Source[1]** is at the position specified by **X0[1]**. If the table is specified for rollover in this dimension, only the points **CompTable[*m*].Data[0][0]** through **CompTable[*m*].Data[0][*Nx[0]-1*]** are used.

If the table is not specified for rollover in the dimension of the motor specified by **Source[1]**, the boundary table entry **CompTable[*m*].Data[*Nx[1]*][0]** specifies the correction when motor specified by **Source[1]** is at the position (**X0[1]** + **Dx[1]**) and the motor specified by **Source[0]** is at the position specified by **X0[0]**. If the table is specified for rollover in this dimension, only the points **CompTable[*m*].Data[0][0]** through **CompTable[*m*].Data[*Nx[1]-1*][0]** are used.

Starting in V2.0 firmware, released 1st quarter 2015, for two-dimensional data points, the point index values ***i*** and ***j*** can take any integer constant value from 0 to 64,511 (within the range declared by **Nx[*n*]**) or local L-variable L0 – L1023. No fractional constants or mathematical expressions are permitted.

In older V1.*x* firmware, for two-dimensional data points, the point index values ***i*** and ***j*** can take any integer constant value from 0 to 65,533 (within the range declared by **Nx[*n*]**) or local L-variable L0 for ***i*** and L0 or L1 for ***j***. No fractional constants or mathematical expressions are permitted.

CompTable[*m*].Data[*k*][*j*][*i*]

Description: Compensation table three-dimensional entry

Range: Floating-point

Units: User-defined table units

Default: nan (not-a-number)

CompTable[*m*].Data[*k*][*j*][*i*] specifies the value of a specific entry (correction) of a three-dimensional table. Corrections are computed by interpolation between adjacent entries in all three dimensions. The correction is then multiplied by a scale factor before being written to the target register. Usually the scale factor is 1.0, so these entries are specified in units of the target register (position or torque).

CompTable[*m*].Data[0][0][0] specifies the correction when the motor specified by **Source[0]** is at the position specified by **X0[0],** the motor specified by **Source[1]** is at the position specified by **X0[1]**, and the motor specified by **Source[2]** is at the position specified by **X0[2]**.

CompTable[*m*].Data[0][0][1] specifies the correction when motor specified by **Source[0]** is at the position (**X0[0]** + **Dx[0]**/**Nx[0]**), the motor specified by **Source[1]** is at the position specified by **X0[1]**, and the motor specified by **Source[2]** is at the position specified by **X0[2]**.

CompTable[*m*].Data[0][1][0] specifies the correction when motor specified by **Source[0]** is at the position **X0[0]** and the motor specified by **Source[1]** is at the position specified by (**X0[1]** + **Dx[1]**/**Nx[1]**), and the motor specified by **Source[2]** is at the position specified by **X0[2]**.

CompTable[*m*].Data[1][0][0] specifies the correction when motor specified by **Source[0]** is at the position **X0[0]** and the motor specified by **Source[1]** is at the position specified by **X0[1]**, and the motor specified by **Source[2]** is at the position specified by(**X0[2]** + **Dx[2]**/**Nx[2]**).

If the table is not specified for rollover in the dimension of the motor specified by **Source[0]**, the boundary table entry **CompTable[*m*].Data[0][0][*Nx[0]*]** specifies the correction when motor specified by **Source[0]** is at the position (**X0[0]** + **Dx[0]**), the motor specified by **Source[1]** is at the position specified by **X0[1]**, and the motor specified by **Source[2]** is at the position specified by **X0[2]**. If the table is specified for rollover in this dimension, only the points **CompTable[*m*].Data[0][0][0]** through **CompTable[*m*].Data[0][0][*Nx[0]-1*]** are used in this line.

If the table is not specified for rollover in the dimension of the motor specified by **Source[1]**, the boundary table entry **CompTable[*m*].Data[0][*Nx[1]*][0]** specifies the correction when motor specified by **Source[1]** is at the position (**X0[1]** + **Dx[1]**), the motor specified by **Source[0]** is at the position specified by **X0[0]**, and the motor specified by **Source[2]** is at the position specified by **X0[2]**. If the table is specified for rollover in this dimension, only the points **CompTable[*m*].Data[0][0][0]** through **CompTable[*m*].Data[0][*Nx[1]-1*][0]** are used in this line.

If the table is not specified for rollover in the dimension of the motor specified by **Source[2]**, the boundary table entry **CompTable[*m*].Data[*Nx[1]*][0][0]** specifies the correction when motor specified by **Source[2]** is at the position (**X0[2]** + **Dx[2]**), the motor specified by **Source[0]** is at the position specified by **X0[0]**, and the motor specified by **Source[2]** is at the position specified by **X0[2]**. If the table is specified for rollover in this dimension, only the points **CompTable[*m*].Data[0][0][0]** through **CompTable[*m*].Data[*Nx[1]-1*][0][0]** are used in this line.

Starting in V2.0 firmware, released 1st quarter 2015, for three-dimensional data points, the point index value ***i*** can take any integer constant value from 0 to 64,511, the point index values ***j*** and ***k*** can take any integer constant value from 0 to 223 (within the range declared by **Nx[*n*]**) or local L-variables L0 to L31.

In older V1.*x* firmware, for three-dimensional data points, the point index value ***i*** can take any integer constant value from 0 to 65,533, the point index values ***j*** and ***k*** can take any integer constant value from 0 to 253 (within the range declared by **Nx[*n*]**) or local L-variable L0 for ***i***, L0 or L1 for ***j*** and L0, L1 or L2 for ***k***. No fractional constants or mathematical expressions are permitted.

CompTable[*m*].Dx[*n*]

Description: Compensation table dimension source position span

Range: Non-negative floating-point

Units: Motor units of the source

Default: 0.0

CompTable[*m*].Dx[*n*] specifies the span distance of the source motor in the table dimension denoted by the index of **X0[*n*]**. This index can take a value of 0, 1, or 2, specifying the 1st, 2nd, and 3rd dimensions of the table, respectively.

In a two-dimensional table, **Dx[0]** specifies the span distance in the dimension that varies with the second index of the data point, and **Dx[1]** specifies the span distance in the dimension that varies with the first index of the data point. **Dx[2]** is not used in this case.

In a three-dimensional table, **Dx[0]** specifies the span distance in the dimension that varies with the third index of the data point, **Dx[1]** specifies the span distance in the dimension that varies with the second index of the data point, and **Dx[2]** specifies the span distance in the dimension that varies with the first index of the data point.

This span distance is in units of the source motor. The minimum position used from the source motor for the dimension is specified by **X0[*n*]**. The maximum position used is **X0[*n*]** + **Dx[*n*]**. The spacing between table points in the dimension is **Dx[*n*]** / **Nx[*n*]**.

CompTable[*m*].MaxDac

Description: Compensation table maximum automatically set torque magnitude

Range: 0.0 .. 32,767.999

Units: 16-bit DAC equivalent

Default: 0.0

CompTable[*m*].MaxDac specifies the magnitude of the maximum torque *offset* value that will be applied to a **CompTable[*m*].Data[*i*]** element by the automatic "iterative learning control" (ILC) algorithm for the table. It does *not* limit what a user can directly write to one of these elements. It is only used if **CompTable[*m*].DacEnable** is set to 1 to enable the ILC algorithm.

The ILC algorithm works to minimize position errors that are repeated in multiple cycles of the table by adjusting the **Data[*i*]** values in each zone of the table to provide appropriate torque corrections for the zone.

CompTable[*m*].MaxDac is new in V2.1 firmware, released 1st quarter 2016.

CompTable[*m*].MinPosError

Description: Compensation table minimum following error for automatic torque adjustment

Range: Non-negative floating-point

Units: Target motor position units

Default: 0.0

CompTable[*m*].MinPosError specifies the magnitude of the minimum following error of the target motor for which the "iterative learning control" (ILC) algorithm will automatically adjust the value of a **CompTable[*m*].Data[*i*]** element to try to reduce the error in subsequent table cycles. It is only used if **CompTable[*m*].DacEnable** is set to 1to enable the ILC algorithm.

The ILC algorithm works to minimize position errors that are repeated in multiple cycles of the table by adjusting the **Data[*i*]** values in each zone of the table to provide appropriate torque corrections for the zone.

CompTable[*m*].MinPosError is new in V2.1 firmware, released 1st quarter 2016.

CompTable[*m*].Nx[*n*]

Description: Compensation table dimension number of zones

Range: Non-negative integers

Units: Table zones between points

Default: 0

CompTable[*m*].Nx[*n*] specifies the number of zones in the table dimension denoted by the index of **Nx[*n*]**. This index can take a value of 0, 1, or 2, specifying the 1st, 2nd, and 3rd dimensions of the table, respectively. If the value of the element specifies a number of zones greater than 0, the dimension is "active", but if it is 0, the dimension is not active. (A zone is the space between two adjacent table points in the dimension.) For example, in a one-dimensional table, **CompTable[*m*].Nx[0]** will be greater than 0, but **CompTable[*m*].Nx[1]** and **CompTable[*m*].Nx[2]** will be equal to 0.

In a two-dimensional table, **Nx[0]** specifies the number of zones in the dimension that varies with the second index of the data point, and **Nx[1]** specifies the number of zones in the dimension that varies with the first index of the data point.

In a three-dimensional table, **Nx[0]** specifies the number of zones in the dimension that varies with the third index of the data point, **Nx[1]** specifies the number of zones in the dimension that varies with the second index of the data point, and **Nx[2]** specifies the number of zones in the dimension that varies with the first index of the data point.

If the table is specified *not* to roll over in a particular dimension as determined by **CompTable[*m*].Ctrl**, data points should be entered in that dimension with indices from 0 through the number of zones specified for that dimension, so there is one more data point than zone in that dimension.

If the table *is* specified to roll over in a particular dimension as determined by **CompTable[*m*].Ctrl**, data points should be entered in that dimension with indices from 0 through one less than the number of zones specified for that dimension, so the number of data points in that dimension is the same as the number of zones. An additional point in that dimension (with index equal to the number of zones) may be entered without error, but it will not be used, as the corrections in the adjacent zone will be calculated interpolating between the previous point and the point with an index of 0 in that dimension.

CompTable[*m*].OutCtrl

Description: Compensation table correction output control

Range: 0 .. 255

Units: Bit field

Default: 0

CompTable[*m*].OutCtrl specifies how the compensation table treats its target registers. It is an 8-bit value with each bit ***q*** (with value of 2^{q}) controlling how the register specified by **CompTable[*m*].Target[*q*]** is treated. If the bit is set to its default value of 0, the correction value overwrites the value found in that register. If the bit is set to 1, the correction value is added to the value found in that register, and the resulting sum is written to the register.

The addition option permits multiple corrections to apply to a single target register. One common use for this feature is the application of "coarse" and "fine" tables to a single motor.

Note that the first (lowest-numbered) table that writes to a given register in a servo cycle must have its **OutCtrl bit** value set to 0 to overwrite the previous cycle's correction. Any subsequent (higher-numbered) tables that write to the same register in the servo cycle must have their **OutCtrl** bit values set to 1 so their corrections are added to the existing 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.

CompTable[*m*].Sf[*q*]

Description: Compensation table correction target scale factor

Range: Floating-point

Units: Target-register units per table correction unit

Default: 1.0

CompTable[*m*].Sf[*q*] specifies the scale factor that multiplies the correction computed by the table before it is written to the target register. As there can be multiple target registers for a table, each target register can have its own scale factor. The scale factor of a given index (0 to 7) affects the target register of the same index (e.g. **Sf[2]** affects the register of **Target[2]**).

Most users will leave the **CompTable[*m*].Sf[*n*]** elements at the default value of 1.0 so that the table entries are in the units of the target register. This is true for tables whose target registers are position registers, so the table units are equal to the position units of the target motor, and for tables whose target registers are torque registers, so the table units are those of 16-bit signed outputs, the same as the servo command output units.

CompTable[*m*].Source[*n*]

Description: Compensation table dimension source motor number

Range: 0 .. 255

Units: Motor Number

Default: 0

CompTable[*m*].Source[*n*] specifies the number of the motor whose position is used as the "source" of the location in the table dimension denoted by the index of **Source[*n*]**. The index for **Source[*n*]** can take a value of 0, 1, or 2. In a one-dimensional table, only **Source[0]** need be specified; in a 2D table, and **Source[1]** must also be specified; in a 3D table, **Source [2]** must be specified as well. A given source will only be used in the table if the setup element **CompTable[*m*].Nx[*n*]** for the same final index is greater than 0.

In a two-dimensional table, **Source[0]** specifies the source motor whose corrections vary with the second index of the data point, and **Source[1]** specifies the source motor whose corrections vary with the first index of the data point. **Source[2]** is not used in this case.

In a three-dimensional table, **Source[0]** specifies the source motor whose corrections vary with the third index of the data point, **Source[1]** specifies the source motor whose corrections vary with the second index of the data point, and **Source[2]** specifies the source motor whose corrections vary with the first index of the data point.

CompTable[*m*].SourceCtrl

Description: Compensation table source register control

Range: 0 .. 7

Units: Bit field

Default: 0

CompTable[*m*].SourceCtrl specifies which position value is used from the source motor for each dimension of the compensation table. It is a 3-bit value, with bit *n* determining which position is used for the motor selected by **CompTable[*m*].Source[*n*]**. If the bit is set to the default value of 0, the value of the net desired position (**Motor[*x*].DesPos**) is used. If the bit is set to 1, the value of the uncorrected actual position (**Motor[*x*].Pos**) is used.

Use of desired position eliminates possible interactions of corrections with servo-loop dynamics, and reduces the level of noise introduced into the corrections. Generally, this permits the use of higher servo-loop gains. Use of actual position provides a more immediate and precise correction, and can provide higher accuracy if servo interactions and measurement noise are low.

Bit *n* of **CompTable[*m*].SourceCtrl** is only used if **CompTable[*m*].Nx[*n*]** is greater than zero, "activating" that dimension of the table. So for the most common one-dimensional tables, with only **Nx[0]** greater than zero, only bit 0 of **SourceCtrl** is used.

CompTable[*m*].Target[*q*]

Description: Compensation table correction target address

Range: Selected Power PMAC addresses

Units: Power PMAC addresses

Default: 0 (inactive target)

CompTable[*m*].Target[*q*] specifies the address of the register to which the correction computed by the table will be written (the "target"). A single table can have multiple target registers. The index of **Target[*q*]** can take a value from 0 to 7, so there can be up to 8 targets for a table. If **CompTable[*m*].Target[*q*]** is set to 0, this target is not active, and any targets with a greater index value ***q*** are also not active, even if they are addressing a register.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

For a position compensation table that is to correct for mechanical or sensor errors, **CompTable[*m*].Target[*q*]** is set to the address of the target motor's outer-loop compensation register (**CompPos**), and often the inner-loop compensation register (**CompPos2**), where it can help reduce the ripple in the velocity loop from measurement errors. The values in these registers are added to the raw measured values in **Motor[*x*].Pos** and **Motor[*x*].Pos2**, respectively, to get the net corrected actual positions **Motor[*x*].ActPos** and **Motor[*x*].ActPos2**, respectively.

For example:

CompTable[0].Target[0]=Motor[1].CompPos.a

CompTable[0].Target[1]=Motor[1].CompPos2.a

For a position compensation table that is to create motion as for an electronic cam, **CompTable[*m*].Target[*q*]** is set to the address of the target motor's desired position compensation register (**CompDesPos**), where it can superimpo. The values in this register is added to the mathematically calculated trajectory value and the master position value to get the net desired position **Motor[*x*].DesPos**.

For example:

CompTable[0].Target[0]=Motor[4].CompDesPos.a

For a torque compensation table, **CompTable[*m*].Target[*q*]** is set to the address of the target motor's servo-output compensation register **Motor[*x*].CompDac**.

For example:

CompTable[1].Target[0]=Motor[1].CompDac.a

For a backlash compensation table, **CompTable[*m*].Target[*q*]** is set to the address of the target motor's backlash compensation register **Motor[*x*].BlCompSize**.

For example:

CompTable[2].Target[0]=Motor[1].BlCompSize.a

CompTable[*m*].X0[*n*]

Description: Compensation table dimension starting source position

Range: Floating-point

Units: Motor units of the source

Default: 0.0

CompTable[*m*].X0[*n*] specifies the starting (minimum) position of the source motor in the table dimension denoted by the index of **X0[*n*]**. This index can take a value of 0, 1, or 2, specifying the 1st, 2nd, and 3rd dimensions of the table, respectively.

In a two-dimensional table, **X0[0]** specifies the minimum source position in the dimension that varies with the second index of the data point, and **X0[1]** specifies the minimum source position in the dimension that varies with the first index of the data point. **X0[2]** is not used in this case.

In a three-dimensional table, **X0[0]** specifies the minimum source position in the dimension that varies with the third index of the data point, **X0[1]** specifies the minimum source position in the dimension that varies with the second index of the data point, and **X0[2]** specifies the minimum source position in the dimension that varies with the first index of the data point.

This position is in units of the source motor, relative to the motor's zero position.

Coord[*x*]. Saved Data Structure Elements

The **Coord[*x*].** data structure provides all of the software information for the specified coordinate system. This section describes those elements in the data structure that are saved to non-volatile memory.

Note: The coordinate-system index value in the square brackets may be specified by an integer constant in the range of valid coordinate-system numbers for the Power PMAC system, or a "local" L-variable for the program or communications thread (L0 – L1007 can be used). No other method of specifying the index value may be used. Index values can go from 0 to **Sys.MaxCoords** - 1, with **Sys.MaxCoords** having a maximum value of 128.

Coord[*x*].AbortAllMode

Description: Coordinate system action on "abort all" input

Range: 0 ..3

Units: Enumeration

Default: 0

Coord[*x*].AbortAllMode specifies the action this coordinate system takes when the Power PMAC receives an "abort all" input as specified by global saved setup elements **Sys.pAbortAll**, **Sys.AbortAllBit**, and **Sys.AbortAllLimit**.

If **Coord[*x*].AbortAllMode** is set to the default value of 0, all motors in the coordinate system are brought to a closed-loop controlled stop as if an **`abort`** command had been issued to the coordinate system. This provides a "Category 2" controlled safe stop under the IEC-61800-5-2 machine safety standard. If a "Category 1" safe stop under this standard – a controlled stop followed by a software-free disabling – is desired, 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 1, all motors in the coordinate system are immediately disabled (killed) as if a **`disable`** command had been issued to the coordinate system. There is no delay for brake engagement even if a "delayed disable" would provide this.

If **Coord[*x*].AbortAllMode** is set to 2, all motors in the coordinate system are first brought to a closed-loop controlled stop as if an **`abort`** command had been issued to the coordinate system. Then as each motor reaches a desired velocity of zero, it executes a "delayed kill", as if a **`dkill`** command had been issued to it, with an immediate engagement of the brake (if one has been specified with **Motor[*x*].pBrakeOut**) followed by a disabling of the motor after the interval specified by **Motor[*x*].BrakeOnDelay**.

If **Coord[*x*].AbortAllMode** is set to 3, the coordinate system is not affected by the "abort all" input.

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 useful to implement a Category 1 stop along with a time-delay relay that removes power from a key circuit (e.g. Safe Torque Off) after the controlled stop without software intervention.

Coord[*x*].AbortTimeBase

Description: Minimum time base value used in abort deceleration

Range: Non-negative floating-point

Units: Milliseconds

Default: 0.0

Coord[*x*].AbortTimeBase specifies the minimum time base value used by the coordinate system in executing the motor decelerations in an "abort" sequence, whether the sequence is due to a software or hardware command, or from an error that causes a controlled stop. The intent of this parameter is to be able to command quick but properly controlled abort decelerations regardless of the time base value at the start of the sequence.

When both the instantaneous and target time base values are greater than **AbortTimeBase** but not greater than 100% (**Coord[*x*].AbortTimeBase** < **Coord[*x*].TimeBase**, **Coord[*x*].DesTimeBase** <= **Sys.ServoPeriod**) at the start of the abort sequence, the time base value is not changed, and the abort deceleration profiles will use the present time base values. For example in this case, if the time base were at 50%, the deceleration profiles would take twice the time that they would at 100%, with the rates of deceleration at ¼ of what they would be at 100%.

When either the instantaneous or target time base value is outside of the range between **AbortTimeBase** and +100% (**Coord[*x*].TimeBase** or **Coord[*x*].DesTimeBase** < **Coord[*x*].AbortTimeBase**, **Coord[*x*].TimeBase** or **Coord[*x*].DesTimeBase** > **Sys.ServoPeriod**), including negative time base values, at the start of the abort sequence, the time base value is immediately set to +100%, changing **Coord[*x*].pDesTimeBase** to its default value of **Coord[*x*].DesTimeBase.a** if it is set to a different address so it will use the "internal" time base, the present velocity and acceleration values for all motors in the coordinate system are rescaled for continuity, then deceleration profiles computed starting from these rescaled values and executed according to the settings of **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**.

Note that if the time base value is exactly 0.0 at the start of the abort sequence (as when the coordinate system is in feed-hold mode), the present velocity values are rescaled to 0.0, so there is no need for any abort deceleration profile. An abort command is often used when a motion program is already suspended at 0% time base if it is decided not to resume the program and instead be able to load and/or execute another program.

In this situation, the resulting time base source and value can differ depending on whether **Coord[*x*].AbortTimeBase** is equal to 0.0 or greater than 0.0. If it is equal to 0.0, the instantaneous time base value is immediately set to the desired (non-feedhold) time base value, and the time base source register is not changed. If the 0% time base is not due to a feed hold, it will remain at 0%. This operation is compatible with older versions of the firmware that did not have this parameter.

However, if **Coord[*x*].AbortTimeBase** is greater than 0.0 in this case, the time base value is set to the 100% value, and **Coord[*x*].pDesTimeBase** is set to its default value of **Coord[*x*].DesTimeBase.a** to use its internal time base.

The following table summarizes the time base actions for different starting time bases when **Coord[*x*].AbortTimeBase** is greater than 0.0. MATB (minimum abort time base) is equal to (100% * **Coord[*x*].AbortTimeBase** / **Sys.ServoPeriod**).

Starting % Value	Starting TimeBase, DesTimeBase Values	Resulting % Value, Source	Deceleration Profile Executed?
< 0.0%	Either < 0.0	+100%, Internal	Yes
= 0.0%	**TimeBase** = 0.0	+100.0%, Internal	No
> 0.0%, < MATB%	Both > 0.0 Either < **AbortTimeBase**	+100.0%, Internal	Yes
>= MATB%, <= 100.0%	Both >= **AbortTimeBase** Both <= **Sys.ServoPeriod**	No change in value or source	Yes
> 100%	Either > **Sys.ServoPeriod**	+100%, Internal	Yes

If you wish the abort profiles always to be executed at 100% time base (i.e. always at the programmed rates), **Coord[*x*].AbortTimeBase** should be set equal to **Sys.ServoPeriod**.

The following table summarizes the time base actions for different starting time bases when **Coord[*x*].AbortTimeBase** is equal to 0.0.

Starting % Value	Starting TimeBase, DesTimeBase Values	Resulting % Value, Source	Deceleration Profile Executed?
< 0.0%	Either < 0.0	+100%, Internal	Yes
= 0.0%	**TimeBase** = 0.0	(= **DesTimeBase**), No source change	No
> 0.0%, <= 100%	Both > 0.0 Both <= **Sys.ServoPeriod**	No change in value or source	Yes
> 100%	Either > **Sys.ServoPeriod**	+100%, Internal	Yes

The cases when starting time base is less than 0% or greater than 100% are not compatible with older firmware versions, but will create better controlled deceleration profiles.

Example

In a system with a 4 kHz servo update frequency, **Sys.ServoPeriod** would be set to 0.250 (milliseconds). If it is desired that at any time base value below 60%, the abort profile be executed at 100%, **Coord[*x*].AbortTimeBase** should be set to 0.250 * 0.6 = 0.150.

Coord[x].AddedDwellTime

Description: Blend-disable dwell time

Range: Non-negative integers

Units: Real-time interrupt periods

Default: 0

Legacy I-variable alias: I*x*82

Coord[*x*].AddedDwellTime specifies the dwell time that is automatically inserted between programmed moves when blending is disabled. This dwell time is inserted after programmed moves that are never blended (**`rapid`**-mode moves, programmed homing-search moves), or after moves that can be blended, but for which blending has been disabled by setting **Coord[*x*].NoBlend** to 1, or because the corner is sharper than the angles specified by **Coord[*x*].CornerBlendBp** and **Coord[*x*].CornerDwellBp** (If the angle is sharper than that specified by **Coord[*x*].CornerBlendBp** but not sharper than that specified by **Coord[*x*].CornerDwellBp**, blending will be disabled at the corner, but no dwell will be added.) The units of this dwell are in real-time interrupt periods (**Sys.RtIntPeriod** + 1 servo cycles), not milliseconds as in a directly programmed dwell.

If **Coord[*x*].InPosTimeout** is 0, this dwell time is inserted immediately after the end of commanded execution of the previous move. If **Coord[*x*].InPosTimeout** is greater than 0, this dwell time is inserted after all axes in the coordinate system have been verified to be "in position".

Coord[x].AltFeedMode

Description: Calculation mode for non-feedrate axes

Range: 0 .. 1

Units: none

Default: 0

Coord[*x*].AltFeedMode specifies the mode for calculations of potential move times for "non-feedrate" axes in feedrate-specified moves (**`linear`** or **`circle`** mode). In these moves, Power PMAC first computes the potential move time for the "feedrate" axes (as specified by the **`frax`** command; X, Y, and Z by default) by dividing the vector move distance of these axes by the commanded feedrate as specified by the **`F`** command.

It then compares this time to the potential time for any non-feedrate axes. If **Coord[*x*].AltFeedMode** is at the default value of 0, a potential move time is computed individually for each non-feedrate axis by dividing its commanded move distance by **Coord[*x*].AltFeedRate**. The largest of these times is then compared to the time for the vector feedrate axes, and the greater of these two times is used for the move.

If **Coord[*x*].AltFeedMode** is set to 1, the potential move time for the non-feedrate axes is computed by considering these axes as another Cartesian vector set, with a single collective vector distance computed as the square root of the sum of the squares of the individual axis distances, and this distance divided by **Coord[*x*].AltFeedRate**. This time is then compared to the time for the vector feedrate axes, and the greater of these two times is used for the move.

The default setting of 0 for **Coord[*x*].AltFeedMode** is generally used when the non-feedrate axes are rotary axes (e.g. A, B, and C). In this case, computing a vector distance for these axes does not make geometric sense.

The setting of 1 for **Coord[*x*].AltFeedMode** is generally used when the non-feedrate axes comprise a second Cartesian set (e.g. XX, YY, and ZZ), so computing a vector distance for this axis set does make geometric sense. It is also useful for a "dry run" mode of an XYZ system, where **`nofrax`** is declared so the program **`F`** value is ignored, and the (usually higher) value of **Coord[*x*].AltFeedRate** is used instead as a vector feedrate.

Coord[*x*].AltFeedRate

Description: Programmed speed for non-vector axes

Range: Non-negative floating-point

Units: User axis velocity units

Default: 1.0

Legacy I-variable alias: I*sx*86

Coord[*x*].AltFeedRate controls the speed of motion for a feedrate-specified blended move (**`linear`** or **`circle`** mode) when the motion of "non-feedrate" axes is predominant. "Feedrate", or "vector-feedrate" axes are those specified by the **`frax`** command; X, Y, and Z are the feedrate axes by default.

If **Coord[*x*].AltFeedRate** is greater than 0.0, Power PMAC compares the move time for the vector feedrate axes, computed as the vector distance of the feedrate axes divided by the specified feedrate (the **`F`** value in the program or **Coord[*x*].Tm**), to the move time for the non-feedrate axes, computed as the longest distance for these axes divided by **Coord[*x*].AltFeedRate**. It then uses the longer of these two times as the move time for all axes, feedrate and non-feedrate.

If **Coord[*x*].AltFeedRate** is 0.0, and Power PMAC sees a feedrate-specified move in which the vector distance is zero (i.e. no motion of the vector feedrate axes), the move of the non-feedrate axes will use the feedrate value specified for the vector axes. (In firmware versions older than 2.1 – released 1st quarter 2016 – this move would have a specified move time of zero, and so would be controlled by the acceleration times.)

Coord[*x*].AltFeedRate has two main uses. First, it automatically controls the motion of "non-feedrate" axes when they are commanded alone on a line in feedrate mode. Typically these are rotary axes in a combined linear/rotary system where only the linear axes are vector feedrate axes.

Second, it permits a fast "dry-run" mode in which the programmed feedrates are ignored. If no axes in the coordinate system are vector feedrate axes (implemented with the **nofrax** command), then **Coord[*x*].AltFeedRate** will be used for all moves, regardless of the **F** values in the program.

The exact method of calculation using **Coord[*x*].AltFeedRate** is dependent on the setting of **Coord[*x*].AltFeedMode**.

Example

```
Coord[1].FeedTime=1000          // Speeds are specified as per-second
Coord[1].AltFeedRate=5          // Alternate feedrate of 5 user units per second

inc                             // Moves specified by distance
frax(X,Y,Z)                     // X, Y, and Z are vector feedrate axes
X20 C5 F10                      // Vector move time = 20 units / 10 (units/sec) = 2 sec
                                // Non-vector move time = 5 units / 5 (units/sec) = 1sec
                                // Use 2 sec (vector feedrate controls)
X10 C20                         // Vector move time = 10 units / 10 (units/sec) = 1 sec
                                // Non-vector move time = 20 units / 5 (units/sec) = 4 sec
                                // Use 4 sec (alternate feedrate controls)
C20                             // Move time = 20 units / 5 (units/sec) = 4 sec
```

Coord[*x*].AutoTxyzScale

Description: Automatic transformation matrix rescaling control

Range: 0 .. 1

Units: Boolean

Default: 0

Coord[*x*].AutoTxyzScale specifies whether Power PMAC will automatically compute the transformation-matrix rescaling factor **Coord[*x*].TxyzScale** based on the selected transformation matrix scaling of the XYZ Cartesian space in the coordinate system. If it is set to the default value of 0, no automatic computation will be done.

If **Coord[*x*].AutoTxyzScale** is set to 1, Power PMAC will automatically compute the value of status element **Coord[*x*].TxyzScale** based on the 3x3 XYZ minor matrix of the selected transformation matrix. If no transformation matrix has been selected for the coordinate system with a program **tsel*{data}*** command, this calculation will not be done.

The value of **TxyzScale** is only computed in this automatic mode when a program **tsel*{data}*** command is actually executed. Even if a given transformation matrix has already been selected, if the relevant values in that matrix are changed, **TxyzScale** will not change until that matrix is selected again with another **tsel*{data}*** command.

Technically speaking, **TxyzScale** is computed as the cube root of the determinant of the 3x3 XYZ minor matrix of the selected transformation matrix. This will maintain the feedrate and tool radius in the "base" (untransformed) units of the X, Y, and Z axes even when the programmed units of these axes have been rescaled with the selected transformation matrix. In use, the commanded values of feedrate and tool radius are divided by **TxyzScale** before use in a move.

For these calculations to be appropriate, the transformed X, Y, and Z axes must all have the same scaling. Even if only 1 or 2 of these axes is actually used (e.g. a 2D XY system), the additional axis or axes must be rescaled as well.

Coord[*x*].CCAddedArcBp

Description: Cutter compensation outside-corner added-arc break point

Range: -1.0 .. 1.0 (floating-point)

Units: Angle cosine

Default: 0.0 (90° equivalent angle)

Legacy I-variable alias: I*sx*99

Coord[*x*].CCAddedArcBp specifies the threshold in the coordinate system between outside corner angles for which an extra arc move is added in 2D cutter compensation, and those for which the incoming and outgoing moves are directly combined at the compensated intersection point.

Coord[*x*].CCAddedArcBp is expressed as the "equivalent" cosine of the change in directed angle between the incoming and outgoing **`linear`**-mode moves that would produce the same ratio of cutter radius to distance between uncompensated and compensated intersection points. (The change in directed angle is equal to 180° minus the included angle of the corner.)

As such, it can take a value between -1.0 and +1.0. If the two moves have the same directed angle at the move boundary (i.e. they are moving in the same direction), the change in directed angle is 0°, and the cosine is 1.0. As the change in directed angle increases, the corner gets sharper, and the cosine of the change in directed angle decreases. For a total reversal, the change in directed angle is 180°, and the cosine is -1.0. The change in directed angle is evaluated in the plane defined by the **`normal`** command (default is the XY-plane); if the corner also involves axis movement perpendicular to this plane, it is the projection of movement into this plane that matters.

For the corner formed by two **`linear`**-mode moves, **Coord[*x*].CCAddedArcBp** does express the cosine of the threshold in change in directed angles. For **`circle`**-mode moves, the threshold angle will be different depending on whether the circular moves are "concave in" or "concave out" and what the circle radii are, as Power PMAC is choosing based on the ratio of cutter radius to distance between uncompensated and compensated intersection points. With **`circle`**-mode moves, it is possible that there is no compensated intersection point. In this case, an arc move is always added to cover the outside corner.

If the (equivalent) cosine of the change in directed angle at a corner is less than **Coord[*x*].CCAddedArcBp** (a large change in directed angle; a sharp corner), Power PMAC will add an arc move automatically with a radius equal to the cutter radius to join the incoming and outgoing moves. This prevents the cutter from moving too far out when going around the outside of a sharp corner. Note that the minimum time for an added arc is the declared acceleration time

as set by **Ta** and **Ts**. If the added arc is very short, as for very slight angle changes, this could cause an undesired slowing.

If the (equivalent) cosine of the change in directed angle at a corner is greater than or equal to **Coord[*x*].CCAddedArcBp** (a small change in directed angle; a gradual corner), Power PMAC will directly join the incoming and outgoing moves at the compensated intersection point. If the moves are to be blended together, it will use its normal blending algorithms at this point.

The operation of **Coord[*x*].CCAddedArcBp** is mostly independent of the operation of the similar function of **Coord[*x*].CornerBlendBp**, which controls for corners whether the incoming and outgoing moves will be blended together based on the change in directed angle. **Coord[*x*].CornerBlendBp** works regardless of whether cutter-radius compensation is active or not, or whether the corner is an inside or outside corner when cutter-radius compensation is active. However, if this is an outside compensated corner with an added arc, Power PMAC computes the "corner" angles between the added arc move and the incoming and outgoing moves. These angles are almost always 0°, so would always be blended together.

Example

If it is desired that an arc move be added to a compensated outside corner if the change in directed angle is greater than 60° (included angle less than 120°), then **Coord[*x*].CCAddedArcBp** should be set to 0.5, because cos $\Delta\theta$ = cos 60° = 0.5.

Coord[*x*].CCCtrl

Description: 2D cutter radius compensation control bits

Range: 0 .. 15

Units: Bit field

Default: 0

Legacy I-variable alias: I*sx*84

Coord[*x*].CCCtrl specifies several modal aspects of the 2D cutter radius compensation algorithm. It consists of several control bits.

Bit 0 (value 1) of **Coord[*x*].CCCtrl** controls where the programmed movement stops when blending is disabled on an outside corner in 2D cutter-radius compensation with an added arc at the corner. If bit 0 of **Coord[*x*].CCCtrl** is 0, the programmed movement will stop at the end of the added arc; if bit 0 **Coord[*x*].CCCtrl** is 1, the programmed movement will stop at the beginning of the added arc.

Bit 0 of **Coord[*x*].CCCtrl** also controls where any buffered synchronous variable assignment commands are executed at the corner, whether blending is enabled or not. If it is set to 0, any buffered assignments are executed at the beginning of the outgoing move after end of the added arc; if it is set to 1, any buffered assignments are executed at the beginning of the buffered arc. In addition, the move-target positions reported in response to the on-line **`t`** command and the buffered **`tread`** command are those where the synchronous assigments would be executed.

If blending is disabled due to single-step operation (**s**), program termination (**q** or **/**), or to a setting of **Coord[*x*].NoBlend** to 1, programmed movement will always stop at the end of the added arc. If there are out-of-plane moves between the two moves that form the corner in the plane of compensation, they are always executed at the end of the added arc.

Bit 1 (value 2) of **Coord[*x*].CCCtrl** specifies the action taken when interference is detected in the 2D cutter radius compensation algorithm. Interference conditions, which are detected by the reversal of the direction of the compensated move compared to the uncompensated move, or by intersection of the compensated path with the compensated path of another move in the pre-computation buffer, usually means that an "overcut" will occur.

If bit 1 of **Coord[*x*].CCCtrl** is 0, program operation will stop in an error condition when interference is detected. If sufficient pre-computation has been specified with **Coord[*x*].CCDistance**, the stop will occur before the overcut occurs. This mode of operation is typically used when interference and overcut imply a programming error.

If bit 1 of **Coord[*x*].CCCtrl** is 1, program operation will attempt to "correct" the interference case by eliminating the compensated path between the first time the intersection point is reached and the second time, then directly combining the ingoing and outgoing paths from the intersection point. If it cannot correct the interference, it will stop. This mode of operation is typically used during "rough cut" passes with a large cutter that cannot get into small features of the part.

Bit 2 (value 4) of **Coord[*x*].CCCtrl** specifies whether the interference checking is done in "operational" mode or "test" mode. If bit 2 is set to 0, "operational" mode is specified, and the algorithm will either stop on detecting interference (bit 1 = 0) or "correct" the interference condition. This mode is typically used for actual production.

If bit 2 of **Coord[*x*].CCCtrl** is set to 1, "test" mode is specified, and the algorithm will either bypass any interference checking (bit 1 = 0), permitting overcut to occur, or correct if possible (bit 1 = 1), but continue and permit overcut to occur if it cannot. It will only stop if it cannot compute any solution. This mode is typically used to debug part programs.

Bit 3 (value 8) of **Coord[*x*].CCCtrl** specifies whether the vector feedrate for a circle-mode move with 2D cutter compensation active is that of the tool center or the tool edge along the programmed path. If bit 3 is set to 0, the tool center will move at the programmed feedrate, so the tool edge along the programmed path will be slower if compensation is to the outside of the arc, or faster if compensation is to the inside of the arc.

If bit 3 of **Coord[*x*].CCCtrl** is set to 1, the tool edge along the programmed path will move at the programmed feedrate, so the tool center will be faster if compensation is to the outside of the arc, or slower if compensation is to the inside of the arc.

A value of 1 for bit 3 of **Coord[*x*].CCCtrl** is new in V2.1 firmware, released 1st quarter 2016.

Coord[*x*].CCCtrl constitutes bits 18 – 21 of the full-word element **Coord[*x*].Control[0]**.

Coord[x].CCDistance

Description: Cutter compensation in-plane lookahead distance

Range: 0 .. 255

Units: Programmed moves

Default: 0

Coord[*x*].CCDistance specifies the number of "in-plane" moves Power PMAC will pre-compute when 2D tool (cutter) radius compensation is active before finalizing the computation of the present move. Power PMAC must always pre-compute at least 2 in-plane moves to find the next 2 compensated intersection points. The first point constitutes the end of present compensated move; the second checks for a direction reversal in the next compensated move, which means the present move would cause an overcut due to interference. If **Coord[*x*].CCDistance** is less than 2, Power PMAC will still check 2 moves ahead.

If the user desires the option of being able to discard one or more "reversed" compensated moves in order to continue operation when interference is detected, **Coord[*x*].CCDistance** must be large enough to cover these moves as well. That is, **Coord[*x*].CCDistance** must be set to at least 2 plus the number of in-plane moves that could be discarded.

Coord[*x*].CCSize, which defines the size of the buffer that stores these pre-computed moves, must be set to a value at least as large as **Coord[*x*].CCDistance**. (It must be sized large enough to contain any "out-of-plane" and "no-motion" moves as well.)

In use, the pre-computation of this length is almost completely invisible to the user. When introducing compensation, Power PMAC will automatically fill the buffer by this number of in-plane moves before executing the lead-in move. During a compensated move sequence, Power PMAC will automatically be keeping the buffer full to this many in-plane moves, working ahead of the executing move. When compensation is turned off, Power PMAC will work through the moves remaining in the buffer, allowing it to empty automatically.

In single-step mode, Power PMAC will calculate enough programmed moves to keep the buffer full to the required extent while executing a single move from the buffer.

With a longer pre-computation length, standard variable value-assignment commands will occur further ahead of the execution of adjacent moves in the program than with a shorter distance. The execution of synchronous variable value-assignment commands will be delayed until the start of actual execution of the next commanded move in the program, regardless of the length of lookahead.

Coord[x].CCSize

Description: Cutter compensation move buffer length

Range: 0 .. 255

Units: Programmed moves

Default: 0

Coord[*x*].CCSize specifies the length of the buffer for programmed moves when 2D tool (cutter) radius compensation is active. It must be set at least to 2 to permit 2D compensation to operate.

Moves must be buffered in 2D cutter compensation for several reasons. First, in order to compute the compensated endpoint for a move, the starting direction of the next move must be known in order to compute the compensated intersection point. Second, in order to catch basic overcut conditions in time, the next intersection point must also be known (because direction reversal of the compensated move indicates interference), which means the following move must be computed as well. If it is desired to be able to discard one or more reversed moves when an interference condition is detected, one or more additional moves must be added to the buffer to keep it full enough of actually executing moves..

Finally, if there is the possibility of moves with no component of motion in the plane of compensation (moves perpendicular to the plane or "no-motion" moves such as dwells and delays), these must be buffered so that Power PMAC can calculate far enough ahead to compute the next two (or more) "in-plane" moves after these.

Therefore, the buffer must be sized large enough to contain enough "in-plane" moves to calculate compensated intersections and detect basic interference conditions (so a minimum of 2), plus enough to be able to discard moves that cause interference, and any moves with no motion in the plane of compensation.

For example, if it is possible to have a sequence during cutter compensation of {*dwell*, *cutter up*, *dwell*, *cutter down*, *dwell*} before resuming in-plane motion, there could be 5 moves with no motion in the plane of compensation between two compensated moves, and **Coord[*x*].CCSize** must be set at least to 7 for proper compensation.

Power PMAC will reserve (880 * **CCSize**) bytes of data for the coordinate system's cutter compensation buffer. This buffer is part of the default 16 MByte space allotted for all of the coordinate system buffers – cutter compensation, target position, and dynamic lookahead.

The cutter compensation buffer is independent of the lookahead buffer that limits dynamic quantities such as velocity and acceleration. Either buffer can be used with or without the other.

Coord[*x*].CCSize does not specify how many pre-computed moves are actually buffered; that is specified by the related element **Coord[*x*].CCDistance**. The defined buffer size must be large enough to hold all of the pre-computed moves.

Coord[*x*].Control[*i*]

Description: Coordinate system full-word element(s) for enabling elements

Range: $00000000 .. $FFFFFFFF

Units: Bit field

Default: $00000000

Coord[*x*].Control[*i*] is the array of full-word elements that comprises the setup elements for enabling the basic functionality of the coordinate system. (Presently it only consists of **Control[0]**, but it may be expanded in the future to more elements.)

Coord[*x*].Control[0] contains the following partial-word elements:

Component	Bits	Functionality
--	31	(*reserved for future use*)
NoBlend	30	Move blend disable control
SoftLimitStopDis	29	Action on soft limit control
RapidVelCtrl	28	Rapid-move move velocity mode
StepMode	27 – 24	Motion program single-step mode control
SegLinToPvt	23 – 22	Linear moves executed as PVT enable
CCCtrl	21 – 18	Cutter compensation control
Ndisplay	17	Synchronous line label display control
HomeRequired	16	Homing before motion program required
--	15 – 08	(*reserved for future use*)
GoBack	07 – 00	Motion program jumps back before blend disabled

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even though the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Coord[*x*].CornerAccel

Description: Cornering acceleration for blended moves

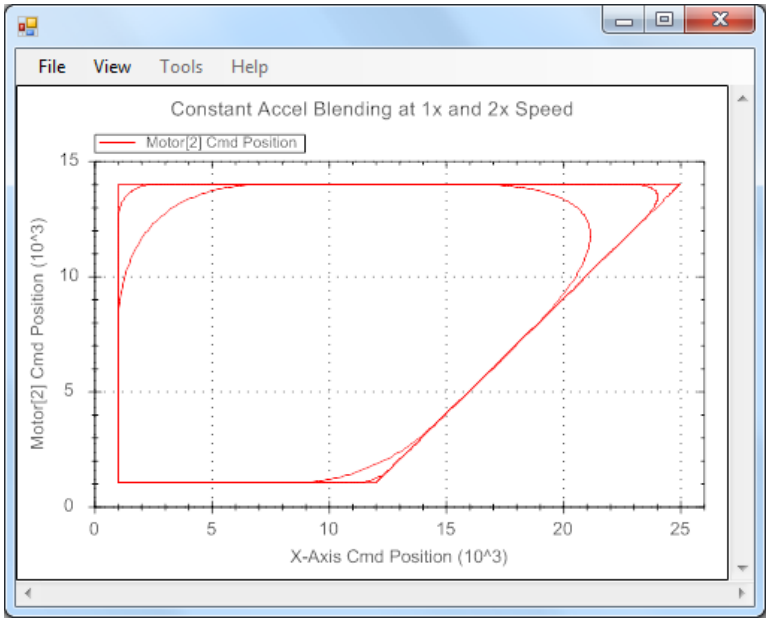
Range: Non-negative floating-point

Units: Axis units / msec^2

Default: 0.0 (disabled)

Coord[*x*].CornerAccel, if set to a positive value, specifies the vector cornering acceleration that Power PMAC will use as it blends moves together. Based on the vector speed and corner angle, Power PMAC will compute the blending time that provides this acceleration. This blending time, along with the vector speed, determines the corner "size". The most common use of this parameter is to permit execution of the smallest possible corner blend that can be executed using the specified acceleration.

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

Note that if **Coord[*x*].CornerRadius** is set to a positive value, **Coord[*x*].CornerAccel** will not be used – corners will be defined by size, not acceleration. If blending is disabled at the corner for any reason, including a corner sharper than the threshold defined by **Coord[*x*].CornerBlendBp**, then **Coord[*x*].CornerAccel** will not be used at the corner.

The corner angle is computed in the plane specified by the **`normal`** vector. If either move is a **`circle`**-mode move, the direction of the move immediately at the unblended corner is used to compute the corner angle. If the motion at the corner includes a component perpendicular to the plane (e.g. Z motion relative to the specified XY plane), it is the projection onto the specified plane that is used for calculating the corner angle. If the vector feedrate changes between the incoming and outgoing moves, the RMS average of the two feedrates is used in the calculations. However, the tangential acceleration required to effect this change is not factored into the cornering acceleration limit calculations.

When **Coord[*x*].CornerAccel** is used to control the blending time, the value of **Coord[*x*].Ta** (which can be set by the **`ta{data}`** command), which would otherwise control this, is not used. However, its value is maintained, so it can be used later as appropriate (e.g. for the initial acceleration of the next move sequence) without its value having to be re-specified. If **Coord[*x*].CornerAccel** is set to the default value of 0.0, the value of **Coord[*x*].Ta** in force at the time specifies the blending time.

In computing the required blending time to obtain the specified cornering acceleration, the present value of **Coord[*x*].Ts** (which can be set by the **`ts{data}`** command) is used unchanged. The total blend time is a combination of the computed "Ta" time and the pre-specified Ts time.

If the blending time computed to obtain the specified corner acceleration is less than **Coord[*x*].Td**, the value of **Coord[*x*].Td** is used instead. (If **Coord[*x*].Td** is less than **Coord[*x*].Ts**, then the **Ts** time acts as the effective **Td** time.) If the computed blending time is so large that it would require the blend to start before the beginning of the "constant-speed" portion

of the incoming move, the blending time is reduced so that the blend starts where the constant-speed portion would have begun.

Note that if subsequent motor acceleration limits are applied in the segmented lookahead algorithm, the path and the corner size are not changed; the speed along the path may be reduced to observe the motor limits.

If **Coord[*x*].CornerError** is also greater than 0.0, Power PMAC will then compare the size of the error between the blended path computed using **Coord[*x*].CornerAccel** and the programmed corner point. If this error is larger than **Coord[*x*].CornerError**, the blend time is reduced so this error threshold is not exceeded. This modification means that the acceleration value will be exceeded at the programmed feedrate; buffered lookahead can be used to reduce the commanded speed at the corner and keep the acceleration within limits.

Coord[*x*].CornerBlendBp

Description: Corner angle blending break point

Range: -1.0 .. 1.0 (floating-point)

Units: Angle cosine

Default: 0.0 (disabled)

Legacy I-variable alias: I*sx*83

Coord[*x*].CornerBlendBp specifies the threshold in the coordinate system between corner angles for which **`linear`** and **`circle`**-mode moves are directly blended together, and those for which the axes are stopped in between the incoming and outgoing moves for the corner. **Coord[*x*].CornerBlendBp** is only used if **Coord[*x*].NoBlend** is set to 0 permit blending. If **Coord[*x*].NoBlend** is set to 1, no blending occurs between any moves, and **Coord[*x*].CornerBlendBp** is not used.

If **Coord[*x*].CornerBlendBp** is set exactly to the default value of 0.0, no decision on blending is made based on corner angle. If a threshold corner angle of 90° is desired – for which **Coord[*x*].CornerBlendBp** would be 0.0 – **Coord[*x*].CornerBlendBp** should be set to a value that is not exactly 0.0, for example to 0.00001.

Coord[*x*].CornerBlendBp is expressed as the cosine of the change in directed angle between the incoming and outgoing moves. (The change in directed angle is equal to 180° minus the included angle of the corner.) As such, it can take a value between -1.0 and +1.0. If the two moves have the same directed angle at the move boundary (i.e. they are moving in the same direction), the change in directed angle is 0°, and the cosine is 1.0. As the change in directed angle increases, the corner gets sharper, and the cosine of the change in directed angle decreases. For a total reversal, the change in directed angle is 180°, and the cosine is -1.0. The change in directed angle is evaluated in the plane defined by the **`normal`** command (default is the XY-plane); if the corner also involves axis movement perpendicular to this plane, it is the projection of movement into this plane that matters.

If the cosine of the change in directed angle at a corner is less than **Coord[*x*].CornerBlendBp** (a large change in directed angle; a sharp corner), Power PMAC will automatically disable blending between the two moves, and bring the commanded trajectory to a stop at the end of the incoming move. If **Coord[*x*].InPosTimeout** is greater than 0, Power PMAC then verifies that all motors in the coordinate system are "in position". Next, if the cosine of this angle is also less than **Coord[*x*].CornerDwellBp**, a dwell of **Coord[*x*].AddedDwellTime** real-time interrupt periods is executed. Finally, the outgoing move is automatically started.

If the cosine of the change in directed angle at a corner is greater than or equal to **Coord[*x*].CornerBlendBp** (a small change in directed angle; a gradual corner), Power PMAC will directly blend the incoming and outgoing moves with its normal blending algorithms.

The operation of **Coord[*x*].CornerBlendBp** is mostly independent of the operation of the similar function of **Coord[*x*].CCAddedCornerBp**, which controls for outside corners in 2D cutter-radius compensation whether an arc move will be added based on the change in directed angle. **Coord[*x*].CornerBlendBp** works regardless of whether cutter-radius compensation is active or not, or whether the corner is an inside or outside corner when cutter-radius compensation is active. However, if this is an outside compensated corner with an added arc, Power PMAC computes the "corner" angles between the incoming and outgoing moves and the added arc moves. These angles are almost always 0^{o}, so would always be blended together.

Example

If it is desired that motion be stopped if the change in directed angle is greater than 30° (included angle less than 150°), then **Coord[*x*].CornerBlendBp** should be set to 0.866, because cos $\Delta\theta$ = cos 30° = 0.866.

Coord[*x*].CornerDwellBp

Description: Corner angle dwelling break point

Range: -1.0 .. 1.0 (floating-point)

Units: Angle cosine

Default: 0.0 (disabled)

Legacy I-variable alias: I*sx*85

Coord[*x*].CornerDwellBp specifies the threshold in the coordinate system between corner angles for which **`linear`** and **`circle`**-mode are made without an automatically intervening dwell, and those for which a dwell of **Coord[*x*].AddedDwellTime** real-time-interrupt periods is automatically added. **Coord[*x*].CornerDwellBp** is only used if **Coord[*x*].NoBlend** is set to 0 permit blending. If **Coord[*x*].NoBlend** is set to 1, no blending occurs between any moves, and **Coord[*x*].CornerBlendBp** is not used.

The corner angle is only evaluated against **Coord[*x*].CornerDwellBp** if the blending at the corner has been disabled due to the action of **Coord[*x*].CornerBlendBp**. This means that **Coord[*x*].CornerDwellBp** must be set to a value less than **Coord[*x*].CornerBlendBp** if it is to have any effect.

If **Coord[*x*].CornerDwellBp** is set exactly to the default value of 0.0, no dwell is ever automatically added. If a threshold corner angle of 90° is desired – for which the **Coord[*x*].CornerDwellBp** would be 0.0 – **Coord[*x*].CornerDwellBp** should be set to a value that is not exactly 0.0, for example to 0.00001.

Coord[*x*].CornerDwellBp is expressed as the cosine of the change in directed angle between the incoming and outgoing moves. (The change in directed angle is equal to 180° minus the included angle of the corner.) As such, it can take a value between -1.0 and +1.0. If the two moves have the same directed angle at the move boundary (i.e. they are moving in the same direction), the change in directed angle is 0°, and the cosine is 1.0. As the change in directed angle increases, the corner gets sharper, and the cosine of the change in directed angle decreases. For a total reversal, the change in directed angle is 180°, and the cosine is -1.0. The change in directed angle is evaluated in the plane defined by the **`normal`** command (default is the XY-plane); if the corner also involves axis movement perpendicular to this plane, it is the projection of movement into this plane that matters.

If the cosine of the change in directed angle at a corner is less than **Coord[*x*].CornerDwellBp** (a large change in directed angle; a sharp corner), Power PMAC will automatically add a dwell of **Coord[*x*].AddedDwellTime** real-time-interrupt periods before the outgoing move is started.

If the cosine of the change in directed angle at a corner is greater than or equal to **Coord[*x*].CornerDwellBp** (a small change in directed angle; a gradual corner), Power PMAC will not automatically add a dwell.

The operation of **Coord[*x*].CornerDwellBp** is mostly independent of the operation of the similar function of **Coord[*x*].CCAddedCornerBp**, which controls for outside corners in 2D cutter-radius compensation whether an arc move will be added based on the change in directed angle. **Coord[*x*].CornerDwellBp** works regardless of whether cutter-radius compensation is active or not, or whether the corner is an inside or outside corner when cutter-radius compensation is active.However, this is an outside compensated corner with an added arc, Power PMAC computes the "corner" angles between the incoming and outgoing moves and the added arc moves. These angles are almost always 0^o, so a dwell would not be added.

Example

If it is desired that motion be stopped if the change in directed angle is greater than 45° (included angle less than 135°), then **Coord[*x*].CornerDwellBp** should be set to 0.707, because cos $\Delta\theta$ = cos 45° = 0.707.

Coord[*x*].CornerError

Description: Cornering error for blended moves

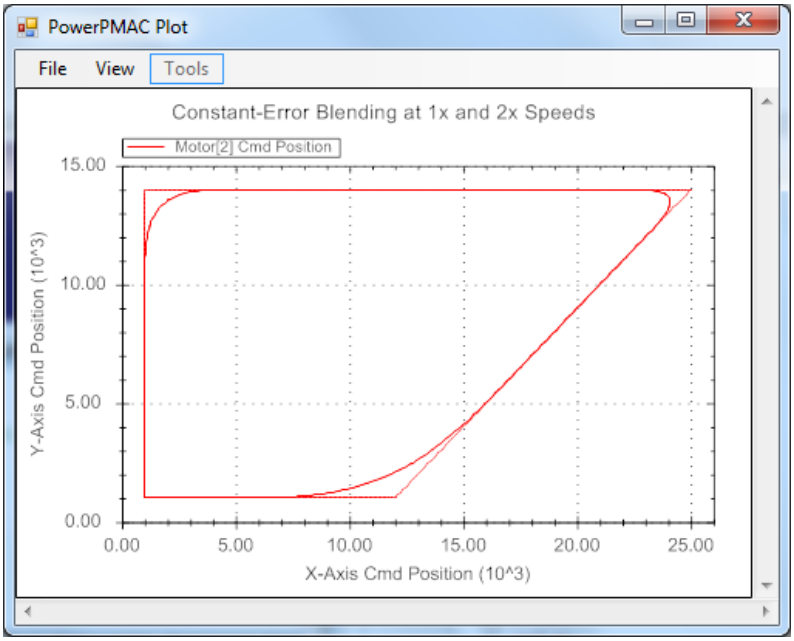
Range: Non-negative floating-point

Units: Axis units

Default: 0.0 (disabled)

Coord[*x*].CornerError, if set to a positive value, specifies the cornering error that Power PMAC will use as it blends moves together. This error is the minimum distance between the blended path at the corner and the programmed point for the corner. Based on the vector speed and corner angle, Power PMAC will compute the blending times that provide this error.

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

Note that if **Coord[*x*].CornerRadius** is set to a positive value, **Coord[*x*].CornerError** will not be used – corners will be defined by size, not blending error. If blending at the corner is disabled for any reason, including a corner sharper than the threshold defined by **Coord[*x*].CornerBlendBp**, then **Coord[*x*].CornerError** will not be used at the corner.

If **Coord[*x*].CornerAccel** is set to a positive value, the blending time will first be calculated to obtain the specified cornering acceleration. If **Coord[*x*].CornerError** is also set to a positive value, it then computes the cornering error from this time, and if this error exceeds the limit of the parameter, the blending time will be reduced so that the corner error limit is not exceeded. (This will yield a cornering acceleration at the programmed speed that is higher than the parameter value; subsequent buffered lookahead calculations can be used to reduce the speed at the corner and keep the acceleration within limits.)

The corner angle is computed in the plane specified by the **`normal`** vector. If either move is a **`circle`**-mode move, the direction of the move immediately at the unblended corner is used to compute the corner angle. If the motion at the corner includes a component perpendicular to the plane, (e.g. Z motion relative to the specified XY plane), it is the projection onto the specified plane that is used for calculating the corner angle. If the vector feedrate changes between the incoming and outgoing moves, the RMS average of the two feedrates is used in the calculations.

When **Coord[*x*].CornerError** is used to control the blending time, the value of **Coord[*x*].Ta** (which can be set by the **ta*{data}*** command), which would otherwise control this, is not used. However, its value is maintained, so it can be used later as appropriate (e.g. for the initial acceleration of the next move sequence) without its value having to be re-specified.

The corner error calculation routine maintains the ratio of **Coord[*x*].Ts** to **Coord[*x*].Ta** in calculating the blending parameters for the corner, as this ratio determines the shape of the corner. (The larger the relative value of **Ts**, the squarer the corner shape will be.) At least one of these values must be greater than 0.0 for the corner error algorithm to work.

If the blending time computed to obtain the specified corner error is less than **Coord[*x*].Td**, the value of **Coord[*x*].Td** is used instead. (Remember that a **ta*{data}*** command sets the value of **Coord[*x*].Td** as well as **Coord[*x*].Ta**.) If the computed blending time is so large that it would require the blend to start before the beginning of the "constant-speed" portion of the incoming move, the blending time is reduced so that the blend starts where the constant-speed portion would have begun.

Note that if subsequent motor acceleration limits are applied in the segmented lookahead algorithm, the path and the corner size are not changed; the speed along the path may be reduced to observe the motor limits.

Coord[*x*].CornerRadius

Description: Cornering radius for blended moves

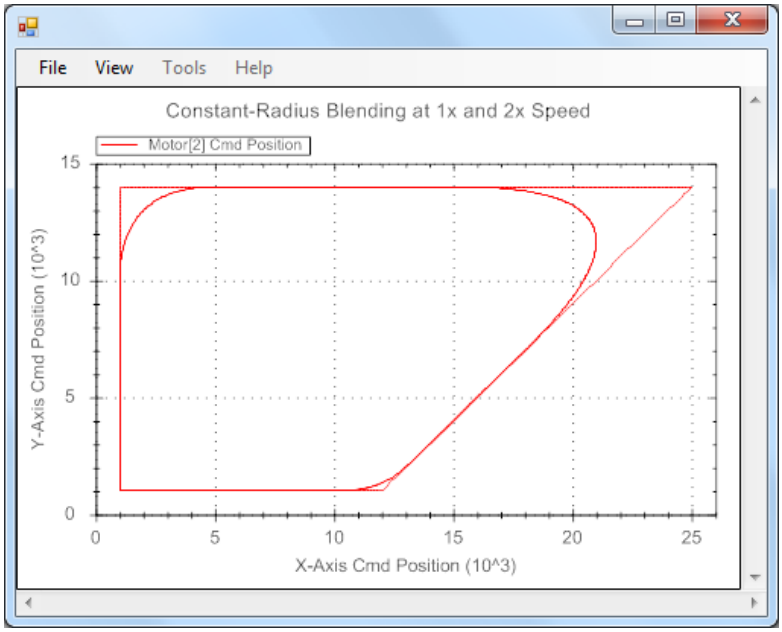
Range: Non-negative floating-point

Units: Axis units

Default: 0.0 (disabled)

Coord[*x*].CornerRadius, if set to a positive value, specifies the cornering "radius" that Power PMAC will use as it blends moves together. Based on the vector speed and corner angle, Power PMAC will compute the blending time that provides this "radius". This blending time, along with the vector speed, determines the corner "size". Note that the blended corner path is not a true circular arc – the "radius specified here is the distance from the start and end of the blended corner along lines perpendicular to the path to the intersection of these two lines.

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 corner angle is computed in the plane specified by the **normal** vector. If either move is a **circle**-mode move, the direction of the move immediately at the unblended corner is used to compute the corner angle. If the motion at the corner includes a component perpendicular to the plane (e.g. Z motion relative to the specified XY plane), it is the projection onto the specified plane that is used for calculating the corner angle. If the vector feedrate changes between the incoming and outgoing moves, the RMS average of the two feedrates is used in the calculations.

When **Coord[*x*].CornerRadius** is used to control the blending time, the value of **Coord[*x*].Ta** (which can be set by the **ta*{data}*** command), which would otherwise control this, is not used. In addition, no S-curve time is used in the blending acceleration, regardless of the value of **Coord[*x*].Ts** (which can be set by the **ts*{data}*** command). However, the values of these elements are maintained, so they can be used later as appropriate (e.g. for the initial acceleration of the next move sequence) without its value having to be re-specified. If **Coord[*x*].CornerRadius** is set to the default value of 0.0, the values of **Coord[*x*].Ta** and **Coord[*x*].Ts** in force at the time specify the blending time.

If the blending time computed to obtain the specified corner acceleration is less than **Coord[*x*].Td**, the value of **Coord[*x*].Td** is used instead. (If **Coord[*x*].Td** is less than **Coord[*x*].Ts**, then the **Ts** time acts as the effective **Td** time.) If the computed blending time is so large that it would require the blend to start before the beginning of the "constant-speed" portion of the incoming move, the blending time is reduced so that the blend starts where the constant-speed portion would have begun.

Coord[*x*].Dprog

Description: Subprogram number for D-code subroutines

Range: Positive integers

Units: Subprogram numbers

Default: 1003

Coord[*x*].Dprog controls which subprogram is called on the execution of a "D-code". When program execution encounters the **D*{data}*** syntax, program flow jumps to the subprogram with the number specified by **Coord[*x*].Dprog**, at the line jump label whose number is 1000 times the value of ***{data}***. For example, if **Coord[*x*].Dprog** is set to the default value of 1003, **D43** causes a jump to line jump label **N43000:** of **subprog 1003**.

This data structure element permits different coordinate systems to call different subroutines with the same D-code. However, in the default configuration, all coordinate systems use **subprog 1003** for their D-code subroutines.

Coord[x].EndDelay

Description: Delay time automatically added to end of move sequence

Range: Non-negative floating-point

Units: Milliseconds

Default: 0.0

Coord[*x*].EndDelay specifies how much time, if any, is to be added as a "delay" period at the end of a continuous move sequence of linear and/or circle-mode moves. During this period, all axes in the coordinate system are commanded to hold their most recent position.

The most common reason to add such a period is the use of low-pass or band-reject filtering in either the inverse-kinematic subroutine or the motor trajectory pre-filters. These effectively extend the time of the commanded move trajectory past the time directly specified by the move commands, as the filter output positions approach the settled input positions. The added delay permits the outputs of these filters to converge onto the final commanded position values.

The addition of this delay time is caused by a **dwell** command (even a **dwell 0**) at the end of a sequence of blended moves in continuous execution mode. The delay is added before the dwell time. The added time, with zero velocity into the filters, before the dwell period, is (**Coord[*x*].EndDelay** + **Coord[*x*].Ts**). If **EndDelay** is less than **Ts**, the added time is (2 * **Coord[*x*].Ts**).

During the added delay period, the **Coord[*x*].TimersEnabled** status bit is set to 1, indicating that a timed command is being executed by all of the motors in the coordinate system. If segmentation was active for the incoming move sequence (**Coord[*x*].SegMoveTime** > 0), segmentation will be active during the added delay, so the inverse-kinematic subroutine will continue to execute during this period.

Coord[*x*].EndDelay is new in V2.1 firmware, released 1st quarter 2016.

Coord[x].FeedHoldSlew

Description: Feed-hold slew rate

Range: Floating-point

Units: Milliseconds per servo cycle

Default: 0.00001

Legacy I-variable alias: I*x*95

Coord[*x*].FeedHoldSlew controls the rate of change of the time base for the coordinate system when it is going into, or coming out of, feed hold. If a feed hold command is issued to the coordinate system, the actual time base value will decrement ("slew") towards 0 at a rate determined by **Coord[*x*].FeedHoldSlew**.

That is, each servo cycle, the value in **Coord[*x*].TimeBase**, which is used to increment the numerical time value (in milliseconds) for the coordinate system, has subtracted from it the value of **Coord[*x*].FeedHoldSlew** until it reaches 0. When the subsequent resuming command (e.g. **r** or **resume**) is issued to the coordinate system, the time base value will increment back to its previous value at this same rate.

The "real-time" time base value (%100) is determined by the global data structure element **Sys.ServoPeriod**, which should be set (with units of milliseconds) to the true period of the servo clock cycle as determined by the Servo or MACRO ASIC that controls the system's clock frequencies. The time (in servo cycles) for a change of 100% to occur can be calculated as:

$$100\%SlewTime\ (servo\_cycles) = \frac{Sys.ServoPeriod}{Coord\ [x].FeedHoldSlew}$$

Similarly, the time (in milliseconds) for this change to occur can be calculated as:

$$100\%SlewTime\ \ (\text{sec}) = \frac{Sys.ServoPeriod^2}{Coord\ [x].FeedHoldSlew}$$

Inverting this relationship, the required value of this element for a given slew time is:

$$Coord\ [x].FeedHoldSlew = \frac{Sys.ServoPeriod^2}{100\%SlewTime\ \ (\text{sec})}$$

This control of the rate of change of the actual time base used in the interpolation calculations permits step changes in the desired time-base value without a resulting step change in the velocity of axes using the time base.

If **Coord[*x*].FeedHoldSlew** is set to a positive value, the coordinate system time-base value will change at a rate equal to the magnitude of this value each servo cycle, regardless of the resulting acceleration values for the motors in the coordinate system.

If **Coord[*x*].FeedHoldSlew** is set to a negative value, the rate of change coordinate time-base value may be further limited if changing at a rate equal to the magnitude of this value would cause the acceleration of any motor in the coordinate system to exceed its **Motor[*x*].InvAmax** limit, or the deceleration of any motor in the coordinate system to exceed its **Motor[*x*].InvDmax** limit. If the trajectory was originally specified not to exceed these constraints at 100% time base, then changes within the +/-100% range with **FeedHoldSlew** less than zero will not exceed these constraints.

If **Coord[*x*].FeedHoldSlew** is set exactly to 0.0, no changes in the active time-base value due to a hold command or a restart command are possible.

Coord[*x*].FeedTime

Description: Velocity time units

Range: Non-negative floating-point

Units: Milliseconds

Default: 1000 (seconds)

Legacy I-variable alias: I*sx*90

Coord[*x*].FeedTime defines the time units used in commanded velocities (feedrates) in motion programs executed by the coordinate system. Velocity units are comprised of length or angle units divided by time units. The length/angle units are determined in the axis definition statements or kinematic subroutines, plus any selected transformation matrix, for the coordinate system.

Coord[*x*].FeedTime sets the time units. **Coord[*x*].FeedTime** itself has units of milliseconds, so if it is set to 60,000, the time units are 60,000 milliseconds, or minutes. The default value of **Coord[*x*].FeedTime** is 1000, specifying velocity time units of seconds.

This parameter affects two types of motion program values: vector feedrates for blended moves (**`linear`** and **`circle`** mode), and signed axis velocities for PVT-mode moves.

Coord[x].FProtect

Description: Illegal feedrate protection control

Range: 0 .. 1

Units: Boolean

Default: 0

Coord[*x*].FProtect specifies whether the Power PMAC will automatically protect against "illegal" feedrate specifications or not. The protection is mainly intended for CNC-style applications where blended moves are only specified by feedrate and never move time.

If **Coord[*x*].FProtect** is set to its default value of 0, a negative value associated with the program **F** command (e.g. **F-100**) will be accepted as valid and treated as a "move time" command with the move time in milliseconds set to the magnitude of the value (e.g. 100 msec for F**-100**). Element **Coord[*x*].Tm** is set to the negative of the F-value whether that value is positive or negative.

If the saved value of **Coord[*x*].FProtect** is 0, then the saved value of **Coord[*x*].Tm** is automatically used as the initial feedrate (if less than 0.0) or move time (if greater than or equal to 0.0). In execution, the value of **Coord[*x*].Tm** is never automatically changed; it is only changed on an **F** or **tm** program command, or by a user command to set the element value directly.

If **Coord[*x*].FProtect** is set to 0, if a linear or circle-mode blended move is commanded with the value of **Coord[*x*].Tm** exactly equal to 0.0, the move is treated as a valid one in "move time" mode, with the move time set to 0.0, so acceleration times control.

If **Coord[*x*].FProtect** is set to 1, a program **F** command in a motion program with a negative value (e.g. **F-50**) will be rejected, and the program will be stopped with a run-time error (**Coord[*x*].ErrorStatus** = 21).

If **Coord[*x*].FProtect** is set to 1, then at the start of execution of a motion program from the beginning of the program, **Coord[*x*].Tm** is automatically set to 0.0. If a linear or circle-mode blended move is commanded with the value of **Coord[*x*].Tm** less than or equal to 0.0 (move-time specification mode), the command is rejected and the motion program is stopped with a run-time error.

Coord[*x*].FProtect is new in version 2.2 firmware, released 3rd quarter 2016. At its default value of 0, operation is compatible with earlier firmware versions.

Coord[*x*].GoBack

Description: Motion program jumps back before blend disabled

Range: 0 .. 255

Units: Enumeration

Default: 0

Coord[*x*].GoBack specifies how many times motion-program execution in the coordinate system can "jump back" while looking for the next move, dwell, or delay command before it will automatically disable blending into the next move. Execution can jump back (**GoBack** + 1) times without disabling blending.

Note that the motion command(s) found must be create motion of enough time to advance the trajectory into the next real-time interrupt period (as set by **Sys.RtIntPeriod**) if not a segmented move, or into the next segmentation period (as set by **Coord[*x*].SegMoveTime**) for Power PMAC to consider its present round of motion commands to be complete.

A "jump back" is caused either by the end of a **while** or **do..while** loop, or by a **goto** command to a previous line in the program. Returns from subroutines or subprograms do not count for this purpose.

The automatic disabling of blending on multiple "jumps back" is intended to prevent the case where a blended move is executing while the program is stuck in an indefinite loop trying to find the next command that generates equations of motion. If the execution of the previous move reaches the point where it is expecting to blend on the fly into a new move, but the equations of motion for that new move have not yet been calculated, a "run time error" is generated, and Power PMAC automatically generates an "abort" command, halting program execution and decelerating all motors to a stop according to their **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs** parameters. Note that the deceleration will not necessarily be along any programmed path, and in general will not end at a programmed point.

When this function detects (**GoBack** + 2) jumps back in the program while looking for the next command that creates equations of motion of sufficient time, it automatically recalculates the end of the most recently calculated move to bring it to a stop at its programmed endpoint, just as if there were a **dwell** command after the move. Program execution is suspended until the move execution has stopped at this point, at which time execution automatically resumes. Indefinite looping while all motors are stopped does not present potential run-time-error problems.

Note

Saved setup element **Coord[*x*].GoBack**, which controls the number of jumps back permitted in motion program execution, is independent of non-saved setup element **Coord[*x*].Ldata.GoBack**, which controls the number of jumps back permitted in kinematic subroutines called by the motion program.

Coord[*x*].GoBack constitutes bits 0 – 7 of the full-word element **Coord[*x*].Control[0]**.

Coord[*x*].Gprog

Description: Subprogram number for G-code subroutines

Range: Positive integers

Units: Subprogram numbers

Default: 1000

Coord[*x*].Gprog controls which subprogram is called on the execution of a "G-code". When program execution encounters the **G*{data}*** syntax, program flow jumps to the subprogram with the number specified by **Coord[*x*].Gprog**, at the line jump label whose number is 1000 times the value of ***{data}***. For example, if **Coord[*x*].Gprog** is set to the default value of 1000, **G17** causes a jump to line jump label **N17000:** of **subprog 1000**.

This data structure element permits different coordinate systems to call different subroutines with the same G-code. However, in the default configuration, all coordinate systems use **subprog 1000** for their G-code subroutines.

Coord[x].HomeRequired

Description: Homing before motion program requirement control

Range: 0 .. 1

Units: Boolean

Default: 0 (disabled)

Coord[*x*].HomeRequired determines whether all motors assigned to position axes in the coordinate system must have established a position reference before motion programs can be executed in the coordinate system, or not

If **Coord[*x*].HomeRequired** is set to the default value of 0, motion program execution is permitted in the coordinate system even if one or more motors in the coordinate system have not established a position reference. This setting is backward compatible with older revisions of the Power PMAC firmware which did not have this setup element.

If **Coord[*x*].HomeRequired** is set to 1, motion program execution is not permitted in the coordinate system unless all motors assigned to position axes in the coordinate system have established a position reference. This motor position reference can be established through a successful homing search move, a successful absolute position sensor read, or a "homez" zero-move homing. In each case, motor status bit **Motor[*x*].HomeComplete**, which is 0 at power-on/reset, is set to 1. If this bit is 1 for every motor assigned to a position axis in the coordinate system, then coordinate system status bit **Coord[*x*].HomeComplete** is set to 1. This bit must be equal to 1 in this mode to permit motion program execution in the coordinate system.

Motors given the null definition (**#*x*->0**) in the coordinate system and motors assigned as a spindle axis (**#*x*->S**, **S0**, **S1**) in the coordinate system do not need to have a position reference established to permit program execution. (Note, however, that motors assigned as spindle axes are often dynamically changed to position axes, and so should generally be referenced before starting motion program execution.) Motors assigned as inverse-kinematic axes count as position axes and must have a position reference established to permit motion program execution in this mode.

When **Coord[*x*].HomeRequired** is 1, a command to start motion program execution (e.g. **r**, **s**, **start**) when status bit **Coord[*x*].HomeComplete** is 0 will be rejected with an error.

Coord[x].InPosTimeout

Description: Blend-disable in-position timeout value

Range: Non-negative integers

Units: Real-time interrupt periods

Default: 0 (disabled)

Legacy I-variable alias: I*x*81

Coord[*x*].InPosTimeout, if set to a positive value, specifies that when blending is disabled between programmed moves, Power PMAC will determine that all axes in the coordinate system will be "in-position" before starting execution of the next move in the motion program. In this case, the value of **Coord[*x*].InPosTimeout** specifies the "time-out" value for the in-position check, expressed in real-time interrupt periods (**Sys.RtIntPeriod** + 1 servo cycles). If all axes in the coordinate system are not in-position within this time from the end of the incoming commanded move, the program will be stopped with a run-time error.

If **Coord[*x*].InPosTimeout** is set to 0, Power PMAC merely brings the commanded trajectory to a momentary stop before starting the next move in the cases where blending is disabled; it does not verify that any of the actual positions have reached this point.

The in-position check as specified by **Coord[*x*].InPosTimeout** is performed after programmed moves that are never blended (**`rapid`**-mode moves, programmed homing-search moves), or after moves that can be blended, but for which blending has been disabled setting **Coord[*x*].NoBlend** to 1, or because the corner is sharper than the angle specified by **Coord[*x*].CornerBlendBP**.

The in-position check enabled by **Coord[*x*].InPosTimeout** uses the **Motor[*x*].InPosBand** threshold for each motor, and the **Motor[*x*].InPosTime** number of scans for each motor. When all motors in the coordinate system have been verified to be "in position", the coordinate-system "in-position" status bit is set. After the in-position check, a dwell of the time specified by **Coord[*x*].AddedDwellTime** is inserted before execution of the next programmed move is started.

Coord[*x*].LHDistance

Description: Buffered Lookahead Distance

Range: Non-negative integers

Units: Move segments

Default: 0

Legacy I-variable alias: I*x*20

Coord[*x*].LHDistance controls the enabling of the lookahead buffering function for Coordinate System *x*, and if enabled, determines how far ahead the buffer will look ahead.

If **Coord[*x*].LHDistance** is set to 0 (the default), the buffered lookahead function is not used, even if a lookahead buffer has been defined.

If **Coord[*x*].LHDistance** is set to a value greater than 0, PMAC will look **Coord[*x*].LHDistance** segments ahead on **`linear`** and **`circle`** mode moves, provided that the coordinate system is in

segmentation mode (**Coord[*x*].SegMoveTime** > 0) and a lookahead buffer has been defined. The lookahead algorithm can extend the time for each segment in the buffer as needed to keep velocities under the **Motor[*x*].MaxSpeed** limits and the accelerations under those set by the **Motor[*x*].InvAmax** limits.

For proper lookahead control, **Coord[*x*].LHDistance** must be set to a value large enough so that Power PMAC looks ahead far enough that it can create a controlled stop from the maximum speed within the acceleration limit. This required stopping time for a motor can be expressed as:

$$StopTime(msec) = V_{\max} * \frac{1}{A_{\max}} = Motor[x].MaxSpeed \quad * Motor[x].InvAmax$$

All motors in the coordinate system should be evaluated to see which motor has the longest stopping time. This motor's stopping time will be used to compute **Coord[*x*].LHDistance**.

The average speed during this stopping time is $V_{max}/2$, so as the moves enter the lookahead algorithm at V_{max} (the worst case), the required time to look ahead is *StopTime*/2. Therefore, the required number of segments always corrected in the lookahead buffer can be expressed as:

$$SegmentsAhead = \frac{StopTime(msec)/2}{SegTime(msec/seg)} = \frac{Motor[x].MaxSpeed * Motor[x].InvAmax}{2 * Coord[x].SegMoveTime}$$

Because Power PMAC does not completely correct the lookahead buffer as each segment is added, the lookahead distance specified by **Coord[*x*].LHDistance** must be slightly larger than this. The formula for the minimum value of **Coord[*x*].LHDistance** that guarantees sufficient lookahead for the stopping distance is:

$$Coord[x].LHDistance = \frac{4}{3} * SegmentsAhead$$

If a fractional value results, round up to the next integer. A value of **Coord[*x*].LHDistance** less than this amount will not result in velocity or acceleration limits being violated; however, the algorithm will not permit maximum velocity to be reached, even if programmed.

Coord[*x*].LHDistance should not be set greater than the number of segments reserved in the **`define lookahead`** command, stored in the status element **Coord[*x*].LHSize**. If the lookahead algorithm runs out of buffer space, Power PMAC will automatically reduce the lookahead distance used to reflect the amount of space that is available.

It is possible to change the value of **Coord[*x*].LHDistance** dynamically while the program is running. This can be used to optimize the lookahead distance, setting it to the minimum required number of segments to provide the stopping distance based on the present commanded feedrate and feedrate override values. Setting this value dynamically can maximize the responsiveness of the system to changes such as new override values.

Example

The axes in a system have a maximum speed of 24,000 mm/min, or 400 mm/sec (900 in/min or 15 in/sec). They have a maximum acceleration of 0.05g or 500 mm/sec^2 (20 in/sec^2), and a count

resolution of 1μm. The motor units are counts. A maximum block rate of 200 blocks/sec is desired, so **Motor[*x*].SegMoveTime** is set to 5 msec. The parameters can be computed as:

- **Motor[*x*].MaxSpeed** = 400 mm/sec * 0.001 sec/msec * 1000 cts/mm = 400 cts/msec
- **Motor[*x*].InvAmax** = 0.002 sec^2/mm * 1000^2 $msec^2/sec^2$ * 0.001 mm/ct = 2.0 $msec^2$/ct
- **Coord[*x*].LHDistance** = [4/3] * [(400 cts/msec * 2.0 $msec^2$/ct) / (2 * 5 msec/seg)] = 107 seg

Coord[*x*].MaxCirAccel

Description: Maximum centripetal acceleration for circle moves

Range: Non-negative floating-point

Units: Axis units / (C.S. time units)^2

Default: 0.0 (disabled)

Legacy I-variable alias: I*x*78

Coord[*x*].MaxCirAccel, if set to a positive value, specifies the maximum centripetal acceleration that Power PMAC will permit for a feedrate-specified (**F**) **circle**-mode move in the coordinate system. If the move as programmed would yield a higher centripetal acceleration, Power PMAC will automatically lower the programmed speed for the move so that the limit is not exceeded. The centripetal acceleration is expressed as:

$$A = \frac{V^2}{R}$$

This limitation is done at the initial move calculation time, so it is not required to use the special lookahead buffer in conjunction with **Coord[*x*].MaxCirAccel**. It still may be desirable to use the special lookahead buffer, especially to manage the tangential acceleration into and out of a reduced-speed arc move.

The most important difference between limiting centripetal acceleration with **Coord[*x*].MaxCirAccel** and limiting it with the individual motor **Motor[*x*].InvAmax** acceleration limits has to do with the exact path generated. Power PMAC's circular interpolation algorithms introduce a radial error term that can be described by:

$$E = \frac{V^2 T^2}{6R}$$

where V is the speed along the arc determined by the motion program (possibly limited by **Coord[*x*].MaxCirAccel**), T is the **Coord[*x*].SegMoveTime** segmentation time, and R is the radius of the arc. If **Coord[*x*].MaxCirAccel** is used to limit the speed of the arc, this error will be reduced. However, if the special lookahead is used to limit the speed, the error will be as large as if the arc move were run at full speed.

Coord[*x*].MaxCirAccel is expressed in the user length units for the linear axes (usually millimeters or inches) divided by the square of the user "feedrate time units" set by **Coord[*x*].FeedTime** for the coordinate system (usually seconds or minutes).

Example 1:

You want to limit the centripetal acceleration to 1.0g with **Coord[*x*].MaxCirAccel**. Your length units are millimeters, and your time units are seconds. **Coord[*x*].MaxCirAccel** can be calculated as follows:

$$MaxCirAccel = 1.0g * \frac{9.8\frac{m}{s^2}}{g} * \frac{1000mm}{m} = 9800\left(\frac{mm}{s^2}\right)$$

Example 2:

You want to limit your circular interpolation calculation errors to 0.001 inches. Your length units are inches, your time units are minutes, and your **Coord[*x*].SegMoveTime** segmentation time is 10 milliseconds. **Coord[*x*].MaxCirAccel** can be calculated as follows:

$$MaxCirAcce\,l = \frac{V^2}{R} = \frac{6E}{T^2} = \frac{6 * 0.001in}{0.01^2 s^2} * \frac{60^2 s^2}{min^2} = 216{,}000\left(\frac{in}{min^2}\right)$$

Example 3:

Your system is capable of 10 m/s^2 acceleration (about 1g). Your length units are millimeters, your time units are minutes, and your **Coord[*x*].SegMoveTime** segmentation time is 2 milliseconds. **Coord[*x*].MaxCirAccel** can be calculated as follows:

$$MaxCirAcce\,l = 10\frac{m}{s^2} * \frac{1000mm}{m} * \frac{3600s^2}{min^2} = 36{,}000{,}000\left(\frac{mm}{min^2}\right)$$

At this setting, your maximum circular interpolation calculation errors can be computed as:

$$E = \frac{V^2 T^2}{6R} = MaxCirAcce\,l * \frac{T^2}{6} = \frac{36{,}000{,}000}{6}\frac{mm}{min^2} * 0.002^2 s^2 * \frac{min^2}{60^2 s^2} = 0.0067mm$$

Coord[*x*].MaxFeedrate

Description: Maximum permitted vector feedrate

Range: Non-negative floating-point

Units: Axis units / C.S. time units

Default: 0.0 (inactive)

Legacy I-variable alias: I*sx*98

Coord[*x*].MaxFeedrate specifies the greatest vector-velocity ("feedrate") magnitude that can be specified for the coordinate system, preventing a program from commanding an excessive value.

If the magnitude of **Coord[*x*].Tm**, when less than zero to specify vector velocity, is greater than this parameter, the magnitude of this parameter is used instead. This is true whether

Coord[*x*].Tm is set using the program command **F*{data}***, or is set directly through an assignment command. The value of the **Coord[*x*].Tm** element is not changed due to this limit.

This check of the commanded feedrate against the limit is done at the programmed move block calculation time, before any motor calculations are done. Motor calculations, whether with the special lookahead buffer or not, compare individual motor velocities against the **Motor[*x*].MaxSpeed** limits; they do not check vector velocities. In Cartesian systems, the key velocity limit for **linear** and **circle**-mode programmed moves will probably be **Coord[*x*].MaxFeedrate**, as the motor limits will often permit higher speeds for **rapid**-mode moves.

Coord[*x*].MaxFeedrate is expressed in the user length units for the feedrate axes (usually millimeters or inches) divided by the user "feedrate time units" set by **Coord[*x*].FeedTime** for the coordinate system (usually seconds or minutes).

If **Coord[*x*].MaxFeedrate** is set to the default value of 0.0, Power PMAC will not check the programmed feedrate value against a limit.

Coord[*x*].MinArcLen

Description: Circle-Mode Minimum Arc Length

Range: Non-negative floating point

Units: Radians

Default: 0.0

Legacy I-variable alias: I*sx*97

Coord[*x*].MinArcLen specifies the magnitude of the angle subtended by the shortest arc that Power PMAC's **circle** mode can generate. Any programmed **circle**-mode move with an IJK-vector representation of the center that covers an angle smaller than **Coord[*x*].MinArcLen** is executed as a full circle plus the programmed angle change. Any such move which covers an angle greater than **Coord[*x*].MinArcLen** is executed as an arc smaller than a full circle.

The purpose of **Coord[*x*].MinArcLen** is to support the circle programming standard that permits a full-circle move to be commanded simply by making the end point equal to the starting point (0-degree arc), yet allow for round-off errors.

Note that if **Coord[*x*].MinArcLen** is set to its default value of 0.0, the end point of the circle axes must be exactly equal to the start point to full double-precision floating-point resolution in order for a full-circle move to be executed.

Coord[*x*].Mprog

Description: Subprogram number for M-code subroutines

Range: Positive integers

Units: Subprogram numbers

Default: 1001

Coord[*x*].Mprog controls which subprogram is called on the execution of a "M-code". When program execution encounters the **M*{data}*** syntax, program flow jumps to the subprogram with the number specified by **Coord[*x*].Mprog**, at the line jump label whose number is 1000 times the value of ***{data}***. For example, if **Coord[*x*].Mprog** is set to the default value of 1001, **M24** causes a jump to line jump label **N24000:** of **subprog 1001**.

This data structure element permits different coordinate systems to call different subroutines with the same M-code. However, in the default configuration, all coordinate systems use **subprog 1001** for their G-code subroutines.

Coord[*x*].Ndisplay

Description: Synchronous line label display control

Range: 0 .. 1

Units: Boolean

Default: 0

Coord[*x*].Ndisplay controls whether the present value of status element **Coord[*x*].Nsync** is reported at the start of the response when axis target positions are queried with the **&*x*t** command, or axis distances to go are queried with the **&*x*g** command. The value of **Coord[*x*].Nsync** is set to the value of the synchronizing line label **N*{data}*** at the start of execution of the next move in the motion program, so it can be used to show which move is presently executing.

If **Coord[*x*].Ndisplay** is set to its default value of 0, the value of **Coord[*x*].Nsync** is not reported at the start of the response, so only the axis values will be reported, and a sample response would be `X10 Y20 Z30`.

If **Coord[*x*].Ndisplay** is set to 1, the value of **Coord[*x*].Nsync** is reported at the start of the response, and a sample response would be `N560 X10 Y20 Z30`.

Coord[*x*]. Ndisplay constitutes bit 17 of the full-word element **Coord[*x*].Control[0]**.

Coord[*x*].NoBlend

Description: Move blend disable control

Range: 0 .. 3

Units: none

Default: 0

Legacy I-variable alias: I*sx*92

Coord[*x*].NoBlend controls whether programmed moves for the coordinate system are automatically blended or not. If it is set to its default value of 0, consecutive programmed moves of **`linear`**, **`circle`**, and **`spline`** mode are blended together with no intervening stop. Upcoming moves are calculated during the executing moves. However, if bit 0 (value 1) is set to 1, there is a brief stop in between each **`linear`** or **`circle`**-mode programmed move, during which the next move is calculated. If bit 1 (value 2) is set to 1, there is a brief stop in between each **`spline`**-mode move.

When blending is disabled, if **Coord[*x*].InPosTimeOut** is set to its default value of 0, the commanded trajectory is brought to a momentary stop before the next move is started, regardless of the actual positions of the motors. If **Coord[*x*].InPosTimeOut** is set to a value greater than 0, all of the motors in the coordinate system must be "in position" before the next move is started.

Coord[*x*].NoBlend constitutes bit 30 of the full-word element **Coord[*x*].Control[0]**.

Coord[*x*].NoCornerBp

Description: Pseudo-corner angle for blend and dwell calculations

Range: -1.0 .. 1.0 (floating point)

Units: Effective angle cosine

Default: 0.0

Coord[*x*].NoCornerBp specifies the effective angle that is to be used in corner calculations in the coordinate system when either the incoming or outgoing move has no component in the plane of calculation (as for a zero-distance move or one perpendicular to the plane), or does not exist (as at the beginning of the first move or end of the last move of a sequence). It is expressed as a cosine of the change in directed angle of motion of a "corner" in the plane defined by the **`normal`** vector. A value of 1.0 is equivalent to no change in angle; a value of 0.0 is equivalent to a 90° angle; a value of -1.0 is equivalent to a 180° reversal. This parameter permits the user to specify the behavior at these move boundaries.

At such a "corner", the value of **Coord[*x*].NoCornerBp** is compared to that of **Coord[*x*].CornerBlendBp** (if **CornerBlendBp** is activated with a setting not equal to 0.0) to decide whether the moves will be blended or not. If **NoCornerBp** is less than **CornerBlendBp**, Power PMAC will automatically disable blending at the move boundary, bringing the commanded trajectory to a stop at the end of the incoming move. If **Coord[*x*].InPosTimeout** is greater than 0, Power PMAC then verifies that all motors in the coordinate system are "in position" before proceeding. Next if **NoCornerBp** is also less than **Coord[*x*].CornerDwellBp** (if **CornerDwellBp** is activated with a setting not equal to 0.0), a dwell of **Coord[*x*].AddedDwellTime** real-time interrupt periods is executed. Finally, the outgoing move is automatically started.

If **NoCornerBp** is greater than or equal to **CornerBlendBp**, Power PMAC will directly blend the incoming and outgoing moves with its normal blending algorithms.

Coord[*x*].pDesTimeBase

Description: Time-base source pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: **Coord[*x*].DesTimeBase.a**

Legacy I-variable alias: I*x*93

Coord[*x*].pDesTimeBase specifies which register the coordinate system uses for its desired time-base value in its trajectory interpolation calculations. It contains the address of this register. It expects the contents of this register to be (double-precision) floating-point, in units of milliseconds.

When **Coord[*x*].pDesTimeBase** is set to its default value, the address of **Coord[*x*].DesTimeBase** for the same coordinate system, it is using the register that automatically responds to `%{constant}` on-line commands. If the time-base is left alone, or is under programmatic control, **Coord[*x*].pDesTimeBase** should be left at its default value.

For "external time base", in which the coordinate system's time base value is proportional to the frequency of an incoming encoder signal or pulse train, **Coord[*x*].pDesTimeBase** should be set to **EncTable[*i*].DeltaPos.a**, where the "*i*th" entry of the encoder table is processing this encoder or pulse train, and **DeltaPos** is the scaled calculated change in the count value for the servo cycle.

Coord[*x*].PosRollOver[*i*]

Description: Axis position rollover range

Range: Floating-point

Units: Axis units

Default: 0.0

Coord[*x*].PosRollOver[*i*] is an array of 6 values (indices 0 to 5) for the coordinate system where the values specify whether the 6 rotary axes for the coordinate system (A, B, C, AA, BB, and CC, corresponding to the indices 0 to 5, respectively) can be programmed in "rollover" mode or not, and if so, how large the rollover range is. If **Coord[*x*].PosRollOver[*i*]** is set to the default value of 0, rollover mode is not enabled, and the axis is treated the same as a linear axis.

The following table shows the axis affected by each individual element in the array:

Element	Axis	Element	Axis
Coord[*x*].PosRollover[0]	A	**Coord[*x*].PosRollover[3]**	AA
Coord[*x*].PosRollover[1]	B	**Coord[*x*].PosRollover[4]**	BB
Coord[*x*].PosRollover[2]	C	**Coord[*x*].PosRollover[5]**	CC

If **Coord[*x*].PosRollOver[*i*]** is greater than zero, then for a programmed axis move in absolute (**abs**) mode, the axis will take the shortest path around the circular range defined by **Coord[*x*].PosRollOver[*i*]** to get to the destination point. No absolute-mode move will be longer than half of a revolution (**Coord[*x*].PosRollOver[*i*]** /2) with rollover active in this mode. **Coord[*x*].PosRollOver[*i*]** will almost always be set to 360 in this mode, as this type of rotary axis is usually programmed in degrees.

If **Coord[*x*].PosRollOver[*i*]** is less than zero, then for a programmed axis move in absolute (**abs**) mode, the sign of the axis value in the move command is used as the direction to travel in the move. No absolute-mode move will be longer than a full revolution (**Coord[*x*].PosRollOver[*i*]**) with rollover active in this mode. **Coord[*x*].PosRollOver[*i*]** will almost always be set to -360 in this mode, as this type of rotary axis is usually programmed in degrees.

The sign of the commanded absolute destination in this mode is also part of the destination value. So a command of **A-90** in this mode is a command to go to -90 degrees (= +270 degrees) in the negative direction. For commands to move in the positive direction, the + sign is not required, but it is permitted (e.g. to command a move to 90 degrees in the positive direction, either **A90** or **A+90** can be used).

Power PMAC cannot store the difference between a +0.0 and a –0.0 destination command, so a command with a tiny non-zero magnitude for the end position must be used (e.g. **A+0.0000001** and **A-0.0000001**). This increment can be small enough not to have any effect on the final destination. A command exactly to 0.0 will cause travel in the shortest direction to the zero position in the cycle.

If using commands from a similar mode in which only the magnitude, and not the sign, of the value specifies the destination position, then the destination values for negative-direction moves must be modified so that the magnitude is 360 degrees minus the magnitude in the other mode. For example, if the command were **C-120**, specifying a move to (+)120 degrees in the negative direction, the command would have to be modified for Power PMAC to **C-240**, which specifies a move to -240 degrees (= +120 degrees) in the negative direction. Commands for positive-direction moves do not have to be modified.

Axis moves in incremental (**inc**) mode are not affected by either rollover mode. Jog moves for motors assigned to these axes are not affected by rollover. Reported motor and axis position is not affected by rollover. (To obtain position information "rolled over" to within one revolution, use the modulo (remainder) operator, either in Power PMAC or in the host computer.)

Coord[*x*].RadiusErrorLimit

Description: Circle-mode radius error limit

Range: Non-negative floating point

Units: Axis units

Default: 0.0 (disabled)

Legacy I-variable alias: I*sx*96

Coord[*x*].RadiusErrorLimit specifies the magnitude of the largest difference between the starting radius and ending radius of a **circle**-mode move that will constitute a legal move that can be executed. If the difference is larger than this magnitude, execution will be halted at the beginning of this move with the motion program aborted. In this case, a bit in the status element **Coord[*x*].RadiusError** (which is part of **Coord[*x*].ErrorStatus** and **Coord[*x*].Status[0]**, and can be queried with the **&*x*?** command) to be set. If **Coord[*x*].RadiusErrorLimit** is set to the default value of 0.0, this checking is disabled.

In a **circle**-mode move with vector-distance (I/J/K or II/JJ/KK) specification of the center point from the starting point, there is nothing that constrains the commanded end point to be the same distance from the center. If the distances are different, Power PMAC will execute an exponential spiral from starting point to ending point instead of a true circular arc.

In a **circle**-mode move with radius (**R**) specification, if the distance from the starting point to the ending point is more than twice the radius size, no circular arc is possible, and Power PMAC will generate an exponential spiral to the end point.

Sometimes this will be done intentionally; other times it will be due to programming error. Since in many cases, there can be tiny errors due to numerical round-off, some users want to be able to set a threshold distinguishing errors due to round-off from true programming errors. **Coord[*x*].RadiusErrorLimit** provides this threshold.

Coord[*x*].RapidVelCtrl

Description: Rapid move velocity mode control

Range: 0 .. 1

Units: none

Default: 0

Legacy I-variable alias: I*sx*79

Coord[*x*].RapidVelCtrl determines how Power PMAC computes the speed of "shorter" axes in a multi-axis rapid-mode move. If it is set to the default value of 0, all motors involved in the multi-axis move will be commanded to move at their specified speed (**Motor[*x*].MaxSpeed** if **Motor[*x*].RapidSpeedSel** = 1; **Motor[*x*].JogSpeed** if **Motor[*x*].RapidSpeedSel** = 0). This means that motors with lower distance/speed ratios will finish sooner than those with higher ratios, and the path in a Cartesian system will not be linear in the general case.

If **Coord[*x*].RapidVelCtrl** is set to 1, Power PMAC will compute the ratio of move distance to rapid speed for each motor in the move. Only the motor with the highest distance/speed ratio will actually be commanded at its specified speed. The commanded speeds for other motors will be

lessened so that they have the same ratio of distance to speed, yielding the same move time for all motors (before acceleration and deceleration are added). This makes the move path in a Cartesian system at least approximately linear (and truly linear if the acceleration and deceleration *times* are the same).

In both cases, each motor involved in the multi-axis move will use its own acceleration parameters **Motor[*x*].JogTa** and **Motor[*x*].JogTs** to calculate its acceleration and deceleration profiles.

Coord[*x*].RapidVelCtrl constitutes bit 28 of the full-word element **Coord[*x*].Control[0]**.

Coord[*x*].SegLinToPvt

Description: Linear moves executed as PVT enable

Range: 0 .. 3

Units: none

Default: 0

Coord[*x*].SegLinToPvt specifies whether **`linear`** mode moves commanded in segmentation mode (**Coord[*x*].SegMoveTime** > 0) are first automatically converted to PVT-mode moves or not. At the default value of 0, they are not converted, so are executed according to the rules of **`linear`** mode.

If **Coord[*x*].SegLinToPvt** is set to a value greater than 0, **`linear`** mode segmented moves are converted internally to **`pvt`** mode moves before execution. The commanded positions for each move are the same as for linear mode; the commanded velocities and times for each move are computed for the commanded path to pass smoothly right through the commanded positions at the programmed speed.

In this mode, Power PMAC uses the value of **Coord[*x*].Tm**, typically set by the **`F`** or **`tm`** program commands, but it does not use the values of **Coord[*x*].Ta**, **Td**, or **Ts**, so it ignores values set by the **`ta`**, **`td`**, and **`ts`** program commands.

This conversion provides a superior path quality when many points are spaced closely together on a contour. First, the commanded path passes exactly through the programmed points, whereas standard linear mode always passes to the inside of the programmed points. Second, no undesired "flats" are created, yielding a smoother path and more consistent axis velocity profiles.

If **Coord[*x*].SegLinToPvt** is set to 1, then at relatively sharp corners, velocity is increased somewhat to preserve a better path; if **Coord[*x*].SegLinToPvt** is set to 2, then at relatively sharp corners, velocity is maintained better at the cost of "bulges" in the path around the corner. A value of 3 is reserved for future use.

Coord[*x*].SegLinToPvt constitutes bits 22 – 23 of full-word element **Coord[*x*].Control[0]**.

Coord[x].SegMoveTime

Description: Time for coarse-interpolation segments

Range: Non-negative floating-point

Units: Milliseconds

Default: 0.0

Legacy I-variable alias: I*x*13

Coord[*x*].SegMoveTime controls whether the coordinate system is in "segmentation mode" or not, and if it is, what the "segmentation time" is in units of milliseconds.

If **Coord[*x*].SegMoveTime** is greater than 0.0, the coordinate system is in segmentation mode, and all **`linear`**, **`circle`**, and `pvt` mode trajectories are created by computing intermediate "segment" points with a coarse interpolation algorithm every **Coord[*x*].SegMoveTime** milliseconds, then executing a fine interpolation using a cubic spline algorithm every servo cycle.

While it is possible to execute programmed moves ("blocks") of a shorter time than this segmentation time, the segmentation algorithm will automatically skip over these blocks, effectively performing a smoothing function over multiple blocks.

This coarse/fine interpolation method activated by putting the coordinate system into segmentation mode is required for the coordinate system to be able to use any of the following features:

- Circular interpolation
- Cutter radius compensation
- Multi-block lookahead
- Inverse kinematics

If none of these features is required in the coordinate system, it is usually best to leave **Coord[*x*].SegMoveTime** at the default value of 0.0 to free up calculation time for other tasks. If **Coord[*x*].SegMoveTime** is 0.0, **`circle`** mode moves are executed as **`linear`** moves, cutter radius compensation is not performed, and inverse-kinematic calculations are only performed at move-end positions.

Typical values of **Coord[*x*].SegMoveTime** for segmentation mode are 2 to 20 msec. The smaller the value, the tighter the fit to the true curve, but the more computation is required for the moves, and the less is available for background tasks. If **Coord[*x*].SegMoveTime** is set too low, Power PMAC will not be able to do all of its move calculations in the time allotted, and it will stop the motion program with a run-time error.

The formula for the interpolation error introduced on a curved path by the segmentation mode is:

$$E = \frac{V^2 T^2}{6R}$$

where *V* is the velocity, *T* is the segmentation time, and *R* is the local radius, all expressed in consistent units. On a straight-line path, *R* is infinite, making the error equal to 0. If the velocity is expressed as a feedrate *F*, in units per minute, the formula is:

$$E = \frac{F^2\left(\frac{units^2}{min^2}\right)*\left(\frac{min^2}{60{,}000^2\, msec^2}\right)*(SegMoveTime)^2\left(msec^2\right)}{6R(units)} = \frac{F^2(SegMoveTime)^2}{2.16\times10^{10}R}$$

Example

At a feedrate of 5000 mm/min (200 in/min), and a radius of 50 mm (2 in), a value of **Coord[*x*].SegMoveTime** of 10 msec produces an interpolation error of 2.3 μm (0.00009 in).

Coord[*x*].SegOverrideSlew

Description: Slew rate for segmentation override changes

Range: Non-negative floating-point

Units: Fraction of segment per segment

Default: 0.0

Legacy I-variable alias: I*sx*16

Coord[*x*].SegOverrideSlew controls the rate of change of the "segmentation override" value for the coordinate system. The segmentation override feature permits advanced "feedrate override" capabilities for segmented moves (**`linear`**, **`circle`**, and **`pvt`**, with **Coord[*x*].SegMoveTime** > 0) without overriding the acceleration values as well.

This parameter determines how fast the internal override value that is used every segment approaches the user-commanded value of **Coord[*x*].SegOverride**, if the two are different. Both the internal and the command values are normalized, so a value of 1.0 provides "real time". Each segment, the "coarse interpolation" function increments the time along the segmented move by the product of **Coord[*x*].SegMoveTime** and the internal override value and computes the new positions at that "time". For example, with **Coord[*x*].SegMoveTime** set to 10 milliseconds and the internal override at 0.6, the mathematical time increment will be 10 * 0.6 = 6 msec. Since this move segment will be executed in a physical 10 msec, the move will execute at 60% of the programmed speed.

If the internal override value in **Coord[*x*].ActSegOverride** does not match the commanded **Coord[*x*].SegOverride** value, then each segment, the value of **Coord[*x*].SegOverrideSlew** will be added to or subtracted from the internal value until it reaches the commanded value.

Example

For a change in override of 100% in 1.0 seconds with a 5-msec time in **Coord[*x*].SegMoveTime**, we want a change of 1.0 in the internal override value over 200 segments, so **Coord[*x*].SegOverrideSlew** should be set to 1/200, or 0.005.

Coord[x].SoftLimitStopDis

Description: Software limit action mode control

Range: 0 .. 1

Units: none

Default: 0

Coord[*x*].SoftLimitStopDis determines what action is taken when a motor in the coordinate system hits one of its software overtravel limits (**Motor[*x*].MinPos** or **Motor[*x*].MaxPos**). If set to the default value of 0, the motion program is halted all motors in the coordinate system are "aborted" – decelerated to a closed-loop stop according to their parameters **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**.

However, if **Coord[*x*].SoftLimitStopDis** is set to 1, the motion program will continue, with the position of the motor that hit its software limit saturating at the limit value.

Coord[*x*].SoftLimitStopDis constitutes bit 29 of the full-word element **Coord[*x*].Control[0]**.

Coord[x].SplineTimeRotate

Description: Spline mode move-time table control

Range: 0 .. 1

Units: Boolean

Default: 0

Coord[*x*].SplineTimeRotate specifies how the table of move times in spline mode operates, controlling how a spline trajectory of varying move times executes. If it is set to the default value of 0, the table is fully "manually" controlled by the user program, providing increased flexibility in specifying the spline profile.

If **Coord[*x*].SplineTimeRotate** is set to 1, user specified move times are automatically "rotated" through the trajectory as each new move is commanded, making the resulting trajectories equivalent to those commanded in the same way in Turbo PMAC.

Coord[x].StepMode

Description: Motion-program single step mode

Range: 0 .. 3

Units: Enumeration

Default: 0

Coord[*x*].StepMode controls the coordinate system's action in response to a "single-step" (on-line `s` or buffered **`step`**) command.

If **Coord[*x*].StepMode** is set to the default value of 0, when a single-step command is given when the program is not currently executing, program calculation will continue until a single programmed move is executed. (A "move" here can be a dwell or delay.) This could entail program calculations of a single line, of part of a line, or of multiple lines.

In this mode, when a single-step command is given when the program is currently executing, further program calculations will be performed until a single additional programmed move can be calculated, and the already calculated moves will be executed. Compare this to the action of a "quit" (on-line **`q`** or buffered **`pause`**) command, which causes no further program calculations and just finishes executing previously calculated moves.

If a **`bstart`** (block start) command is encountered in the program before calculations are finished, single-step execution will continue until a **`bstop`** (block stop) command is encountered, no matter how many program lines or move commands are in between.

If the program is in 2D cutter compensation mode (**`ccmode1`** or **`ccmode2`**), the calculations necessary for the execution of a single move can involve more than one move, particularly when starting compensation, one move, or no moves, particularly when ending compensation. (Note that use of **`bstart`** and **`bstop`** overrides this potential for calculating a different number of moves depending on the status of the compensation mode.)

If **Coord[*x*].StepMode** is set to 1, when a single-step command is given when the program is not currently executing, program calculation will generally continue until it encounters the end of this program line at the "stack level" where execution started (typically the top-level motion program), even if no programmed moves are calculated. However, if the end of a move command occurs before the end of the program line, calculation will stop at the end of the move command.

It is possible for multiple program moves to execute in a subprogram called from this program line, permitting the execution of a full "canned cycle" by one single-step command.

In this mode, when a single-step command is given when the program is currently executing, no further program calculations are performed and move execution simply completes already calculated moves.

If the program is in 2D cutter compensation mode (**`ccmode1`** or **`ccmode2`**), the calculations necessary for the execution of a single move can involve more than one move, particularly when starting compensation, one move, or no moves, particularly when ending compensation.

If there are any move or dwell commands in subroutines called from the executing line of the program (as in a G04 dwell subroutine or a canned cycle), and it is desired to let execution continue past the end of the subroutine and to the end of the calling line, the subroutine should start with a **`bstart`** command and end with a **`bstop`** command immediately before the **`return`**. This will permit RS-274 style CNC programs with subroutine implementations of the codes (G, M, T, and D-codes) to execute in single-step mode according to standard expectations.

This mode is most commonly used in the operation of CNC-style applications if it is desired to give the operator an interactive single-step mode for test runs of part programs.

If **Coord[*x*].StepMode** is set to 2, when a single-step command is given when the program is not currently executing, program calculation will generally continue until it encounters the end of a program line at any "stack level", including in a called subprogram, even if no programmed moves are calculated. However, if the end of a move command occurs before the end of the program line, calculation will stop at the end of the move command.

If the program is in 2D cutter compensation mode (**ccmode1** or **ccmode2**), the calculations necessary for the execution of a single move can involve more than one move, particularly when starting compensation, one move, or no moves, particularly when ending compensation.

In this mode, when a single-step command is given when the program is currently executing, no further program calculations are performed and move execution simply completes already calculated moves.

This mode is most commonly used in the development of CNC-style applications to support the development and debugging of multi-move subroutines such as canned cycles.

If **Coord[*x*].StepMode** is set to 3, when a single-step command is given when the program is not currently executing, program calculation will generally continue until it encounters the end of a program line at any "stack level", including in a called subprogram, even if no programmed moves are calculated. However, if the end of a move command occurs before the end of the program line, calculation will stop at the end of the move command.

If the program is in 2D cutter compensation mode (**ccmode1** or **ccmode2**), the calculations for at most one compensated move are performed, even if this does not result in the actual execution of any moves because the compensated move pre-calculation buffer is not full.

In this mode, when a single-step command is given when the program is currently executing, no further program calculations are performed and move execution simply completes already calculated moves.

This mode is most commonly used in the development of CNC-style applications to support the development and debugging of sequences of compensated moves, and multi-move subroutines such as canned cycles.

Coord[*x*].SyncOps

Description: C.S synchronous assignment buffer size

Range: 0 .. 8192 * **Sys.MaxCoords**

Units: Buffered assignments

Default: 8192

Coord[*x*].SyncOps specifies the size of this coordinate system's synchronous assignment buffer. The default buffer configuration allocates 8192 assignments per coordinate system. A coordinate

system's buffer must be large enough to store all of the pending assignments from the time the commands are encountered at calculation time until they are executed at the beginning of execution of the next commanded move in the program following the command that uses axes or motors in the coordinate system. Note that these delayed synchronous assignments are a coordinate system function, even if commanded from a PLC program.

In the large majority of Power PMAC applications, the default value of **Coord[*x*].SyncOps** for each coordinate system of 8192 is more than enough to handle all requirements for synchronous assignments. However, for applications in which the special lookahead buffer is scanning many moves ahead with the potential for multiple synchronous assignments per move and significant retrace capability, a particular coordinate system's buffer may have to be sized larger than this.

At power-up/reset, the Power PMAC CPU uses the saved values of all of the **Coord[*x*].SyncOps** elements to organize the synchronous assignment buffer. A total of 1M (1,048,576) synchronous assignments can be buffered for all coordinate systems. It sets the value of **Coord[*x*].SyncOffset** for each coordinate system, which specifies the offset of the coordinate system's buffer from the start of the overall buffer space, to be equal to the number of assignments allocated to all of the lower-numbered coordinate systems – that is, to the sum of **Coord[*y*].SyncOps** for ($0 <= y < x$). Therefore, if you wish to change the value of one or more **Coord[*x*].SyncOps** elements, you should issue a **save** command and reset the Power PMAC before attempting to use any of the synchronous assignment buffers.

Coord[*x*].Ta

Description: Acceleration time for blended moves

Range: Non-negative floating-point

Units: Milliseconds

Default: 100.0

Legacy I-variable alias: I*x*87

Coord[*x*].Ta sets the time for commanded acceleration for programmed **linear** or **circle**-mode moves in the coordinate system, in units of milliseconds. This time is used for the initial acceleration from stop at the beginning of a sequence of blended moves, and for blending between moves in the sequence. (The final deceleration to a stop at the end of the sequence is controlled by **Coord[*x*].Td**.)

If **Coord[*x*].Ts** is less than **Coord[*x*].Ta**, the time used for these accelerations will be **Coord[*x*].Ta** plus **Coord[*x*].Ts**. If **Coord[*x*].Ts** is greater than **Coord[*x*].Ta**, the time used for these accelerations will be 2***Coord[*x*].Ts**. If both **Coord[*x*].Ta** and **Coord[*x*].Ts** are 0.0, there will be no commanded acceleration profile, and there will be step changes in the commanded velocity profile. However, if segmentation is enabled (**Coord[*x*].SegMoveTime** > 0), the segmentation process will smooth this profile slightly.

A **ta*{data}*** command in a motion program executed by this coordinate system will automatically set this parameter to the value of ***{data}***.

For **linear** mode moves with segmentation disabled (**Coord[*x*].SegMoveTime** = 0), or for **linear** and **circle** mode moves with the special lookahead buffer enabled (**Coord[*x*].LHDistance** > 0), if the acceleration requested of a motor from the motion program given these parameters exceeds its maximum permitted acceleration as specified by **Motor[*x*].InvAmax**, the acceleration time is increased so that the limit is not violated. Note that the time for all motors in the coordinate system is increased to maintain full coordination.

Coord[*x*].Td

Description: Final deceleration time for blended moves

Range: Non-negative floating-point

Units: Milliseconds

Default: 100.0

Legacy I-variable alias: I*x*80

Coord[*x*].Td sets the time for commanded deceleration for programmed **linear** or **circle**-mode moves in the coordinate system, in units of milliseconds. This time is used for the final deceleration to a stop at the end of a blended sequence. The initial acceleration from stop at the beginning of a sequence of blended moves, and for blending between moves in the sequence, is controlled by **Coord[*x*].Ta**.)

If **Coord[*x*].Ts** is less than **Coord[*x*].Td**, the time used for these final decelerations will be **Coord[*x*].Td** plus **Coord[*x*].Ts**. If **Coord[*x*].Ts** is greater than **Coord[*x*].Td**, the time used for these final decelerations will be 2*__Coord[*x*].Ts__. If both **Coord[*x*].Td** and **Coord[*x*].Ts** are 0.0, there will be no commanded final deceleration profile, and there will be step changes in the commanded velocity profile. However, if segmentation is enabled (**Coord[*x*].SegMoveTime** > 0), the segmentation process will smooth this profile slightly.

Either a **td*{data}*** command or a **ta*{data}*** command in a motion program executed by this coordinate system will automatically set this parameter to the value of ***{data}***. If separate acceleration and deceleration times are desired, the **td*{data}*** command must be used after the **ta*{data}*** command.

If **Coord[*x*].CornerAccel** > 0.0 or **Coord[*x*].CornerError** > 0.0 to specify automatic calculation of corner blend times, **Coord[*x*].Td** acts as the minimum time that can be used at these corners.

For **linear** mode moves with segmentation disabled (**Coord[*x*].SegMoveTime** = 0), or for **linear** and **circle** mode moves with the special lookahead buffer enabled (**Coord[*x*].LHDistance** > 0), if the final deceleration requested of a motor from the motion program given these parameters exceeds its maximum permitted final deceleration as specified by **Motor[*x*].InvDmax**, the final deceleration time is increased so that the limit is not violated. Note that the time for all motors in the coordinate system is increased to maintain full coordination.

Coord[x].TimeBaseSlew

Description: Time-base slew rate

Range: Floating-point

Units: Milliseconds per servo cycle

Default: 0.00001

Legacy I-variable alias: I*x*94

Coord[*x*].TimeBaseSlew controls the rate of change of the time base for the coordinate system. If there is a change in the value of the register addressed by **Coord[*x*].pDesTimeBase**, the actual time base value will increment ("slew") towards this "desired" time-base value at a rate determined by **Coord[*x*].TimeBaseSlew**.

That is, each servo cycle, the value in **Coord[*x*].TimeBase**, which is used to increment the numerical time value (in milliseconds) for the coordinate system, has added to it or subtracted from it the value of **Coord[*x*].TimeBaseSlew** until it reaches the value of the desired time base.

The "real-time" time base value (%100) is determined by the global data structure element **Sys.ServoPeriod**, which should be set (with units of milliseconds) to the true period of the servo clock cycle as determined by the Servo or MACRO ASIC that controls the system's clock frequencies. The time (in servo cycles) for a change of 100% to occur can be calculated as:

$$100\%SlewTime(servo\_cycles) = \frac{Sys.ServoPeriod}{Coord[x].TimeBaseSlew}$$

Similarly, the time (in milliseconds) for this change to occur can be calculated as:

$$100\%SlewTime(\text{sec}) = \frac{Sys.ServoPeriod^2}{Coord[x].TimeBaseSlew}$$

Inverting this relationship, the required value of this element for a given slew time is:

$$Coord[x].TimeBaseSlew = \frac{Sys.ServoPeriod^2}{100\%SlewTime(\text{sec})}$$

This control of the rate of change of the actual time base used in the interpolation calculations permits step changes in the desired time-base value without a resulting step change in the velocity of axes using the time base.

If **Coord[*x*].TimeBaseSlew** is set to a positive value, the coordinate system time-base value will change at a rate equal to the magnitude of this value each servo cycle, regardless of the resulting acceleration values for the motors in the coordinate system.

If **Coord[*x*].TimeBaseSlew** is set to a negative value, the rate of change coordinate time-base value may be further limited if changing at a rate equal to the magnitude of this value would cause the acceleration of any motor in the coordinate system to exceed its **Motor[*x*].InvAmax**

limit, or the deceleration of any motor in the coordinate system to exceed its **Motor[*x*].InvDmax** limit. If the trajectory was originally specified not to exceed these constraints at 100% time base, then changes within the +/-100% range with **TimeBaseSlew** less than zero will not exceed these constraints.

If **Coord[*x*].TimeBaseSlew** is set exactly to 0.0, no changes in the active time-base value are possible.

Coord[*x*].Tm

Description: Time/feedrate for blended moves

Range: Floating-point

Units: Milliseconds (if >= 0) or user velocity units (if < 0)

Default: -1.0 (feedrate mode, 1.0 axis unit / time unit)

Legacy I-variable alias: I*x*89

Coord[*x*].Tm sets the time or vector speed ("feedrate") for **`linear`** or **`circle`**-mode moves in the coordinate system. If it is greater than zero, it specifies the move time in milliseconds, and the speed will be whatever is required to complete the move in the specified time. If it is less than zero, its magnitude specifies the vector speed for moves, with velocity units of vector axis units per coordinate-system time units as set by **Coord[*x*].Feedtime**. The move time will be whatever is required to complete the move at the specified speed.

A **`tm`*`{data}`** command in a motion program executed by this coordinate system will automatically set this parameter to the value of ***`{data}`***. An **`F`*`{data}`** command in a motion program executed by this coordinate system will automatically set this parameter to the negative of the value of ***`{data}`***.

If the magnitude of the speed requested of a motor from the motion program given this parameter exceeds its maximum permitted speed as specified by **Motor[*x*].MaxSpeed**, the move time is increased so that the limit is not violated. Note that the time for all motors in the coordinate system is increased to maintain full coordination.

Coord[*x*].TPCoords

Description: Target position reporting axis mask word

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Coord[*x*].TPCoords specifies which axes for the coordinate system will have their target positions or distances reported in response to the on-line **t** (target position) query command, if **Coord[*x*].TPSize** is set greater than zero to enable this buffering function. The values for all active axes in the coordinate system are always computed and placed into local D-variables for the coordinate system, but only the values for the specified axes are included in the response string.

Coord[*x*].TPCoords is a 32-bit value with one bit for each possible axis in the coordinate system. If a bit is set to 1, the target position for the matching axis is reported; if it is set to 0, the target position for the axis is not reported. The bits ***i*** and the axes they represent are shown in the following table:

Axis	*i*	Axis	*i*	Axis	*i*	Axis	*i*	Axis	*i*	Axis	*i*	Axis	*i*	Axis	*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 adds a value of 2^i to **Coord[*x*].TPCoords**.

Example

To enable target position reporting for the A, C, X, Y, and Z axes, bits 0, 2, 6, 7, and 8 would be set, so **Coord[*x*].TPCoords** would be equal to $1C5.

Coord[*x*].Tprog

Description: Subprogram number for T-code subroutines

Range: Positive integers

Units: Subprogram numbers

Default: 1002

Coord[*x*].Tprog controls which subprogram is called on the execution of a "T-code". When program execution encounters the **T*{data}*** syntax, program flow jumps to the subprogram with the number specified by **Coord[*x*].Tprog**, at the line jump label whose number is 1000 times the value of ***{data}***. For example, if **Coord[*x*].Tprog** is set to the default value of 1002, **T5** causes a jump to line jump label **N5000:** of **subprog 1002**.

This data structure element permits different coordinate systems to call different subroutines with the same T-code. However, in the default configuration, all coordinate systems use **subprog 1002** for their T-code subroutines.

Coord[*x*].TPSize

Description: Buffer size for move target position storage

Range: Non-negative integer

Units: Programmed moves

Default: 0 (no buffering)

Coord[*x*].TPSize specifies the size of the move "target position" buffer for the coordinate system. This buffer facilitates the reporting of the move-end (target) positions and "distance-to-go" for the presently executing move in applications such as CNC machining. Target positions can be reported from this buffer for the axes specified by **Coord[*x*].TPCoords**.

Power PMAC computes the target positions as the move equations are calculated; with features such as blending, splining, dynamic lookahead, and cutter radius compensation, this can occur many moves ahead of the executing move. This buffer permits the target positions to be stored until the moves are executed, so the user can easily obtain relevant data about the executing move. Note that this buffer does not affect how the moves are executed, and is not required for proper move execution.

The buffer must be sized large enough to store all programmed moves from move calculation time to move execution time. For simple blending or splining, it must be able to store at least 3 moves. For dynamic lookahead, where the buffer size is specified in the derived move segments, it must also be able to store a number of moves equal to the number of segments specified in **Coord[*x*].LHDistance** (or **Coord[*x*].LHSize** if reversal is used) divided by the minimum number of segments per move. For 2D cutter radius compensation, it must be able to store a number of moves equal to the moves buffered in **Coord[*x*].CCSize**, which includes the length of the interference check and the number of possible out-of-plane/zero-distance moves. 3D cutter radius compensation does not require any additional buffering.

All of these numbers of moves must be added to get the total number of moves to buffer in **Coord[*x*].TPSize**. Since the memory requirements for this buffer are relatively small, it is recommended to "oversize" this buffer by a substantial margin to ensure that the target positions can always be stored long enough to provide proper reporting.

If **Coord[*x*].TPSize** is set to the default value of 0, no target position buffering is performed.

Target positions from the buffer for the presently executing move can be obtained with the on-line **t** command, the buffered program **tread** command, or from the data structure elements **Coord[*x*].TPExec.Pos[*i*]**.

Power PMAC will reserve (288 * **TPSize**) bytes of data for the coordinate system's target position buffer. This buffer is part of the 16 MByte space allotted for all of the coordinate system buffers – cutter compensation, target position, and dynamic lookahead.

Example

An application is using blended linear and circle mode moves with dynamic lookahead of 200 segments and as little as 5 segments per move. 2D cutter radius compensation is used with up to 8 moves buffered in the cutter comp buffer. **Coord[*x*].TPSize** should be set to at least 3 + 200/5 + 8 = 51. Setting it to 75 provides good margin.

Coord[*x*].Ts

Description: S-curve accel/decel time for blended moves

Range: Non-negative floating-point

Units: Milliseconds

Default: 50.0

Legacy I-variable alias: I*sx*88

Coord[*x*].Ts sets the time for each half of the commanded "S-curve" acceleration for programmed **linear** or **circle**-mode moves in the coordinate system, in units of milliseconds. This time is used for the initial acceleration from stop at the beginning of a sequence of blended moves, for blending between moves in the sequence, and for the final deceleration to a stop at the end of the sequence.

If **Coord[*x*].Ts** is less than **Coord[*x*].Ta** (or **Coord[*x*].Td**), there will be a constant acceleration or deceleration time between the two halves of the "S" and the overall acceleration (or deceleration) time will be the sum of **Ta** (or **Td**) and **Ts**. If **Coord[*x*].Ts** is greater than **Coord[*x*].Ta** (or **Coord[*x*].Td**), the time used for these accelerations or decelerations will be 2***Coord[*x*].Ts**. If both **Coord[*x*].Ta** (or **Coord[*x*].Td**) and **Coord[*x*].Ts** are 0.0, there will be no commanded acceleration (or deceleration) profile, and there will be step changes in the commanded velocity profile. However, if segmentation is enabled (**Coord[*x*].SegMoveTime** > 0), the segmentation process will smooth this profile slightly.

A **ts*{data}*** command in a motion program executed by this coordinate system will automatically set this parameter to the value of ***{data}***.

For **linear** mode moves with segmentation disabled (**Coord[*x*].SegMoveTime** = 0), if the "jerk" (rate of change of acceleration) requested of a motor from the motion program given these parameters exceeds its maximum permitted jerk as specified by **Motor[*x*].InvJmax**, the S-curve time is increased so that the limit is not violated. Note that the time for all motors in the coordinate system is increased to maintain full coordination.

ECAT[*i*]. Saved Data Structure Elements

These data structure elements affect various different aspect of EtherCAT use and can be saved to non-volatile memory through the **save** command.

ECAT[*i*]. Network General Configuration Elements

These parameters can be used to configure settings that affect the entire EtherCAT network.

ECAT[*i*].AmpEnaTimeout

Description: Amplifier enable timeout wait period

Range: Non-negative integers

Units: Nanoseconds

Default: 0

ECAT[*i*].AmpEnaTimeout specifies the amount of time to wait (in nanoseconds) after issuing the amplifier enable command before determining whether the amplifier failed to enable. The default value of 0 will result in 500,000 nanoseconds (500 microseconds) to be used. This variable may help in situations where an amplifier needs a longer transition time before turning on. (See "transition state of the power drive system finite state automaton" as defined in IEC 61800-7-201).

(Starting in firmware version V2.0.2, this element is no longer used. Its functionality is performed by saved setup element **Motor[*x*].EcatAmpFaultLimit**.)

ECAT[*i*].DCRefBand

Description: Maximum tolerated drift from distributed clock reference

Range: Non-negative integers

Units: Nanoseconds

Default: 10,000

ECAT[*i*].DCRefBand specifies the magnitude of the maximum drift between Power PMAC's internal clock and the distributed clock reference before Power PMAC's clock frequency will be adjusted to reduce the drift. The adjustment will be made by automatically changing the value of the Power PMAC parameter that sets the frequency of the internal phase clock signal (from which the servo clock signal is derived). The present (signed) value of the drift can be found in status element **ECAT[*i*].DCClockDiff**.

(Starting in firmware version V2.0.2, this element is no longer used. Improved clock synchronization algorithms no longer require user settings.)

ECAT[*i*].DCRefMinus

Description: Negative compensation for drift from distributed clock reference

Range: Non-negative integers

Units: Nanoseconds

Default: 2

ECAT[*i*].DCRefMinus specifies the adjustment made to increase Power PMAC's interrupt frequency if the time drift between this clock and the distributed clock reference has exceeded the limit as set by **ECAT[*i*].DCRefBand** because Power PMAC's frequency was too low. The present (signed) value of the drift can be found in status element **ECAT[*i*].DCClockDiff**.

(Starting in firmware version V2.0.2, this element is no longer used. Improved clock synchronization algorithms no longer require user settings.)

ECAT[*i*].DCRefPlus

Description: Positive compensation for drift from distributed clock reference

Range: Non-negative integers

Units: Nanoseconds

Default: 2

ECAT[*i*].DCRefPlus specifies the adjustment made to reduce Power PMAC's interrupt frequency if the time drift between this clock and the distributed clock reference has exceeded the limit as set by **ECAT[*i*].DCRefBand** because Power PMAC's frequency was too high. The present (signed) value of the drift can be found in status element **ECAT[*i*].DCClockDiff**.

This parameter dictates the amount by which to decrease PMAC's phase interrupt rate if the clock drift of the ***i***th EtherCAT master is greater than **ECAT[*i*].DCRefBand**. The drift between the master and slave can be monitored with **ECAT[*i*].DCClockDiff**.

(Starting in firmware version V2.0.2, this element is no longer used. Improved clock synchronization algorithms no longer require user settings.)

ECAT[*i*].DCRefSlave

Description: Slave used for the distributed clocks reference

Range: 0 .. ***n***, ***n*** = number of slaves on EtherCAT network

Units: none

Default: 0

ECAT[*i*].DCRefSlave specifies the index of the slave device that provides the reference clock for the EtherCAT distributed clocks feature. If the specified slave device is either not found or not capable of providing the reference clock, Power PMAC will use the first slave device that can. **ECAT[*i*].DCRefSlave** is usually left at the default value of 0 to select the first compatible slave device on the network.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].DistrClocks

Description: EtherCAT distributed-clock feature enable

Range: 0 .. 1

Units: Boolean

Default: 0

ECAT[*i*].DistrClocks controls whether the "distributed-clocks" feature of the EtherCAT network with index ***i*** is enabled or not. If it is set to the default value of 0, this feature is disabled. If it is set to 1, the feature is enabled, with a period set by saved setup element **ECAT[*i*].DistrClocksCount**.

The distributed-clocks feature of the EtherCAT network forces the cycle clocks of all slave devices into proper synchronization based on a local reference clock in the first slave device. This synchronization includes compensation for transmission delays between devices, and proper phase locking to the Power PMAC master clock. For proper coordination of multiple axes, this feature should be enabled.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].DistrClocksCount

Description: Synchronization period for distributed clocks feature

Range: 0 .. 255

Units: EtherCAT cycle update periods

Default: 0

ECAT[*i*].DistrClocksCount specifies the period between consecutive synchronization operations that keep the slave devices on the EtherCAT network with index ***i*** properly locked to the Power PMAC master clock. This operation occurs every (**ECAT[*i*].DistrClocksCount** + 1) network update cycles. Because the network is updated every (**ECAT[*i*].ServoExtension** + 1) Power PMAC servo cycles, this synchronization period is every (**ECAT[*i*].DistrClocksCount** + 1) * (**ECAT[*i*].ServoExtension** + 1) Power PMAC servo cycles.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].InCount

Description: Number of slave input devices

Range: 0 .. 2048

Units: Devices

Default: 0

ECAT[*i*].InCount specifies the number of slave input devices that have been configured for operation on the EtherCAT network with index ***i***.

The element is only used if **Sys.EcatType** is set to 1.

ECAT[*i*].IOCount

Description: Number of slave I/O devices configured for cyclic operation

Range: 0 .. 2048

Units: Devices

Default: 0

ECAT[*i*].IOCount specifies the number of slave input/output devices that have been configured for cyclic operation on the EtherCAT network with index ***i***.

In operation, cyclic data transfers will be active for slave I/O devices configured in structures **ECAT[*i*].IO[*j*].** with index values ***j*** from 0 through **ECAT[*i*].IOCount** – 1. These devices must have valid values in elements **ECAT[*i*].IO[*j*].Slave**, **ECAT[*i*].IO[*j*].Index**, **ECAT[*i*].IO[*j*].SubIndex**, **ECAT[*i*].IO[*j*].BitLength**, and **ECAT[*i*].IO[*j*].Input**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPIOCount

Description: Number of slave I/O devices configured for background operation

Range: 0 .. 2048

Units: Units

Default: 0

ECAT[*i*].LPIOCount specifies the number of slave input/output devices that have been configured for background (low-priority) operation on the EtherCAT network with index ***i***.

In operation, background data transfers will be active for slave I/O devices configured in structures **ECAT[*i*].LPIO[*j*].** with index values ***j*** from 0 through **ECAT[*i*].LPIOCount** - 1, with valid values in elements **ECAT[*i*].LPIO[*j*].Slave**, **ECAT[*i*].LPIO[*j*].Index**, **ECAT[*i*].LPIO[*j*].SubIndex**, **ECAT[*i*].LPIO[*j*].BitLength**, and **ECAT[*i*].LPIO[*j*].Input**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPnotLRW

Description: Combined low-priority read/write disable control

Range: 0 .. 1

Units: Boolean

Default: 0

ECAT[*i*].LPnotLRW controls whether combined read/write operations for process data I/O in background (low-priority) devices are disabled or not for the EtherCAT network with index ***i***. If it is set to the default value of 0, combined read/write operations using the EtherCAT LRW command are enabled. If it is set to 1, these combined operations are disabled, and separate operations must be used (i.e. LRD for read operations and LRW for write operations).

Some EtherCAT slave devices (such as those made with the Hilscher slave IC) are not compatible with combined read/write operations. If any of these devices are on the network, **ECAT[*i*].LPnotLRW** must be set to 1.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPStateCheck

Description: EtherCAT low-priority state-check enable

Range: 0 .. 1

Units: Boolean

Default: 0

ECAT[*i*].LPStateCheck controls whether the "state-check" feature for low-priority devices of the EtherCAT network with index ***i*** is enabled or not. If it is set to the default value of 0, this feature is disabled. If it is set to 1, the feature is enabled, with a period set by saved setup element **ECAT[*i*].LPStateCheckCount**.

This state-check feature of the EtherCAT network periodically monitors the state of the low-priority devices on the network. The results are reported in status elements **ECAT[*i*].LPDomainState**, and **ECAT[*i*].LPOutputDomainState**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPStateCheckCount

Description: Synchronization period for low-priority state-check feature

Range: 0 .. 255

Units: EtherCAT cycle update periods

Default: 0

ECAT[*i*].LPStateCheckCount specifies the period between consecutive state checks for low-priority devices of the EtherCAT network with index ***i***. This operation occurs every (**ECAT[*i*].LPStateCheckCount** + 1) network update cycles. Because the network is updated every (**ECAT[*i*].ServoExtension** + 1) Power PMAC servo cycles, this synchronization period is every (**ECAT[*i*].LPStateCheckCount** + 1) * (**ECAT[*i*].ServoExtension** + 1) Power PMAC servo cycles.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].OutCount

Description: Number of slave output devices

Range: 0 .. 2048

Units: Devices

Default: 0

ECAT[*i*].OutCount specifies the number of slave input devices that have been configured for operation on the EtherCAT network with index ***i***.

The element is only used if **Sys.EcatType** is set to 1.

ECAT[*i*].RTnotLRW

Description: Combined real-time read/write disable control

Range: 0 .. 1

Units: Boolean

Default: 0

ECAT[*i*].RTnotLRW controls whether combined read/write operations for process data I/O in cyclic (real-time) devices are disabled or not for the EtherCAT network with index ***i***. If it is set to the default value of 0, combined read/write operations using the EtherCAT LRW command are enabled. If it is set to 1, these combined operations are disabled, and separate operations must be used (i.e. LRD for read operations and LRW for write operations).

Some EtherCAT slave devices (such as those made with the Hilscher slave IC) are not compatible with combined read/write operations. If any of these devices are on the network, **ECAT[*i*].RTnotLRW** must be set to 1.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].RTStateCheck

Description: EtherCAT real-time state-check enable

Range: 0 .. 1

Units: Boolean

Default: 0

ECAT[*i*].RTStateCheck controls whether the "state-check" feature for real-time devices of the EtherCAT network with index ***i*** is enabled or not. If it is set to the default value of 0, this feature is disabled. If it is set to 1, the feature is enabled, with a period set by saved setup element **ECAT[*i*].RTStateCheckCount**.

This state-check feature of the EtherCAT network periodically monitors the state of the master, domain, and slave devices on the network. The results are reported in status elements **ECAT[*i*].MasterState**, **ECAT[*i*].RTDomainState**, **ECAT[*i*].RTOutputDomainState**, and **ECAT[*i*].Slave[*j*].State**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].RTStateCheckCount

Description: Synchronization period for real-time state-check feature

Range: 0 .. 255

Units: EtherCAT cycle update periods

Default: 0

ECAT[*i*].RTStateCheckCount specifies the period between consecutive state checks for real-time devices of the EtherCAT network with index ***i***. This operation occurs every (**ECAT[*i*].RTStateCheckCount** + 1) network update cycles. Because the network is updated every (**ECAT[*i*].ServoExtension** + 1) Power PMAC servo cycles, this synchronization period is every (**ECAT[*i*].RTStateCheckCount** + 1) * (**ECAT[*i*].ServoExtension** + 1) Power PMAC servo cycles.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].ServoExtension

Description: EtherCAT cyclic update period extension

Range: 0 .. 255

Units: Power PMAC servo cycles

Default: 0

ECAT[*i*].ServoExtension specifies the cyclic update period on the EtherCAT network with index ***i***. It is updated every (**ECAT[*i*].ServoExtension** + 1) Power PMAC servo interrupt cycles.

Note that this period can be controlled independently from each motor's servo update period extension value **Motor[*x*].Stime**, which sets that motor's update period at (**Motor[*x*].Stime** + 1) servo interrupt periods. However, generally these settings should be the same in an application, so Power PMAC computes new data for the network once and only once per network update cycle.

ECAT[*i*].SlaveCount

Description: Number of slave servo devices configured for cyclic operation

Range: 0 .. 512

Units: Units

Default: 0

ECAT[*i*].SlaveCount specifies the number of slave servo devices that have been configured for cyclic operation on the EtherCAT network with index ***i***. It is typically set automatically during the execution of the **`ecat assign`** and **`ecat config`** commands. The user should be very careful in setting this element directly, as an improper setting could create defective network communications.

In operation, cyclic data transfers will be active for slave servo devices configured in structures **ECAT[*i*].Slave[*j*].** with index values ***j*** from 0 through **ECAT[*i*].SlaveCount** – 1. These devices must have valid values in elements **ECAT[*i*].Slave[*j*].VendorID**, **ECAT[*i*].Slave[*j*].ProductCode**, **ECAT[*i*].Slave[*j*].Position**, and **ECAT[*i*].Slave[*j*].Alias**, and **ECAT[*i*].Slave[*j*].Enable** must be set to 1.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*]. Cyclic I/O Configuration Elements

The following structures are related to configuring I/O devices for use in the High Priority interrupt. These devices fall into a category different from Slave amplifiers.

ECAT[*i*].IO[*k*].BitLength

Description: EtherCAT cyclic I/O bank source slave device PDO number of bits

Range: 0, 1, 2, 3, 4, 5, 6, 7, 8, 16, 32, or integers greater than 32.

Units: Bits

Default: 0

ECAT[*i*].IO[*k*].BitLength specifies the number of bits transferred to or from the process data object (PDO) on the slave device specified by **ECAT[*i*].IO[*k*].Slave** that is the source for the cyclic inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***.

If this parameter exceeds 32, the I/O data will not be stored in **ECAT[*i*].IO[*k*].Data**; it will instead be stored in **ECAT[*i*].IOBuffer[*n*]**.

ECAT[*i*].IO[*k*].BitPosition

Description: Transfer bit number for slave device PDO

Range: 0 … 31

Units: Bits

Default: 0

If **ECAT[*i*].IO[*k*].BitLength** is set to 1, **ECAT[*i*].IO[*k*].BitPosition** specifies which bit of the process data object (PDO) word to read or write on the slave device specified by **ECAT[*i*].IO[*k*].Slave** that is the source for the cyclic inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***.

ECAT[*i*].IO[*k*].Index

Description: EtherCAT cyclic I/O bank source slave device PDO index

Range: $0 ... $FFFF

Units: Enumeration

Default: $0

ECAT[*i*].IO[*k*].Index specifies the process data object (PDO) index number on the slave device specified by **ECAT[*i*].IO[*k*].Slave** that is the source for the cyclic inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***.

ECAT[*i*].IO[*k*].Input

Description: EtherCAT cyclic I/O bank source slave device PDO direction

Range: 0 ... 1

Units: Boolean

Default: 0

ECAT[*i*].IO[*k*].Input specifies direction of the data transfer for the cyclic inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***. If it is set to 1, this I/O bank comprises outputs. If it is set to 0, this I/O bank comprises inputs.

ECAT[*i*].IO[*k*].Slave

Description: EtherCAT cyclic I/O bank source slave device number

Range: 0 ... 255

Units: Enumeration

Default: 0

ECAT[*i*].IO[*k*].Slave specifies the data structure index value ***j*** of the slave device that is the source for the cyclic inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***. That is, these I/O points will be located on the device assigned to data structure **ECAT[*i*].Slave[*j*]**. This data structure for the device must be properly configured and the device enabled. Note that the I/O bank index value ***k*** does not have to be the same as the slave device index ***j***.

ECAT[*i*].IO[*k*].SubIndex

Description: EtherCAT cyclic I/O bank source slave device PDO subindex

Range: $0 ... $FFFF

Units: Enumeration

Default: $0

ECAT[*i*].IO[*k*].SubIndex specifies the process data object (PDO) subindex number on the slave device specified by **ECAT[*i*].IO[*k*].Slave** that is the source for the cyclic inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***.

ECAT[*i*]. Low-Priority I/O Module Configuration Elements

The following structures configure Slave I/O to be used in background (i.e. in the Low-Priority EtherCAT interrupt). These structures are only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPIO[*k*].BitLength

Description: EtherCAT background I/O bank source slave device PDO number of bits

Range: 0, 1 ... 8, 16, 32, 64

Units: Bits

Default: 0

ECAT[*i*].LPIO[*k*].BitLength specifies the number of bits transferred to or from the process data object (PDO) on the slave device specified by **ECAT[*i*].LPIO[*k*].Slave** that is the source for the background (low-priority) inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPIO[*k*].BitPosition

Description: Transfer bit number for background I/O slave device PDO

Range: 0 … 31

Units: Bits

Default: 0

If **ECAT[*i*].LPIO[*k*].BitLength** is set to 1, **ECAT[*i*].LPIO[*k*].BitPosition** specifies which bit of the process data object (PDO) word to read or write on the slave device specified by **ECAT[*i*].LPIO[*k*].Slave** that is the source for the cyclic inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPIO[*k*].Index

Description: EtherCAT background I/O bank source slave device PDO index

Range: $0 … $FFFF

Units: Enumeration

Default: $0

ECAT[*i*].LPIO[*k*].Index specifies the process data object (PDO) index number on the slave device specified by **ECAT[*i*].LPIO[*k*].Slave** that is the source for the background (low-priority) inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPIO[*k*].Input

Description: EtherCAT background I/O bank source slave device PDO direction

Range: 0 ... 1

Units: Boolean

Default: 0

ECAT[*i*].LPIO[*k*].Input specifies direction of the data transfer for the background (low-priority) inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***. If it is set to 1, this I/O bank comprises outputs. If it is set to 0, this I/O bank comprises inputs.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPIO[*k*].Slave

Description: EtherCAT background I/O bank source slave device number

Range: 0 ... 255

Units: Enumeration

Default: 0

ECAT[*i*].LPIO[*k*].Slave specifies the data structure index value ***j*** of the slave device that is the source for the background (low-priority) inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***. That is, these I/O points will be located on the device assigned to data structure **ECAT[*i*].Slave[*j*]**. This data structure for the device must be properly configured and the device enabled. Note that the I/O bank index value ***k*** does not have to be the same as the slave device index ***j***.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPIO[*k*].SubIndex

Description: EtherCAT background I/O bank source slave device PDO subindex

Range: $0 ... $FFFF

Units: Enumeration

Default: $0

ECAT[*i*].LPIO[*k*].SubIndex specifies the process data object (PDO) subindex number on the slave device specified by **ECAT[*i*].LPIO[*k*].Slave** that is the source for the background (low-priority) inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*] Slave Configuration Elements

Each slave device on an EtherCAT network must have a properly configured structure before it can be used in operation. The elements in this section comprise that structure.

ECAT[*i*].Slave[*j*].Alias

Description: EtherCAT slave device location alias value

Range: 0 ... 65,535

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].Alias specifies the location alias value for the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***. This number allows the slave device to have a unique numerical identifier regardless of its physical position in the EtherCAT network. This number for the device can be found using the **`ecat slaves`** command. It can automatically be assigned to this element with the **`ecat assign`** or **`ecat config`** command. It must be set to a valid number before the device can be enabled for network communications.

The alias value in the slave device itself is set either with hardware switches or through the **`ecat alias`** command and stored in non-volatile memory, depending on the device. The value of **ECAT[*i*].Slave[*j*].Alias** in the Power PMAC must match the value set in the device.

See also **ECAT[*i*].Slave[*j*].Position**, which is used in conjunction with this parameter.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].AssignActivate

Description: EtherCAT slave device vendor-specific element

Range: Device-specific

Units: Device-specific

Default: 0

ECAT[*i*].Slave[*j*].AssignActivate is a vendor-specific value and should be taken from the XML device description file for the slave device. It is often not necessary for the basic operation of a slave device.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].Enable

Description: EtherCAT slave device enable control

Range: 0 ... 1

Units: Boolean

Default: 0

ECAT[*i*].Slave[*j*].Enable controls whether the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i*** is enabled for cyclic commands or not. If it is set to the default value of 0, the device is disabled. If it is set to 1, the device is enabled.

Configuration elements **ECAT[*i*].Slave[*j*].VendorID**, **ECAT[*i*].Slave[*j*].ProductCode**, **ECAT[*i*].Slave[*j*].Alias**, and **ECAT[*i*].Slave[*j*].Position** must be set properly before the device can be enabled. The Power PMAC IDE's System Setup tool can aid in setting these structures properly and quickly.

ECAT[*i*].Slave[*j*].Position

Description: EtherCAT slave device location position value

Range: 0 ... 65,535

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].Position specifies the absolute or relative position of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***. If **ECAT[*i*].Slave[*j*].Alias** is set to 0, then

ECAT[*i*].Slave[*j*].Position specifies the absolute position on the network. If **ECAT[*i*].Slave[*j*].Alias** is set to a value greater than 0, then **ECAT[*i*].Slave[*j*].Position** specifies the position on the network relative to the first device with this same alias value.

The absolute position of a slave device on the network is the first value reported back for the device in response to an **`ecat slaves`** query command. The alias value and the relative position (if the alias value is non-zero) of a slave device are the fourth and fifth values reported back for the device, separated by a colon (:) character.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].ProductCode

Description: EtherCAT slave device product code number

Range: $0 ... $FFFFFFFF

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].ProductCode specifies the product code number for the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***. This number for the device can be found using the **`ecat slaves`** command. It can automatically be assigned to this element with the **`ecat assign`** or **`ecat config`** command. It must be set to a valid number before the device can be enabled for network communications.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].Sync0Cycle

Description: SYNC0 cycle time [nanoseconds] used for distributed clocks.

Range: Device-specific

Units: Device-specific

Default: 0

ECAT[*i*].Slave[*j*].Sync0Cycle is the SYNC0 cycle time [nanoseconds] used for distributed clocks.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].Sync0Shift

Description: SYNC0 shift time [nanoseconds] used for distributed clocks.

Range: Device-specific

Units: Device-specific

Default: 0

ECAT[*i*].Slave[*j*].Sync0Shift is the SYNC0 shift time [nanoseconds] used for distributed clocks.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].Sync1Cycle

Description: SYNC1 cycle time [nanoseconds] used for distributed clocks.

Range: Device-specific

Units: Device-specific

Default: 0

ECAT[*i*].Slave[*j*].Sync1Cycle is the SYNC1 cycle time [nanoseconds] used for distributed clocks.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].Sync1Shift

Description: SYNC1 shift time [nanoseconds] used for distributed clocks.

Range: Device-specific

Units: Device-specific

Default: 0

ECAT[*i*].Slave[*j*].Sync1Shift is the SYNC1 shift time [nanoseconds] used for distributed clocks.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].VendorID

Description: EtherCAT slave device vendor identification number

Range: $0 ... $FFFFFFFF

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].VendorID specifies the vendor identification number for the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***. This number for the device can be found using the **`ecat slaves`** command. It can automatically be assigned to this element with the **`ecat assign`** or **`ecat config`** command. It must be set to a valid number before the device can be enabled for network communications.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*]. Slave Process Data Object Configuration Elements

ECAT[*i*].Slave[*j*].PDO[*k*].BitLength

Description: Slave device process data object length in bits

Range: $0 ... $20 (32)

Units: Bits

Default: 0

ECAT[*i*].Slave[*j*].PDO[*k*].BitLength specifies the number of bits in the process data object (PDO) of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i*** that will be transferred to or from the Power PMAC PDO image with index ***k***. The value of this element must correspond to **ECAT[*i*].Slave[*j*].PDO[*k*].Index** and **ECAT[*i*].Slave[*j*].PDO[*k*].SubIndex**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].PDO[*k*].Index

Description: Slave device process data object index value

Range: $0 ... $FFFF

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].PDO[*k*].Index specifies the index number of the process data object (PDO) of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i*** that is to be assigned to the Power PMAC PDO image with index ***k***. There can be up to 64 PDOs per slave.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].PDO[*k*].Input

Description: Slave device process data object direction control

Range: 0 ... 1

Units: Boolean

Default: 0

ECAT[*i*].Slave[*j*].PDO[*k*].Input specifies the "direction" of the process data object (PDO) of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i*** assigned to the Power PMAC PDO image with index ***k***. If it is set to 0, the PDO is an output PDO. If it is set to 1, it is an input PDO. The value of this element must correspond to **ECAT[*i*].Slave[*j*].PDO[*k*].Index** and **ECAT[*i*].Slave[*j*].PDO[*k*].SubIndex**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].PDO[*k*].SubIndex

Description: Slave device process data object subindex value

Range: $0 ... $FFFF

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].PDO[*k*].SubIndex specifies the subindex number of the process data object (PDO) of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i*** that is to be assigned to the Power PMAC PDO image with index ***k***. The value of this element must correspond to **ECAT[*i*].Slave[*j*].PDO[*k*].Index**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*]. Slave PDO Mapping Configuration Elements

ECAT[*i*].Slave[*j*].PDOMapping[*m*].Index

Description: Slave device process data object mapping index value

Range: $0 ... $FFFF

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].PDOMapping[*m*].Index specifies the index number for the mapping of the process data object assigned to the Power PMAC PDO mapping with index ***m*** of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***. Typically, PDO mappings for inputs are in the range of $1600 ... $17FF, and for outputs in the range of $1A00 ... $1BFF. The proper value for a slave device is supplied by the slave's vendor.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].PDOMapping[*m*].PDOCount

Description: Slave device number of process data objects in a mapping

Range: $0 ... $40 (64)

Units: Objects

Default: 0

ECAT[*i*].Slave[*j*].PDOMapping[*k*].PDOCount specifies the number of process data objects (PDOs) in the PDO mapping with index ***m*** of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].PDOMapping[*m*].pPDO

Description: Slave device process data object mapping address of first PDO

Range: Valid **ECAT[*i*].Slave[*j*].PDO[*k*].a** values

Units: Objects

Default: 0

ECAT[*i*].Slave[*j*].PDOMapping[*m*].pPDO specifies the address of the first process data object (PDO) to be assigned to the PDO mapping with index ***m*** of the slave device of index ***j*** on the EtherCAT network with index ***i***. The address is typically specified with the data structure name for the PDO with the "**.a**" suffix. It is generally not necessary to know the numerical value for this address. Other PDOs assigned to this mapping must have index values ***k*** consecutively numbered to the PDO specified by this address.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*]. Slave Synchronization Manager Configuration Elements

ECAT[*i*].Slave[*j*].SyncManager[*n*].Dir

Description: Slave device synchronization manager direction control

Range: 0 … 2

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].SyncManager[*n*].Dir specifies the direction of the synchronization manager with index ***n*** of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***. A value of 1 is used for output and a value of 2 is used for input. That is, SyncManagers used for syncing input PDOs should use a value of 2, and a value of 1 if syncing for output PDOs.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].SyncManager[*n*].Index

Description: Slave device synchronization manager index value

Range: 0 ... 8

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].SyncManager[*n*].Index specifies the index number for the synchronization manager with index ***n*** of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***. Typically a value of 2 is used for cyclic output mappings and a value of 3 is used for cyclic input mappings. Typically, any slave making use of foreground data transfer will have two SyncManagers: one assigned as input (for input PDOs) and one assigned as output (for output PDOs).

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].SyncManager[*n*].PDOMappingCount

Description: Slave device synchronization manager number of PDO mappings

Range: 0 … 16

Units: Mappings

Default: 0

ECAT[*i*].Slave[*j*].SyncManager[*n*].Index specifies the number of process data object (PDO) mappings assigned to the synchronization manager with index ***n*** of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].SyncManager[*n*].pPDOMapping

Description: Slave device synchronization manager address of first PDO mapping

Range: Valid **ECAT[*i*].Slave[*j*].PDOMapping[*m*].a** values

Units: Objects

Default: 0

ECAT[*i*].Slave[*j*].SyncManager[*n*].pPDOMapping specifies the address of the first process data object (PDO) mapping to be assigned to the synchronization manager with index ***n*** of the slave device of index ***j*** on the EtherCAT network with index ***i***. The address is typically specified with the data structure name for the PDO with the "**.a**" suffix. It is generally not necessary to know the numerical value for this address. Other PDO mappings assigned to this manager must have index values ***m*** consecutively numbered to the PDO mapping specified by this address.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].SyncManager[*n*].WatchdogMode

Description: Slave device synchronization manager watchdog mode

Range: 0 ... 2

Units: Enumeration

Default: 0

ECAT[*i*].Slave[*j*].SyncManager[*n*].Watchdog specifies the operational mode of the watchdog timer of the synchronization manager with index ***n*** of the slave device assigned to slave index ***j*** on the EtherCAT network with index ***i***. A value of 0 specifies the default state for the synchronization manager; a value of 1 forces the enabling of the watchdog; a value of 2 forces the disabling of the watchdog.

This element is only used if **Sys.EcatType** is set to 0.

EncTable[*n*]. Saved Data Structure Elements

The **EncTable[*n*].** data structure implements the "encoder conversion table", which pre-processes feedback and master position data every servo cycle, preparing the raw data for use by the servo algorithms. Please refer to the chapter *Setting Up the Encoder Conversion Table* in the Power PMAC User's Manual for a comprehensive overview.

Each index value for **EncTable[*n*]** specifies an "entry", which processes some data to provide a single "result" value that is typically intended to be used by a motor each servo cycle as a feedback or master position value. Index values can go from 0 to **Sys.MaxEncoders** – 1, with system constant **Sys.MaxEncoders** equal to 768 for most Power PMAC configurations.

EncTable[*n*].CosBias

Description: Encoder table entry cosine-term offset

Range: -32,768 .. 32,767

Units: Units of 16-bit ADC

Default: 0

EncTable[*n*].CosBias specifies a correction factor used to compensate for offsets in the "cosine" input for certain types of feedback. It is a signed 16-bit integer value. It and **EncTable[*n*].SinBias** share a register with the 32-bit setup element **EncTable[*n*].MaxDelta**, which is used in other types of feedback.

Type = 0 (end of table): The value specified by **CosBias** is not used for this type.

Type = 1 (single-word read): The word for the **CosBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 2 (double-word read): The word for the **CosBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 3 (software 1/T encoder extension): The value specified by **CosBias** is not used for this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 4 (sinusoidal encoder arctangent extension): The value of **CosBias** specifies the offset that is added to the measured "cosine" term of the sinusoidal encoder. It should contain the value opposite of that which the matching A/D converter reports in **EncTable[*n*].Cos** when it should ideally be reporting a value of zero. Both the measured term and the bias term are treated as 16-bit values, even if the actual A/D converter has a different resolution. The measured data is left-justified, so if a 14-bit ADC is used, the data is found in the high 14 bits of the 16-bit word.

Type = 5 (four-byte read): The word for the **CosBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 6 (resolver direct conversion): The value of **CosBias** specifies the offset that is added to the measured "cosine" term of the resolver. It should contain the value opposite of that which the

matching A/D converter reports in **EncTable[*n*].Cos** when it should ideally be reporting a value of zero. Both the measured term and the bias term are treated as 16-bit values, even if the actual A/D converter has a different resolution. The measured data is left-justified, so if a 14-bit ADC is used, the data is found in the high 14 bits of the 16-bit word.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **CosBias** is not used for this type.

Type = 8 (addition of two sources): The word for the **CosBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 9 (subtraction of two sources): The word for the **CosBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 10 (triggered time base): The value specified by **CosBias** is not used for this type.

Type = 11 (floating-point read): The value specified by **CosBias** is not used for this type.

Type = 12 (single-word read with error check): The value specified by **CosBias** is not used for this type.

EncTable[*n*].CoverSerror

Description: Encoder table entry sine/cosine magnitude mismatch correction factor

Range: -32,768 .. 32,767

Units: 32,768 * fraction of magnitude ratio

Default: 0

EncTable[*i*].CoverSerror (Cosine-over-Sine-error) specifies a correction factor used to compensate for cosine versus sine magnitude errors in certain types of feedback. At the default value of 0, the action of all conversion methods is the same as in older firmware versions.

Type = 0 (end of table): The value specified by **CoverSerror** is not used for this type.

Type = 1 (single-word read): The value specified by **CoverSerror** is not used for this type.

Type = 2 (double-word read): The value specified by **CoverSerror** is not used for this type.

Type = 3 (software 1/T encoder extension): The value specified by **CoverSerror** is not used for this type.

Type = 4 (sinusoidal encoder arctangent extension): The value of **CoverSerror** specifies the (signed) fractional component of the factor that multiplies the measured "cosine" signal (from the IC channel's **Adc[1]** or **AdcEnc[1]** register) to correct for magnitude mismatch between the measured "sine" and "cosine" signals.

In operation, the value of **CoverSerror** is divided by 32,768 to put it in the fractional range of -1.0 to +0.9999, this fraction is added to 1.0, and the sum (in the range of 0.0 to 1.9999) multiplies the measured cosine signal value before the arctangent calculation is performed. Note that at the default value for **CoverSerror** of 0, the correction factor is 1.0, and no magnitude correction is performed.

This magnitude correction is performed after the offset corrections using **SinBias** and **CosBias**, and before the phase correction using **TanHalfPhi**.

Example: If the magnitude of the measured sine signal (**Adc[0]** or **AdcEnc[0]**) is 5% smaller than that of the measured cosine signal (**Adc[1]** or **AdcEnc[1]**), the cosine signal values should be multiplied by a factor of 0.95, and **CoverSerror**, which represents the fractional component of -0.05, should be set to -1638 (= -0.05 * 32,768).

Type = 5 (four-byte read): The value specified by **CoverSerror** is not used for this type.

Type = 6 (resolver direct conversion): The value specified by **CoverSerror** is not used for this type.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **CoverSerror** is not used for this type.

Type = 8 (addition of two sources): The value specified by **CoverSerror** is not used for this type.

Type = 9 (subtraction of two sources): The value specified by **CoverSerror** is not used for this type.

Type = 10 (triggered time base): The value specified by **CoverSerror** is not used for this type.

Type = 11 (floating-point read): The value specified by **CoverSerror** is not used for this type.

Type = 12 (single-word read with error check): The value specified by **CoverSerror** is not used for this type.

EncTable[*n*].EncBias

Description: Encoder table entry pre-integration offset

Range: -2^{31} .. $2^{31}-1$

Units: Units of source 32-bit register

Default: 0

EncTable[*n*].EncBias specifies a pre-integration offset if the entry is set up to integrate the source data (**index4** > 0, and **index2** < 32). Each servo cycle, the value of **EncBias** is added to the 32-bit source data before any shifting operations as specified by **index1** and **index2** are performed, and before the numerical integration as specified by **index4**. In this way, **EncBias** can act to compensate for biases such as analog offsets in the source data.

EncTable[*n*].EncBias shares a register with status element **EncTable[*n*].PrevDelta**, which is used in non-integrating entries to store the last change in source position.

Type = 0 (end of table): The value specified by **EncBias** is not used for this type.

Type = 1 (single-word read): The value specified by **EncBias** is added to the 32-bit source value if the entry is set up for integration.

Type = 2 (double-word read): The value specified by **EncBias** is added to the combined 32-bit source value if the entry is set up for integration.

Type = 3 (software 1/T encoder extension): The value specified by **EncBias** is not used for this type.

Type = 4 (software arctangent encoder extension): The value specified by **EncBias** is not used for this type.

Type = 5 (four-byte read): The value specified by **EncBias** is added to the combined 32-bit value if the entry is set up for integration.

Type = 6 (resolver direct conversion): The value specified by **EncBias** is not used for this type.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **EncBias** is not used for this type.

Type = 8 (addition of two sources): The value specified by **EncBias** is added to the combined 32-bit source value if the entry is set up for integration.

Type = 9 (subtraction of two sources): The value specified by **EncBias** is added to the combined 32-bit source value if the entry is set up for integration.

Type = 10 (triggered time base): The value specified by **EncBias** is not used for this type.

Type = 11 (floating-point read): The value specified by **EncBias** is added to the intermediate integer value converted from the source floating-point register if the entry is set up for integration.

Type = 12 (single-word read with error check): The value specified by **EncBias** is added to the 32-bit source value if the entry is set up for integration.

EncTable[*n*].index1

Description: Encoder table entry first conversion factor

Range: (Type specific, in 0 .. 255 range)

Units: (Type specific)

Default: Auto-configured based on hardware

EncTable[*n*].index1 specifies the first conversion factor used (if any) to process the raw data for this entry. Its function varies by conversion method, which is specified by **EncTable[*n*].type**.

Type = 0 (end of table): The value specified by **index1** is not used for this type.

Type = 1 (single-word read): The value of **index1** normally specifies the number of bits the 32-bit word from the source register is shifted left after it is first shifted right as specified by **index2**. The purpose of this operation is to leave the most significant bit of actual data from the source register in the MSB of the resulting value, eliminating possible "garbage data" and permitting proper handling of the rollover of source data. At the beginning of this operation, the LSB of actual data is typically found in the LSB of the 32-bit register, so **index1** is usually set to 32 minus the number of bits of actual data. For example, if there are 14 bits of real data, **index1** would be set to 18. Note that the table setup menu in the IDE asks you how many bits of actual data are used, and it computes this value from your response.

However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter", along with **index4**, which acts as an "exponent" term. The operation of the tracking filter is explained below, under **index2**. The tracking filter is seldom used for this conversion method, unless the source register is from an A/D converter.

Type = 2 (double-word read): The value of **index1** normally specifies the number of bits the 32-bit word assembled from the two source registers is shifted left after it is first shifted right as specified by **index2**. The purpose of this operation is to leave the most significant bit of actual data from the source registers in the MSB of the resulting value, eliminating possible "garbage data" and permitting proper handling of the rollover of source data. At the beginning of this operation, the LSB of actual data is typically found in the LSB of the 32-bit register, so **index1** is usually set to 32 minus the number of bits of actual data. For example, if there are 28 bits of real data, **index1** would be set to 4. Note that the table setup menu in the IDE asks you how many bits of actual data are used, and it computes this value from your response.

However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained below, under **index2**. The tracking filter is seldom used for this conversion method.

Type = 3 (software 1/T encoder extension): If the value of **index2** is less than 32, then the **index1** element is not a setup element for this type.

However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained below, under **index2**. The tracking filter is seldom used for this conversion method.

Type = 4 (sinusoidal encoder arctangent extension): If the value of **index2** is less than 32, then the **index1** element is not a setup element for this type.

However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained below, under **index2**. A tracking filter can be valuable to filter out noise in the analog readings of the encoder voltages.

Type = 5 (four-byte read): The value of **index1** normally specifies the number of bits the 32-bit word assembled from the four source registers is shifted left after it is first shifted right as specified by **index2**. The purpose of this operation is to leave the most significant bit of actual

data from the source registers in the MSB of the resulting value, eliminating possible "garbage data" and permitting proper handling of the rollover of source data. At the beginning of this operation, the LSB of actual data is typically found in the LSB of the 32-bit register, so **index1** is usually set to 32 minus the number of bits of actual data. For example, if there are 24 bits of real data, **index1** would be set to 8. Note that the table setup menu in the IDE asks you how many bits of actual data are used, and it computes this value from your response.

However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained below, under **index2**. The tracking filter is seldom used for this conversion method.

Type = 6 (resolver direct conversion): If the value of **index2** is less than 32, then the **index1** element is not a setup element for this type.

However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained below, under **index2**. A tracking filter is strongly recommended to filter out noise in the analog readings of the resolver windings.

Type = 7 (extended hardware sinusoidal interpolation): If the value of **index2** is less than 32, then the **index1** element is not a setup element for this type.However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained below, under **index2**. The tracking filter is seldom used for this conversion method, as the interpolator has typically done its own filtering in hardware.

Type = 8 (addition of two sources): The value of **index1** normally specifies the number of bits the 32-bit word from the sum of the source registers is shifted left after it is first shifted right as specified by **index2**. The purpose of this operation is to leave the most significant bit of actual data from the source register in the MSB of the resulting value, eliminating possible "garbage data" and permitting proper handling of the rollover of source data. At the beginning of this operation, the LSB of actual data is typically found in the LSB of the 32-bit register, so **index1** is usually set to 32 minus the number of bits of actual data. For example, if there are 18 bits of real data, **index1** would be set to 14. Note that the table setup menu in the IDE asks you how many bits of actual data are used, and it computes this value from your response.

However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained below, under **index2**. The tracking filter is seldom used for this conversion method, unless the source register is from an A/D converter.

Type = 9 (subtraction of two sources): The value of **index1** normally specifies the number of bits the 32-bit word from the sum of the source registers is shifted left after it is first shifted right as specified by **index2**. The purpose of this operation is to leave the most significant bit of actual data from the source register in the MSB of the resulting value, eliminating possible "garbage data" and permitting proper handling of the rollover of source data. At the beginning of this operation, the LSB of actual data is typically found in the LSB of the 32-bit register, so **index1** is usually set to 32 minus the number of bits of actual data. For example, if there are 18 bits of real data, **index1** would be set to 14. Note that the table setup menu in the IDE asks you how many bits of actual data are used, and it computes this value from your response.

However, if the value of **index2** is 32 or greater, then **index1** acts as the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained below, under **index2**. The

tracking filter is seldom used for this conversion method, unless the source register is from an A/D converter.

Type = 10 (triggered time base): In this temporary state, if **index1** is set to 0, the time base is "frozen" with no change in the output and no response to a trigger. If it is set to 2, the time base is "armed", with no change in the output, but looking for a trigger from a PMAC2-style IC at the address specified by **pEnc1**. If it is set to 3, the time base is "armed", with no change in the output, but looking for a trigger from a PMAC3-style IC at the address specified by **pEnc1**. When the trigger occurs, **index1** is automatically changed to 0 as the Type reverts to 3 or 1.

Type = 11 (floating-point read): The value specified by **index1** is used in the same manner as in Type 1 (q.v.)

Type = 12 (single-word read with error check): The value specified by **index1** is used in the same manner as in Type 1 (q.v.)

EncTable[*n*].index2

Description: Encoder table entry second conversion factor

Range: (Type specific, in 0 .. 255 range)

Units: (Type specific)

Default: Auto-configured based on hardware

EncTable[*n*].index2 specifies the second conversion factor used (if any) to process the raw data for this entry. Its function varies by conversion method, which is specified by **EncTable[*n*].type**.

Type = 0 (end of table): The value specified by **index2** is not used for this type.

Type = 1 (single-word read): If less than 32, the value of **index2** specifies the number of bits the 32-bit word from the source register is shifted right as an initial operation. The purpose of this operation is to leave the least significant bit of actual data from the source register in the LSB of the resulting value, eliminating possible "garbage data" in the low bits of the source register, so **index2** is usually set to bit number of the LSB of actual data in the source register. For example, if the LSB of actual data is in bit 8 of the source register, **index2** would be set to 8. Note that the table setup menu in the IDE asks you what the "starting bit number" is, and sets **index2** to the value of your response.

If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below. No shifting operations are done when the tracking filter is active. The tracking filter is seldom used for this conversion method, unless the source register is from an A/D converter.

Type = 2 (double-word read): If less than 32, the value of **index2** specifies the number of bits the 32-bit word assembled from the two source registers is shifted right as an initial operation. The purpose of this operation is to leave the least significant bit of actual data from the assembled word in the LSB of the resulting value, eliminating possible "garbage data" in the low bits of the

assembled 32-bit word, so **index2** is usually set to bit number of the LSB of actual data in the assembled word (this is a value 8 less than the location of the LSB in the first source 32-bit register). For example, if the LSB of actual data is in bit 4 of the assembled word, **index2** would be set to 4. Note that the table setup menu in the IDE asks you what the "starting bit number" is, and sets **index2** to the value of your response. The tracking filter is seldom used for this conversion method.

If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below. No shifting operations are done when the tracking filter is active.

Type = 3 (software 1/T encoder extension): If less than 32, the value specified by **index2** is not used for this type. If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below. The tracking filter is seldom used for this conversion method.

Type = 4 (sinusoidal encoder arctangent extension): If less than 32, the value specified by **index2** is not used for this type. If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below.

Type = 5 (four-byte read): If less than 32, the value of **index2** specifies the number of bits the 32-bit word assembled from the four source registers is shifted right as an initial operation. The purpose of this operation is to leave the least significant bit of actual data from the assembled word in the LSB of the resulting value, eliminating possible "garbage data" in the low bits of the assembled 32-bit word, so **index2** is usually set to bit number of the LSB of actual data in the assembled word (this is a value 8 less than the location of the LSB in the first source 32-bit register). For example, if the LSB of actual data is in bit 0 of the assembled word, **index2** would be set to 0. Note that the table setup menu in the IDE asks you what the "starting bit number" is, and sets **index2** to the value of your response. The tracking filter is seldom used for this conversion method.

If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below. No shifting operations are done when the tracking filter is active.

Type = 6 (resolver direct conversion): If less than 32, the value specified by **index2** is not used for this type. If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below.

Type = 7 (extended hardware sinusoidal interpolation): If less than 32, the value specified by **index2** is not used for this type. If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below. No shifting operations are done when the tracking filter is active. The tracking filter is seldom used for this conversion method, as the interpolator has already done filtering in hardware.

Type = 8 (addition of two sources): If less than 32, the value of **index2** specifies the number of bits the 32-bit sum of the two source registers is shifted right as an initial operation. The purpose of this operation is to leave the least significant bit of actual data from the source registers in the LSB of the resulting value, eliminating possible "garbage data" in the low bits of the source registers, so **index2** is usually set to bit number of the LSB of actual data in the source registers. For example, if the LSB of actual data is in bit 6 of the source register, **index2** would be set to 6.

If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below. No shifting operations are done when the tracking filter is active. The tracking filter is seldom used for this conversion method, unless the source register is from an A/D converter.

Type = 9 (subtraction of two sources): If less than 32, the value of **index2** specifies the number of bits the 32-bit difference of the two source registers is shifted right as an initial operation. The purpose of this operation is to leave the least significant bit of actual data from the source registers in the LSB of the resulting value, eliminating possible "garbage data" in the low bits of the source registers, so **index2** is usually set to bit number of the LSB of actual data in the source registers. For example, if the LSB of actual data is in bit 6 of the source register, **index2** would be set to 6.

If equal to 32 or greater, the value of **index2** specifies the proportional gain term *Kp* in a "tracking filter". In this case, the value of **index1** specifies the integral gain term *Ki* in the filter. The operation of the tracking filter is explained below. No shifting operations are done when the tracking filter is active. The tracking filter is seldom used for this conversion method, unless the source register is from an A/D converter.

Type = 10 (triggered time base): The value specified by by **index2** is not used while in this temporary "frozen" stage, but may be used in the Type 1 or Type 3 entry this reverts to as soon as the trigger occurs.

Type = 11 (floating-point read): The value specified by **index2** is used in the same manner as in Type 1 (q.v.)

Type = 12 (single-word read with error check): The value specified by **index2** is used in the same manner as in Type 1 (q.v.)

Use in Tracking Filter

The encoder conversion table's software tracking filter is a digital low-pass filter with an integrator. It is dynamically equivalent to the hardware tracking filters commonly used in tracking resolver-to-digital (R/D) converters, and is useful for reducing measurement noise without introducing steady-state error at constant velocity or position.

The equations executed for the tracking filter each servo cycle *k* are:

$$Err(k) = In(k) - Out(k-1)$$

$$Temp = \left(\frac{256 - index2}{256}\right) * Err(k)$$

$$Int(k) = Int(k-1) + \left(\frac{index1}{256 * 2^{index4}} \right) * Err(k)$$

$$Out(k) = Out(k-1) + Temp + Int(k)$$

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 - **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, but some may find it useful for filtering master position values.

If the integral gain term specified by **index1** is greater than 0, then the filter is a second-order filter. The natural frequency ω_n of this filter can be computed as:

$$\omega_n = \frac{1}{T_s} \sqrt{\frac{index1}{256 * 2^{index4}}}$$

where T_s is the servo update time in seconds (= **Sys.ServoPeriod** / 1000).

The damping ratio ς of this filter can be computed as:

$$\varsigma = \frac{256 - index2}{2\sqrt{256 * \frac{index1}{2^{index4}}}}$$

Working the other way, for a desired natural frequency ω_n and damping ratio ς of this filter, **index2** and **index1** can be computed using the following equations:

$$index2 = 256 - 512 * \varsigma * \omega * T_s$$

$$\frac{index1'}{2^{index4}} = 256 * \omega_n^2 * T_s^2$$

If the value for **index1** is too small, rounding to the nearest integer may yield inaccuarate results. It is recommended that if **index1'** is less than 64, 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.

EncTable[*n*].index3

Description: Encoder table entry third conversion factor

Range: (Type specific, in 0 .. 15 range)

Units: (Type specific)

Default: Auto-configured based on hardware

EncTable[*n*].index3 specifies the third conversion factor used (if any) to process the raw data for this entry. Its function varies by conversion method, which is specified by **EncTable[*n*].type**.

Type = 0 (end of table): The value specified by **index3** is not used for this type.

Type = 1 (single-word read): The value of **index3** specifies what derivative of the source data can be limited, and how it is limited. If **index3** is equal to 0, the first derivative of the source data (typically velocity) will be limited provided the **MaxDelta** element is greater than 0. In this case, if the limit is violated during a cycle, but was not violated during the previous cycle, Power PMAC will calculate a new result using the last cycle's change. If the limit is violated during a cycle, and was also violated during the previous cycle, it will use the **MaxDelta** value as the change for this cycle.

If **index3** is greater than 0, the second derivative of the source data (typically acceleration) will be limited provided the **MaxDelta** element is greater than 0. In this case, index3 specifies the number of consecutive cycles that if the acceleration limit is violated, Power PMAC will calculate a new result using that last valid cycle's change. If the limit is violated for more than this number of consecutive cycles, the source position will be used regardless of the limit.

Note, however, that if **index4** is set to a value greater than 0 to specify numerical integration of the source data, **index3** is not used.

Type = 2 (double-word read): The value specified by **index3** is used in the same manner as in Type 1 (q.v.)

Type = 3 (software 1/T encoder extension): The value specified by **index3** is not used for this type.

Type = 4 (sinusoidal encoder arctangent extension): The working value of **index3** must match the encoder decode control value for the ASIC channel (**Gate*n*[*i*].Chan[*j*].EncCtrl**), which must be a 3 or a 7 for the conversion to work. If the value of **index3** is set to 0, the entry will read from the ASIC in the next servo cycle and automatically set **index3** to the value read. It is strongly recommended that **index3** be set to 0 on initial setup of the entry, or if the decode control value in the ASIC is changed.

Type = 5 (four-byte read): The value specified by **index3** is used in the same manner as in Type 1 (q.v.)

Type = 6 (resolver direct conversion): The value specified by **index3** must be a 0 or 1, and specifies the direction sense of the conversion. Changing the value of **index3** changes which direction of rotation causes the result value to increase.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **index3** is used in the same manner as in Type 1 (q.v.)

Type = 8 (addition of two sources): The value specified by **index3** is used in the same manner as in Type 1 (q.v.)

Type = 9 (subtraction of two sources): The value specified by **index3** is used in the same manner as in Type 1 (q.v.)

Type = 10 (triggered time base): The value specified by **index3** is not used for this type.

Type = 11 (floating-point read): The value specified by **index3** is used in the same manner as in Type 1 (q.v.)

Type = 12 (single-word read with error check): The value specified by **index3** is used in the same manner as in Type 1 (q.v.)

EncTable[*n*].index4

Description: Encoder table entry fourth conversion factor

Range: (Type specific, in 0 .. 15 range)

Units: (Type specific)

Default: Auto-configured based on hardware

EncTable[*n*].index4 specifies the fourth conversion factor used (if any) to process the raw data for this entry. Its function varies by conversion method, which is specified by **EncTable[*n*].type**.

Type = 0 (end of table): The value specified by **index4** is not used for this type.

Type = 1 (single-word read): If the value of **index2** for the entry is less than 32, then the value of **index4** specifies the number of integrations performed on the source data. The valid values are 0, 1, and 2. The default value of 0 (no integrations) is appropriate for the most common case of a position sensor used in a position loop.

A value of 1 causes a single integration of the source data. This is appropriate for the use of a velocity sensor in a position loop (e.g. a tachometer read through an A/D converter). It is also appropriate for simulating a simple "Type 1" servo loop, with just one integration in the simulated plant (e.g. from velocity to position). The output of the loop is written either to a register in user shared memory or to an ASIC's output register (DAC, PWM, or PFM). It is then read with this entry and integrated for the simulated position feedback.

A value of 2 causes a pure double integration of the source data. This is appropriate for the use of an acceleration sensor in a position loop (e.g. an accelerometer through an A/D converter). It is also appropriate for simulating a more complex "Type 2" servo loop, with two integrations in the simulated plant (e.g. from torque/acceleration to position). The output of the loop is written either to a register in user shared memory or to an ASIC's output register (DAC, PWM, or PFM). It is then read with this entry and double-integrated for the simulated position feedback.

A value of 3 to 15 causes a double integration of the source data with damping. (With a value of 2, there is no damping.) In this case, the value at the first integration stage is input to an exponential low-pass filter with a weighting of $1/(2^{16-\text{index4}})$. Higher values of **index4** provide heavier (quicker) damping. This filtering is primarily used for simulated servo loops with double integration – in open-loop mode without this damping, it can be very difficult to stop the output from changing. Values of 3 to 15 for this element are new in V2.0 firmware, released 1st quarter 2015.

If the source data is integrated, either once or twice, the entry element **MaxDelta** acts as a velocity limit (i.e. a limit on the first derivative of the integrated data), regardless of the setting of **index3**. This limit is enforced indefinitely, again regardless of the setting of **index3**.

However, if the value of **index2** for the entry is 32 or greater, then the value of **index4** specifies an "exponent" used in conjunction with **index1** to set the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained above, under **index2**. The tracking filter is seldom used for this conversion method, unless the source register is from an A/D converter.

Type = 2 (double-word read): The value specified by **index4** is used in the same manner as in Type 1 (q.v.). The tracking filter is seldom used for this conversion method.

Type = 3 (software 1/T encoder extension): If the value of **index2** for the entry is less than 32, then the value specified by **index4** is not used for this type. However, if the value of **index2** for the entry is 32 or greater, then the value of **index4** specifies an "exponent" used in conjunction with **index1** to set the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained above, under **index2**. The tracking filter is seldom used for this conversion method.

Type = 4 (sinusoidal encoder arctangent extension): If the value of **index2** for the entry is less than 32, then the value specified by **index4** is not used for this type. However, if the value of **index2** for the entry is 32 or greater, then the value of **index4** specifies an "exponent" used in conjunction with **index1** to set the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained above, under **index2**.

Type = 5 (four-byte read): The value specified by **index4** is used in the same manner as in Type 1 (q.v.). The tracking filter is seldom used for this conversion method.

Type = 6 (resolver direct conversion): If the value of **index2** for the entry is less than 32, then the value specified by **index4** is not used for this type. However, if the value of **index2** for the entry is 32 or greater, then the value of **index4** specifies an "exponent" used in conjunction with **index1** to set the integral gain term *Ki* in a "tracking filter". The operation of the tracking filter is explained above, under **index2**.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **index4** is used in the same manner as in Type 1 (q.v.).

Type = 8 (addition of two sources): The value specified by **index4** is used in the same manner as in Type 1 (q.v.).

Type = 9 (subtraction of two sources): The value specified by **index4** is used in the same manner as in Type 1 (q.v.).

Type = 10 (triggered time base): The value specified by **index4** is not used for this type.

Type = 11 (floating-point read): The value specified by **index4** is used in the same manner as in Type 1 (q.v.)

Type = 12 (single-word read with error check): The value specified by **index4** is used in the same manner as in Type 1 (q.v.)

EncTable[*n*].index5

Description: Encoder table entry fifth conversion factor

Range: 0 .. 255

Units: (Type specific)

Default: 0

EncTable[*i*].index5 specifies the fifth conversion factor used (if any) to process the raw data for this entry. Its function varies by conversion method, which is specified by **EncTable[*n*].type**. At the default value of 0, the action of all conversion methods is the same as in older firmware versions.

Type = 0 (end of table): The value specified by **index5** is not used for this type.

Type = 1 (single-word read): The value specified by **index5** is not used for this type.

Type = 2 (double-word read): The value specified by **index5** is not used for this type.

Type = 3 (software 1/T encoder extension): The value specified by **index5** is not used for this type.

Type = 4 (sinusoidal encoder arctangent extension): The value of **index5** specifies which type of Servo IC format is used for the source data. The default value of 0 specifies a PMAC2-style IC, as used in the ACC-51E interpolator. In this case, **pEnc** and **pEnc1** for the entry should specify the addresses of **Gate1[*i*]** elements. A value of 1 specifies a PMAC3-style IC, as used in the ACC-24E3 interpolator. In this case, **pEnc** and **pEnc1** for the entry should specify the addresses of **Gate3[*i*]** elements.

Type = 5 (four-byte read): The value specified by **index5** is not used for this type.

Type = 6 (resolver direct conversion): The value specified by **index5** is not used for this type.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **index5** is not used for this type.

Type = 8 (addition of two sources): The value specified by **index5** is not used for this type.

Type = 9 (subtraction of two sources): The value specified by **index5** is not used for this type.

Type = 10 (triggered time base): The value specified by **index5** is not used for this type.

Type = 11 (floating-point read): The value of **index5** specifies the pre-scaling factor for the floating-point value in the source register. This value is multiplied by (**index5** + 1) before it is converted to a 32-bit integer value for intermediate processing. This provides more flexibility in the usable ranged of the input value. For use in direct microstepping, values of **index5** of around 63 (for a multiplication factor of 64) are commonly used.

Type = 12 (single-word read with error check): The value specified by **index5** is not used for this type.

EncTable[*n*].index6

Description: Encoder table entry sixth conversion factor

Range: 0 .. 2^{32}-1

Units: (Type specific)

Default: 0

EncTable[*i*].index6 specifies the sixth conversion factor used (if any) to process the raw data for this entry. Its function varies by conversion method, which is specified by **EncTable[*n*].type**. **EncTable[*i*].index6** uses the same registers as **EncTable[*n*].CoverSerror**, which is used in types 4 and 6 conversions.

Type = 0 (end of table): The value specified by **index6** is not used for this type.

Type = 1 (single-word read): The value specified by **index6** is not used for this type.

Type = 2 (double-word read): The value specified by **index6** is not used for this type.

Type = 3 (software 1/T encoder extension): The value specified by **index6** is not used for this type.

Type = 4 (sinusoidal encoder arctangent extension): The value of **index6** will change as **EncType[*n*].CoverSerror** is changed in this method, as the two elements share the same register.

Type = 5 (four-byte read): The value specified by **index6** is not used for this type.

Type = 6 (resolver direct conversion): The value specified by **index6** is not used for this type.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **index6** is not used for this type.

Type = 8 (addition of two sources): The value specified by **index6** is not used for this type.

Type = 9 (subtraction of two sources): The value specified by **index6** is not used for this type.

Type = 10 (triggered time base): The value specified by **index6** is not used for this type.

Type = 11 (floating-point read): The value of **index6** specifies whether the source register is to be treated as a single-precision (32-bit) floating point value or a double-precision (64-bit) floating-point value. If set to the default value of 0, it will treat the source value as single-precision. This is compatible with older firmware versions, which could only use single-precision values. If **index6** is set to 1, it will treat the source value as double-precision.

Type = 12 (single-word read with error check): The value specified by **index6** acts as a 32-bit mask word that is logically combined through a bit-by-bit logical AND operation with the value read from the register specified by **pEnc1** for the entry. If the resulting 32-bit value is non-zero, the position data for this servo cycle is considered invalid, and a calculated replacement value is used instead, extrapolating from last valid readings.

If bit *n* (*n* = 0 to 31) of **index6** is set to 1, then bit *n* of the register specified by **pEnc1** is considered an error bit. This register is usually the status register for a serial encoder interface.

EncTable[*n*].MaxDelta

Description: Encoder table entry maximum legal output change

Range: 0 .. 32,767

Units: LSBs per servo cycle, or LSBs per servo cycle per servo cycle

Default: 0

Type = 0 (end of table): The value specified by **MaxDelta** is not used for this type.

Type = 1 (single-word read): The value of **MaxDelta** specifies whether any derivative limiting is performed on the source data, and if so, what the derivative limit is. If **MaxDelta** is equal to 0, no derivative limiting is enabled.

If **MaxDelta** is greater than 0, it enables derivative limiting and specifies the magnitude of the limit of the derivative to be limited. If the **index3** element for the entry is 0, **MaxDelta** specifies the magnitude of the first-derivative limit (velocity). If the **index3** element for the entry is greater than 0, **MaxDelta** specifies the magnitude of the second-derivative limit (acceleration).

In both cases, the starting (pre-derivative) units for the **MaxDelta** limit are those of the source data after the shift-right specified by **index2**. Since the intent of this shift is to put the LSB of actual data in bit 0 of the word, the starting units for the limit will typically be in LSBs of the actual source data. If **MaxDelta** specifies a velocity limit, its units are LSBs per servo cycle. If **MaxDelta** specifies an acceleration limit, its units are LSBs per servo cycle per servo cycle.

If **MaxDelta** acts as a velocity limit (**index3** = 0), if the limit is exceeded in one servo cycle when it was not exceeded in the previous servo cycle, the position result is computed using the previous cycle's velocity (found in the entry's **PrevDelta** status element). If the limit is again exceeded in immediately following servo cycles, the position result is computed using **MaxDelta** as the velocity. This permits quick slewing to a new position value when the position source changes.

If **MaxDelta** acts as an acceleration limit (**index3** > 0), if the limit is exceeded in up to **index3** consecutive servo cycles, the position result is computed using the previous cycle's acceleration (found in the entry's **PrevDelta** status element). If the limit is again exceeded in an immediately following servo cycle, the position result jumps to the source value, permitting a new position value when the position source changes.

If **index4** is set to a value greater than 0 to specify numerical integration of the source data, **MaxDelta** acts as a velocity limit (i.e. a limit on the first derivative of the integrated data)

expressed in LSBs per servo cycle, regardless of the setting of **index3**. This limit is enforced indefinitely, again regardless of the setting of **index3**.

Type = 2 (double-word read): The value specified by **MaxDelta** is used in the same manner as in Type 1 (q.v.).

Type = 3 (software 1/T encoder extension): The value specified by **MaxDelta** is not used for this type.

Type = 4 (sinusoidal encoder arctangent extension): The word for the **MaxDelta** element is not used for a limiting function in this type. Instead, the word is used to hold both the **SinBias** and **CosBias** elements (q.v.).

Type = 5 (four-byte read): The value specified by **MaxDelta** is used in the same manner as in Type 1 (q.v.).

Type = 6 (resolver direct conversion): The word for the **MaxDelta** element is not used for a limiting function in this type. Instead, the word is used to hold both the **SinBias** and **CosBias** elements (q.v.).

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **MaxDelta** is used in the same manner as in Type 1 (q.v.).

Type = 8 (addition of two sources): The value specified by **MaxDelta** is used in the same manner as in Type 1 (q.v.).

Type = 9 (subtraction of two sources): The value specified by **MaxDelta** is used in the same manner as in Type 1 (q.v.).

Type = 10 (triggered time base): The value specified by **MaxDelta** is not used for this (temporary) type. However, it will be used for those entries that revert to Type = 1 when the trigger occurs.

Type = 11 (floating-point read): The value specified by **MaxDelta** is used in the same manner as in Type 1 (q.v.), after the floating-point source has been prescaled with **index5** and converted to an intermediate fixed-point value.

Type = 12 (single-word read with error check): The value specified by **MaxDelta** is used in the same manner as in Type 1 (q.v.).

EncTable[*n*].pEnc

Description: Encoder table entry primary source address

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

EncTable[*n*].pEnc specifies which register the entry uses for its primary source of raw data. It contains the address of this register. Note that despite the element name, it is not required that encoders be used for feedback.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

The typical addresses used depend on the conversion type:

Type = 0 (end of table): The address specified by **pEnc** is not used for this type.

Type = 1 (single-word read): Many sensor devices provide data that can be read from a single register.

For a raw encoder counter without any type of extension to sub-count data, the PMAC2-style ASIC channel's "phase capture" register should be read. This is true both for real encoder feedback and (especially) when the counter is used for simulated feedback from the channel's pulse output. For example:

EncTable[1].pEnc=Gate1[4].Chan[0].PhaseCapt.a

EncTable[2].pEnc=Gate2[0].Chan[1].PhaseCapt.a

For a PMAC3-style ASIC channel's encoder counter value with hardware interpolation from either timer data (1/T) or arctanengent data (sine/cosine), the channel's "servo capture" register should be read. For example:

EncTable[3].pEnc=Gate3[2].Chan[2].ServoCapt.a

For a serial encoder read through a PMAC3-style ASIC or an ACC-84 FPGA, the first data register for the channel should be read. For example:

EncTable[3].pEnc=Gate3[4].Chan[3].SerialEncDataA.a

EncTable[4].pEnc=Acc84E[1].Chan[0].SerialEncDataA.a

For a magnetostrictive linear displacement transducer (MLDT) interrogated directly by a PMAC ASIC pulse ("RS-422 mode"), the position is measured by the time elapsed until the echo pulse is received, which can be read in the "TimeBetweenCts" register:

EncTable[4].pEnc=Gate1[4].Chan[3].TimeBetweenCts.a

For an analog-to-digital converter (ADC) register, as from a potentiometer or linear variable displacement transducer (LVDT), the ADC register (or the de-multiplexed result) can be read:

EncTable[5].pEnc=Gate1[6].Chan[0].Adc[0].a

EncTable[6].pEnc=AdcDemux.ResultLow[3].a

EncTable[7].pEnc=AdcDemux.ResultHigh[1].a

For a parallel data word from I/O in a single register, this simply specifies the address of the register:

EncTable[8].pEnc=Gate2[0].LowIoData.a

For further processing of the result of a previous entry, this specifies the stored position result of that entry:

EncTable[9].pEnc=EncTable[8].PrevEnc.a

If the register you wish to read does not have a pre-defined data structure element assigned to it, you can specify it by address offset:

EncTable[10].pEnc=Sys.piom+$A00004

If the hardware feedback register has also been read for phase commutation position feedback and the servo feedback can use the same position value (e.g. without further extension), the entry can read the stored value from the motor's commutation algorithm in memory, which is substantially quicker than reading a hardware register. This is usually only important if the computational requirements of the Power PMAC are being pushed to the limit. This setting looks like:

EncTable[11].pEnc=Motor[11].PrevPhaseEnc.a

For data received over the MACRO ring, this specifies the register of the MACRO IC containing the position:

EncTable[12].pEnc=Gate2[0].Macro[4][0].a

EncTable[13].pEnc=Gate3[0].MacroInA[8][0].a

For data received of the EtherCAT network, this specifies the register of the EtherCAT holding register mapped to the EtherCAT drive's actual position register:

EncTable[14].pEnc=ECAT[0].IO[92].Data.a

Type = 2 (two-word parallel read): This format is typically used when 32 bits of feedback are read from an interface with a 24-bit bus. The low 24 bits are read from the register specified by **pEnc**. (The high 8 bits are read from the register specified by **pEnc1**.) For example:

EncTable[11].pEnc=Gate2[0].LowIoData

Type = 3 (software 1/T encoder extension): In this conversion type, the table reads the encoder count value and the timer values separately from a PMAC2-style ASIC, and then mathematically combines them. Here the **pEnc** element should be set to the address of the ASIC channel's "servo capture" register. (The **pEnc1** element specifies the timer registers.) For example:

EncTable[12].pEnc=Gate1[6].Chan[3].ServoCapt.a

EncTable[13].pEnc=Gate2[0].Chan[1].ServoCapt.a

Type = 4 (sinusoidal encoder arctangent extension): In this conversion type, the table reads the encoder count value and the sine/cosine ADC values separately, and then mathematically

combines them. Here the **pEnc** element should be set to the address of the ASIC channel's "status" register. For example:

EncTable[14].pEnc=Gate1[8].Chan[0].Status.a

Type = 5 (4-byte read): In this conversion type, the table reads 8 bits in each of 4 registers (if not all 4 registers are used, some can be masked out). Here the **pEnc** element should be set to the address of the least significant byte. The most common source of this data format is the "IOGATE" ASIC on ACC-14E or equivalent cards. For example:

EncTable[15].pEnc=GateIo[0].DataReg[0].a // Port A LSByte

EncTable[16].pEnc=GateIo[0].DataReg[3].a // Port B LSByte

Type = 6 (resolver direct conversion): In this conversion type, the table reads the register containing the present latched value of the excitation word. Refer to the resolver accessory manual for specific settings.

Type = 7 (extended hardware sinusoidal interpolation): In this conversion type, the table reads the extended encoder count value in the DSPGATE3 IC that is latched on the servo interrupt and combines it with the higher-resolution arctangent element. Here the **pEnc** element should be set to the address of the ASIC channel's "servo capture" register. For example:

EncTable[17].pEnc=Gate3[1].Chan[2].ServoCapt.a

Type = 8 (addition of two sources): In this conversion type, the table reads the register containing the first of two 32-bit values to be added together. Almost always, this register is an intermediate result register (before final output scaling) of a previous (lower-numbered) entry in the table. For example:

EncTable[18].pEnc=EncTable[9].PrevEnc.a

Type = 9 (subtraction of two sources): In this conversion type, the table reads the register containing the 32-bit value from which a second 32-bit value will be subtracted. Almost always, this register is an intermediate result register (before final output scaling) of a previous (lower-numbered) entry in the table. For example:

EncTable[19].pEnc=EncTable[11].PrevEnc.a

Type = 10 (triggered time base): In this conversion type, which is just a temporary variant ("frozen" state) of a Type = 1 or Type = 3 entry, the value of **EncTable[*n*].pEnc** should be left at the same value as in the "running" state. For a Type = 1 entry, this should be something like:

EncTable[20].pEnc=Gate3[2].Chan[3].ServoCapt.a

For a Type = 3 entry, this should be something like:

EncTable[21].pEnc=Gate1[8].Chan[2].ServoCapt.a

Type = 11 (floating-point read): In this conversion type, the table reads a floating-point value from the register. If **index6** is set to the default value of 0, it is a single-precision (32-bit) floating-point value; if **index6** is set to 1, it is a double-precision (64-bit) floating-point value.

For single-precision values, **pEnc** can be set to the address of the motor's servo command output **IqCmd** register, which is useful for simulated servo loops, or of an **Fdata[*i*]** register in the user shared memory buffer. For example:

EncTable[22].pEnc=Motor[6].IqCmd.a

EncTable[23].pEnc=Sys.Fdata[495].a

For double-precision values, **pEnc** can be set to the address of the motor's commutation angle **PhasePos** register, which is used for direct microstepping control, of the motor's net desired position **DesPos** register, which is useful for tracking a commanded trajectory, or of a **Ddata[*i*]** register in the user shared memory buffer. For example:

EncTable[24].pEnc=Motor[2].PhasePos.a

EncTable[25].pEnc=Motor[19].DesPos.a

EncTable[26].pEnc=Sys.Ddata[3182].a

Type = 12 (single-word read with error check): While many sensor devices provide data that can be read from a single register, this method is primarily intended to work with data from a serial encoder. In this case, the first data register for the serial encoder should be read. For example:

EncTable[27].pEnc=Gate3[0].Chan[3].SerialEncDataA.a

EncTable[28].pEnc=Acc84E[2].Chan[0].SerialEncDataA.a

EncTable[*n*].pEnc1

Description: Encoder table entry secondary source address

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

EncTable[*n*].pEnc1 specifies which register the entry uses for its secondary source of raw data, if any. It contains the address of this register. Note that despite the element name, it is not required that encoders be used for feedback.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

The typical addresses used depend on the conversion type:

Type = 0 (end of table): The address specified by **pEnc1** is not used for this type.

Type = 1 (single-word parallel read): The address specified by **pEnc1** is not used for this type.

Type = 2 (two-word parallel read): This format is typically used when 32 bits of feedback are read from an interface with a 24-bit bus. The high 8 bits are read from the register specified by **pEnc1**. For example:

EncTable[11].pEnc1=Gate2[0].HighIoData

Type = 3 (software 1/T encoder extension): In this conversion type, the table reads the encoder count value and the timer values separately from a PMAC2-style ASIC, and then mathematically combines them. Here the **pEnc1** element should be set to the address of the ASIC channel's "time between counts" register. For example:

EncTable[12].pEnc1=Gate1[6].Chan[3].TimeBetweenCts.a

EncTable[13].pEnc1=Gate2[0].Chan[1]. TimeBetweenCts.a

Type = 4 (sinusoidal encoder arctangent extension): In this conversion type, the table reads the encoder count value and the sine/cosine ADC values separately, and then mathematically combines them. Here the **pEnc1** element should be set to the address of the ASIC channel's "ADC Phase A" register. For example:

EncTable[14].pEnc1=Gate1[8].Chan[0].Adc[0].a

Type = 5 (four-byte read): In this conversion type, the table reads 8 bits in each of 4 registers (if not all 4 registers are used, some can be masked out). Here the **pEnc1** element should be set to the address of the second least significant byte. The most common source of this data format is the "IOGATE" ASIC on ACC-14E or equivalent cards. For example:

EncTable[15].pEnc1=GateIo[0].DataReg[1].a // Port A 2nd LSByte

EncTable[16].pEnc1=GateIo[0].DataReg[4].a // Port B 2nd LSByte

Type = 6 (resolver direct conversion): In this conversion type, the table reads the excitation value and the sine/cosine ADC values separately, and then mathematically combines them. Here the **pEnc1** element should be set to the address of the ASIC channel's "ADC Phase A" register. For example:

EncTable[17].pEnc1=Gate1[10].Chan[2].Adc[0].a

Type = 7 (extended hardware sinusoidal interpolation): In this conversion type, the table reads the extended encoder count value in the DSPGATE3 IC that is latched on the servo interrupt and combines it with the higher-resolution arctangent element. Here the **pEnc1** element should be set to the address of the ASIC channel's arctangent element. For example:

EncTable[17].pEnc1=Gate3[1].Chan[2].AtanSumOfSqr.a

Type = 8 (addition of two sources): In this conversion type, the table reads the register containing the second of two 32-bit values to be added together. Almost always, this register is an intermediate result register (before final output scaling) of a previous (lower-numbered) entry in the table. For example:

EncTable[18].pEnc1=EncTable[10].PrevEnc.a

Type = 9 (subtraction of two sources): In this conversion type, the table reads the register containing the 32-bit value that is subtracted from the first 32-bit value. Almost always, this register is an intermediate result register (before final output scaling) of a previous (lower-numbered) entry in the table. For example:

EncTable[19].pEnc1=EncTable[12].PrevEnc.a

Type = 10 (triggered time base): In this conversion type, the table reads the encoder channel's status register to check for the triggere condition. Here the **pEnc1** element should be set to the address of the ASIC channel's status register. For example:

EncTable[20].pEnc1=Gate1[12].Chan[3].Status.a

Note that if the "running" state of the entry is the Type = 3 software 1/T encoder extension, this element will automatically revert to a value of **Gate1[*i*].Chan[*j*].TimeBetweenCts.a**.

Type = 11 (floating-point read): The address specified by **pEnc1** is not used for this type.

Type = 12 (single-word read with error check): In this conversion type, **pEnc1** should be set to the address of the register containing one or more error bits for the feedback. This type is intended mainly for serial encoder protocols, so the second or third data register for the channel's serial encoder interface should be read. For example:

EncTable[21].pEnc1=Gate3[6].Chan[0].SerialEncDataB.a

EncTable[22].pEnc1=Acc84E[2].Chan[1].SerialEncDataC.a

The **index6** element for this table entry will specify which bit or bits of the specified register will be treated as an error bit.

EncTable[*n*].PrevDelta

Description: Encoder table entry pre-integration offset/previous cycle change

Range: -2^{31}.. $2^{31}-1$

Units: Units of source 32-bit register

Default: 0

If the **EncTable[*n*]** entry is set up to integrate the source data (**index4** = 1 or 2, and **index2** < 32), **EncTable[*n*].PrevDelta** is a saved setup element for the entry that acts as a pre-integration offset for the source data. Each servo cycle, the value of **PrevDelta** is added to the 32-bit source data before any shifting operations as specified by **index1** and **index2**, and before the numerical integration as specified by **index4**. In this way, **PrevDelta** can act to compensate for biases such as analog offsets in the source data.

If the entry is not set up to integrate the source data, **PrevDelta** is a status element for the entry representing the latest servo cycle's first derivative (if **index3** = 0 and **MaxDelta** > 0 to limit velocity) or second derivative (if **index3** > 0 and **MaxDelta** > 0 to limit acceleration). If it is a

first-derivative value, its units are LSBs of the source data per servo cycle; if it is a second-derivative value, its units are LSBs of the source data per servo cycle per servo cycle. In both cases, it assumes the LSB of true source data is in bit 0 of the intermediate 32-bit register after the right-shift by **index2**.

EncTable[*n*].ScaleFactor

Description: Encoder table entry output scale factor

Range: Floating-point

Units: Output units per LSB of shifted data

Default: Auto-configured based on hardware

In the last operation of a conversion table entry, the previously processed intermediate result is multiplied by **EncTable[*n*].ScaleFactor** to compute the output. This operation also converts the integer format of the intermediate result to a floating-point format for the final output. While there are no set rules as to what **ScaleFactor** must be, the value of **ScaleFactor** effectively defines the units of the entry's output value, and so to have these values make sense, there are guidelines as to how to set **ScaleFactor** for each type of conversion so the output units are sensible.

Remember that as a floating-point value, the final result can have fractional resolution that has a real effect on the operation of the servo loop. The fractional values are not "lost". Note that changing the **ScaleFactor** of a value used in a feedback loop changes the overall gain of the loop, so if **ScaleFactor** is changed, some of the loop gains may have to be changed to compensate.

Type = 0 (end of table): The value specified by **ScaleFactor** is not used for this type.

Type = 1 (single-word read): After the two shifting operations specified by **index2** and **index1**, the LSB of the actual data is typically in Bit (32 minus number of bits) of the 32-bit word. Since most users want the output to be in units of LSBs of actual data, this intermediate result must be multiplied by $2^{-(32-[\#\ of\ bits])}$. For example, if there were 20 bits of real data, **ScaleFactor** should be set to $2^{-(32-20)}$, or 2^{-12}, which is 0.00244140625.

If you are setting this directly into the element, you may find it easier to use a mathematical expression, such as **EncTable[1].ScaleFactor=1/4096**, or **EncTable[1].ScaleFactor=1/exp2(12)**.

Note that the table setup menu in the IDE asks you what the "Output units per LSB" is, suggesting 1.0 as a default and sets **ScaleFactor** to the value of your response multiplied by $2^{-(32-[\#\ of\ bits])}$, where "# of bits" is another menu value.

Type = 2 (double-word read): The value specified by **ScaleFactor** is used in the same manner as in Type 1 (q.v.).

Type = 3 (software 1/T encoder extension): After the timer-based fractional count is combined with the whole-count value, the intermediate result has 9 bits of fractional count, so it is in units

of 1/512 of a quadrature count. Since most users want the output to be in units of quadrature counts, this intermediate result must be multiplied by 1/512, which is equal to 0.001953125.

If you are setting this directly into the element, you may find it easier to use a mathematical expression, such as **EncTable[1].ScaleFactor=1/512**, or **EncTable[1].ScaleFactor=1/exp2(9)**.

Note that the table setup menu in the IDE asks you what the "Output units per quadrature count" is, suggesting 1.0 as a default and sets **ScaleFactor** to the value of your response multiplied by 1/512.

Type = 4 (sinusoidal encoder arctangent extension): When used with a PMAC2-style IC (**index5** = 0), after the arctangent-based fractional count is combined with the whole-count value, the intermediate result has 10 bits of fractional count, so it is in units of 1/1024 of a quadrature count (or 1/4096 of an encoder line). Different users may want different output units for the entry.

If you want the units to be LSBs of interpolated count, **ScaleFactor** should be set to 1.0. If you want the units to be quadrature counts (1/4 encoder line), **ScaleFactor** should be set to 1/1024, or 0.0009765625. Remember that many servo loop gain terms will vary in inverse proportion to the scale factor selected.

When used with a PMAC3-style IC (**index5** = 1), after the arctangent-based fractional count is combined with the whole-count value, the intermediate result has 14 bits of fractional count, so it is in units of 1/16,384 of a quadrature count (or 1/65,536 of an encoder line). Different users may want different output units for the entry.

If you want the units to be LSBs of interpolated count, **ScaleFactor** should be set to 1.0. If you want the units to be quadrature counts (1/4 encoder line), **ScaleFactor** should be set to 1/16,384. Remember that many servo loop gain terms will vary in inverse proportion to the scale factor selected.

Type = 5 (four-byte read): The value specified by **ScaleFactor** is used in the same manner as in Type 1 (q.v.).

Type = 6 (resolver direct conversion): The intermediate result provides the arctangent calculation in the high 16 bits of the 32-bit register. The value of **ScaleFactor** used is dependent on what you want to consider an increment of measurement. In general, you want to set **ScaleFactor** to $2^{-(32-[\#\ of\ bits])}$, where "# of bits" is the precision you have.

If you want to treat the conversion as a 16-bit conversion (16-bit ADCs required) you would set **ScaleFactor** to 2^{-16}, which is 1/65,536, or 0.0000152587890625.

If you want to treat the conversion as a 14-bit conversion you would set **ScaleFactor** to 2^{-18}, which is 1/262,144, or 0.000003814697265625.

Type = 7 (extended hardware sinusoidal interpolation): After the extended arctangent fractional-count value is combined with the whole-count value, the intermediate result has 14 bits of fractional count, so it is in units of 1/16,384 of a quadrature count (or 1/65,536 of an encoder line). Different users may want different output units for the entry.

If you want the units to be LSBs of the interpolated count, **ScaleFactor** should be set to 1.0. If you want the units to be quadrature counts (1/4 encoder line), **ScaleFactor** should be set to

1/16384. If you want the units to be encoder lines, **ScaleFactor** should be set to 1/65536. Remember that many servo loop gain terms will vary in inverse proportion to the scale factor selected.

Type = 8 (addition of two sources): The value specified by **ScaleFactor** is used in the same manner as in Type 1 (q.v.).

Type = 9 (subtraction of two sources): The value specified by **ScaleFactor** is used in the same manner as in Type 1 (q.v.).

Type = 10 (triggered time base): The value specified by by **ScaleFactor** is not used while in this temporary "frozen" stage, but will be used in the Type 1 or Type 3 entry this reverts to as soon as the trigger occurs.

Type = 11 (floating-point read): The value specified by **ScaleFactor** multiplies the intermediate fixed-point value, which is equivalent to the value of the source floating-point register multiplied by 256 * (**index5** + 1). If the pre-scaling term **index5** is set to its default value of 0, a value for **ScaleFactor** of 1/256 provides an entry output value that is equal to the input value.

Type = 12 (single-word read with error check): The value specified by **ScaleFactor** is used in the same manner as in Type 1 (q.v.).

EncTable[*n*].SinBias

Description: Encoder table entry sine-term offset

Range: -32,768 .. 32,767

Units: Units of 16-bit ADC

Default: 0

Type = 0 (end of table): The value specified by **SinBias** is not used for this type.

Type = 1 (single-word read): The word for the **SinBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 2 (double-word read): The word for the **SinBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 3 (software 1/T encoder extension): The value specified by **SinBias** is not used for this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 4 (sinusoidal encoder arctangent extension): The value of **SinBias** specifies the offset that is added to the measured "sine" term of the sinusoidal encoder. It should contain the value opposite of that which the matching A/D converter reports in **EncTable[*i*].Sin** when it should ideally be reporting a value of zero. Both the measured term and the bias term are treated as 16-bit values, even if the actual A/D converter has a different resolution. The measured data is left-justified, so if a 14-bit ADC is used, the data is found in the high 14 bits of the 16-bit word.

Type = 5 (four-byte read): The word for the **SinBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 6 (resolver direct conversion): The value of **SinBias** specifies the offset that is added to the measured "sine" term of the resolver. It should contain the value opposite of that which the matching A/D converter reports in **EncTable[*n*].Sin** when it should ideally be reporting a value of zero. Both the measured term and the bias term are treated as 16-bit values, even if the actual A/D converter has a different resolution. The measured data is left-justified, so if a 14-bit ADC is used, the data is found in the high 14 bits of the 16-bit word.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **SinBias** is not used for this type.

Type = 8 (addition of two sources): The word for the **SinBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 9 (subtraction of two sources): The word for the **SinBias** element is not used for an offset function in this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 10 (triggered time base): The value specified by **SinBias** is not used for this type.

Type = 11 (floating-point read): The value specified by **SinBias** is not used for this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

Type = 12 (single-word read with error check): The value specified by **SinBias** is not used for this type. Instead, the word is used to hold the **MaxDelta** element (q.v.).

EncTable[*n*].TanHalfPhi

Description: Encoder table entry sine/cosine phase error correction factor

Range: -32,768 .. 32,767

Units: Angle tangent * 65,536

Default: 0

EncTable[*i*].TanHalfPhi specifies a correction factor used to compensate for cosine versus sine phase errors in certain types of feedback. At the default value of 0, the action of all conversion methods is the same as in older firmware versions.

Type = 0 (end of table): The value specified by **TanHalfPhi** is not used for this type.

Type = 1 (single-word read): The value specified by **TanHalfPhi** is not used for this type.

Type = 2 (double-word read): The value specified by **TanHalfPhi** is not used for this type.

Type = 3 (software 1/T encoder extension): The value specified by **TanHalfPhi** is not used for this type.

Type = 4 (sinusoidal encoder arctangent extension): The value of **TanHalfPhi** specifies the tangent of half of the phase angle error between the sine and cosine signals of the sinusoidal encoder. The phase angle error ("phi") is the difference from the ideal 90° separation of the two signals. It is positive if the difference is greater than 90°; it is negative if the difference is less than 90°. This factor is used to adjust the signal values so the corrected signals are separated by 90°.

In operation, the value of **TanHalfPhi** is divided by 65,536 to put it in the range of -0.5 to +0.49999, and this value is used to adjust the measured signal values to produce corrected values representing signals that are ideally 90° apart before the arctangent calculation is performed. The range of +/-0.5 for **TanHalfPhi** covers half angles of +/-26.5°, and so phase angle errors of +/-53° of the signal cycle. Note that at the default value for **TanHalfPhi** of 0, the tangent is 0.0, and no phase correction is performed.

This phase correction is performed after the offset corrections using **SinBias** and **CosBias**, and after the magnitude correction using **CoverSerror**.

Example: If the two signals were found to be separated by 105° instead of the ideal 90°, the phase error *Phi* would be +15°, so the half angle would be +7.5°, and the tangent of the half angle would be +0.13165, and **TanHalfPhi** should be set to 8628 (= 0.13165 * 65,536).

Type = 5 (four-byte read): The value specified by **TanHalfPhi** is not used for this type.

Type = 6 (resolver direct conversion): The value specified by **TanHalfPhi** is not used for this type.

Type = 7 (extended hardware sinusoidal interpolation): The value specified by **TanHalfPhi** is not used for this type.

Type = 8 (addition of two sources): The value specified by **TanHalfPhi** is not used for this type.

Type = 9 (subtraction of two sources): The value specified by **TanHalfPhi** is not used for this type.

Type = 10 (triggered time base): The value specified by **TanHalfPhi** is not used for this type.

Type = 11 (floating-point read): The value specified by **TanHalfPhi** is not used for this type.

Type = 12 (single-word read with error check): The value specified by **TanHalfPhi** is not used for this type.

EncTable[*n*].type

Description: Encoder table entry conversion method

Range: 0 .. 15

Units: none

Default: Auto-configured based on hardware

EncTable[*n*].type specifies the conversion method used for this entry of the table. A value of 0 specifies "end of table"; no higher-numbered entry will be executed, even if that entry specifies a real conversion.

The following conversion types (methods) are supported:

0: End of table
1: Single-word (32-bit) read
2: Double-word (24-bit + 8-bit) read
3: Software 1/T encoder extension
4: Sinusoidal encoder arctangent extension
5: Four-byte read
6: Resolver direct conversion
7: Extended hardware sinusoidal interpolation
8: Addition of two sources
9: Subtraction of two sources
10: Triggered time base
11: Floating-point read
12: Single-word read with error check
13+: (*Reserved for future use*)

Depending on the conversion type used for a particular entry, the other setup elements for the entry will have different meanings. These are documented in the description of each element.

Each conversion type is described below.

Type = 0 (end of table): Each servo cycle, Power PMAC starts processing the table with Entry 0, followed by each entry of increasing index. When it encounters an entry with **type** = 0, it stops processing the table. It does not matter what the other setup elements for that entry are, or what the settings for any higher-numbered entries are; no further processing is done.

Type = 1 (single-word read): In this conversion type, Power PMAC reads a single 32-bit register from the specified address. Often, the actual data forms only a part of this register, and subsequent processing in the entry eliminates possibly spurious data from the rest of the word and scales the actual data properly.

This conversion type is useful for many data formats. It can be used for position from a parallel or serial absolute encoder, from an analog-to-digital converter, from an MLDT timer register. With the PMAC3-style DSPGATE3 ASIC, it can be used to get data pre-processed in the hardware of the ASIC, such as 1/T-interpolated quadrature encoder position, arctangent-interpolated sinusoidal encoder position, or arctangent-processed resolver position.

Type = 2 (double-word read): In this conversion type, Power PMAC reads two separate registers to assemble a preliminary 32-bit data value. In the first read, it takes data from the high 24 bits of the 32-bit register, and uses these bits as the low 24 bits of the preliminary value. In the second read, it takes data from the second lowest byte (bits 8 – 15) of the 32-bit register, and uses these bits as the high 8 bits of the preliminary value. If the actual data forms only a part of this preliminary value, subsequent processing in the entry can eliminate possibly spurious data from the rest of the word and scale the actual data properly.

This conversion type is useful to obtain 32 bits of data from PMAC2 ASICs, which have 24-bit registers. Data coming into the ASIC on the I/O00 – I/O31 pins can be processed with this conversion type

Type = 3 (software 1/T encoder extension): In this conversion type, Power PMAC reads the counter and timer data from an encoder channel on a PMAC2-style DSPGATE1 or DSPGATE2 ASIC, and uses the timer data to compute a fractional-count value to extend the whole-count data of the quadrature counter. The resulting data has a resolution of 1/512 of a quadrature count. (Note that the PMAC3-style DSPGATE3 ASIC used on the ACC-24E3 and similar interfaces does this processing itself in hardware, so for this type of feedback, the simple 32-bit single-register conversion type would be used instead.)

Type = 4 (sinusoidal encoder arctangent extension): In this conversion type, Power PMAC reads the quadrature counter and associated analog-to-digital converter registers from a channel on a PMAC2-style or PMAC3-style ASIC, and uses the ADC data from the sine and cosine signals of the encoder through an arctangent calculation to interpolate between the quadrature-count (zero-crossing) data of the counter. With a PMAC2-style ASIC, the resulting data has a resolution of 1/1024 of a quadrature count, which is 1/4096 of an encoder line. With a PMAC3-style ASIC, the resulting data has a resolution of 1/16,384 of a quadrature count, which is 1/65,536 of an encoder line. (Note that the PMAC3-style DSPGATE3 ASIC used on the ACC-24E3 and similar interfaces can do this processing itself in hardware, so for this type of feedback using the DSPGATE3 ASIC, the simple 32-bit single-register conversion type could be used instead. However, this software conversion provides additional correction terms.)

Type = 5 (four-byte read): In this conversion type, Power PMAC reads four separate registers to assemble a preliminary 32-bit data value. In each register read, it takes data from the second lowest byte (bits 8 – 15) of the 32-bit register, and assembles these four bytes into a 32-bit value. Data from the first register forms the lowest byte of this value; data from the last register forms the highest byte. If the actual data forms only a part of this preliminary value, subsequent processing in the entry can eliminate possibly spurious data from the rest of the word and scale the actual data properly.

This conversion type is useful to obtain position data from accessories such as the ACC-14E using the IOGATE ASICs, which have 8-bit registers.

Type = 6 (resolver direct conversion): In this conversion type, Power PMAC reads the analog-to-digital converter registers from a channel on a PMAC2-style DSPGATE1 or DSPGATE2 ASIC and the simultaneous value of the excitation circuit, and uses the ADC data from the sine and cosine signals of the resolver through an arctangent calculation to calculate the position of the resolver. (Note that the PMAC3-style DSPGATE3 ASIC used on the ACC-24E3 and similar interfaces does this processing itself in hardware, so for this type of feedback using the DSPGATE3 ASIC, the simple 32-bit single-register conversion type would be used instead.)

Type = 7 (extended hardware sinusoidal interpolation): In this conversion type, Power PMAC reads the interpolated count value from a channel of a PMAC3-style DSPGATE3 IC and combines it with the extended value from the hardware-calculated arctangent element from the same channel. The resulting data has a resolution of 1/16,384 of a quadrature count, which is 1/65,536 of an encoder line.

Type = 8 (addition of two sources): In this conversion type, Power PMAC reads two 32-bit source registers from the specified registers and adds the values together. This conversion type is particularly useful for averaging two sensors together

Type = 9 (subtraction of two sources): In this conversion type, Power PMAC reads two 32-bit source registers from the specified registers and subtracts the value of the second from that of the first. This conversion type is particularly useful for skew control of parallel motors.

Type = 10 (triggered time base): This conversion type is a temporary variant of the **type** = 3 software 1/T encoder extension (when using a PMAC2-style IC) or of the **type** = 1 single register read (when using an increment encoder through a PMAC3-style IC). While **type** = 10, the output is "frozen" (no change), regardless of the change in the input. Once it is "armed" by setting **index1** to 2 or 3, the entry will automatically revert to **type** = 1 or **type** = 3, respectively, when the specified trigger occurs.

Type = 11 (floating-point read): In this conversion type, Power PMAC reads a single-precision (32-bit) or a double-precision (64-bit) floating-point register (depending on the setting of **index6**), pre-scales it using **index5** as it converts it to an intermediate fixed-point format, then processes it as if it were a fixed-point source as in **type** = 1. Presently, the valid source registers for this method are a motor's **IqCmd** single-precision servo-output register, a motor's double-precision **DesPos** net desired position register and **PhasePos** commutation angle register, and user shared memory buffer floating-point registers. This method permits simulated servo loops for virtual motors and open-loop direct-microstepping operation.

Type = 12 (single-word read with error check): In this conversion type, Power PMAC reads a single 32-bit register from the specified address for position data. It reads a second register for possible error bits, masking the value read from this register with the value of **index6** to treat only the specified bits as errors. If an error condition is found in a given servo cycle, the position value read in that cycle is not used; it is replaced by a value calculated by extrapolating the most recent valid position values.

Gate1[*i*]. (PMAC2-Style Servo IC) Saved Data Structure Elements

This section describes the saved data structure elements for PMAC2-style "DSPGATE1" Servo ICs.

In the Power PMAC script environment, it is also possible to use the name of the accessory on which the IC is present as the structure name instead of "**Gate1[*i*]**. The names of the accessories for which this is supported are:

- **Acc24E2[*i*].** ACC-24E2 UMAC PWM axis-interface board
- **Acc24E2A[*i*].** ACC-24E2A UMAC analog axis-interface board
- **Acc24E2S[*i*].** ACC-24E2S UMAC stepper axis-interface board
- **Acc51E[*i*].** ACC-51E UMAC sine-encoder interpolator board
- **Acc24C2[*i*].** ACC-24C2 Compact UMAC PWM axis-interface board
- **Acc24C2A[*i*].** ACC-24C2A Compact UMAC analog axis-interface board
- **Acc51C[*i*].** ACC-51C Compact UMAC sine-encoder interpolator board

These names are simply "aliases" for the **Gate1[*i*]** name. The index number ***i*** is the same whether the **Gate1[*i*]** name or the alias name is used. Each DSPGATE1 IC used must have a unique index number. C programs must use the **Gate1[*i*]** name; the aliases are not available in the C environment. Index values ***i*** presently range from 4 to 19 (0 to 3 are not used by any existing hardware), and are determined by the hardware address DIP switch setting of the accessory board.

Note: In converting from a Turbo PMAC application, the Gate1 IC numbers for a Power PMAC system are two times the value of the IC number "*m*" with the identical hardware addressing setup in a Turbo PMAC system. For example, "Servo IC 2" in a Turbo PMAC system is **Gate1[4]** in a Power PMAC system; and "Servo IC 6" in a Turbo PMAC system is **Gate1[12]** in a Power PMAC system.

For Turbo PMAC alternate IC numbers "*m**", the equivalent Power PMAC IC number is 2*m** + 1. For example, "Servo IC 2*" in a Turbo PMAC system is **Gate1[5]** in a Power PMAC system; and "Servo IC 6*" in a Turbo PMAC system is **Gate1[13]** in a Power PMAC system.

Gate1[*i*]. Multi-Channel Setup Elements

This section describes the "multi-channel" saved data structure elements for PMAC2-style "DSPGATE1" Servo ICs. These elements affect all 4 channels on the ASIC.

Gate1[*i*].AdcStrobe

Description: Servo IC ADC strobe word

Range: $000000 .. $FFFFFF

Units: Serial data stream (MSB first, starting on rising edge of phase clock)

Default: $3FFFFF

Legacy I-variable alias: I7*m*06

Gate1[*i*].AdcStrobe controls the ADC strobe signal for all machine interface channels on Servo IC m. The 24-bit word set by this variable is shifted out serially on the ADC_STROB lines, MSB first, one bit per ADC_CLK cycle starting on the rising edge of the phase clock.

Bit 0 (the LSB) of **Gate1[*i*].AdcStrobe** is a control bit that determines whether the Servo IC will expect "header" information on the return data streams that precedes the numerical data from the ADCs. If bit 0 is 0, no header information is expected, and the low output from this bit is held until the next rising edge of the phase clock.

If bit 0 of **Gate1[*i*].AdcStrobe** is 1, up to 4 bits of header information can be accepted on the returned serial data streams from the ADCs (as with the ADCs in Delta Tau's Geo power block amplifiers). These bits are "rolled over" and end up in bits 0 – 3 of the ADC register in the Servo IC, and the numerical data ends up with its MSB in bit 23 of the ADC register. If fewer than 4 header bits are expected, the beginning of the strobe word should be delayed by setting the first bit(s) of **Gate1[*i*].AdcStrobe** to 0. Specifically, if (4 – *n*) header bits are expected, the first *n* bits of **Gate1[*i*].AdcStrobe** should be set to 0. In this setting, the ADC_STROB output is taken low and held low after bit 0 is shifted out.

The first bit that is a "1" creates a rising edge on the ADC_STROB output that is typically used as a "start-convert" signal. Some A/D converters just need this rising edge for the conversion; others need the signal to stay high all of the way through the conversion. Intermediate bits of the ADC_STROB output can be used to transmit other information in some applications.

The default **Gate1[*i*].AdcStrobe** value of $3FFFFF is appropriate for most Delta Tau "Geo" power-block amplifiers. A value of $FFFFFE is suitable for use with Delta Tau "Quad Amps", most third-party direct-PWM amplifiers, and with ACC-28B A/D converters. Refer to the specific amplifier or ADC manual for details.

Gate1[*i*].ClockCtrl

Description: Servo IC full-word clock control element

Range: $0 .. $FFFFFF

Units: Bit field

Default: $3008D2

Gate1[*i*].ClockCtrl is the full-word element that comprises the multi-channel setup for generating clock signals for the IC. It is comprised of the following partial-word elements:

Partial-Word Element	Script Bits	C Bits	Functionality
ServoClockDiv	23 – 20	31 – 28	Servo clock frequency divider control
PhaseClockDiv	19 – 16	27 – 24	Phase clock frequency divider control
(*reserved*)	15 – 14	23 – 22	(*Reserved for future use*)
PhaseServoDir	13 – 12	21 – 20	Phase and servo clock direction control
HardwareClockCtrl	11 – 00	19 – 08	Hardware clock frequency divider control
(*null*)	..	07 – 00	(*No hardware present*)

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word, even though there are only 24 bits in the physical register. These 24 bits are found at the high end of the 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate1[*i*].DacStrobe

Description: Servo IC DAC strobe word

Range: $000000 .. $FFFFFF

Units: Serial data stream (MSB first, starting on rising edge of phase clock)

Default: $7FFFC0 (for 18-bit DAC)

Legacy I-variable alias: I7*m*05

Gate1[*i*].DacStrobe controls the DAC strobe signal for machine interface channels on the selected PMAC2-style "DSPGATE1" Servo IC. The 24-bit word set by **Gate1[*i*].DacStrobe** is shifted out serially on the DAC_STROB lines, MSB first, one bit per DAC_CLK cycle starting on the rising edge of the phase clock. The value in the LSB is held until the next phase clock cycle.

For typical *n*-bit DACs, the strobe line is held high for *n*-1 clock cycles. Therefore, the common settings of this variable are:

- 18-bit DACs: $7FFFC0 (high for 17 clock cycles)
- 16-bit DACs: $7FFF00 (high for 15 clock cycles)
- 12-bit DACs: $7FF000 (high for 11 clock cycles)

The default **Gate1[*i*].DacStrobe** value of $7FFFC0 is suitable for the 18-bit DACs on Delta Tau products. **Gate1[*i*].DacStrobe** should not be changed from the default unless different DACs are used.

Note

New revisions of the ACC-24E2A UMAC analog axis interface board and the ACC-24C2A Compact UMAC analog axis interface board with 16-bit DACs can use the value of **DacStrobe** for 18-bit DACs for software compatibility with older 18-bit versions.

Gate1[*i*].HardwareClockCtrl

Description: Servo IC hardware clocks frequency control

Range: 0 .. 4095

Units: Individual clock dividers

Gate1[*i*].HardwareClockCtrl = Encoder SCLK Divider
+ 8 * PFM_CLK Divider
+ 64 * DAC_CLK Divider
+ 512 * ADC_CLK Divider

where:
Encoder SCLK Frequency = 39.3216 MHz / (2 ^ Encoder SCLK Divider)
PFM_CLK Frequency = 39.3216 MHz / (2 ^ PFM_CLK Divider)
DAC_CLK Frequency = 39.3216 MHz / (2 ^ DAC_CLK Divider)
ADC_CLK Frequency = 39.3216 MHz / (2 ^ ADC_CLK Divider)

Default: 2258 = 2 + (8 * 2) + (64 * 3) + (512 * 4)
Encoder SCLK Frequency = 39.3216 MHz / (2 ^ 2) = 9.8304 MHz
PFM_CLK Frequency = 39.3216 MHz / (2 ^ 2) = 9.8304 MHz
DAC_CLK Frequency = 39.3216 MHz / (2 ^ 3) = 4.9152 MHz
ADC_CLK Frequency = 39.3216 MHz / (2 ^ 4) = 2.4576 MHz

Legacy I-variable alias: I7*m*03

Gate1[*i*].HardwareClockCtrl controls the frequency of four hardware clock frequencies – SCLK, PFM_CLK, DAC_CLK, and ADC_CLK – for the four machine interface channels on a PMAC2-style "DSPGATE1" Servo IC. It is a 12-bit variable consisting of four independent 3-bit controls, one for each of the clocks. Each of these clock frequencies can be divided down from a starting 39.3216 MHz frequency by powers of 2, 2^N, from 1 to 128 times (N = 0 to 7). This means that the possible frequency settings for each of these clocks are:

Frequency	Divide by	Divider N in $1/2^N$	Frequency	Divide by	Divider N in $1/2^N$
39.3216 MHz	1	0	2.4576 MHz	16	4
19.6608 MHz	2	1	1.2288 MHz	32	5
9.8304 MHz	4	2	611.44 kHz	64	6
4.9152 MHz	8	3	305.72 kHz	128	7

Very few Power PMAC users will be required to change the setting of **Gate1[*i*].HardwareClockCtrl** from the default value.

SCLK: The encoder sample clock signal SCLK controls how often the Servo IC's digital hardware looks at the encoder and flag inputs. The Servo IC can take at most one count per SCLK cycle, so the SCLK frequency is the absolute maximum encoder count frequency. SCLK also controls the signal propagation through the digital delay filters for the encoders and flags; the lower the SCLK frequency, the greater the noise pulse that can be filtered out. The SCLK frequency should optimally be set to the lowest value that can accept encoder counts at the maximum possible rate.

PFM_CLK: The pulse-frequency-modulation clock PFM_CLK controls the PFM circuitry that is commonly used for stepper drives. The maximum pulse frequency possible is 1/4 of the PFM_CLK frequency. The PFM_CLK frequency should optimally be set to the lowest value that can generate pulses at the maximum frequency required.

DAC_CLK: The DAC_CLK controls the serial data frequency into D/A converters. If these converters are on Delta Tau-provided accessories, the DAC_CLK setting should be left at the default value.

ADC_CLK: The ADC_CLK controls the serial data frequency from A/D converters. If these converters are on Delta Tau-provided accessories, the ADC_CLK setting should be left at the default value.

To determine the clock frequencies set by a given value of **Gate1[*i*].HardwareClockCtrl**, use the following procedure:

1.) Divide **Gate1[*i*].HardwareClockCtrl** by 512 and round down to the nearest integer. This value N1 is the ADC_CLK divider.
2.) Multiply N1 by 512 and subtract the product from **Gate1[*i*].HardwareClockCtrl** to get **Gate1[*i*].HardwareClockCtrl'**. Divide **Gate1[*i*].HardwareClockCtrl'** by 64 and round down to the nearest integer. This value N2 is the DAC_CLK divider.
3.) Multiply N2 by 64 and subtract the product from **Gate1[*i*].HardwareClockCtrl'** to get **Gate1[*i*].HardwareClockCtrl"**. Divide **Gate1[*i*].HardwareClockCtrl"** by 8 and round down to the nearest integer. This value N3 is the PFM_CLK divider.
4.) Multiply N3 by 8 and subtract the product from **Gate1[*i*].HardwareClockCtrl"**. The resulting value N4 is the SCLK divider.

Gate1[*i*].HardwareClockCtrl constitutes bits 0 – 11 of the full-word element **Gate1[*i*].ClockCtrl** (bits 8 – 19 of the 32-bit element in C).

Examples

The maximum encoder count frequency in the application is 800 kHz, so the 1.2288 MHz SCLK frequency is chosen. A pulse train up to 500 kHz needs to be generated, so the 2.4576 MHz PFM_CLK frequency is chosen. The default serial DACs and ADCs provided by Delta Tau are used, so the default DAC_CLK frequency of 4.9152 MHz and the default ADC_CLK frequency of 2.4576 MHz are chosen. From the table:

SCLK Divider N: 5
PFM_CLK Divider N: 4
DAC_CLK Divider N: 3
ADC_CLK Divider N: 4

Gate1[*i*].HardwareClockCtrl = 5 + (8 * 4) + (64 * 3) + (512 * 4) = 5 + 32 + 192 + 2048 = 2277

Gate1[*i*].HardwareClockCtrl has been set to 3429. What clock frequencies does this set?

N1 = INT (3429/512) = 6 ADC_CLK = 611.44 kHz
Gate1[*i*].HardwareClockCtrl' = 3429 - (512*6) = 357
N2 = INT (357/64) = 5 DAC_CLK = 1.2288 MHz
Gate1[*i*].HardwareClockCtrl" = 357 - (64*5) = 37

N3 = INT (37/8) = 4	PFM_CLK = 2.4576 MHz
N4 = 37 - (8*4) = 5	SCLK = 1.2288 MHz

Gate1[*i*].PhaseClockDiv

Description: Servo IC phase-clock frequency divider control

Range: 0 .. 15

Units: Phase Clock Freq. = MaxPhase Freq. / (**Gate1[*i*].PhaseClockDiv**+1)

Default: 0
Phase Clock Freq. = 9.0346 kHz / 1 = 9.0346 kHz
(with default value of **Gate1[*i*].PwmPeriod**)

Legacy I-variable alias: I7*m*01

Gate1[*i*].PhaseClockDiv, in conjunction with **Gate1[*i*].PwmPeriod**, determines the frequency of the Phase clock generated inside each PMAC2-style "DSPGATE1" Servo IC. However, only the Servo or MACRO IC told to use and output its own Phase clock with **Gate*n*[*i*].PhaseServoDir** uses the Phase clock signal it generates. Each cycle of the Phase clock, motor phase commutation and digital current-loop algorithms are performed for specified motors.

Specifically, **Gate1[*i*].PhaseClockDiv** controls how many times the Phase clock frequency is divided down from the "maximum phase" clock, whose frequency is set by **Gate1[*i*].PwmPeriod**. The Phase clock frequency is equal to the "maximum phase" clock frequency divided by (**Gate1[*i*].PhaseClockDiv**+1). **Gate1[*i*].PhaseClockDiv** has a range of 0 to 15, so the frequency division can be by a factor of 1 to 16. The equation for **Gate1[*i*].PhaseClockDiv** is

$$Gate1[i].PhaseClockDiv = \frac{MaxPhaseFreq\,(kHz)}{PhaseFreq\,(kHz)} - 1$$

The ratio of MaxPhase Freq. to Phase Clock Freq. must be an integer.

Note: If the phase clock frequency is set too high, lower priority tasks such as communications can be starved for time. If the background tasks are completely starved, the watchdog timer will trip, shutting down the board. If a normal reset of the board does not re-establish a state where the watchdog timer has not tripped and communications works well, it will be necessary to re-initialize the board by powering up with a USB memory stick installed. This restores default settings, so communication is possible, and **Gate1[*i*].PwmPeriod** and **Gate1[*i*].PhaseClockDiv** can be set to supportable values.

Gate1[*i*].PhaseClockDiv constitutes bits 16 – 19 of the full-word element **Gate1[*i*].ClockCtrl** (bits 24 – 27 of the 32-bit element in C).

Example

With a 20 kHz MaxPhase Clock frequency established by **Gate1[*i*].PwmPeriod**, and a desired 6.67 kHz PHASE clock frequency, the ratio between MaxPhase and Phase is 3:

Gate1[*i*].PhaseClockDiv = (20 / 6.67) - 1 = 3 -1 = 2

Gate1[*i*].PhaseServoDir

Description: Servo IC Phase/Servo Clock Direction

Range: 0 .. 3

Units: Bit field

Default: IC- and system-dependent

Legacy I-variable alias: I7*m*07

Gate1[*i*].PhaseServoDir controls whether Servo IC m uses its own internally generated Phase and Servo clock signals as controlled by **Gate1[*i*].PwmPeriod**, **Gate1[*i*].PhaseClockDiv**, and **Gate1[*i*].ServoClockDiv**, or whether it uses Phase and Servo clock signals from an outside source.

In any Power PMAC system, there must be one and only one source of servo and phase clock signals for the system – either one of the Servo ICs or MACRO ICs, or a source external to the system.

Gate1[*i*].PhaseServoDir is a 2-bit value. Bit 0 is set to 0 for the IC to use its own Phase clock signal and output it; it is set to 1 to use an externally input Phase clock signal. Bit 1 is set to 1 for the IC to use its own Servo clock signal and output it; it is set to 1 to use an externally input Servo clock signal. This yields 4 possible values for **Gate1[*i*].PhaseServoDir**:

- **Gate1[*i*].PhaseServoDir** = 0: Internal Phase clock; internal Servo clock
- **Gate1[*i*].PhaseServoDir** = 1: External Phase clock; internal Servo clock
- **Gate1[*i*].PhaseServoDir** = 2: Internal Phase clock; external Servo clock
- **Gate1[*i*].PhaseServoDir** = 3: External Phase clock; external Servo clock

In all normal use, **Gate1[*i*].PhaseServoDir** is either set to 0 (on at most one IC) or 3 (on all the other ICs). Generally, on re-initialization, the Power PMAC automatically selects which IC it will use as the source of its system Phase and Servo clock signals, setting these variable values. Most users will never change these settings.

Gate1[*i*].PhaseServoDir constitutes bits 12 – 13 of the full-word element **Gate1[*i*].ClockCtrl** (bits 20 – 21 of the 32-bit element in C).

Gate1[*i*].PwmCtrl

Description: Servo IC full-word PWM control element

Range: $0 .. $FFFFFF

Units: Bit field

Default: $197F0F

Gate1[*i*].PwmCtrl is the full-word element that comprises the multi-channel setup for generating PWM signals for the IC. It is comprised of the following partial-word elements:

Partial-Word Element	Script Bits	C Bits	Functionality
PwmPeriod	23 – 08	31 – 16	MaxPhase/PWM period control
PwmDeadTime	07 – 00	15 – 08	PWM deadtime/PFM pulse width control
(null)	..	07 – 00	(no hardware present)

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word, even though there are only 24 bits in the physical register. These 24 bits are found at the high end of the 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate1[*i*].PwmDeadTime

Description: Servo IC PWM deadtime/PFM pulse-width control

Range: 0 .. 255

Units: 16*PWMCLK cycles, PFMCLK cycles
PWM deadtime = [16 / PWMCLK freq. (MHz)] * **Gate1[*i*].PwmDeadTime**
= 0.135 µsec * **Gate1[*i*].PwmDeadTime**
PFM pulse width = [1 / PFMCLK freq. (MHz) * **Gate1[*i*].PwmDeadTime**
= PFMCLK period (µsec) * **Gate1[*i*].PwmDeadTime**

Default: 15
PWM deadtime = 0.135 µsec * 15 = 2.03 µsec
PFM pulse width = [1 / 9.8304 MHz} * 15 = 1.526 µsec
(with default **Gate1[*i*].HardwareClockCtrl**)

Legacy I-variable alias: I7*m*04

Gate1[*i*].PwmDeadTime controls the deadtime period between top and bottom on-times in the automatic PWM generation for machine interface channels on PMAC2-style "DSPGATE1" Servo IC. In conjunction with **Gate1[*i*].HardwareClockCtrl**, it also controls the pulse width for the IC's automatic pulse-frequency modulation (PFM) generation for the machine interface channels on the Servo IC.

The PWM deadtime, which is the delay between the top signal turning off and the bottom signal turning on, and vice versa, is specified in units of 16 PWM_CLK cycles. This means that the deadtime can be specified in increments of 0.135 μsec. The equation for **Gate1[*i*].PwmDeadTime** as a function of PWM deadtime is:

$$Gate1[i].PwmDeadTime = \frac{DeadTime(\mu sec)}{0.135\mu sec}$$

Note

Most direct-PWM drives enforce a minimum deadtime period as a safety feature. However, they do so with a lower-frequency clock that is asynchronous to this IC's PWM_CLK signal. This means that relying on the drive's minimum deadtime setting (which occurs when this element specifies a smaller deadtime) decreases the effective command resolution and even can introduce a beat frequency into the output.

The PFM pulse width is specified in PFM_CLK cycles, as defined by **Gate1[*i*].PwmDeadTime**. The equation for **Gate1[*i*].PwmDeadTime** as a function of PFM pulse width and PFM_CLK frequency is:

$$Gate1[i].PwmDeadTime = PfmClkFreq(MHz) * PfmPulseWidth(\mu sec) - 1$$

In the DSPGATE1's PFM pulse generation, the minimum off time between pulses is equal to the pulse width. This means that the maximum PFM output frequency is

$$PfmMaxFreq(MHz) = \frac{PfmClkFreq(MHz)}{2 * [Gate1[i].PwmDeadTime + 1]}$$

Gate1[*i*].PwmDeadTime constitutes bits 0 – 7 of the full-word element **Gate1[*i*].PwmCtrl** (bits 8 – 15 of the 32-bit element in C).

Examples:

A PWM deadtime of approximately 1 microsecond is desired:

Gate1[*i*].PwmDeadTime ≅ 1 μsec / 0.135 μsec ≅ 7

With a 2.4576 MHz PFM_CLK frequency, a pulse width of 0.4 μsec is desired:

Gate1[*i*].PwmDeadTime ≅ 2.4576 MHz * 0.4 μsec ≅ 1

Gate1[*i*].PwmPeriod

Description: Servo IC MaxPhase/PWM period control

Range: 0 .. 32767

Units: MaxPhase Freq. = 117,964.8 kHz / [2***Gate1[*i*].PwmPeriod**+3]
PWM Freq. = 117,964.8 kHz / [4* **Gate1[*i*].PwmPeriod**+6]

Default: 6527
MaxPhase Freq. = 117,964.8 / 13057 = 9.0346 kHz
PWM Freq. = 117,964.8 / 26114 = 4.5173 kHz

Legacy I-variable alias: I7*m*00

Gate1[*i*].PwmPeriod controls the internal "MaxPhase" clock frequency, and the PWM frequency for the four machine interface channels, on a PMAC2-style "DSPGATE1" Servo IC. The internally generated Phase and Servo clocks on Servo IC m are derived from the MaxPhase clock.

If the Servo IC is used to generate the Phase and Servo clocks for the PMAC system (as set by the **Gate1[*i*].PhaseServoDir** variables), this variable is part of the control for the frequency of these system clocks.

Gate1[*i*].PwmPeriod controls these frequencies by setting the limits of the PWM up-down counter, which increments and decrements at the PWMCLK frequency of 117,964.8 kHz (117.9648 MHz).

The actual Phase clock frequency is divided down from the maximum phase clock according to the setting of **Gate1[*i*].PhaseClockDiv**. On the falling edge of the phase clock, PMAC2 samples any serial analog-to-digital converters connected to its Servo ICs (as for phase current measurement), and interrupts the processor to start any necessary phase commutation and digital current-loop algorithms. Even if phasing and current-loop algorithms are not used, the MaxPhase and Phase Clock frequencies are important because the servo clock is derived from the phase clock.

The PWM frequency determines the actual switching frequency of amplifiers connected to any of four machine interface channels with the direct PWM command. It is only important if the direct PWM command signal format is used.

The maximum value that can be written into the PWM command register without full saturation is **Gate1[*i*].PwmPeriod**+1 on the positive end, and – **Gate1[*i*].PwmPeriod**–2 on the negative end. Generally, the "PWM scale factor" **Motor[*i*].PwmSf** for each motor, which determines the maximum PWM command magnitude, is set to **Gate1[*i*].PwmPeriod** + 10%.

Usually, **Gate1[*i*].PwmPeriod** for Servo ICs that are not controlling the system Phase clock frequency are set to the same value as the one that is. If a different PWM frequency is desired for the PWM outputs on other Servo ICs, **Gate1[*i*].PwmPeriod** should be set so that:

$$\frac{2 * PwmFreq(kHz)}{PhaseFreq} = \{Integer\}$$

This will keep the PWM hardware on these channels in synchronization with the software algorithms driven by the system Phase clock. For example, if the phase frequency is 10 kHz, the PWM frequency for other Servo ICs can be 5, 10, 15, 20, (etc.) kHz.

To set **Gate1[*i*].PwmPeriod** for a desired PWM frequency, the following formula can be used:

$$Gate1[i].PwmPeriod = \frac{117{,}964.8\ (kHz)}{4 * PwmFreq\ (kHz)} - 1 \text{ (rounded down)}$$

To set **Gate1[*i*].PwmPeriod** for a desired "maximum phase" clock frequency, the following formula can be used:

$$Gate1[i].PwmPeriod = \frac{117{,}964.8\ (kHz)}{2 * MaxPhaseFr\ eq\ (kHz)} - 1 \text{ (rounded down)}$$

Gate1[*i*].PwmPeriod constitutes bits 8 – 23 of the full-word element **Gate1[*i*].PwmCtrl** (bits 16 – 31 of the 32-bit element in C).

Examples:

To set a PWM frequency of 10 kHz and therefore a MaxPhase clock frequency of 20 kHz:

Gate1[*i*].PwmPeriod = (117,964.8 kHz / [4*10 kHz]) - 1 = 2948

To set a PWM frequency of 7.5 kHz and therefore a MaxPhase clock frequency of 15 kHz:

Gate1[*i*].PwmPeriod = (117,964.8 kHz / [4*7.5 kHz]) - 1 = 3931

Gate1[*i*].ServoClockDiv

Description: Servo IC servo-clock frequency divider control

Range: 0 .. 15

Units: Servo Clock Freq. = Phase Clock Freq. / (**Gate1[*i*].ServoClockDiv** +1)

Default: 3
Servo Clock Freq. = 9.0346 kHz / (3+1) = 2.2587 kHz
(with default values of **Gate[*i*].PwmPeriod** and **Gate1[*i*].PhaseClockDiv**)

Legacy I-variable alias: I7*m*02

Gate1[*i*].ServoClockDiv, in conjunction with **Gate1[*i*].PhaseClockDiv** and **Gate1[*i*].PwmPeriod**, determines the frequency of the Servo clock generated inside each PMAC2-style "DSPGATE1" Servo IC. However, only the Servo IC told to use and output its own Servo clock with **Gate1[*i*].PhaseServoDir** uses the Servo clock signal it generates. Each cycle of the Servo clock, Power PMAC updates the commanded position for each activated motor, and executes the servo algorithm to compute the command to the amplifier or the commutation algorithm.

Specifically, **Gate1[*i*].ServoClockDiv** controls how many times the Servo clock frequency is divided down from the Phase clock, whose frequency is set by **Gate1[*i*].PhaseClockDiv** and **Gate1[*i*].PwmPeriod**. The Servo clock frequency is equal to the Phase clock frequency divided by (**Gate1[*i*].ServoClockDiv** 1). **Gate1[*i*].ServoClockDiv** has a range of 0 to 15, so the frequency division can be by a factor of 1 to 16. The equation for **Gate1[*i*].ServoClockDiv** is:

$$Gate1[i].ServoClockDiv = \frac{PhaseFreq(kHz)}{ServoFreq(kHz)} - 1$$

The ratio of Phase Clock frequency to Servo Clock frequency must be an integer.

For execution of trajectories at the proper speed, **Sys.ServoPeriod** must be set properly to tell the trajectory generation software what the Servo clock cycle time is. The formula for **Sys.ServoPeriod** is:

$$Sys.ServoPeriod = \frac{1}{ServoFreq(kHz)}$$

In terms of the variables that determine the Servo clock frequency from a PMAC2-style IC, the formula for **Sys.ServoPeriod** is:

$$Sys.ServoPeriod = \frac{(2 * PwmPeriod + 3)(PhaseClockDiv + 1)(ServoClockDiv + 1)}{117{,}964.8}$$

At the default servo clock frequency, **Sys.ServoPeriod** should be set to 0.442718 in order that Power PMAC's interpolation routines use the proper servo update time.

Note: If the servo clock frequency is set too high, lower priority tasks such as communications can be starved for time. If the background tasks are completely starved, the watchdog timer will trip, shutting down the board. If a normal reset of the board does not re-establish a state where the watchdog timer has not tripped and communications works well, it will be necessary to re-initialize the board by powering up with a USB memory stick installed. This restores default settings, so communication is possible, and **Gate1[*i*].PwmPeriod**, **Gate1[*i*].PhaseClockDiv**, and **Gate1[*i*].ServoClockDiv** can be set to supportable values.

Gate1[*i*].ServoClockDiv constitutes bits 20 – 23 of the full-word element **Gate1[*i*].ClockCtrl** (bits 28 – 31 of the 32-bit element in C).

Example

With a 6.67 kHz Phase Clock frequency established by **Gate1[*i*].PwmPeriod** and **Gate1[*i*].PhaseClockDiv**, and a desired 3.33 kHz Servo Clock frequency:

Gate1[*i*].ServoClockDiv = (6.67 / 3.33) - 1 = 2 - 1 = 1

Gate1[*i*]. Channel-Specific Setup Elements

This section describes the channel-specific saved data structure elements for PMAC2-style "DSPGATE1" Servo ICs. Each of the 4 channels on the IC can be set up independently with regard to these variables.

In these setup elements, the channel index values **Chan[*j*]** for Power PMAC are one less than the channel number "*n*" in Turbo PMAC software and in the hardware documentation. Power PMAC channel index values "*j*" range from 0 to 3, and correspond to Turbo PMAC channel numbers "*n*" and hardware channel numbers 1 to 4, respectively.

Gate1[*i*].Chan[*j*].CaptCtrl

Description: Servo IC channel position-capture control

Range: 0 .. 15

Units: bit field

Default: 1

Legacy I-variable alias: I7*mn*2

Gate1[*i*].Chan[*j*].CaptCtrl determines which input signal or combination of signals for this channel of a PMAC2-style Servo IC, and which polarity, triggers a hardware position capture of the counter for the encoder of this channel. If a flag input (home, limit, or user) is used, **Gate1[*i*].Chan[*j*].CaptFlagSel** determines which flag.

Proper setup of this variable is essential for a successful homing search move or other move-until-trigger for the motor using this channel for its position-loop feedback and flags if the super-accurate hardware position capture function is used. If **Motor[*x*].CaptureMode** is set to 0, 1, or 3 to select hardware trigger (with or without hardware position capture), this variable must be set up properly.

The following settings of **Gate1[*i*].Chan[*j*].CaptCtrl** may be used:

- **Gate1[*i*].Chan[*j*].CaptCtrl** = 0: Continuous or Hall capture
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 1: Capture on Index (CHCn) high
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 2: Capture on Flag n high
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 3: Capture on (Index high AND Flag n high)
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 4: Continuous or Hall capture
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 5: Capture on Index (CHCn) low
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 6: Capture on Flag n high
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 7: Capture on (Index low AND Flag n high)
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 8: Continuous or Hall capture
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 9: Capture on Index (CHCn) high
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 10: Capture on Flag n low
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 11: Capture on (Index high AND Flag n low)
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 12: Continuous or Hall capture

- **Gate1[*i*].Chan[*j*].CaptCtrl** = 13: Capture on Index (CHCn) low
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 14: Capture on Flag n low
- **Gate1[*i*].Chan[*j*].CaptCtrl** = 15: Capture on (Index low AND Flag n low)

Note that only flags and index inputs of the same channel number as the encoder may be used for hardware capture of that encoder's position. This means that to use the hardware capture feature for the homing search move, **Motor[*x*].pEncCtrl** must use flags of the same channel number as the encoder that **Motor[*x*].pEnc** uses for position-loop feedback.

If bits 0 and 1 of this variable are both set to 0 (value 0, 4, 8, or 12) and **Gate1[*i*].Chan[*j*].GatedIndexSel** for the same channel is 0, the capture trigger will occur immediately if the trigger is armed. If **Gate1[*i*].Chan[*j*].GatedIndexSel** for the same channel is 1, the trigger will occur on the next transition of any of the U, V, or W "Hall" input flags.

The trigger is armed when the position-capture register **Gate1[*i*].Chan[*j*].HomeCapt** is read. This sets the status element **Gate1[*i*].Chan[*j*].PosCapt** to 0, indicating that the trigger is armed, but the next trigger has not occurred. In Power PMAC's move-until-trigger functions, this read is performed automatically at the beginning of the move so the trigger is always armed. After this, as soon as the Servo IC hardware sees that the specified input lines are in the specified states, the trigger will occur – it is level-triggered, not edge-triggered. No transition is required.

Gate1[*i*].Chan[*j*].CaptCtrl constitutes bits 4 – 7 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bits 12 – 15 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].CaptFlagSel

Description: Servo IC channel position-capture flag select

Range: 0 .. 3

Units: Enumeration

Default: 0

Legacy I-variable alias: I7*mn*3

Gate1[*i*].Chan[*j*].CaptFlagSel determines which of the four "flag" inputs for the channel will be used for hardware position capture (if one is used) of the channel's encoder counter on a PMAC2-style Servo IC. **Gate1[*i*].Chan[*j*].CaptCtrl** determines whether a flag is used and which polarity of the flag will cause the trigger. The possible values of **Gate1[*i*].Chan[*j*].CaptFlagSel** and the flag each selects is:

- **Gate1[*i*].Chan[*j*].CaptFlagSel** = 0: HOMEn (Home Flag n)
- **Gate1[*i*].Chan[*j*].CaptFlagSel** = 1: PLIMn (Positive End Limit Flag n)
- **Gate1[*i*].Chan[*j*].CaptFlagSel** = 2: MLIMn (Negative End Limit Flag n)
- **Gate1[*i*].Chan[*j*].CaptFlagSel** = 3: USERn (User Flag n)

Gate1[*i*].Chan[*j*].CaptFlagSel is typically set to 0 for homing search moves in order to use the home flag for the channel. It is commonly set to 3 afterwards to select the User flag if other uses

of the hardware position capture function are desired, such as for probing and registration. If you wish to capture on the PLIMn or MLIMn overtravel limit flags, you probably will want to disable their normal shutdown functions by temporarily setting **Motor[*x*].pLimits** to 0.

Gate1[*i*].Chan[*j*].CaptFlagSel constitutes bits 8 – 9 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bits 16 – 17 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].Ctrl

Description: Servo IC channel full-word control element

Range: $0 .. $FFFFFF

Units: Bit field

Default: $C00017

Gate1[*i*].Chan[*j*].Ctrl is the full-word element that comprises the setup elements for the signal interfaces for the channel of the IC. It is comprised of the following partial-word elements:

Partial-Word Element	Script Bits	C Bits	Functionality
OutputMode	23 – 22	31 – 30	Output mode select
OutputPol	21 – 20	29 – 28	Output polarity control
PfmDirPol	19	27	PFM direction output polarity control
OneOverTEna	18	26	Hardware 1/T enable
IndexGateState	17 – 16	25 – 24	Gated index capture state control
GatedIndexSel	15	23	Gated index capture select
AmpEna	14	22	Amplifier enable output state
Equ1Ena	13	21	Compare circuit encoder select
EquWrite	12 – 11	20 – 19	Compare output force state
PosClear	10	18	Encoder counter reset control
CaptFlagSel	09 – 08	17 – 16	Position-capture flag select
CaptCtrl	07 – 04	15 – 12	Position-capture control
EncCtrl	03 – 00	11 – 08	Encoder decode control
(*null*)	..	07 – 00	(*No hardware present*)

Those elements shown in bold are saved setup elements and are documented individually in this chapter. Those not in bold are not saved, and are documented in the chapter on Non-Saved Setup Elements.

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word, even though there are only 24 bits in the physical register. These 24 bits are found at the high end of the 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate1[*i*].Chan[*j*].EncCtrl

Description: Servo IC channel encoder decode control

Range: 0 .. 15

Units: none

Default: 7

Legacy I-variable alias: I7*mn*0

Gate1[*i*].Chan[*j*].EncCtrl controls how the encoder input signal for the channel on a PMAC2-style Servo IC is decoded into counts. As such, this defines the sign and magnitude of a "count". The following settings may be used to decode an input signal.

- **Gate1[*i*].Chan[*j*].EncCtrl** = 0: Pulse and direction CW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 1: x1 quadrature decode CW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 2: x2 quadrature decode CW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 3: x4 quadrature decode CW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 4: Pulse and direction CCW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 5: x1 quadrature decode CCW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 6: x2 quadrature decode CCW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 7: x4 quadrature decode CCW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 8: Internal pulse and direction
- **Gate1[*i*].Chan[*j*].EncCtrl** = 9: Not used
- **Gate1[*i*].Chan[*j*].EncCtrl** = 10: Not used
- **Gate1[*i*].Chan[*j*].EncCtrl** = 11: x6 hall-format decode CW
- **Gate1[*i*].Chan[*j*].EncCtrl** = 12: MLDT pulse timer control (internal pulse resets timer; external pulse latches timer)
- **Gate1[*i*].Chan[*j*].EncCtrl** = 13: Not used
- **Gate1[*i*].Chan[*j*].EncCtrl** = 14: Not used
- **Gate1[*i*].Chan[*j*].EncCtrl** = 15: x6 hall-format decode CCW

In any of the quadrature-decode modes, the Servo IC is expecting two input waveforms on CHAn and CHBn, each with approximately 50% duty cycle, and approximately one-quarter of a cycle out of phase with each other. "Times-one" (x1) decode provides one count per cycle; x2 provides two counts per cycle; and x4 provides four counts per cycle. The vast majority of users select x4 decode to get maximum resolution.

The "clockwise" (CW) and "counterclockwise" (CCW) options simply control which direction counts up. If you get the wrong direction sense, simply change to the other option (e.g. from 7 to 3 or vice versa).

WARNING

Changing the direction sense of the decode for the feedback encoder of a motor that is operating properly will result in unstable positive feedback and a dangerous runaway condition in the absence of other changes. The output polarity must be changed as well to re-establish polarity match for stable negative feedback.

In the pulse-and-direction decode modes, the Servo IC is expecting the pulse train on CHAn, and the direction (sign) signal on CHBn. If the signal is unidirectional, the CHBn line can be allowed to pull up to a high state, or it can be hardwired to a high or low state.

If **Gate1[*i*].Chan[*j*].EncCtrl** is set to 8, the decoder inputs the pulse and direction signal generated by the channel's own pulse frequency modulator (PFM) output circuitry. This permits the PMAC2 to create a phantom closed loop when driving an open-loop stepper system. *No jumpers or cables are needed to do this; the connection is entirely within the Servo IC.* The counter polarity automatically matches the PFM output polarity.

If **Gate1[*i*].Chan[*j*].EncCtrl** is set to 11 or 15, the channel is expecting three Hall-sensor format inputs on CHAn, CHBn, and CHCn, each with approximately 50% duty cycle, and approximately one-third (120°e) of a cycle out of phase with each other. The decode circuitry will generate one count on each edge of each signal, yielding 6 counts per signal cycle ("x6 decode"). The difference between 11 and 15 is which direction of signal causes the counter to count up.

If **Gate1[*i*].Chan[*j*].EncCtrl** is set to 12, the timer circuitry is set up to read magnetostrictive linear displacement transducers (MLDTs) such as Temposonics™. In this mode, the timer is cleared when the PFM circuitry sends out the excitation pulse to the sensor on PULSEn, and it is latched into the memory-mapped register when the excitation pulse is received on CHAn.

Gate1[*i*].Chan[*j*].EncCtrl constitutes bits 0 – 3 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bits 8 – 11 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].Equ1Ena

Description: Servo IC channel compare circuit encoder select

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: I7*mn*1

Gate1[*i*].Chan[*j*].Equ1Ena controls which channel's encoder counter is tied to the position compare circuitry for the channel on a PMAC2-style Servo IC. It has the following possible settings:

- **Gate1[*i*].Chan[*j*].Equ1Ena** = 0: Use channel's own encoder counter for position compare function

- **Gate1[*i*].Chan[*j*].Equ1Ena** = 1: Use IC's first-channel encoder counter for position compare function

When **Gate1[*i*].Chan[*j*].Equ1Ena** is set to 0, the channel's position compare registers are tied to the channel's own encoder counter, and the position compare signal appears only on the EQU output for that channel.

When **Gate1[*i*].Chan[*j*].Equ1Ena** is set to 1, the channel's position compare register is tied to the first encoder counter on the Servo IC, and the position compare signal appears both on this channel's own EQU output, and combined into the EQU output for Channel 1 on the Servo IC; executed as a logical OR.

Gate1[*i*].Chan[0].Equ1Ena performs no effective function, so is always 1. It cannot be set to 0.

Gate1[*i*].Chan[*j*].Equ1Ena constitutes bit 13 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bit 21 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].GatedIndexSel

Description: Servo IC channel gated-index capture select

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: I7*mn*4

Gate1[*i*].Chan[*j*].GatedIndexSel controls whether the raw encoder index channel input or a version of the input gated by the AB-quadrature state is used for position capture of the channel's encoder on a PMAC2-style Servo IC . It has the following possible settings:

- **Gate1[*i*].Chan[*j*].GatedIndexSel** = 0: Use ungated index for encoder position capture

- **Gate1[*i*].Chan[*j*].GatedIndexSel** = 1: Use index gated by quadrature channels for position capture

When **Gate1[*i*].Chan[*j*].GatedIndexSel** is set to 0, the encoder index channel input (CHCn) is passed directly into the position capture circuitry.

When **Gate1[*i*].Chan[*j*].GatedIndexSel** is set to 1, the encoder index channel input (CHCn) is logically combined with ("gated by") the quadrature signals of Encoder n before going to the position capture circuitry. The intent is to get a "gated index" signal exactly one quadrature state wide. This provides a more accurate and repeatable capture, and makes the use of the capture function to confirm the proper number of counts per revolution very straightforward.

In order for the gated index capture to work reliably, the index pulse must reliably span one, but only one, "high-high" or "low-low" AB quadrature state of the encoder.
Gate1[*i*].Chan[*j*].IndexGateState allows you to select which of these two possibilities is used.

Note: If **Gate1[*i*].Chan[*j*].GatedIndexSel** is set to 1, but **Gate1[*i*].Chan[*j*].CaptCtrl** bit 0 and bit 1 are set to 0, so the index is not used in the position capture, then the encoder position is captured on the first edge of any of the U, V, or W flag inputs for the channel. In this case, **Gate1[*i*].Chan[*j*].HallState** (bits 0, 1, and 2 of the channel status word) tell what hall-state edge caused the capture.

Gate1[*i*].Chan[*j*].GatedIndexSel constitutes bit 15 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bit 23 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].IndexGateState

Description: Servo IC channel gated-index capture state control

Range: 0 .. 3

Units: Bit field

Default: 0

Legacy I-variable alias: I7*mn*5

Gate1[*i*].Chan[*j*].IndexGateState is a 2-bit variable that controls two functions for the index signal of the channel's encoder.

When using the "gated index" feature of a PMAC2-style Servo IC for more accurate position capture (**Gate1[*i*].Chan[*j*].GatedIndexSel** = 1), bit 0 (value 1) of **Gate1[*i*].Chan[*j*].IndexGateState** specifies whether the raw index-channel signal fed into the encoder input of the Servo IC is passed through to the position capture signal only on the "high-high" quadrature state (bit 0 = 0), or only on the "low-low" quadrature state (bit 0 = 1).

Bit 1 (value 2) of **Gate1[*i*].Chan[*j*].IndexGateState** controls whether the Servo IC "de-multiplexes" the index pulse and the 3 hall-style commutation states from the third channel based on the quadrature state, as with Yaskawa incremental encoders. If bit 1 is set to 0, this de-multiplexing function is not performed, and the signal on the "C" channel of the encoder is used as the index only. If bit 1 is set to 1, the Servo IC breaks out the third-channel signal into four separate values, one for each of the four possible AB-quadrature states. The de-multiplexed hall commutation states can be used to provide power-on phase position.

Note: Immediately after power-up, the Yaskawa encoder automatically cycles its AB outputs forward and back through a full quadrature cycle to ensure that all of the hall commutation states are available to the controller before any movement is started. However, if the encoder is powered up at the same time as the Turbo PMAC, this will happen before the Servo IC is ready to accept these signals. Bit 2 of the channel's status word, "Invalid De-multiplex", will be set to 1 if the Servo IC has not seen all of these states when it was ready for them. To use this feature, it is recommended that the power to the encoder be provided through a software-controlled relay to

ensure that valid readings of all states have been read before using these signals for power-on phasing.

Gate1[*i*].Chan[*j*].IndexGateState has the following possible settings:

- **Gate1[*i*].Chan[*j*].IndexGateState** = 0: Gate index with "high-high" quadrature state (GI = A & B & C), no demux
- **Gate1[*i*].Chan[*j*].IndexGateState** = 1: Gate index with "low-low" quadrature state (GI = A/ & B/ & C), no demux
- **Gate1[*i*].Chan[*j*].IndexGateState** = 2 or 3: De-multiplex hall and index from third channel, gating irrelevant

Gate1[*i*].Chan[*j*].IndexGateState constitutes bits 16 – 17 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bits 24 – 25 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].OneOverTEna

Description: Servo IC channel hardware 1/T enable

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: I7*mn*9

Gate1[*i*].Chan[*j*].OneOverTEna controls whether the "hardware-1/T" functionality is enabled for the channel of a PMAC2-style Servo IC. If **Gate1[*i*].Chan[*j*].OneOverTEna** is set to the default value of 0, the hardware-1/T functionality is disabled, permitting the use of the "software-1/T" position extension that is calculated by default with encoder conversion method 3. If **Gate1[*i*].Chan[*j*].OneOverTEna** is set to 1, the hardware-1/T functionality is enabled, and the software-1/T cannot be used.

When the hardware-1/T functionality is enabled, the IC computes a new fractional-count position estimate based on timers every SCLK (encoder sample clock) cycle. This permits the fractional count data to be used for hardware capture and compare functions, enhancing their resolution. This is particularly useful when the IC is used on an ACC-51 high-resolution analog-encoder interpolator board. However, it replaces the timer registers for the channel with fractional count position data, so the traditional software-1/T method of the conversion table cannot work if this is enabled.

Use of the hardware 1/T interpolation of the PMAC2-style Servo ICs is not presently supported for servo position by the Power PMAC firmware.

Gate1[*i*].Chan[*j*].OneOverTEna constitutes bit 18 of the full-word element **Gate1[*j*].Chan[*j*].Ctrl** (bit 26 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].OutputMode

Description: Servo IC channel output mode select

Range: 0 .. 3

Units: Bit field

Default: 0

Legacy I-variable alias: I7*mn*6

Gate1[*i*].Chan[*j*].OutputMode controls what output formats are used on the command output signal lines for the channel of a PMAC2-style Servo IC. It has the following possible settings:

- **Gate1[*i*].Chan[*j*].OutputMode** = 0: Outputs A & B are PWM; Output C is PWM
- **Gate1[*i*].Chan[*j*].OutputMode** = 1: Outputs A & B are DAC; Output C is PWM
- **Gate1[*i*].Chan[*j*].OutputMode** = 2: Outputs A & B are PWM; Output C is PFM
- **Gate1[*i*].Chan[*j*].OutputMode** = 3: Outputs A & B are DAC; Output C is PFM

If a three-phase direct PWM command format is desired, **Gate1[*i*].Chan[*j*].OutputMode** should be set to 0. If signal outputs for (external) digital-to-analog converters are desired, **Gate1[*i*].Chan[*j*].OutputMode** should be set to 1 or 3. In this case, the C output can be used as a supplemental (non-servo) output in either PWM or PFM form. For example, it can be used to excite an MLDT sensor (e.g. Temposonics™) in PFM form.

Gate1[*i*].Chan[*j*].OutputMode constitutes bits 22 – 23 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bits 30 – 31 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].OutputPol

Description: Servo IC channel output polarity control

Range: 0 .. 3

Units: Bit field

Default: 0

Legacy I-variable alias: I7*mn*7

Gate1[*i*].Chan[*j*].OutputPol controls the high/low polarity of the command output signals for the channel on a PMAC2-style Servo IC. It has the following possible settings:

- **Gate1[*i*].Chan[*j*].OutputPol** = 0: Do not invert Outputs A & B; Do not invert Output C
- **Gate1[*i*].Chan[*j*].OutputPol** = 1: Invert Outputs A & B; Do not invert Output C
- **Gate1[*i*].Chan[*j*].OutputPol** = 2: Do not invert Outputs A & B; Invert Output C
- **Gate1[*i*].Chan[*j*].OutputPol** = 3: Invert Outputs A & B; Invert Output C

The default non-inverted outputs are high true. For PWM signals on Outputs A, B, and C, this means that the transistor-on signal is high. Delta Tau PWM-input amplifiers, and most other PWM-input amplifiers, expect this non-inverted output format. For such a 3-phase motor drive, **Gate1[*i*].Chan[*j*].OutputPol** should be set to 0.

Caution

If the high/low polarity of the PWM signals is wrong for a particular amplifier, what was intended to be deadtime between top and bottom on-states as set by **Gate1[*i*].PwmDeadTime** becomes overlap. If the amplifier-input circuitry does not lock this out properly, this causes an effective momentary short circuit between bus power and ground. This would destroy the power transistors very quickly.

For PFM signals on Output C, non-inverted means that the pulse-on signal is high (direction polarity is controlled by **Gate1[*i*].Chan[*j*].PfmDirPol**). During a change of direction, the direction bit will change synchronously with the leading edge of the pulse, which in the non-inverted form is the rising edge. If the drive requires a set-up time on the direction line before the rising edge of the pulse, the pulse output can be inverted so that the rising edge is the trailing edge, and the pulse width (established by **Gate1[*i*].PwmDeadTime**) is the set-up time.

For DAC signals on Outputs A and B, non-inverted means that a 1 value to the DAC is high. DACs used on Delta Tau accessory boards, as well as all other known DACs always expect non-inverted inputs, so **Gate1[*i*].Chan[*j*].OuputPol** should always be set to 0 or 2 when using DACs on this channel.

WARNING

Changing the high/low polarity of the digital data to the DACs has the effect of inverting the voltage sense of the DACs' analog outputs. This changes the polarity match between output and feedback. If the feedback loop had been stable with negative feedback, this change would create destabilizing positive feedback, resulting in a dangerous runaway condition.

Gate1[*i*].Chan[*j*].OutputPol constitutes bits 20 – 21 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bits 28 – 29 of the 32-bit element in C).

Gate1[*i*].Chan[*j*].PfmDirPol

Description: Servo IC channel PFM direction output polarity control

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: I7*mn*8

Gate1[*i*].Chan[*j*].PfmDirPol controls the polarity of the direction output signal in the pulse-and-direction format for the channel of a PMAC2-style Servo IC. It is only active if **Gate1[*i*].Chan[*j*].OutputMode** has been set to 2 or 3 to use Output C as a pulse-frequency-modulated (PFM) output. It has the following possible settings:

- **Gate1[*i*].Chan[*j*].PfmDirPol** = 0: Do not invert direction signal (+ = low; - = high)
- **Gate1[*i*].Chan[*j*].PfmDirPol** = 1: Invert direction signal (- = low; + = high)

If **Gate1[*i*].Chan[*j*].PfmDirPol** is set to the default value of 0, a positive direction command provides a low output; if **Gate1[*i*].Chan[*j*].PfmDirPol** is set to 1, a positive direction command provides a high output.

Gate1[*i*].Chan[*j*].PfmDirPol constitutes bit 19 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bit 27 of the 32-bit element in C).

Gate2[*i*]. (PMAC2-Style MACRO IC) Saved Data Structure Elements

This section describes the saved data structure elements for PMAC2-style "DSPGATE2" MACRO ICs. Depending on how they are set up, and on the hardware around them, these ICs support MACRO, servo, and general-purpose I/O functions.

In the Power PMAC script environment, it is also possible to use the name of the accessory on which the IC is present as the structure name instead of "**Gate2[*i*]**. Presently, the only Power PMAC accessory using the DSPGATE2 IC is the ACC-5E, so the alias name **Acc5E[*i*].** can be used instead.

Gate2[*i*]. Multi-Channel Setup Elements

This section describes the "multi-channel" saved data structure elements for PMAC2-style "DSPGATE1" Servo ICs. These elements both channels on the IC, or are for other functions of the IC. These include those that control the general-purpose I/O and the MACRO functionality.

Gate2[*i*].AdcStrobe

Description: MACRO IC ADC strobe word

Range: $000000 .. $FFFFFF

Units: Serial data stream (MSB first, starting on rising edge of phase clock)

Default: $3FFFFF

Legacy I-variable alias: I6*m*06, I6*m*56

Gate2[*i*].AdcStrobe controls the ADC strobe signal for all machine interface channels on the MACRO IC. The 24-bit word set by this variable is shifted out serially on the ADC_STROB lines, MSB first, one bit per ADC_CLK cycle starting on the rising edge of the phase clock.

The first bit that is a "1" creates a rising edge on the ADC_STROB output that is typically used as a "start-convert" signal. Some A/D converters just need this rising edge for the conversion; others need the signal to stay high all of the way through the conversion. Intermediate bits of the ADC_STROB output can be used to transmit other information in some applications.

Gate2[*i*].ClockCtrl

Description: MACRO IC full-word clock control element

Range: $0 .. $FFFFFF

Units: Bit field

Default: $3008D2

Gate2[*i*].ClockCtrl is the full-word element that comprises the multi-channel setup for generating clock signals for the IC. It is comprised of the following partial-word elements:

Partial-Word Element	Script Bits	C Bits	Functionality
ServoClockDiv	23 – 20	31 – 28	Servo clock frequency divider control
PhaseClockDiv	19 – 16	27 – 24	Phase clock frequency divider control
(reserved)	15 – 14	23 – 22	(Reserved for future use)
PhaseServoDir	13 – 12	21 – 20	Phase and servo clock direction control
HardwareClockCtrl	11 – 00	19 – 08	Hardware clock frequency divider control
(null)	..	07 – 00	(no hardware present)

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word, even though there are only 24 bits in the physical register. These 24 bits are found at the high end of the 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate2[*i*].DacStrobe

Description: MACRO IC DAC strobe word

Range: $000000 .. $FFFFFF

Units: Serial data stream (MSB first, starting on rising edge of phase clock)

Default: $7FFFC0 (for 18-bit DAC)

Legacy I-variable alias: I6*m*05, I6*m*55

Gate2[*i*].DacStrobe controls the DAC strobe signal for machine interface channels on the selected PMAC2-style "DSPGATE2" MACRO IC. The 24-bit word set by **Gate2[*i*].DacStrobe** is shifted out serially on the DAC_STROB lines, MSB first, one bit per DAC_CLK cycle starting on the rising edge of the phase clock. The value in the LSB is held until the next phase clock cycle.

For typical *n*-bit DACs, the strobe line is held high for *n*-1 clock cycles. Therefore, the common settings of this variable are:

- 18-bit DACs: $7FFFC0 (high for 17 clock cycles)
- 16-bit DACs: $7FF00 (high for 15 clock cycles)

- 12-bit DACs: $7FF000 (high for 11 clock cycles)

The default **Gate2[*i*].DacStrobe** value of $7FFFC0 is suitable for the 18-bit DACs on Delta Tau products. **Gate2[*i*].DacStrobe** should not be changed from the default unless different DACs are used.

Gate2[*i*].DispDir

Description: MACRO IC Direction Control for Display Port Lines

Range: $000 … $FFF

Units: Bit field

Default: $FFF

Gate2[*i*].DispDir controls the direction of input/output for the signals DISP0 – DISP7 and CTRL0 – CTRL3 of the "display port" on the DSPGATE2 IC. This element is a set of 12 bits, with each bit controlling the direction of one pin. Bits 0 – 7 control the direction of DISP0 – DISP7, respectively. Bits 8 – 11 control the direction of CTRL0 – CTRL3, respectively.

If the bit is set to a value of 0, the signal is an input. If the bit is set to 1, the signal is an output.

Gate2[*i*].DispMode

Description: MACRO IC Mode Control for Display Port Lines

Range: $000 .. $FFF

Units: Bit field

Default: $FFF

Gate2[*i*].DispMode is the control register for the uses of on the pins DISP0 – DISP7 and CTRL0 – CTRL3 of the "display port" on the DSPGATE2 IC. This element is a set of 12 bits, with each bit controlling the use of one pin. Presently there is no "servo" use for these pins, so these bits are forced to "1", making the 12-bit register value always equal to $FFF.

Gate2[*i*].DispPol

Description: MACRO IC Direction Control for Display Port Lines

Range: $000 .. $FFF

Units: Bit field

Default: $000

Gate2[*i*].DispPol controls the polarity of input/output for the signals DISP0 – DISP7 and CTRL0 – CTRL3 of the "display port" on the DSPGATE2 IC. This element is a set of 12 bits, with each bit controlling the polarity of one pin. Bits 0 – 7 control the polarity of DISP0 – DISP7, respectively. Bits 8 – 11 control the polarity of CTRL0 – CTRL3, respectively. The bit is used regardless of whether the pin is used as an input or output.

If the bit is set to a value of 0, the signal is non-inverting (high voltage = 1). If the bit is set to 1, the signal is inverting (low voltage = 1).

Gate2[*i*].GrayCodeBitLenCtrl

Description: MACRO IC Gray-to-Binary Conversion Bit-Length Control

Range: 0 .. $10 (0 .. 16)

Units: Bits

Default: 0

Gate2[*i*].GrayCodeBitLenCtrl controls how the Gray-code-to-binary conversion is performed on the input from data lines IO00 – IO23 on the DSPGATE2 IC. These 24 inputs can be split into two separate Gray-code words. This element specifies the bit length of the less-significant word. A value of 0 means that the word is not split at all. The unconverted Gray-code data can be found in **Gate2[*i*].LowIoData**; the data converted to numerical binary format can be found in **Gate2[*i*].LowIoGrayData**.

Gate2[*i*].HardwareClockCtrl

Description: MACRO IC hardware clocks frequency control

Range: 0 .. 4095

Units: Individual Clock Dividers
Gate2[*i*].HardwareClockCtrl = Encoder SCLK Divider
+ 8 * PFM_CLK Divider
+ 64 * DAC_CLK Divider
+ 512 * ADC_CLK Divider
where:
Encoder SCLK Frequency = 39.3216 MHz / (2 ^ Encoder SCLK Divider)
PFM_CLK Frequency = 39.3216 MHz / (2 ^ PFM_CLK Divider)
DAC_CLK Frequency = 39.3216 MHz / (2 ^ DAC_CLK Divider)
ADC_CLK Frequency = 39.3216 MHz / (2 ^ ADC_CLK Divider)

Default: 2258 = 2 + (8 * 2) + (64 * 3) + (512 * 4)
Encoder SCLK Frequency = 39.3216 MHz / (2 ^ 2) = 9.8304 MHz
PFM_CLK Frequency = 39.3216 MHz / (2 ^ 2) = 9.8304 MHz
DAC_CLK Frequency = 39.3216 MHz / (2 ^ 3) = 4.9152 MHz
ADC_CLK Frequency = 39.3216 MHz / (2 ^ 4) = 2.4576 MHz

Legacy I-variable alias: I6*m*03, I6*m*53

Gate2[*i*].HardwareClockCtrl controls the frequency of four hardware clock frequencies – SCLK, PFM_CLK, DAC_CLK, and ADC_CLK – for the four machine interface channels on a PMAC2-style "DSPGATE2" MACRO IC. It is a 12-bit variable consisting of four independent 3-bit controls, one for each of the clocks. Each of these clock frequencies can be divided down from a starting 39.3216 MHz frequency by powers of 2, 2^N, from 1 to 128 times (N=0 to 7). This means that the possible frequency settings for each of these clocks are:

Frequency	Divide by	Divider N in $1/2^N$	Frequency	Divide by	Divider N in $1/2^N$
39.3216 MHz	1	0	2.4576 MHz	16	4
19.6608 MHz	2	1	1.2288 MHz	32	5
9.8304 MHz	4	2	611.44 kHz	64	6
4.9152 MHz	8	3	305.72 kHz	128	7

Very few Power PMAC users will be required to change the setting of **Gate2[*i*].HardwareClockCtrl** from the default value.

SCLK: The encoder sample clock signal SCLK controls how often the MACRO IC's digital hardware looks at the encoder and flag inputs. The MACRO IC can take at most one count per SCLK cycle, so the SCLK frequency is the absolute maximum encoder count frequency. SCLK also controls the signal propagation through the digital delay filters for the encoders and flags; the lower the SCLK frequency, the greater the noise pulse that can be filtered out. The SCLK frequency should optimally be set to the lowest value that can accept encoder counts at the maximum possible rate.

PFM_CLK: The pulse-frequency-modulation clock PFM_CLK controls the PFM circuitry that is commonly used for stepper drives. The maximum pulse frequency possible is 1/4 of the PFM_CLK frequency. The PFM_CLK frequency should optimally be set to the lowest value that can generate pulses at the maximum frequency required.

DAC_CLK: The DAC_CLK controls the serial data frequency into D/A converters. If these converters are on Delta Tau-provided accessories, the DAC_CLK setting should be left at the default value.

ADC_CLK: The ADC_CLK controls the serial data frequency from A/D converters. If these converters are on Delta Tau-provided accessories, the ADC_CLK setting should be left at the default value.

To determine the clock frequencies set by a given value of **Gate2[*i*].HardwareClockCtrl**, use the following procedure:

1. Divide **Gate2[*i*].HardwareClockCtrl** by 512 and round down to the nearest integer. This value N1 is the ADC_CLK divider.
2. Multiply N1 by 512 and subtract the product from **Gate2[*i*].HardwareClockCtrl** to get **Gate2[*i*].HardwareClockCtrl'**. Divide **Gate2[*i*].HardwareClockCtrl'** by 64 and round down to the nearest integer. This value N2 is the DAC_CLK divider.

3. Multiply N2 by 64 and subtract the product from **Gate2[*i*].HardwareClockCtrl'** to get **Gate2[*i*].HardwareClockCtrl"**. Divide **Gate2[*i*].HardwareClockCtrl"** by 8 and round down to the nearest integer. This value N3 is the PFM_CLK divider.
4. Multiply N3 by 8 and subtract the product from **Gate2[*i*].HardwareClockCtrl"**. The resulting value N4 is the SCLK divider.

Gate2[*i*].HardwareClockCtrl constitutes bits 0 – 11 of the full-word element **Gate2[*i*].ClockCtrl** (bits 8 – 19 of the 32-bit element in C).

Examples

The maximum encoder count frequency in the application is 800 kHz, so the 1.2288 MHz SCLK frequency is chosen. A pulse train up to 500 kHz needs to be generated, so the 2.4576 MHz PFM_CLK frequency is chosen. The default serial DACs and ADCs provided by Delta Tau are used, so the default DAC_CLK frequency of 4.9152 MHz and the default ADC_CLK frequency of 2.4576 MHz are chosen. From the table:

SCLK Divider N: 5
PFM_CLK Divider N: 4
DAC_CLK Divider N: 3
ADC_CLK Divider N: 4

Gate2[*i*].HardwareClockCtrl = 5 + (8 * 4) + (64 * 3) + (512 * 4) = 5 + 32 + 192 + 2048 = 2277

Gate2[*i*].HardwareClockCtrl has been set to 3429. What clock frequencies does this set?

N1 = INT (3429/512) = 6	ADC_CLK = 611.44 kHz
Gate2[*i*].HardwareClockCtrl' = 3429 - (512*6) = 357	
N2 = INT (357/64) = 5	DAC_CLK = 1.2288 MHz
Gate2[*i*].HardwareClockCtrl" = 357 - (64*5) = 37	
N3 = INT (37/8) = 4	PFM_CLK = 2.4576 MHz
N4 = 37 - (8*4) = 5	SCLK = 1.2288 MHz

Gate2[*i*].HighIoDir

Description: MACRO IC Direction Control for IO24-31

Range: $00 .. $FF

Units: Bit field

Default: $00

Gate2[*i*].HighIoDir controls the direction of input/output for the signals IO24 – IO31 on the DSPGATE2 IC. This element is a set of 8 bits, with each bit controlling the direction of one pin. Bit *n* controls the use of the IO(*n*+24) pin.

If the bit is set to a value of 0, the signal is an input. If the bit is set to 1, the signal is an output.

Gate2[*i*].HighIoMode

Description: MACRO IC Mode Control for IO24-31

Range: $00 .. $FF

Units: Bit field

Default: $00

Gate2[*i*].HighIoMode controls what signals are used on the pins IO24 – IO31 on the DSPGATE2 IC. This element is a set of 8 bits, with each bit controlling the direction of one pin. Bit *n* controls the use of the IO(*n*+24) pin. Presently, there is no alternate use for these pins, so the setting of this element does not matter.

Gate2[*i*].HighIoPol

Description: MACRO IC Polarity Control for IO24 -31

Range: $00 .. $FF

Units: Bit field

Default: $00

Gate2[*i*].HighIoPol controls the polarity of input/output for the signals IO24 – IO31 on the DSPGATE2 IC. This element is a set of 8 bits, with each bit controlling the polarity of one pin. Bit *n* controls the direction of the IO(*n*+24) pin. The bit is used regardless of whether the pin is used as an input or output.

If the bit is set to a value of 0, the signal is non-inverting (high voltage = 1). If the bit is set to 1, the signal is inverting (low voltage = 1).

Gate2[*i*].LowIoDir

Description: MACRO IC Direction Control for IO00-23

Range: $000000 .. $FFFFFF

Units: Bit field

Default: $000000

Gate2[*i*].LowIoDir controls the direction of input/output for the signals IO00 – IO23 on the DSPGATE2 IC. This element is a set of 24 bits, with each bit controlling the use of one pin. Bit *n* controls the direction of the IO*n* pin. The bit is only used if the corresponding bit of **Gate2[*i*].LowIoMode** is set to 1 to specify the pin's use as general-purpose I/O.

If the bit is set to a value of 0, the signal is an input. If the bit is set to 1, the signal is an output.

Gate2[*i*].LowIoMode

Description: MACRO IC Mode Control for IO00-23

Range: $000000 .. $FFFFFF

Units: Bit field

Default: $FFFFFFF

Gate2[*i*].LowIoMode controls what signals are used on the pins IO00 – IO23 on the DSPGATE2 IC. This element is a set of 24 bits, with each bit controlling the use of one pin. Bit *n* controls the use of the IO*n* pin.

If the bit is set to a value of 0, the pin is used for its "servo" function. If the bit is set to 1, the pin is used for general-purpose I/O.

The following list shows what each bit controls:

Bits:	0	I/O00 Data Type Control (0=FlagW1; 1=I/O00)
	1	I/O01 Data Type Control (0=FlagV1; 1=I/O01)
	2	I/O02 Data Type Control (0=FlagU1; 1=I/O02)
	3	I/O03 Data Type Control (0=FlagT1; 1=I/O03)
	4	I/O04 Data Type Control (0=USER1; 1=I/O04)
	5	I/O05 Data Type Control (0=MLIM1; 1=I/O05)
	6	I/O06 Data Type Control (0=PLIM1; 1=I/O06)
	7	I/O07 Data Type Control (0=HMFL1; 1=I/O07)
	8	I/O08 Data Type Control (0=PWM_B_BOT1; 1=I/O08)
	9	I/O09 Data Type Control (0=PWM_B_TOP1; 1=I/O09)
	10	I/O10 Data Type Control (0=PWM_A_BOT1; 1=I/O10)
	11	I/O11 Data Type Control (0=PWM_A_TOP1; 1=I/O11)
	12	I/O12 Data Type Control (0=PWM_B_BOT2; 1=I/O12)
	13	I/O13 Data Type Control (0=PWM_B_TOP2; 1=I/O13)
	14	I/O14 Data Type Control (0=PWM_A_BOT2; 1=I/O14)
	15	I/O15 Data Type Control (0=PWM_A_TOP2; 1=I/O15)
	16	I/O16 Data Type Control (0=HMFL2; 1=I/O16)
	17	I/O17 Data Type Control (0=PLIM2; 1=I/O17)
	18	I/O18 Data Type Control (0=MLIM2; 1=I/O18)
	19	I/O19 Data Type Control (0=USER2; 1=I/O19)
	20	I/O20 Data Type Control (0=FlagT2; 1=I/O20)
	21	I/O21 Data Type Control (0=FlagU2; 1=I/O21)
	22	I/O22 Data Type Control (0=FlagV2; 1=I/O22)
	23	I/O23 Data Type Control (0=FlagW2; 1=I/O23)

Gate2[*i*].LowIoPol

Description: MACRO IC Polarity Control for IO00-23

Range: $000000 – $FFFFFF

Units: Bit field

Default: $000000

Gate2[*i*].LowIoPol controls the polarity of input/output for the signals IO00 – IO23 on the DSPGATE2 IC. This element is a set of 24 bits, with each bit controlling the polarity of one pin. Bit *n* controls the direction of the IO*n* pin. The bit is only used if the corresponding bit of **Gate2[*i*].LowIoMode** is set to 1 to specify the pin's use as general-purpose I/O. However, it is used regardless of whether the pin is used as an input or output.

If the bit is set to a value of 0, the signal is non-inverting (high voltage = 1). If the bit is set to 1, the signal is inverting (low voltage = 1).

Gate2[*i*].MacroEnable

Description: MACRO IC Node Enable Control

Range: $000000 .. $FFFFFF

Units: Bit field

Default: $000000

Legacy I-variable alias: I6*m*41, I6*m*91

Gate2[*i*].MacroEnable controls which of the 16 MACRO nodes on the MACRO IC are activated. It also controls the master station number of the IC, and the node number of the packet that creates a synchronization signal. The bits are arranged as follows:

Bit #	Value	Type	Function
0	1($1)	Config	Node 0 Activate
1	2($2)	Config	Node 1 Activate
2	4($4)	Config	Node 2 Activate
3	8($8)	Config	Node 3 Activate
4	16($10)	Config	Node 4 Activate
5	32($20)	Config	Node 5 Activate
6	64($40)	Config	Node 6 Activate
7	128($80)	Config	Node 7 Activate
8	256($100)	Config	Node 8 Activate
9	512($200)	Config	Node 9 Activate
10	1024($400)	Config	Node 10 Activate
11	2048($800)	Config	Node 11 Activate
12	4096($1000)	Config	Node 12 Activate
13	8192($2000)	Config	Node 13 Activate
14	16384($4000)	Config	Node 14 Activate
15	32768($8000)	Config	Node 15 Activate
16-19	$X0000	Config	Packet Sync Node Slave Address (X=0-F)
20-23	$X00000	Config	Master Station Number (X=0-F)

Bits 0 to 15 are individual control bits for the matching node number 0 to 15. If the bit is set to 1, the node is activated; if the bit is set to 0, the node is de-activated.

If the MACRO IC is a master station (likely) as determined by **Gate2[*i*].MacroMode**, it will send out a packet for each activated node every ring cycle (every phase cycle). When it receives a packet for an activated node, it will latch in that packet and not pass anything on.

If the MACRO IC is a slave station (unlikely but possible) as determined by **Gate2[*i*].MacroMode**, when it receives a packet for an activated node, it will latch in the contents of that packet into its read registers for that node address, and automatically substitute the contents of its write registers into the packet.

If a node is disabled, whether master or slave, it will still latch in the contents of a packet it receives, but it will also pass on the packet unchanged. This feature is particularly useful for the MACRO broadcast feature, in which multiple stations need to receive the same packet.

Bits 16-19 together specify the slave number part of the packet address (0-15) that will cause a sync lock pulse on the card, if this function is enabled by **Gate2[*i*].MacroMode**. This function is useful for a Power PMAC that is a slave or non-synchronizing master on the ring, to keep it locked to the synchronizing master. If the master address check for this node is disabled with **Gate2[*i*].MacroMode**, only the slave number must match to create the sync lock pulse. If the master address check is left enabled, the master number part of the packet address must match the master number for the card, as set in bits 20-23 of **Gate2[*i*].MacroEnable**.

If this card is the synchronizing master, this function is not enabled, so the value of these bits does not matter; they can be left at the default of 0.

Bits 20-23 specify the master number for the MACRO IC (0-15). Each MACRO IC on a ring must have a separate master number, even multiple MACRO ICs on the same card. The number must be specified whether the card is used as a master or a slave.

Bit 15 of **Gate2[*i*].MacroEnable** is automatically set to 1 by the firmware at power-up/reset, regardless of the saved value of **Gate2[*i*].MacroEnable**.

Gate2[*i*].MacroMode

Description: MACRO IC Ring Configuration/Status

Range: $0000 .. $FFFF

Units: Bit field

Default: $408F (for first DSPGATE2)

Legacy I-variable alias: I6*m*40, I6*m*90

Gate2[*i*].MacroMode contains configuration and status bits for MACRO ring operation of the MACRO IC on the Power PMAC. There are 11 configuration bits and 5 status bits, as follows:

Bit #	Value	Type	Function
0	1($1)	Status	Data Overrun Error (cleared when read)
1	2($2)	Status	Byte Violation Error (cleared when read)
2	4($4)	Status	Packet Parity Error (cleared when read)
3	8($8)	Status	Packet Underrun Error (cleared when read)
4	16($10)	Config	Master Station Enable
5	32($20)	Config	Synchronizing Master Station Enable
6	64($40)	Status	Sync Node Packet Received (cleared when read)
7	128($80)	Config	Sync Node Phase Lock Enable
8	256($100)	Config	Node 8 Master Address Check Disable
9	512($200)	Config	Node 9 Master Address Check Disable
10	1024($400)	Config	Node 10 Master Address Check Disable
11	2048($800)	Config	Node 11 Master Address Check Disable
12	4096($1000)	Config	Node 12 Master Address Check Disable
13	8192($2000)	Config	Node 13 Master Address Check Disable
14	16384($4000)	Config	Node 14 Master Address Check Disable
15	32768($8000)	Config	Node 15 Master Address Check Disable

In most applications, the only important configuration bits are bits 4, 5, and 7. In every MACRO ring, there must be one and only one synchronizing master station (each MACRO IC counts as a separate station; only one MACRO IC on any card in the ring can be a synchronizing master station). For this MACRO IC, bits 4 and 5 should be set (1), but bit 7 should be clear (0). This results in a value of $30, or $xx30 if any of the high bits are to be set.

If there are more than one MACRO ICs acting as masters on the ring, the others should not be synchronizing masters, but they should be set up as regular (non-synchronizing) masters. If they are receiving the phase clock signal directly from the synchronizing master IC, bit 4 should be set (1), and bits 5 and 7 should be clear (0). This results in a value of $10, or $xx10 if any of the high bits are set.

If they are not receiving the phase clock signal directly from the synchronizing master IC, they should enable "sync node phase lock" to stay synchronized with the synchronizing master by receipt of the "sync packet". For these MACRO ICs, bit 4 should be set (1), bit 5 should be clear (0), and bit 7 should be set (1), resulting in a value of $90, or $xx90 if any of the high bits are to be set.

Bits 8-15 can be set individually to disable the "master address check" for their corresponding node numbers. This capability is for multi-master broadcast and synchronization. If the master address check is disabled, only the slave node number part of the packet address must match for a packet to be latched in. In this way, the synchronizing master can send the same data packet to multiple other master and slave stations. This common packet can be used to keep multiple stations synchronized using the sync lock function enabled with bit 7 of **Gate2[*i*].MacroMode**; the packet number is specified in **Gate2[*i*].MacroEnable** (packet 15 is suggested for this purpose).

Gate2[*i*].MuxDir

Description: MACRO IC Direction Control for Multiplexer Port Lines

Range: $0000 .. $FFFF

Units: Bit field

Default: $0000

Gate2[*i*].MuxDir controls the direction of input/output for the signals DAT0 – DAT7 and SEL0 – SEL7 of the "multiplexer port" on the DSPGATE2 IC. This element is a set of 16 bits, with each bit controlling the direction of one pin. Bits 0 – 7 control the direction of DAT0 – DAT7, respectively. Bits 8 – 15 control the direction of SEL0 – SEL7, respectively. The bit is only used if the corresponding bit of **Gate2[*i*].MuxMode** is set to 1 to specify the pin's use as general-purpose I/O.

If the bit is set to a value of 0, the signal is an input. If the bit is set to 1, the signal is an output. For use with multiplexer port accessories such as the ACC-34 I/O boards, DAT0 – DAT7 must be set to inputs, and SEL0 – SEL7 must be set to outputs, resulting in a value of $FF00. The multiplexed data on ACC-34 boards can also be accessed using the **MuxIo** data structure.

Gate2[*i*].MuxMode

Description: MACRO IC Mode Control for Multiplexer Port Lines

Range: $0000 .. $FFFF

Units: Bit field

Default: $FFFF

Gate2[*i*].MuxMode controls what signals are used on the pins DAT0 – DAT7 and SEL0 – SEL7 of the "multiplexer port" on the DSPGATE2 IC. This element is a set of 16 bits, with each bit controlling the use of one pin.

If the bit is set to a value of 0, the pin is used for its "servo" function. If the bit is set to 1, the pin is used for general-purpose I/O on the multiplexer port.

The following list shows what each bit controls:

Bits:	0	DAT0 Data Type Control (0=CHC1; 1 =DAT0)
	1	DAT1 Data Type Control (0=CHC2; 1 =DAT1)
	2	DAT2 Data Type Control (0=Fault1; 1 =DAT2)
	3	DAT3 Data Type Control (0=Fault2; 1 =DAT3)
	4	DAT4 Data Type Control (0=EQU1; 1 =DAT4)
	5	DAT5 Data Type Control (0=EQU2; 1 =DAT5)
	6	DAT6 Data Type Control (0=AENA1; 1 =DAT6)
	7	DAT7 Data Type Control (0=AENA2; 1 =DAT7)
	8	SEL0 Data Type Control (0=ADC_STROB; 1=SEL0)
	9	SEL1 Data Type Control (0=ADC_CLK; 1=SEL1)
	10	SEL2 Data Type Control (0=ADC_A1; 1=SEL2)
	11	SEL3 Data Type Control (0=ADC_B1; 1=SEL3)
	12	SEL4 Data Type Control (0=ADC_A2; 1=SEL4)
	13	SEL5 Data Type Control (0=ADC_B2; 1=SEL5)
	14	SEL6 Data Type Control (0=SCLK; 1=SEL6)

15 SEL7 Data Type Control (0=SCLK_DIR; 1=SEL7)

Gate2[*i*].MuxPol

Description: MACRO IC Polarity Control for Multiplexer Port Lines

Range: $0000 .. $FFFF

Units: Bit field

Default: $FF00

Gate2[*i*].MuxPol controls the polarity of input/output for the signals DAT0 – DAT7 and SEL0 – SEL7 of the "multiplexer port" on the DSPGATE2 IC. This element is a set of 16 bits, with each bit controlling the polarity of one pin. Bits 0 – 7 control the polarity of DAT0 – DAT7, respectively. Bits 8 – 15 control the polarity of SEL0 – SEL7, respectively. The bit is only used if the corresponding bit of **Gate2[*i*].MuxMode** is set to 1 to specify the pin's use as general-purpose I/O. However, it is used regardless of whether the pin is used as an input or output.

If the bit is set to a value of 0, the signal is non-inverting (high voltage = 1). If the bit is set to 1, the signal is inverting (low voltage = 1).

Gate2[*i*].PhaseClockDiv

Description: MACRO IC phase-clock frequency divider control

Range: 0 .. 15

Units: Phase Clock Freq. = MaxPhase Freq. / (**Gate2[*i*].PhaseClockDiv**+1)

Default: 0
Phase Clock Freq. = 9.0346 kHz / 1 = 9.0346 kHz
(with default value of **Gate2[*i*].PwmPeriod**)

Legacy I-variable alias: I6*m*01, I6*m*51

Gate2[*i*].PhaseClockDiv, in conjunction with **Gate2[*i*].PwmPeriod**, determines the frequency of the Phase clock generated inside each PMAC2-style "DSPGATE2" MACRO IC. However, only the Servo or MACRO IC told to use and output its own Phase clock with **Gate*n*[*i*].PhaseServoDir** uses the Phase clock signal it generates. Each cycle of the Phase clock, motor phase commutation and digital current-loop algorithms are performed for specified motors.

Specifically, **Gate2[*i*].PhaseClockDiv** controls how many times the Phase clock frequency is divided down from the "maximum phase" clock, whose frequency is set by **Gate2[*i*].PwmPeriod**. The Phase clock frequency is equal to the "maximum phase" clock frequency divided by (**Gate2[*i*].PhaseClockDiv**+1). **Gate2[*i*].PhaseClockDiv** has a range of 0 to 15, so the frequency division can be by a factor of 1 to 16. The equation for **Gate2[*i*].PhaseClockDiv** is

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

The ratio of MaxPhase Freq. to Phase Clock Freq. must be an integer.

Note: If the phase clock frequency is set too high, lower priority tasks such as communications can be starved for time. If the background tasks are completely starved, the watchdog timer will trip, shutting down the board. If a normal reset of the board does not re-establish a state where the watchdog timer has not tripped and communications works well, it will be necessary to re-initialize the board by powering up with a USB memory stick installed. This restores default settings, so communication is possible, and **Gate2[*i*].PwmPeriod** and **Gate2[*i*].PhaseClockDiv** can be set to supportable values.

Gate2[*i*].PhaseClockDiv constitutes bits 16 – 19 of the full-word element **Gate2[*i*].ClockCtrl** (bits 24 – 27 of the 32-bit element in C).

Example

With a 20 kHz MaxPhase Clock frequency established by **Gate2[*i*].PwmPeriod**, and a desired 6.67 kHz PHASE clock frequency, the ratio between MaxPhase and Phase is 3:

Gate2[*i*].PhaseClockDiv = (20 / 6.67) - 1 = 3 -1 = 2

Gate2[*i*].PhaseServoDir

Description: MACRO IC Phase/Servo Clock Direction

Range: 0 .. 3

Units: Bit field

Default: IC- and system-dependent

Legacy I-variable alias: I6*m*07, I6*m*57

Gate2[*i*].PhaseServoDir controls whether the MACRO IC uses its own internally generated Phase and Servo clock signals as controlled by **Gate2[*i*].PwmPeriod**, **Gate2[*i*].PhaseClockDiv**, and **Gate2[*i*].ServoClockDiv**, or whether it uses Phase and Servo clock signals from an outside source.

In any Power PMAC system, there must be one and only one source of servo and phase clock signals for the system – either one of the Servo ICs or MACRO ICs, or a source external to the system.

Gate2[*i*].PhaseServoDir is a 2-bit value. Bit 0 is set to 0 for the IC to use its own Phase clock signal and output it; it is set to 1 to use an externally input Phase clock signal. Bit 1 is set to 1 for the IC to use its own Servo clock signal and output it; it is set to 1 to use an externally input Servo clock signal. This yields 4 possible values for **Gate2[*i*].PhaseServoDir**:

- **Gate2[*i*].PhaseServoDir** = 0: Internal Phase clock; internal Servo clock

- **Gate2[*i*].PhaseServoDir** = 1: External Phase clock; internal Servo clock
- **Gate2[*i*].PhaseServoDir** = 2: Internal Phase clock; external Servo clock
- **Gate2[*i*].PhaseServoDir** = 3: External Phase clock; external Servo clock

In all normal use, **Gate2[*i*].PhaseServoDir** is either set to 0 (on at most one IC) or 3 (on all the other ICs). Generally, on re-initialization, the Power PMAC automatically selects which IC it will use as the source of its system Phase and Servo clock signals, setting these variable values. Most users will never change these settings.

Gate2[*i*].PhaseServoDir constitutes bits 12 – 13 of the full-word element **Gate2[*i*].ClockCtrl** (bits 20 – 21 of the 32-bit element in C).

Gate2[*i*].PwmCtrl

Description: MACRO IC full-word PWM control element

Range: $0 .. $FFFFFF

Units: Bit field

Default: $197F0F

Gate2[*i*].PwmCtrl is the full-word element that comprises the multi-channel setup for generating PWM signals for the IC. It is comprised of the following partial-word elements:

Partial-Word Element	Script Bits	C Bits	Functionality
PwmPeriod	23 – 08	31 – 16	MaxPhase/PWM period control
PwmDeadTime	07 – 00	15 – 08	PWM deadtime/PFM pulse width control
(*null*)	..	07 – 00	(*No hardware present*)

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word, even though there are only 24 bits in the physical register. These 24 bits are found at the high end of the 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate2[*i*].PwmDeadTime

Description: MACRO IC PWM deadtime/PFM pulse-width control

Range: 0 .. 255

Units: 16*PWMCLK cycles, PFMCLK cycles
PWM deadtime = [16 / PWMCLK freq. (MHz)] * **Gate2[*i*].PwmDeadTime**
= 0.135 µsec * **Gate2[*i*].PwmDeadTime**
PFM pulse width = [1 / PFMCLK freq. (MHz) * **Gate2[*i*].PwmDeadTime**
= PFMCLK period (µsec) * **Gate2[*i*].PwmDeadTime**

Default: 15
PWM deadtime = 0.135 µsec * 15 = 2.03 µsec
PFM pulse width = [1 / 9.8304 MHz} * 15 = 1.526 µsec
(with default **Gate2[*i*].HardwareClockCtrl**)

Legacy I-variable alias: I6*m*04, I6*m*54

Gate2[*i*].PwmDeadTime controls the deadtime period between top and bottom on-times in the automatic PWM generation for machine interface channels on PMAC2-style "DSPGATE2" MACRO IC. In conjunction with **Gate2[*i*].HardwareClockCtrl**, it also controls the pulse width for the IC's automatic pulse-frequency modulation (PFM) generation for the machine interface channels on the MACRO IC.

The PWM deadtime, which is the delay between the top signal turning off and the bottom signal turning on, and vice versa, is specified in units of 16 PWM_CLK cycles. This means that the deadtime can be specified in increments of 0.135 µsec. The equation for **Gate2[*i*].PwmDeadTime** as a function of PWM deadtime is:

$$Gate2[i].PwmDeadTime = \frac{DeadTime(\mu sec)}{0.135\,\mu sec}$$

Note

Most direct-PWM drives enforce a minimum deadtime period as a safety feature. However, they do so with a lower-frequency clock that is asynchronous to this IC's PWM_CLK signal. This means that relying on the drive's minimum deadtime setting (which occurs when this element specifies a smaller deadtime) decreases the effective command resolution and even can introduce a beat frequency into the output.

The PFM pulse width is specified in PFM_CLK cycles, as defined by **Gate2[*i*].PwmDeadTime**. The equation for **Gate2[*i*].PwmDeadTime** as a function of PFM pulse width and PFM_CLK frequency is:

$$Gate2[i].PwmDeadTime = PfmClkFreq(MHz) * PfmPulseWidth\ (\mu sec) - 1$$

In the DSPGATE2's PFM pulse generation, the minimum off time between pulses is equal to the pulse width. This means that the maximum PFM output frequency is

$$PfmMaxFreq(MHz) = \frac{PfmClkFreq\,(MHz)}{2 * \left[Gate2[i].PwmDeadTime + 1\right]}$$

Gate2[*i*].PwmDeadTime constitutes bits 0 – 7 of the full-word element **Gate2[*i*].PwmCtrl** (bits 8 – 15 of the 32-bit element in C).

Examples

A PWM deadtime of approximately 1 microsecond is desired:

Gate2[*i*].PwmDeadTime ≅ 1 μsec / 0.135 μsec ≅ 7

With a 2.4576 MHz PFM_CLK frequency, a pulse width of 0.4 μsec is desired:

Gate2[*i*].PwmDeadTime ≅ 2.4576 MHz * 0.4 μsec ≅ 1

Gate2[*i*].PwmPeriod

Description: MACRO IC MaxPhase/PWM Frequency Control

Range: 0 .. 32767

Units: MaxPhase Freq. = 117,964.8 kHz / [2***Gate2[*i*].PwmPeriod**+3]
PWM Freq. = 117,964.8 kHz / [4* **Gate2[*i*].PwmPeriod**+6]

Default: 6527
MaxPhase Freq. = 117,964.8 / 13057 = 9.0346 kHz
PWM Freq. = 117,964.8 / 26114 = 4.5173 kHz

Legacy I-variable alias: I6*m*00, I6*m*50

Gate2[*i*].PwmPeriod controls the internal "MaxPhase" clock frequency, and the PWM frequency for the two machine interface channels, on a PMAC2-style "DSPGATE2" MACRO IC. The internally generated Phase and Servo clocks on a MACRO IC m are derived from the MaxPhase clock.

If the MACRO IC is used to generate the Phase and Servo clocks for the PMAC system (as set by the **Gate2[*i*].PhaseServoDir** variables), this variable is part of the control for the frequency of these system clocks.

Gate2[*i*].PwmPeriod controls these frequencies by setting the limits of the PWM up-down counter, which increments and decrements at the PWMCLK frequency of 117,964.8 kHz (117.9648 MHz).

The actual Phase clock frequency is divided down from the maximum phase clock according to the setting of **Gate2[*i*].PhaseClockDiv**. On the falling edge of the phase clock, Power PMAC samples any serial analog-to-digital converters connected to its MACRO ICs (as for phase current measurement), and interrupts the processor to start any necessary phase commutation and digital current-loop algorithms. Even if phasing and current-loop algorithms are not used, the MaxPhase and Phase Clock frequencies are important because the servo clock is derived from the phase clock.

The PWM frequency determines the actual switching frequency of amplifiers connected to any of four machine interface channels with the direct PWM command. It is only important if the direct PWM command signal format is used.

The maximum value that can be written into the PWM command register without full saturation is **Gate2[*i*].PwmPeriod**+1 on the positive end, and – **Gate2[*i*].PwmPeriod**–2 on the negative end. Generally, the "PWM scale factor" **Motor[*i*].PwmSf** for each motor, which determines the maximum PWM command magnitude, is set to **Gate2[*i*].PwmPeriod** + 10%.

Usually, **Gate2[*i*].PwmPeriod** for MACRO ICs that are not controlling the system Phase clock frequency are set to the same value as the one that is. If a different PWM frequency is desired for the PWM outputs on other MACRO ICs, **Gate2[*i*].PwmPeriod** should be set so that:

$$\frac{2 * PwmFreq(kHz)}{PhaseFreq} = \{Integer\}$$

This will keep the PWM hardware on these channels in synchronization with the software algorithms driven by the system Phase clock. For example, if the phase frequency is 10 kHz, the PWM frequency for other MACRO ICs can be 5, 10, 15, 20, (etc.) kHz.

To set **Gate2[*i*].PwmPeriod** for a desired PWM frequency, the following formula can be used:

$$Gate2[i].PwmPeriod = \frac{117{,}964.8(kHz)}{4 * PwmFreq(kHz)} - 1 \text{ (rounded down)}$$

To set **Gate2[*i*].PwmPeriod** for a desired "maximum phase" clock frequency, the following formula can be used:

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

Gate2[*i*].PwmPeriod constitutes bits 8 – 23 of the full-word element **Gate2[*i*].PwmCtrl** (bits 16 – 31 of the 32-bit element in C).

Example

To set a PWM frequency of 10 kHz and therefore a MaxPhase clock frequency of 20 kHz:

Gate2[*i*].PwmPeriod = (117,964.8 kHz / [4*10 kHz]) - 1 = 2948

To set a PWM frequency of 7.5 kHz and therefore a MaxPhase clock frequency of 15 kHz:

Gate2[*i*].PwmPeriod = (117,964.8 kHz / [4*7.5 kHz]) - 1 = 3931

Gate2[*i*].ServoClockDiv

Description: MACRO IC servo-clock frequency divider control

Range: 0 .. 15

Units: Servo Clock Freq. = Phase Clock Freq. / (**Gate2[*i*].ServoClockDiv** +1)

Default: 3
Servo Clock Freq. = 9.0346 kHz / (3+1) = 2.2587 kHz
(with default values of **Gate2[*i*].PwmPeriod** and **Gate2[*i*].PhaseClockDiv**)

Legacy I-variable alias: I6*m*02, I6*m*52

Gate2[*i*].ServoClockDiv, in conjunction with **Gate2[*i*].PhaseClockDiv** and **Gate2[*i*].PwmPeriod**, determines the frequency of the Servo clock generated inside each PMAC2-style "DSPGATE2" MACRO IC. However, only the MACRO IC told to use and output its own Servo clock with **Gate2[*i*].PhaseServoDir** uses the Servo clock signal it generates. Each cycle of the Servo clock, Power PMAC updates the commanded position for each activated motor, and executes the servo algorithm to compute the command to the amplifier or the commutation algorithm.

Specifically, **Gate2[*i*].ServoClockDiv** controls how many times the Servo clock frequency is divided down from the Phase clock, whose frequency is set by **Gate2[*i*].PhaseClockDiv** and **Gate2[*i*].PwmPeriod**. The Servo clock frequency is equal to the Phase clock frequency divided by (**Gate2[*i*].ServoClockDiv** 1). **Gate2[*i*].ServoClockDiv** has a range of 0 to 15, so the frequency division can be by a factor of 1 to 16. The equation for **Gate2[*i*].ServoClockDiv** is:

$$Gate2[i].ServoClockDiv = \frac{PhaseFreq(kHz)}{ServoFreq(kHz)} - 1$$

The ratio of Phase Clock frequency to Servo Clock frequency must be an integer.

For execution of trajectories at the proper speed, **Sys.ServoPeriod** must be set properly to tell the trajectory generation software what the Servo clock cycle time is. The formula for **Sys.ServoPeriod** is:

$$Sys.ServoPeriod = \frac{1}{ServoFreq(kHz)}$$

In terms of the variables that determine the Servo clock frequency, the formula for **Sys.ServoPeriod** is:

$$Sys.ServoPeriod = \frac{(2 * PwmPeriod + 3)(PhaseClockDiv + 1)(ServoClockDiv + 1)}{117{,}964.8}$$

At the default servo clock frequency, **Sys.ServoPeriod** should be set to 0.442718 in order that Power PMAC's interpolation routines use the proper servo update time.

Note: If the servo clock frequency is set too high, lower priority tasks such as communications can be starved for time. If the background tasks are completely starved, the watchdog timer will trip, shutting down the board. If a normal reset of the board does not re-establish a state where the watchdog timer has not tripped and communications works well, it will be necessary to re-initialize the board by powering up with a USB memory stick installed. This restores default

settings, so communication is possible, and **Gate2[*i*].PwmPeriod**, **Gate2[*i*].PhaseClockDiv**, and **Gate2[*i*].ServoClockDiv** can be set to supportable values.

Gate2[*i*].ServoClockDiv constitutes bits 20 – 23 of the full-word element **Gate2[*i*].ClockCtrl** (bits 28 – 31 of the 32-bit element in C).

Example

With a 6.67 kHz Phase Clock frequency established by **Gate2[*i*].PwmPeriod** and **Gate2[*i*].PhaseClockDiv**, and a desired 3.33 kHz Servo Clock frequency:

Gate2[*i*].PhaseClockDiv = (6.67 / 3.33) - 1 = 2 - 1 = 1

Gate2[*i*]. Channel-Specific Setup Elements

This section describes the channel-specific saved data structure elements for PMAC2-style "DSPGATE2" MACRO ICs. Each of the 2 channels on the IC can be set up independently with regard to these variables.

In these setup elements, the channel index values **Chan[*j*]** for Power PMAC are one less than the channel number "*n*" in Turbo PMAC software and in the hardware documentation. Power PMAC channel index values "*j*" range from 0 to 1, and correspond to Turbo PMAC channel numbers "*n*" and hardware channel numbers 1 to 2, respectively.

Gate2[*i*].Chan[*j*].CaptCtrl

Description: MACRO IC channel position-capture control

Range: 0 .. 15

Units: Bit field

Default: 1

Legacy I-variable alias: I68*n*2

Gate2[*i*].Chan[*j*].CaptCtrl determines which input signal or combination of signals for this channel of a PMAC2-style MACRO IC, and which polarity, triggers a hardware position capture of the counter for Encoder n. If a flag input (home, limit, or user) is used, **Gate2[*i*].Chan[*j*].CaptFlagSel** determines which flag. Proper setup of this variable is essential for a successful homing search move or other move-until-trigger for the motor using this channel for its position-loop feedback and flags if the super-accurate hardware position capture function is used. If **Motor[*x*].CaptureMode** is set to 0, 1, or 3 to select hardware trigger (with or without hardware position capture), this variable must be set up properly.

The following settings of **Gate2[*i*].Chan[*j*].CaptCtrl** may be used:

- **Gate2[*i*].Chan[*j*].CaptCtrl** = 0: Continuous or Hall capture
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 1: Capture on Index (CHCn) high
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 2: Capture on Flag n high
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 3: Capture on (Index high AND Flag n high)

- **Gate2[*i*].Chan[*j*].CaptCtrl** = 4: Continuous or Hall capture
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 5: Capture on Index (CHCn) low
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 6: Capture on Flag n high
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 7: Capture on (Index low AND Flag n high)
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 8: Continuous or Hall capture
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 9: Capture on Index (CHCn) high
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 10: Capture on Flag n low
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 11: Capture on (Index high AND Flag n low)
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 12: Continuous or Hall capture
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 13: Capture on Index (CHCn) low
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 14: Capture on Flag n low
- **Gate2[*i*].Chan[*j*].CaptCtrl** = 15: Capture on (Index low AND Flag n low)

Note that only flags and index inputs of the same channel number as the encoder may be used for hardware capture of that encoder's position. This means that to use the hardware capture feature for the homing search move, **Motor[*x*].pEncCtrl** must use flags of the same channel number as the encoder that **Motor[*x*].pEnc** uses for position-loop feedback.

If bits 0 and 1 of this variable are both set to 0 (value 0, 4, 8, or 12) and **Gate2[*i*].Chan[*j*].GatedIndexSel** for the same channel is 0, the capture trigger will occur immediately if the trigger is armed. If **Gate2[*i*].Chan[*j*].GatedIndexSel** for the same channel is 1, the trigger will occur on the next transition of any of the U, V, or W "Hall" input flags.

The trigger is armed when the position-capture register **Gate2[*i*].Chan[*j*].HomeCapt** is read. This sets the status element **Gate2[*i*].Chan[*j*].PosCapt** to 0, indicating that the trigger is armed, but the next trigger has not occurred. In Power PMAC's move-until-trigger functions, this read is performed automatically at the beginning of the move so the trigger is always armed. After this, as soon as the MACRO IC hardware sees that the specified input lines are in the specified states, the trigger will occur – it is level-triggered, not edge-triggered. No transition is required.

Gate2[*i*].Chan[*j*].CaptCtrl constitutes bits 4 – 7 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bits 12 – 15 of the 32-bit element in C).

Gate2[*i*].Chan[*j*].CaptFlagSel

Description: MACRO IC channel position-capture flag select

Range: 0 .. 3

Units: Enumeration

Default: 0

Legacy I-variable alias: I68*n*3

Gate2[*i*].Chan[*j*].CaptFlagSel determines which of the four "flag" inputs for the channel will be used for hardware position capture (if one is used) of the channel's encoder counter on a PMAC2-style MACRO IC. **Gate2[*i*].Chan[*j*].CaptCtrl** determines whether a flag is used and which

polarity of the flag will cause the trigger. The possible values of **Gate2[*i*].Chan[*j*].CaptFlagSel** and the flag each selects is:

- **Gate2[*i*].Chan[*j*].CaptFlagSel** = 0: HOMEn (Home Flag n)
- **Gate2[*i*].Chan[*j*].CaptFlagSel** = 1: PLIMn (Positive End Limit Flag n)
- **Gate2[*i*].Chan[*j*].CaptFlagSel** = 2: MLIMn (Negative End Limit Flag n)
- **Gate2[*i*].Chan[*j*].CaptFlagSel** = 3: USERn (User Flag n)

Gate2[*i*].Chan[*j*].CaptFlagSel is typically set to 0 for homing search moves in order to use the home flag for the channel. It is commonly set to 3 afterwards to select the User flag if other uses of the hardware position capture function are desired, such as for probing and registration. If you wish to capture on the PLIMn or MLIMn overtravel limit flags, you probably will want to disable their normal shutdown functions.

Gate2[*i*].Chan[*j*].CaptFlagSel constitutes bits 8 – 9 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bits 16 – 17 of the 32-bit element in C).

Gate2[*i*].Chan[*j*].Ctrl

Description: MACRO IC channel full-word control element

Range: $0 .. $FFFFFF

Units: Bit field

Default: $C00017

Gate2[*i*].Chan[*j*].Ctrl is the full-word element that comprises the setup elements for the signal interfaces for the channel of the IC. It is comprised of the following partial-word elements:

Partial-Word Element	Script Bits	C Bits	Functionality
OutputMode	23 – 22	31 – 30	Output mode select
OutputPol	21 – 20	29 – 28	Output polarity control
PfmDirPol	19	27	PFM direction output polarity control
OneOverTEna	18	26	Hardware 1/T enable
IndexGateState	17 – 16	25 – 24	Gated index capture state control
GatedIndexSel	15	23	Gated index capture select
AmpEna	14	22	Amplifier enable output state
Equ1Ena	13	21	Compare circuit encoder select
EquWrite	12 – 11	20 – 19	Compare output force state
PosClear	10	18	Encoder counter reset control
CaptFlagSel	09 – 08	17 – 16	Position-capture flag select
CaptCtrl	07 – 04	15 – 12	Position-capture control
EncCtrl	03 – 00	11 – 08	Encoder decode control
(*null*)	..	07 – 00	(*No hardware present*)

Those elements shown in bold are saved setup elements and are documented individually in this chapter. Those not in bold are not saved, and are documented in the chapter on Non-Saved Setup Elements.

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word, even though there are only 24 bits in the physical register. These 24 bits are found at the high end of the 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate2[*i*].Chan[*j*].EncCtrl

Description: MACRO IC channel encoder decode control

Range: 0 .. 15

Units: none

Default: 7

Legacy I-variable alias: I68*n*0

Gate2[*i*].Chan[*j*].EncCtrl controls how the encoder input signal for the channel on the PMAC2-style MACRO IC is decoded into counts. As such, this defines the sign and magnitude of a "count". The following settings may be used to decode an input signal.

- **Gate2[*i*].Chan[*j*].EncCtrl** = 0: Pulse and direction CW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 1: x1 quadrature decode CW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 2: x2 quadrature decode CW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 3: x4 quadrature decode CW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 4: Pulse and direction CCW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 5: x1 quadrature decode CCW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 6: x2 quadrature decode CCW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 7: x4 quadrature decode CCW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 8: Internal pulse and direction
- **Gate2[*i*].Chan[*j*].EncCtrl** = 9: Not used
- **Gate2[*i*].Chan[*j*].EncCtrl** = 10: Not used
- **Gate2[*i*].Chan[*j*].EncCtrl** = 11: x6 hall-format decode CW
- **Gate2[*i*].Chan[*j*].EncCtrl** = 12: MLDT pulse timer control (internal pulse resets timer; external pulse latches timer)
- **Gate2[*i*].Chan[*j*].EncCtrl** = 13: Not used
- **Gate2[*i*].Chan[*j*].EncCtrl** = 14: Not used
- **Gate2[*i*].Chan[*j*].EncCtrl** = 15: x6 hall-format decode CCW

In any of the quadrature-decode modes, the MACRO IC is expecting two input waveforms on CHAn and CHBn, each with approximately 50% duty cycle, and approximately one-quarter of a cycle out of phase with each other. "Times-one" (x1) decode provides one count per cycle; x2 provides two counts per cycle; and x4 provides four counts per cycle. The vast majority of users select x4 decode to get maximum resolution.

The "clockwise" (CW) and "counterclockwise" (CCW) options simply control which direction counts up. If you get the wrong direction sense, simply change to the other option (e.g. from 7 to 3 or vice versa).

WARNING

Changing the direction sense of the decode for the feedback encoder of a motor that is operating properly will result in unstable positive feedback and a dangerous runaway condition in the absence of other changes. The output polarity must be changed as well to re-establish polarity match for stable negative feedback.

In the pulse-and-direction decode modes, the MACRO IC is expecting the pulse train on CHAn, and the direction (sign) signal on CHBn. If the signal is unidirectional, the CHBn line can be allowed to pull up to a high state, or it can be hardwired to a high or low state.

If **Gate2[*i*].Chan[*j*].EncCtrl** is set to 8, the decoder inputs the pulse and direction signal generated by the channel's own pulse frequency modulator (PFM) output circuitry. This permits the PMAC2 to create a phantom closed loop when driving an open-loop stepper system. *No jumpers or cables are needed to do this; the connection is entirely within the MACRO IC.* The counter polarity automatically matches the PFM output polarity.

If **Gate2[*i*].Chan[*j*].EncCtrl** is set to 11 or 15, the channel is expecting three Hall-sensor format inputs on CHAn, CHBn, and CHCn, each with approximately 50% duty cycle, and approximately one-third (120°e) of a cycle out of phase with each other. The decode circuitry will generate one count on each edge of each signal, yielding 6 counts per signal cycle ("x6 decode"). The difference between 11 and 15 is which direction of signal causes the counter to count up.

If **Gate2[*i*].Chan[*j*].EncCtrl** is set to 12, the timer circuitry is set up to read magnetostrictive linear displacement transducers (MLDTs) such as Temposonics™. In this mode, the timer is cleared when the PFM circuitry sends out the excitation pulse to the sensor on PULSEn, and it is latched into the memory-mapped register when the excitation pulse is received on CHAn.

Gate2[*i*].Chan[*j*].EncCtrl constitutes bits 0 – 3 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bits 8 – 11 of the 32-bit element in C).

Gate2[*i*].Chan[*j*].Equ1Ena

Description: MACRO IC channel compare circuit encoder select

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: I68*n*1

Gate2[*i*].Chan[*j*].Equ1Ena controls which channel's encoder counter is tied to the position compare circuitry for the channel on a PMAC2-style MACRO IC. It has the following possible settings:

- **Gate2[*i*].Chan[*j*].Equ1Ena** = 0: Use channel's own encoder counter for position compare function
- **Gate2[*i*].Chan[*j*].Equ1Ena** = 1: Use IC's first-channel encoder counter for position compare function

When **Gate2[*i*].Chan[*j*].Equ1Ena** is set to 0, the channel's position compare registers are tied to the channel's own encoder counter, and the position compare signal appears only on the EQU output for that channel.

When **Gate2[*i*].Chan[*j*].Equ1Ena** is set to 1, the channel's position compare register is tied to the first encoder counter on the MACRO IC, and the position compare signal appears both on this channel's own EQU output, and combined into the EQU output for Channel 1 on the MACRO IC; executed as a logical OR.

Gate2[*i*].Chan[0].Equ1Ena performs no effective function, so is always 1. It cannot be set to 0.

Gate2[*i*].Chan[*j*].Equ1Ena constitutes bit 13 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bit 21 of the 32-bit element in C).

Gate2[*i*].Chan[*j*].GatedIndexSel

Description: MACRO IC channel gated-index capture select

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: I68*n*4

Gate2[*i*].Chan[*j*].GatedIndexSel controls whether the raw encoder index channel input or a version of the input gated by the AB-quadrature state is used for position capture of the channel's encoder on a PMAC2-style MACRO IC. It has the following possible settings:

- **Gate2[*i*].Chan[*j*].GatedIndexSel** = 0: Use ungated index for encoder position capture
- **Gate2[*i*].Chan[*j*].GatedIndexSel** = 1: Use index gated by quadrature channels for position capture

When **Gate2[*i*].Chan[*j*].GatedIndexSel** is set to 0, the encoder index channel input (CHCn) is passed directly into the position capture circuitry.

When **Gate2[*i*].Chan[*j*].GatedIndexSel** is set to 1, the encoder index channel input (CHCn) is logically combined with ("gated by") the quadrature signals of Encoder n before going to the position capture circuitry. The intent is to get a "gated index" signal exactly one quadrature state wide. This provides a more accurate and repeatable capture, and makes the use of the capture function to confirm the proper number of counts per revolution very straightforward.

In order for the gated index capture to work reliably, the index pulse must reliably span one, but only one, "high-high" or "low-low" AB quadrature state of the encoder.
Gate2[*i*].Chan[*j*].IndexGateState allows you to select which of these two possibilities is used.

Note: If **Gate2[*i*].Chan[*j*].GatedIndexSel** is set to 1, but **Gate2[*i*].Chan[*j*].CaptCtrl** bit 0 and bit 1 are set to 0, so the index is not used in the position capture, then the encoder position is captured on the first edge of any of the U, V, or W flag inputs for the channel. In this case, bits 0, 1, and 2 of the channel status word tell what hall-state edge caused the capture.

Gate2[*i*].Chan[*j*].GatedIndexSel constitutes bit 15 of the full-word element
Gate2[*i*].Chan[*j*].Ctrl (bit 23 of the 32-bit element in C).

Gate2[*i*].Chan[*j*].IndexGateState

Description: MACRO IC channel gated-index capture state control

Range: 0 .. 3

Units: Bit field

Default: 0

Legacy I-variable alias: I68*n*5

Gate2[*i*].Chan[*j*].IndexGateState is a 2-bit variable that controls two functions for the index signal of the channel's encoder.

When using the "gated index" feature of a PMAC2-style MACRO IC for more accurate position capture (**Gate2[*i*].Chan[*j*].GatedIndexSel** = 1), bit 0 of **Gate2[*i*].Chan[*j*].IndexGateState** specifies whether the raw index-channel signal fed into the encoder input of the MACRO IC is passed through to the position capture signal only on the "high-high" quadrature state (bit 0 = 0), or only on the "low-low" quadrature state (bit 0 = 1).

Bit 1 of **Gate2[*i*].Chan[*j*].IndexGateState** controls whether the MACRO IC "de-multiplexes" the index pulse and the 3 hall-style commutation states from the third channel based on the quadrature state, as with Yaskawa incremental encoders. If bit 1 is set to 0, this de-multiplexing function is not performed, and the signal on the "C" channel of the encoder is used as the index only. If bit 1 is set to 1, the MACRO IC breaks out the third-channel signal into four separate values, one for each of the four possible AB-quadrature states. The de-multiplexed hall commutation states can be used to provide power-on phase position.

Note: Immediately after power-up, the Yaskawa encoder automatically cycles its AB outputs forward and back through a full quadrature cycle to ensure that all of the hall commutation states are available to the controller before any movement is started. However, if the encoder is

powered up at the same time as the Turbo PMAC, this will happen before the MACRO IC is ready to accept these signals. Bit 2 of the channel's status word, "Invalid De-multiplex", will be set to 1 if the MACRO IC has not seen all of these states when it was ready for them. To use this feature, it is recommended that the power to the encoder be provided through a software-controlled relay to ensure that valid readings of all states have been read before using these signals for power-on phasing.

Gate2[*i*].Chan[*j*].IndexGateState has the following possible settings:

- **Gate2[*i*].Chan[*j*].IndexGateState** = 0: Gate index with "high-high" quadrature state (GI = A & B & C), no demux
- **Gate2[*i*].Chan[*j*].IndexGateState** = 1: Gate index with "low-low" quadrature state (GI = A/ & B/ & C), no demux
- **Gate2[*i*].Chan[*j*].IndexGateState** = 2 or 3: De-multiplex hall and index from third channel, gating irrelevant

Gate2[*i*].Chan[*j*].IndexGateState constitutes bits 16 – 17 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bits 24 – 25 of the 32-bit element in C).

Gate2[*i*].Chan[*j*].OutputMode

Description: MACRO IC channel output mode select

Range: 0 .. 3

Units: Bit field

Default: 0

Legacy I-variable alias: I68*n*6

Gate2[*i*].Chan[*j*].OutputMode controls what output formats are used on the command output signal lines for the channel of a PMAC2-style MACRO IC. It has the following possible settings:

- **Gate2[*i*].Chan[*j*].OutputMode** = 0: Outputs A & B are PWM; Output C is PWM
- **Gate2[*i*].Chan[*j*].OutputMode** = 1: Outputs A & B are DAC; Output C is PWM
- **Gate2[*i*].Chan[*j*].OutputMode** = 2: Outputs A & B are PWM; Output C is PFM
- **Gate2[*i*].Chan[*j*].OutputMode** = 3: Outputs A & B are DAC; Output C is PFM

If a three-phase direct PWM command format is desired, **Gate2[*i*].Chan[*j*].OutputMode** should be set to 0. If signal outputs for (external) digital-to-analog converters are desired, **Gate2[*i*].Chan[*j*].OutputMode** should be set to 1 or 3. In this case, the C output can be used as a supplemental (non-servo) output in either PWM or PFM form. For example, it can be used to excite an MLDT sensor (e.g. Temposonics™) in PFM form.

Gate2[*i*].Chan[*j*].OutputMode constitutes bits 22 – 23 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bits 30 – 31 of the 32-bit element in C).

Gate2[*i*].Chan[*j*].OutputPol

Description: MACRO IC channel output polarity control

Range: 0 .. 3

Units: Bit field

Default: 0

Legacy I-variable alias: I68*n*7

Gate2[*i*].Chan[*j*].OutputPol controls the high/low polarity of the command output signals for the channel on a PMAC2-style MACRO IC. It has the following possible settings:

- **Gate2[*i*].Chan[*j*].OutputPol** = 0: Do not invert Outputs A & B; Do not invert Output C
- **Gate2[*i*].Chan[*j*].OutputPol** = 1: Invert Outputs A & B; Do not invert Output C
- **Gate2[*i*].Chan[*j*].OutputPol** = 2: Do not invert Outputs A & B; Invert Output C
- **Gate2[*i*].Chan[*j*].OutputPol** = 3: Invert Outputs A & B; Invert Output C

The default non-inverted outputs are high true. For PWM signals on Outputs A, B, and C, this means that the transistor-on signal is high. Delta Tau PWM-input amplifiers, and most other PWM-input amplifiers, expect this non-inverted output format. For such a 3-phase motor drive, **Gate2[*i*].Chan[*j*].OutputPol** should be set to 0.

Caution

If the high/low polarity of the PWM signals is wrong for a particular amplifier, what was intended to be deadtime between top and bottom on-states as set by **Gate2[*i*].PwmDeadTime** becomes overlap. If the amplifier-input circuitry does not lock this out properly, this causes an effective momentary short circuit between bus power and ground. This would destroy the power transistors very quickly.

For PFM signals on Output C, non-inverted means that the pulse-on signal is high (direction polarity is controlled by **Gate2[*i*].Chan[*j*].PfmDirPol**). During a change of direction, the direction bit will change synchronously with the leading edge of the pulse, which in the non-inverted form is the rising edge. If the drive requires a set-up time on the direction line before the rising edge of the pulse, the pulse output can be inverted so that the rising edge is the trailing edge, and the pulse width (established by **Gate2[*i*].PwmDeadtime**) is the set-up time.

For DAC signals on Outputs A and B, non-inverted means that a 1 value to the DAC is high. DACs used on Delta Tau accessory boards, as well as all other known DACs always expect non-inverted inputs, so **Gate2[*i*].Chan[*j*].OutputPol** should always be set to 0 or 2 when using DACs on this channel.

WARNING

Changing the high/low polarity of the digital data to the DACs has the effect of inverting the voltage sense of the DACs' analog outputs. This changes the polarity match between output and feedback. If the feedback loop had been stable with negative feedback, this change would create destabilizing positive feedback, resulting in a dangerous runaway condition.

Gate2[*i*].Chan[*j*].OutputPol constitutes bits 20 – 21 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bits 28 – 29 of the 32-bit element in C).

Gate2[*i*].Chan[*j*].PfmDirPol

Description: MACRO IC channel PFM direction output polarity control

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: I68*n*8

Gate2[*i*].Chan[*j*].PfmDirPol controls the polarity of the direction output signal in the pulse-and-direction format for the channel of the PMAC2-style MACRO IC. It is only active if **Gate2[*i*].Chan[*j*].OutputMode** has been set to 2 or 3 to use Output C as a pulse-frequency-modulated (PFM) output. It has the following possible settings:

- **Gate2[*i*].Chan[*j*].PfmDirPol** = 0: Do not invert direction signal (+ = low; - = high)
- **Gate2[*i*].Chan[*j*].PfmDirPol** = 1: Invert direction signal (- = low; + = high)

If **Gate2[*i*].Chan[*j*].PfmDirPol** is set to the default value of 0, a positive direction command provides a low output; if **Gate2[*i*].Chan[*j*].PfmDirPol** is set to 1, a positive direction command provides a high output.

Gate2[*i*].Chan[*j*].PfmDirPol constitutes bit 19 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bit 27 of the 32-bit element in C).

Gate3[*i*]. (PMAC3-Style IC) Saved Data Structure Elements

The **Gate3[*i*].** data structure comprises memory-mapped registers in the "DSPGATE3" machine-interface ASIC designed by Delta Tau. This IC provides many state-of-the-art features for processing input and output signals for servo and general-purpose I/O use.

In the Power PMAC script environment, it is also possible to use the name of the device on which the IC is present as the structure name instead of "**Gate3[*i*]**". The names of the accessories for which this is presently supported are:

- **Acc24E3[*i*].** ACC-24E3 UMAC axis-interface board
- **Acc5E3[*i*].** ACC-5E3 UMAC MACRO interface board
- **Acc5EP3[*i*].** ACC-5EP3 Etherlite MACRO interface board
- **Acc59E3[*i*].** ACC-59E3 UMAC A/D & D/A-converter board
- **Clipper[*i*].** Power Clipper controller board
- **PowerBrick[*i*].** Power Brick controller board

These names are simply "aliases" for the **Gate3[*i*]** name. The index number ***i*** is the same whether the **Gate3[*i*]** name or the alias name is used. Each DSPGATE3 IC used must have a unique index number. C programs must use the **Gate3[*i*]** name; the aliases are not available in the C environment.

The index value ***i*** for an IC can range from 0 to 15. It is determined by the hardware address DIP switch setting for the board.

Note: In order to prevent potentially dangerous changes to setup elements in the **Gate3[*i*].** data structure by unauthorized users, many of the setup elements are "write protected" and cannot be changed unless the "write-protect key" in the IC has been set properly before writing to the protected setup element.

The write-protect key register can be accessed through the element **Gate3[*i*].WpKey**, and the value that must be in this register to enable a write operation to a protected setup element is $AAAAAAAA. The IC automatically clears the key value to 0 after writing to the protected element, so the key must be set once for each write-operation to a protected element.

In a C program, this must be done explicitly. That is, each operation to write a value to a protected setup element must be preceded by an operation to write $AAAAAAAA to **Gate3[*i*].WpKey**.

In the script environment – either buffered program statements or on-line commands – there is a second method that can make the process easier. Before any script operation that writes to a protected setup element, Power PMAC will copy the value in the global memory element **Sys.WpKey** into the actual write-protect key register **Gate3[*i*].WpKey**. This permits the user to write the key value of $AAAAAAAA just once into the global element, and Power PMAC will copy it automatically into the IC register every time it is needed.

Sys.WpKey is *not* a saved value, so it must be set after power-up/reset every time you wish to change protected setup elements in the IC. The special windows in the Integrated Development Environment (IDE) program for setting up the IC automatically manage the value in **Sys.WpKey**.

Of course, if you wish to prevent changes to these setup elements in the actual application, **Sys.WpKey** should not be set to the proper key value, and unauthorized users should not be given information about the key value or mechanism.

Gate3[*i*].Multi-Channel Setup Elements

The data structure elements in this section configure signals common to all four servo channels on the IC. These include those for the general-purpose I/O and MACRO functionality.

Gate3[*i*].AdcAmpClockDiv

Description: IC "amplifier" A/D converter clock frequency control

Range: 0 .. 15

Units: none

Default: 5 (3.125 MHz)

Gate3[*i*].AdcAmpClockDiv specifies the frequency of the clock signal that controls the serial data frequency from the "amplifier" A/D converters connected to the IC's channels. These ADCs are usually used for current feedback in "direct PWM" control of motors. This element controls the frequency by specifying how many times an internal 100-MHz clock signal is divided by 2. The equation for the clock frequency is:

$$f_{AdcAmpClk} = \frac{100MHz}{2^{AdcAmpClockDiv}}$$

The following table shows the possible settings of **Gate3[*i*].AdcAmpClockDiv** and the 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

Generally, the highest clock frequency for which the ADCs are rated should be chosen, to minimize the delays in obtaining the data.

Gate3[*i*].AdcAmpClockDiv constitutes bits 28 – 31 of the full-word element **Gate3[*i*].HardwareClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].AdcAmpCtrl

Description: IC control word for amplifier ADC configuration

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $FFFFFC02
$FFFFFD00 (Power Brick amplifiers)

Gate3[*i*].AdcAmpCtrl is the full-word element that controls the interface to "amplifier ADCs" for the IC. These ADCs are typically used for current feedback from a direct PWM "power-block" amplifier. **Gate3[*i*].AdcAmpCtrl** is comprised of the following partial-word elements:

Component	Bits	Functionality
AdcAmpStrobe	31 – 08	Strobe word for amplifier ADCs
AdcAmpDelay	07 – 04	Strobe delay for amplifier ADCs
AdcAmpUtoS	03	Amplifier ADC format conversion control
AdcAmpHeaderBits	02 – 00	Amplifier ADC number of header bits

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Gate3[*i*].AdcAmpCtrl is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate3[*i*].AdcAmpDelay

Description: IC amplifier ADC strobe start delay

Range: 0 .. 15

Units: Amplifier ADC clock cycles

Default: 0

Gate3[*i*].AdcAmpDelay specifies the time between the rising edge of the phase clock signal and the start of the transmission of the amplifier ADC strobe signal, expressed in cycles of the

amplifier ADC clock. At the default value of 0, the transmission starts on the first amplifier ADC clock edge after the rising edge of phase clock.

Gate3[*i*].AdcAmpDelay permits the optimization of the timing of the sampling of the amplifier phase currents, particularly if there are delays in the transmission of the PWM outputs to the power stage or noise due to other signals at the phase clock edge. However, in most systems, satisfactory performance will be achieved at the default value of 0.

Gate3[*i*].AdcAmpDelay constitutes bits 4 – 7 of the full-word element **Gate3[*i*].AdcAmpCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].AdcAmpHeaderBits

Description: IC amplifier ADC number of header bits

Range: 0 .. 7

Units: none

Default: 2
0 (Power Brick amplifiers)

Gate3[*i*].AdcAmpHeaderBits specifies the number of "header bits" the IC is to expect at the beginning of the serial data stream from the "amplifier" ADCs for all channels on the IC. The specified number of header bits is "barrel shifted" from the most significant end of the resulting data word to the least significant end. The first bit received after the specified number of header bits is placed in the most significant bit of the register.

This parameter also permits the IC to compensate for delays in returning the ADC data caused by intermediate processing logic. For each clock-cycle delay, this parameter should be increased by 1.

This element permits the Power PMAC to interface easily to ADCs that provide one or more leading bits of information before the true converted data. Most commonly, interfaces from Delta Tau direct-PWM amplifiers utilize ADCs with 1 header bit, and have a one-clock-cycle delay.

Gate3[*i*].AdcAmpHeaderBits constitutes bits 0 – 2 of the full-word element **Gate3[*i*].AdcAmpCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].AdcAmpStrobe

Description: IC amplifier ADC strobe word

Range: $000000 .. $FFFFFF

Units: Bit field

Default: $FFFFFC
$FFFFFD (Power Brick amplifiers)

Gate3[*i*].AdcAmpStrobe specifies the strobe signal for "amplifier" ADCs connected to all channels on the IC. The24-bit word specified by this element is shifted out serially on the ADC Amp strobe lines, MSB first, one bit per ADC Amp clock cycle, starting on the rising edge of the phase clock (possibly delayed by **Gate3[*i*].AdcAmpDelay**). The value in the LSB is held until the next phase clock cycle.

The first '1' creates a rising edge on the amplifier ADC strobe output that is typically used as a "start-convert" signal. Some A/D converters just need this rising edge for the conversion; others need the signal to stay high all of the way through the conversion. The LSB of the strobe word should always be set to 0 so that a rising edge is created on the next cycle. The default value of $FFFFFC is suitable for virtually all A/D converters.

Many amplifiers will use strobe word bits after the first bit (MSB) as control and mode bits. Consult the specific amplifier manual for details.

Gate3[*i*].AdcAmpStrobe constitutes bits 8 – 31 of the full-word element **Gate3[*i*].AdcAmpCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].AdcAmpUtoS

Description: IC amplifier ADC data format conversion control

Range: 0 .. 1

Units: none

Default: 0

Gate3[*i*].AdcAmpUtoS specifies whether the "amplifer" ADC circuits on all channels of the IC perform a format conversion on the incoming data before latching into registers readable by the processor. At the default value of 0, no format conversion is performed.

If **Gate3[*i*].AdcAmpUtoS** is set to 1, the IC will convert unsigned data to signed (two's-complement) format before latching it into the readable register. Power PMAC's automatic routines that use these values, such as digital current-loop closure, expect a signed data format, but some ADCs provide data in unsigned format. This conversion permits the use of unsigned ADCs with these automatic algorithms.

Gate3[*i*].AdcAmpUtoS constitutes bit 3 of the full-word element **Gate3[*i*].AdcAmpCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].AdcEncClockDiv

Description: IC "encoder" A/D converter clock frequency control

Range: 0 .. 15

Units: none

Default: 5 (3.125 MHz)

Gate3[*i*].AdcEncClockDiv specifies the frequency of the clock signal that controls the serial data frequency from the "encoder" A/D converters connected to the IC's channels. These ADCs are usually used for position feedback from sinusoidal encoders and resolvers. This element controls the frequency by specifying how many times an internal 100-MHz clock signal is divided by 2. The equation for the clock frequency is:

$$f_{AdcEncClk} = \frac{100MHz}{2^{AdcEncClockDiv}}$$

The following table shows the possible settings of **Gate3[*i*].AdcEncClockDiv** and the 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

Generally, the highest clock frequency for which the ADCs are rated should be chosen, to minimize the delays in obtaining the data.

Gate3[*i*].AmpEncClockDiv constitutes bits 24 – 27 of the full-word element **Gate3[*i*].HardwareClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].AdcEncCtrl

Description: IC control word for encoder ADC configuration

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $FFFFFC01

Gate3[*i*].AdcEncCtrl is the full-word element that controls the interface to "encoder ADCs" for the IC. These ADCs are typically used for analog position feedback sensors such as sinusoidal encoders and resolvers. **Gate3[*i*].AdcEncCtrl** is comprised of the following partial-word elements:

Component	Bits	Functionality
AdcEncStrobe	31 – 08	Strobe word for encoder ADCs
AdcEncDelay	07 – 04	Strobe delay for encoder ADCs
AdcEncUtoS	03	Encoder ADC format conversion control
AdcEncHeaderBits	02 – 00	Encoder ADC number of header bits

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Gate3[*i*].AdcEncCtrl is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate3[*i*].AdcEncDelay

Description: IC encoder ADC strobe start delay

Range: 0 .. 15

Units: Encoder ADC clock cycles

Default: 0

Gate3[*i*].AdcEncDelay specifies the time between the rising edge of the phase clock signal and the start of the transmission of the encoder ADC strobe signal, expressed in cycles of the encoder ADC clock. At the default value of 0, the transmission starts on the first encoder ADC clock edge after the rising edge of phase clock.

Gate3[*i*].AdcEncDelay permits the optimization of the timing of the sampling of the analog position sensor values, particularly if there is noise due to other signals at the phase clock edge. However, in most systems, satisfactory performance will be achieved at the default value of 0.

Gate3[*i*].AdcEncDelay constitutes bits 4 – 7 of the full-word element **Gate3[*i*].AdcEncCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].AdcEncHeaderBits

Description: IC encoder ADC number of header bits

Range: 0 .. 7

Units: none

Default: 1

Gate3[*i*].AdcEncHeaderBits specifies the number of "header bits" the IC is to expect at the beginning of the serial data stream from the "encoder" ADCs for all channels on the IC. The specified number of header bits is "barrel shifted" from the most significant end of the resulting data word to the least significant end. The first bit received after the specified number of header bits is placed in the most significant bit of the register.

This parameter also permits the IC to compensate for delays in returning the ADC data caused by intermediate processing logic. For each clock-cycle delay, this parameter should be increased by 1.

This element permits the Power PMAC to interface easily to ADCs that provide one or more leading bits of information before the true converted data. Most commonly, interfaces from Delta Tau analog feedback devices utilize ADCs with 1 header bit and have no intermediate delays.

Gate3[*i*].AdcEncHeaderBits constitutes bits 0 – 2 of the full-word element **Gate3[*i*].AdcEncCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].AdcEncStrobe

Description: IC encoder ADC strobe word

Range: $000000 .. $FFFFFF

Units: Bit field

Default: $FFFFFC

Gate3[*i*].AdcEncStrobe specifies the strobe signal for "encoder" ADCs connected to all channels on the IC. The24-bit word specified by this element is shifted out serially on the ADC Enc strobe lines, MSB first, one bit per ADC Enc clock cycle, starting on the rising edge of the phase clock (possibly delayed by **Gate3[*i*].AdcEncDelay**). The value in the LSB is held until the next phase clock cycle.

The first '1' creates a rising edge on the amplifier ADC strobe output that is typically used as a "start-convert" signal. Some A/D converters just need this rising edge for the conversion; others need the signal to stay high all of the way through the conversion. The LSB of the strobe word should always be set to 0 so that a rising edge is created on the next cycle. The default value of $FFFFFC is suitable for virtually all A/D converters.

Delta Tau's "Auto-Correcting Interpolator" for sinusoidal encoders uses bits after the first bit (MSB) as control bits. Consult the appropriate manual for details.

Gate3[*i*].AdcEncStrobe constitutes bits 8 – 31 of the full-word element **Gate3[*i*].AdcEncCtrl**.

Gate3[*i*].AdcEncUtoS

Description: IC encoder ADC data format conversion control

Range: 0 .. 1

Units: none

Default: 0

Gate3[*i*].AdcEncUtoS specifies whether the "encoder" ADC circuits on all channels of the IC perform a format conversion on the incoming data before latching into registers readable by the processor. At the default value of 0, no format conversion is performed.

If **Gate3[*i*].AdcEncUtoS** is set to 1, the IC will convert unsigned data to signed (two's-complement) format before latching it into the readable register. Power PMAC's automatic routines that use these values, such as arctangent position angle calculations, expect a signed data format, but some ADCs provide data in unsigned format. This conversion permits the use of unsigned ADCs with these automatic algorithms.

Gate3[*i*].AdcEncUtoS constitutes bit 3 of the full-word element **Gate3[*i*].AdcEncCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].ClockPol

Description: IC DAC & ADC clock polarity control

Range: 0 .. 7

Units: Bit field

Default: 0

Gate3[*i*].ClockPol specifies the high/low polarity of the three clock signals that control the data frequency for D/A and A/D converters connected to the IC. It is a 3-bit value, with each bit controlling one clock signal. If the bit is 0, the clock signal is not inverted (it goes high at the beginning of the clock cycle). If the bit is 1, the clock signal is inverted (it goes low at the beginning of the clock cycle). The individual bits and the clock signals they control are:

- Bit 0 (Value 1): DAC clock
- Bit 1 (Value 2): Encoder ADC clock
- Bit 2 (Value 4): Amplifier ADC clock

Consult your specific hardware manual for the proper setting of this element. In general, Delta Tau products using this IC use the default value of 0 for this element

Gate3[*i*].ClockPol constitutes bits 5 – 7 of the full-word element **Gate3[*i*].HardwareClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].DacClockDiv

Description: IC D/A converter clock frequency control

Range: 0 .. 15

Units: none

Default: 5 (3.125 MHz)

Gate3[*i*].DacClockDiv specifies the frequency of the clock signal that controls the serial data frequency to the D/A converters connected to the IC's channels. These DACs are usually used to create analog velocity, torque, or phase-current signals to servo amplifiers. This element controls the frequency by specifying how many times an internal 100-MHz clock signal is divided by 2. The equation for the clock frequency is:

$$f_{DacClk} = \frac{100MHz}{2^{DacClockDiv}}$$

The following table shows the possible settings of **Gate3[*i*].DacClockDiv** and the 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

Generally, the highest clock frequency for which the DACs are rated should be chosen, to minimize the delays in transmitting the data.

Gate3[*i*].DacClockDiv constitutes bits 20 – 23 of the full-word element **Gate3[*i*].HardwareClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].DacStrobe

Description: IC DAC strobe word

Range: $00000000 .. $FFFFFFFF

Units: Bit field

Default: $FFFF0000

Gate3[*i*].DacStrobe specifies the strobe signal for DACs connected to all channels on the IC. The 32-bit word specified by this element is shifted out serially on the DAC strobe lines, MSB first, one bit per DAC clock cycle, starting on the rising edge of the phase clock. The value in the MSB is held until the next phase clock cycle.

If an *n*-bit data field is transferred to a DAC, the strobe word is typically held high for *n*-1 clock cycles. Therefore, the common settings of this variable are:

- 24-bit field (for ACC-24E3 18-bit DACs): $FFFFFF00
- 18-bit field (for Brick Controller 18-bit DACs): $FFFFC000
- 16-bit field: $FFFF0000
- 12-bit field: $FFF00000

The 16-bit DACs that come standard with the analog amplifier board on the ACC-24E3 have a 16-bit data field, so a value of $FFFF0000 is used for them. The optional 18-bit DACs for these boards have a 24-bit field, so a value of $FFFFFF00 is used for them.

If there is a difference between the number of bits in the data field and the number of actual DAC bits, as with the 18-bit DAC option on the ACC-24E3, **Motor[*x*].DacShift** must be set to the

value of this difference (i.e. to 6 for that option). If the number of bits is the same, **Motor[*x*].DacShift** can be left at its default value of 0.

Gate3[*i*].DacStrobe is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].EncClockDiv

Description: IC encoder sampling clock frequency control

Range: 0 .. 15

Units: none

Default: 5 (3.125 MHz)

Gate3[*i*].EncClockDiv specifies the frequency of the clock signal that controls the encoder input sampling in the IC's channels. Each encoder clock cycle, the "A" and "B" encoder digital inputs are sampled. The decoding logic only permits one signal edge during any clock cycle, so it cannot handle an edge rate higher than this clock frequency, which means that the maximum count frequency cannot be higher than this clock frequency. Note that the absolute maximum signal frequency is one fourth of this clock frequency. It is generally recommended that a 20% safety margin for signal imperfections be provided.

Presently, the ASIC itself can accept higher signal frequencies than the encoder receiver circuits on Power PMAC hardware. These receivers are limited to an absolute maximum 10 MHz signal frequency (40 MHz count frequency). With safety margin, no signal frequency greater than 8 MHz should be provided. These receivers are used both directly for digital quadrature encoders, and as comparators for analog sinusoidal encoders.

This clock signal also controls the signal propagation through the digital delay filters for the encoder and flag inputs, with any input state present for only a single clock cycle filtered out. The lower the clock frequency, the longer a noise pulse can be filtered out. (There is the option to have the flag inputs further filtered by a second digital delay filter whose propagation is controlled by a clock frequency determined by **Gate3[*i*].FiltClockDiv**.)

This element controls the frequency by specifying how many times an internal 100-MHz clock signal is divided by 2. The equation for the clock frequency is:

$$f_{EncClk} = \frac{100MHz}{2^{EncClockDiv}}$$

The following table shows the possible settings of **Gate3[*i*].EncClockDiv** and the 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

Generally, the lowest clock frequency that permits the maximum desired count rate should be chosen.

Gate3[*i*].EncClockDiv constitutes bits 12 – 15 of the full-word element **Gate3[*i*].HardwareClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].EncLatchDelay

Description: IC analog encoder latch/strobe delay

Range: 0 .. 255

Units: 16 x encoder ADC clock cycles

Default: 0

Gate3[*i*].EncLatchDelay specifies the delay from the rising edge of the phase clock to the time when the encoder ADCs are strobed and the encoder counter value is latched, in units of 16 cycles of the encoder ADC clock for the IC. It is only used on channels where **Gate3[*i*].Chan[*j*].AtanEna** is set to 1 to enable the automatic combination of the fractional-count arctangent result from the encoder ADCs with the latched whole-count value from the counter.

This element permits the user to minimize the delay between the sampling of the encoder or resolver and the phase or servo interrupt that causes the software to use the resulting value. There must be least 25 encoder ADC clock cycles between the strobe/latch and the phase/servo interrupt to receive and process the ADC data.

Gate3[*i*].EncLatchDelay constitutes bits 0 – 7 of the full-word element **Gate3[*i*].PhaseServoClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].FiltClockDiv

Description: IC secondary filter clock frequency control

Range: 0 .. 15

Units: none

Default: 4 ($f_{EncClock}/16$)

Gate3[*i*].FiltClockDiv specifies the frequency of the clock signal that controls the signal propagation through optional secondary digital delay filters for the flag inputs, with any input state present for only a single clock cycle filtered out. The lower the clock frequency, the longer a noise pulse can be filtered out. This secondary filter is only used if **Gate3[*i*].Chan[*j*].FlagFilt2Ena** is set to 1.

Some users will find it desirable to employ additional filtering using a lower clock frequency on the flag inputs before their state is read by the processor. The encoder sample clock frequency that controls the primary digital delay filter must be kept relatively high to permit high count rates. Note that any capture and trigger functions of the flags are performed before the secondary filter, so use of the secondary filter does not delay these functions.

This element controls the frequency by specifying how many times encoder sample clock signal (not the 100 MHz original clock signal) is divided by 2. The equation for the clock frequency is:

$$f_{FiltClk} = \frac{f_{EncClk}}{2^{EncClockDiv}}$$

The following table shows the possible settings of **Gate3[*i*].FiltClockDiv** and the frequencies they produce:

Setting	Divisor	Frequency	Setting	Divisor	Frequency
0	1	f_{EncClk}	8	256	$f_{EncClk}/256$
1	2	$f_{EncClk}/2$	9	512	$f_{EncClk}/512$
2	4	$f_{EncClk}/4$	10	1,024	$f_{EncClk}/1024$
3	8	$f_{EncClk}/8$	11	2,048	$f_{EncClk}/2048$
4	16	$f_{EncClk}/16$	12	4,096	$f_{EncClk}/4096$
5	32	$f_{EncClk}/32$	13	8,192	$f_{EncClk}/8192$
6	64	$f_{EncClk}/64$	14	16,384	$f_{EncClk}/16384$
7	128	$f_{EncClk}/128$	15	32,768	$f_{EncClk}/32768$

Generally, the lowest clock frequency that permits a fast enough response to real flag transitions should be chosen.

Gate3[*i*].FiltClockDiv constitutes bits 8 – 11 of the full-word element **Gate3[*i*].HardwareClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].GpioCtrl

Description: IC general-purpose I/O control

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].GpioCtrl determines how the general-purpose I/O on the IC is processed when used as inputs. This 32-bit element consists of 4 independent bytes, one for each "block" of I/O points. The settings of this variable are only relevant for IC pins used as general-purpose inputs.

The four bytes of **Gate3[*i*].GpioCtrl** and the I/O banks they control are:

Byte	Bits	I/O Bank Index	I/O Bank Pin #	Note
3	31 – 24	3	4	Shares pins with Chan[3]
2	23 – 16	2	3	Shares pins with Chan[2]
1	15 – 08	1	2	Shares pins with Chan[1]
0	07 – 00	0	1	Dedicated I/O pins

Within each byte, the control is broken down by bits:

- Bits 7 & 6: Gray-to-binary conversion control
  = 00: No Gray-to-binary conversion performed
  = 01: Independent Gray-to-binary conversion on input bits 31 – 08 and 07 – 00
  = 10: Independent Gray-to-binary conversion on input bits 31 – 16 and 15 – 00
  = 11: Full Gray-to-binary conversion on input bits 31 – 00

- Bit 5: Secondary digital delay filter (using FiltClock) control
  = 0: Secondary filter disabled (bypassed)
  = 1: Secondary filter enabled

- Bit 4: Primary digital delay filter (using EncClock) control
  = 0: Primary filter disabled (bypassed)
  = 1: Primary filter enabled

- Bit 3: Latch inhibit signal level control:
  = 0: High inhibits latching
  = 1: Low inhibits latching

- Bit 2: Latch inhibit enable control
  = 0: Latch inhibit disabled
  = 1: Latch inhibit enabled

- Bit 1: Latching clock edge select
  = 0: Latch on clock falling edge
  = 1: Latch on clock rising edge

- Bit 0: Latching clock select
  = 0: Use Phase clock to latch
  = 1: Use Servo clock to latch

Gate3[*i*].GpioCtrl can also be accessed as **Gate3[*i*].GpioMode[0]**. Power PMAC will save and restore the value of this element using **Gate3[*i*].GpioMode[0]**. This element is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].GpioDir[*j*]

Description: IC general-purpose I/O bank direction

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: Hardware-specific
$00FF0000 (**Gate3[0].GpioDir[0]**) Power Brick
$00000000 (others)

Gate3[*i*].GpioDir[*j*] determines whether the general-purpose I/O pins of the selected bank of general-purpose I/O on the IC are used as inputs or outputs. This is an array of 4 setup elements, each 32 bits in length. Each element **GpioDir[0]** to **GpioDir[3]** corresponds to the GPIO bank of the same index value 0 to 3 (for the 1st through 4th banks, respectively).

Each element **Gate3[*i*].GpioDir[*j*]** consists of 32 independent bits 0 – 31. Each bit *n* controls the direction of signal GPIO*n* for the bank. A default value of 0 in the bit specifies its use as an input; a 1 in the bit specifies its use as an output. The bit is only used if the signal pin has been selected for GPIO use by the corresponding bit of **Gate3[*i*].GpioMode[*j*]**. The 1st bank (index 0) is always used for GPIO.

Gate3[*i*].GpioDir[*j*] is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].GpioMode[*j*]

Description: IC general-purpose I/O bank enable

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].GpioMode[*j*] determines whether the pins of the selected bank of general-purpose I/O on the IC are used as general-purpose I/O or for dedicated servo functions. There are four banks of 32 I/O points on each IC. The first bank is always used for general-purpose I/O. The second,

third, and fourth banks share pins with the second, third, and fourth servo channels on the IC. Different board designs will use these pins for different purposes.

Because the first bank (index 0) of general-purpose I/O has dedicated pins, **Gate3[*i*].GpioMode[0]** does not have this function. **Gate3[*i*].GpioMode[0]** shares a register with **Gate3[*i*].GpioCtrl** and serves only as an alternate name for that element. Do not use **GpioMode[0]** as you would use **GpioMode[1]**, **[2]**, or **[3]**.

Gate3[*i*].GpioMode[1] determines whether pins in the second bank (index 1) are used for general-purpose I/O or dedicated servo functions.

Gate3[*i*].GpioMode[2] determines whether pins in the third bank (index 2) are used for general-purpose I/O or dedicated servo functions.

Gate3[*i*].GpioMode[3] determines whether pins in the fourth bank (index 3) are used for general-purpose I/O or dedicated servo functions.

Each **Gate3[*i*].GpioMode[*j*]** (***j*** = 1 to 3) element contains 32 independent bits. Each bit controls a separate pin in the bit. The following table shows each bit with the GPIO signal it would select if set to 1, and the servo signal it would select if set to 0. The hardware channel *k* is equal to *j* + 1.

The following table shows the IC pin use for each bit:

Bit	GPIO Signal	Servo Signal	Bit	GPIO Signal	Servo Signal
00	GPIO00	ENCA*k*-	16	GPIO16	AENA*k*
01	GPIO01	ENCA*k*+	17	GPIO17	OUTB*k*
02	GPIO02	ENCB*k*-	18	GPIO18	OUTC*k*
03	GPIO03	ENCB*k*+	19	GPIO19	OUTD*k*
04	GPIO04	ENCA*k*	20	GPIO20	ADCENCA*k*
05	GPIO05	ENCB*k*	21	GPIO21	ADCENCB*k*
06	GPIO06	ENCC*k*	22	GPIO22	ADCAMPA*k*
07	GPIO07	FAULT*k*	23	GPIO23	ADCAMPB*k*
08	GPIO08	HOME*k*	24	GPIO24	PWMAT*k*
09	GPIO09	MLIM*k*	25	GPIO25	PWMAB*k*
10	GPIO10	PLIMk	26	GPIO26	PWMBT*k*
11	GPIO11	USER*k*	27	GPIO27	PWMBB*k*
12	GPIO12	FLAGT*k*	28	GPIO28	PWMCT*k*
13	GPIO13	FLAGU*k*	29	GPIO29	PWMCB*k*
14	GPIO14	FLAGV*k*	30	GPIO30	PWMDT*k*
15	GPIO15	FLAGW*k*	31	GPIO31	PWMDB*k*

The proper setting of each **Gate3[*i*].GpioMode[*j*]** element is dependent on the hardware in which the IC is used. For axis-interface hardware such as the ACC-24E3, these elements should be set to $0 to select the dedicated servo signals.

Gate3[*i*].GpioMode[*j*] is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].GpioPol[*j*]

Description: IC general-purpose I/O bank polarity

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].GpioPol[*j*] determines whether the general-purpose I/O pins of the selected bank of general-purpose I/O on the IC are inverting or non-inverting I/O points. This is an array of 4 setup elements, each 32 bits in length. Each element **GpioPol[0]** to **GpioPol[3]** corresponds to the GPIO bank of the same index value 0 to 3 (for the 1st through 4th banks, respectively).

Each element **Gate3[*i*].GpioPol[*j*]** consists of 32 independent bits 0 – 31. Each bit *n* controls the direction of signal GPIO*n* for the bank. A default value of 0 in the bit specifies its use as a non-inverting (high-true) I/O point; a 1 in the bit specifies its use as an inverting (low-true) I/O point. The bit is only used if the signal pin has been selected for GPIO use by the corresponding bit of **Gate3[*i*].GpioMode[*j*]**. The 1st bank (index 0) is always used for GPIO. The bit is used regardless of whether the I/O point is used as an input or output.

If selected as a non-inverting I/O point, a 0 in the matching bit of **Gate3[*i*].GpioData[*j*]** for the same bank corresponds to a low voltage at the IC signal pin. A 1 in the matching bit of **Gate3[*i*].GpioData[*j*]** corresponds to a high voltage at the IC signal pin.

If selected as an inverting I/O point, a 0 in the matching bit of **Gate3[*i*].GpioData[*j*]** for the same bank corresponds to a high voltage at the IC signal pin. A 1 in the matching bit of **Gate3[*i*].GpioData[*j*]** corresponds to a low voltage at the IC signal pin.

Note that this definition concerns the I/O polarity at the IC itself. External input or output hardware may cause a separate inversion.

Gate3[*i*].GpioPol[*j*] is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].HardwareClockCtrl

Description: IC control word for hardware clock configuration

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $55555403

Gate3[*i*].HardwareClockCtrl is the full-word element that comprises the multi-channel setup for generating many of the clock frequencies used in the IC. It is comprised of the following partial-word elements:

Partial-Word Element	Bits	Functionality
Gate3[*i*].AdcAmpClockDiv	31 – 28	Amplifier ADC clock division power
Gate3[*i*].AdcEncClockDiv	27 – 24	Encoder ADC clock division power
Gate3[*i*].DacClockDiv	23 – 20	DAC clock division power
Gate3[*i*].PfmClockDiv	19 – 16	PFM generation clock division power
Gate3[*i*].EncClockDiv	15 – 12	Encoder sampling clock division power
Gate3[*i*].FiltClockDiv	11 – 08	Input filtering clock division power
Gate3[*i*].ClockPol	07 – 05	DAC and ADC clock output polarity
Gate3[*i*].ServoClockDiv	03 – 00	Servo clock division factor

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only, and as a 32-bit word.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Gate3[*i*].HardwareClockCtrl is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate3[*i*].MacroEnableA

Description: IC MACRO Bank A enable configuration

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].MacroEnableA is the full-word element that comprises the enabling components for the first set of MACRO interfaces ("A") for the IC. It is comprised of the following components (which cannot be accessed as independent elements):

Component	Bits	Hex Digit #	Functionality
MacroMasterNumA	31 – 28	1	MACRO A master address
MacroSyncNodeA	27 – 24	2	MACRO A sync packet node number
MacroNodeEnaA	23 – 08	3 – 6	MACRO A node enable bits
(Reserved)	07 – 00	7 – 8	(*Reserved for future use*)

The 4-bit component *MacroMasterNumA* specifies the number of the master address for all 16 "A" nodes in the IC. This 4-bit value, along with the 4-bit node number, will be transmitted as part of the header byte for each node packet.

The 4-bit component *MacroSyncNodeA* specifies the number of the "sync packet" node. If this IC is not acting as the synchronizing master for the ring, and is not receiving the phase clock signal directly from another IC in the same device, receipt of a data packet for this node will cause an internal synchronizing signal to be generated, keeping this IC properly locked to the ring timing.

The 16-bit component *MacroNodeEnaA* specifies which "A" nodes are enabled on the ring. Bit *n* controls Node *n*; a value of 0 in the bit disables the node; a value of 1 in the bit enables the node.

Gate3[*i*].MacroEnableA is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].MacroEnableB

Description: IC MACRO Bank B enable configuration

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].MacroEnableB is the full-word element that comprises the enabling components for the second set of MACRO interfaces ("B") for the IC. It is comprised of the following components (which cannot be accessed as independent elements):

Component	Bits	Hex Digit #	Functionality
MacroMasterNumB	31 – 28	1	MACRO B master address
MacroSyncNodeB	27 – 24	2	MACRO B sync packet node number
MacroNodeEnaB	23 – 08	3 – 6	MACRO B node enable bits
(Reserved)	07 – 00	7 – 8	(*Reserved for future use*)

The 4-bit component *MacroMasterNumB* specifies the number of the master address for all 16 "B" nodes in the IC. This 4-bit value, along with the 4-bit node number, will be transmitted as part of the header byte for each node packet.

The 4-bit component *MacroSyncNodeB* specifies the number of the "sync packet" node. If this IC is not acting as the synchronizing master for the ring, and is not receiving the phase clock signal directly from another IC in the same device, receipt of a data packet for this node will cause an internal synchronizing signal to be generated, keeping this IC properly locked to the ring timing.

The 16-bit component *MacroNodeEnaB* specifies which "B" nodes are enabled on the ring. Bit *n* controls Node *n*; a value of 0 in the bit disables the node; a value of 1 in the bit enables the node.

Gate3[*i*].MacroEnableB is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].MacroModeA

Description: IC MACRO Bank A status and control

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].MacroModeA is the full-word element that comprises the control and status components for the first set of MACRO interfaces ("A") for the IC. It is comprised of the following components (which cannot be accessed as independent elements):

Component	Bits	Hex Digit #	Functionality
(Reserved)	31 – 24	1 – 2	(*Reserved for future use*)
MacroMasterChkDisA	23 – 16	3 – 4	MACRO A master check disable
MacroSyncEnaA	15	5	MACRO A phase clock sync enable
MacroSyncRcvdA	14	5	MACRO A sync packet received status
MacroStationTypeA	13 – 12	5	MACRO A station type
MacroUnderrunErrA	11	6	MACRO A data underrun error status
MacroParityErrA	10	6	MACRO A parity/CRC error status
MacroCodeErrA	09	6	MACRO A byte coding error status
MacroOverrunErrA	08	6	MACRO A data overrun error status
(Reserved)	07 – 00	7 – 8	(*Reserved for future use*)

The 8-bit component *MacroMasterChkDisA* specifies whether the IC will check the master number for the highest-numbered eight "A" nodes on the IC. Bit *n* of the component (which is bit *n*+8 of the full-word element) controls Node *n*+8 of the IC. If the bit is 0, the IC will check the master number of the incoming packet for the matching node number, so the packet can be used for point-to-point communication across the ring. If the bit is 1, the IC will not check the master number of the incoming packet for the matching node number, so the packet can be used for "broadcast" purposes.

The 1-bit component *MacroSyncEnaA* specifies whether the IC's phase clock will be synchronized by the receipt of the specified "sync packet" or not. If it is set to 0, no synchronization will be performed; if it is set to 1, the IC's phase clock timer will be synchronized to the timing of the receipt of the sync packet on the ring. This bit is not used if the IC is the synchronizing master for the ring, or if it is receiving the phase clock from another IC in the same device.

The 1-bit component *MacroSyncRcvdA* is a read-only status bit that is set to 1 when the specified sync packet is received. It is automatically set to 0 when this register is read, so it indicates to the processor whether a sync packet has been received since the last time the register was read.

The 2-bit component *MacroStationTypeA* specifies the function of the station containing this IC on the MACRO ring. . Its possible values specify the following frequencies:

- 0: Slave
- 1: Master
- 2: (Reserved for future use)
- 3: Synchronizing master

The 1-bit component *MacroUnderrunErrA* is a read-only status bit that is set to 1 when the IC receives a data packet with too few bytes in it. It is automatically set to 0 when this register is read, so it indicates to the processor whether an underrun error has occurred since the last time the register was read.

The 1-bit component *MacroParityErrA* is a read-only status bit that is set to 1 when the IC receives a data packet with a parity or CRC check error. It is automatically set to 0 when this register is read, so it indicates to the processor whether a parity/CRC error has occurred since the last time the register was read.

The 1-bit component *MacroCodeErrA* is a read-only status bit that is set to 1 when the IC receives a data packet with an illegally coded byte in it. It is automatically set to 0 when this register is read, so it indicates to the processor whether a coding error has occurred since the last time the register was read.

The 1-bit component *MacroOverrunErrA* is a read-only status bit that is set to 1 when the IC receives a data packet with too many bytes in it. It is automatically set to 0 when this register is read, so it indicates to the processor whether an overrun error has occurred since the last time the register was read.

The control components of **Gate3[*i*].MacroModeA** are write-protected, so cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].MacroModeB

Description: IC MACRO Bank B status and control

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].MacroModeB is the full-word element that comprises the control and status components for the first set of MACRO interfaces (“B”) for the IC. It is comprised of the following components (which cannot be accessed as independent elements):

Component	Bits	Hex Digit #	Functionality
(Reserved)	31 – 24	1 – 2	(*Reserved for future use*)
MacroMasterChkDisB	23 – 16	3 – 4	MACRO B master check disable
MacroSyncEnaB	15	5	MACRO B phase clock sync enable
MacroSyncRcvdB	14	5	MACRO B sync packet received status
MacroStationTypeB	13 – 12	5	MACRO B station type
MacroUnderrunErrB	11	6	MACRO B data underrun error status
MacroParityErrB	10	6	MACRO B parity/CRC error status
MacroCodeErrB	09	6	MACRO B byte coding error status
MacroOverrunErrB	08	6	MACRO B data overrun error status
(Reserved)	07 – 00	7 – 8	(*Reserved for future use*)

The 8-bit component *MacroMasterChkDisB* specifies whether the IC will check the master number for the highest-numbered eight "B" nodes on the IC. Bit *n* of the component (which is bit *n*+8 of the full-word element) controls Node *n*+8 of the IC. If the bit is 0, the IC will check the master number of the incoming packet for the matching node number, so the packet can be used for point-to-point communication across the ring. If the bit is 1, the IC will not check the master number of the incoming packet for the matching node number, so the packet can be used for "broadcast" purposes.

The 1-bit component *MacroSyncEnaB* specifies whether the IC's phase clock will be synchronized by the receipt of the specified "sync packet" or not. If it is set to 0, no synchronization will be performed; if it is set to 1, the IC's phase clock timer will be synchronized to the timing of the receipt of the sync packet on the ring. This bit is not used if the IC is the synchronizing master for the ring, or if it is receiving the phase clock from another IC in the same device.

The 1-bit component *MacroSyncRcvdB* is a read-only status bit that is set to 1 when the specified sync packet is received. It is automatically set to 0 when this register is read, so it indicates to the processor whether a sync packet has been received since the last time the register was read.

The 2-bit component *MacroStationTypeB* specifies the function of the station containing this IC on the MACRO ring. . Its possible values specify the following frequencies:

- 0: Slave
- 1: Master
- 2: (Reserved for future use)
- 3: Synchronizing master

The 1-bit component *MacroUnderrunErrB* is a read-only status bit that is set to 1 when the IC receives a data packet with too few bytes in it. It is automatically set to 0 when this register is read, so it indicates to the processor whether an underrun error has occurred since the last time the register was read.

The 1-bit component *MacroParityErrB* is a read-only status bit that is set to 1 when the IC receives a data packet with a parity or CRC check error. It is automatically set to 0 when this register is read, so it indicates to the processor whether a parity/CRC error has occurred since the last time the register was read.

The 1-bit component *MacroCodeErrB* is a read-only status bit that is set to 1 when the IC receives a data packet with an illegally coded byte in it. It is automatically set to 0 when this register is read, so it indicates to the processor whether a coding error has occurred since the last time the register was read.

The 1-bit component *MacroOverrunErrB* is a read-only status bit that is set to 1 when the IC receives a data packet with too many bytes in it. It is automatically set to 0 when this register is read, so it indicates to the processor whether an overrun error has occurred since the last time the register was read.

The control components of **Gate3[*i*].MacroModeB** are write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].PfmClockDiv

Description: IC pulse-frequency modulation clock frequency control

Range: 0 .. 15

Units: none

Default: 5 (3.125 MHz)

Gate3[*i*].PfmClockDiv specifies the frequency of the clock signal that controls the pulse-frequency modulation circuitry the IC's channels. Each PFM clock cycle, the 24-bit active Phase D command value is added into a 24-bit accumulator. Every time the accumulator "rolls over", an output pulse is generated. Thus, the pulse output frequency is proportional to both the command value (which can change often) and the PFM clock frequency (which is usually fixed for an application).

This element controls the frequency by specifying how many times an internal 100-MHz clock signal is divided by 2. The equation for the clock frequency is:

$$f_{PfmClk} = \frac{100MHz}{2^{PfmClockDiv}}$$

The following table shows the possible settings of **Gate3[*i*].PfmClockDiv** and the 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

Generally, a clock frequency that permits both the maximum and minimum pulse output frequencies is chosen.

Gate3[*i*].PfmClockDiv constitutes bits 16 – 19 of the full-word element **Gate3[*i*].HardwareClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].PhaseClockDiv

Description: IC phase clock input division factor

Range: 0 .. 3

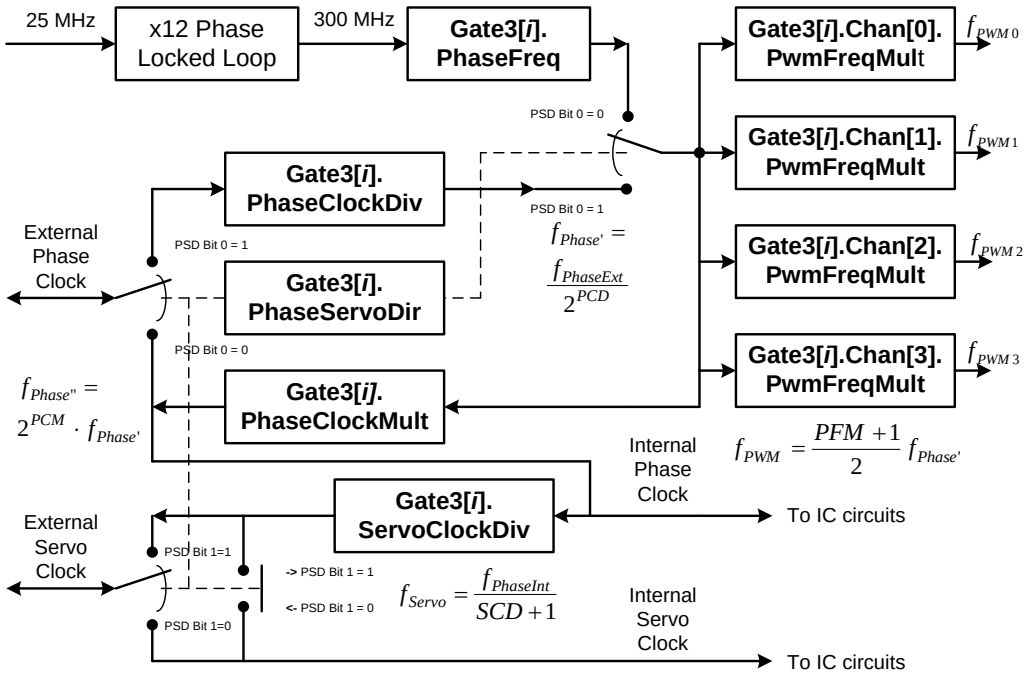
Units: none

Default: 0

Gate3[*i*].PhaseClockDiv specifies the division factor between the external input phase clock frequency and the internal phase clock frequency. The external input phase clock frequency is divided by a factor of (2 raised to the power of **Gate3[*i*].PhaseClockDiv**) to obtain the internal phase clock frequency.

In most applications, this will be set to 0 so that the external input and internal phase clock frequencies are the same. However, since the range of possible PWM output frequencies is determined by the internal phase clock, and the processor's phase interrupt frequency is determined by the external input phase clock, this element provides some additional flexibility in frequency configuration.

This block diagram shows the how the internal and external clock frequencies relate in the DSPGATE3 IC:



Gate3[*i*].PhaseClockDiv is only used if bit 0 (value 1) of **Gate3[*i*].PhaseServoDir** is set to 1, telling the IC to input and use an externally generated phase clock signal instead of generating its own phase clock signal internally and output it.. In general, all ICs in a Power PMAC system must use a single phase clock signal in order to stay properly synchronized.

Gate3[*i*].PhaseClockDiv constitutes bits 12 – 13 of the full-word element **Gate3[*i*].PhaseServoClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].PhaseClockMult

Description: IC phase clock output multiplication factor

Range: 0 .. 3

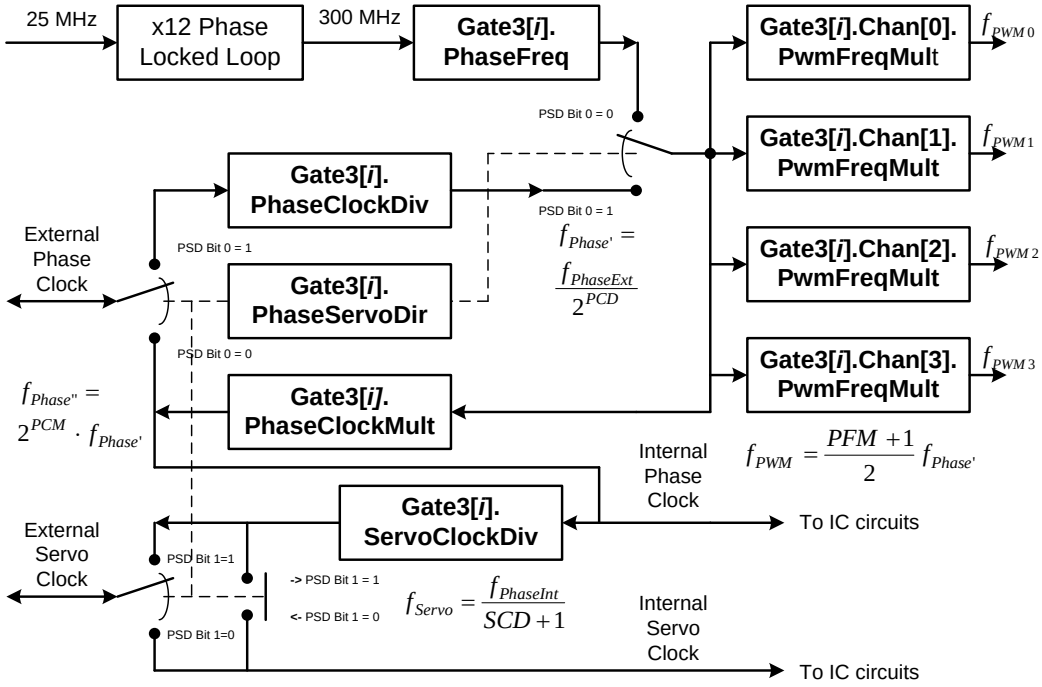
Units: none

Default: 0

Gate3[*i*].PhaseClockMult specifies the multiplication factor between the internal phase clock frequency and the output phase clock frequency. The internal phase clock frequency is multiplied by a factor of (2 raised to the power of **Gate3[*i*].PhaseClockDiv**) to obtain the output phase clock frequency.

In most applications, this will be set to 0 so that the internal and output phase clock frequencies are the same. However, since the range of possible PWM output frequencies is determined by the internal phase clock, and the processor's phase interrupt frequency is determined by the output phase clock, this element provides some additional flexibility in frequency configuration.

This block diagram shows the how the internal and external clock frequencies relate in the DSPGATE3 IC:



Gate3[*i*].PhaseClockMult is only used if bit 0 (value 1) of **Gate3[*i*].PhaseServoDir** is set to 0, telling the IC to generate its own phase clock signal and output it instead of inputting and using an externally generated phase clock signal. In general, all ICs in a Power PMAC system must use a single phase clock signal in order to stay properly synchronized.

Gate3[*i*].PhaseClockMult constitutes bits 14 – 15 of the full-word element **Gate3[*i*].PhaseServoClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].PhaseFreq

Description: IC internal phase clock frequency

Range: 35.76 .. 2,343,463

Units: Hertz

Default: 9035.69

Gate3[*i*].PhaseFreq specifies the frequency of the internal phase clock signal for the IC in units of Hertz. If bit 0 (value 1) of **Gate3[*i*].PhaseServoDir** is set to 0, telling the IC to generate its own phase clock signal and output it instead of inputting and using an externally generated phase clock signal, it also is used for the frequency of the system phase clock. In general, all ICs in a Power PMAC system must use a single phase clock signal in order to stay properly synchronized.

The phase clock signal output from the IC (which provides the "phase interrupt" to the processor) can be from one to four times the frequency of the internal phase clock frequency, as determined by **Gate3[*i*].PhaseClockMult**. However, the range of possible PWM frequencies generated on the channels of the IC is dependent on the internal phase clock frequency, not the output signal frequency. PWM frequencies can be from 0.5 to 4.0 times the internal phase clock frequency, as specified by **Gate3[*i*].Chan[*j*].PwmFreqMult**.

If bit 0 (value 1) of **Gate3[*i*].PhaseServoDir** is set to 1, telling the IC input an externally generated phase clock signal, **Gate3[*i*].PhaseFreq** should be set as closely as possible to the frequency of the external clock as divided by **Gate3[*i*].PhaseClockDiv**. In this case, **Gate3[*i*].PhaseFreq** is still used to drive the internal circuitry, but it is kept synchronized to the (possibly divided) external clock signal through a phase-locked loop (PLL) circuit.

Gate3[*i*].PhaseFreq is a pre-scaled 16-bit floating-point value with a 12-bit fractional mantissa and a 4-bit exponent. It constitutes bits 16 – 31 of the full-word element **Gate3[*i*].PhaseServoClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].PhaseServoClockCtrl

Description: IC control word for phase and servo clock configuration

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $7F950C00

Gate3[*i*].PhaseServoCtrl is the full-word element that comprises the multi-channel setup for generating the phase and servo clocks for the IC. It is comprised of the following partial-word elements:

Partial-Word Element	Bits	Functionality
PhaseFreq	31 – 16	Internal phase clock frequency factor
PhaseClockMult	15 – 14	Output phase clock multiplication factor
PhaseClockDiv	13 – 12	Input phase clock division factor
PhaseServoDir	11 – 10	Phase and servo clock direction control
(Reserved)	09 – 08	(*Reserved for future use*)
EncLatchDelay	07 – 00	Delay in latching encoder counter value

The partial-word element **PhaseFreq** is stored as a normalized 16-bit floating-point value with a 4-bit exponent (base 2) in bits 28 – 31 and a 12-bit fractional mantissa (implied "1") in bits 16 – 27. The stored value is divided down by a factor of 35.76 (600,000,000/16,777,216) from the value entered into the element.

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Gate3[*i*].PhaseServoCtrl is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate3[*i*].PhaseServoDir

Description: IC phase and servo clock direction control

Range: 0 .. 3

Units: Bit field

Default: 0 or 3 (auto-configured on re-initialization)

Gate3[*i*].PhaseServoDir specifies whether the IC uses and outputs its own internally generated phase and servo clock signals, or it uses and inputs externally generated phase and servo clock signals. It is a 2-bit value, with bit 0 (value 1) controlling the phase clock direction, and bit 1 controlling the servo clock direction. A bit value of 0 specifies the use of the internally generated clock signal; a bit value of 1 specifies the use of an external clock signal. This yields four possible values:

- **Gate3[*i*].PhaseServoDir** = 0: Internal phase clock, internal servo clock
- **Gate3[*i*].PhaseServoDir** = 1: External phase clock, internal servo clock
- **Gate3[*i*].PhaseServoDir** = 2: Internal phase clock, external servo clock
- **Gate3[*i*].PhaseServoDir** = 3: External phase clock, external servo clock

It is very rare that opposite directions of the two clock signals would be used in a single IC, so the only commonly used values are 0 and 3.

In any Power PMAC system, there must be one and only one source of the phase and servo clock signals for the system – one of the **Gate1**, **Gate2**, or **Gate3** ICs, or a source external to the system. All other ICs must input these clock signals to stay synchronized.

Gate3[*i*].PhaseServoDir constitutes bits 10 – 11 of the full-word element **Gate3[*i*].PhaseServoClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].ResolverCtrl

Description: IC control word for resolver excitation configuration

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].ResolverCtrl is the full-word element that comprises the multi-channel setup for the resolver interfaces for the IC. It is comprised of the following components (which cannot be accessed as independent elements):

Component	Bits	Hex Digit #	Functionality
ResolverExciteShift	31 – 24	1 – 2	Resolver excitation signal phase shift
ResolverExciteGain	23 – 22	3	Resolver excitation phase control
ResolverExciteFreq	21 – 20	3	Resolver excitation frequency control
(Reserved)	19 – 00	4 – 8	(Reserved for future use)

The 8-bit component *ResolverExciteShift* specifies the phase shift (delay) of the excitation signal. It is in units of 1/512 of an excitation signal period (which is set by component *ResolverExciteFreq*), and can take a value from 0 to 255. This component is generally set

experimentally to the setting that maximizes the magnitude of the returned signals (which can be determined by reading the data structure element **Gate3[*i*].Chan[*j*].SumOfSquares**).

The 2-bit component *ResolverExciteGain* specifies the magnitude of the output. Its possible values specify the following magnitudes:

- 0: ¼ of full magnitude
- 1: ½ of full magnitude
- 2: ¾ of full magnitude
- 3: Full magnitude

The 2-bit component *ResolverExciteFreq* specifies the frequency of the output as derived from the Phase clock frequency. Its possible values specify the following frequencies:

- 0: Phase clock frequency
- 1: Phase clock frequency divided by 2
- 2: Phase clock frequency divided by 4
- 3: Phase clock frequency divided by 6

Gate3[*i*].ResolverCtrl is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].SerialEncCtrl

Description: IC control word for serial encoder configuration

Range: $0 .. $FFFFFFFF

Units: Bit field

Default: $F4000000

Gate3[*i*].SerialEncCtrl is the full-word element that comprises the multi-channel setup for serial encoder interfaces for the IC. Further individual channel setup is implemented in **Gate3[*i*].Chan[*j*].SerialEncCmd**.

Gate3[*i*].SerialEncCtrl is comprised of the following components (which cannot be accessed as independent elements):

Component	Bits	Hex Digit #	Functionality
SerialClockMDiv	31 – 24	1 – 2	Serial clock linear division factor
SerialClockNDiv	23 – 20	3	Serial clock exponent division factor
(Reserved)	19 – 18	4	(Reserved for future use)
SerialTrigClockSel	17	4	Serial trigger source select
SerialTrigEdgeSel	16	4	Serial trigger source edge select
SerialTrigDelay	15 – 08	5 – 6	Serial trigger delay from source edge
SerialProtocol	07 – 00	7 – 8	Serial encoder protocol select

The component *SerialClockMDiv* 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 component *SerialClockNDiv*, described below.

The equation for this intermediate clock frequency is:

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

where *M* is short for *SerialClockMDiv*. 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 component *SerialClockNDiv* controls how the final serial-encoder clock frequency is generated from the intermediate clock frequency set by *SerialClockMDiv*. The equation for this final frequency is:

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

where *N* is short for *SerialClockNDiv*. 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 most serial-encoder protocols with an explicit clock signal, 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 clocked protocols with fractional-cycle delay compensation, 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.

The component *SerialTrigClockSel* controls which Power PMAC clock signal causes the encoder to be triggered. If this single-bit component is set to 0, the encoder will be triggered on the phase clock; if it is set to 1, the encoder will be triggered on the servo clock. If the encoder feedback is required for commutation rotor angle feedback, it should be triggered on the phase clock; otherwise it can be triggered on the servo clock.

The component *SerialTrigEdgeSel* controls which edge of the clock signal (phase or servo) selected by *SerialTrigClockSel* initiates the triggering process. If this single-bit component is set to 0, the triggering process starts on the rising edge; if it is set to 1, the triggering process starts on the falling edge.

Power PMAC software expects to have the resulting encoder data available to it immediately after the falling edge of the relevant phase or servo clock signal, which interrupts the processor to initiate the activity that reads this data. Since minimum delay from trigger to use is desirable, it is better to start the triggering on rising clock edge if the data can be fully transferred before the falling edge. If this is not possible, the falling edge should be used to start the triggering process.

The component *SerialTrigDelay* specifies the delay from the specified clock edge to the actual start of the output signal that will trigger the encoder response, in units of the serial encoder clock. A non-zero value can be used to minimize the delay between triggering the encoder and its resulting use by the Power PMAC. A smaller time delay will reduce the phase lag of the servo loop and improve the stability characteristics of the loop.

The component *SerialProtocol* controls which serial-encoder protocol is selected for all channels of the IC. This 4-bit component can take a value from 0 to 15. The following table shows the protocol selected for each value of this component:

Value	Protocol	Value	Protocol	Value	Protocol	Value	Protocol
0	None	4	Hiperface	8	Panasonic	12	(Reserved)
1	SPI	5	Sigma I	9	Mitutoyo	13	(Reserved)
2	SSI	6	Sigma II	10	Kawasaki	14	(Reserved)
3	EnDat	7	Tamagawa	11	(Reserved)	15	SW control

Gate3[*i*].SerialEncCtrl is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Additional control of the serial encoder interface occurs on an individual-channel basis with the saved setup element **Gate3[*i*].Chan[*j*].SerialEncCmd**.

More details are given in the Hardware Reference Manual for the particular interface device (e.g. ACC-24E3) used to process the serial encoder. Below are common settings for each supported protocol.

SPI Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for an SPI encoder.

SerialClockMDiv: = (100 / f_{bit}) – 1 // Serial clock frequency = bit transmission frequency
SerialClockNDiv: = 0 // No further division unless f < 400 kHz
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $01 // Selects SPI protocol

For example, for a 4 MHz bit transmission rate, ***SerialClockMDiv*** = (100 / 4) - 1 = 24 ($18) and **Gate3[*i*].SerialEncCtrl** is set to $18000001 for triggering on the rising edge of phase clock without delay.

1				8				0				0				0				0				0				1			
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	1	1	0	0	0	0	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
SerialClockMDiv								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*								*SerialProtocol*							

SSI Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for an SSI encoder.

SerialClockMDiv: = (100 / f_{bit}) - 1 // Serial clock frequency = bit transmission frequency
SerialClockNDiv: = 0 // No further division unless f < 400 kHz
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $02 // Selects SSI protocol

For example, for a 2.5 MHz bit transmission rate, ***SerialClockMDiv*** = (100 / 2.5) - 1 = 39 ($23) and **Gate3[*i*].SerialEncCtrl** is set to $23000002 for triggering on the rising edge of phase clock without delay.

2				3				0				0				0				0				0				2			
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	1	0	0	0	1	1	0	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
SerialClockMDiv								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*								*SerialProtocol*							

EnDat 2.1/2.2 Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for an EnDat encoder. The serial clock frequency is set 25 times higher than the external clock frequency, which is the bit transmission frequency, to permit the implementation of delay compensation, which allows high-frequency transmission over long distances.

SerialClockMDiv: = (4 / f_{bit}) - 1 // Serial clock freq. = 25x bit transmission freq.
SerialClockNDiv: = 0 // No further division
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $03 // Selects EnDat protocol

For example, for a 2.0 MHz bit transmission rate, ***SerialClockMDiv*** = (4 / 2) - 1 = 1 ($01) and **Gate3[*i*].SerialEncCtrl** is set to $01000003 for triggering on the rising edge of phase clock without delay.

0				1				0				0				0				0				0				3			
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	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
SerialClockMDiv				*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*												*SerialProtocol*							

Hiperface Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for a Hiperface encoder. Because there is no explicit clock signal, the serial clock frequency is set 20 times higher than the bit transmission frequency to "oversample" the input data stream. For the default 9600 baud transmission of the Hiperface encoder, this clock frequency should be 192 kHz. This requires the two-stage division by both ***SerialClockMDiv*** and ***SerialClockNDiv***. It is recommended to divide down as much as possible in the first stage to get as close to the ideal frequency as possible.

SerialClockMDiv:	= $82	// Divide by 130 to get ~768 kHz
SerialClockNDiv:	= 2	// Divide by 4 to get ~ 192 kHz
SerialTrigClockSel:	= 1	// Use servo clock
SerialTrigEdgeSel:	= 1	// Use falling clock edge
SerialTrigDelay:	= 0	// No delay
SerialProtocol:	= $04	// Selects Hiperface protocol

For example, for a 9600 bit/sec transmission rate, ***SerialClockMDiv*** = 130 ($82), ***SerialClockNDiv*** = 2 and **Gate3[*i*].SerialEncCtrl** is set to $82230004 for triggering on the falling edge of servo clock without delay. Since this is a "one-shot" read, the selection of the triggering clock edge does not matter much.

8				2				2				3				0				0				0				4			
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
1	0	0	0	0	0	1	0	0	0	1	0	-	-	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
SerialClockMDiv				*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*												*SerialProtocol*							

Yaskawa Sigma I Protocol

Note

Yaskawa no longer produces Sigma I absolute encoders. However, newer generations of Yaskawa Sigma servo drives synthesize the Yaskawa Sigma I protocol for return to the controller even when using newer Sigma II, III, and V encoders.

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for a Yaskawa Sigma I absolute encoder. Because there is no explicit clock signal, the serial clock frequency is set 20 times higher than the bit transmission frequency to "oversample" the input data stream. For the default 9600 baud transmission of the Yaskawa Sigma I encoder, this clock frequency should be 192 kHz. This requires the two-stage division by both ***SerialClockMDiv*** and ***SerialClockNDiv***. It is recommended to divide down as much as possible in the first stage to get as close to the ideal frequency as possible.

SerialClockMDiv:	= $82	// Divide by 130 to get ~768 kHz
SerialClockNDiv:	= 2	// Divide by 4 to get ~ 192 kHz
SerialTrigClockSel:	= 1	// Use servo clock

SerialTrigEdgeSel: = 1 // Use falling clock edge
SerialTrigDelay: = 0 // No delay
SerialProtocol: = $05 // Selects Yaskawa Sigma I protocol

For example, for a 9600 bit/sec transmission rate, ***SerialClockMDiv*** = 130 ($82), ***SerialClockNDiv*** = 2 and **Gate3[*i*].SerialEncCtrl** is set to $82230005 for triggering on the falling edge of servo clock without delay. Since this is a "one-shot" read, the selection of the triggering clock edge does not matter much.

8				2				2				3				0				0				0				5			
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
1	0	0	0	0	0	1	0	0	0	1	0	-	-	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
SerialClockMDiv				*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*												*SerialProtocol*							

Yaskawa Sigma II/III/V Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for a Yaskawa II/III/V encoder.

SerialClockMDiv: = 0 // 100 MHz serial clock freq. = 25x bit transmission freq.
SerialClockNDiv: = 0 // No further division
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $06 // Selects Yaskawa II/III/V protocol

For example, for the standard 4.0 MHz bit transmission rate, a 100 MHz serial clock frequency is used, and **Gate3[*i*].SerialEncCtrl** is set to $00000006 for triggering on the rising edge of phase clock without delay.

0				0				0				0				0				0				0				6			
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	0	0	0	0	0	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
SerialClockMDiv				*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*												*SerialProtocol*							

Tamagawa FA-Coder Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for a Tamagawa FA-Coder encoder. Because there is no explicit clock signal, the serial encoder clock in the DSPGATE3 must be 20 times the bit transmission frequency to "oversample" the input data stream.

SerialClockMDiv: = (5 / f_{bit}) - 1 // Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv: = 0 // No further division
SerialTrigClockSel: = 0 // Use phase clock if possible
SerialTrigEdgeSel: = 0 // Use rising clock edge if possible
SerialTrigDelay: = 0 // Can increase from 0 if possible to reduce latency
SerialProtocol: = $07 // Selects Tamagawa FA-Coder protocol

For example, for a 2.5 MHz bit transmission rate, a 50 MHz serial clock frequency is used, and **Gate3[*i*].SerialEncCtrl** is set to $01000007 for triggering on the rising edge of phase clock without delay.

0				1				0				0				0				0				0				7			
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	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
SerialClockMDiv				*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*												*SerialProtocol*							

Panasonic Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for a Panasonic serial encoder. Because there is no explicit clock signal, the serial encoder clock in the DSPGATE3 must be 20 times the bit transmission frequency to "oversample" the input data stream.

SerialClockMDiv:	= (5 / f_{bit}) - 1	// Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv:	= 0	// No further division
SerialTrigClockSel:	= 0	// Use phase clock if possible
SerialTrigEdgeSel:	= 0	// Use rising clock edge if possible
SerialTrigDelay:	= 0	// Can increase from 0 if possible to reduce latency
SerialProtocol:	= $08	// Selects Panasonic protocol

For example, for a 2.5 MHz bit transmission rate, a 50 MHz serial clock frequency is used, and **Gate3[*i*].SerialEncCtrl** is set to $01000008 for triggering on the rising edge of phase clock without delay.

0				1				0				0				0				0				0				8			
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	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
SerialClockMDiv				*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*												*SerialProtocol*							

Mitutoyo Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for a Mitutoyo serial encoder. Because there is no explicit clock signal, the serial encoder clock in the DSPGATE3 must be 20 times the bit transmission frequency to "oversample" the input data stream.

SerialClockMDiv:	= (5 / f_{bit}) - 1	// Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv:	= 0	// No further division
SerialTrigClockSel:	= 0	// Use phase clock if possible
SerialTrigEdgeSel:	= 0	// Use rising clock edge if possible
SerialTrigDelay:	= 0	// Can increase from 0 if possible to reduce latency
SerialProtocol:	= $09	// Selects Mitutoyo protocol

For example, for a 2.5 MHz bit transmission rate, a 50 MHz serial clock frequency is used, and **Gate3[*i*].SerialEncCtrl** is set to $01000009 for triggering on the rising edge of phase clock without delay.

0				1				0				0				0				0				0				9			
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	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
SerialClockMDiv				*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*												*SerialProtocol*							

Kawasaki Protocol

The following list shows typical settings of **Gate3[*i*].SerialEncCtrl** for a Kawasaki serial encoder. Because there is no explicit clock signal, the serial encoder clock in the DSPGATE3 must be 20 times the bit transmission frequency to "oversample" the input data stream.

SerialClockMDiv:	= (5 / f_{bit}) - 1	// Serial clock freq. = 20x bit transmission freq.
SerialClockNDiv:	= 0	// No further division
SerialTrigClockSel:	= 0	// Use phase clock if possible
SerialTrigEdgeSel:	= 0	// Use rising clock edge if possible
SerialTrigDelay:	= 0	// Can increase from 0 if possible to reduce latency
SerialProtocol:	= $0A	// Selects Kawasaki protocol

For example, for a 2.5 MHz bit transmission rate, a 50 MHz serial clock frequency is used, and **Gate3[*i*].SerialEncCtrl** is set to $0100000A for triggering on the rising edge of phase clock without delay.

0				1				0				0				0				0				0				A			
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	0	0	0	1	0	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
SerialClockMDiv								*SerialClockNDiv*				--	--	*TC*	*TE*	*SerialTrigDelay*								*SerialProtocol*							

Gate3[*i*].ServoClockDiv

Description: IC servo clock frequency control

Range: 0 .. 15

Units: none

Default: 3

Gate3[*i*].ServoClockDiv specifies the frequency of the internally generated servo clock signal for the IC by controlling how many times it is divided down from the IC's internal phase clock frequency (which is set by **Gate3[*i*].PhaseFreq**). The equation for the servo clock frequency is:

$$f_{ServoClock} = \frac{f_{PhaseClock}}{ServoClockDiv + 1}$$

This internally generated servo clock is only used if **Gate3[*i*].PhaseServoDir** bit 1 (value 2) is set to 0, specifying that the IC is to use its own servo clock and output it. If **Gate3[*i*].PhaseServoDir** bit 1 is set to 1, the IC will expect an externally generated servo clock signal and use it instead of its own. In general, all ICs in a Power PMAC system must use a single servo clock signal in order to stay properly synchronized.

For execution of trajectories at the proper speed, **Sys.ServoPeriod** must be set properly to tell the trajectory generation software what the Servo clock cycle time is. The formula for **Sys.ServoPeriod** is:

$$Sys.ServoPeriod = \frac{1}{ServoFreq(kHz)}$$

In terms of the variables that determine the Servo clock frequency, the formula for **Sys.ServoPeriod** is:

$$Sys.ServoPeriod = 1000 * \frac{(Gate3[i].ServoClockDiv + 1)}{Gate3[i].PhaseFreq}$$

Gate3[*i*].ServoClockDiv constitutes bits 0 – 3 of the full-word element **Gate3[*i*].HardwareClockCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*]. Channel-Specific Setup Elements

The setup elements in this section are used to configure the use of the inputs and outputs for the selected servo channel on the IC. Note that channel index values ***j*** can range from 0 to 3, representing hardware channel numbers 1 to 4, respectively.

Gate3[*i*].Chan[*j*].AdcOffset[*k*]

Description: IC channel encoder ADC bias correction

Range: -2^{31} .. $2^{31}-1$

Units: 32-bit ADC LSBs

Default: 0

Gate3[*i*].Chan[*j*].AdcOffset[*k*] elements specify the bias-correction values that are automatically added to the 32-bit registers for the primary "encoder" ADC data values for the channel before the IC performs subsequent arctangent and sum-of-squares calculations. The actual ADC data from *n*-bit ADCs will be in the most-significant *n* bits of these 32-bit registers. Most commonly, 16-bit ADCs are used here, but other resolutions are possible.

The value in **AdcOffset[0]** is automatically added to the input data value in **AdcEnc[0]**; the value in **AdcOffset[1]** is automatically added to the input data value in **AdcEnc[1]**. These are the input values strobed on the rising edge of the phase clock for the A and B phases, yielding the primary position feedback values for sinusoidal feedback types (encoders and resolvers). These offset terms permit the automatically calculated arctangent and sum-of-squares values to include compensation for biases in the analog circuits.

If the required offset value has been determined in units of LSBs of an *n*-bit ADC, this value should be multiplied by 2^{32-n} to get the 32-bit value that should be written into one of these offset registers. For example, if an offset of -7 LSBs of a 16-bit ADC is desired, a value of -7*65,536, or -458,752, should be written to **AdcOffset[*k*]**.

Note that the values in the readable registers **AdcEnc[0]** and **AdcEnc[1]** are not affected by this process; those values are the uncorrected values.

There are no comparable hardware bias-correction terms in the IC for the "amplifier" ADC values. In the case of use of those circuits for motor digital current feedback, the bias correction is done with software terms **Motor[*x*].IaBias** and **Motor[*x*].IbBias**.

Gate3[*i*].Chan[*j*].AtanEna

Description: IC channel arctangent extension enable

Range: 0 .. 1

Units: Boolean

Default: 0

Gate3[*i*].Chan[*j*].AtanEna controls whether the automatic arctangent calculations from the "encoder ADC" registers are used to compute the "fractional count" value in the channel's **PhaseCapture** and **ServoCapture** registers. These calculations permit the IC to calculate directly the angle value from the "sine" and "cosine" inputs of resolvers and sinusoidal encoders. Every phase-clock cycle, the two **AdcEnc[*k*]** registers for the phase are read, the user-set **AdcOffset[*k*]** bias values are added to them, then the (two-argument, +/-180^{o}) arctangent is computed, as is the sum of squares. The arctangent and the sum of squares values are written to read-only channel elements **Atan** and **SumOfSquares**, respectively. These calculations occur regardless of the setting of **AtanEna**.

If **Gate3[*i*].Chan[*j*].AtanEna** is set to the default value of 0, these arctangent calculations are not used to determine the fractional count value. In this mode, how the fractional count value in the channel's **PhaseCapture** and **ServoCapture** registers is calculated is determined by the channel's **TimerMode** setting. The whole-count value in these registers is the value of the encoder counter latched by the falling edge of the phase and servo clock signals, respectively.

If **Gate3[*i*].Chan[*j*].AtanEna** is set to 1, the arctangent calculations are used to calculate the fractional count value of the **PhaseCapture** and **ServoCapture** registers. In this mode, the low 14 bits of these two registers comes from the arctangent calculations, providing 16,384 states per encoder line, or 4096 states per quadrature count. The whole-count value in these registers is the value of the encoder counter latched after the rising edge of the phase clock signal – with a delay set by **Gate3[*i*].EncLatchDelay** before the falling edge of the phase and servo clock signals, respectively. The encoder ADCs are strobed at this same time in this mode.

Gate3[*i*].Chan[*j*].AtanEna constitutes bit 18 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].CaptCtrl

Description: IC Channel position-capture control

Range: 0 .. 15

Units: none

Default: 1

Gate3[*i*].Chan[*j*].CaptCtrl determines which input signal or combination of signals for this PMAC3-style interface IC, and which polarity, triggers a hardware position capture of the counter for this channel's encoder. If a flag input (home, limit, or user) is used, **Gate3[*i*].Chan[*j*].CaptFlagChan** determines which channel's flag is used, and **Gate3[*i*].Chan[*j*].CaptFlagSel** determines which individual flag from the selected channel.

Proper setup of this variable is essential for a successful homing search move or other move-until-trigger for the motor using this channel for its position-loop feedback and flags if the super-accurate hardware position capture function is used. If **Motor[*x*].CaptureMode** is set to 0, 1, or 3

to select hardware trigger (with or without hardware position capture), this variable must be set up properly.

The following settings of **Gate3[*i*].Chan[*j*].CaptCtrl** may be used:

- **Gate3[*i*].Chan[*j*].CaptCtrl** = 0: Immediate or Hall capture
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 1: Capture on Index (CHCn) high
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 2: Capture on Flag high
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 3: Capture on (Index low AND Flag low)
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 4: Immediate or Hall capture
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 5: Capture on Index (CHCn) low
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 6: Capture on Flag high
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 7: Capture on (Index high AND Flag low)
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 8: Immediate or Hall capture
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 9: Capture on Index (CHCn) high
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 10: Capture on Flag low
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 11: Capture on (Index low AND Flag high)
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 12: Immediate or Hall capture
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 13: Capture on Index (CHCn) low
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 14: Capture on Flag low
- **Gate3[*i*].Chan[*j*].CaptCtrl** = 15: Capture on (Index high AND Flag high)

If bits 0 and 1 of this variable are both set to 0 (element value 0, 4, 8, or 12) and **Gate3[*i*].Chan[*j*].GatedIndexSel** for the same channel is 0, the capture trigger will occur immediately if the trigger is armed. If **Gate3[*i*].Chan[*j*].GatedIndexSel** for the same channel is 1, the trigger will occur on the next transition of any of the U, V, or W "Hall" input flags.

The trigger is armed when the position-capture register **Gate3[*i*].Chan[*j*].HomeCapt** is read. This sets the status element **Gate3[*i*].Chan[*j*].PosCapt** to 0, indicating that the trigger is armed, but the next trigger has not occurred. In Power PMAC's move-until-trigger functions, this read is performed automatically at the beginning of the move so the trigger is always armed. After this, as soon as the IC hardware sees a transition that puts the specified input lines are into the specified states, the trigger will occur – it is edge-triggered, not level-triggered (unlike the older DSPGATE1 IC that uses the **Gate1[*i*]** data structure. A transition is required after arming to produce a trigger.

Note

The settings for values of 3 and 15 are reversed compared to **Gate1[*i*].Chan[*j*].CaptCtrl**, as are those for 7 and 11.

Gate3[*i*].Chan[*j*].CaptCtrl constitutes bits 6 – 9 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].CaptFlagChan

Description: IC channel position-capture flag channel select

Range: 0 .. 3

Units: none

Default: (this channel's index number)

Gate3[*i*].Chan[*j*].CaptFlagChan determines which of the four channels for this IC will provide the flag for this channel's hardware position-capture function. **Gate3[*i*].Chan[*j*].CaptFlagSel** determines which flag from the selected channel is used, and **Gate3[*i*].Chan[*j*].CaptCtrl** determines whether a flag is used and which polarity of the flag will cause the trigger.

Gate3[*i*].Chan[*j*].CaptFlagChan can take a value from 0 to 3, specifying the index of the selected channel. Note that the hardware numbering of channels used in the signal descriptions ranges from 1 to 4, so this variable should be set to a value one less than the hardware number of the channel. The default value for this variable is the channel's own index value, so by default, the channel will use its own flags for position capture.

Gate3[*i*].Chan[*j*].CaptFlagChan constitutes bits 12 – 13 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].CaptFlagSel

Description: IC channel position-capture flag select

Range: 0 .. 3

Units: none

Default: 0

Gate3[*i*].Chan[*j*].CaptFlagSel determines which of the four "flag" inputs for the channel selected by **Gate3[*i*].Chan[*j*].CaptFlagChan** will be used for hardware position capture (if one is used) of the channel's encoder counter on a PMAC3-style interface IC.
Gate3[*i*].Chan[*j*].CaptCtrl determines whether a flag is used and which polarity of the flag will cause the trigger. The possible values of **Gate3[*i*].Chan[*j*].CaptFlagSel** and the flag each selects is:

- **Gate3[*i*].Chan[*j*].CaptFlagSel** = 0: HOMEn (Home Flag n)
- **Gate3[*i*].Chan[*j*].CaptFlagSel** = 1: PLIMn (Positive End Limit Flag n)
- **Gate3[*i*].Chan[*j*].CaptFlagSel** = 2: MLIMn (Negative End Limit Flag n)
- **Gate3[*i*].Chan[*j*].CaptFlagSel** = 3: USERn (User Flag n)

Gate3[*i*].Chan[*j*].CaptFlagSel is typically set to 0 for homing search moves in order to use the home flag for the channel. It is commonly set to 3 afterwards to select the User flag if other uses

of the hardware position capture function are desired, such as for probing and registration. If you wish to capture on the PLIMn or MLIMn overtravel limit flags, you probably will want to disable their normal shutdown functions by temporarily setting **Motor[*x*].pLimits** to 0.

Gate3[*i*].Chan[*j*].CaptFlagSel constitutes bits 10 – 11 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].EncCtrl

Description: IC channel encoder decode control

Range: 0 .. 15

Units: none

Default: 7

Gate3[*i*].Chan[*j*].EncCtrl determines how the input signal for this channel's encoder is decoded into counts. As such, this defines the sign and magnitude of a "count". The following settings may be used to decode an input signal.

- **Gate3[*i*].Chan[*j*].EncCtrl** = 0: External pulse and direction CW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 1: x1 quadrature decode CW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 2: x2 quadrature decode CW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 3: x4 quadrature decode CW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 4: External pulse and direction CCW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 5: x1 quadrature decode CCW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 6: x2 quadrature decode CCW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 7: x4 quadrature decode CCW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 8: Internal pulse and direction CW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 9: Pulse up/pulse down CW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 10: (Reserved for future use)
- **Gate3[*i*].Chan[*j*].EncCtrl** = 11: x6 Hall-format decode CW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 12: Internal pulse and direction CCW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 13: Pulse up/pulse down CCW
- **Gate3[*i*].Chan[*j*].EncCtrl** = 14: (Reserved for future use)
- **Gate3[*i*].Chan[*j*].EncCtrl** = 15: x6 Hall-format decode CCW

In any of the quadrature decode modes, the IC is expecting two input waveforms on CHAn and CHBn, each with approximately 50% duty cycle, and approximately one-quarter of a cycle out of phase with each other. "Times-one" (x1) decode provides one count per cycle; x2 provides two counts per cycle; and x4 provides four counts per cycle. The vast majority of users select x4 decode to get maximum resolution.

The "clockwise" (CW) and "counterclockwise" (CCW) options in any decode mode simply control which direction counts up. If you get the wrong direction sense, simply change to the other option (e.g. from 7 to 3 or vice versa).

WARNING

Changing the direction sense of the decode for the feedback encoder of a motor that is operating properly will result in unstable positive feedback and a dangerous runaway condition in the absence of other changes. The output polarity must be changed as well to re-establish polarity match for stable negative feedback.

In the external pulse-and-direction decode modes, the IC is expecting the pulse train on CHAn, and the direction (sign) signal on CHBn. If the signal is unidirectional, the CHBn line can be allowed to pull up to a high state, or it can be hardwired to a high or low state.

In the internal pulse-and-direction decode modes, the IC uses the channel's own pulse-frequency-modulation (PFM) pulse train as input to the decoder. This permits the Power PMAC to create a phantom closed loop when driving an open-loop stepper system. *No jumpers or cables are needed to do this; the connection is entirely within the IC.*

In the pulse-up/pulse-down decode modes, the pulse to create a count in one direction is expected on CHAn, and the pulse to create a count in the other direction is expected on CHBn. Which pulse creates a count in which direction is determined by the CW/CCW selection.

In the "x6" Hall-format decode modes, the IC is expecting three Hall-sensor format inputs on CHAn, CHBn, and CHCn, each with approximately 50% duty cycle, and approximately one-third (120°e) of a cycle out of phase with each other. The decode circuitry will generate one count on each edge of each signal, yielding 6 counts per signal cycle ("x6 decode").

Gate3[*i*].Chan[*j*].EncCtrl constitutes bits 0 – 3 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].Equ1Ena

Description: IC channel compare encoder select

Range: 0 .. 1

Units: Boolean

Default: 0

Gate3[*i*].Chan[*j*].Equ1Ena determines whether the channel's compare circuitry is referenced to the channel's own encoder counter, or to the encoder of the first channel (Chan[0]) on the IC. If **Gate3[*i*].Chan[*j*].Equ1Ena** is set to the default value of 0, the channel's compare circuitry is referenced to the channel's own encoder counter. If **Gate3[*i*].Chan[*j*].Equ1Ena** is set to 1, the channel's compare circuitry is referenced to the first channel's encoder counter.

Of course, the setting of **Gate3[*i*].Chan[0].Equ1Ena** makes no difference in operation.

Referencing multiple compare circuits to a single encoder counter permits functions such as the "windowing" of high-frequency compare outputs to a specific position region.

Gate3[*i*].Chan[*j*].Equ1Ena constitutes bit 5 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].EquOutMask

Description: IC channel compare state output mask

Range: $0 .. $F

Units: Bit field

Default: **Gate3[*i*].Chan[0].EquOutMask**=$1
Gate3[*i*].Chan[1].EquOutMask=$2
Gate3[*i*].Chan[2].EquOutMask=$4
Gate3[*i*].Chan[3].EquOutMask=$8

Gate3[*i*].Chan[*j*].EquOutMask determines which channels' internal position-compare states are used in the generation of this channel's position-compare (EQU) output. Each channel of the IC has an internal compare state based on the history of the comparisons of the encoder counter to the **CompA** and **CompB** register values.

The channel's compare output level can be a logical combination of any of the IC's four internal compare states. Each bit (0 to 3) of **Gate3[*i*].Chan[*j*].EquOutMask** that is set to 1 instructs the IC to include the corresponding channel's internal compare state in a logical OR for this channel's compare output.

Logically combining compare states permits functions such as multi-dimensional compares, or if multiple internal compare states use the same encoder, precise "windowing" of high-frequency auto-incrementing compare outputs to a given zone.

The default settings specify the use of only the channel's own internal compare state for the creation of the channel's compare output level. To use all 4 channels' internal compare states in a channel's compare output, this element would be set to $F (15).

After the logical OR, the output level can be inverted with **Gate3[*i*].Chan[*j*].EquOutPol**, described below.

Gate3[*i*].Chan[*j*].EquOutMask constitutes bits 0 – 3 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].EquOutPol

Description: IC channel compare output polarity

Range: 0 .. 1

Units: Bit field

Default: 0

Gate3[*i*].Chan[*j*].EquOutPol determines whether the channel's compare output level is logically inverted or not. The channel's compare output level is first created as the logical OR of up to four of the IC's individual channel internal compare states. If **Gate3[*i*].Chan[*j*].EquOutPol** is 0, this intermediate state *is not* inverted before being output on the EQU signal for the channel. If **Gate3[*i*].Chan[*j*].EquOutPol** is 1, this intermediate state *is* inverted before being output on the EQU signal for the channel.

The inversion control capability provided by **Gate3[*i*].Chan[*j*].EquOutPol**, combined with the logical OR capability of **Gate3[*i*].Chan[*j*].EquOutMask** and the initial-state setting capability of **Gate3[*i*].Chan[*j*].EquWrite** gives the user the ability to create virtually any desired logical combination of compare states.

If **Gate3[*i*].Chan[*j*].EquOutMask** for the channel is set to 0, meaning that no internal compare states are used in the creation of the channel's compare output level, then **Gate3[*i*].Chan[*j*].EquOutPol** can be used directly to control the output level, enabling the use of the EQU output as a general-purpose software-controlled digital output.

Gate3[*i*].Chan[*j*].EquOutPol constitutes bit 5 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].FlagFilt2Ena

Description: IC channel flag secondary filter enable

Range: 0 .. 1

Units: Boolean

Default: 0

Gate3[*i*].Chan[*j*].FlagFilt2Ena controls whether a secondary digital delay filter is applied to the channel's input flags before their values are latched into the channel's status register **Gate3[*i*].Chan[*j*].Status**, where they can be read by the processor. The secondary filter can provide additional noise immunity and help prevent spurious trips.

All flags (as well as encoder signals) automatically pass through a primary digital delay filter that samples the signal at the encoder sample clock frequency, which typically is set to multiple megahertz. The filter provides a best two-of-three voting on the most recent samples, so a noise spike that is present only in a single sample will not be passed through the filter. However,

because high-frequency encoder signals pass through filters with the same sample rate, the sample rate must be set high, and spurious spikes on the flag inputs may last for multiple sample periods and get through the primary filter.

If **Gate3[*i*].Chan[*j*].FlagFilt2Ena** is set to 1, the flag inputs are subsequently passed through a second digital delay filter, identical in structure to the first filter, but sampling at a frequency set by the IC's filter clock. The frequency of this clock for the IC is set by **Gate3[*i*].FiltClockDiv**, which determines how far it is divided down from the encoder sample clock frequency. If **Gate3[*i*].Chan[*j*].FlagFilt2Ena** is set to 0, this secondary filter is bypassed.

Note that if even if this secondary filter is enabled, the timing of the capture functions of the flags is not altered, as that functionality occurs before the secondary filter.

Gate3[*i*].Chan[*j*].FlagFilt2Ena constitutes bit 17 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].GatedIndexSel

Description: IC channel gated-index capture select

Range: 0 .. 1

Units: none

Default: 0

Gate3[*i*].Chan[*j*].GatedIndexSel controls whether the raw encoder index channel input or a version of the input gated by the AB-quadrature state is used for position capture of the channel's encoder on a PMAC3-style interface IC . It has the following possible settings:

- **Gate3[*i*].Chan[*j*].GatedIndexSel** = 0: Use ungated index for encoder position capture
- **Gate3[*i*].Chan[*j*].GatedIndexSel** = 1: Use index gated by quadrature channels for position capture

When **Gate3[*i*].Chan[*j*].GatedIndexSel** is set to 0, the encoder index channel input (CHCn) is passed directly into the position capture circuitry.

When **Gate3[*i*].Chan[*j*].GatedIndexSel** is set to 1, the encoder index channel input (CHCn) is logically combined with ("gated by") the quadrature signals of Encoder n before going to the position capture circuitry. The intent is to get a "gated index" signal exactly one quadrature state wide. This provides a more accurate and repeatable capture, and makes the use of the capture function to confirm the proper number of counts per revolution very straightforward.

In order for the gated index capture to work reliably, the index pulse must reliably span one, but only one, "high-high" or "low-low" AB quadrature state of the encoder. **Gate3[*i*].Chan[*j*].IndexGateState** allows you to select which of these two possibilities is used.

Note: If **Gate3[*i*].Chan[*j*].GatedIndexSel** is set to 1, but **Gate3[*i*].Chan[*j*].CaptCtrl** bits 0 and 1 are set to 0, so the index is not used in the position capture, then the encoder position is captured on the first edge of any of the U, V, or W flag inputs for the channel. In this case, bits 16, 17, and 18 of the channel status word tell what Hall-state edge caused the capture.

Gate3[*i*].Chan[*j*].GatedIndexSel constitutes bit 14 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].InCtrl

Description: IC channel control word for input signals

Range: 0 .. $FFFFFFFF

Units: Bit field

Default: $00400047

Gate3[*i*].Chan[*j*].InCtrl is the full-word element that comprises the setup elements for input signal interfaces for the channel of the IC. It is comprised of the following partial-word elements:

Component	Bits	Functionality
(*Reserved*)	31 – 23	(*Reserved for future use*)
PackInData	22 – 21	Enables combining ADC data in one word
SerialEncEna	20	Enables serial encoder interface on channel
CountReset	19	Resets encoder counters and timers to zero
AtanEna	18	Enables arctangent calcs from encoder ADCs
FlagFilt2Ena	17	Enables secondary digital delay filter for flags
IndexDemuxEna	16	Enables demux of hall data from index
IndexGateState	15	Selects quadrature state for gating index capture
GatedIndexSel	14	Enables gating of index by quad state for capture
CaptFlagChan	13 – 12	Selects channel of flags for this channel's capture
CaptFlagSel	11 – 10	Selects flag for this channel's capture
CaptCtrl	09 – 06	Select trigger state for this channel's capture
TimerMode	05 – 04	Selects timer mode for channel
EncCtrl	03 – 00	Selects encoder decode method for channel

Those elements shown in bold are saved setup elements and are documented individually.

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Gate3[*i*].InCtrl is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate3[*i*].Chan[*j*].IndexDemuxEna

Description: IC channel index demultiplex enable

Range: 0 .. 1

Units: Boolean

Default: 0

Gate3[*i*].Chan[*j*].IndexDemuxEna controls whether the IC channel "de-multiplexes" the index pulse and the 3 hall-style commutation states from the third channel based on the quadrature state, as with Yaskawa Sigma I incremental encoders. If it is set to 0, this de-multiplexing function is not performed, and the signal on the "C" channel of the encoder is used as the index only. If it is set to 1, the IC breaks out the third-channel signal into four separate values, one for each of the four possible AB-quadrature states. The de-multiplexed hall commutation states can be used to provide power-on phase position and subsequent phase position monitoring.

Note: Immediately after power-up, the Yaskawa encoder automatically cycles its AB outputs forward and back through a full quadrature cycle to ensure that all of the hall commutation states are available to the controller before any movement is started. However, if the encoder is powered up at the same time as the Power PMAC, this will happen before the IC is ready to accept these signals. Bit 19 of the channel's status word **Gate3[*i*].Chan[*j*].Status**, "Invalid De-multiplex", will be set to 1 if the Servo IC has not seen all of these states when it was ready for them. To use this feature, it is recommended that the power to the encoder be provided through a software-controlled relay to ensure that valid readings of all states have been read before using these signals for power-on phasing.

Gate3[*i*].Chan[*j*].IndexDemuxEna constitutes bit 16 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].IndexGateState

Description: IC channel gated-index capture state control

Range: 0 .. 1

Units: none

Default: 0

Gate3[*i*].Chan[*j*].IndexGateState specifies whether the raw index-channel signal fed into the encoder input of the Servo IC is passed through to the position capture signal only on the "high-high" quadrature state (= 0), or only on the "low-low" quadrature state (= 1) if the "gated index" feature is enabled by setting **Gate3[*i*].Chan[*j*].GatedIndexSel** to 1.

Gate3[*i*].Chan[*j*].IndexGateState constitutes bit 15 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].OutCtrl

Description: IC channel control word for output signals

Range: 0 .. $FFFFFFFF

Units: Bit field

Default: $0F800001

Gate3[*i*].Chan[*j*].OutCtrl is the full-word element that comprises the setup elements for output signal interfaces for the channel of the IC. It is comprised of the following partial-word elements:

Component	Bits	Functionality
PwmDeadTime	31 – 24	Switching dead time for PWM outputs
PackOutData	23	Enables combining PWM/DAC data in one word
PwmFreqMult	22 – 20	Multiplication factor for PWM frequency
PfmFormat	19	Selects pulse and direction or quadrature
PfmDirPol	18	Selects output polarity of PFM direction signal
OutputPol	17 – 16	Selects output polarity for command signals
OutputMode	15 – 12	Selects output format for phase command signals
OutFlagD	11	Sets high/low state for channel's D flag output
OutFlagC	10	Sets high/low state for channel's C flag output
OutFlagB	09	Sets high/low state for channel's B flag output
AmpEna	08	Sets high/low state for channel's amplifier enable (A flag) output flag
EquWrite	07 – 06	Forces compare output state
Equ1Ena	05	Selects channel of encoder for this channel's compare
EquOutPol	04	Selects output polarity for this channel's compare
EquOutMask	03 – 00	Selects which channel's compare states ORed into this channel's compare output

Those elements shown in bold are saved setup elements and are documented individually in this chapter. Those not in bold are not saved, and are documented in the chapter on Non-Saved Setup Elements.

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

In C programs, the user must access the full-word element only.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

The saved elements of **Gate3[*i*].Chan[*j*].OutCtrl** are write-protected, so cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Refer to the entry for each partial-word element for a detailed description of that element's function.

Gate3[*i*].Chan[*j*].OutputMode

Description: IC channel phase output mode select

Range: $0 .. $F

Units: Bit field

Default: $0

Gate3[*i*].Chan[*j*].OutputMode determines which output format is used on the command signal line for each phase (A to D) of the IC's channel. It is a 4-bit value, with each bit controlling the output format for the corresponding phase as follows:

Bit #	Bit Value	Phase	Output Format (0)	Output Format (1)
0	1	A	PWM	DAC
1	2	B	PWM	DAC
2	4	C	PWM	DAC
3	8	D	PWM	PFM

For PWM outputs on Phases A, B, and C, required for direct-PWM output to a 3-phase motor, **Gate3[*i*].Chan[*j*].OutputMode** must be set to 0 or 8.

For PWM outputs on all 4 phases, required for direct-PWM control of two independent phases, as for a typical stepper motor, **Gate3[*i*].Chan[*j*].OutputMode** must be set to 0.

For a single DAC output on Phase A, required for an analog velocity or torque-mode amplifier, or for dual DAC outputs on Phases A and B, required for analog "sine-wave" amplifiers when Power PMAC is performing the phase commutation for a brushless motor, **Gate3[*i*].Chan[*j*].OutputMode** must be set to 3, 7, 11, or 15.

For PFM output on Phase D, required for pulse-and-direction control of traditional stepper drives, or MLDT excitation pulsing, **Gate3[*i*].Chan[*j*].OutputMode** must be set to 8 or higher.

Gate3[*i*].Chan[*j*].OutputMode constitutes bits 12 – 15 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].OutputPol

Description: IC channel phase output inversion control

Range: 0 .. 3

Units: Bit field

Default: 0

Gate3[*i*].Chan[*j*].OutputPol determines whether the command signal lines for IC's channel are inverted or not. It is a 2-bit value, with bit 0 (value 1) controlling the inversion of Phases A, B, and C (whether in PWM or DAC mode), and bit 1 controlling the inversion of Phase D (whether in PWM or PFM mode). A 0 in the bit specifies non-inverted output (high-true). A 1 in the bit specifies inverted output (low-true). The variable can take the following possible values:

- **Gate3[*i*].Chan[*j*].OutputPol** = 0: Non-inverted A, B, & C; non-inverted D
- **Gate3[*i*].Chan[*j*].OutputPol** = 1: Inverted A, B, & C; non-inverted D
- **Gate3[*i*].Chan[*j*].OutputPol** = 2: Non-inverted A, B, & C; inverted D
- **Gate3[*i*].Chan[*j*].OutputPol** = 3: Inverted A, B, & C; inverted D

The default non-inverted outputs are high true. For PWM signals, this means that the transistor-on signal is high. Delta Tau PWM-input amplifiers, and most other PWM-input amplifiers, expect this non-inverted output format. For such a 3-phase motor drive, **Gate1[*i*].Chan[*j*].OutputPol** should be set to 0 or 8.

Caution

If the high/low polarity of the PWM signals is wrong for a particular amplifier, what was intended to be deadtime between top and bottom on-states as set by **Gate3[*i*].Chan[*j*].PwmDeadTime** becomes overlap. If the amplifier-input circuitry does not lock this out properly, this causes an effective momentary short circuit between bus power and ground. This would destroy the power transistors very quickly.

For PFM signals on Output D, non-inverted means that the pulse-on signal is high (direction polarity is controlled by **Gate3[*i*].Chan[*j*].PfmDirPol**). During a change of direction, the direction bit will change synchronously with the leading edge of the pulse, which in the non-inverted form is the rising edge. If the drive requires a set-up time on the direction line before the rising edge of the pulse, the pulse output can be inverted so that the rising edge is the trailing edge, and the pulse width (established by **Gate3[*i*].PwmDeadTime**) is the set-up time.

For DAC signals on Outputs A, B, and C, non-inverted means that a 1 value to the DAC is high. DACs used on Delta Tau accessory boards, as well as all other known DACs always expect non-inverted inputs, so the appropriate bits of **Gate3[*i*].Chan[*j*].OuputPol** should always be set to 0 when using DACs on this channel.

WARNING

Changing the high/low polarity of the digital data to the DACs has the effect of inverting the voltage sense of the DACs' analog outputs. This changes the polarity match between output and feedback. If the feedback loop had been stable with negative feedback, this change would create destabilizing positive feedback, resulting in a dangerous runaway condition.

Gate3[*i*].Chan[*j*].OutputPol constitutes bits 16 – 17 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].PackInData

Description: IC channel ADC input "pack" enable

Range: 0 .. 3

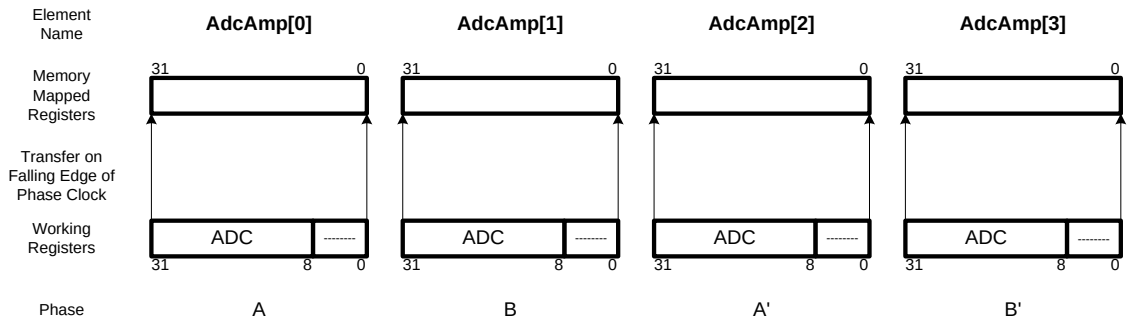
Units: Bit field

Default: 2

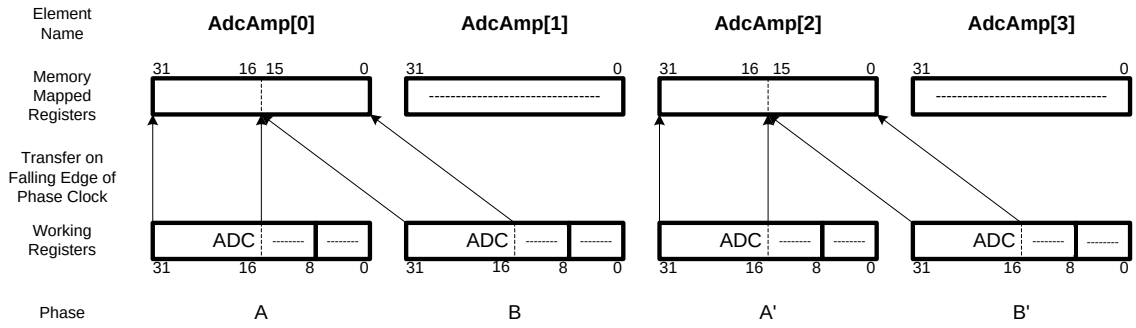
Gate3[*i*].Chan[*j*].PackInData controls whether the two-phase ADC data for the IC channel is "packed" into a single 32-bit register or not. It is a two-bit variable. Bit 0 (value 1) controls whether the channel's "encoder" **AdcEnc[*k*]** values are packed or not; bit 1 (value 2) controls whether the "amplifier" **AdcAmp[*k*]** values are packed or not. A 0 in the bit disables packing; a 1 in the bit enables packing.

If the data is packed, the high 16 bits of **AdcEnc[1]** or **AdcAmp[1]** are copied into the low 16 bits of **AdcEnc[0]** or **AdcAmp[0]**, respectively, and the high 16 bits of **AdcEnc[3]** or **AdcAmp[3]** are copied into the low 16 bits of **AdcEnc[2]** or **AdcAmp[2]**, respectively. In this mode, a single 32-bit register read gives the processor access to both phases of information. If the ADCs have more than 16 bits of data each, the lower bits will be lost in this mode.

The following diagram shows how the data is transferred in unpacked mode for the **AdcAmp** elements. The same arrangement is true for the **AdcEnc** elements if unpacked.



The following diagram shows how the data is transferred in packed mode for the **AdcAmp** elements. The same arrangement is true for the **AdcEnc** elements if packed.



If this channel is used for current feedback when Power PMAC is closing the digital current loop for a motor, packing the data significantly improves the efficiency of the algorithm, so **PackInData** should be set to 2 or 3 in this case. **Motor[*x*].PhaseCtrl** bit 0 (value 1) should be set to 1 to tell the motor algorithm to expect packed data and only do a single read; in this case, the channel's **PackOutData** element should also be set to 1 to pack the command outputs two to a register. If the current-feedback data is not packed, **Motor[*x*].PhaseCtrl** bit 2 (value 4) should instead be set to 1 to tell the motor algorithm not to expect packed data and to do two reads.

No automatic functions of the Power PMAC presently use the **AdcEnc(*k*)** values in packed mode. The encoder conversion table type 4 software interpolation of analog sinusoidal encoders required unpacked data, so bit 0 (value 1) of **PackInData** must be 0 to use this method.

Gate3[*i*].Chan[*j*].PackInData constitutes bits 21 – 22 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].PackOutData

Description: IC channel PWM/DAC "pack" enable

Range: 0 .. 1

Units: Boolean

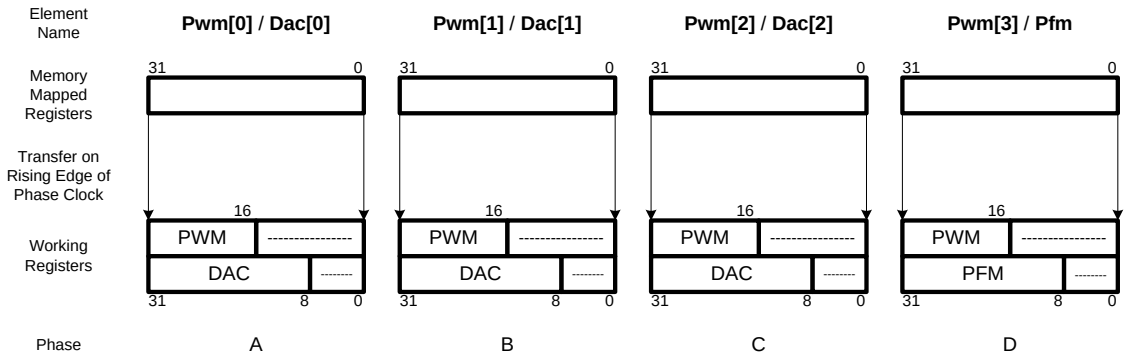
Default: 1

Gate3[*i*].Chan[*j*].PackOutData controls whether the four-phase output (PWM or DAC/PWM) data for the IC channel is "packed" into two 32-bit registers or not. If it is set to 0, the data is not packed, and the four phase-output command values must be written into individual registers. If it is set to 1, the data is packed, and the four phase-output commands are written to the upper and lower halves of two registers.

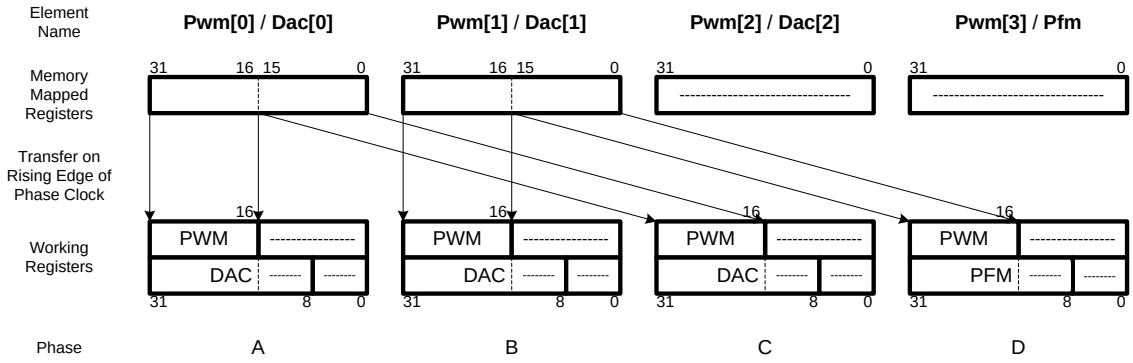
If the data is packed, the high 16 bits of the Phase C command (**Pwm[2]** or **Dac[2]** if not packed) should be written to the lower 16 bits of the 32-bit register for the Phase A command (**Pwm[0]** or **Dac[0]**) and the high 16 bits of the Phase D command (**Pwm[3]** or **Pfm** if not packed) should be

written to the lower 16 bits of the 32-bit register for the Phase B command (**Pwm[1]** or **Dac[1]**). The IC will automatically transfer the values from the low 16 bits of the memory-mapped 32-bit registers for the A and B phases to the high 16 bits of the working registers for the C and D phases, respectively.

The following diagram shows how the output data is transferred in unpacked mode.



The following diagram shows how the output data is transferred in packed mode.



If this channel is used for multiple phase outputs when Power PMAC is performing commutation and digital current-loop calculations for a motor, packing the data significantly improves the efficiency of the algorithm, so **PackOutData** should be set to 1 in this case. **Motor[*x*].PhaseCtrl** bit 0 (value 1) should be set to 1 to tell the motor algorithm to expect the packed data format and do packed writes; in this case, the channel's **PackInData** element should also be set to 1 to pack the current-feedback inputs two to a register as well. If the input and output data are not packed, **Motor[*x*].PhaseCtrl** bit 2 (value 4) should instead be set to 1 to tell the motor algorithm not to expect packed data and to do a separate read/write operation for each input and output register.

If this channel is used for multiple phase outputs when Power PMAC is performing commutation calculations for the motor, but not closing a digital current loop, setting **PackOutData** to 1 and bit 0 of **Motor[*x*].PhaseCtrl** to 1 will still improve the efficiency of the algorithm. However, in this "sine-wave output" mode, sometimes the output data will have more than 16 bits of resolution. (The ACC-24E3's analog amplifier interface board is available with 18-bit DACs as well as the standard 16-bit DACs.) In packed mode, no more than 16 bits of resolution can be used, so if the outputs have greater than 16-bit resolution and full use of this resolution is desired, **PackOutData** should be set to 0, and bit 2 of **Motor[*x*].PhaseCtrl** should be set to 1 instead.

Gate3[*i*].Chan[*j*].PackOutData constitutes bit 23 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].Pfm

Description: IC channel pulse-frequency modulation register command value

Range: -2,147,483,392 .. 2,147,483,391 ($-2^{31} + 256$ .. $2^{31} - 257$)

Units: Signed 32-bit PFM units

Default: 0

Gate3[*i*].Chan[*j*].Pfm specifies the command value for the pulse-frequency modulation register for the specified IC and channel. Each channel has a single PFM register that generates a pulse-and-direction signal pair with a frequency proportional to this command value. The PFM outputs are commonly used to command traditional stepper-motor drives, to excite MLDT sensors, and to modulate lasers and other outputs. Internally the signals can be fed back into the channel's encoder counters and timers for simulated feedback loops, precise trigger timing (for timer-assisted software position capture), and other frequency/timing uses.

Each PFMCLK clock cycle, the value in the high 24 bits of **Gate3[*i*].Chan[*j*].Pfm** is added into a 24-bit accumulator (the low 8 bits are not used). Every time the accumulator rolls over, an output pulse is generated. If it rolls over in the positive direction, the direction output signal is set for "plus"; if it rolls over in the negative direction, the direction output is set for "minus". In this way, the generated pulse frequency is proportional to the value of the element.

The frequency of the PFMCLK signal is set by saved setup element **Gate3[*i*].PfmClockDiv**. The default frequency is 3.125 MHz. The resulting output frequency for a given PFMCLK frequency and 32-bit command value can be computed as:

$$f_{out} = \frac{Gate3[i].Chan[j].Pfm}{4{,}294{,}483{,}648} * f_{PFMCLK}$$

The resulting output frequency for a given PFMCLK frequency and 16-bit servo output value (in the high 16 bits) can be computed as:

$$f_{out} = \frac{ServoOut}{65{,}536} * f_{PFMCLK}$$

If the phase is configured for PFM mode output by setting bit 3 (value 8) of **Gate3[*i*].Chan[*j*].OutputMode** to 1, then the pulse and direction signals are output on the channel's Phase D output pins instead of the PWM top and bottom signal pins generated from command value **Gate3[*i*].Chan[*j*].Pwm[3]**. Note that the internal pulse generation and the ability to feed the pulse signals into the channel's encoder counters or timers occur regardless of the output mode.

If saved setup element **Gate3[*i*].Chan[*j*].PfmFormat** is set to the default value of 0, the signal is output as a pulse and direction pair. The width of each pulse, in PFMCLK cycles, is determined by the saved setup element **Gate3[*i*].Chan[*j*].PfmWidth**. If the next pulse is generated before the end of the previous pulse, the output will simply stay "on", effectively saturating the output. The polarity of the direction output for positive and negative command values is determined by saved setup element **Gate3[*i*].Chan[*j*].PfmDirPol**.

If **Gate3[*i*].Chan[*j*].PfmFormat** is set to 1, the signal is output as a quadrature A and B-channel pair. Each pulse creates an edge of one of the channels. The direction sense of the quadrature output for positive and negative command values is determined by saved setup element **Gate3[*i*].Chan[*j*].PfmDirPol**.

If saved setup element **Gate3[*i*].Chan[*j*].TimerMode** is set to 1 for "MLDT pulse-echo timing" mode, a pulse is automatically output from the on the falling edge of the servo clock each cycle, regardless of the value of **Gate[3].Chan[*j*].Pfm**.

If saved setup element **Gate3[*i*].Chan[*j*].EncCtrl** is set to 8 or 12, the internal pulse and direction signals drive the channel's encoder counter (instead of external signals), with each pulse generating one count. If **Gate3[*i*].Chan[*j*].TimerMode** is set to 3, the internal pulse and direction signals drive the channel's timer circuit as a counter, leaving the encoder counter available for external signals (although without timer-based extension).

Gate3[*i*].Chan[*j*].Pfm shares a register with the pulse-width modulation command value **Gate3[*i*].Chan[*j*].Pwm[3]**. If the register is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

If saved setup element **Gate3[*i*].Chan[*j*].PackOutData** is set to 1, 16-bit values for the 4 phases of the channel are "packed" into 2 of these 32-bit elements, so the processor can command all 4 phases with just 2 write operations. In this case, the high 16 bits of the PFM command value are written into the low 16 bits of the **Pwm[1]** element for the channel. The low 8 bits of the 24-bit PFM command value are always 0 in this mode. The value in the **Pfm** element for the channel is not used in this mode.

Unlike other ASIC command output elements, the value of this element can be saved to non-volatile flash memory so it is automatically restored on power-up/reset. This allows a fixed frequency to be generated in an application without the need for an explicit command to be executed after every power-up/reset. This is particularly valuable for its use in the timer-assisted software position capture function.

In the Script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

Gate3[*i*].Chan[*j*].PfmDirPol

Description: IC channel PFM direction polarity control

Range: 0 .. 1

Units: Boolean

Default: 0

Gate3[*i*].Chan[*j*].PfmDirPol determines the polarity of the PFM direction output signal format for the IC channel. It is only active if **Gate3[*i*].Chan[*j*].OutputMode** is set to a value of 8 or greater to specify PFM output on Phase D.

If **Gate3[*i*].Chan[*j*].PfmDirPol** is set to the default value of 0, the output is non-inverted. In pulse-and-direction mode (set by **Gate3[*i*].Chan[*j*].PfmFormat** = 0) a positive direction provides a low output.

If **Gate3[*i*].Chan[*j*].PfmDirPol** is set to 1, the output is inverted. In pulse-and-direction mode (set by **Gate3[*i*].Chan[*j*].PfmFormat** = 0) a positive direction provides a high output.

In quadrature mode (set by **Gate3[*i*].Chan[*j*].PfmOutFormat** = 0), **Gate3[*i*].Chan[*j*].PfmDirPol** controls the inversion of the second quadrature phase (output on the "direction" line), effectively controlling the direction sense of the quadrature signal.

Gate3[*i*].Chan[*j*].PfmDirPol constitutes bit 18 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].PfmFormat

Description: IC channel PFM output format control

Range: 0 .. 1

Units: Boolean

Default: 0

Gate3[*i*].Chan[*j*].PfmFormat determines the signal output format of the PFM circuit on Phase D of the IC's channel. It is only active if **Gate3[*i*].Chan[*j*].OutputMode** is set to a value of 8 or greater to specify PFM output on Phase D.

If **Gate3[*i*].Chan[*j*].PfmFormat** is set to the default value of 0, the output is in pulse-and-direction mode. This mode is useful for control of traditional stepper drives and the excitation of MLDT sensors.

If **Gate3[*i*].Chan[*j*].PfmFormat** is set to 1, the output is in quadrature mode. This mode is useful for creating a synthesized incremental encoder signal.

(In some pre-release versions of the Power PMAC firmware, this element was named **Gate3[*i*].Chan[*j*].PfmOutFormat**.)

Gate3[*i*].Chan[*j*].PfmFormat constitutes bit 19 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the

proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].PfmWidth

Description: Channel pulse-frequency-modulation output pulse width

Range: 0 .. 4095

Units: PFM_CLK cycles

Default: 15

Gate3[*i*].Chan[*j*].PfmWidth specifies the duration of each pulse output in the channel's pulse-frequency modulation output. This is expressed in units of cycles of the PFM_CLK signal, whose frequency is determined by the saved setup element **Gate3[*i*].PfmClockDiv** for the IC.

The equation for pulse width as a function of **Gate3[*i*].Chan[*j*].PfmWidth** and PFM_CLK frequency is:

$$PFM\_Pulse\_Width\ (\mu sec) = \frac{Pfm\,Width}{PFM\_CLK\_Freq\ (MHz)}$$

If **PfmWidth** is set to 0, no pulses are created.

The equation for **Gate3[*i*].Chan[*j*].PfmWidth** as a function of desired PFM pulse width and PFM_CLK frequency is:

$$Pfm\,Width = PFM\_CLK\_Freq(MHz) * PFM\_Pulse\_Width(\mu sec)$$

Example

The PFM_CLK frequency for the IC is 25 MHz. A 600-nanosecond (0.6 µsec) pulse width is desired. The value can be computed as:

Gate3[*i*].Chan[*j*].PfmWidth = 25 cyc/µsec * 0.6 µsec = 15

Gate3[*i*].Chan[*j*].PwmDeadTime

Description: IC channel PWM deadtime control

Range: 0 .. 255

Units: 16 * PWM_CLK cycles (53.33 nsec)

Default: 15 (800 nsec)

Gate3[*i*].Chan[*j*].PwmDeadTime determines the duration of the "dead time" period between the turn-off of one of the top-and-bottom pair for each phase and the turn-on of the other for the IC's

channel. When the PWM signals are used for the actual on/off control of the power transistors in "direct PWM" control, this delay is essential because full turn-off takes some time, and an immediate turn-on of the other transistor in the pair would cause a momentary short circuit.

Gate3[*i*].Chan[*j*].PwmDeadTime is expressed in units of 16 PWM_CLK cycles. The frequency of the PWM_CLK signal is fixed at 300 MHz, so one cycle is 3.33 nanoseconds, and 16 cycles is 53.3 nanoseconds. The equation for this parameter as a function of the desired deadtime is:

$$Gate3[i].Chan[j].PwmDeadTime = \frac{DesiredDeadTime\,(\mu sec)}{0.0533\mu sec}$$

Note

Most direct-PWM drives enforce a minimum deadtime period as a safety feature. However, they do so with a lower-frequency clock that is asynchronous to this IC's PWM_CLK signal. This means that relying on the drive's minimum deadtime setting (which occurs when this element specifies a smaller deadtime) decreases the effective command resolution and even can introduce a beat frequency into the output.

Gate3[*i*].Chan[*j*].PwmDeadTime constitutes bits 24 – 31 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].PwmFreqMult

Description: IC channel PWM frequency multiplication factor

Range: 0 .. 7

Units: multiplication factor

Default: 0

Gate3[*i*].Chan[*j*].PwmFreqMult determines how the PWM output frequency for the IC's channel is related to the IC's internal phase clock frequency. The equation for this relationship is:

$$f_{PWM} = \frac{PwmFreqMult + 1}{2} f_{IntPhase}$$

Thus, the possible values of 0 through 7 for **Gate3[*i*].Chan[*j*].PwmFreqMult** can set PWM frequency values for the channel of 0.5 through 4 times the internal phase clock frequency.

Gate3[*i*].Chan[*j*].PwmFreqMult constitutes bits 20 – 22 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].SerialEncCmd

Description: IC channel control word for serial encoder commands

Range: 0 .. $FFFFFFFF

Units: Bit field

Default: $0

Gate3[*i*].Chan[*j*].SerialEncCmd is the full-word element that comprises the command information for the serial encoder interface of the channel on the IC. Note that the specific protocol, trigger timing, and clock frequency are determined by the multi-channel element **Gate3[*i*].SerialEncCtrl**.

Gate3[*i*].Chan[*j*].SerialEncCmd is comprised of the following components. Note that the components cannot be separately accessed as individual elements.

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

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.

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. If the interface circuitry is used for a digital quadrature encoder (**Gate3[*i*].Chan[*j*].SerialEncEna** = 0), this bit controls the direction sense of the "times-4" quadrature decode of the signal.

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. If the interface circuitry is used for a digital quadrature encoder (**Gate3[*i*].Chan[*j*].SerialEncEna** = 0), setting this bit to 1 clears the count value that can be read in status element **Gate3[*i*].Chan[*j*].SerialEncDataA** to 0.

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 12 – 63.

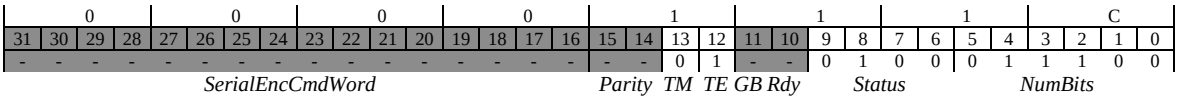
More details are given in the Hardware Reference Manual for the particular interface device (e.g. ACC-24E3) used to process the serial encoder. Below are common settings for each supported protocol.

SPI Protocol

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for an SPI encoder.

SerialEncCmdWord:	= 0	// No command word supported for SPI protocol
SerialEncParity:	= 0	// No parity check supported for SPI protocol
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray-to-binary conversion supported
SerialEncDataReady:	= 0	// Read-only status bit
SerialEncStatusBits:	= ??	// Encoder-specific number of status bits returned
SerialEncNumBits:	= ??	// Encoder-specific number of position bits returned

For example, for an SPI encoder with 28 position bits and 4 status bits, **Gate3[*i*].Chan[*j*].SerialEncCmd** would be set to $0000111C. (It may report back as $0000151C if the data-ready status bit is set.)



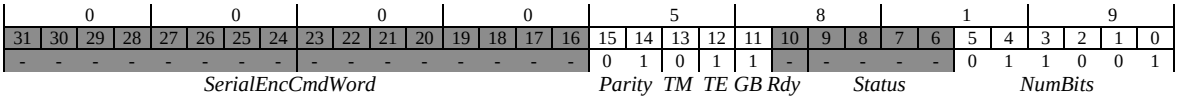
SSI Protocol

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for an SSI encoder.

SerialEncCmdWord:	= 0	// No command word supported for SSI protocol
SerialEncParity:	= ??	// Encoder-specific parity check
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= ??	// Encoder-specific data format
SerialEncDataReady:	= 0	// Read-only status bit

SerialEncStatusBits: = 0 // No status bits supported for SSI protocol
SerialEncNumBits: = ?? // Encoder-specific number of position bits returned

For example, for an SSI encoder with 25 position bits in Gray-code format with odd parity, **Gate3[*i*].Chan[*j*].SerialEncCmd** would be set to $00005819. (It may report back as $0005C19 if the data-ready status bit is set.)



EnDat2.1/2.2 Protocol

The DSPGATE3 IC EnDat interface supports two 6-bit command codes to the encoder: 000111 ($07) for reporting position, and 101010 ($2A) for resetting the encoder. These 6 bits fit at the low end of the 16-bit *SerialEncCmdWord* command field of **Gate3[*i*].Chan[*j*].SerialEncCmd**. These are command codes that can be used either by EnDat2.1 or EnDat2.1 encoders.

Note

By the EnDat standard, EnDat2.2 encoders should be able to accept and process EnDat2.1 command codes as well. However, not all encoders sold as meeting the EnDat2.2 standard can do this.

The most significant bit of *SerialEncCmdWord*, which is bit 31 of the full-word element, is the *StartDelayComp* control bit. Setting this bit to 1 starts a delay identification and compensation cycle that measures the propagation delay between the encoder and the controller. The delay is measured three times, and the average of these delay values is used in the compensation. When these calculations are done, the *StartDelayComp* bit is automatically cleared. This delay identification operation must be performed after every power-up cycle. Delay compensation permits high bit transmission rates over very long cables.

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for position reporting from an EnDat encoder.

SerialEncCmdWord: = 7 // Command word for encoder to report position
SerialEncParity: = 0 // No parity check supported for EnDat protocol
SerialEncTrigMode: = 0 // Continuous triggering (EnDat2.2)
SerialEncTrigEna: = 1 // Enable triggering
SerialEncGtoB: = 0 // No Gray code supported for EnDat protocol
SerialEncDataReady: = 0 // Read-only status bit
SerialEncStatusBits: = 0 // No status bits supported for EnDat protocol
SerialEncNumBits: = ?? // Encoder-specific number of position bits returned

For example, for an EnDat2.2 encoder with 37 position bits, **Gate3[*i*].Chan[*j*].SerialEncCmd** would be set to $00071025 for continuous position reporting. (It may report back as $00071425 if the data-ready status bit is set.)

	0							0							0							7							1							0							2							5						
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	-	-	0	1	-	-	-	-	-	-	1	0	0	1	0	1
DC								*SerialEncCmdWord*										*Parity*	*TM*	*TE*	*GB*	*Rdy*				*Status*						*NumBits*																						

To perform the delay identification and compensation cycle on this encoder, set **Gate3[*i*].Chan[*j*].SerialEncCmd** to $80071025, then wait for the MSB to clear.

	8							0							0							7							1							0							2							5						
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
1	-	-	-	-	-	-	-	-	-	0	0	0	1	1	1	-	-	0	1	-	-	-	-	-	-	1	0	0	1	0	1
DC								*SerialEncCmdWord*										*Parity*	*TM*	*TE*	*GB*	*Rdy*				*Status*						*NumBits*																						

This same encoder can be reset with a command word value of 42 ($2A) sent in one-shot mode with **Gate3[*i*].Chan[*j*].SerialEncCmd** set to $002A3025.

	0							0							2							A							3							0							2							5						
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	-	-	-	-	-	-	-	-	-	1	0	1	0	1	0	-	-	1	1	-	-	-	-	-	-	1	0	0	1	0	1
DC								*SerialEncCmdWord*										*Parity*	*TM*	*TE*	*GB*	*Rdy*				*Status*						*NumBits*																						

For an EnDat2.1 encoder with 24 position bits, **Gate3[*i*].Chan[*j*].SerialEncCmd** would be set to $00073018 for one-shot position reporting (at power-up).

	0							0							0							7							3							0							1							8						
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	-	-	-	-	-	-	0	1	1	0	0	0
DC								*SerialEncCmdWord*										*Parity*	*TM*	*TE*	*GB*	*Rdy*				*Status*						*NumBits*																						

Hiperface Protocol

The DSPGATE3 IC Hiperface interface supports three 8-bit command codes to the encoder: $42 for reporting position, $50 for reporting status, and $53 for resetting the encoder. These 8 bits fit at the low end of the 16-bit *SerialEncCmdWord* command field of **Gate3[*i*].Chan[*j*].SerialEncCmd**.

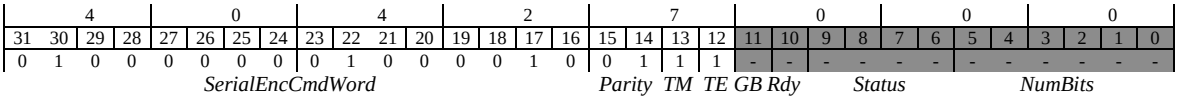
The high 8 bits of *SerialEncCmdWord* contain the address of the encoder in the interface. The Hiperface protocol permits up to 8 separate encoders to be "daisy-chained" on a single multi-drop interface. While this can be done using an ACC-24E3, it is expected that each channel of the ACC-24E3 will be connected to a separate individual encoder, simplifying the wiring. In this configuration, this address field can either match the encoder's address value (+ $40), or it can be set to $FF (broadcast mode).

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for position reporting from an Hiperface encoder.

```
SerialEncCmdWord:    = $4042    // Encoder at address 0 to report position
SerialEncParity:     = ??       // Encoder-specific parity check
SerialEncTrigMode:   = 1        // One-shot triggering
SerialEncTrigEna:    = 1        // Enable triggering (cleared after one-shot)
SerialEncGtoB:       = 0        // No Gray code supported for Hiperface protocol
SerialEncDataReady:  = 0        // Read-only status bit
SerialEncStatusBits: = 0        // No status bits supported for Hiperface protocol
```

SerialEncNumBits: = ?? // Encoder-specific number of position bits returned

For example, for a Hiperface encoder at user address 0 with odd parity, **Gate3[*i*].Chan[*j*].SerialEncCmd** would be set to $40427000 for one-shot position reporting. (It may report back as $40426400 when the trigger-enable bit is cleared and the data-ready status bit is set.)



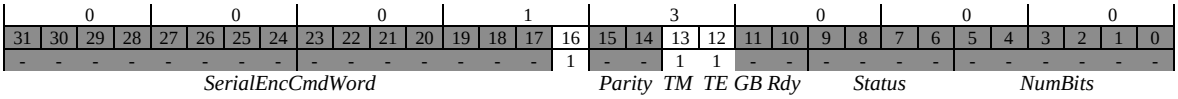
Yaskawa Sigma I Protocol

Yaskawa no longer produces Sigma I absolute encoders. However, newer generations of Yaskawa Sigma servo drives synthesize the Yaskawa Sigma I protocol for return to the controller even when using newer Sigma II, III, and V encoders.

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for a Yaskawa Sigma I encoder.

SerialEncCmdWord: = 1 // Set bit to strobe encoder
SerialEncParity: = 0 // Fixed parity check for Yaskawa protocol
SerialEncTrigMode: = 1 // One-shot triggering
SerialEncTrigEna: = 1 // Enable triggering (cleared on one-shot completion)
SerialEncGtoB: = 0 // No Gray code supported for Yaskawa protocol
SerialEncDataReady: = 0 // Read-only status bit
SerialEncStatusBits: = 0 // No status bits supported for Yaskawa protocol
SerialEncNumBits: = 0 // Fixed number of position bits returned

Gate3[*i*].Chan[*j*].SerialEncCmd would be set to $00013000 for continuous position reporting. (It may report back as $00002400 if the trigger-enable bit is cleared at the end of the one-shot and the ready status bit is set.)

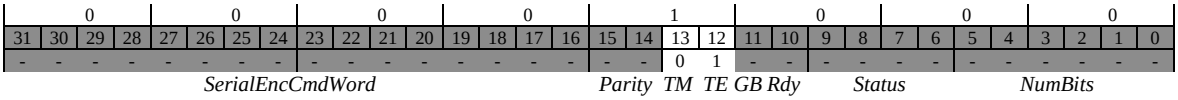


Yaskawa Sigma II/III/V Protocol

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for a Yaskawa Sigma II/III/V encoder.

SerialEncCmdWord: = 0 // No command word for position reporting in Yaskawa
SerialEncParity: = 0 // No parity check supported for Yaskawa protocol
SerialEncTrigMode: = 0 // Continuous triggering
SerialEncTrigEna: = 1 // Enable triggering
SerialEncGtoB: = 0 // No Gray code supported for Yaskawa protocol
SerialEncDataReady: = 0 // Read-only status bit
SerialEncStatusBits: = 0 // No status bits supported for Yaskawa protocol
SerialEncNumBits: = 0 // Fixed number of position bits returned

Gate3[*i*].Chan[*j*].SerialEncCmd would be set to $00001000 for continuous position reporting. (It may report back as $00001400 if the ready status bit is set.)

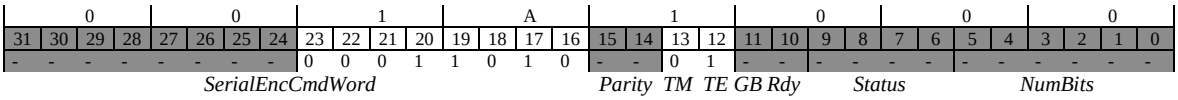


Tamagawa FA-Coder Protocol

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for a Tamagawa FA-Coder encoder.

SerialEncCmdWord:	= $1A	// Command word for position reporting in Tamagawa
SerialEncParity:	= 0	// No parity check supported for Tamagawa protocol
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray code supported for Tamagawa protocol
SerialEncDataReady:	= 0	// Read-only status bit
SerialEncStatusBits:	= 0	// No status bits supported for Tamagawa protocol
SerialEncNumBits:	= 0	// Fixed number of position bits returned

Gate3[*i*].Chan[*j*].SerialEncCmd would be set to $001A1000 for continuous position reporting. (It may report back as $001A1400 if the ready status bit is set.)



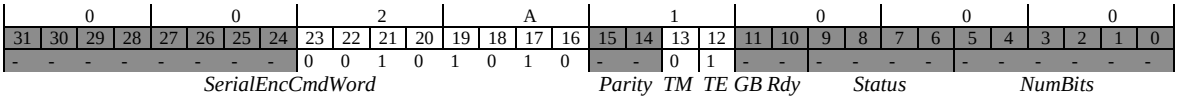
If the *SerialEncCmdWord* component is set to $BA, $C2, or $62, the multi-turn position value in the encoder is reset to 0. This should be done in "one-shot" mode, making the element equal to $00BA3000, $00C23000, or $00623000, respectively. When the reset operation is done, the component should report as $00BA2000, $00C22000, or $00622000, respectively.

Panasonic Protocol

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for a Panasonic serial encoder.

SerialEncCmdWord:	= $2A	// Command word for multi-turn position in Panasonic
SerialEncParity:	= 0	// No parity check supported for Panasonic protocol
SerialEncTrigMode:	= 0	// Continuous triggering
SerialEncTrigEna:	= 1	// Enable triggering
SerialEncGtoB:	= 0	// No Gray code supported for Panasonic protocol
SerialEncDataReady:	= 0	// Read-only status bit
SerialEncStatusBits:	= 0	// No status bits supported for Panasonic protocol
SerialEncNumBits:	= 0	// Fixed number of position bits returned

Gate3[*i*].Chan[*j*].SerialEncCmd would be set to $002A1000 for continuous multi-turn position reporting. (It may report back as $002A1400 if the data-ready status bit is set.)



If the *SerialEncCmdWord* component is set to $52 for single-turn position reporting with alarm code, **Gate3[*i*].Chan[*j*].SerialEncCmd** would be set to $00521000. (It may report back as $00521400 if the data-ready status bit is set.) In this case, the encoder ID value is reported where multi-turn position is normally reported.

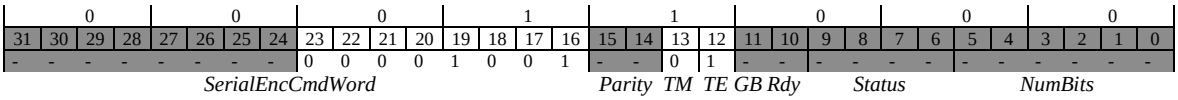
If the *SerialEncCmdWord* component is set to $4A, $7A, $DA, or $F2, the multi-turn position value in the encoder is reset to 0. This should be done in "one-shot" mode, making the element equal to $004A3000, $007A3000, $00DA3000, or $00F23000, respectively. When the reset operation is done, the component should report as $004A2000, $007A2000, $00DA2000, or $00F22000, respectively.

Mitutoyo Protocol

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for a Mitutoyo serial encoder.

```
SerialEncCmdWord:   = $01   // Command word for position reporting in Mitutoyo
SerialEncParity:    = 0     // No parity check supported for Panasonic protocol
SerialEncTrigMode:  = 0     // Continuous triggering
SerialEncTrigEna:   = 1     // Enable triggering
SerialEncGtoB:      = 0     // No Gray code supported for Mitutoyo protocol
SerialEncDataReady: = 0     // Read-only status bit
SerialEncStatusBits: = 0    // No status bits supported for Mitutoyo protocol
SerialEncNumBits:   = 0     // Fixed number of position bits returned
```

Gate3[*i*].Chan[*j*].SerialEncCmd would be set to $00011000 for continuous position reporting. (It may report back as $00011400 if the data-ready status bit is set.)



If the ***SerialEncCmdWord*** component is set to $89, the multi-turn position value in the encoder is reset to 0 (after 8 cycles). If the ***SerialEncCmdWord*** component is set to $9D, the encoder ID value is reported in bits 8 – 15 of **SerialEncDataB**. If the ***SerialEncCmdWord*** component is set to $85, absolute position is reported, just as if the value were $01.

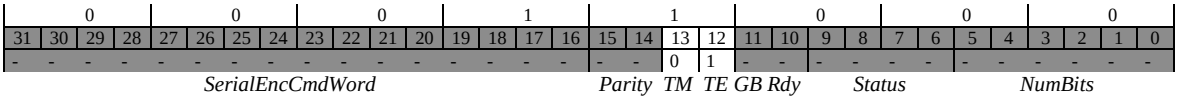
Kawasaki Protocol

The following list shows typical settings of **Gate3[*i*].Chan[*j*].SerialEncCmd** for a Kawasaki serial encoder.

```
SerialEncCmdWord:   = $00   // No command word in Kawasaki protocol
SerialEncParity:    = 0     // No parity check supported for Kawasaki protocol
SerialEncTrigMode:  = 0     // Continuous triggering
SerialEncTrigEna:   = 1     // Enable triggering
```

SerialEncGtoB: = 0 // No Gray code supported for Kawasaki protocol
SerialEncDataReady: = 0 // Read-only status bit
SerialEncStatusBits: = 0 // No status bits supported for Kawasaki protocol
SerialEncNumBits: = 0 // Fixed number of position bits returned

Gate3[*i*].Chan[*j*].SerialEncCmd would be set to $00001000 for continuous position reporting.



Gate3[*i*].Chan[*j*].SerialEncEna

Description: IC channel serial encoder enable

Range: 0 .. 1

Units: Boolean

Default: 0

Gate3[*i*].Chan[*j*].SerialEncEna controls whether the serial encoder interface for the IC's channel is enabled. If it is set to 1, the interface is enabled and SENC_MODE*n* for the channel is set high (which can enable the serial encoder driver and receiver circuits in many products). If it is set to 0, the serial interface is disabled and the output pin SENC_MODE*n* for the channel is held low. However, in this case, the serial clock and data pins can be used to accept digital quadrature input from an incremental encoder, with the counter position available in status element **Gate3[*i*].Chan[*j*].SerialEncDataA**.

Gate3[*i*].Chan[*j*].SerialEncEna constitutes bit 20 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

Gate3[*i*].Chan[*j*].TimerMode

Description: IC Channel encoder timer control

Range: 0 .. 3

Units: none

Default: 0

Gate3[*i*].Chan[*j*].TimerMode controls how the channel's encoder timers are used, and what information appears in the channel's timer registers. The following settings may be used:

- **Gate3[*i*].Chan[*j*].TimerMode** = 0: Hardware 1/T
- **Gate3[*i*].Chan[*j*].TimerMode** = 1: MLDT pulse echo timing
- **Gate3[*i*].Chan[*j*].TimerMode** = 2: Trigger input timing

- **Gate3[*i*].Chan[*j*].TimerMode** = 3: PFM pulse counting

Gate3[*i*].Chan[*j*].TimerMode = 0: In "hardware 1/T" mode, the channel's timers are used to measure the time between the last two decoded counts (regardless of the decode mode), and the time since the last count, both measured in encoder SCLK cycles. Each SCLK cycle, the ratio of these values is used to compute the estimated fractional-count value, and this value, combined with the whole-count value, can be latched at any time by the servo interrupt, phase interrupt, or the specified trigger condition, giving all of these tasks "sub-count" resolution. It is also available to be checked against the "position compare" A and B output registers, giving that function sub-count resolution as well.

In this mode, the channel's 32-bit **PhaseCapt**, **ServoCapt**, and **HomeCapt** registers all contain 8 bits of fractional count value computed by the 1/T circuitry in their LSByte. The channel's **TimerA** and **TimerB** registers both contain 12 bits of fractional count value computed by the 1/T circuitry in their least significant bits, with **TimerA** latched on the servo interrupt, and **TimerB** latched on the selected capture trigger.

Gate3[*i*].Chan[*j*].TimerMode = 1: In "MLDT pulse-echo timing" mode, the channel's timers are used to measure the time between an automatically generated output on PULSEn (PWM_BOT_Dn) to a magnetostrictive linear displacement transducer (MLDT) and the receipt of the "echo pulse" on CHAn. This time is proportional to the distance of the moving sensor from the base.

The timers are set to zero on an output pulse from the PFM circuit, which is automatically synchronized to the servo cycle. They then count up at a 600 MHz rate. In this mode, the channel's **TimerA** register contains the timer value latched on the receipt of the "echo" count pulse, representing the absolute position of the sensor. The channel's **TimerB** register contains the timer value latched when the next output pulse is sent and the timer is reset to 0. This is mainly for reference purposes. In this mode, the low 8 bits of the channel's 32-bit **PhaseCapt**, **ServoCapt**, and **HomeCapt** registers all contain a value of $80, providing an effective fractional count value of ½, because 1/T sub-count extension is disabled (unless **AtanEna** for the channel is set to 1, in which case the arctangent sub-count extension is used).

Gate3[*i*].Chan[*j*].TimerMode = 2: In "trigger input timing" mode, the channel's timers are used to measure the time since the beginning of the servo cycle to enable effective "hardware capture" functionality for sensors that do not utilize the channel's counter circuitry.

The timers are set to zero at the beginning of each servo cycle, and then count up at the SCLK frequency. If the specified capture trigger condition occurs during the servo cycle, the first timer value is latched into the **TimerA** register. The value of the second timer at the beginning of the next servo is latched into the **TimerB** register. The ratio of these timers can then be used to calculate the moment within the servo cycle at which the trigger occurred. This can then be used to interpolate the position at which the trigger occurred. In this mode, the low 8 bits of the channel's 32-bit **PhaseCapt**, **ServoCapt**, and **HomeCapt** registers all contain a value of $80, providing an effective fractional count value of ½, because 1/T sub-count extension is disabled (unless **AtanEna** for the channel is set to 1, in which case the arctangent sub-count extension is used).

Note

The setting of **Gate3[*i*].Chan[*j*].TimerMode** = 2 is not used for Power PMAC's "timer-assisted capture mode" (**Motor[*x*].CaptureMode** = 2). Another similar technique for determining the time at which the trigger occurred is used instead.

Gate3[*i*].Chan[*j*].TimerMode = 3: In PFM pulse-counting mode, the channel's timers are used to count the pulses generated by the channel's PFM output. The connection is made internally in the ASIC; no external wiring is required. This mode permits the channel's PFM output to command a pulse-and-direction stepper drive, let Power PMAC close a simulated loop based on the timer register value, and still use a "confirmation encoder", all on a single interface channel.

In this mode, the channel's **TimerA** register contains the pulse-count value latched on the servo interrupt. It can be used to close a simulated servo loop. The channel's **TimerB** register contains the pulse-count value latched on the selected trigger. In this mode, the low 8 bits of the channel's 32-bit **PhaseCapt**, **ServoCapt**, and **HomeCapt** registers all contain a value of $80, providing an effective fractional count value of ½, because 1/T sub-count extension is disabled (unless **AtanEna** for the channel is set to 1, in which case the arctangent sub-count extension is used).

Gate3[*i*].Chan[*j*].TimerMode constitutes bits 4 – 5 of the full-word element **Gate3[*i*].Chan[*j*].InCtrl**. It is write-protected, so it cannot be changed without first writing the proper key value into **Gate3[*i*].WpKey** (which happens automatically in the script environment if the proper key value is in **Sys.WpKey**).

GateIo[*i*]. Saved Data Structure Elements

The saved setup elements in the **GateIo[*i*].Init** sub-structure are used to initialize the setup registers of the IOGATE digital I/O ASICs. These elements do not actually reside in the IOGATE ICs; rather, they are memory registers whose contents are copied to the IC setup registers during Power PMAC's power-on/reset process. When the Power PMAC is re-initialized using the **$$$***** command, these elements are set to default values appropriate for each auto-detected accessory that has an IOGATE IC.

During the standard power-on/reset process, the hardware setup registers in the IOGATE ICs first receive the factory default values, and then receive the saved values from these memory setup elements. Note that these memory setup elements are only used during the power-on/reset process, so to cause a change in how an IOGATE IC is configured, the value of the memory setup element must be changed, the configuration must be copied to flash memory with a **save** command, and the Power PMAC must be reset.

Note that these structures can also be represented by their alias names – **Acc11E[*i*]**, **Acc14E[*i*]**, **Acc65E[*i*]**, **Acc66E[*i*]**, **Acc67E[*i*]**, and **Acc68E[*i*]** – and are saved in the backup file as such. Index values ***i*** can range from 0 to 15, and are determined by the hardware address DIP switch setting for the accessory card.

GateIo[*i*].Init.CtrlReg

Description: IOGATE register direction initialization control

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Default: Configuration dependent

The software element **GateIo[*i*].Init.CtrlReg** contains the value that is written to the IC hardware setup register **GateIo[*i*].CtrlReg** automatically on a power-up/reset at the end of the configuration process for the IC. Only the low 6 bits of this 8 bit element are used for direction control; the high 2 bits must be set to 0 in this last configuration step so that actual data can be written to and/or read from the 6 data registers. This makes the effective range of this element 0 to 63. Earlier in the configuration process, the high 2 bits will be used automatically in the process of configuring the 6 data registers.

Bit ***j*** (***j*** = 0 to 5) of **GateIo[*i*].Init.CtrlReg** will determine the direction of the 8 I/O points in **GateIo[*i*].DataReg[*j*]**. A value of 0 in the control bit permits a write operation to the data register, enabling the output function for each line in the register. Enabling the output function does not prevent the use of any or all of the lines as inputs, as long as the outputs are off (non-conducting).

A value of 1 in the control bit does not permit a write operation to the data register, disabling the output, reserving the register for inputs. This setting is strongly recommended when using these lines for inputs, as a write operation to the data register cannot then disable the inputs.

On re-initialization, the value of this element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

GateIo[*i*].Init.DataReg0[*j*]

Description: IOGATE register power-on default values

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Default: 0

The software element **GateIo[*i*].Init.DataReg0[*j*]** contains the value that is written to the IC hardware setup register **GateIo[*i*].DataReg[*j*]** automatically on a power-up/reset when the hardware setup element **GateIo[*i*].CtrlReg** is set to 0 (bit 7 = 0, bit 6 = 0). In order for a new value of **GateIo[*i*].Init.DataReg0[*j*]** to have an effect on the hardware, the value must be stored to non-volatile flash memory with a **`save`** command, and the Power PMAC system must be reset.

IC index values ***i*** can range from 0 to 15, and must match the index of the hardware IC as set by the addressing switches for the IC. Register index values ***j*** can range from 0 to 5, and must match the address offset of the hardware data register in the IC. Each data register represents 8 I/O points.

The saved value of 8-bit element **GateIo[*i*].Init.DataReg0[*j*]** sets the initial value of the 8 outputs associated with the hardware register **GateIo[*i*].DataReg[*j*]**. Each bit specifies the initial value of the corresponding output point, matched in numerical order. If the register is used for inputs, the saved value of this element has no effect.

On re-initialization, the value of this element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides default "off" states for all outputs.

GateIo[*i*].Init.DataReg64[*j*]

Description: IOGATE register inversion initialization control

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Default: Configuration dependent

The software element **GateIo[*i*].Init.DataReg64[*j*]** contains the value that is written to the IC hardware setup register **GateIo[*i*].DataReg[*j*]** automatically on a power-up/reset when the hardware setup element **GateIo[*i*].CtrlReg** is set to 64 (bit 7 = 0, bit 6 = 1). In order for a new value of **GateIo[*i*].Init.DataReg64[*j*]** to have an effect on the hardware, the value must be stored

to non-volatile flash memory with a **save** command, and the Power PMAC system must be reset.

IC index values ***i*** can range from 0 to 15, and must match the index of the hardware IC as set by the addressing switches for the IC. Register index values ***j*** can range from 0 to 5, and must match the address offset of the hardware data register in the IC. Each data register represents 8 I/O points.

The saved value of **GateIo[*i*].Init.DataReg64[*j*]** determines the inversion of the 8 I/O points associated with the hardware register **GateIo[*i*].DataReg[*j*]**. Each bit specifies the inversion of the corresponding I/O point, matched in numerical order. A value of 0 specifies an inverting I/O point for the matching bit. That is, for an output, a value of 0 produces a low (conducting) output from the IC itself, and a value of 1 produces a high (non-conducting) output. For an input, a line pulled low into the IC produces a 1 value, and a line pulled high or permitted to float high produces a 0 value.

A value of 1 in a bit of **GateIo[*i*].Init.DataReg64[*j*]** specifies a non-inverting I/O point for the matching bit. That is, for an output, a value of 0 produces a high (non-conducting) output from the IC itself, and a value of 1 produces a low (conducting) output. For an input, a line into the IC pulled low produces a 0 value, and a line pulled high or permitted to float high produces a 1 value.

For IOGATE accessory boards that use isolators and driver ICs to interface to the field wiring (e.g. ACC-65E, ACC-66E, ACC-67E, and ACC-68E), the inverting setting means that the "conducting" state, whether sinking or sourcing, corresponds to a "1" in the bit of the data register for both inputs and outputs.

For IOGATE accessory boards in which the IC pins connect directly to the field wiring and have pull-up resistors on the board (e.g. ACC-14E), the inverting setting on an output means that the sinking transistor on the IC pin is turned on when a "1" is written to the bit of the data register, pulling the output line low. The transistor is turned off when a "0" is written to the bit of the data register, permitting the external resistor to pull the line high. The inverting setting on an input means that when the input pin is actively pulled low, a "1" can be read from the bit of the data register. When the input pin is actively pulled high, or allowed to be pulled up by the external resistor, a "0" can be read from the bit of the data register.

On re-initialization, the value of this element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

GateIo[*i*].Init.DataReg128[*j*]

Description: IOGATE register read initialization control

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Default: 0

The software element **GateIo[*i*].Init.DataReg128[*j*]** contains the value that is written to the IC hardware setup register **GateIo[*i*].DataReg[*j*]** automatically on a power-up/reset when the hardware setup element **GateIo[*i*].CtrlReg** is set to 128 (bit 7 = 1, bit 6 = 0). In order for a new value of **GateIo[*i*].Init.DataReg128[*j*]** to have an effect on the hardware, the value must be stored to non-volatile flash memory with a **save** command, and the Power PMAC system must be reset.

IC index values ***i*** can range from 0 to 15, and must match the index of the hardware IC as set by the addressing switches for the IC. Register index values ***j*** can range from 0 to 5, and must match the address offset of the hardware data register in the IC. Each data register represents 8 I/O points.

The saved value of **GateIo[*i*].Init.DataReg128[*j*]**, along with the saved value of **GateIo[*i*].Init.DataReg192[*j*]**, determines what is reported for the 8 I/O points associated with the hardware register **GateIo[*i*].DataReg[*j*]** in a read operation on the register. Each bit specifies the reporting of the corresponding I/O point, matched in numerical order.

The action of a bit of **GateIo[*i*].Init.DataReg128[*j*]** is dependent on the setting of the matching bit of **GateIo[*i*].Init.DataReg192[*j*]** for the same index ***j***. If the matching bit of **GateIo[*i*].Init.DataReg192[*j*]** is 0, selecting unlatched inputs, the bit of **GateIo[*i*].Init.DataReg128[*j*]** controls whether the pin value is read, or the value in the writeable register is read. A value of 0 in the bit of **GateIo[*i*].Init.DataReg128[*j*]** selects the pin value to be read from the matching bit of **GateIo[*i*].DataReg[*j*]** at the index ***j***; a value of 1 in the bit selects the writeable register value.

If the matching bit of **GateIo[*i*].Init.DataReg192[*j*]** is 1, selecting latched inputs, the bit of **GateIo[*i*].Init.DataReg128[*j*]** controls whether the directly latched data is read, or the value that is the result of a Gray-code-to-binary conversion. A value of 0 in the bit of **GateIo[*i*].Init.DataReg128[*j*]** selects the directly latched value to be read from the matching bit of **GateIo[*i*].DataReg[*j*]** at the index ***j***; a value of 1 in the bit selects the value that is the result of a Gray-code-to-binary conversion from the latched value.

On re-initialization, the value of this element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

GateIo[*i*].Init.DataReg192[*j*]

Description: IOGATE register latch initialization control

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Default: Configuration dependent

The software element **GateIo[*i*].Init.DataReg192[*j*]** contains the value that is written to the IC hardware setup register **GateIo[*i*].DataReg[*j*]** automatically on a power-up/reset when the hardware setup element **GateIo[*i*].CtrlReg** is set to 192 (bit 7 = 1, bit 6 = 1). In order for a new value of **GateIo[*i*].Init.DataReg192[*j*]** to have an effect on the hardware, the value must be

stored to non-volatile flash memory with a **save** command, and the Power PMAC system must be reset.

IC index values ***i*** can range from 0 to 15, and must match the index of the hardware IC as set by the addressing switches for the IC. Register index values ***j*** can range from 0 to 5, and must match the address offset of the hardware data register in the IC. Each data register represents 8 I/O points.

The saved value of **GateIo[*i*].Init.DataReg192[*j*]**, along with the saved value of **GateIo[*i*].Init.DataReg128[*j*]**, determines what is reported for the 8 I/O points associated with the hardware register **GateIo[*i*].DataReg[*j*]** in a read operation on the register. Each bit specifies the reporting of the corresponding I/O point, matched in numerical order.

The action of a bit of **GateIo[*i*].Init.DataReg192[*j*]** is dependent on the setting of the matching bit of **GateIo[*i*].Init.DataReg128[*j*]** for the same index ***j***. If the matching bit of **GateIo[*i*].Init.DataReg128[*j*]** is 0, the bit of **GateIo[*i*].Init.DataReg192[*j*]** controls whether the transparent pin value is read, or the value most recently latched (typically on a phase or servo clock edge) is read. A value of 0 in the bit of **GateIo[*i*].Init.DataReg192[*j*]** selects the transparent pin value to be read from the matching bit of **GateIo[*i*].DataReg[*j*]** at the index ***j***; a value of 1 in the bit selects the latched value.

If the matching bit of **GateIo[*i*].Init.DataReg128[*j*]** is 1, the bit of **GateIo[*i*].Init.DataReg192[*j*]** controls whether the value written to the register by the processor is read, or the value that is the result of a Gray-code-to-binary conversion from the latched input is read. A value of 0 in the bit of **GateIo[*i*].Init.DataReg192[*j*]** selects the most recent value written to the bit to be read from the matching bit of **GateIo[*i*].DataReg[*j*]** at the index ***j***; a value of 1 in the bit selects the value that is the result of a Gray-code-to-binary conversion from the latched value.

On re-initialization, the value of this element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

GateIo[*i*].Init.IntrReg64

Description: IOGATE interrupt-line inversion initialization control

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Default: 0

The software element **GateIo[*i*].Init.IntrReg64** contains the value that is written to the IC hardware setup register **GateIo[*i*].IntrReg** automatically on a power-up/reset when the hardware setup element **GateIo[*i*].CtrlReg** is set to 64 (bit 7 = 0, bit 6 = 1). In order for a new value of **GateIo[*i*].Init.IntrReg64** to have an effect on the hardware, the value must be stored to non-volatile flash memory with a **save** command, and the Power PMAC system must be reset.

IC index values ***i*** can range from 0 to 15, and must match the index of the hardware IC as set by the addressing switches for the IC.

The saved value of **GateIo[*i*].Init.IntrReg64** determines the inversion of the 8 the "E*n*" line inputs to the IOGATE IC. Each bit specifies the inversion of the corresponding "E*n*" line, matched in numerical order. A value of 0 specifies an inverting I/O point for the matching bit. A low level into the IC produces a 1 value, and a high level produces a 0 value.

A value of 1 in a bit of **GateIo[*i*].Init.DataReg0[*j*]** specifies a non-inverting I/O point for the matching bit. A low level produces a 0 value, and a high level produces a 1 value.

On re-initialization, the value of this element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

On present Power PMAC I/O accessories, this feature does not have significant use. The E*n* inputs to the IC can only come from the system servo or phase clock signals (as selected by jumpers), which can be used to latch the input data.

GateIo[*i*].Init.IntrReg128

Description: IOGATE interrupt-line mask initialization control

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Default: Configuration dependent

The software element **GateIo[*i*].Init.IntrReg128** contains the value that is written to the IC hardware setup register **GateIo[*i*].IntrReg** automatically on a power-up/reset when the hardware setup element **GateIo[*i*].CtrlReg** is set to 128 (bit 7 = 1, bit 6 = 0). In order for a new value of **GateIo[*i*].Init.IntrReg128** to have an effect on the hardware, the value must be stored to non-volatile flash memory with a **`save`** command, and the Power PMAC system must be reset.

IC index values ***i*** can range from 0 to 15, and must match the index of the hardware IC as set by the addressing switches for the IC.

The saved value of **GateIo[*i*].Init.IntrReg128** determines which of the "E*n*" line inputs to the IOGATE IC can create an "interrupt" output from the IC. Each bit specifies the use of the corresponding E*n* line, matched in numerical order. A value of 0 disables the use for the matching E*n* line. A value of 1 enables the use of the matching E*n* line. A falling edge of the E*n* input will create a falling edge of the IC's "INT" output.

On present Power PMAC I/O accessories, this feature does not have significant use. The E*n* inputs to the IC can only come from the system servo or phase clock signals (as selected by jumpers), which can be used to latch the input data. The "INT" output from the IC cannot interrupt the processor, but it can be read in the "ID" chip for the accessory.

GateIo[*i*].Init.IntrReg192

Description: IOGATE interrupt-line edge/level initialization control

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Default: Configuration dependent

The software element **GateIo[*i*].Init.IntrReg192** contains the value that is written to the IC hardware setup register **GateIo[*i*].IntrReg** automatically on a power-up/reset when the hardware setup element **GateIo[*i*].CtrlReg** is set to 192 (bit 7 = 1, bit 6 = 1). In order for a new value of **GateIo[*i*].Init.IntrReg192** to have an effect on the hardware, the value must be stored to non-volatile flash memory with a **`save`** command, and the Power PMAC system must be reset.

IC index values ***i*** can range from 0 to 15, and must match the index of the hardware IC as set by the addressing switches for the IC.

The saved value of **GateIo[*i*].Init.IntrReg192** determines which of the "E*n*" line inputs to the IOGATE IC can provide a level-triggered interrupt and which can provide an edge-triggered interrupt on the "interrupt" output from the IC. Each bit specifies the use of the corresponding E*n* line, matched in numerical order. A value of 0 specifies a level-triggered interrupt for the matching E*n* line. A value of 1 specifies an edge-triggered interrupt the matching E*n* line. This setting is only relevant if the input is enabled for interrupts.

On present Power PMAC I/O accessories, this feature does not have significant use. The E*n* inputs to the IC can only come from the system servo or phase clock signals (as selected by jumpers), which can be used to latch the input data. The "INT" output from the IC cannot interrupt the processor, but it can be read in the "ID" chip for the accessory.

Macro. Saved Data Structure Elements

Power PMAC's MACRO ring functionality has several saved setup elements specifying overall ring functionality.

Macro.IOTimeout

Description: Maximum detected ring errors for valid ring operation

Range: 0 .. 32,767

Units: Servo interrupt periods

Default: 0 (specifies 100 servo periods)

Macro.IOTimeout specifies the maximum time delay in receiving a response to a MACRO auxiliary (master-to-slave or master-to-master) communications command before timing out and declaring a communications error. If left at the factory default value of 0, it specifies a delay of 100 servo periods.

Macro.IOTimeout is equivalent to I78 and I79 (combined) in Turbo PMAC.

Macro.TestMaxErrors

Description: Maximum detected ring errors for valid ring operation

Range: 0 .. 32,767

Units: Errors

Default: 0

Macro.TestMaxErrors specifies the maximum number of MACRO ring communications errors that can be detected in a single ring test period without causing a ring fault shutdown. It is only used if **Macro.TestPeriod** is set to a value greater than 0 to enable this testing.

With testing active, each servo interrupt in the test period the Power PMAC checks to see if a ring communications error has been detected since the previous cycle. There are four types of these errors that can be detected by the interface hardware: byte violation error, packet parity error, packet overflow error, or packet underflow error. This test is performed on all the detected ring(s).

If any of these errors has been detected since the previous test cycle, Power PMAC increments the error counter. At the end of this test period, if this count is greater than **Macro.TestMaxErrors**, a ring fault (**Macro.Status[*i*].ErrorsFault**) is generated.

A setting for **Macro.TestMaxErrors** of 10% of **Macro.TestPeriod** is suggested.

The value of this element can also be set with the on-line **MacroStationRingCheck** command.

Macro.TestMaxErrors is equivalent to I81 in a Turbo PMAC system.

Macro.TestPeriod

Description: MACRO ring check test period

Range: 0 .. 32,767

Units: Servo interrupt periods

Default: 0 (disabled)

Macro.TestPeriod specifies the period for the Power PMAC to evaluate whether there has been a MACRO ring failure. If it is set to the default value of 0, no automatic checking is performed. (This is not recommended because it disables all ring integrity checking, including ring-break detection.)

If it is set greater than 0, Power PMAC checks each MACRO ring every servo interrupt period to see if any ring errors have been detected since the last servo cycle (note that there are usually several ring update cycles each servo cycle) and if any "sync packets" have been received since the last servo cycle. It accumulates the error count and the sync packet count during the test period. This test is performed on all the detected ring(s).

If, at the end of the test period, the number of ring errors detected is greater than the threshold set by saved setup element **Macro.TestMaxErrors**, or the number of sync packets received is less than the threshold set by saved setup element **Macro.TestReqdSynchs**, then a ring fault (**Macro.Status[*i*].RingError**) is determined to have occurred.

Typically, a value of about 40 servo interrupt periods is used for **Macro.TestPeriod**.

The value of this element can also be set with the on-line **MacroStationRingCheck** command.

Macro.TestPeriod is equivalent to I80 in a Turbo PMAC system

Macro.TestReqdSynchs

Description: Minimum received sync packets for valid ring operation

Range: 0 .. 32,767

Units: Packet receptions

Default: 0

Macro.TestReqdSynchs specifies the minimum number of MACRO "sync packets" that must be detected in a single ring test period so as not to cause a ring fault shutdown. It is only used if **Macro.TestPeriod** is set to a value greater than 0 to enable this testing.

Which MACRO data packet is the "sync packet" is specified by part of saved setup element **Gate*n*[*i*].MacroEnable**. In almost all cases, the Node 15 ($F) data packet is used as the sync packet, because it is used by all MACRO devides.

With testing active, each cycle in the test period the Power PMAC checks to see if a sync packet has been received since the previous cycle. If one has been received since the previous test cycle, Power PMAC increments the sync counter. At the end of the test period, if this count is less than **Macro.TestReqdSynchs**, a ring fault (**Macro.Status[*i*].SynchFault**) is generated. This test is performed on all the detected ring(s).

Typically a value for **Macro.TestReqdSynchs** equal to **Macro.TestPeriod** – **Macro.TestMaxErrors** is used, meaning that a sync packet must be detected in all test cycles where a ring error was not detected.

The value of this element can also be set with the on-line **`MacroStationRingCheck`** command.

Macro.TestReqdSynchs is equivalent to I82 in a Turbo PMAC system.

Motor[*x*]. Saved Data Structure Elements

The **Motor[*x*].** data structure provides all of the software information for the specified motor. This section describes those elements in the data structure that are saved to non-volatile memory.

Note: The motor index value in the square brackets may be specified by an integer constant in the range of valid motor numbers for the Power PMAC system, or a "local" L-variable for the program or communications thread (L0 – L1007 can be used). No other method of specifying the index value may be used. Index values can go from 0 to **Sys.MaxMotors** – 1, with **Sys.MaxMotors** having a maximum value of 256.

Motor[*x*].AbortTa

Description: Abort deceleration time or inverse rate

Range: Floating-point

Units: Milliseconds (if >= 0) or milliseconds^2 per motor unit (if < 0)

Default: -2.0 (= 0.5 motor units per msec^2)

Legacy I-variable alias: I*x*15

Motor[*x*].AbortTa sets the time or the magnitude of the deceleration to a stop for the motor on an "abort". If it is greater than or equal to zero, it specifies the time for the abort deceleration in milliseconds. If it is less than zero, it specifies the inverse of the peak magnitude of the abort deceleration, in milliseconds squared per motor unit (usually raw count).

If **Motor[*x*].AbortTa** specifies a time and **Motor[*x*].AbortTs** (S-curve time) specifies a smaller time, the total time spent in abort deceleration will be equal to the sum of **AbortTa** and **AbortTs**. However, if **Motor[*x*].AbortTa** specifies a time and **Motor[*x*].AbortTs** specifies a larger time, the total time spent in abort deceleration will be 2 times **Motor[*x*].AbortTs**. Therefore, if **Motor[*x*].AbortTa** is set to 0, **Motor[*x*].JogTs** alone controls the acceleration time in "pure" S-curve form.

If **Motor[*x*].AbortTs** is less than zero, specifying a maximum "jerk", **Motor[*x*].AbortTa** must also be less than zero, specifying a maximum acceleration.

The motor will be aborted if its coordinate system is given an **a** on-line command or an **abort** buffered direct program command. It will also be aborted automatically if it exceeds either a hardware or software overtravel position limit, or if a "run-time error" occurs in its coordinate system.

Example

To set an abort deceleration of 20 m/s^2 (about 2g) with motor units of microns:

$$AbortTa = -\frac{s^2}{20m} * \frac{10^{-6} m}{motor\_unit} * \frac{10^6 ms^2}{s^2} = -0.05\left(\frac{ms^2}{motor\_unit}\right)$$

Motor[x].AbortTs

Description: Abort S-curve deceleration time or inverse jerk rate

Range: Floating-point

Units: Milliseconds (if >= 0) or milliseconds^3 per motor unit (if < 0)

Default: 0.0 (no S-curve time)

Legacy I-variable alias: I*x*19

Motor[*x*].AbortTs sets the time of the S-curve portion of the deceleration to a stop for the motor on an "abort", or the maximum magnitude of the "jerk" (rate of change of acceleration) during this deceleration. If it is greater than or equal to zero, it specifies the time for each S-curve section the abort deceleration in milliseconds. If it is less than zero, it specifies the inverse of the peak magnitude of the abort jerk, in milliseconds cubed per motor unit (usually raw count).

If **Motor[*x*].AbortTs** is less than zero, specifying a maximum "jerk", **Motor[*x*].AbortTa** must also be less than zero, specifying a maximum abort deceleration. If **Motor[*x*].AbortTs** is set to 0.0, there is no S-curve acceleration portion, and the jog acceleration and deceleration rates will be constant.

If **Motor[*x*].AbortTa** specifies a time and **Motor[*x*].AbortTs** (S-curve time) specifies a smaller time, the total time spent in acceleration will be equal to the sum of **AbortTa** and **AbortTs**. However, if **Motor[*x*].AbortTa** specifies a time and **Motor[*x*].AbortTs** specifies a larger time, the total time spent in abort deceleration will be 2 times **Motor[*x*].AbortTs**. Therefore, if **Motor[*x*].AbortTa** is set to 0, **Motor[*x*].AbortTs** alone controls the abort deceleration time in "pure" S-curve form.

The motor will be aborted if its coordinate system is given an **a** on-line command or an **abort** buffered direct program command. It will also be aborted automatically if it exceeds either a hardware or software overtravel position limit, or if a "run-time error" occurs in its coordinate system.

Example

To set an abort jerk rate of 50 m/s^3 (about 5g/sec) with motor units of microns:

$$AbortTs = -\frac{s^3}{50m} * \frac{10^{-6} m}{motor\_unit} * \frac{10^9 ms^3}{s^3} = -20\left(\frac{ms^3}{motor\_unit}\right)$$

Motor[x].AbsPhasePosForce

Description: Commutation position forcing value

Range: 0.0 .. 2047.999

Units: 1/2048 commutation cycle

Default: 0.0

Motor[*x*].AbsPhasePosForce provides a savable value that can be used to "force" the present value of the phase position. It is commonly used when the initial phase referencing is only approximate, as with a Hall-sensor read, or a phasing-search move on a heavily loaded motor. Usually these methods are sufficient to get basic motion capabilities, but the remaining errors in the phase reference prevent optimal motor performance. In these cases, a homing search move whose trigger includes the encoder index pulse is sufficient to get the motor to a precisely known location in its commutation cycle. The following command can then be executed (either on-line or in a program):

```
Motor[x].PhasePos = Motor[x].AbsPhasePosForce
```

The desired value of **Motor[*x*].AbsPhasePosForce** is typically determined by performing a "stepper-motor" phasing search on the unloaded motor (which is very accurate without load), then doing a homing-search move using the encoder index pulse as the trigger. The value of **Motor[*x*].PhasePos** at the home position can then be copied into **Motor[*x*].AbsPhasePosForce** and saved.

Note

There is no automatic use of **Motor[*x*].AbsPhasePosForce**. It simply provides a convenient saved value for user applications.

Motor[*x*].AbsPhasePosFormat

Description: Power-on absolute commutation position data format

Range: $0 .. $FFFFFFFF

Units: Byte field

Default: $0

Legacy I-variable alias: I*x*91

Motor[*x*].AbsPhasePosFormat determines how Power PMAC will read and interpret the data at the address specified by **Motor[*x*].pAbsPhasePos** to get the absolute commutation rotor-angle position. After this data is formatted, it is scaled into commutation-angle units (including possible direction inversion) with **Motor[*x*].AbsPhasePosSf**, and then offset with **Motor[*x*].AbsPhasePosOffset** to obtain the resulting commutation angle.

Motor[*x*].AbsPhasePosFormat is a 32-bit value consisting of four byte fields. It can be considered to have the hexadecimal format $*aabbccdd*, where the $*aa*, $*bb*, $*cc*, and $*dd* hexadecimal character pairs represent each of the four bytes.

- $*dd* specifies the number of the starting bit to be used in the (first) register addressed by **Motor[*x*].pAbsPhasePos**. It can take a value from $00 to $1F (0 to 31) to specify this. This bit will be used as the least significant bit of the resulting value. A value of $20 here specifies that the value will be obtained over the MACRO ring as explained below.

If the $*dd* byte is set to $20 (32), the motor is specified to obtain its absolute position over the MACRO ring through a query of the MI920 parameter of the MACRO slave node for the motor. In this case, the $*cc* byte specifies the number of bits to be used from the reported value of the MI920 parameter, and the $*bb* byte specifies the number of the starting bit to be used from this value. The $*aa* byte specifies how this value is to be interpreted as for other data sources.

When absolute phase position for a motor is read over the MACRO ring, it is automatically read from the MACRO node specified by **Motor[*x*].pEncStatus**. In this case, the address specified by **Motor[*x*].pAbsPhasePos** is not used, but **Motor[*x*].pAbsPhasePos** must be set to a non-zero value to specify an absolute position read. It is recommended to set it to the same address as **Motor[*x*].pEncStatus** (e.g. **Gate2[*i*].Macro[*j*][*k*].a** or **Gate3[*i*].MacroInA[*j*][*k*].a**).

- $*cc* specifies the total number of bits to be used to create the resulting value. This can take a value of up to $20 (32). Remember that phase position need only be absolute over a single commutation cycle, and that Power PMAC's internal commutation cycle only has 11-bit resolution (2048 parts). For linear encoders, where the commutation cycle size is generally not a power of 2, typically the entire range of the encoder must be used. If the range has more than 32 bits, lower bits should be excluded. When a Hall-sensor format is specified, only 3 bits will be used regardless of the setting of this byte, but it is advisable to set this byte to $03 for clarity.

- $*bb* specifies how to use data in subsequent registers. It will only be used if the number of bits specified in $*cc* cannot be found in the first register. A value in $*bb* of $00 to $1F (0 to 31) specifies the number of the starting bit to be used in the next higher-addressed register (at **Motor[*x*].pAbsPhasePos** + 4). A value in $*bb* of $20 (32) specifies the byte-wide format used in the "IOGATE" IC. In this format, data in each subsequent register (up to 3) is used starting at the same bit number as the first register ($*dd*). If all the position information is found in a single register, this byte is not used, and is usually set to $00.

- $*aa* specifies how the Power PMAC will interpret the data that is read. It can take the following values:

    - $00: Interpret as numerical binary (no conversion required)
    - $02: Interpret as Gray code and convert to numerical binary
    - $04: Interpret as 3-phase digital Hall-sensor data with 120° signal spacing and convert to numerical binary
    - $05: Interpret as 3-phase digital Hall-sensor data with 60° signal spacing and convert to numerical binary

  Note that because the commutation position data is used in a cyclical fashion, there is no need to specify a signed or unsigned format as in the absolute servo position format.

Examples

A 17-bit rotary encoder provides data in numerical binary format starting in bit 0 of a 32-bit register. **Motor[*x*].AbsPhasePosFormat** is set to $00001100. (*aa* = $00 for numerical binary, *bb* = $00 for no extended register, *cc* = $11 for 17 bits, *dd* = $00 for starting in bit 0)

A 13-bit rotary encoder provides Gray-code data starting in bit 8 of a 32-bit register. **Motor[*x*].AbsPhasePosFormat** is set to $02000D08. (*aa* = $02 for Gray code, *bb* = $00 for no extended register, *cc* = $0D for 13 bits, *dd* = $08 for starting in bit 8)

A 3-phase 120° digital Hall sensor read at bits 28 – 30 of the ASIC's channel register is used. **Motor[*x*].AbsPhasePosFormat** is set to $0400031C. (*aa* = $04 for 120° Hall format, *bb* = $00 for no extended register, *cc* = $03 for 3 bits, *dd* = $1C for starting in bit 28)

24 bits of numerical-binary data read through byte-wide registers in an ACC-14E is used. **Motor[*x*].AbsPhasePosFormat** is set to $00201808. (*aa* = $00 for numerical binary, *bb* = $20 for multiple byte-wide extended registers, *cc* = $18 for 24 bits, *dd* = $08 for starting in bit 8)

A 17-bit absolute position is read over the MACRO ring, with the LSB in bit 0 of the returned position. **Motor[*x*].AbsPhasePosFormat** is set to $00001120. (*aa* = $00 for numerical binary, *bb* = $00 for no extended register, *cc* = $11 for 17 bits, *dd* = $20 for MACRO ring query)

A linear encoder with 36 bits covering the full range starting in bit 8 of a 32-bit register and continuing starting in bit 8 of the next register is used. Only the high 32 of the 36 bits will be used. **Motor[*x*].AbsPhasePosFormat** is set to $0008200C. (*aa* = $00 for numerical binary, *bb* = $08 for continuing in bit 8 of the next register, *cc* = $20 for 32 bits, *dd* = $0C for starting in bit 12)

Motor[*x*].AbsPhasePosOffset

Description: Power-on absolute commutation position offset

Range: Floating-point

Units: 1/2048 commutation cycle

Default: 0.0

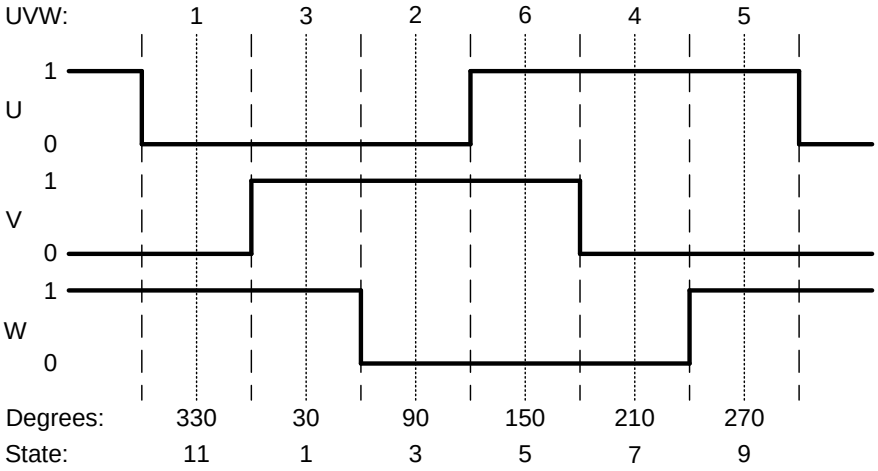
Legacy I-variable alias: I*x*75

Motor[*x*].AbsPhasePosOffset is used in the process of establishing the power-on phase position for a synchronous (zero-slip) motor commutated by the Power PMAC. It is used differently depending on whether this position is established by an absolute position read (**Motor[*x*].PhaseFindingTime** = 0) or by a phasing-search move (**Motor[*x*].PhaseFindingTime** != 0).

For an absolute position read, **Motor[*x*].AbsPhasePosOffset** specifies the amount added to the absolute position value *after* it has been scaled by **Motor[*x*].AbsPhasePosSf** into the commutation units of 1/2048 of a cycle. As such, this term expressed in the units of 1/2048 of a cycle.

The proper value for this term is usually set by performing the "stepper motor" phasing search, which drives the motor to the zero point in the commutation cycle, and reading the value of the absolute sensor at this point. The negative of this value is multiplied by **Motor[*x*].AbsPosSf** and the product is written into **Motor[*x*].AbsPhasePosOffset**.

Hall commutation sensors require special consideration. The following diagram shows the standard configuration for Hall sensors with 120° signal separation. (For the case of 60° separation of signals, simply invert the "V" signal in this diagram.)



Power PMAC considers the 0° point in the Hall cycle to be the transition of the V signal when U is low (0) and W is high (1). In the "standard" direction sense shown in this diagram (W leads V and V leads U), if this V transition is at the 0° point in Power PMAC's commutation cycle, **Motor[*x*].AbsPhasePosOffset** would be set to 0.0.

In general, this V transition will not be at the zero position in Power PMAC's commutation cycle, so it is important to find out where this transition is. As with other sensors, this is usually done by first performing a stepper-motor phasing search on the unloaded motor to establish the phase reference. (Make sure that **Motor[*x*].AbsPhasePosSf** is set properly first.) Then, while monitoring the commutation angle in **Motor[*x*].PhasePos** and the U, V, and W states, the motor should be moved slowly until this V transition is found. (Killing the motor and moving it by hand often works well.) Note the value of **Motor[*x*].PhasePos** at this transition.

If the direction senses of the sensor and the commutation cycle match (so **Motor[x].AbsPhasePosOffset** is positive), put this value in **Motor[*x*].AbsPhasePosOffset**. If the direction senses do not match (so **Motor[x].AbsPhasePosOffset** is negative), put the opposite of this value in **Motor[*x*].AbsPhasePosOffset**. Note that since this phasing is only approximate and will be corrected later, it is not necessary to be too precise here.

For a phasing-search move, **Motor[*x*].AbsPhasePosOffset** specifies the minimum motion that qualifies the search as being a valid search. If the move provides less motion than this, Power PMAC will consider the search to have failed, in which case it will clear the motor's **PhaseFound** status bit, "kill" the motor at the end of the search and not permit the motor to be enabled by the user.

For a "stepper-motor" phasing-search (**Motor[*x*].PhaseFindingTime** > 255), the movement between the first and second "steps" is evaluated. Ideally, this should be +/-1/4 of a commutation

cycle (~512 commutation units). Setting **Motor[*x*].AbsPhasePosOffset** to about (~1/5 of a cycle) will provide good assurance that a valid move is being performed without risking nuisance faults.

For a "four-guess" phasing-search (**Motor[*x*].PhaseFindingTime** < 256), the sum of the squares of the movement magnitudes (in motor units, not commutation units) of the "sine" and "cosine" guesses is evaluated and compared to the square of **Motor[*x*].AbsPhasePosOffset**. If it is less than this value, it will consider the search to have failed. The optimum value in this case will depend on the typical magnitude of motion in the search.

The magnitude of motion for the "sine" guesses is stored in **Motor[*x*].New[0].Vel**; the magnitude of motion for the "cosine" guesses is stored in **Motor[*x*].New[0].Accel**. These magnitudes should be evaluated for a significant number of searches starting at various locations in the cycle. **Motor[*x*].AbsPhasPosOffset** can then be set to a value a little bit less than the minimum valid square root of the sum of squares of these values.

Motor[*x*].AbsPhasePosSf

Description: Power-on absolute commutation position scale factor

Range: Floating-point

Units: 1/2048 commutation cycle per unit of source data

Default: 0.0

Motor[*x*].AbsPhasePosSf specifies the multiplication factor used to scale the absolute rotor-angle data read from the register at **Motor[*x*].pAbsPhasePos** and interpreted according to **Motor[*x*].AbsPhasePosFormat** into the commutation units of 1/2048 of a cycle. As such, it has units of the commutation units per LSB of the source data. Note that this scale factor does not need to be the same as for the ongoing commutation position data.

When Power PMAC decodes 3-phase Hall commutation sensors, it considers the resulting position value to have 12 units per cycle – 6 states and 6 edges.

If the up/down direction sense of the absolute phase position sensor does not match the up/down direction sense of the commutation cycle, this value should be set to a negative number. For Hall sensors with the standard 120° separation of signals, if W leads V and V leads U in the counting-up sense of the commutation cycle, the direction sense agrees, and this should be a positive number. However, if U leads V, and V leads W in the counting-up sense, the direction sense does not agree, so this should be set to a negative number. (For the rare 60° separation, the rule is the opposite.)

Note that unlike **Motor[*x*].PhasePosSf**, which is used in ongoing commutation calculations and must be very precise to keep accurate position values over many revolutions, **Motor[*x*].AbsPhasePosSf** is used only once, and so can tolerate small round-off errors.

Examples

A 17-bit rotary encoder is used for power-on position on a 4-pole motor. The encoder provides 131,072 counts per mechanical revolution. There are 2 commutation cycles per revolution,

yielding 4,096 commutation units per revolution. **Motor[*x*].AbsPhasePosSf** is set to 4096/131,072, or 0.03125.

A 3-phase Hall sensor with 120° signal separation is used for power-on position. U leads V and V leads W in the counting-up sense of the commutation cycle, so it is considered to have an "inverted" direction sense. Since the Hall sensor is considered to provide 12 counts per commutation cycle (6 states plus 6 edges), **Motor[*x*].AbsPhasePosSf** is set to -2048/12, or -170.667.

Motor[*x*].AbsPosFormat

Description: Power-on absolute position data format

Range: $0 .. $FFFFFFFF

Units: Byte field

Default: $0

Legacy I-variable alias: I*x*95

Motor[*x*].AbsPosFormat determines how Power PMAC will read and interpret the data at the address specified by **Motor[*x*].pAbsPos** to get the absolute motor position. After this data is formatted, it is scaled into motor units with **Motor[*x*].AbsPosSf**, and then offset with **Motor[*x*].HomeOffset** to obtain the resulting absolute motor position.

Motor[*x*].AbsPosFormat is a 32-bit value consisting of four byte fields. It can be considered to have the hexadecimal format $*aabbccdd*, where the $*aa*, $*bb*, $*cc*, and $*dd* hexadecimal character pairs represent each of the four bytes.

- $*dd* specifies the number of the starting bit to be used in the (first) register addressed by **Motor[*x*].pAbsPos**. When reading the data directly, this byte can take a value from $00 to $1F (0 to 31). This bit will be used as the least significant bit of the resulting value.

  If the $*dd* byte is set to $20 (32), the motor is specified to obtain its absolute position over the MACRO ring through a query of the MI920 parameter of the MACRO slave node for the motor from a remote MACRO station (but not another PMAC acting as an auxiliary slave). In this case, the $*cc* byte specifies the number of bits to be used from the reported value of the MI920 parameter, and the $*bb* byte specifies the number of the starting bit to be used from this value. The $*aa* byte specifies how this value is to be interpreted as for other data sources.

  When absolute position for a motor is read over the MACRO ring, it is automatically read from the MACRO node specified by **Motor[*x*].pEncStatus**. In this case, the address specified by **Motor[*x*].pAbsPos** is not used, but **Motor[*x*].pAbsPos** must be set to a non-zero value to specify an absolute position read. It is recommended to set it to the same address as **Motor[*x*].pEncStatus** (e.g. **Gate2[*i*].Macro[*j*][*k*].a** or **Gate3[*i*].MacroInA[*j*][*k*].a**).

- $*cc* specifies the total number of bits to be used to create the resulting value. This can take a value of up to $40 (64). Remember that this position must be absolute over the entire travel of the motor in order to be useful.

- $*bb* specifies how to use data in subsequent registers. It will only be used if the number of bits specified in $*cc* cannot be found in the first register. A value in $*bb* of $00 to $1F (0 to 31) specifies the number of the starting bit to be used in the next higher-addressed register (at **Motor[*x*].pAbsPos** + 4). A value in $*bb* of $20 (32) specifies the byte-wide format used in the "IOGATE" IC. In this format, data in each subsequent register (up to 3) is used starting at the same bit number as the first register ($*dd*). If all the position information is found in a single register, this byte is not used, and is usually set to $00.

- $*aa* specifies how the Power PMAC will interpret the data that is read. Typically, it takes one of the following values:

    - $00: Interpret as unsigned numerical binary (no conversion required)
    - $01: Interpret as signed numerical binary (no conversion required)
    - $02: Interpret as Gray code and convert to unsigned numerical binary
    - $03: Interpret as Gray code and convert to signed numerical binary

  However, if any of the high 5 bits of the 8-bit $*aa* value is set to 1, the data read from the register specified in **Motor[*x*].pAbsPos** is first shifted left before subsequent processing is performed. The purpose of this feature is to eliminate "gaps" in the position data between this register and the subsequent register, as occurs in some serial encoder protocols, by shifting the most significant bit of position data to bit 31 of the intermediate register.

  The high 4 bits, which comprise the first "*a*" hex digit, specify a value from 0 to 15 denoting the number of bits to shift left the data read from the source register. If the next bit, which is the "8's" bit of the 2nd hex digit, is set to 1, the data from the source register is shifted left an additional 16 bits. (No presently supported protocols require this additional shift.) Note that the "starting bit" number specified in $*dd* denotes the number of the low bit to use *after* this initial shift.

  For example, several serial encoder protocols provide 17 bits of "single turn" position data in the low end of **SerialEncDataA** and 16 more bits of "multi-turn" position data in **SerialEncDataB**. If the data comes from the 32-bit register **Gate3[*i*].Chan[*j*].SerialEncDataA**, a 15-bit left shift is required, making the first "*a*" hex digit a $F and the $*dd* starting bit value equal to $0F. If the data comes from the 24-bit register **Acc84E[*i*].Chan[*j*].SerialEncDataA**, a 7-bit left shift is required, making the first "*a*" hex digit a $7 and the $*dd* starting bit value equal to $07.

Examples

A 29-bit multi-turn rotary encoder provides data in numerical binary format starting in bit 0 of a 32-bit register. We want to interpret the data as unsigned. **Motor[*x*].AbsPosFormat** is set to $00001D00. (*aa* = $00 for unsigned numerical binary, *bb* = $00 for no extended register, *cc* = $1D for 29 bits, *dd* = $00 for starting in bit 0)

A 27-bit multi-turn rotary encoder provides Gray-code data starting in bit 8 of a 32-bit register, continuing in bit 0 of the next register. We want to interpret the data as signed. **Motor[*x*].AbsPosFormat** is set to $03001B08. (*aa* = $03 for signed Gray code, *bb* = $00 for continue in bit 0 of extended register, *cc* = $1B for 27 bits, *dd* = $08 for starting in bit 8)

A 33-bit multi-turn serial encoder provides 17 bits of single-turn data in the low bits of a 32-bit register, and 16 bits of multi-turn data in the low bits of the next 32-bit register. We want to interpret the data as unsigned numerical binary. **Motor[*x*].AbsPosFormat** is set to $F0002100. (*aa* = $F0 for 15-bit left shift of primary register, *bb* = $00 for start at bit 0 of extended register, *cc* = $21 for 33 bits, *dd* = $00 for starting in bit 0)

36 bits of parallel numerical-binary data read through byte-wide registers in an ACC-14E is used. We want to interpret the data as signed. **Motor[*x*].AbsPosFormat** is set to $01202408. (*aa* = $01 for signed numerical binary, *bb* = $20 for multiple byte-wide extended registers, *cc* = $24 for 36 bits, *dd* = $08 for starting in bit 8)

A 23-bit MLDT timer register on an ACC-24E2*x* board provides unsigned position in bits 8 – 30 of a 32-bit register. **Motor[*x*].AbsPosFormat** is set to $00001708. (*aa* = $00 for unsigned numerical binary, *bb* = $00 for no extended register, *cc* = $17 for 23 bits, *dd* = $08 for starting in bit 8)

A 24-bit absolute position is read over the MACRO ring, to be interpreted as a signed value, with the LSB in bit 0 of the returned position. **Motor[*x*].AbsPosFormat** is set to $01001820. (*aa* = $01 for signed numerical binary, *bb* = $00 for no extended register, *cc* = $18 for 24 bits, *dd* = $20 for MACRO query)

Motor[*x*].AbsPosSf

Description: Power-on absolute position scale factor

Range: Floating-point

Units: Motor units per source unit

Default: 0.0

Motor[*x*].AbsPosSf specifies the multiplication factor used to scale the absolute position data read from the register at **Motor[*x*].pAbsPos** and interpreted according to **Motor[*x*].AbsPosFormat** into the user's motor units. As such, it has units of the motor units per LSB of the source data. (This is the LSB of the data actually used from the source register(s), not necessarily the LSB of the entire register.) Note that this scale factor does not need to be the same as for the ongoing servo position data.

If the up/down direction sense of the absolute position sensor does not match the up/down direction sense of the commutation cycle, this value should be set to a negative number.

After this multiplication is performed, the value of **Motor[*x*].HomeOffset** is added to the product, and the resulting sum is used to establish the motor's absolute position in motor units.

Motor[x].AdcMask

Description: Current-feedback bit-pass mask word

Range: $00000000 .. $FFFFFFFF

Units: Bit mask

Default: $FFF00000 (use high 12 bits)

Legacy I-variable alias: I*x*84

Motor[*x*].AdcMask tells Power PMAC what bits of the 32-bit current feedback word(s) to use as the actual current value in the current-loop equations. It is only used if current-loop closure is enabled for this motor.

Generally, the current feedback data appears in the most significant bits of registers in Servo ICs or MACRO ICs, left justified as data shifted in from serial ADCs. The most common resolution is 12 bits, but other resolutions are possible. Some amplifiers will transmit status and fault information in less significant bits of the same words, and it is important to mask out these values from the current loop equations.

Motor[*x*].AdcMask specifies a 32-bit mask word that is combined with the 32-bit feedback word through a logical AND operation to produce the value that is used in the current loop equations. There should be a 1 in every bit that is used, and a 0 in every bit that is not. Since the data is typically left justified, **Motor[*x*].AdcMask** should start with 1s and end with 0s. Usually **Motor[*x*].AdcMask** is represented as a hexadecimal number, with 4 bits per digit, and a total of eight digits.

Examples

For a 12-bit ADC: **Motor[*x*].AdcMask** = $FFF00000
For a 14-bit ADC: **Motor[*x*].AdcMask** = $FFFC0000
For a 16-bit ADC: **Motor[*x*].AdcMask** = $FFFF0000

Motor[x].AdvGain

Description: Commutation phase advance gain

Range: Non-negative floating-point

Units: 1/2048 commutation cycle per (inner-loop motor unit per 16 servo cycles)

Default: 0.0

Legacy I-variable alias: I*x*56

Motor[*x*].AdvGain permits the Power PMAC to compensate for lags in the electrical circuits of the motor phases, and/or for calculation and transport delays in the commutation of the motor, therefore improving high-velocity performance. The compensation is simply **Motor[*x*].AdvGain** multiplied by the inner-loop filtered actual motor velocity value in **Motor[*x*].FltrVel2**.

Motor[*x*].AdvGain is only used if Power PMAC is commutating Motor *x* (**Motor[*x*].PhaseCtrl** bit 0 or bit 2 = 1). It should be only be used for the commutation of synchronous motors (**Motor[*x*].SlipGain** = 0) such as permanent magnet brushless motors. **Motor[*x*].AdvGain** should be set to 0 for asynchronous motors (**Motor[*x*].SlipGain** > 0) such as AC induction motors.

If Power PMAC is commutating the motor, but not closing the current loop (**Motor[*x*].pAdc** = 0), **Motor[*x*].AdvGain** can improve performance typically starting at a few thousand RPM, because it compensates for inductive lags in the motor windings. If Power PMAC is also closing the current loop for the motor (**Motor[*x*].pAdc** > 0), the DC field-frame current loop closure compensates for inductive lags, and only small calculation delays need to be compensated; these are usually not significant until well over 10,000 rpm for low-pole-count servo motors, or over about 1200 rpm for a high-pole-count stepper motor.

This parameter is often set experimentally by running the motor at high speeds, and finding the setting that minimizes the current draw of the motor.

Motor[*x*].AmpEnableBit

Description: Bit number of amplifier enable signal in **pAmpEnable** register

Range: 0 .. 31

Units: Bit number (little-endian)

Default: Auto-configured based on hardware

Motor[*x*].AmpEnableBit determines which bit of the 32-bit register whose address is specified by **Motor[*x*].pAmpEnable** Power PMAC will use to control the amplifier-enable output for the motor. The bit number is specified in the "little-endian" convention, where bit *n* has a value of 2^n in the 32-bit word.

In the large majority of applications, this value will be specified properly as part of the hardware auto-detection and configuration done at re-initialization (on the **$$$***** command), either done at the factory or by the user at system assembly, or subsequently. Also, when the value of **Motor[*x*].EncType** is set through the Power PMAC script environment, whether from an on-line command, buffered program command, or from the saved value at power-on/reset, this value will be set to the standard value for the type of hardware interface (e.g. PMAC2-style, PMAC3-style, MACRO) specified by the value of **Motor[*x*].EncType**.

To use the standard amplifier-enable output signal of a PMAC2-style "DSPGATE1" Servo IC, as in an ACC-24E2, ACC-24E2A, or ACC-24E2S, **Motor[*x*].AmpEnableBit** should be set to 22. (Note the signal is mapped into bit 14 of the 24-bit word in the IC, but this word is in the high 24-bits of the 32-bit Power PMAC bus, so it appears as bit 22 to the software.)

To use the standard amplifier-enable output signal of a PMAC3-style "DSPGATE3" IC, as in an ACC-24E3 or Power PMAC "Brick", **Motor[*x*].AmpEnableBit** should be set to 8.

To use the standard amplifier-enable bit of the MACRO-ring protocol, whether through a PMAC2-style MACRO IC (as in the ACC-5E) or a PMAC3-style IC (as in the ACC-5E3) **Motor[*x*].AmpEnableBit** should be set to 22.

To use the amplifier-enable bit of an EtherCAT drive conforming to the DS-402 standard, **Motor[*x*].AmpEnableBit** should be set to 0.

Motor[*x*].AmpFaultBit

Description: Bit number of amplifier fault signal in **pAmpFault** register

Range: 0 .. 31

Units: Bit number (little-endian)

Default: Auto-configured based on hardware

Motor[*x*].AmpFaultBit determines which bit of the 32-bit register whose address is specified by **Motor[*x*].pAmpFault** Power PMAC will use to detect the amplifier-fault input for the motor. The bit number is specified in the "little-endian" convention, where bit *n* has a value of 2^n in the 32-bit word.

In the large majority of applications, this value will be specified properly as part of the hardware auto-detection and configuration done at re-initialization (on the **$$$***** command), either done at the factory or by the user at system assembly, or subsequently. Also, when the value of **Motor[*x*].EncType** is set through the Power PMAC script environment, whether from an on-line command, buffered program command, or from the saved value at power-on/reset, this value will be set to the standard value for the type of hardware interface (e.g. PMAC2-style, PMAC3-style, MACRO) specified by the value of **Motor[*x*].EncType**.

To use the standard amplifier-fault input signal of a PMAC2-style "DSPGATE1" Servo IC, as in an ACC-24E2, ACC-24E2A, or ACC-24E2S, **Motor[*x*].AmpFaultBit** should be set to 23. (Note the signal is mapped into bit 15 of the 24-bit word in the IC, but this word is in the high 24-bits of the 32-bit Power PMAC bus, so it appears as bit 23 to the software.)

To use the standard amplifier-fault input signal of a PMAC3-style "DSPGATE3" IC, as in an ACC-24E3 or Power PMAC "Brick", **Motor[*x*].AmpFaultBit** should be set to 7.

To use the standard amplifier-fault bit of the MACRO-ring protocol, whether through a PMAC2-style MACRO IC (as on the ACC-5E) or a PMAC3-style IC (as on the ACC-5E3) **Motor[*x*].AmpFaultBit** should be set to 23.

To use the amplifier-fault bit of an EtherCAT drive conforming to the DS-402 standard, **Motor[*x*].AmpFaultBit** should be set to 0.

Motor[*x*].AmpFaultLevel

Description: Amplifier fault logical state

Range: 0 .. 3

Units: Bit field

Default: 0

Motor[*x*].AmpFaultLevel specifies the fault state and the required number of detections in this state of the amplifier fault input bit that will cause an amplifier fault error.

Bit 0 (value 1) specifies the binary value of the amplifier-fault input bit that is interpreted as a fault. If it is set to 0, a 0 value is considered a fault; if it is set to 1, a 1 value is considered a fault. The amplifier fault bit is read from the register specified by **Motor[*x*].pAmpFault** at the bit number specified by **Motor[*x*].AmpFaultBit**. If **Motor[*x*].pAmpFault** is set to 0 to disable the amplifier fault function, the setting **Motor[*x*].AmpFaultLevel** does not matter.

The voltage level that creates a 0 or 1 value is dependent on the nature of the interface circuitry. Many Power PMAC amplifier-fault circuits employ bi-directional opto-couplers. In these circuits, current flowing in either direction through the opto-coupler in either direction will result in a 0 value.

The MACRO protocol simply passes the value of the amplifier-fault bit from the drive back to the Power PMAC, so the proper value of this bit of **Motor[*x*].AmpFaultLevel** is dependent on the individual drive.

For an EtherCAT drive conforming to the DS-402 standard, the fault bit is high-true, so bit 0 of **Motor[*x*].AmpFaultLevel** should be set to 1.

Bit 1 (value 2) specifies the number of consecutive scans the Power PMAC must detect the amplifier fault bit to be in its specified fault state to cause an amplifier fault trip that "kills" the motor. If it is set to 0, a single scan detecting the fault state will cause a trip. If it is set to 1, two consecutive scans (in adjacent real-time interrupt periods) must detect the fault state to cause a trip. A single detection will set and latch the **Motor[*x*].AmpWarn** status bit.

It is generally recommended to set this bit to 1 to prevent nuisance trips due to electrical noise, or to delayed clearing of the fault status on re-enabling a drive. The delay in reaction to a real fault is not significant in the vast majority of applications.

Motor[*x*].AmpFaultLevel constitutes bits 8 and 9 of the full-word element **Motor[*x*].Control[0]**.

Motor[*x*].AuxFaultBit

Description: Bit number of sensor-loss flag in **pAuxFault** register

Range: 0 .. 31

Units: Bit number (little-endian)

Default: 0

Motor[*x*].AuxFaultBit determines which bit of the 32-bit register whose address is specified by **Motor[*x*].pAuxFault** Power PMAC will use to detect the auxiliary fault for the motor. The bit number is specified in the "little-endian" convention, where bit *n* has a value of 2^n in the 32-bit word. The polarity of this auxiliary-fault input bit is set by **Motor[*x*].AuxFaultLevel**.

For motor thermal sensors connected Power Brick PMAC3-style interface boards, which have processing circuitry on the "T" flag input, **Motor[*x*].AuxFaultBit** should be set to 15, as the **T** flag input bit is found in bit 15 of the register.

For quadrature encoders connected to ACC-24E2*x* PMAC2-style axis-interface boards, which have "exclusive-OR" quadrature loss detection circuits, **Motor[*x*].AuxFaultBit** should be set to 13, as the **Gate1[*i*].Chan[*j*].EncLossN** status bit is found in bit 13 of the register.

For quadrature encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which also have "exclusive-OR" quadrature loss detection circuits, **Motor[*x*].AuxFaultBit** should be set to 28, as the **Gate3[*i*].Chan[*j*].LossStatus** bit is found in bit 28 of the register.

For analog sinusoidal encoders or resolvers connected to ACC-24E3 PMAC3-style axis-interface boards, which have "sum-of-squares" loss detection circuits, **Motor[*x*].AuxFaultBit** should be set to 31, as the **Gate3[*i*].Chan[*j*].SosError** bit is found in bit 31 of the register.

For serial encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which have several protocol-dependent loss-detection circuits, **Motor[*x*].AuxFaultBit** should usually be set to 31, as the channel's "time-out" error bit is found in bit 31 of the **Gate3[*i*].Chan[*j*].SerialEncDataB** register, although this may differ depending on the protocol.

If **Motor[*x*].LimitBits** is set to a value in the range of 96 to 127, or 224 to 255, **Motor[*x*].AuxFaultBit** specifies the bit in the register addressed by **Motor[*x*].pAuxFault** that is used as the negative hardware overtravel limit input bit. This functionality is new in V2.1 firmware, released 1st quarter 2016.

The Type = 12 encoder conversion table entry (new in V2.0 firmware, released 1st quarter 2015) can monitor each servo cycle for multiple error bits in a single register. If any of these bits is set, the entry sets bit 0 of **EncTable[*m*].Status** to 1, and this bit can be used as the auxiliary-fault bit for the motor.

Motor[*x*].AuxFaultLevel

Description: Auxiliary fault logical state

Range: 0 .. 1

Units: Boolean

Default: 1

Motor[*x*].AuxFaultLevel specifies the loss state of the auxiliary-fault input bit that will cause an auxiliary fault error. The required number of detections in this state is specified by **Motor[*x*].AuxFaultLimit**.

If **Motor[*x*].AuxFaultLevel** is set to 0, a 0 value is considered a fault; if it is set to 1, a 1 value is considered a fault. The auxiliary fault bit is read from the register specified by **Motor[*x*].pAuxFault** at the bit number specified by **Motor[*x*].AuxFaultBit**. If **Motor[*x*].pAuxFault** is set to 0 to disable the auxiliary-fault detection function, the setting **Motor[*x*].AuxFaultLevel** does not matter.

For motor thermal sensors connected Power Brick PMAC3-style interface boards, which have processing circuitry on the "T" flag input, **Motor[*x*].AuxFaultLevel** should be set to 1.

For quadrature encoders connected to ACC-24E2*x* PMAC2-style axis-interface boards, which have "exclusive-OR" quadrature loss detection circuits, **Motor[*x*].AuxFaultLevel** should be set to 0.

For quadrature encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which also have "exclusive-OR" quadrature loss detection circuits, **Motor[*x*].AuxFaultLevel** should be set to 1.

For analog sinusoidal encoders or resolvers connected to ACC-24E3 PMAC3-style axis-interface boards, which have "sum-of-squares" loss detection circuits, **Motor[*x*].AuxFaultLevel** should be set to 1.

For serial encoders connected to ACC-24E3 PMAC3-style axis-interface boards or to the Power Brick control boards, which have several protocol-dependent loss-detection circuits, **Motor[*x*].AuxFaultLevel** should be set to 1.

If **Motor[*x*].LimitBits** is set to a value in the range of 96 to 127 or 224 to 255, the auxiliary fault function is used for the negative hardware overtravel limit, and **Motor[*x*].AuxFaultLevel** is not used. The polarity of the limit input is instead determined by bit 7 (value 128) of **Motor[*x*].LimitBits**. This functionality is new in V2.1 firmware, released 1st quarter 2016.

Motor[*x*].AuxFaultLimit

Description: Auxiliary fault maximum accumulated number of fault detections

Range: 0 .. 255

Units: Scans

Default: 0

Motor[*x*].AuxFaultLimit specifies the maximum number of accumulated scans the Power PMAC can find the auxiliary fault input in its "fault" state without tripping on an auxiliary fault error. If the number of accumulated scans in the fault state exceeds this value, Power PMAC will automatically "kill" this motor, and kill or abort other motors in the same coordinate system, depending on the setting of **Motor[*x*].FaultMode**. Power PMAC checks the state of the specified encoder-loss bit every real-time interrupt period.

Many of the auxiliary-fault detection circuits can temporarily sense a fault due to electrical noise or similar factors. The ability to confirm a true loss by requiring multiple consecutive scans detecting the loss can be valuable in eliminating false trips.

The register for the auxiliary-fault bit is specified by **Motor[*x*].pAuxFault**. The bit within this register is specified by **Motor[*x*].AuxFaultBit**. The loss state of this bit is specified by **Motor[*x*].AuxFaultLevel**. If **Motor[*x*].pAuxFault** is set to 0 to disable the auxiliary-fault detection function, the setting of **Motor[*x*].AuxFaultLimit** does not matter. The accumulated number of scans with the specified bit found in the "fault" state is found in status element **Motor[*x*].AuxFaultCount**. This bit is incremented each time the bit is found in the fault state, and decremented (if greater than zero) each time the bit is not found in the fault state.

If **Motor[*x*].LimitBits** is set to a value in the range of 96 to 127 or 224 to 255, the auxiliary fault function is used for the negative hardware overtravel limit, and **Motor[*x*].AuxFaultLimit** is not used. A single scan with the input bit in the fault state will trigger the specified action for hitting a hardware overtravel limit. This functionality is new in V2.1 firmware, released 1st quarter 2016.

Motor[*x*].BlHysteresis

Description: Backlash hysteresis size

Range: floating-point

Units: Motor units

Default: 0.0

Legacy I-variable alias: I*x*87

Motor[*x*].BlHysteresis specifies the magnitude of the direction reversal in motor commanded position that must occur before Power PMAC starts to introduce backlash compensation in the new direction. The purpose of this variable is to allow the user to ensure that a very small direction reversal (e.g. from the dithering of a master encoder) does not cause the backlash compensation to start instantly.

Motor[*x*].BlSize

Description: Backlash size

Range: floating-point

Units: Motor units

Default: 0.0

Legacy I-variable alias: I*x*86

Motor[*x*].BlSize specifies the size of the correction distance Power PMAC uses to compensate for backlash in the motor's gearing. On reversal of commanded direction, this distance is added into or subtracted from the raw measured position. This offset will not appear when actual position is queried. The rate at which backlash is added or subtracted ("taken up") is determined by **Motor[*x*].BlSlewRate**. Note that backlash compensation will not actually occur unless **Motor[*x*].BlSlewRate** is set to a positive value (its default is 0.0).

Motor[x].BlSlewRate

Description: Backlash take-up rate

Range: floating-point

Units: Motor units per real-time interrupt

Default: 0.0

Legacy I-variable alias: I*x*85

Motor[*x*].BlSlewRate specifies how fast backlash is "taken up" on direction reversal. The magnitude of the backlash is determined by **Motor[*x*].BlSize**. When a reversal in commanded direction larger than **Motor[*x*].BlHysteresis** is detected, then each real-time interrupt, Power PMAC will add in or take out an amount equal to **Motor[*x*].BlSlewRate** until the total backlash magnitude is compensated.

Motor[*x*].BlSlewRate is usually set interactively and experimentally to as high a value as possible without creating dynamic problems. The default value of 0.0 effectively disables backlash compensation.

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*].BlSlewRate** 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*].BlSlewRate** if **Sys.MotorsPerRtInt** > 0 is given by the equation:

$$N = ceil\left(\frac{Sys.ServoMotors + 1}{Sys.MotorsPerRtInt}\right)$$

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*].BlSlewRate** should be multiplied by *N* compared to cases where motor status updates are performed every real-time interrupt.

Motor[x].BrakeOffDelay

Description: Delay time after enabling for brake release

Range: Non-negative floating-point

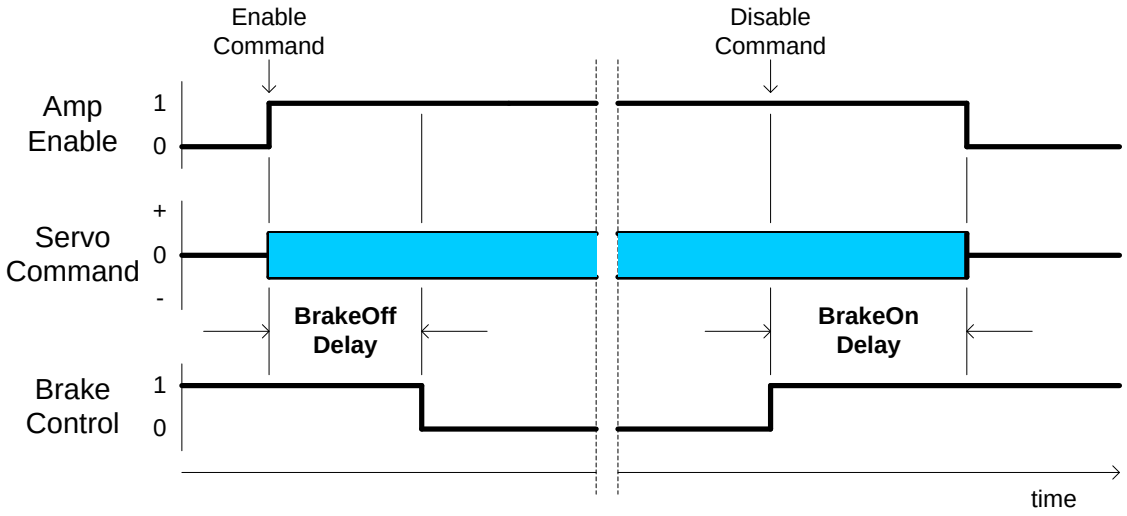
Units: Milliseconds

Default: 0.0

Motor[*x*].BrakeOffDelay specifies the delay from the time the motor is enabled until the brake is automatically released. It is only used if the automatic brake-control function is enabled for the motor by setting **Motor[*x*].pBrakeOut** to a non-zero value.

When a motor with the automatic brake-control function active is enabled by a command such as **`j/`**, **`outn`**, or **`enable`**, then the brake is released **BrakeOffDelay** milliseconds later by setting the specified brake-control output bit to 1. This delay gives the system time to achieve full control of the motor before the brake is released, preventing momentary movement (usually a vertical drop) before such control is established.

This diagram shows the timing of the brake control output relative to the enabling and (delayed) disabling of the motor.



If the automatic brake-control function is enabled, the user should not attempt to enable a disabled motor with an actual motor move command such as **`j+`** or **`hm`**.

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*].BrakeOffDelay** 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*].BrakeOffDelay** if **Sys.MotorsPerRtInt** > 0 is given by the equation:

$$N = ceil\left(\frac{Sys.ServoMotors + 1}{Sys.MotorsPerRtInt}\right)$$

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*].BrakeOffDelay** should be divided by *N* compared to cases where motor status updates are performed every real-time interrupt.

Motor[x].BrakeOnDelay

Description: Delay time after brake engage for disabling

Range: Non-negative floating-point

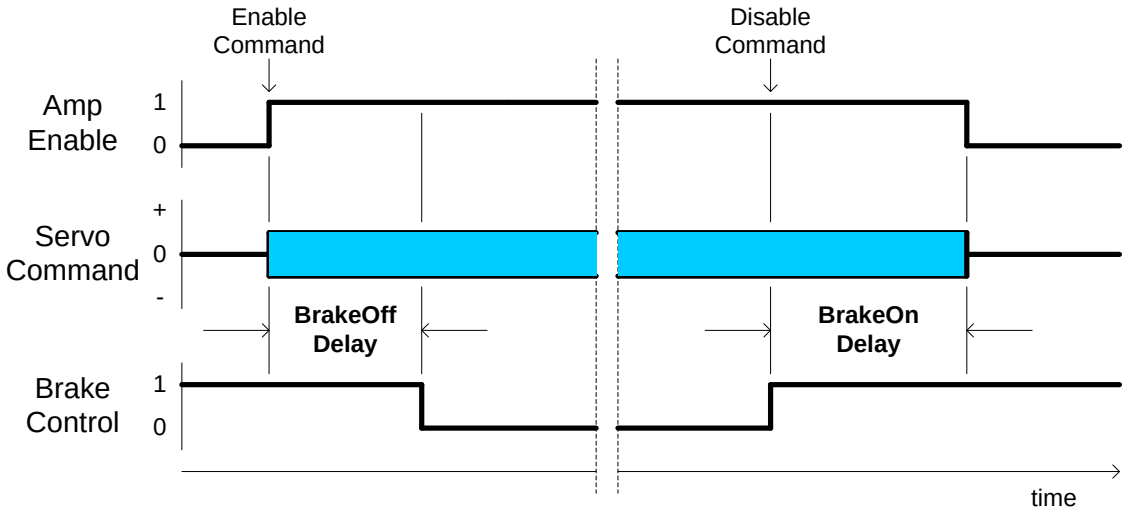
Units: Milliseconds

Default: 0.0

Motor[*x*].BrakeOnDelay specifies the delay from the time the brake is engaged on a "delayed disable" command until the motor is disabled. It is only used if the automatic brake-control function is enabled for the motor by setting **Motor[*x*].pBrakeOut** to a non-zero value.

When a motor with the automatic brake-control function active in a closed-loop zero-velocity state and disabled by a special "delayed disable" command, the brake is engaged immediately by setting the specified brake-control output bit to 0, but the motor control is not disabled until **BrakeOnDelay** milliseconds later. This delay gives the brake hardware time to fully engage before control is disabled, preventing momentary movement (usually a vertical drop) before the brake is fully engage.

This diagram shows the timing of the brake control output relative to the enabling and (delayed) disabling of the motor.



This delay is used on the motor command **dkill** and the coordinate-system command **ddisable**, provided that the motors are in the closed-loop zero-velocity-state. These commands can be used in either on-line form (e.g. **#1dkill**, **&2ddisable**) or buffered program form (e.g. **dkill1**, **ddisable2**).

If the standard disabling commands (**k**, **kill**, **disable**) are used, if the motor is disabled automatically due to a problem such as amplifier fault or fatal following error, or if the motor is not in the closed-loop zero-velocity state, then the motor is disabled immediately, and the same time the brake output is engaged; there is no delay.

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*].BrakeOnDelay** 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*].BrakeOnDelay** if **Sys.MotorsPerRtInt** > 0 is given by the equation:

$$N = ceil\left(\frac{Sys.ServoMotors + 1}{Sys.MotorsPerRtInt}\right)$$

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*].BrakeOnDelay** should be divided by *N* compared to cases where motor status updates are performed every real-time interrupt.

Motor[*x*].BrakeOutBit

Description: Bit number of brake output signal in **pBrakeOut** register

Range: 0 .. 31

Units: Bit number (little-endian)

Default: 9

Motor[*x*].BrakeOutBit determines which bit of the 32-bit register whose address is specified by **Motor[*x*].pBrakeOut** Power PMAC will use to control the brake-control output for the motor. The bit number is specified in the "little-endian" convention, where bit *n* has a value of 2^n in the 32-bit word.

To use the output flag "B" of a PMAC3-style "DSPGATE3" IC, as in a Power PMAC Brick, **Motor[*x*].BrakeOutBit** should be set to 9.

To use a general-purpose output on a UMAC I/O board, such as an ACC-65E, ACC-68E, or ACC-11E, **Motor[*x*].BrakeOut** should be set to a value of 8 to 15, depending on the particular output used.

Motor[*x*].CaptControl

Description: Motor full-word element for captured position processing

Range: $00000000 .. $FFFFFFFF

Units: Bit field

Default: $00000000

Motor[*x*].CaptControl is the full-word element that comprises the setup elements for processing capture flags and captured positions in triggered moves such as homing-search moves

Motor[*x*].CaptControl contains the following partial-word elements:

Component	Bits	Functionality
CaptFlagBit	31 – 24	Bit number for capture flag
CaptPosRightShift	23 – 16	Number of bits to shift captured position right
CaptPosLeftShift	15 – 08	Number of bits to shift capture position left
--	07 – 01	(*reserved for future use*)
CaptPosRound	00	Half-count offset control for captured position

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even if the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Motor[x].CaptEnaBit

Description: Encoder capture trigger arming bit number

Range: 0 .. 31

Units: Bit number

Default: 0

Motor[*x*].CaptEnaBit specifies which bit of the 32-bit register whose address is specified by **Motor[*x*].pCaptEna** Power PMAC will use to arm the position capture trigger for a triggered move (homing-search move, jog-until-trigger, programmed rapid-mode move-until-trigger). This function permits the manual arming of the capture trigger at this address.

This write operation is only performed if **Motor[*x*].pCaptEna** is set to the address of a valid element (not to the default value of 0). It is commonly used to set up the "touch probe" control word ($60B8_h$) of an EtherCAT drive. In the EtherCAT standard, setting bit 0 of this control word to 1 enables "touch probe 1". Setting bit 8 of this control word enables "touch probe 2".

No user-specified arming bit is required to arm the capture trigger when using a Delta Tau "DSPGATE*n*" ASIC or standard MACRO-ring device for the trigger.

Motor[*x*].CaptEnaBit is new in V2.1 firmware, released 1st quarter 2016.

Motor[x].CaptEnaInvert

Description: Encoder capture trigger arming invert control

Range: 0 .. 1

Units: Boolean

Default: 0

Motor[*x*].CaptEnaInvert specifies the polarity of the trigger-arming bit in the 32-bit register whose address is specified by **Motor[*x*].pCaptEna** and whose bit number is specified by **Motor[*x*].CaptEnaBit**.

If **Motor[*x*].CaptEnaInvert** is set to the default value of 0, Power PMAC will write a 1 to this bit to arm the trigger at the start of a triggered move (homing-search move, jog-until-trigger, programmed rapid-mode move-until-trigger). If it is set to 1, Power PMAC will write a 0 to this bit at the start of a triggered move.

When the trigger is found, or when the move ends without finding a trigger, Power PMAC will automatically set this bit to the opposite value.

This write operation is only performed if **Motor[*x*].pCaptEna** is set to the address of a valid element (not to the default value of 0). It is commonly used to set up the "touch probe" control word ($60BB_h$) of an EtherCAT drive. In the EtherCAT standard, setting bit 0 of this control word to 1 enables "touch probe 1". Setting bit 8 of this control word to 1 enables "touch probe 2". In both cases, **Motor[*x*].CaptEnaInvert** should be set to 0.

Motor[*x*].CaptEnaInvert is new in V2.1 firmware, released 1st quarter 2016.

Motor[x].CaptFlagBit

Description: Bit number of capture flag in **pCaptFlag** register

Range: 0 .. 31

Units: Bit number (little-endian)

Default: Auto-configured based on hardware

Motor[*x*].CaptFlagBit determines which bit of the 32-bit register whose address is specified by **Motor[*x*].pCaptFlag** Power PMAC will use to detect the capture-trigger flag for the motor. The bit number is specified in the "little-endian" convention, where bit *n* has a value of 2^n in the 32-bit word.

In the large majority of applications, this value will be specified properly as part of the hardware auto-detection and configuration done at re-initialization (on the **$$$***** command), either done at the factory or by the user at system assembly, or subsequently. Also, when the value of **Motor[*x*].EncType** is set through the Power PMAC script environment, whether from an on-line command, buffered program command, or from the saved value at power-on/reset, this value will

be set to the standard value for the type of hardware interface (e.g. PMAC2-style, PMAC3-style, MACRO) specified by the value of **Motor[*x*].EncType**.

To use the standard capture-trigger flag of a PMAC2-style "DSPGATE1" Servo IC, as in an ACC-24E2, ACC-24E2A, or ACC-24E2S, **Motor[*x*].CaptFlagBit** should be set to 19. (Note the signal is mapped into bit 11 of the 24-bit word in the IC, but this word is in the high 24-bits of the 32-bit Power PMAC bus, so it appears as bit 19 to the software.)

To use the standard capture-trigger flag of a PMAC3-style "DSPGATE3" IC, as in an ACC-24E3 or Power PMAC "Brick", **Motor[*x*].CaptFlagBit** should be set to 20.

To use the standard capture-trigger flag of the MACRO-ring protocol, whether through a PMAC2-style MACRO IC (as on the ACC-5E) or a PMAC3-style IC (as on the ACC-5E3) **Motor[*x*].CaptFlagBit** should be set to 19.

To use a standard "probe latch" flag of an EtherCAT drive conforming to the DS-402 standard, **Motor[*x*].CaptFlagBit** should be set to 1 to use the first probe, or to 9 to use the second probe.

Motor[*x*].CaptFlagInvert

Description: Motor capture flag invert control

Range: 0 .. 1

Units: Boolean

Default: 0

Motor[*x*].CaptFlagInvert controls whether the motor capture trigger bit specified by **Motor[*x*].pCaptFlag** and **Motor[*x*].CaptFlagBit** is inverted or not. This bit is used for triggered moves such as homing search moves, motor jog-until-trigger moves, and programmed rapid-mode triggered moves.

If **Motor[*x*].CaptFlagInvert** is set to its default value of 0, then the 0-to-1 transition of the specified bit is considered the trigger condition. This setting should be used whenever any Delta Tau servo-interface ASIC (**Gate1[*i*]** or **Gate3[*i*]**) flags and/or encoder index inputs are used for the trigger, because the capture trigger status bit changes from 0 to 1 on the specified trigger condition, regardless of whether the rising or falling edge of the input signal(s) is specified to cause the trigger.

If **Motor[*x*].CaptFlagInvert** is set to 1, then the 1-to-0 transition of the specified bit is considered the trigger condition. This permits the use of general-purpose inputs or network-transmitted input bits as the trigger condition. (In these cases, **Motor[*x*].CaptureMode** should be set to 1 to specify software capture of the motor position when the trigger is found, because these inputs do not support hardware trigger features.)

However, if **Motor[*x*].CaptToggle** is set to 1, **Motor[*x*].CaptFlagInvert** is not used as a setup element. In this case, the first transition of the trigger-flag bit after the start of the move, whichever one it is, is considered to be the trigger. For these moves, **Motor[*x*].CaptFlagInvert** is simply used to store the starting state of trigger-flag bit so a change can be detected.

Motor[*x*].CaptFlagInvert is new in V2.1 firmware, released 1st quarter 2016. At its default value of 0, operation is backward compatible with older firmware versions.

Motor[*x*].CaptPosLeftShift

Description: Number of bits to shift captured position data left

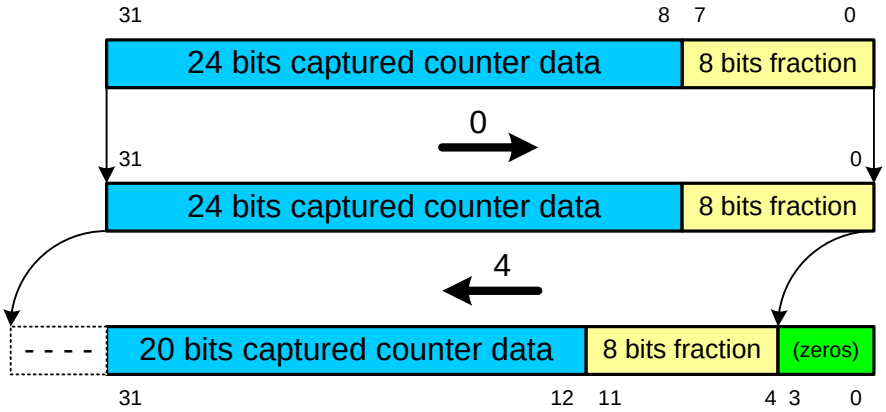
Range: 0 .. 31

Units: Bits

Default: Auto-configured based on hardware

Motor[*x*].CaptPosLeftShift determines the number of bits Power PMAC will shift the hardware-captured position from the 32-bit register whose address is specified by **Motor[*x*].pCaptPos** to the left after it has first been shifted right as specified by **Motor[*x*].CaptPosRightShift**. The purpose of this operation is to create a value with the same resolution as ongoing position-loop feedback data specified by **Motor[*x*].pEnc** through its conversion table entry.

This diagram shows an example of a zero-bit right shift followed by a 4-bit left shift.



In the large majority of applications, this value will be specified properly as part of the hardware auto-detection and configuration done at re-initialization (on the **$$$***** command), either done at the factory or by the user at system assembly, or subsequently. Also, when the value of **Motor[*x*].EncType** is set through the Power PMAC script environment, whether from an on-line command, buffered program command, or from the saved value at power-on/reset, this value will be set to the standard value for the type of hardware interface (e.g. PMAC2-style, PMAC3-style, MACRO) specified by the value of **Motor[*x*].EncType**.

To use the standard 24-bit whole-count captured-position register of a PMAC2-style "DSPGATE1" Servo IC, when the servo-loop position feedback is a digital quadrature encoder with 1/T extension, as in an ACC-24E2, ACC-24E2A, or ACC-24E2S, **Motor[*x*].CaptPosLeftShift** should be set to 9, because the servo feedback has 9 bits of fractional-count extension.

To use the standard 24-bit whole-count captured-position register of a PMAC2-style "DSPGATE1" Servo IC, when the servo-loop position feedback is a sinusoidal encoder with arctangent extension, as in an ACC-51E, **Motor[*x*].CaptPosLeftShift** should be set to 10, because the servo feedback has 10 bits of fractional-count extension.

To use the standard 24-bit whole-count captured-position register of a PMAC2-style "DSPGATE1" Servo IC, when the servo-loop position feedback is processed in the phase interrupt (**Motor[*x*].PhaseCtrl** = 8) using the IC's 24-bit whole-count "phase capture" register without any sub-count extension, **Motor[*x*].CaptPosLeftShift** should be set to 8, because a "count" of servo feedback is found in bit 8 of the 32-bit Power PMAC bus in this mode.

To use just the 24-bit whole-count portion of the 32-bit capture-position register of a PMAC3-style "DSPGATE3" IC, as in an ACC-24E3 or Power PMAC "Brick", when the servo feedback comes from a quadrature encoder with 1/T extension, **Motor[*x*].CaptPosLeftShift** should be set to 8, because the servo feedback has 8 bits of fractional count extension in this mode.

To use just the 24-bit whole-count portion of the 32-bit capture-position register of a PMAC3-style "DSPGATE3" IC, as in an ACC-24E3 or Power PMAC "Brick", when the servo feedback comes from a sinusoidal encoder with arctangent extension, **Motor[*x*].CaptPosLeftShift** should be set to 12, because the servo feedback has 12 bits of fractional count extension in this mode.

To use the entire 32-bit capture-position register of a PMAC3-style "DSPGATE3" IC, including the 1/T fractional count estimation in the low 8 bits, when the servo feedback comes from a quadrature encoder with 1/T extension, **Motor[*x*].CaptPosLeftShift** should be set to 0, because the servo feedback also has 8 bits of fractional count extension in this mode.

To use the entire 32-bit capture-position register of a PMAC3-style "DSPGATE3" IC, including the 1/T fractional count estimation in the low 8 bits, when the servo feedback comes from a sinusoidal encoder with arctangent extension, **Motor[*x*].CaptPosLeftShift** should be set to 4, because the servo feedback has 12 bits of fractional count extension in this mode.

To use the captured position data sent back over the MACRO ring, **Motor[*x*].CaptPosLeftShift** should be set to 13, because the servo loop feedback has whole-count data in bit 13.

To use the 32-bit "probe latched" position from an EtherCAT drive conforming to the DS-402 standard, **Motor[*x*].CaptPosLeftShift** should be set to 0.

Motor[*x*].CaptPosRightShift

Description: Number of bits to shift captured position register right

Range: 0 .. 31

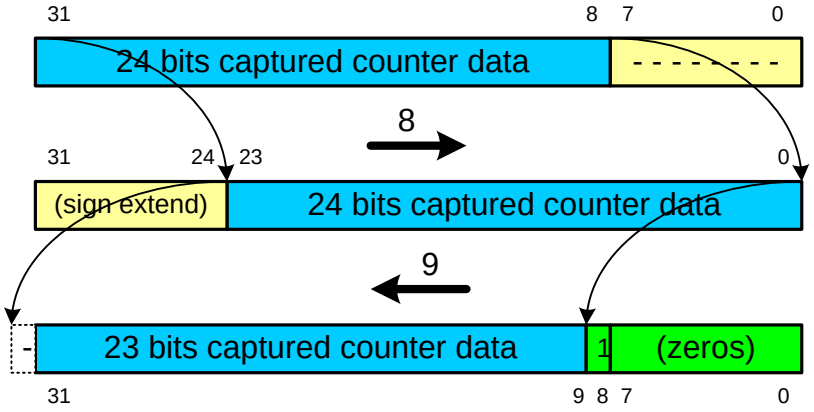
Units: Bits

Default: Auto-configured based on hardware

Motor[*x*].CaptPosRightShift determines the number of bits Power PMAC will shift the hardware-captured position from the 32-bit register whose address is specified by **Motor[*x*].pCaptPos** to the right as an initial processing step. The purpose of this operation is to

leave the least significant bit of the actual data in the 32-bit register in bit 0 (little-endian) of the resulting value, eliminating possible "garbage data" in the low bits of the 32-bit source register, so **Motor[*x*].CaptPosRightShift** is typically set to the bit number of the LSB of actual data in the 32-bit source word.

This diagram shows an example of a 8-bit right shift followed by a 9-bit left shift.



In the large majority of applications, this value will be specified properly as part of the hardware auto-detection and configuration done at re-initialization (on the **$$$***** command), either done at the factory or by the user at system assembly, or subsequently. Also, when the value of **Motor[*x*].EncType** is set through the Power PMAC script environment, whether from an on-line command, buffered program command, or from the saved value at power-on/reset, this value will be set to the standard value for the type of hardware interface (e.g. PMAC2-style, PMAC3-style, MACRO) specified by the value of **Motor[*x*].EncType**.

To use the standard 24-bit captured-position register of a PMAC2-style "DSPGATE1" Servo IC, as in an ACC-24E2, ACC-24E2A, ACC-24E2S, or ACC-51E, **Motor[*x*].CaptPosRightShift** should be set to 8, because the low 8 bits on the 32-bit Power PMAC bus are of indeterminate value.

To use just the 24-bit whole-count portion of the 32-bit capture-position register of a PMAC3-style "DSPGATE3" IC, as in an ACC-24E3 or Power PMAC "Brick", **Motor[*x*].CaptPosRightShift** should be set to 8 to eliminate any possible 1/T fractional count estimate value in the low 8 bits. This setting is commonly used for captures in homing-search moves for both digital-quadrature and sinusoidal encoders, as most users want to use a whole-count edge for the home-capture position.

To use the entire 32-bit capture-position register of a PMAC3-style "DSPGATE3" IC, including the 1/T fractional count estimation in the low 8 bits, **Motor[*x*].CaptPosRightShift** should be set to 0 because the data includes the LSB of the 32-bit register. This setting is commonly used for captures in probing and registration for both digital-quadrature and sinusoidal encoders, as most users want the highest possible resolution for these captures.

To use the captured position data sent back over the MACRO ring, **Motor[*x*].CaptPosRightShift** should be set to 0 because the data includes the LSB of the returned data register.

To use the 32-bit "probe latched" position from an EtherCAT drive conforming to the DS-402 standard, **Motor[*x*].CaptPosRigthShift** should be set to 0.

Motor[*x*].CaptPosRound

Description: Captured position round-off control

Range: 0 .. 1

Units: Boolean

Default: Auto-configured based on hardware

Motor[*x*].CaptPosRound determines whether Power PMAC offsets the captured position value in triggered moves by a half count or not. If it is set to 0, Power PMAC does not perform this offset; if it is set to 1, Power PMAC does perform this offset.

In the large majority of applications, this value will be specified properly as part of the hardware auto-detection and configuration done at re-initialization (on the **$$$***** command), either done at the factory or by the user at system assembly, or subsequently. Also, when the value of **Motor[*x*].EncType** is set through the Power PMAC script environment, whether from an on-line command, buffered program command, or from the saved value at power-on/reset, this value will be set to the standard value for the type of hardware interface (e.g. PMAC2-style, PMAC3-style, MACRO) specified by the value of **Motor[*x*].EncType**.

This offset should be performed if the servo-loop feedback position for the motor does use sub-count extension, but the trigger captured position does not. This is because the sub-count extension automatically includes a half-count offset to put the integer count values halfway between the whole-count edges.

If both the servo-loop feedback position and the trigger captured position use sub-count extension, or both do not, the offset should not be performed.

Motor[x].CaptToggle

Description: Encoder capture trigger pre-arming word

Range: 0 .. 1

Units: Boolean

Default: 0

Motor[*x*].CaptToggle specifies whether the capture trigger for a triggered move (homing-search move, jog-until-trigger, programmed rapid-mode move-until-trigger) is defined to be a particular edge of the bit specified by **Motor[*x*].pCaptFlag** and **Motor[*x*].CaptFlagBit**, or it is the first edge to be seen after the start of the move.

If **Motor[*x*].CaptToggle** is set to its default value of 0, the edge that causes the trigger is specified by **Motor[*x*].CaptFlagInvert**. With that element at its default value of 0, the 0-to-1 transition of the input bit is considered to be the trigger. This is compatible with older versions of the firmware that do not have these new elements.

If **Motor[*x*].CaptToggle** is set to 1, Power PMAC reads the state of the specified trigger-flag bit at the start of a triggered move, storing this value in **Motor[*x*].CaptFlagInvert**. When the flag bit changes to the opposite state, this transition is considered to be the trigger. In this case, the capture-flag bit can be the least significant bit of a capture-event counter.

Motor[*x*].CaptToggle is new in V2.1 firmware, released 1st quarter 2016. At its default value of 0, operation is backward compatible with older firmware versions.

Motor[*x*].CaptureMode

Description: Triggered move position-capture mode

Range: 0 .. 3

Units: none

Default: 0

Legacy I-variable alias: I*x*97

Motor[*x*].CaptureMode specifies how the position-capture function works for the motor in triggered moves – homing search moves, jog-until-trigger moves, and motion-program go-until-trigger moves.

If **Motor[*x*].CaptureMode** is set to its default value of 0, the capture trigger will come from an input trigger (flag and/or index-channel change) on the ASIC channel specified by **Motor[*x*].pCaptFlag.** The captured position will be taken from the register whose address is specified by **Motor[*x*].pCaptPos**, which should be the "home capture" encoder register of the same ASIC channel. This register is latched immediately on the trigger condition, so is exact, but requires that the motor's actual position be derived from a sensor processed through this ASIC channel's encoder counter.

If **Motor[*x*].CaptureMode** is set to 1, the capture trigger will come from an input trigger (flag and/or index-channel change) on the ASIC channel specified by **Motor[*x*].pCaptFlag.** The captured position will be taken from the motor's present "actual position" register, regardless of the source of the position value. This "software capture" technique can work with any type of position sensor, but has a delay and uncertainty in the capture of up to a servo cycle.

If **Motor[*x*].CaptureMode** is set to 2, the capture trigger will come from the fact of the motor's following error exceeding its warning limit set by **Motor[*x*].WarnFeLimit**. The captured position will be taken from the motor's "actual position" register, regardless of the source of the position value. This mode is usually used for "torque-limit" triggering such as homing into a hard stop, or screwing in until cinched.

If **Motor[*x*].CaptureMode** is set to 3, the capture trigger will come from an input trigger (flag and/or index-channel change) on the PMAC3-style ASIC channel specified by **Motor[*x*].pCaptFlag.** The captured position will be derived from the motor's "actual position" history, interpolated between two of those values based on time value latched by the trigger. This "timer-assisted software capture" technique can achieve accuracy close to that of the "hardware capture" technique, even for feedback not processed through the channel's encoder counter.

For this timer-assisted software capture technique to work, the channel's PFM output must be configured to generated 16 pulses per servo cycle by the proper setting of **Gate3[*i*].Chan[*j*].Pfm**. This pulse train must be fed into the channel's counter by setting **Gate3[*i*].Chan[*j*].EncCtrl** to 8. Hardware 1/T extension of the pulse counting must be enabled by setting **Gate3[*i*].Chan[*j*].TimerMode** to the default value of 0. **Gate3[*i*].Chan[*j*].PackOutData** must be set to the default value of 0 so that the PFM command is in a separate register. **Motor[*x*].pCaptPos** must be set to **Gate3[*i*].Chan[*j*].HomeCapt.a** to use the latched value of the pulse counter for the timing.

Motor[*x*].CaptureMode constitutes bits 16 – 17 of the full-word element **Motor[*x*].Control[0]**.

Motor[x].CascadeMode

Description: Cascaded servo command integration mode

Range: 0 .. 1

Units: Boolean

Default: 0 (integration disabled)

Motor[*x*].CascadeMode specifies how the cascaded servo command value from the motor is written to the register addressed by **Motor[*x*].pCascadeCmd**. If **Motor[*x*].CascadeMode** is set to its default value of 0, the computed value is simply written to the register, overwriting any value that was already there.

However, if **Motor[*x*].CascadeMode** is set to 1, the computed value is added to the value already in the target register. This provides an effective numerical integration in the act of cascading, permitting the outer loop to command the inner loop indefinitely, as in a web tensioning application.

Motor[*x*].CascadeMode is only used if **Motor[*x*].pCascadeCmd** is set to a valid address.

Motor[*x*].CmdMotor

Description: Gantry leader motor number in gantry following mode

Range: 0 .. **Sys.MaxMotors** - 1

Units: Power PMAC motor numbers

Default: 0

Motor[*x*].CmdMotor specifies which motor is the "gantry leader" for this motor when this motor is in "gantry following" mode. It is only used if **Motor[*x*].ServoCtrl** for this motor is set to 8 to enable the special "gantry following" mode.

In gantry following mode, this motor will always use the commanded trajectory calculated by the motor specified in **Motor[*x*].CmdMotor** instead of calculating its own trajectory. In this way, multiple motors can use the same command trajectory without it needing to be calculated multiple times. Note that each motor still closes its own servo loop independently based on its own feedback, and performs any specified phase-interrupt tasks (commutation and current-loop closure) independently.

Motor[*x*].Control[*i*]

Description: Motor full-word element(s) for enabling elements

Range: $00000000 .. $FFFFFFFF

Units: Bit field

Default: $00000000

Motor[*x*].Control[*i*] is the array of full-word elements that comprises the setup elements for enabling the basic functionality of the motor.

Motor[*x*].Control[0] contains the following partial-word elements:

Component	Bits	Functionality
ServoCtrl	31 – 28	Control flags to enable servo tasks
PhaseCtrl	27 – 24	Control flags to enable phase commutation tasks
MasterCtrl	23 – 20	Control flags to enable master following tasks
CaptureMode	19 – 16	Triggered move position-capture mode
RapidSpeedSel	15	Rapid move mode speed select
PowerOnMode	14 – 12	Power-on/reset action control
PhaseSplineCtrl	11 – 10	Phase interrupt spline interpolation order
AmpFaultLevel	09 – 08	Amplifier fault logical state
PhaseMode	07 – 04	Phase commutation control mode flags
FaultMode	03 – 01	Fault condition action control
--	00	(*reserved for future use*)

Motor[*x*].Control[1] contains the following partial-word elements:

Component	Bits	Functionality
DacShift	31 – 27	Command output value shift-right distance
PhaseEncRightShift	26 – 22	# of bits to right shift phase position value
PhaseEncLeftShift	21 – 17	# of bits to left shift phase position value
EncLossLevel	16	Fault state of sensor loss bit
--	15 – 00	(*reserved for future use*)

Users will generally access these partial-word elements through the script environment for setting and checking the setup. Power PMAC will save and restore the full-word element for efficiency.

If the value of the full-word element is queried in the script environment, Power PMAC will return the value in hexadecimal format, even though the partial-word element values are reported in decimal.

Refer to the entry for each partial-word element for a detailed description of that element's function.

Motor[*x*].Ctrl

Description: Pointer to selected servo loop algorithm

Range: Legitimate addresses

Units: Power PMAC memory addresses

Default: **Sys.ServoCtrl**

Motor[*x*].Ctrl determines which servo algorithm is used for the specified motor by containing the memory address of that algorithm's code. If one the standard algorithms is used, it can be set using a system address constant (the numerical value of the address does not need to be known).

If **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** (an address), then this motor will use the very basic PID servo algorithm, which executes extremely quickly.

If it is set to **Sys.ServoCtrl** (another address), it will use the standard servo algorithm, which is a superset of the basic PID algorithm, adding many useful features. This is the default setting.

If it is set to **Sys.LegacyCtrl** (another address), it will use a variant of the standard servo algorithm that has the same topology as the Turbo PMAC servo algorithm, providing the easiest possible conversion.

If **Motor[*x*].Ctrl** is set to **Sys.GantryXCtrl** (yet another address), then this motor will use a special dual-motor gantry-motor algorithm with cross-coupling terms, along with the next higher-numbered motor. **Motor[*x*].ExtraMotors** should be set to 1 in this case so that both motors' servo calculations are executed in this motor's algorithm.

If **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** (still another address), then this motor will use the adaptive servo algorithm, which can automatically detect changes in inertia and change gain values to compensate.

If **Motor[*x*].Ctrl** is set to **Sys.PosCtrl** (still another address), then this motor will simply output its command position to the 32-bit register specified by **Motor[*x*].pDac** each servo cycle; no servo-loop closure for the motor is performed in the Power PMAC. This setting is primarily for motion networks in which all of the loops are closed in the networked drives.

For a custom user-written servo, the IDE program will permit you to specify the address interactively by selecting an algorithm from a list of those available. Typically, the setting will be to **UserAlgo.ServoCtrlAddr[*i*]**.

Motor[*x*].CurrentNullPeriod

Description: Number of cycles for current auto-null averaging

Range: -32,768 .. 32,767

Units: Phase interrupt periods

Default: 0 (disabled)

Motor[*x*].CurrentNullPeriod controls the enabling and length of the process for "auto-nulling" the current feedback measurements for the digital current loop for the motor. It is only used if the digital current loop is enabled for the motor (**Motor[*x*].PhaseCtrl** > 0, **Motor[*x*].pAdc** > 0). If **Motor[*x*].CurrentNullPeriod** is set to its default value of 0, the auto-nulling process is not enabled.

If **Motor[*x*].CurrentNullPeriod** is set to a non-zero value, Power PMAC will automatically execute an auto-nulling process each time the motor is commanded to close its servo loop from an open-loop state. In this process, the motor's A and B-phase current feedback ADC values are repeatedly sampled when they should report zero current. The magnitude of **Motor[*x*].CurrentNullPeriod** specifies the number of times each value is sampled in this process. The average of these samples for each phase is then computed, and the bias terms for the phases – **Motor[*x*].IaBias** and **Motor[*x*].IbBias** – are automatically set to the negative of these averages. Note that if auto-nulling is enabled, the saved values of these bias terms are not used, as they will automatically be overwritten each time the motor is enabled.

Note

Power Brick amplifiers perform their own current-feedback auto-nulling in the power stage, so it is not necessary to use this software feature for that purpose.

This process will be performed on a phase referencing for the motor. This can be triggered by an on-line **$** command for the motor, or by directly setting **Motor[*x*].PhaseFindingStep** to 1. (You can also set **Motor[*x*].PhaseFindingStep** to 8 to trigger the current auto-nulling without the actual phase referencing.) If you set **Motor[*x*].PhaseFindingStep** yourself, it is your responsibility to ensure the motor is in an appropriate "resting state" to perform the auto-nulling. If bit 1 (value 2) of **Motor[*x*].PowerOnMode** is set to 1, this phase referencing will be performed automatically on power-on/reset of the Power PMAC. Auto-nulling at this time is generally not recommended, as it is not possible to confirm that the drive is ready to perform the measurements for a proper auto-nulling.

The auto-nulling will also be performed on the execution of an **enable** command (on-line or program) for the coordinate system to which the motor is assigned, if the motor is not already enabled. In addition, it will be performed on the execution of an on-line **j/** command for the motor that closes the loop for the motor.

If automatic brake control for this motor is enabled (**Motor[*x*].pBrakeOut** != 0), the delay period for enabling of control after brake disengagement is commanded (**Motor[*x*].BrakeOffDelay** milliseconds) starts after the auto-nulling process is completed.

If **Motor[*x*].CurrentNullPeriod** is set to a positive value, this auto-nulling process occurs before the amplifer-enable signal to the drive is set true. This means that the measurements are taken with the PWM outputs for all phases completely disabled, guaranteeing that no current is flowing in the phases. In this case, the bias being measured is that of the current feedback circuits alone. After the auto-nulling process is completed, the amplifier-enable signal is set true, possibly after a brake disengagement delay, so motor control can begin.

If **Motor[*x*].CurrentNullPeriod** is set to a negative value, the amplifier-enable signal to the drive is set true before the auto-nulling process occurs. In this case, Power PMAC temporarily

forces **Motor[*x*].PwmSf** to zero for the duration of the auto-nulling process so that all phases are commanding a nominal zero voltage value. In this case, the bias being measured is a combination of the PWM voltage command circuits and the current feedback circuits. After the auto-nulling process is completed, the previous value of **Motor[*x*].PwmSf** is restored, possibly after a brake disengagement delay, so motor control can begin.

At the beginning of the auto-nulling process, status element **Motor[*x*].CurrentNullTimer** is set to the magnitude of **CurrentNullPeriod**. Each subsequent phase cycle, it is decremented by 1. While it is greater than zero, status elements **Motor[*x*].IaMeas** and **Motor[*x*].IbMeas** accumulate the sum of the values read from the beginning of the process, instead of simply reporting the present cycle's value. When **CurrentNullTimer** reaches 0, the sums are divided by **CurrentNullPeriod**, and the negatives of the resulting values are automatically written to **Motor[*x*].IaBias** and **Motor[*x*].IbBias**, ending the auto-nulling itself. After this, the rest of the phase-referencing or enabling process continues. To monitor for the end of the entire process, the **Motor[*x*].ClosedLoop** or **Coord[*x*].ClosedLoop** status bit can be checked.

It is intended that the motor not be moving when this process is performed, as the back EMF voltage resulting from motion can skew the readings. If a velocity greater than **Motor[*x*].MaxSpeed**/1000 is detected during the process, the nulling will be considered to have failed and the motor cannot be enabled afterwards.

Motor[*x*].CurrentScale

Description: Motor phase current unit scaling

Range: Non-negative floating-point

Units: 16-bit ADC LSBs per ampere of phase current

Default: 0.0

Motor[*x*].CurrentScale expresses the scaling of motor phase current into internal Power PMAC units. Power PMAC makes no automatic use of this element; the element is intended for use with the IDE automatic motor setup routines to store and later retrieve user unit information for the motor. It is most commonly set from the IDE setup routines.

Motor[*x*].CurrentScale is intended to store the current scaling for a Power PMAC motor in direct-PWM mode with digital current-loop closure inside the Power PMAC. It is expressed as the number of 16-bit ADC LSBs per ampere of phase current. It is usually calculated by dividing 32,768 (the maximum reading of a 16-bit ADC) by the maximum instantaneous phase current for the direct-PWM amplifier that is used. In control block-diagram terms, it can be considered the ADC "gain" term (K_{adc}).

Note that the current ampere rating is in terms of instantaneous phase current, not RMS current, and not converted field-frame current. Note also that it is also in terms of 16-bit ADC LSBs, even if the actual ADCs used have a different resolution (commonly 12 or 14 bits).

Motor[*x*].CurrentScale is new in V2.1 firmware, released 1st quarter 2016.

Motor[x].DacBias

Description: Servo output offset

Range: -32,768.0 .. 32,768.0

Units: 16-bit DAC equivalent

Default: 0

Motor[x].DacBias specifies the offset to be added to the calculated output command from the position/velocity servo loop. This offset is added *after* the magnitude of the command value is compared to the limit of **Motor[x].MaxDac**, so it is possible that the offset value can be greater than the magnitude limit.

Note that despite the name of this element, it is not required that D/A converters be used for output. If Power PMAC is performing commutation and/or current-loop calculations for this motor, **Motor[x].DacBias** offsets the torque input value into those algorithms. It is independent of the commutation phase-bias terms **Motor[x].IaBias** and **Motor[x].IbBias**.

Motor[x].DacBias is in units of a 16-bit output, even if the actual resolution of the output device is different. For example, with 18-bit DAC outputs, a value for **Motor[x].DacBias** of 100 is equivalent to an offset of 400 written to the DAC.

Motor[x].DacShift

Description: Output value shift right distance

Range: 0 .. 31

Units: Bits

Default: 0

Motor[x].DacShift specifies the number of bits the command output value[s] is [are] shifted right before being written to the output register[s]. It permits the use of output devices that are not automatically “left justified”.

For most output devices, **Motor[x].DacShift** will be set to the default value of 0. For the 18-bit DAC option on the ACC-24E3, it should be set to 6, because these DACs use 18 bits right-justified in a 24-bit data field.

Motor[x].DacShift constitutes bits 27 – 31 of the full-word element **Motor[x].Control[1]**.

Motor[x].DtOverRotorTc

Description: Induction motor slip constant

Range: Non-negative floating-point

Units: none (ratio of times)

Default: 0.0

Legacy I-variable alias: I*x*78

Motor[*x*].DtOverRotorTc controls the relationship between the torque command and the slip frequency of magnetic field on the rotor of an AC asynchronous (induction) motor. While it is usually set experimentally, it can be calculated as the ratio between the phase update period and the rotor (not stator) L/R electrical time constant.

If **Motor[*x*].DtOverRotorTc** is greater than zero, Power PMAC estimates the rotor magnetization current **Motor[*x*].Imag** each cycle by filtering the stator direct current value commanded from **Motor[*x*].IdCmd** and divides this current value into **DtOverRotorTc** to get the present "slip constant" stored in **Motor[*x*].SlipGain**. This value is then multiplied by the quadrature current value controlled by **Motor[*x*].IqCmd** (the servo loop output) to get the slip frequency. If bit 1 (value 2) of **Motor[*x*].PhaseCtrl** is set to 0, these slip calculations use the measured values for quadrature and direct current; if this bit is set to 1, these calculations use the commanded values for the currents.

Note

If **DtOverRotorTc** is set to 0.0, **SlipGain** can be set directly by the user and saved. This is useful for open-loop direct microstepping.

Motor[*x*].DtOverRotorTc can be set experimentally by giving the motor an open-loop (torque) output command and watching the velocity response, probably with the data-gathering feature. As the velocity saturates because the back EMF reaches the supply voltage, the velocity should fall back about 5% to reach a steady-state value. If it falls back more than this, the slip time constant is too high; if it falls back less than this, or not at all, the slip time constant is too low.

0.00015 is a typical value of **Motor[*x*].DtOverRotorTc** for a standard induction motor at a phase update rate of about 9 kHz.

Motor[*x*].DtOverRotorTc is only active if bit 0 (value 1) or bit 2 (value 4) of **Motor[*x*].PhaseCtrl** is set to 1 to specify Power PMAC commutation of this motor. **Motor[*x*].DtOverRotorTc** must be set to 0 when commutating synchronous motors such as permanent-magnet brushless motors, stepper motors, and switched (variable) reluctance motors.

Motor[x].EcatAmpFaultLimit

Description: Maximum delay after enabling before amplifier fault clear

Range: $0 .. $FFFF (0 .. 65,535)

Units: Real-time interrupt periods

Default: $64 (100)

Motor[*x*].EcatAmpFaultLimit specifies the maximum number of real-time interrupt periods an EtherCAT drive has to clear its "fault" state after Power PMAC enables the drive before Power PMAC will consider that the fault will not clear and return the Power PMAC motor status to the amplifier fault state. One real-time interrupt period is (**Sys.RtIntPeriod** + 1) servo interrupt periods.

Normally, any time after Power PMAC has enabled a motor, it checks the amplifier fault input regularly, and if it finds it in the fault state, it disables the motor and either disables or stops other motors in the coordinate system (as determined by **Motor[*x*].FaultMode**). However, with the delays of the EtherCAT network and the state machine of many EtherCAT drives, if the Power PMAC starts checking for amplifier fault immediately after enabling the motor, the old fault state may not yet have been cleared, and Power PMAC would consider this a new fault.

Motor[*x*].EcatAmpFaultLimit should be set to a value large enough to reliably ensure that any existing fault bit will be cleared before the Power PMAC delay is over and it starts checking. It should not be set so large that Power PMAC would have a significant delay in responding to an actual fault condition.

Motor[*x*].EncLossBit

Description: Bit number of sensor-loss flag in **pEncLoss** register

Range: 0 .. 31

Units: Bit number (little-endian)

Default: 0

Motor[*x*].EncLossBit determines which bit of the 32-bit register whose address is specified by **Motor[*x*].pEncLoss** Power PMAC will use to detect the encoder-loss for the motor. The bit number is specified in the "little-endian" convention, where bit *n* has a value of 2^n in the 32-bit word. The polarity of this encoder-loss status bit is set by **Motor[*x*].EncLossLevel**.

For quadrature encoders connected to ACC-24E2*x* PMAC2-style axis-interface boards, which have "exclusive-OR" quadrature loss detection circuits, **Motor[*x*].EncLossBit** should be set to 13, as the **Gate1[*i*].Chan[*j*].EncLossN** status bit is found in bit 13 of the register.

For quadrature encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which also have "exclusive-OR" quadrature loss detection circuits, **Motor[*x*].EncLossBit** should be set to 28, as the **Gate3[*i*].Chan[*j*].LossStatus** bit is found in bit 28 of the register.

For analog sinusoidal encoders or resolvers connected to ACC-24E3 PMAC3-style axis-interface boards, which have "sum-of-squares" loss detection circuits, **Motor[*x*].EncLossBit** should be set to 31, as the **Gate3[*i*].Chan[*j*].SosError** bit is found in bit 31 of the register.

For serial encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which have several protocol-dependent loss-detection circuits, **Motor[*x*].EncLossBit** should usually be set to 31, as the channel's "time-out" error bit is found in bit 31 of the **Gate3[*i*].Chan[*j*].SerialEncDataB** register, although this may differ depending on the protocol.

The Type = 12 encoder conversion table entry (new in V2.0 firmware, released 1st quarter 2015) can monitor each servo cycle for multiple error bits in a single register. If any of these bits is set, the entry sets bit 0 of **EncTable[*m*].Status** to 1, and this bit can be used as the encoder-loss bit for the motor.

Motor[*x*].EncLossLevel

Description: Sensor-loss fault logical state

Range: 0 .. 1

Units: Boolean

Default: 1

Motor[*x*].EncLossLevel specifies the loss state of the encoder-loss status bit that will cause an encoder-loss error. The required number of detections in this state is specified by **Motor[*x*].EncLossLimit**.

If **Motor[*x*].EncLossLevel** is set to 0, a 0 value is considered a fault; if it is set to 1, a 1 value is considered a fault. The encoder loss bit is read from the register specified by **Motor[*x*].pEncLoss** at the bit number specified by **Motor[*x*].EncLossBit**. If **Motor[*x*].pEncLoss** is set to 0 to disable the encoder-loss detection function, the setting **Motor[*x*].EncLossLevel** does not matter.

For quadrature encoders connected to ACC-24E2*x* PMAC2-style axis-interface boards, which have "exclusive-OR" quadrature loss detection circuits, **Motor[*x*].EncLossLevel** should be set to 0.

For quadrature encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which also have "exclusive-OR" quadrature loss detection circuits, **Motor[*x*].EncLossLevel** should be set to 1.

For analog sinusoidal encoders or resolvers connected to ACC-24E3 PMAC3-style axis-interface boards, which have "sum-of-squares" loss detection circuits, **Motor[*x*].EncLossLevel** should be set to 1.

For serial encoders connected to ACC-24E3 PMAC3-style axis-interface boards or to the Power Brick control boards, which have several protocol-dependent loss-detection circuits, **Motor[*x*].EncLossLevel** should be set to 1.

Motor[x].EncLossLimit

Description: Sensor-loss maximum accumulated number of fault detections

Range: 0 .. 255

Units: Scans

Default: 0

Motor[x].EncLossLimit specifies the maximum number of accumulated scans the Power PMAC can find the encoder in its "loss" state without tripping on an encoder-loss fault. If the number of accumulated scans in the loss state exceeds this value, Power PMAC will automatically "kill" this motor, and kill or abort other motors in the same coordinate system, depending on the setting of **Motor[x].FaultMode**. Power PMAC checks the state of the specified encoder-loss bit every real-time interrupt period.

Many of the encoder-loss detection circuits can temporarily sense a loss due to electrical noise or similar factors. The ability to confirm a true loss by requiring multiple consecutive scans detecting the loss can be valuable in eliminating false trips.

The register for the encoder loss bit is specified by **Motor[x].pEncLoss**. The bit within this register is specified by **Motor[x].EncLossBit**. The loss state of this bit is specified by **Motor[x].EncLossLevel**. If **Motor[x].pEncLoss** is set to 0 to disable the encoder-loss detection function, the setting **Motor[x].EncLossLimit** does not matter. The accumulated number of scans with the specified bit found in the "loss" state is found in status element **Motor[x].EncLossCount**. This bit is incremented each time the bit is found in the loss state, and decremented (if greater than zero) each time the bit is not found in the loss state.

Motor[x].EncType

Description: Position feedback type for capture control

Range: 0 .. 255

Units: Enumeration

Default: Auto-configured based on hardware

Motor[x].EncType denotes what type of servo position feedback is used for the purposes of matching it properly with hardware-captured positions for triggered moves such as homing searches, and what type of flag interface is used. The value of **Motor[x].EncType** does not control any functions by itself, but the act of setting the value of **Motor[x].EncType** in the Power PMAC Script environment (on-line commands, buffered program commands, copying from flash memory during power-up/reset) forces the value of multiple setup elements that do control these functions.

The following table shows the possible values of **Motor[x].EncType** and the position/flag interface types expected for them.

Type	Interface Type	Feedback Type	Hardware
0	(*None*)	(*No hardware assigned*)	(*None*)
1	PMAC2 style Servo IC	Quadrature encoder, raw count	ACC-24E2*x*
2	PMAC2 style Servo IC	Quadrature encoder, 1/T extension	ACC-24E2*x*
3	PMAC2 style Servo IC	Sinusoidal encoder, arctan extension	ACC-51E
4	MACRO interface IC	Type 1 MACRO ring feedback, pure slave device	ACC-5E, 5E3
5	PMAC3-style IC	Quadrature encoder, 1/T extension	ACC-24E3, Power Brick
6	PMAC3-style IC	Sinusoidal encoder, arctan extension	ACC-24E3, Power Brick
7	PMAC3-style IC	Sinusoidal encoder, auto-corrected extension	ACC-24E3
8-11	(*reserved*)	(*reserved*)	(*reserved*)
12	MACRO interface IC	Type 1 MACRO ring feedback, PMAC acting as auxiliary slave	ACC-5E, 5E3
13+	(*reserved*)	(*reserved*)	(*reserved*)

This next table shows for each implemented value of **Motor[*x*].EncType** what value each of the **Motor[*x*]** "dependent" variables automatically receives when **Motor[*x*].EncType** is set to an implemented type through the Script environment.

EncType ->	1	2	3	4, 12	5	6	7
AmpEnableBit	22	22	22	22	8	8	8
AmpFaultBit	23	23	23	23	7	7	7
CaptFlagBit	19	19	19	19	20	20	20
CaptPosLeftShift	8	9	10	13	8	4	6
CaptPosRightShift	8	8	8	0	8	0	0
CaptPosRound	0	1	1	1	1	0	0
LimitBits	25	25	25	25	9	9	9

In addition, when **Motor[*x*].EncType** is set to an implemented type through the Script environment, **Motor[*x*].pCaptFlag** (address of the capture-flag register) is automatically set to the value of **Motor[*x*].pEncStatus**, and **Motor[*x*].pCaptPos** (address of the captured-position register) is automatically set to the address of the hardware capture register for the IC channel specified by **Motor[*x*].pEncStatus** (**Gate*n*[*i*].Chan[*j*].HomeCapt.a**).

Note that **Motor[*x*].pCaptPos** is not automatically set for **EncType** = 4 or 12 (MACRO), because the captured position is obtained through a MACRO communications request in this mode, not simply from reading a register.

Motor[*x*].ExtraMotors

Description: Additional motors controlled by servo loop

Range: 0 .. 255

Units: Power PMAC motors

Default: 0

Motor[*x*].ExtraMotors specifies the number of additional motors whose servo-loop closure is performed by this motor's servo algorithm. These additional motors must be numbered consecutively starting one greater than this motor's number. For example, if **Motor[7].ExtraMotors** is set to 4, then Motor 7's servo algorithm is expected to perform the servo loop closure for Motors 8, 9, 10, and 11 as well as for Motor 7.

Non-zero values of **Motor[*x*].ExtraMotors** are intended to facilitate the use of "multiple-input, multiple-output" (MIMO), or "cross-coupled" servo algorithms. **Motor[*x*].ExtraMotors** should be set to 0 for any of the built-in "single-input, single-output" (SISO) servo algorithms, e.g. when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, or **Sys.PosCtrl**. It should be set to 1 for the leader motor when the built-in cross-coupled gantry algorithm is used (**Motor[*x*].Ctrl** for the leader motor is set to **Sys.GantryXCtrl** and the follower motor is numbered one greater than the leader motor).

For custom MIMO algorithms, **Motor[*x*].ExtraMotors** for the lowest numbered motor in the group should be set to a value one less than the number of motors in the group When writing these custom MIMO algorithms, it should be noted that the servo output commands for any of the "extra" motors must be written directly to the holding register for that motor: **Motor[*x*].ServoOut**. The servo output command for the lowest-numbered motor of the group should be the function's "return" value.

If **Motor[*x*].ExtraMotors** is set to a value *n* greater than 0, separate servo algorithms for these next *n* motors will not be executed.

Motor[*x*].FatalFeLimit

Description: Fatal (shutdown) following error limit

Range: Non-negative floating-point

Units: Motor units

Default: 2000.0

Legacy I-variable alias: I*x*11

Motor[*x*].FatalFeLimit sets the magnitude of the following error for the specified motor at which operation will shut down. When the magnitude of the following error exceeds **Motor[*x*].FatalFeLimit**, the specified motor is "killed" (open loop, amplifier disabled, zero command output). If the motor's coordinate system is executing a program at the time, the program is aborted. Other motors in the same coordinate system are also killed or aborted, depending on the value of **Motor[*x*].FaultMode**, but motors in other coordinate systems are not affected.

Setting **Motor[*x*].FatalFeLimit** to zero disables the fatal-following error limit for the motor. This may be desirable during initial development work, but it is *strongly discouraged* in an actual application. A fatal following error limit is a very important protection against various types of faults, such as loss of feedback or reversed feedback, that often cannot be detected directly, and that can cause severe damage to people and equipment.

The default setting of **Motor[*x*].FatalFeLimit** is reasonable for many systems when the motor units are defined as the "raw counts" of the feedback. The user who changes the motor units into engineering units such as millimeters or degrees, which are typically much larger than the raw counts, should make **Motor[*x*].FatalFeLimit** proportionally smaller in order to keep it as a useful safety limit.

Motor[*x*].FaultMode

Description: Fault action control

Range: 0 .. 7

Units: Bit field

Default: 0

Motor[*x*].FaultMode specifies the action taken on certain fault conditions for the motor. It is a 3-bit value, with each bit controlling certain actions independently.

Bit 0 (value 1) controls the action when this motor exceeds its fatal following error limit as set by **Motor[*x*].FatalFeLimit**, detects an amplifier-fault condition (either from an input signal or by exceeding its integrated-current limit as set by **Motor[*x*].I2tTrip**), or detects and "encoder loss" condition. This motor is always "killed" in these cases (open-loop, zero command output, amplifier disabled).

If this bit is set to the default value of 0, the other motors in the coordinate system are "aborted" (decelerated to a closed-loop stop according to **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**) in these cases. If this bit is set to 1, the other motors in the coordinate system are also killed in these cases. For motors whose motion is hard-coupled mechanically, as in gantry systems, it is important that all of the coupled motors be killed on the failure of any one of them. Note that other motors in the coordinate system are aborted or killed on one of these faults even if they are not in coordinated motion with this motor at the time of the fault.

Motors in other coordinate systems are never affected in these cases.

Starting in V2.0 firmware, released 1st quarter 2015, if a motor faults during an independent motor move (jog, homing search, open-loop), as opposed to faulting during a coordinated program move, other motors in the coordinate system are not aborted or killed.

Bit 1 (value 2) controls the action when this motor hits a hardware or software position limit in open-loop enabled mode (as from an on-line **outn** or buffered **coutn** command). If this bit is set to the default value of 0, the motor is "aborted" (decelerated to a closed-loop stop according to **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**). If it is set to 1, the motor is "killed" (open-loop, zero command output, amplifier disabled).

Bit 2 (value 4) controls the action when this motor hits a hardware limit in closed-loop enabled mode when it has not already hit a software limit. If this bit is set to the default value of 0, the motor is "aborted" (decelerated to a closed-loop stop according to **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**). If it is set to 1, the motor is "killed" (open-loop, zero command output,

amplifier disabled). The idea behind killing the motor in this case is that this should only occur when position information is lost, meaning that a controlled stop is not possible.

Motor[*x*].GantrySlewRate

Description: Gantry follower motor homing de-skew rate

Range: Non-negative floating-point

Units: Motor units per servo cycle

Default: 0.0

Motor[*x*].GantrySlewRate specifies the rate at which a motor that is activated in "gantry follower" mode (**Motor[*x*].ServoCtrl** = 8) will slew to its own relative home position once it finds its own home trigger. Other than this slewing move, a gantry follower motor does not generate its own commanded trajectories; instead it uses the trajectories generated by the "gantry leader" motor specified in **Motor[*x*].CmdMotor**. This slewing move permits the "de-skewing" of the follower motor relative to the leader motor based on the difference in their home trigger positions.

In typical use, a gantry-follower motor is commanded to execute a homing-search move at the same time as the gantry-leader motor (e.g. **#1,2hm**). During the homing-search move, it is simply executing the trajectory generated by the leader motor. However, once both the leader and follower motors have both found their home triggers, the follower motor computes the relative distance between the two motor's home positions (based on the trigger positions and values of **Motor[*x*].HomeOffset** for both motors, and superimposes a correction for this at the rate specified by **Motor[*x*].GantrySlewRate** on top of whatever motion (if any) is occurring due to following the gantry leader motor at the time.

The gantry leader motor's homing search move should be set up such that it will virtually always pass both its own home trigger position and the home trigger position(s) of the follower motor(s). In practice, the deceleration distance should be longer than the worst-case power-up skew between the motors. The homing routine should check to ensure that the **Motor[*x*].HomeComplete** status bit for all gantry motors is set to 1 (this occurs when the trigger is found for the motor), and that the **Motor[*x*].GantryHomed** status bit for all gantry follower motors is set to 1 (this occurs when the skew has been fully removed), before proceeding further.

For a gantry follower motor that has already been fully homed (**Motor[*x*].GantryHomed** = 1), **Motor[*x*].GantrySlewRate** is also used whenever closed-loop control is re-established, to eliminate any skew that may have developed while it was open-loop (enabled or disabled).

Note

At the default value for **Motor[*x*].GantrySlewRate** of 0.0, this correction is never made.

Motor[x].HomeOffset

Description: Position referencing offset

Range: Floating-point

Units: Motor units

Default: 0.0

Legacy I-variable alias: I*x*26

Motor[*x*].HomeOffset specifies the (signed) difference between the zero position of sensor(s) for the motor and the motor's own zero "home" position. For a motor that establishes its position reference with a homing-search move, this is the difference between the home trigger position and the motor zero position. For a motor that establishes its position reference with an absolute position read, this is the difference between the absolute sensor's zero position and the motor zero position.

In an absolute position read, the sensor position value is read from the register specified by **Motor[*x*].pAbsPos** as specified by **Motor[*x*].AbsPosFormat**. This value is multiplied by **Motor[*x*].AbsPosSf** to convert it to motor units, and then **Motor[*x*].HomeOffset** is subtracted from this result to get the absolute motor position.

In a homing search move, **Motor[*x*].HomeOffset** specifies the distance between the *actual* position at which the home trigger is found, and the *commanded* end of the post-trigger move, where the motor will come to a stop. It is added to the trigger position to get the end position. The commanded end position of the post-trigger move is then considered motor position zero. (It is possible to use other offsets to create a different *axis* position zero for programming purposes.)

A difference between the trigger position and the motor zero position is particularly useful when using an overtravel limit as a home flag (offsetting out of the limit before re-enabling the limit input as a limit). If **Motor[*x*].HomeOffset** is large enough (greater than 1/2 times home speed times acceleration time), it permits a homing-search move without any reversal of direction.

Motor[x].HomeVel

Description: Home-search command signed velocity

Range: Positive floating-point

Units: Motor units per millisecond

Default: 10.0

Legacy I-variable alias: I*x*23

Motor[*x*].HomeVel establishes the commanded speed and direction of a homing-search move for the motor. Changing the sign reverses the direction of the homing move – a negative value specifies a home search in the negative direction; a positive value specifies the positive direction.

If the post-trigger portion of the homing move is long enough, it will peak at a speed whose magnitude, but not necessarily direction, is set by the magnitude of **Motor[*x*].HomeVel**.

Example

To set a home speed of 4 m/min (about 160 in/min) in the negative direction with motor units of 20 microns:

$$HomeVel = -\frac{4m}{min} * \frac{motor\_unit}{2*10^{-5}m} * \frac{min}{6*10^{4}ms} = -3.333\left(\frac{motor\_unit}{ms}\right)$$

Motor[*x*].I2tSet

Description: Continuous current limit

Range: Non-negative floating-point

Units: 16-bit DAC/ADC equivalent

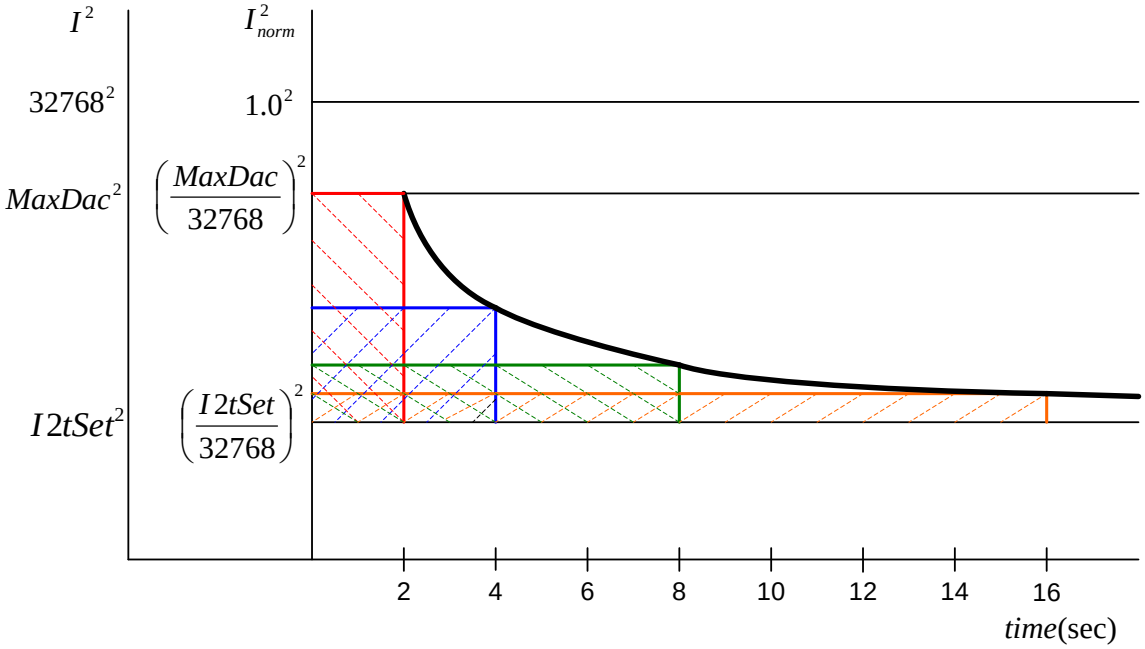
Default: 0.0

Legacy I-variable alias: I*x*57

Motor[*x*].I2tSet specifies the magnitude of the motor's maximum continuous current limit for Power PMAC's integrated current limiting function, when that function is active (**Motor[*x*].I2tSet** must be greater than 0 for the integrated current limit to be active). If Power PMAC is closing a digital current loop for the motor, it uses actual current measurements for this function; otherwise it uses commanded current values. If the magnitude of the actual or commanded current level from Power PMAC is above the magnitude of **Motor[*x*].I2tSet** for a significant period of time, as set by **Motor[*x*].I2tTrip**, Power PMAC will trip this motor on an integrated-current amplifier fault condition.

Motor[*x*].I2tSet is in units of a signed 16-bit DAC or ADC (maximum possible value of 32,767), even if the actual output or input device has a different resolution. Typically, **Motor[*x*].I2tSet** will be set to between 1/3 and 1/2 of the **Motor[*x*].MaxDac** (instantaneous) output limit. Consult your amplifier and motor documentation for their specifications on instantaneous and continuous current limits.

This diagram shows a typical setting of **I2tSet** compared to **MaxDac** and the full possible output.



Allowed Current Levels Above Continuous vs. Time at Level

Technically, **Motor[*x*].I2tSet** is the continuous limit of the vector sum of the quadrature and direct currents. The quadrature (torque-producing) current is the output of the position/velocity-loop servo. The direct (magnetization) current is set by **Motor[*x*].IdCmd**.

In sine-wave output mode (**Motor[*x*].PhaseCtrl** bit 0 or bit 2 = 1, **Motor[*x*].pAdc** = 0), amplifier gains are typically given in amperes of phase current per volt of PMAC output, but motor and amplifier limits are typically given in RMS amperage values. In this case, it is important to realize that peak phase current values are √2 (1.414) times greater than the RMS values.

In direct-PWM mode (**Motor[*x*].PhaseCtrl** bit 0 = 1, **Motor[*x*].pAdc** > 0) of 3-phase motors (**Motor[*x*].PhaseOffset** = +/-683), the corresponding top values of the sinusoidal phase-current ADC readings will be 1/cos(30°), or 1.15, times greater than the vector sum of quadrature and direct current. Therefore, once you have established the top values you want to see in the A/D converters your phase currents on a continuous basis, this value should be multiplied by cos(30°), or 0.866, to get your value for **Motor[*x*].I2tSet**. Remember that if current limits are given as RMS values, you should multiply these by √2 (1.414) to get peak phase current values.

Examples

A torque-mode brushless-motor amplifier has a $15A_{rms}$ maximum intermittent current capability (i.e. a ±10V input commands a ±$15A_{rms}$ current), and the brushless motor has a $5A_{rms}$ continuous current limit, **I2tSet** is calculated as:

$$I2tSet = 32{,}768 * \frac{5A_{rms}}{15A_{rms}} = 10{,}923$$

A sine-wave input amplifier with a transconductance gain for each phase of 2 amps/volt could produce a current output of 20A (peak) on each phase from a full-range (±10V) sinewave command from a Power PMAC with **PwmSf** at the default setting of 32,767. This corresponds to an RMS magnitude of 20/*sqrt*(2) = 14.1A_{rms}. If the motor has a continuous current limit of 4A_{rms}, **I2tSet** can be set by the following equation:

$$I2tSet = 32{,}768 * \frac{4A_{rms}}{14.1A_{rms}} = 9{,}296$$

For a 3-phase brushless servo motor with a 4A_{rms} continuous current limit controlled by a direct-PWM amplifier in which the ADC reports its maximum instantaneous value of 32,768 at a phase current of 16.26A, **I2tSet** can be computed as:

$$I2tSet = 4 * \sqrt{2} * \frac{32{,}768}{16.26} * \cos(30°) = 9{,}872$$

The factor of sqrt(2) in this equation is from the conversion of instaneous current to RMS current. The factor of cos(30°) is from the 3-phase (abc) to 2-phase (dq) conversion.

Motor[*x*].I2tTrip

Description: Integrated current shutdown limit

Range: Non-negative floating-point

Units: (16-bit "DAC" units)^2 * seconds

Default: 0.0 (disabled)

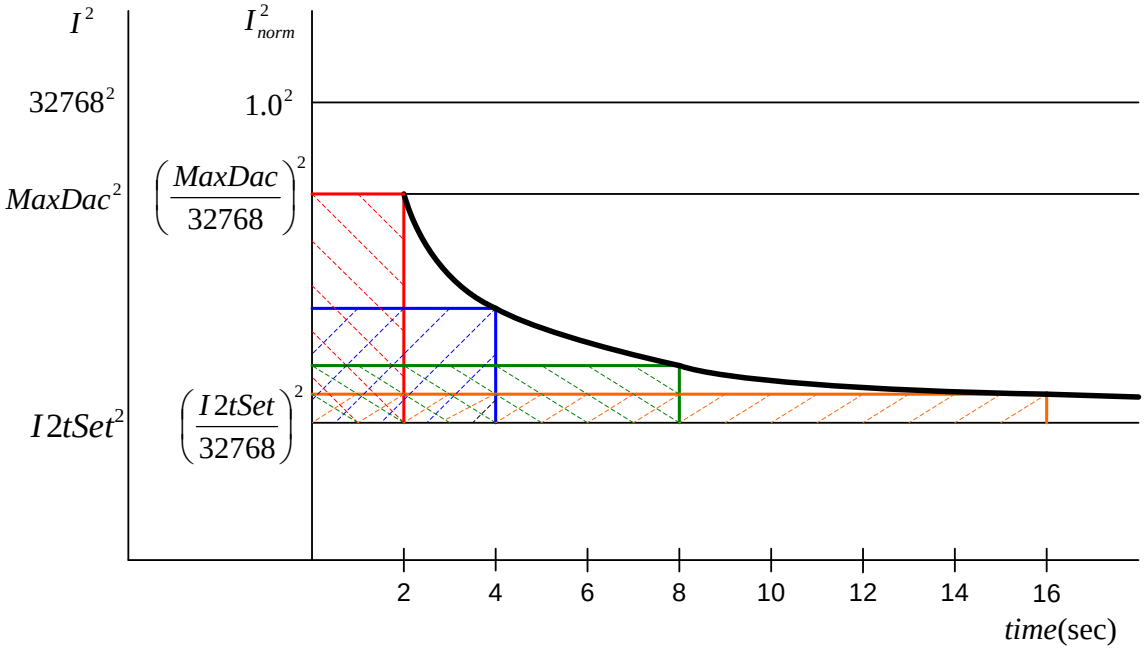
Legacy I-variable alias: I*x*58

Motor[*x*].I2tTrip sets the maximum integrated current limit for Power PMAC's I^2T integrated current limiting function, if that function is active. (If **Motor[*x*].I2tSet** is 0, the I^2T limiting function is disabled.) If **Motor[*x*].I2tSet** is greater than 0, Power PMAC will compare the time-integrated square of the difference between the commanded or actual current and the **Motor[*x*].I2tSet** continuous current limit to **Motor[*x*].I2tTrip**. If the integrated value exceeds the limit set by **Motor[*x*].I2tTrip**, then Power PMAC faults the motor just as it would for receiving an amplifier fault signal, setting both the amplifier-fault and the integrated-current-fault motor status bits.

The **Motor[*x*].I2tTrip** limit is typically set by taking the relationship between the instantaneous current limit (**Motor[*x*].MaxDac** on Power PMAC, in units of a 16-bit DAC or equivalent), the magnetization current (commanded by **Motor[*x*].IdCmd**; typically 0 except for vector control of induction motors) and the continuous current limit (**Motor[*x*].I2tSet** on Turbo PMAC, in units of a 16-bit DAC) and multiplying by the time permitted at the instantaneous limit. The formula is:

$$I2tTrip = \left(MaxDac^2 + IdCmd^2 - I2tSet^2\right) * PermittedTime(\text{sec})$$

This diagram shows the relationship between the integrated current limit at full instantaneous current (red rectangle) and the integrated current limt at lower current levels (blue, green, and orange rectangles of equivalent area).



Allowed Current Levels Above Continuous vs. Time at Level

Caution

Power PMAC's I^2T computations use the value of **Sys.ServoPeriod** to calculate the time in seconds elapsed between samples. The value of **Sys.ServoPeriod** must therefore be correct for accurate I^2T computations.

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:

$$N = ceil\left(\frac{Sys.ServoMotors + 1}{Sys.MotorsPerRtInt}\right)$$

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.

Examples

For a brushless motor with a $5A_{rms}$ continuous limit and a $10A_{rms}$ intermittent limit used with a torque-mode amplifier that has a peak current capability of $15A_{rms}$, if the maximum time permitted for $10A_{rms}$ operation is 2 seconds, **Motor[*x*].I2tTrip** can be calculated as:

$$I2tTrip = \left[\left(32{,}768*\frac{10}{15}\right)^2 + 0^2 - \left(32768*\frac{5}{15}\right)^2\right]*2 = 7.16\times10^8$$

For a brushless motor with a $4A_{rms}$ continuous current limit and a $12A_{rms}$ intermittent current limit used with a sinewave amplifier that has a gain of 2A/V, if the maximum time permitted at the intermittent limit is 3 seconds, **Motor[*x*].I2tTrip** can be calculated as:

$$I2tTrip = \left[\left(32{,}768*\frac{12}{20}\right)^2 + 0^2 - \left(32768*\frac{4}{20}\right)^2\right]*3 = 1.03\times10^9$$

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 direct-PWM amplifier in which the ADC reports its maximum instantaneous value of 32,768 at a phase current of 16.26A, **I2tTrip** can be computed as:

$$I2tTrip = \left[\left(8*\sqrt{2}*\frac{32{,}768}{16.26}*\cos(30^\circ)\right)^2 - \left(8*\sqrt{2}*\frac{32{,}768}{16.26}*\cos(30^\circ)\right)^2\right]*3 = 8.77\times10^8$$

The factor of sqrt(2) in this equation is from the conversion of instaneous current to RMS current. The factor of cos(30°) is from the 3-phase (abc) to 2-phase (dq) conversion.

Motor[*x*].IaBias

Description: Phase A current bias offset

Range: +/-32,768 (floating-point)

Units: Bits of a signed 16-bit input/output

Default: 0.0

Legacy I-variable alias: I*x*29

Motor[*x*].IaBias serves as an offset term on the current for Phase A of a motor commutated by Power PMAC. It is intended as the digital equivalent of an analog offset potentiometer.

If Power PMAC is not performing current-loop closure for the motor, it serves as an offset for the commanded current value for the first phase; it is added to the value calculated for the phase by the commutation algorithm before it is written to the output register.

If Power PMAC is also performing current-loop closure for the motor, it serves as an offset for the measured current value for the first phase; it is added to the value produced for the phase by

the analog-to-digital converter from the phase's current sensor before it is used in the current-loop calculations.

Motor[*x*].IaBias is always in units of a 16-bit DAC or ADC, even if the actual device resolution is different. For example, if an 18-bit DAC output is used for phase-current commands, a **Motor[*x*].IaBias** value of 0.25 corresponds to one LSB of the DAC. If a 12-bit ADC input is used for phase-current measurements a **Motor[*x*].IaBias** value of 16.0 corresponds to one LSB of the ADC.

The value of **Motor[*x*].IaBias** can be set automatically on the enabling of the motor through an "auto-nulling" process if **Motor[*x*].CurrentNull Period** is set to a non-zero value. In this case, the saved value is not used.

Motor[*x*].IbBias

Description: Phase B current bias offset

Range: +/-32,768 (floating-point)

Units: Bits of a signed 16-bit output

Default: 0.0

Legacy I-variable alias: I*x*79

Motor[*x*].IbBias serves as an offset term on the current for Phase B of a motor commutated by Power PMAC. It is intended as the digital equivalent of an analog offset potentiometer.

If Power PMAC is not performing current-loop closure for the motor, it serves as an offset for the commanded current value for the second phase; it is added to the value calculated for the phase by the commutation algorithm before it is written to the output register.

If Power PMAC is also performing current-loop closure for the motor, it serves as an offset for the measured current value for the second phase; it is added to the value produced for the phase by the analog-to-digital converter from the phase's current sensor before it is used in the current-loop calculations.

Motor[*x*].IbBias is always in units of a 16-bit DAC or ADC, even if the actual device resolution is different. For example, if an 18-bit DAC output is used for phase-current commands, a **Motor[*x*].IbBias** value of 0.25 corresponds to one LSB of the DAC. If a 12-bit ADC input is used for phase-current measurements a **Motor[*x*].IbBias** value of 16.0 corresponds to one LSB of the ADC.

The value of **Motor[*x*].IbBias** can be set automatically on the enabling of the motor through an "auto-nulling" process if **Motor[*x*].CurrentNull Period** is set to a non-zero value. In this case, the saved value is not used.

Motor[*x*].IdCmd

Description: Desired magnetization (direct) current

Range: +/-32,768 (floating-point)

Units: Bits of a signed 16-bit output

Default: 0

Legacy I-variable alias: I*x*77

Motor[*x*].IdCmd specifies the desired level of the "direct current" component for a motor commutated by Power PMAC. This current component is commonly called the "magnetization current" because it sets the level of rotor magnetization in AC induction motors. This component is parallel to the rotor magnetic field, and perpendicular to the "quadrature current" component commanded from the position/velocity-loop servo which is the torque-producing component.

Motor[*x*].IdCmd must be a positive value for proper commutation of induction motors. The optimum value for a given induction motor is system-dependent, but 2500 is a good starting point for most of these motors. For levels below magnetic saturation of the rotor, the strength of the rotor magnetic field is proportional to this value. Both the motor's torque constant (K_T) and back-EMF constant (K_E) are proportional to the rotor's magnetic field strength.

It is possible for the user to vary this value dynamically as a function of speed to optimize motor performance over a speed range.

Motor[*x*].IiGain

Description: Current loop integral gain

Range: Non-negative floating-point

Units: LSBs of 16-bit output per LSB of integrated 16-bit current error

Default: 0.001

Legacy I-variable alias: I*x*61

Motor[*x*].IiGain is the integral gain term of the digital current loops, multiplying the difference between the commanded and actual current levels and adding the result into a running integrator that adds into the command output. It is only used if **Motor[*x*].pAdc** > 0 to activate digital current loop execution. Typical values of **Motor[*x*].IiGain** are near 0.2.

Motor[*x*].IiGain can be used with either **Motor[*x*].IpfGain** forward-path proportional gain, or **Motor[*x*].IpbGain** back-path proportional gain. If used with **Motor[*x*].IpfGain**, the value can be quite low, because **Motor[*x*].IpfGain** provides the quick response, and **Motor[*x*].IiGain** just needs to correct for biases. If used with **Motor[*x*].IpbGain**, **Motor[*x*].IiGain** is the only gain that responds directly to command changes, and it must be significantly higher to respond quickly.

Motor[*x*].IiGain is only used if commutation is enabled for the motor and **Motor[*x*].pAdc** > 0 to activate digital current loop execution.

Motor[*x*].InPosBand

Description: In-position threshold

Range: Non-negative floating-point

Units: Motor units

Default: 0.0

Legacy I-variable alias: I*x*28

Motor[*x*].InPosBand specifies the magnitude of the maximum position (following) error at which the motor will be considered "in-position" when not executing a commanded move.

Technically, five conditions must be met for a motor to be considered "in-position":

1.) The motor must be in closed-loop control;
2.) The motor desired velocity must be zero;
3.) The motor must not be executing any move or dwell of definite time;
4.) The magnitude of the following error must be less than or equal to this parameter;
5.) The above four conditions must all be true for (**Motor[*x*].InPosTime** + 1) consecutive servo cycles.

For any active motor, Power PMAC will automatically evaluate whether the motor is "in position" or not every servo cycle.

When a motor is "in position", the status data structure element **Motor[*x*].InPos** bit is set to 1. Otherwise, it is set to 0.

Motor[*x*].InPosTime

Description: In-position number of consecutive scans

Range: 0 .. 255

Units: Servo cycles

Default: 0

Legacy I-variable alias: I*x*88

Motor[*x*].InPosTime specifies the number of consecutive repeat times the in-position conditions for the motor must be true in order for the motor to be considered "in position". At the default value of 0, the first time the in-position conditions (closed-loop, desired velocity zero, error less than or equal to **Motor[*x*].InPosBand**) are met, the in-position bit will be set.

If **Motor[*x*].InPosTime** is greater than 0, then these conditions must also be true for the specified additional number of consecutive checks. Power PMAC checks these conditions every servo cycle for any active motor. If a single one of these additional checks finds one of these conditions is not true, then the count starts over at zero.

This parameter permits the user to ensure that the motor is truly settled at the end position, and not oscillating in and out of the in-position band, before executing the next operation.

Motor[*x*].InvAmax

Description: Inverse of maximum programmed acceleration

Range: Non-negative floating-point

Units: Millisecond^2 / motor unit

Default: 10.0 (= 0.1 motor unit / msec^2)

Legacy I-variable alias: I*x*17

Motor[*x*].InvAmax sets the maximum magnitude of acceleration for several types of programmed moves. For **`linear`** mode moves with segmentation disabled (**Coord[*x*].SegMoveTime** = 0), it is checked at move calculation time against the peak commanded acceleration magnitude as calculated from the **Ta** and **Ts** parameters, and the change in motor speed. This applies to initial acceleration from a stop, and blending between two commanded moves, whether the blending causes an acceleration or deceleration. **Motor[*x*].InvAmax** is not used for the final deceleration to a stop at the end of a sequence of moves; **Motor[*x*].InvDmax** is used for that instead.

Note

The linear-mode move must have a non-zero acceleration time (**Ta** 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.

If a programmed **`linear`** mode move requests a higher acceleration magnitude of this motor, it will extend the time for the acceleration just enough so that this limit is not violated. This will also reduce the acceleration of any other motors in the coordinate system proportionately (even if those motors would not have violated their own limits) so that coordination and path are maintained.

Motor[*x*].InvAmax also sets the maximum magnitude of acceleration for **`linear`**, **`circle`**, and **`pvt`** mode moves on a segment-by-segment basis if the special segmented lookahead algorithm is active for the coordinate system. **Coord[*x*].SegMoveTime** and **Coord[*x*].LHDistance** must be greater than 0 for this acceleration limiting function to work. If the magnitude of the acceleration for a commanded segment exceeds this magnitude, the time for the segment is extended just enough so this limit is not violated. If the times for previous segments must also be extended so that the deceleration to this reduced speed can be made within the acceleration limit, this is done automatically.

Motor[*x*].InvAmax is not used if the coordinate system is in segmentation mode (**Coord[*x*].SegMoveTime** > 0) but the special lookahead buffer is not active (**Coord[*x*].LHDistance** = 0 or no lookahead buffer defined).

Note that **Motor[*x*].InvAmax** is expressed as the inverse of the maximum acceleration magnitude in the base units of milliseconds squared per motor unit (often raw counts). This permits more efficient calculations in the Power PMAC. A value of 0.0 results in no acceleration limiting.

Example

To set a maximum acceleration of 8 m/s^2 (about 0.8g) with motor units of millimeters:

$$InvAmax = \frac{s^2}{8m} * \frac{10^{-3} m}{motor\_unit} * \frac{10^6 ms^2}{s^2} = 125\left(\frac{ms^2}{motor\_unit}\right)$$

Motor[*x*].InvDmax

Description: Inverse of maximum programmed deceleration

Range: Non-negative floating-point

Units: Millisecond^2 / motor unit

Default: 10.0 (= 0.1 motor unit / msec^2)

Motor[*x*].InvDmax sets the maximum magnitude of the final deceleration to a stop (end of continuous move sequence) for **`linear`** mode moves when segmentation mode is disabled for the coordinate system (**Coord[*x*].SegMoveTime** = 0). It is checked at move calculation time against the peak commanded final deceleration magnitude as calculated from the **Td** and **Ts** parameters, and the change in motor speed.

Note

The linear-mode move must have a non-zero deceleration time (**Td** or **Ts** > 0.0) for this check to be made at move calculation time, as it is the deceleration section that is checked against the limit.

If a programmed **`linear`** mode move requests a higher final deceleration magnitude of this motor, it will extend the time for the final deceleration just enough so that this limit is not violated. This will also reduce the final deceleration of any other motors in the coordinate system proportionately (even if those motors would not have violated their own limits) so that coordination and path are maintained.

The acceleration in blending between successive **`linear`** mode moves is limited by **Motor[*x*].InvAmax**, even if it results in a deceleration. In the special segmented lookahead buffer, the magnitude of all accelerations and decelerations is governed by **Motor[*x*].InvAmax**, and **Motor.InvDmax** is not used.

Note that **Motor[*x*].InvDmax** is expressed as the inverse of the maximum deceleration magnitude in the base units of milliseconds squared per motor unit (usually raw count). This permits more efficient calculations in the Power PMAC. A value of 0.0 results in no deceleration limiting.

Example

To set a maximum deceleration of 5 m/s^2 (about 0.5g) with motor units of microns:

$$InvDmax = \frac{s^2}{5m} * \frac{10^{-6} m}{motor\_unit} * \frac{10^6 ms^2}{s^2} = 0.2\left(\frac{ms^2}{motor\_unit}\right)$$

Motor[*x*].InvJmax

Description: Inverse of maximum programmed jerk

Range: Non-negative floating-point

Units: Millisecond^3 / motor unit

Default: 50 (= 0.02 motor unit / msec^3)

Legacy I-variable alias: I*x*18

Motor[*x*].InvJmax sets the maximum magnitude of "jerk" (rate of change of acceleration) for **`linear`** mode moves when segmentation mode is disabled for the coordinate system (**Coord[*x*].SegMoveTime** = 0). It is checked at move calculation time against the peak jerk magnitude as calculated from the **Ta**, **Td** and **Ts** parameters, as appropriate, and the change in motor speed.

Note

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.

If a programmed **`linear`** mode move requests a higher jerk magnitude of this motor, it will extend the time for the "S-curve" portions of the acceleration just enough so that this limit is not violated. This will also reduce the jerk of any other motors in the coordinate system proportionately (even if those motors would not have violated their own limits) so that coordination and path are maintained.

Note that **Motor[*x*].InvJmax** is expressed as the inverse of the maximum jerk magnitude in the base units of milliseconds cubed per motor unit (usually raw count). This permits more efficient calculations in the Power PMAC. A value of 0.0 results in no jerk limiting.

Example

To set a maximum jerk of 100 m/s^3 (about 10g/sec) with motor units of millimeters:

$$InvJmax = \frac{s^3}{100m} * \frac{10^{-3} m}{motor\_unit} * \frac{10^9 ms^3}{s^3} = 10{,}000 \left(\frac{ms^3}{motor\_unit} \right)$$

Motor[x].IpbGain

Description: Current loop back-path proportional gain

Range: Non-negative floating-point

Units: LSBs of 16-bit output per LSB of 16-bit actual current

Default: 1.0

Legacy I-variable alias: I*x*76

Motor[*x*].IpbGain is the proportional gain term of the digital current loop that is in the "back path" of the loop, multiplying the actual current level, and subtracting the result from the command output. Either **Motor[*x*].IpbGain** or **Motor[*x*].IpfGain** (forward path proportional gain) must be used to close the current loop. Generally, only one of these proportional gain terms is used, although both can be. Typical values of **Motor[*x*].IpbGain**, when used, are around 0.9.

If **Motor[*x*].IpbGain** is used as the only proportional gain term, only the **Motor[*x*].IiGain** integral gain term reacts directly to command changes. The act of integration acts as a low-pass filter on the command, which eliminates a lot of noise, but lowers the responsiveness to real changes. Generally **Motor[*x*].IpbGain** is only used when the command value from the position/velocity loop servo have high noise levels (usually due to low position resolution), and the actual current measurements have low noise levels.

Motor[*x*].IpbGain is only used if commutation is enabled for the motor and **Motor[*x*].pAdc** > 0 to activate digital current loop execution.

Motor[x].IpfGain

Description: Current loop forward-path proportional gain

Range: Non-negative floating-point

Units: LSBs of 16-bit output per LSB of 16-bit current error

Default: 1.0

Legacy I-variable alias: I*x*62

Motor[*x*].IpfGain is the proportional gain term of the digital current loops that is in the "forward path" of the loop, multiplying the difference between the commanded and actual current levels. Either **Motor[*x*].IpfGain** or **Motor[*x*].IpbGain** (back path proportional gain) must be used to close the current loop. Generally, only one of these proportional gain terms is used, although both can be. Typical values of **Motor[*x*].IpfGain**, when used, are around 0.9.

Motor[*x*].IpfGain can provide more responsiveness to command changes from the position/velocity loop servo, and therefore a higher current loop bandwidth, than **Motor[*x*].IpbGain**. However, if the command value is very noisy, which can be the case with a low-resolution position sensor, using **Motor[*x*].IpbGain** instead can provide better filtering of the noise.

Motor[*x*].IpfGain is only used if commutation is enabled for the motor and **Motor[*x*].pAdc** > 0 to activate digital current loop execution.

Motor[*x*].IxCoupleGain

Description: Current loop cross-coupled gain

Range: Non-negative floating-point

Units: LSBs of 16-bit output per (LSB of 16-bit current * velocity)

Default: 0.0

Legacy I-variable alias: I*x*89

Motor[*x*].IxCoupleGain is a cross-coupling term between the direct and quadrature current loops. Because of inductance in the motor windings, the voltage in the windings needs to lead the current increasingly as the frequency increases. This creates a cross-coupling effect between the perpendicular direct and quadrature current loops, which are effectively independent at low and zero frequency.

Without this term, it is up to the current-loop integral-gain term in each loop to "charge up" to provide the necessary cross-coupled voltages to obtain the desired currents at higher frequencies. These can do the job eventually, but they are "error driven", and so are generally behind.

Motor[*x*].IxCoupleGain provides a direct method of compensating for this effect. It generally is only used when running at very high speeds (usually over 10,000 rpm) with rapid accelerations and decelerations. Otherwise, the standard current-loop gains are generally good enough.

Motor[*x*].IxCoupleGain is multiplied by the measured quadrature current value and the commutation frequency to obtain an offset to the direct voltage command value. It is also multiplied by the measured direct current value and the commutation frequency to obtain an offset to the quadrature voltage command value.

Motor[*x*].IxCoupleGain is only used if commutation is enabled for the motor and **Motor[*x*].pAdc** > 0 to activate digital current loop execution.

Motor[x].JogOffset

Description: Triggered jog move variable post-trigger offset

Range: Floating-point

Units: Motor units

Default: 0.0

Motor[*x*].JogOffset specifies the signed post-trigger offset distance for triggered jog moves with a variable post-trigger distance (ending in **^***). This offset is the difference between the commanded end position of the post-trigger portion of the move and the actual position where the trigger condition occurred.

Motor[x].JogSpeed

Description: Jog command velocity magnitude

Range: Positive floating-point

Units: Motor units / millisecond

Default: 32.0

Legacy I-variable alias: I*x*22

Motor[*x*].JogSpeed establishes the commanded top speed magnitude of a jog move, or a programmed **rapid**-mode move (if **Motor[*x*].RapidSpeedSel** =0) for the motor. Direction of the move, and so sign of the velocity, is controlled by the move command itself.

A change in this parameter will not take effect until the next move command. For instance, if you wanted to change the jog speed on the fly, you would start the jog move, change this parameter, then issue a new jog command.

Example

To set a top jog speed magnitude of 200 mm/sec (about 8 in/sec) with motor units of 5 microns:

$$JogSpeed = \frac{200mm}{sec} * \frac{motor\_unit}{5*10^{-3}mm} * \frac{sec}{10^{3}ms} = 40\left(\frac{motor\_unit}{ms}\right)$$

Motor[x].JogTa

Description: Jog accel/decel time or inverse rate

Range: Floating-point

Units: Milliseconds (if >= 0) or milliseconds^2 per motor unit (if < 0)

Default: -10 (= 0.1 motor unit / msec^2)

Legacy I-variable alias: I*x*20

Motor[*x*].JogTa sets the time or the magnitude of the acceleration for the motor in a jogging, homing, or programmed **rapid**-mode move when starting, stopping, and changing speeds. If it is greater than or equal to zero, it specifies the time for the acceleration in milliseconds. If it is less than zero, it specifies the inverse of the peak magnitude of the acceleration, in milliseconds squared per motor unit (usually raw count).

Specifying a rate of acceleration is generally better if you want to be able to break into executing moves smoothly. Specifying a time for acceleration is generally better if you want to coordinate jog moves on multiple motors.

If **Motor[*x*].JogTa** specifies a time and **Motor[*x*].JogTs** (S-curve time) specifies a smaller time, the total time spent in acceleration will be equal to the sum of **JogTa** and **JogTs**. (Note that this rule is different from Turbo PMAC.) However, if **Motor[*x*].JogTa** specifies a time and **Motor[*x*].JogTs** specifies a larger time, the total time spent in acceleration will be 2 times **Motor[*x*].JogTs**. Therefore, if **Motor[*x*].JogTa** is set to 0, **Motor[*x*].JogTs** alone controls the acceleration time in "pure" S-curve form.

If **Motor[*x*].JogTs** is less than zero, specifying a maximum "jerk", **Motor[*x*].JogTa** must also be less than zero, specifying a maximum acceleration.

A change in this parameter will not take effect until the next move command. For instance, if you wanted a different deceleration time from acceleration time in a jog move, you would specify the acceleration time, command the jog, change the deceleration time, then command the jog move again (e.g. **j=**), or at least the end of the jog (**j/**).

Example

To set a jog acceleration rate of 5 m/s^2 (about 0.5g) with motor units of millimeters:

$$JogTa = -\frac{s^2}{5m} * \frac{10^{-3} m}{motor\_unit} * \frac{10^6 ms^2}{s^2} = -200\left(\frac{ms^2}{motor\_unit}\right)$$

Motor[*x*].JogTs

Description: Jog accel/decel S-curve time or inverse jerk rate

Range: Floating-point

Units: Milliseconds (if >= 0) or milliseconds^3 per motor unit (if < 0)

Default: -50 (= 0.02 motor units / msec^3)

Legacy I-variable alias: I*x*21

Motor[*x*].JogTs sets the time of the S-curve portion of the acceleration for the motor in a jogging, homing, or programmed **rapid**-mode move when starting, stopping, and changing

speeds, or the maximum magnitude of the "jerk" (rate of change of acceleration) during this acceleration. If it is greater than or equal to zero, it specifies the time for each S-curve section the acceleration in milliseconds. If it is less than zero, it specifies the inverse of the peak magnitude of the jerk, in milliseconds cubed per motor unit (usually raw count).

Specifying a rate of jerk is generally better if you want to be able to break into executing moves smoothly. Specifying an S-curve time is generally better if you want to coordinate jog moves on multiple motors.

If **Motor[*x*].JogTs** is less than zero, specifying a maximum "jerk", **Motor[*x*].JogTa** must also be less than zero, specifying a maximum acceleration. If **Motor[*x*].JogTs** is set to 0.0, there is no S-curve acceleration portion, and the jog acceleration and deceleration rates will be constant.

If **Motor[*x*].JogTa** specifies a time and **Motor[*x*].JogTs** (S-curve time) specifies a smaller time, the total time spent in acceleration will be equal to the sum of **JogTa** and **JogTs**. (Note that this rule is different from Turbo PMAC.) However, if **Motor[*x*].JogTa** specifies a time and **Motor[*x*].JogTs** specifies a larger time, the total time spent in acceleration will be 2 times **Motor[*x*].JogTs**. Therefore, if **Motor[*x*].JogTa** is set to 0, **Motor[*x*].JogTs** alone controls the acceleration time in "pure" S-curve form.

A change in this parameter will not take effect until the next move command. For instance, if you wanted a different deceleration time from acceleration time in a jog move, you would specify the acceleration time, command the jog, change the deceleration time, then command the jog move again (e.g. `j=`), or at least the end of the jog (`j/`).

Example

To set a jog jerk rate of 80 m/s^3 (about 8g/sec) with motor units of millimeters:

$$JogTs = -\frac{s^3}{80m} * \frac{10^{-3} m}{motor\_unit} * \frac{10^9 ms^3}{s^3} = -12{,}500\left(\frac{ms^3}{motor\_unit}\right)$$

Motor[*x*].LimitBits

Description: Bit number of first limit signal in **pLimits** register

Range: 0 .. 255

Units: Field of flags and bit number (little-endian)

Default: Auto-configured based on hardware

Motor[*x*].LimitBits determines which bits of the 32-bit register whose address is specified by **Motor[*x*].pLimits** Power PMAC will use to detect the hardware overtravel-limit inputs for the motor and how it will use them.

If **Motor[*x*].LimitBits** is in the range of 0 to 30, the value of **LimitBits** specifies the bit number of the positive limit flag input; Power PMAC will use the next higher bit for the negative limit flag input. (If bit 7 – value 128 – is set to invert the polarity of the inputs, yielding a value of 128 to 158, **LimitBits** minus 128 specifies the first bit number.) The bit number is specified in the

"little-endian" convention, where bit *n* has a value of 2^n in the 32-bit word. This is by far the most common range setting, as all Delta Tau flag interfaces have the positive limit input bit mapped into the lower of two adjacent bits.

If **Motor[*x*].LimitBits** is in the range of 32 to 63, the value of **LimitBits** minus 32 specifies the bit number of the single limit flag input (connected to both switches) that will inhibit motion in both directions. (If bit 7 – value 128 – is set to invert the polarity of the inputs, yielding a value of 160 to 191, **LimitBits** minus 160 specifies the bit number.) Note that with a single input for both ends, it is not possible to command Power PMAC out of a limit without first disabling the hardware limit function. For example, if **Motor[*x*].LimitBits** is set to 40, bit 8 (= 40 – 32) of the selected register will be used for both limits.

If **Motor[*x*].LimitBits** is in the range of 64 to 94, the value of **LimitBits** minus 64 specifies the bit number of the negative limit flag input; Power PMAC will use the next higher bit for the positive limit flag input. (If bit 7 – value 128 – is set to invert the polarity of the inputs, yielding a value of 192 to 222, **LimitBits** minus 192 specifies the first bit number.) For example, if **Motor[*x*].LimitBits** is set to 80, bit 16 (= 80 – 64) of the selected register will be used for the negative limit and bit 17 for the positive limit.

If **Motor[*x*].LimitBits** is in the range of 96 to 127, the value of **LimitBits** minus 96 specifies the bit number of the positive limit flag input in the register specified by **Motor[*x*].pLimits**. (If bit 7 – value 128 – is set to invert the polarity of the inputs, yielding a value of 224 to 155, **LimitBits** minus 224 specifies the first bit number.) In this case, the register for the negative limit flag input is specified by **Motor[*x*].pAuxFault**, and the bit within that register is specified by **Motor[*x*].AuxFaultBit**. This range is intended for limit switches wired into general-purpose I/O points, particularly when transferred to Power PMAC over a network on a large system.

In this case, the settings of **Motor[*x*].AuxFaultLevel** and **Motor[*x*].AuxFaultLimit** are not used in processing the negative limit input. The polarity is controlled by bit 7 of **LimitBits**, and a single scan in the limit state will cause a trip. Action on hitting this limit – abort or kill – is determined by **Motor[*x*].FaultMode**. If **Motor[*x*].pAuxFault** is at its default value of 0 (not pointing to any register), Power PMAC will use the next higher bit in the register specified by **Motor[*x*].pLimits** for the negative limit input.

For example, if **Motor[*x*].LimitBits** is set to 121, bit 25 (= 121 – 96) of the register selected by **Motor[*x*].pLimits** is used for the positive limit. The bit specified by **Motor[*x*].AuxFaultBit** in the register selected by **Motor[*x*].pAuxFault** is used for the negative limit.

If bit 7 (value 128) of **Motor[*x*].LimitBits** is set to 1, the normally expected polarity of the limit input bits is reversed, so that a "0" in the bit means that the motor is into that limit. If bit 7 is set to its default value of 0, a "1" in the bit means that the motor is into that limit.

Caution

Delta Tau strongly recommends the use of "normally closed" limit switches that open when the limit is reached, because limit switch circuit failures are much more likely to result in a false "open" state, resulting in a safe stopping condition.

All Delta Tau limit-flag interface circuits yield a bit value of 0 for a closed state and a bit value of 1 for an open state. Using the recommended normally closed switches, bit 7 of **LimitBits** would be set to its default value of 0.

Some third-party interfaces, especially those in networked drives, will provide a "0" when into the limit and a "1" when not. For these interfaces, bit 7 of **LimitBits** should be set to 1. For example, if the positive limit is found in bit 20 of the 32-bit register, and the negative limit in bit 21, with a "0" in the bit indicating the motor is in the limit, **Motor[*x*].LimitBits** should be set to 148 (128 + 20).

In the large majority of applications, the value of **Motor[*x*].LimitBits** will be specified properly as part of the hardware auto-detection and configuration done at re-initialization (on the **$$$***** command), either done at the factory or by the user at system assembly, or subsequently. Also, when the value of **Motor[*x*].EncType** is set through the Power PMAC script environment, whether from an on-line command, buffered program command, or from the saved value at power-on/reset, this value will be set to the standard value for the type of hardware interface (e.g. PMAC2-style, PMAC3-style, MACRO) specified by the value of **Motor[*x*].EncType**.

To use the standard overtravel-limit input signals of a PMAC2-style "DSPGATE1" Servo IC, as in an ACC-24E2, ACC-24E2A, or ACC-24E2S, **Motor[*x*].LimitBits** should be set to 25. (Note the signal is mapped into bit 17 of the 24-bit word in the IC, but this word is in the high 24-bits of the 32-bit Power PMAC bus, so it appears as bit 25 to the software.)

To use the standard overtravel-limit input signals of a PMAC3-style "DSPGATE3" IC, as in an ACC-24E3 or Power PMAC "Brick", **Motor[*x*].LimitBits** should be set to 9.

To use the standard overtravel-limit bits of the MACRO-ring protocol, whether through a PMAC2-style MACRO IC (as on the ACC-5E) or a PMAC3-style IC (as on the ACC-5E3) **Motor[*x*].LimitBits** should be set to 25.

Values of 32 and greater for **Motor[*x*].LimitBits** are new in V2.1 firmware, released 1st quarter 2016.

Motor[*x*].MasterCtrl

Description: Control flag to activate position following

Range: 0 .. 15

Units: Bit field

Default: 0

Legacy I-variable alias: I*x*06

Motor[*x*].MasterCtrl determines whether or not the position-following (electronic-gearing) function is enabled for the specified motor, and how the function is to be performed. It is a 4-bit value, but presently only two bits are used.

Bit 0 (value 1) controls whether the following is enabled or not. If set to 1, the motor will track changes to the value in the register addressed by **Motor[*x*].pMasterEnc** according to the gear ratio set in **Motor[*x*].MasterPosSf**. If set to 0, the position-following function is disabled for the motor, and it will not track any changes to the addressed master register.

Bit 1 (value 2) controls whether the following is done in "normal" or "offset" mode. If set to 0, normal mode is specified, and the reference position for programmed moves does not change as the motor tracks the master position. This means that subsequent programmed moves will cancel out the changes due to the following function.

If bit 1 is set to 1, offset mode is specified, and the reference position for programmed moves changes along with the following moves. In this mode, subsequent programmed moves are added on top of the position changes due to the following function. This permits the superimposition of programmed and following moves.

Note that the following-mode bit is important even when following is disabled, because it affects how subsequent programmed moves are calculated. If the following mode is ever changed, a **`pmatch`** position-matching command must be executed before the next programmed move is calculated. Otherwise, that move will use the wrong starting value for its starting position, and a potentially dangerous jump will occur at the beginning of the move. A **`pmatch`** command is automatically executed whenever a motion program is started, but if the change is made in the middle of a motion program, it must be commanded explicitly.

Bits 2 and 3 are reserved for future use.

Motor[*x*].MasterCtrl constitutes bits 20 – 23 of the full-word element **Motor[*x*].Control[0]**.

Motor[*x*].MasterMaxAccel

Description: Position following maximum acceleration magnitude

Range: Non-negative floating-point

Units: Motor units per servo cycle per servo cycle

Default: 0.0 (acceleration limiting disabled)

Motor[*x*].MasterMaxAccel specifies the magnitude of the maximum change in motor velocity that can result from position following in any servo cycle, and therefore sets the maximum acceleration that can result from position following. It also controls whether any "excess" following resulting from following speed limiting is discarded or retained. It is only used if **Motor[*x*].MasterMaxSpeed** is set greater than 0.0, enabling velocity limiting in the position following.

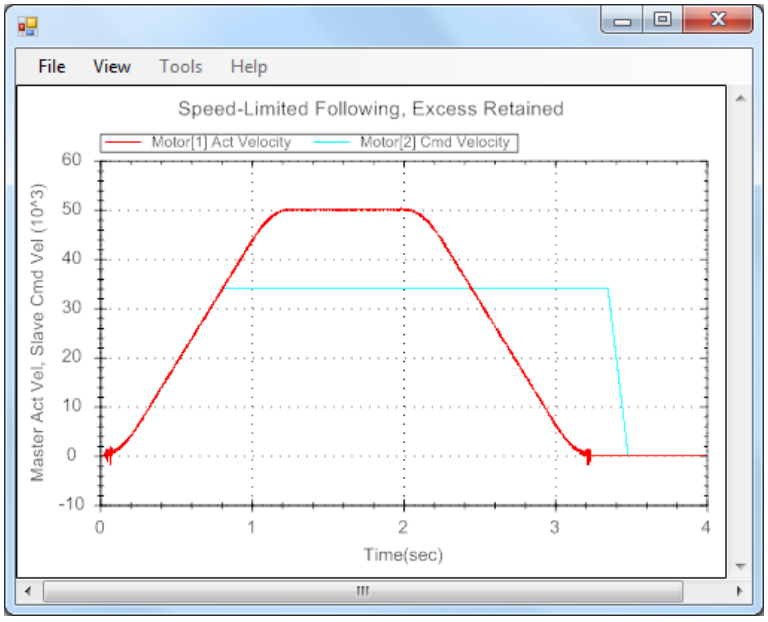
If **Motor[*x*].MasterMaxAccel** is set to the default value of 0.0, this acceleration limiting is disabled, and the full contribution of position following to the motor commanded acceleration is always used. In addition, if following speed limiting is enabled by **Motor[*x*].MasterMaxSpeed**, any difference in following distance between the unlimited command and the limited speed actually used is discarded, so that position synchronization with the master is lost.

If **Motor[*x*].MasterMaxAccel** is set to a value greater than 0.0, acceleration limiting is enabled, and each servo cycle, the change in the magnitude of the following from the previous servo cycle, as computed from the change in the master and the active following ratio, is compared to **Motor[*x*].MasterMaxAccel**. If the magnitude of the change in following is larger than this limit, the value of the limit (in the direction of the following) is used instead, thus providing a clamp on the acceleration resulting from following.

With following acceleration limiting active, any difference in following distance between the unlimited command and the limited speed and/or acceleration actually used is retained, and accumulated, so that it can be "released" when the following is no longer limited. The process of releasing the accumulated excess proceeds at the maximum rate that does not result in either the maximum following speed or acceleration being violated. When the excess is fully released, position synchronization between the motor and its master is restored.

As the excess from limiting is being accumulated, Power PMAC is not just considering the speed and acceleration at the moment, but the speed and acceleration profile required to bring the following profile to a stop within the specified limits, assuming each servo cycle that no further master position change occurs. This can cause the speed and/or acceleration of the following to be limited to lower values than those of the specified limits.

This 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**.



Motor[*x*].MasterMaxSpeed

Description: Position following maximum velocity magnitude

Range: Non-negative floating-point

Units: Motor units per servo cycle

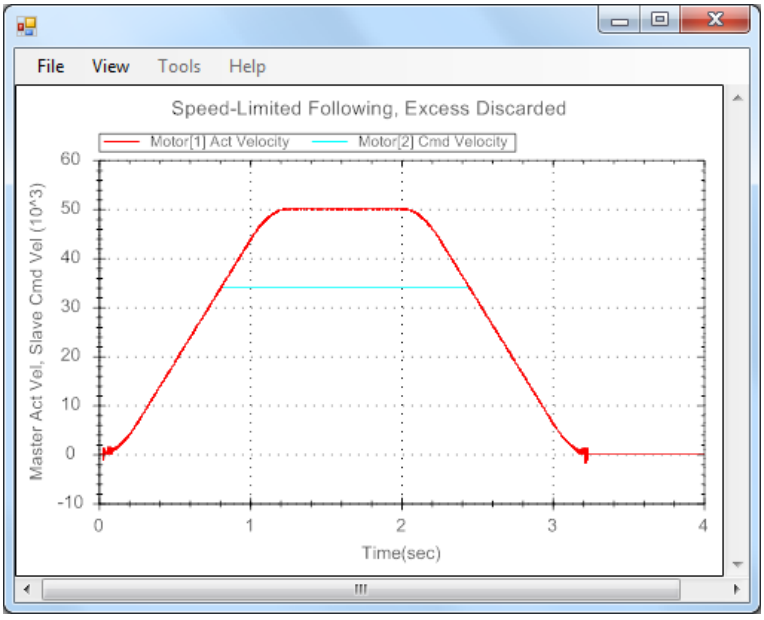
Default: 0.0 (speed limiting disabled)

Motor[*x*].MasterMaxSpeed specifies the magnitude of the maximum change in motor position that can result from position following in any servo cycle, and therefore sets the maximum speed that can result from position following. If **Motor[*x*].MasterMaxSpeed** is set to the default value of 0.0, this speed limiting is disabled, and the full contribution of position following to the motor commanded velocity is always used.

If **Motor[*x*].MasterMaxSpeed** is set to a value greater than 0.0, speed limiting is enabled, and each servo cycle, the magnitude of the following, as computed from the change in the master and the active following ratio, is compared to **Motor[*x*].MasterMaxSpeed**. If the magnitude of the following is larger than this limit, the value of the limit (in the direction of the following) is used instead, thus providing a clamp on the speed resulting from following.

Depending on the setting of **Motor[*x*].MasterMaxAccel**, the "excess" in the following value can be discarded, so that any position synchronization to the master is lost when the following speed is limited, but there will be no residual motion after the master has stopped, or the "excess" in the following value can be retained and "released" when the limit is no longer exceeded, so as to recover full position synchronization with the master.

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.



If trajectory-commanded motion for the motor (from jogging moves or programs) is superimposed on top of the position-following, only the component of motor motion resulting from the following is limited by this function.

Motor[*x*].MasterPosSf

Description: Desired master position scale factor

Range: Floating-point

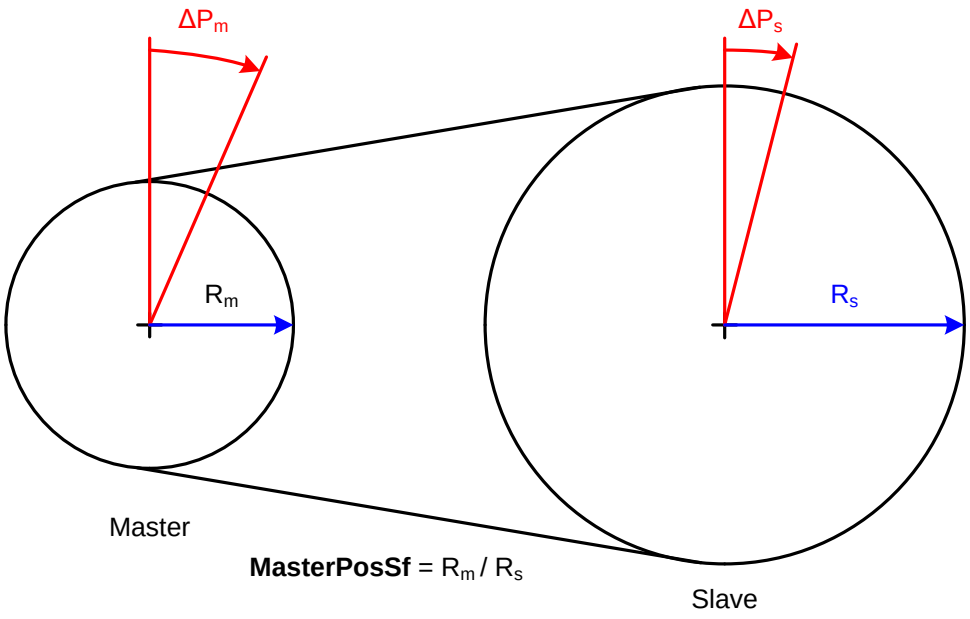
Units: Motor units per units of source data

Default: 1.0

Legacy I-variable alias: I*x*07

Motor[*x*].MasterPosSf specifies the desired scale factor by which the master position value read at the register specified by **Motor[*x*].pMasterEnc** is multiplied before being used as the master value in the position-following calculations for the motor. It effectively sets the "gear ratio" for the position following function.

The functionality can be envisioned through its mechanical analogy as shown in the following figure:



The source of position data is almost always the result register of an encoder conversion table (ECT) entry. Each entry has its own output scale factor, so there is a lot of flexibility as to what units are used in each stage of the process. However, the most common practice is for the floating-point output of the ECT entry to be scaled in "counts" of a counter, or LSBs of a data word or A/D converter, even if there is fractional resolution from some interpolation technique.

This element should be set to the number of motor units you want this motor to move for each unit of increment of the source master data register. For example, if the source master data is in counts of the master encoder counter, the motor is in units of counts of the feedback encoder counter, and you want the motor to move 5 units for each count of the master, you would set **Motor[*x*].MasterPosSf** to 5.0.

If saved setup element **Motor[*x*].SlewMasterPosSf** is set to the default value of 0.0, slew-rate control of the ratio is disabled, and changes in the value of **Motor[*x*].MasterPosSf** take place immediately. However, if **Motor[*x*].SlewMasterPosSf** is set to a positive value, the actual ratio used each servo cycle (found in status element **Motor[*x*].ActiveMasterPosSf**) can change each servo cycle only by the magnitude of **Motor[*x*].SlewMasterPosSf**.

Motor[*x*].MaxDac

Description: Instantaneous servo output limit

Range: 0 .. 32,767.999

Units: 16-bit DAC equivalent

Default: 28,000.0

Legacy I-variable alias: I*x*69

Motor[*x*].MaxDac specifies the magnitude of the motor's maximum instantaneous command from the position/velocity servo loop. If the servo algorithm computes a command output value with a larger magnitude in any servo cycle, the magnitude of the command used will be limited to this value. **Motor[*x*].MaxDac** is used regardless of which servo algorithm is selected for use by **Motor[*x*].Ctrl**, standard or user-written.

Note that despite the name of this element, it is not required that D/A converters be used for output. If Power PMAC is performing commutation and/or current-loop calculations for this motor, **Motor[*x*].MaxDac** limits the magnitude of the torque input value into those algorithms.

If the magnitude of the closed-loop servo output is limited by **MaxDac** in a servo cycle, the integral action provided by **Motor[*x*].Servo.Ki** in the built-in servo algorithms is not used in the next servo cycle. This prevents the integrator from "overcharging", providing what is known as "anti-windup" protection.

The magnitude of open-loop outputs from an **`out`** command is limited to the value of **MaxDac**. So values greater than 100 (% of **MaxDac**) for the **`out`** command have the same result as an **`out100`** command.

Motor[*x*].MaxDac is in units of a 16-bit output, even if the actual resolution of the output device is different. For example, with 18-bit DAC outputs, a value for **Motor[*x*].MaxDac** of 10,000 is equivalent to a value of 40,000 written to the DAC.

When the servo-loop output is a velocity command, driving a "velocity-mode" amplifier, **Motor[*x*].MaxDac** acts as an instantaneous velocity command limit. When the servo-loop output is a torque/current command, driving a "torque-mode", "sinewave mode", or "direct-PWM

power-block" amplifier, **Motor[*x*].MaxDac** acts as an instantaneous torque/current command limit.

In sine-wave output mode (**Motor[*x*].PhaseCtrl** bit 0 or bit 2 = 1, **Motor[*x*].pAdc** = 0), amplifier gains are typically given in amperes of phase current per volt of PMAC output, but motor and amplifier limits are typically given in RMS amperage values. In this case, it is important to realize that peak phase current values are √2 (1.414) times greater than the RMS values.

In direct-PWM mode (**Motor[*x*].PhaseCtrl** bit 0 = 1, **Motor[*x*].pAdc** > 0) of 3-phase motors (**Motor[*x*].PhaseOffset** = +/-683), the corresponding top values of the sinusoidal phase-current ADC readings will be 1/cos(30°), or 1.15, times greater than the vector sum of quadrature and direct current. Therefore, once you have established the top values you want to see in the A/D converters your phase currents on a continuous basis, this value should be multiplied by cos(30°), or 0.866, to get your value for **Motor[*x*].MaxDac**. Remember that if current limits are given as RMS values, you should multiply these by √2 (1.414) to get peak phase current values.

For more details in the computation of **Motor[*x*].MaxDac** in the different modes of operation, refer to the User's Manual chapter *Making Your Application Safe*.

Examples

A torque-mode brushless motor amplifier has a $15A_{rms}$ maximum intermitten current capability (i.e. a ±10V input commands a $\pm 15A_{rms}$ current), but the brushless motor itself has a $10A_{rms}$ intermittent current limit, **MaxDac** is calculated as:

$$MaxDac = 32{,}768 * \frac{10A_{rms}}{15A_{rms}} = 21{,}845$$

A sine-wave input amplifier with a transconductance gain for each phase of 2 amps/volt could produce a current output of 20A (peak) on each phase from a full-range (±10V) sinewave command from a Power PMAC with **PwmSf** at the default setting of 32,767. This corresponds to an RMS magnitude of 20/*sqrt*(2) = $14.1A_{rms}$. If the motor has an intermittent current limit of $12A_{rms}$, **MaxDac** can be set by the following equation:

$$MaxDac = 32{,}768 * \frac{12A_{rms}}{14.1A_{rms}} = 27{,}887$$

For a 3-phase brushless servo motor with an $8A_{rms}$ intermittent current limit controlled by a direct-PWM amplifier in which the ADC reports its maximum instantaneous value of 32,768 at a phase current of 16.26A, **MaxDac** can be computed as:

$$MaxDac = 8 * \sqrt{2} * \frac{32{,}768}{16.26} * \cos(30°) = 19{,}745$$

The factor of sqrt(2) in this equation is from the conversion of instaneous current to RMS current. The factor of cos(30°) is from the 3-phase (abc) to 2-phase (dq) conversion.

Motor[x].MaxPos

Description: Positive position overtravel limit

Range: Floating-point

Units: Motor units

Default: 0.0 (= disabled when **Motor[*x*].MinPos** >= 0)

Legacy I-variable alias: I*x*13

Motor[*x*].MaxPos sets the farthest permitted position in the positive direction for the specified motor. If the motor is commanded to a position farther in the positive direction, Power PMAC will automatically abort that move. If it is able to detect the command soon enough, it will cause the motor to stop at the position defined by **Motor[*x*].MaxPos**.

At move calculation time, the endpoints of the calculated moves or segments are compared to **Motor[*x*].MaxPos** to decide if there is a violation. At move execution time, the instantaneous net desired position value is compared to (**Motor[*x*].MaxPos** + **Motor[*x*].SoftLimitOffset**) to decide if there is a violation.

Motor[*x*].MaxPos is expressed in motor units, relative to the motor zero position as defined by the most recent of: power-up/reset, homing or absolute position read, or commanded **`pset`**, **`pload`**, or **`pclr`** offset. Matrix transformations for an axis assigned to the motor do not affect the action of this limit. **Motor[*x*].MaxPos** must be greater than **Motor[*x*].MinPos** (in an absolute sense) in order for either limit to be active, so setting these two limits equal is a good way to disable them. This limit is not active during homing-search moves, until the home trigger is found. It is active during the post-trigger portion of the move.

The precise stopping action on hitting a software limit is dependent on whether the limit violation is detected at move calculation or move execution time, and if at calculation time, the type of move and the setting of **Coord[*x*].SoftLimitStopDis**. Refer to the section on software position limits in the User's Manual chapter *Making Your Application Safe*.

Motor[x].MaxSpeed

Description: Maximum programmed velocity magnitude

Range: Floating-point

Units: Motor units per millisecond

Default: 32.0

Legacy I-variable alias: I*x*16

Motor[*x*].MaxSpeed sets the maximum magnitude of velocity for several types of programmed moves. For **`linear`** mode moves, it serves as the limit for the motor, on a move-by-move basis if not segmented (**Coord[*x*].SegMoveTime** = 0), or on a segment-by-segment basis if segmented

(**Coord[*x*].SegMoveTime** > 0). If a programmed **linear** mode move requests a higher velocity magnitude of this motor, it will extend the time for the move just enough so that this limit is not violated. This will also reduce the velocity of any other motors in the coordinate system proportionately (even if those motors would not have violated their own limits) so that coordination and path are maintained.

Motor[*x*].MaxSpeed also sets the maximum magnitude of velocity for **circle** and **pvt** mode moves (as well as **linear** mode moves) on a segment-by-segment basis if the special segmented lookahead algorithm is active for the coordinate system.

If **Motor[*x*].MaxSpeed** is set to 0.0, no velocity limit checking is performed, either on a move-by-move basis, or a segment-by-segment basis.

Motor[*x*].MaxSpeed serves as the commanded velocity magnitude for **rapid** mode moves if **Motor[*x*].RapidSpeedSel** is set to the default value of 1 (otherwise **Motor[*x*].JogSpeed** is used).

Example

To set a maximum speed of 30 m/min (about 1200 in/min) with motor units of 10 microns:

$$MaxSpeed = \frac{30m}{\min} * \frac{motor\_unit}{10^{-5}m} * \frac{\min}{6*10^{4}ms} = 50\left(\frac{motor\_unit}{ms}\right)$$

Motor[*x*].MinPos

Description: Negative position overtravel limit

Range: Floating-point

Units: Motor units

Default: 0.0 (= disabled when **Motor[*x*].MaxPos** <= 0)

Legacy I-variable alias: I*x*14

Motor[*x*].MinPos sets the farthest permitted position in the negative direction for the specified motor. If the motor is commanded to a position farther in the negative direction, Power PMAC will automatically abort that move. If it is able to detect the command soon enough, it will cause the motor to stop at the position defined by **Motor[*x*].MinPos**.

At move calculation time, the endpoints of the calculated moves or segments are compared to **Motor[*x*].MinPos** to decide if there is a violation. At move execution time, the instantaneous net desired position value is compared to (**Motor[*x*].MinPos - Motor[*x*].SoftLimitOffset**) to decide if there is a violation.

Motor[*x*].MinPos is expressed in motor units, relative to the motor home position as defined by the most recent of: power-up/reset, homing or absolute position read, or commanded **pset**, **pload**, or **pclr** offset. Matrix transformations for an axis assigned to the motor do not affect the action of this limit. **Motor[*x*].MinPos** must be less than **Motor[*x*].MaxPos** (in an absolute sense) in order for either limit to be active, so setting these two limits equal is a good way to

disable them. This limit is not active during homing-search moves, until the home trigger is found. It is active during the post-trigger portion of the move.

The precise stopping action on hitting a software limit is dependent on whether the limit violation is detected at move calculation or move execution time, and if at calculation time, the type of move and the setting of **Coord[*x*].SoftLimitStopDis**. Refer to the section on software position limits in the User's Manual chapter *Making Your Application Safe*.

Motor[*x*].MotorMode

Description: Operational mode

Range: 0 .. 5

Units: Enumeration

Default: 0 (independent operation)

Motor[*x*].MotorMode specifies the operational mode of the motor, permitting the motor to act as a "network slave", accepting cyclic commands from a network such as a MACRO ring.

Motor[*x*].MotorMode can take the following values, specifying these modes of operation:

0: Independent operation (no cyclic commands)
1: Cyclic position commands*
2: Cyclic velocity commands
3: Cyclic torque/force commands
4: Cyclic phase-current ("sinewave") commands
5: Cyclic phase-voltage ("direct PWM") commands
* Not fully supported

If it is set to a value greater than 0, the motor will expect cyclic commands of the specified type starting at the register whose address is specified in saved setup element **Motor[*x*].pMotorNode**. It will also place feedback values starting at the register whose address is specified by the sum of the value of **Motor[*x*].pMotorNode** and the value of **Motor[*x*].MotorNodeOffset**.

If **Motor[*x*].MotorMode** is set to 1, the motor expects to find a position command value (from the trajectory generator) in the first command register each servo cycle, and command flags in the fourth command register. The second and third command registers are not used for automatic purposes in this mode. It will place its position feedback value, processed through the encoder conversion table, in the first feedback register, and status flags in the fourth feedback register. In this mode, the coordinating Power PMAC needs to output its trajectory commanded position each servo cycle, but not perform any loop closures or phase commutation. *Note that this mode is not fully supported in Power PMAC or the MACRO ring protocol. It is not recommended for use at this time.*

If **Motor[*x*].MotorMode** is set to 2, the motor expects to find a velocity command value (equivalent to the output of a position-only servo loop) in the first command register each servo cycle, and command flags in the fourth command register. The second and third command registers are not used for automatic purposes in this mode. It will place its position feedback

value, processed through the encoder conversion table, in the first feedback register, and status flags in the fourth feedback register. In this mode, the coordinating Power PMAC needs to close the outer position servo loop, but not the inner velocity loop, phase commutation, or current loop for the motor.

If **Motor[*x*].MotorMode** is set to 3, the motor expects to find a "torque" command value (equivalent to the output of a position/velocity servo loop) in the first command register each servo cycle, and command flags in the fourth command register. The second and third command registers are not used for automatic purposes in this mode. It will place its position feedback value, processed through the encoder conversion table, in the first feedback register, and status flags in the fourth feedback register. In this mode, the coordinating Power PMAC needs to close the full position/velocity servo loop, but not perform phase commutation or close the current loops for the motor.

If **Motor[*x*].MotorMode** is set to 4, the motor expects to find phase current command values in the first and second command registers each phase cycle, and command flags in the fourth command register. The third command register is not used in this mode. It will place its position feedback value, processed through the encoder conversion table, in the first feedback register, and status flags in the fourth feedback register. In this mode, the coordinating Power PMAC needs to close the position/velocity servo loop, and perform phase commutation for the motor, but not close the current loops.

If **Motor[*x*].MotorMode** is set to 5, the motor expects to find phase voltage ("PWM") command values in the first, second, and third command registers each phase cycle, and command flags in the fourth command register. It will place its position feedback value, processed through the encoder conversion table, in the first feedback register, current feedback values in the second and third feedback register, and status flags in the fourth feedback register. In this mode, the coordinating Power PMAC needs to close the position/velocity servo loop, perform phase commutation for the motor, and close the current loops.

Motor[*x*].MotorNodeOffset

Description: Cyclic network command difference between input and output addresses

Range: 0 .. 255

Units: Power PMAC addresses

Default: 0

Motor[*x*].MotorNodeOffset specifies the numerical difference between the command and feedback addresses for a motor used as a cyclic network slave. It is only used if **Motor[*x*].MotorMode** is set to a value greater than 0 to tell the motor to accept cyclic network commands.

For a motor in this mode, cyclic network commands are expected in the four consecutive registers starting at the address specified by **Motor[*x*].pMotorNode**. The motor will put its feedback values in the four consecutive registers starting at the address specified by the sum of the value of **Motor[*x*].pMotorNode** and the value of **Motor[*x*].MotorNodeOffset**.

If the registers used are in a PMAC2-style "DSPGATE2" MACRO IC, the feedback registers have the same addresses as the command registers (separate physical registers at the same addresses). In this case, **Motor[*x*].MotorNodeOffset** should be set to 0.

If the registers used are in a PMAC3-style "DSPGATE3" MACRO IC, the feedback registers have different addresses from the command registers. The **MacroOut*α*** registers of a node that are used for feedback start at an address 64 higher than the The **MacroIn*α*** registers of the same node that are used for commands In this case, **Motor[*x*].MotorNodeOffset** should be set to 64.

Motor[*x*].pAbsPhasePos

Description: Power-on absolute commutation position pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0

Legacy I-variable alias: I*x*81

Motor[*x*].pAbsPhasePos specifies the address of the register the motor will use for absolute power-on commutation rotor-angle position, if such information is present. This can be a different address from that of the ongoing commutation position, which is specified by **Motor[*x*].pPhaseEnc**, and the power-on position can have different resolution and even direction sense from the ongoing position.

Only synchronous motors (typically brushless servo motors) commutated by Power PMAC require the establishment of an absolute phase reference at power-on. This phase referencing can be done either by a "phasing-search move" or by the reading of an absolute sensor. If **Motor[*x*].pAbsPhasePos** is set to the default value of 0, or if **Motor[*x*].PhaseFindingTime** is set to a value greater than 4, the phase referencing will be accomplished through a phasing-search move.

However, if **Motor[*x*].pAbsPhasePos** is set to a real address, and **Motor[*x*].PhaseFindingTime** is set to a value less than 5, then the phase referencing will be accomplished by the reading of an absolute sensor. In this case, the data at the specified address will be interpreted according to **Motor[*x*].AbsPhasePosFormat**. The resulting value will be scaled by **Motor[*x*].AbsPhasePosSf**, and it will be offset by **Motor[*x*].AbsPhasePosOffset** to get the rotor-angle value.

Common settings of **Motor[*x*].pAbsPhasePos** are:

- **Gate1[*i*].Chan[*j*].Status.a** for PMAC2-style ASIC Hall-sensors (as on ACC-24E2x)
- **Gate3[*i*].Chan[*j*].Status.a** for PMAC3-style ASIC Hall-sensors (as on ACC-24E3)
- **Gate3[*i*].Chan[*j*].SerialEncDataA.a** for PMAC3-style serial encoder feedback (as on ACC-24E3)

- **Gate3[*i*].Chan[*j*].AtanSumOfSqr.a** for PMAC3-style resolver feedback (as on ACC-24E3)
- **GateIo[*i*].DataReg[*j*].a** for "IOGATE" parallel feedback (as on ACC-14E)
- **Acc28E[*i*].AdcUdata[*j*].a** for A/D-converter feedback (as on ACC-28E)
- **Sys.piom+*{offset}*** for a hardware register without a defined element
- **Motor[*x*].PrevPhaseEnc.a** for cases where the absolute phase position and the ongoing phase position use the same sensor read in the same manner.
- **EncTable[*i*].PrevEnc.a** for an already-processed position value, especially in cases where the absolute position value uses the same sensor as the ongoing servo position (as with the ACC-58E resolver-interface board).
- **Sys.pushm+*{offset}*** or **Sys.Udata[*i*].a** for a register in the user shared memory buffer for cases where the power-on phase position data must undergo transfer or processing that the automatic routines do not support.

If more than one register must be read for this position, **Motor[*x*].pAbsPhasePos** should contain the address of the register containing the least significant numerical value.

The absolute phase position value will be read when the on-line motor command **$** is issued, or when the data structure element **Motor[*x*].PhaseFindingStep** is set to 1. If bit 1 (value 2) of **Motor[*x*].PowerOnMode** is set to 1, it will be read automatically at power-on/reset.

If bit 0 (value 1) of **Motor[*x*].PowerOnMode** is set to 1, the motor will automatically enabled with the position loop closed after a (successful) read. If this bit is set to 0, the motor will be left in the killed state after a read.

Motor[*x*].pAbsPos

Description: Power-on absolute position pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0

Legacy I-variable alias: I*x*10

Motor[*x*].pAbsPos specifies the address of the register the motor will use for absolute power-on motor position, if such information is present. This can be a different address from that of the ongoing servo position feedback, which is specified by **Motor[*x*].pEnc** and **Motor[*x*].pEnc2**, and the power-on position can have different resolution and even direction sense from the ongoing position.

If **Motor[*x*].pAbsPos** is set to the default value of 0, no absolute position read is performed for the motor. In this case, a homing search move must be done to establish a position reference for the motor. With a value of 0 for this element, the on-line command **hmz** and the buffered program command **homez** simply set the present commanded position to zero.

However, if **Motor[*x*].pAbsPos** is set to a real address, then the position referencing will be accomplished by the reading of an absolute sensor. In this case, the data at the specified address will be interpreted according to **Motor[*x*].AbsPosFormat**. The resulting value will be scaled by **Motor[*x*].AbsPosSf**, and it will be offset by **Motor[*x*].HomeOffset** to get the final motor-position value.

Common settings of **Motor[*x*].pAbsPos** are:

- **Gate3[*i*].Chan[*j*].SerialEncDataA.a** for PMAC3-style serial encoder feedback (as on ACC-24E3)
- **GateIo[*i*].DataReg[*j*].a** for "IOGATE" parallel feedback (as on ACC-14E)
- **GateIo[*i*].AdcUdata[*j*].a** for A/D-converter feedback (as on ACC-28E)
- **Sys.piom+*{offset}*** for a hardware register without a defined element
- **ECAT[*i*].IO[*k*].Data.a** for absolute position from an EtherCAT device
- **EncTable[*i*].PrevEnc.a** for an already-processed position value in cases where the absolute position value uses the same sensor as the ongoing servo position.
- **Sys.pushm+*{offset}*** or **Sys.Udata[*i*].a** for a register in the user shared memory buffer for cases where the power-on phase position data must undergo transfer or processing that the automatic routines do not support.

If more than one register must be read for this position, **Motor[*x*].pAbsPos** should contain the address of the register containing the least significant numerical value.

The absolute position value will be read when the on-line motor command **hmz** is issued, or when the buffered program command **homez** is executed. If bit 2 (value 4) of **Motor[*x*].PowerOnMode** is set to 1, it will be read automatically at power-on/reset. If this automatic read is specified, the user must ensure the absolute sensor is always ready to send proper position values at this time.

Motor[*x*].pAdc

Description: Current feedback pointer

Range: 0, legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Legacy I-variable alias: I*x*82

Motor[*x*].pAdc enables digital current-loop closure and specifies which registers the selected motor uses for its digital current-loop feedback values if Power PMAC is performing commutation for the motor. It contains the address of the first of two registers used for two phases of current feedback. (The next register must be at the next higher address.) In this way, it determines which feedback device is used to provide current feedback.

If **Motor[*x*].pAdc** is set to 0, the Power PMAC will not close the current-loop for this motor, even if the commutation algorithm is active. If the commutation algorithm is active (**Motor[*x*].PhaseCtrl** bit 0 or bit 2 = 1) and **Motor[*x*].pAdc** = 0, then "sinewave output" commutation mode is enabled, where the current loop is closed in the amplifier. If the commutation algorithm is not active (**Motor[*x*].PhaseCtrl** bit 0 and bit 2 = 0), the setting of **Motor[*x*].pAdc** does not matter.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Most commonly, **Motor[*x*].pAdc** is set to the address of the first ADC register for a Servo IC channel (**Adc[0]** in a PMAC2-style IC, **AdcAmp[0]** in a PMAC3-style IC) or the first current-feedback register (register[1]) for a node in a MACRO IC. For example:

Motor[1].pAdc = Gate1[4].Chan[0].Adc[0].a
Motor[2].pAdc = Gate1[4].Chan[1].Adc[0].a
Motor[3].pAdc = Gate3[0].Chan[0].AdcAmp[0].a
Motor[4].pAdc = Gate3[0].Chan[1].AdcAmp[0].a
Motor[5].pAdc = Gate2[0].Macro[4][1].a
Motor[6].pAdc = Gate2[0].Macro[5][1].a
Motor[7].pAdc = Gate3[0].MacroInA[4][1].a
Motor[8].pAdc = Gate3[0].MacroInB[5][1].a
Motor[9].pAdc = 0

Motor[*x*].pAmpEnable

Description: Amplifier-enable (output) flag pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Motor[*x*].pAmpEnable specifies which register the selected motor uses for its amplifier-enable output flag signal. It contains the address of this register.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Most commonly, **Motor[*x*].pAmpEnable** is set to the address of a channel control register in a Servo IC. For example:

Motor[1].pAmpEnable = Gate1[4].Chan[0].Ctrl.a

Motor[2].pAmpEnable = Gate3[0].Chan[1].OutCtrl.a

If the motor is controlled through the MACRO ring, **Motor[*x*].pAmpEnable** is typically set to the address of a MACRO-node output flag register (output register 3 of the node). For example:

Motor[3].pAmpEnable = Gate2[0].Macro[4][3].a

Motor[4].pAmpEnable = Gate3[0].MacroOutA[5][3].a

If the motor is controlled through the EtherCAT network, **Motor[*x*].pAmpEnable** is typically set to the address of an EtherCAT holding register that is mapped to the control word of the EtherCAT drive. For example:

Motor[5].pAmpEnable = **ECAT[0].IO[25].Data.a**

In the specified register, Power PMAC uses the bit whose number is contained in **Motor[*x*].AmpEnableBit** to control the amplifier-enable signal.

Note that it is possible to specify the address of the specific bit within a register if it has its own element name (e.g. **Motor[5].pAmpEnable = Gate1[6].Chan[0].AmpEna.a**), but the value will report back as the address of the full-word element (e.g. **Gate1[6].Chan[0].Ctrl.a**).

On a **$$$***** re-initialization command, Power PMAC automatically assigns values of **Motor[*x*].pAmpEnable** based on the Servo ICs and MACRO ICs found.

If **Motor[*x*].pAmpEnable** is set to 0, this motor will not use any automatic amplifier-enable function.

Motor[*x*].pAmpFault

Description: Amplifier-fault (input) flag pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware
(= 0 – disabled – if no hardware available to assign)

Motor[*x*].pAmpFault specifies which register the selected motor uses for its amplifier-fault input flag signal. It contains the address of this register.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

The polarity of the amplifier fault input is set by **Motor[*x*].AmpFaultLevel**.

Most commonly, **Motor[*x*].pAmpFault** is set to the address of a channel status register in a Servo IC. For example:

Motor[1].pAmpFault = Gate1[4].Chan[0].Status.a

Motor[2].pAmpFault = Gate3[4].Chan[1].Status.a

If the motor is controlled through the MACRO ring, **Motor[*x*].pAmpFault** is typically set to the address of a MACRO-node input flag register (input register 3 of the node). For example:

Motor[3].pAmpFault = Gate2[0].Macro[4][3].a

Motor[4].pAmpFault = Gate3[0].MacroInA[5][3].a

If the motor is controlled through the EtherCAT network, **Motor[*x*].pAmpFault** is typically set to the address of an EtherCAT holding register that is mapped to the control word of the EtherCAT drive. For example:

Motor[5].pAmpEnable = **ECAT[0].IO[26].Data.a**

In the specified register, Power PMAC uses the bit whose number is contained in **Motor[*x*].AmpFaultBit** to read the amplifier-fault signal.

Note that it is possible to specify the address of the specific bit within a register if it has its own element name (e.g. **Motor[5].pAmpFault = Gate1[6].Chan[0].AmpFault.a**), but the value will report back as the address of the full-word element (e.g. **Gate1[6].Chan[0].Status.a**).

On a **$$$***** re-initialization command, Power PMAC automatically assigns values of **Motor[*x*].pAmpFault** based on the Servo ICs and MACRO ICs found.

If **Motor[*x*].pAmpFault** is set to 0, this motor will not use any automatic amplifier-fault function. On re-initialization, if no hardware is present for auto-assignment to the motor, this element will automatically be set to 0, disabling the amplifier fault input function for the motor.

Motor[*x*].pAuxFault

Description: Auxiliary fault register pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0 (auxiliary fault detection disabled)

Motor[*x*].pAuxFault specifies which register the motor looks to for its “auxiliary fault” bit. It contains the address of this register. If **Motor[*x*].pAuxFault** is set to the default value of 0, this motor will not use any automatic encoder-loss detection function.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

In the specified register, Power PMAC uses the bit whose number is contained in **Motor[*x*].AuxFaultBit** to read the fault status. The polarity of the encoder-loss status bit is set by **Motor[*x*].AuxFaultLevel**. The cumulative number of scans detecting the specified fault input state before an error is declared is specified by **Motor[*x*].AuxFaultLimit**.

Motor[*x*].pAuxFault has several possible uses. If dual-encoder feedback is used, **Motor[*x*].pEncLoss** can be used to detect the loss of one encoder, and **Motor[*x*].pAuxFault** to detect the other. If the physical motor has a thermal fault output, as can be detected through the Power Brick, **Motor[*x*].pAuxFault** can be used to detect this. Other fault inputs can be used as well.

If an auxiliary fault error is detected, this motor is automatically "killed" (disabled) by the Power PMAC. Other motors in the same coordinate system will either be killed or "aborted" (decelerated to a controlled stop), as determined by bit 1 (value 2) of **Motor[*x*].FaultMode** for this motor. The motor status bit **Motor[*x*].AuxFault** is set to 1.

For motor thermal sensors connected Power Brick PMAC3-style interface boards, which have processing circuitry on the "T" flag input, **Motor[*x*].pAuxFault** can be set to **Gate3[*i*].Chan[*j*].Status.a**. This register contains the **T** flag input bit in bit 15.

For quadrature encoders connected to ACC-24E2*x* PMAC2-style axis-interface boards, which have "exclusive-OR" quadrature loss detection circuits external to the ASIC, **Motor[*x*].pAuxFault** can be set to **Gate1[*i*].Chan[*j*].EncLossN.a**. (Alternately the board structure alias of **ACC24E2*x*[*i*].Chan[*j*].EncLossN.a** can be used.) Since this is not actually in the ASIC, the value reported back will be **Cid[*n*].PartData[*j*].a**. The encoder loss status bit is in bit 13 of this register.

For serial encoders connected to ACC-84E PMAC2-style serial-encoder-interface boards, which have several protocol-dependent loss-detection circuits, **Motor[*x*].pAuxFault** can be set to **ACC84E[*i*].Chan[*j*].SerialEncDataB.a**. Most commonly the "time-out" error bit in bit 31 of this register is used.

For quadrature encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which have "exclusive-OR" quadrature loss detection circuits in the ASIC, **Motor[*x*].pAuxFault** can be set to **Gate3[*i*].Chan[*j*].Status.a**. (Alternately the board structure alias of **ACC24E3[*i*].Chan[*j*].Status.a** can be used.) This register contains the **LossStatus** transparent quadrature-loss bit in bit 28.

For analog sinusoidal encoders or resolvers connected to ACC-24E3 PMAC3-style axis-interface boards, which have "sum-of-squares" loss detection circuits, **Motor[*x*].pAuxFault** can be set to **Gate3[*i*].Chan[*j*].Status.a**. (Alternately the board structure alias of **ACC24E3[*i*].Chan[*j*].Status.a** can be used.) This register contains the **SosError** transparent "sum-of-squares" error bit in bit 31.

For serial encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which have several protocol-dependent loss-detection circuits, **Motor[*x*].pAuxFault** can be set to **Gate3[*i*].Chan[*j*].SerialEncDataB.a**. (Alternately the board structure alias of

ACC24E3[*i*].Chan[*j*].SerialEncDataB.a can be used.) Most commonly the "time-out" error bit in bit 31 of this register is used.

The Type = 12 encoder conversion table entry (new in V2.0 firmware, released 1st quarter 2015) can monitor each servo cycle for multiple error bits in a single register. If any of these bits is set, the entry sets bit 0 of **EncTable[*m*].Status** to 1, and this bit can be used as the auxiliary-fault bit for the motor.

If **Motor[*x*].LimitBits** is set to a value in the range of 96 to 127 or 224 to 255, the positive and negative hardware overtravel limit input bits are specified to come from two separate registers. In this case, **Motor[*x*].pAuxFault** specifies the register for the negative limit input bit. (**Motor[*x*].pLimits** still specifies the register for the positive limit input bit.) **Motor[*x*].AuxFaultBit** specifies which bit of the 32-bit register is used for the negative limit input bit.

When the auxiliary fault input is used for the negative overtravel limit in this way, **Motor[*x*].AuxFaultLevel** and **Motor[*x*].AuxFaultLimit** are not used. The polarity of the input is controlled by bit 7 (value 128) of **Motor[*x*].LimitBits** and the first scan in which the limit is detected will create a fault. Action on hitting the limit – aborting or killing – is controlled by bit 2 (value 4) of **Motor[*x*].FaultMode**.

Note that it is possible to specify the address of the specific bit within a register if it has its own element name (e.g. **Motor[5].pAuxFault = Gate3[1].Chan[0].T.a**), but the value will report back as the address of the full-word element (e.g. **Gate3[1].Chan[0].Status.a**).

Motor[*x*].pBrakeOut

Description: Motor-brake (output) flag pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0

Motor[*x*].pBrakeOut specifies which register the selected motor uses for its motor-brake output flag signal. It contains the address of this register. If **Motor[*x*].pBrakeOut** is set to the default value of 0, this motor will not use any automatic motor brake control function.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Most commonly, **Motor[*x*].pBrakeOut** is set to the address of a channel output control register in a PMAC3-style "DSPGATE3" IC. It can also be set to the address of a general-purpose I/O card output register, as on an ACC-11E or ACC-68E. For example:

Motor[1].pBrakeOut = Gate3[0].Chan[0].OutCtrl.a

Motor[2].pBrakeOut = GateIo[2].DataReg[3].a

Motor[3].pBrakeOut = Sys.piom + $A0000C

In the specified register, Power PMAC uses the bit whose number is contained in **Motor[*x*].BrakeOutBit** to control the brake-output signal.

Note that it is possible to specify the address of the specific bit within a register if it has its own element name (e.g. **Motor[5].pBrakeOut = Gate3[1].Chan[0].OutFlagB.a**), but the value will report back as the address of the full-word element (e.g. **Gate3[1].Chan[0].OutCtrl.a**).

The specified brake output bit always has a value of 0 for the brake-engaged state and a value of 1 for the brake-released state. This software polarity cannot be changed. It is the user's responsibility to implement proper hardware to use this output in a compatible and safe manner. This bit is always set to 0 on power-up/reset to put the brake in an engaged state. If the controller is put in a "reset" state either by user input or by a failure such as a watchdog timer trip, the output is forced to the 0 state.

The timing of the automatic brake-release and brake-engage functions as the motor is enabled and disabled is controlled by saved setup elements **Motor[*x*].BrakeOffDelay** and **Motor[*x*].BrakeOnDelay**. For special operations such as phasing-search moves, the output bit must be written to "manually" (e.g. **`Gate3[0].Chan[0].OutFlagB = 1`**, **`Gate3[0].Chan[0].OutCtrl |= $20`**, **`GateIo[2].DataReg[0] |= 2`**). Note that these operations are typically not recommended for motors requiring a brake.

If the automatic brake-control function is enabled, the user should not attempt to enable a disabled motor with an actual motor move command such as **`j+`** or **`hm`**.

Motor[*x*].pBufPos

Description: Pointer to motor outer-loop position extended storage buffer

Range: 0, **Sys.BufPos[*i*][0].a**, ***i*** = 0 to 7

Units: Power PMAC addresses

Default: 0

Motor[*x*].pBufPos specifies the address of the start of the extended position storage buffer, if any, that this motor will use to store its outer-loop actual position value each servo cycle. If it is set to 0, the motor will not use any extended storage buffer for this.

If **Motor[*x*].pBufPos** is set to the address of the first element of one of the 8 **Sys.BufPos** extended storage buffers (that is, to **Sys.BufPos[*i*][0].a**, where ***i*** = 0 to 7), then the motor will store the value of **Motor[*x*].ActPos** each servo cycle to the status element **Sys.Buf[*i*][*j*]**. The first index (***i***) is the buffer number. The second index (***j***) is the servo cycle index and has a range of 0 to 255. This index value is equal to the value in the low 8 bits of **Sys.ServoCount**, which increments every servo cycle. The buffer is rotary, with the value from 256 (2^8) cycles previously being overwritten each cycle, so there is always a history of the most recent 256 servo cycles.

This extended buffering function is not used for any automatic features, but it can be used for user-specific features that require significant position history. Note that each motor always

buffers the actual positions for the most recent 16 servo cycles in its own array **Motor[*x*].FltrPos[*k*]**.

If multiple motors are configured to write to the same buffer with their inner or outer-loop position, the higher-numbered motor will overwrite the value from the lower-numbered motor each servo cycle. There is no protection against this type of overwriting.

Motor[*x*].pBufPos2

Description: Pointer to motor inner-loop position extended storage buffer

Range: 0, **Sys.BufPos[*i*][0].a**, ***i*** = 0 to 7

Units: Power PMAC addresses

Default: 0

Motor[*x*].pBufPos2 specifies the address of the start of the extended position storage buffer, if any, that this motor will use to store its inner-loop actual position value each servo cycle. If it is set to 0, the motor will not use any extended storage buffer for this.

If **Motor[*x*].pBufPos2** is set to the address of the first element of one of the 8 **Sys.BufPos** extended storage buffers (that is, to **Sys.BufPos[*i*][0].a**, where ***i*** = 0 to 7), then the motor will store the value of **Motor[*x*].ActPos2** each servo cycle to the status element **Sys.Buf[*i*][*j*]**. The first index (***i***) is the buffer number. The second index (***j***) is the servo cycle index and has a range of 0 to 255. This index value is equal to the value in the low 8 bits of **Sys.ServoCount**, which increments every servo cycle. The buffer is rotary, with the value from 256 (2^8) cycles previously being overwritten each cycle, so there is always a history of the most recent 256 servo cycles.

This extended buffering function is not used for any automatic features, but it can be used for user-specific features that require significant position history. Note that each motor always buffers the actual positions for the most recent 16 servo cycles in its own array **Motor[*x*].FltrPos2[*k*]**.

If multiple motors are configured to write to the same buffer with their inner or outer-loop position, the higher-numbered motor will overwrite the value from the lower-numbered motor each servo cycle. There is no protection against this type of overwriting.

Motor[x].pCaptEna

Description: Encoder capture trigger arming pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0 (disabled)

Motor[*x*].pCaptEna specifies which register the motor uses for a manual arming of a position capture trigger at the start of a triggered move (homing-search move, jog-until-trigger, programmed rapid-mode move-until-trigger). If it is set to its default value of 0, no manual arming is performed.

If the position capture is done using a Delta Tau "DSPGATE*n*" ASIC or done over the MACRO ring using standard remote devices, this arming is done automatically, and **Motor[*x*].pCaptEna** can be left at its default value of 0. However, when Power PMAC is interfaced to other devices that perform the capture, as over an EtherCAT or similar motion network, this manual arming may be required.

When **Motor[*x*].pCaptEna** is set to a valid address, then immediately before the start of a triggered move, Power PMAC will write to the bit in this register specified by **Motor[*x*].CaptEnaBit**. If **Motor[*x*].CaptEnaInvert** is set to 0, it will write a 1 to this bit. If **Motor[*x*].CaptEnaInvert** is set to 1, it will write a 0 to this bit. This value should prepare the circuitry for a trigger that could happen as soon as the move starts.

When the trigger is found, or when the move ends without finding a trigger, Power PMAC will automatically set this bit to the opposite value.

When the position capture trigger is done over the EtherCAT network on a drive that conforms to the DS-402 standard, **Motor[*x*].pCaptEna** is typically set to the address of the EtherCAT I/O data structure element that is mapped to the "touch probe" control register ($60B8_h$) for the motor drive, e.g.:

Motor[*x*].pCaptEna = ECAT[*i*].IO[*k*].Data.a

In this case, **Motor[*x*].pCaptFlag** is usually set to the address of the EtherCAT I/O data structure element that is mapped to the touch probe status register ($60B9_h$) for this drive, and **Motor[*x*].pCaptPos** is set to the address of the element that is mapped to one of the touch probe position registers ($60BA_h$, $60BB_h$, $60BC_h$, or $60BD_h$).

Motor[*x*].pCaptEna is new in V2.1 firmware, released 1st quarter 2016.

Motor[*x*].pCaptFlag

Description: Encoder capture trigger (input) flag pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Motor[*x*].pCaptFlag specifies which register the selected motor uses for its hardware capture trigger-flag signals. It contains the address of this register. In this way, it determines which set of flag signals is used for position capture triggering for this motor. Power PMAC will use this register to detect the capture trigger in triggered moves such as homing search moves.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

The related element **Motor[*x*].CaptFlagBit** tells Power PMAC which bit on the 32-bit bus for this register to use as the trigger bit for these moves. When the trigger is found, Power PMAC will use the register specified in **Motor[*x*].pCaptPos** for the captured position if hardware capture is specified.

Most commonly, **Motor[*x*].pCaptFlag** is set to the address of a channel status register in a Servo IC, or the input flag register of a MACRO node. For example:

Motor[1].pCaptFlag = Gate1[4].Chan[0].Status.a

Motor[2].pCaptFlag = Gate3[0].Chan[1].Status.a

Motor[3].pCaptFlag = Gate2[0].Macro[4][3].a

Motor[4].pCaptFlag = Gate3[2].MacroInA[8][3].a

If the motor is controlled through the EtherCAT network, **Motor[*x*].pCaptFlag** can be set to the address of an EtherCAT holding register that is mapped to the touch probe status register ($60B9_h$) for the EtherCAT drive. For example:

Motor[5].pCaptFlag = **ECAT[0].IO[26].Data.a**

Note that it is possible to specify the address of the specific bit within a register if it has its own element name (e.g. **Motor[5].pCaptFlag = Gate1[6].Chan[0].PosCapt.a**), but the value will report back as the address of the full-word element (e.g. **Gate1[6].Chan[0].Status.a**).

On the **$$$***** re-initialization command, **Motor[*x*].pCaptFlag** is automatically assigned to the status register of an auto-detected servo channel or MACRO node according to pre-determined ordering rules.

If the value of the setup element **Motor[*x*].EncType** is set through the Script environment (not from a C program), **Motor[*x*].pCaptFlag** is automatically set to the same address as **Motor[*x*].pEncStatus**.

Motor[*x*].pCaptPos

Description: Encoder captured position pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Motor[*x*].pCaptPos specifies which register the selected motor reads to get its hardware-captured position in triggered moves such as homing-search moves. It contains the address of this

register. The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

If the captured position for triggered moves is obtained over the MACRO ring, Power PMAC obtains the position through a software request over the ring, and **Motor[*x*].pCaptPos** is not used.

If software capture is used for triggered moves (**Motor[*x*].CaptureMode** > 0), **Motor[*x*].pCaptPos** is not used.

The related element **Motor[*x*].pCaptFlag** tells Power PMAC which register to read for the capture trigger bit to tell it that the trigger has occurred. Typically it uses the status-flag register for the same Servo IC channel as the captured position register.

Once the captured position is obtained from this register, it is processed using the values of **Motor[*x*].CaptPosRightShift**, **Motor[*x*].CaptPosLeftShift**, and **Motor[*x*].CaptPosRound**.

Most commonly, **Motor[*x*].pCaptPos** is set to the address of a captured-position register in a Servo IC. For example:

Motor[1].pCaptPos = Gate1[4].Chan[0].HomeCapt.a
Motor[2].pCaptPos = Gate3[0].Chan[1].HomeCapt.a

If the motor is controlled through the EtherCAT network, **Motor[*x*].pCaptPos** can be set to the address of an EtherCAT holding register that is mapped to to one of the touch probe position registers ($60BA_h$, $60BB_h$, $60BC_h$, or $60BD_h$)for the EtherCAT drive. For example:

Motor[5].pCaptPos = **ECAT[0].IO[30].Data.a**

On the **$$$***** re-initialization command, **Motor[*x*].pCaptPos** is automatically assigned to the captured-position register of an auto-detected servo channel according to pre-determined ordering rules.

If the value of the setup element **Motor[*x*].EncType** is set through the Script environment (not from a C program), **Motor[*x*].pCaptPos** is automatically set to the address of the captured position register for the same servo channel as used by **Motor[*x*].pEncStatus**.

Motor[x].pCascadeCmd

Description: Cascaded servo command register pointer

Range: Legitimate addresses

Units: Power PMAC addresses

Default: 0 (cascaded servo function disabled)

Motor[*x*].pCascadeCmd specifies which register (if any) the selected motor uses to output its servo command as a double-precision floating-point value. It contains the address of this register.

The value of this variable is usually set by assigning it to the address of the data structure element to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

This functionality is separate from the command output addressing functionality of **Motor[*x*].pDac**. The register specified by that element must be used in integer format, as actual output registers such as DAC and PWM command registers are formatted.

The intent of **Motor[*x*].pCascadeCmd** is to support the direct "cascading" of multiple servo loops in Power PMAC, without the need to go through intermediate constructs such as holding registers, compensation tables, or encoder conversion table entries, or the conversion back and forth between integer and floating-point formats. This makes the cascading both simpler to set up and more efficient to execute.

The most common target registers for **Motor[*x*].pCascadeCmd** are compensation or offset registers of another motor. Typical settings would be:

Motor[*x*].pCascadeCmd = **Motor[*y*].CompDesPos.a**
Motor[*x*].pCascadeCmd = **Motor[*y*].ActiveMasterPos.a**
Motor[*x*].pCascadeCmd = **Motor[*y*].CompDac.a**

The user must take care that the selected register is not used by any other functionality, such as a compensation table or position following.

If the number of the target motor (***y***) is greater than the number of this motor (***x***), this cascaded value will be used by the target motor in the same servo cycle, so no delay will be added.

If **Motor[*x*].pCascadeCmd** is set to its default value of 0, Power PMAC will not write the calculated servo command to any floating-point register.

If related element **Motor[*x*].CascadeMode** is set to 0, the computed value will overwrite the value in the target register. If **CascadeMode** is set to 1, the computed value will be added to the value already in the register.

Regardless of whether **Motor[*x*].pCascadeCmd** is set to enable or disable the cascading floating-point write of the servo command, the motor will still perform the integer output of the servo command value to the register specified by **Motor[*x*].pDac**, even if this register would not be used for any purpose.

Motor[*x*].pCascadeCmd is new in V2.1 firmware, released 1st quarter 2016. In older versions, the value to be used in cascaded servo loops had to be processed through the encoder conversion table or a "0D" compensation table.

Motor[*x*].Pd*i*

Description: Pre-filter polynomial i^{th}-order denominator term (i = 1 to 4)

Range: Floating-point

Units: none (unit-less z-transform coefficient)

Default: 0.0

The **Motor[*x*].Pd*i*** terms are double-precision coefficients of the 4th-order polynomial denominator term acting on the commanded trajectory position before it is sent the servo algorithm. The transfer function of this filter is:

$$P(z) = 1 - (1 - z^{-1}) \frac{Pn_0 + Pn_1 z^{-1} + Pn_2 z^{-2} + Pn_3 z^{-3} + Pn_4 z^{-4}}{1 + Pd_1 z^{-1} + Pd_2 z^{-2} + Pd_3 z^{-3} + Pd_4 z^{-4}}$$

To use any of these terms, **Motor[*x*].PreFilterEna** must be set to a value greater than 0 to enable the pre-filter for this motor. The value of **Motor[*x*].PreFilterEna** specifies the update period of the filter in servo interrupt periods.

The sum of all **Pn*i*** terms minus the sum of all **Pd*i*** terms must equal exactly 1.0 in order for the filter not to have any net scaling effect on the trajectory "signal" (i.e. to have a "DC gain" of 1.0). Typically, **Pn1** and higher, and **Pd1** and higher are set to obtain the desired dynamics, then **Pn0** is set to 1.0 minus the sum of the higher-order **Pn*i*** terms plus the sum of the **Pd*i*** terms so there is no net scaling due to the filter.

Motor[*x*].pDac

Description: Command output pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Legacy I-variable alias: I*x*02

Motor[*x*].pDac specifies which register (or the first of a set of registers) the selected motor uses for its command output value(s). It contains the address of this register or the first (lowest address) of consecutively addressed multiple registers. In this way, it determines which output signal(s) are used to transmit the commanded output values. Note that despite the name of this element, it is not required that D/A converters be used for output.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Most commonly, **Motor[*x*].pDac** is set to the address of an output register in a Servo IC or a MACRO IC, or to an EtherCAT holding register that is mapped to the command register for an EtherCAT drive. For example:

Motor[1].pDac = Gate1[4].Chan[0].Dac[0].a
Motor[2].pDac = Gate1[4].Chan[1].Pwm[0].a
Motor[3].pDac = Gate1[4].Chan[2].Pfm.a
Motor[4].pDac = Gate2[0].Macro[5][0].a

Motor[5].pDac = Gate3[0].Chan[0].Dac[0].a
Motor[6].pDac = Gate3[0].Chan[1].Pwm[0].a
Motor[7].pDac = Gate3[0].Chan[2].Pfm.a
Motor[8].pDac = Gate3[0].MacroOutA[8][0].a
Motor[9].pDac = ECAT[0].IO[24].Data.a

On re-initialization, the Power PMAC will automatically identify the Servo and MACRO interfaces that are present, and assign **Motor[*x*].pDac** elements to the addresses of available interface channels. For motors with no interface channels available **Motor[*x*].pDac** is set to **Sys.pushm**, the address of the start of the user shared memory buffer.

Motor[*x*].pEnc

Description: Outer (position) loop position feedback pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Legacy I-variable alias: I*x*03

Motor[*x*].pEnc specifies which register the selected motor uses for its outer (usually position) loop feedback value. It contains the address of this register. In this way, it determines which feedback device is used to close the outer loop. Note that despite the element name, it is not required that encoders be used for feedback.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Motor[*x*].pEnc must be set to the address of an encoder conversion table entry so that the outer loop uses the processed result of that entry. The command to do this is of the type:

Motor[*x*].pEnc = EncTable[*i*].a

Often ***x*** and ***i*** will have the same value, and this will usually be the case in the auto-configured default setup created on re-initialization, but this does not need to be the case.

For example:

Motor[1].pEnc = EncTable[1].a

Every servo cycle, the motor will read the specified register, which should contain the *change in position* during the servo cycle. It will take this value, multiply it by **Motor[*x*].PosSf**, and add it to the previous cycle's motor actual position value.

Note that if the motor's servo loop is closed in the phase interrupt, as specified by setting bit 3 of **Motor[*x*].PhaseCtrl** to 1, the register for outer-loop feedback is specified by **Motor[*x*].pPhaseEnc** instead.

Motor[*x*].pEnc2

Description: Inner (velocity) loop position feedback pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Legacy I-variable alias: I*x*04

Motor[*x*].pEnc2 specifies which register the selected motor uses for its inner (usually velocity) loop feedback value. It contains the address of this register. In this way, it determines which feedback device is used to close the outer loop. Note that despite the element name, it is not required that encoders be used for feedback.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Motor[*x*].pEnc2 must be set to the address of an encoder conversion table entry so that the inner loop uses the processed result of that entry. The command to do this is of the type:

Motor[*x*].pEnc2 = EncTable[*i*].a

Often ***x*** and ***i*** will have the same value, and this will usually be the case in the auto-configured default setup created on re-initialization, but this does not need to be the case.

For example:

Motor[1].pEnc2 = EncTable[1].a

Every servo cycle, the motor will read the specified register, which should contain the *change in position* during the servo cycle. It will take this value, multiply it by **Motor[*x*].Pos2Sf**, and add it to the previous cycle's motor actual inner-loop position value.

In most applications, this variable will contain the same value as **Motor[*x*].pEnc**, so both inner and outer loops use the same value. However, this variable provides an easy means to provide dual-loop feedback. Commonly, the inner loop would use an encoder or resolver on the motor for maximum stability, and the outer loop a sensor on the load for maximum accuracy.

Note that if the motor's servo loop is closed in the phase interrupt, as specified by setting bit 3 of **Motor[*x*].PhaseCtrl** to 1, the register for inner-loop feedback is specified by **Motor[*x*].pPhaseEnc** instead.

Motor[x].pEncCtrl

Description: Encoder capture control flag pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Motor[*x*].pEncCtrl specifies which register the selected motor uses for any capture trigger control signals. It contains the address of this register. Presently, this register is only used to prepare the trigger over the MACRO ring. Power PMAC will use this register for triggered moves such as homing search moves.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Most commonly, **Motor[*x*].pEncCtrl** is set to the address of a channel control register in a Servo IC, or the output flag register of a MACRO node. For example:

```
Motor[1].pEncCtrl = Gate1[4].Chan[0].Ctrl.a
Motor[2].pEncCtrl = Gate3[0].Chan[1].InCtrl.a
Motor[3].pEncCtrl = Gate2[0].Macro[4][3].a
Motor[4].pEncCtrl = Gate3[2].MacroOutA[8][3].a
```

On the **$$$***** re-initialization command, **Motor[*x*].pEncCtrl** is automatically assigned to the status register of an auto-detected servo channel or MACRO node according to pre-determined ordering rules.

Motor[x].pEncLoss

Description: Sensor-loss register pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0 (encoder loss detection disabled)

Motor[*x*].pEncLoss specifies which register the motor looks to for its "encoder-loss" bit. It contains the address of this register. If **Motor[*x*].pEncLoss** is set to the default value of 0, this motor will not use any automatic encoder-loss detection function.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

In the specified register, Power PMAC uses the bit whose number is contained in **Motor[*x*].EncLossBit** to read the encoder-loss status. The polarity of the encoder-loss status bit is set by **Motor[*x*].EncLossLevel**. The cumulative number of scans detecting the specified loss input state before a fault is declared is specified by **Motor[*x*].EncLossLimit**.

If an encoder loss fault is detected, this motor is automatically "killed" (disabled) by the Power PMAC. Other motors in the same coordinate system will either be killed or "aborted" (decelerated to a controlled stop), as determined by bit 1 (value 2) of **Motor[*x*].FaultMode** for this motor.

For quadrature encoders connected to ACC-24E2*x* PMAC2-style axis-interface boards, which have "exclusive-OR" quadrature loss detection circuits external to the ASIC, **Motor[*x*].pEncLoss** can be set to **Gate1[*i*].Chan[*j*].EncLossN.a**. (Alternately the board structure alias of **ACC24E2*x*[*i*].Chan[*j*].EncLossN.a** can be used.) Since this is not actually in the ASIC, the value reported back will be **Cid[*n*].PartData[*j*].a**. The encoder loss status bit is in bit 13 of this register.

For serial encoders connected to ACC-84E PMAC2-style serial-encoder-interface boards, which have several protocol-dependent loss-detection circuits, **Motor[*x*].pEncLoss** can be set to **ACC84E[*i*].Chan[*j*].SerialEncDataB.a**. Most commonly the "time-out" error bit in bit 31 of this register is used.

For quadrature encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which have "exclusive-OR" quadrature loss detection circuits in the ASIC, **Motor[*x*].pEncLoss** can be set to **Gate3[*i*].Chan[*j*].Status.a**. (Alternately the board structure alias of **ACC24E3[*i*].Chan[*j*].Status.a** can be used.) This register contains the **LossStatus** transparent quadrature-loss bit in bit 28.

For analog sinusoidal encoders or resolvers connected to ACC-24E3 PMAC3-style axis-interface boards, which have "sum-of-squares" loss detection circuits, **Motor[*x*].pEncLoss** can be set to **Gate3[*i*].Chan[*j*].Status.a**. (Alternately the board structure alias of **ACC24E3[*i*].Chan[*j*].Status.a** can be used.) This register contains the **SosError** transparent "sum-of-squares" error bit in bit 31.

For serial encoders connected to ACC-24E3 PMAC3-style axis-interface boards, which have several protocol-dependent loss-detection circuits, **Motor[*x*].pEncLoss** can be set to **Gate3[*i*].Chan[*j*].SerialEncDataB.a**. (Alternately the board structure alias of **ACC24E3[*i*].Chan[*j*].SerialEncDataB.a** can be used.) Most commonly the "time-out" error bit in bit 31 of this register is used.

The Type = 12 encoder conversion table entry (new in V2.0 firmware, released 1st quarter 2015) can monitor each servo cycle for multiple error bits in a single register. If any of these bits is set, the entry sets bit 0 of **EncTable[*m*].Status** to 1, and this bit can be used as the encoder-loss bit for the motor.

Note that it is possible to specify the address of the specific bit within a register if it has its own element name (e.g. **Motor[5].pEncLoss = Gate3[1].Chan[0].LossStatus.a**), but the value will report back as the address of the full-word element (e.g. **Gate3[1].Chan[0].Status.a**).

Motor[x].pEncStatus

Description: Motor "parent" input flag pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Legacy I-variable alias: I*x*25

Motor[*x*].pEncStatus specifies the "parent" register for input flags for the motor. It contains the address of this register. It serves two main functions.

First, in setup, when the value **Motor[*x*].EncType** is set in the Script environment, the capture-flag pointer element **Motor[*x*].pCaptFlag** is automatically set to the same address as **Motor[*x*].pEncStatus**, and if **Motor[*x*].pEncStatus** is pointing to the status register of a Servo IC, the captured-position pointer element **Motor[*x*].pCaptPos** is automatically set to the address of the hardware-capture position register for the same channel.

Second, in operation, during the status and safety update for the motor, the value in the register specified by **Motor[*x*].pEncStatus** is read first. Then, as specific input flags need to be checked, if the pointer element for each of those flags – **Motor[*x*].pLimits**, **Motor[*x*].pAmpFault**, **Motor[*x*].pCaptFlag**, **Motor[*x*].pEncLoss** – contains the same address as **Motor[*x*].pEncStatus**, the (hardware) register will not be read again. Instead, it will use the register value already read and copied into memory. This saves significant access time for the most common cases where several, if not all, of the input flags are in the same register.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Most commonly, **Motor[*x*].pEncStatus** is set to the address of a channel status register in a Servo IC, or the input flag register of a MACRO node. For example:

```
Motor[1].pEncStatus = Gate1[4].Chan[0].Status.a
Motor[2].pEncStatus = Gate3[0].Chan[1].Status.a
Motor[3].pEncStatus = Gate2[0].Macro[4][3].a
Motor[4].pEncStatus = Gate3[2].MacroInA[8][3].a
```

On the **$$$***** re-initialization command, **Motor[*x*].pEncStatus** is automatically assigned to the status register of an auto-detected servo channel or MACRO node according to pre-determined ordering rules.

Motor[x].PhaseCtrl

Description: Control flag to activate commutation tasks

Range 0 .. 15

Units: none

Default: 0

Legacy I-variable alias: I*x*01

Motor[*x*].PhaseCtrl determines what tasks are done during the phase interrupt for the motor, and how they are done. It is a collection of 4 individual control bits.

Bit 0 (value 1) controls whether Power PMAC performs the commutation for the motor using "packed" data transfers or not. If this bit set to 1, Power PMAC will execute the commutation algorithm for the motor every phase interrupt, using the servo command output as an input value, and transferring data to and from the ASIC with the information for two phases "packed" as 16-bit values in a single 32-bit register. This is the default mode for the PMAC3-style 32-bit ASIC (DSPGATE3). Packed transfers save considerable read and write time. This mode cannot be used with PMAC2-style ASICs or with MACRO transfers. This bit cannot be set to 1 if bit 2 is also set to 1.

Bit 1 (value 2) controls how slip calculations are performed in the commutation of AC induction motors or in direct microstepping control. It is only used if bit 0 or bit 2 is set to 1. If bit 1 is set to its default value of 0, the slip calculations use the measured current values. If bit 1 is set to 1, the slip calculations use the commanded current values. (Turbo PMAC always used measured current values.) It must be set to 1 for direct microstepping control.

Bit 2 (value 4) controls whether Power PMAC performs the commutation for the motor using "unpacked" data transfers or not. If this bit set to 1, Power PMAC will execute the commutation algorithm for the motor every phase interrupt, using the servo command output as an input value, and transferring data to and from the ASIC with the information for each phase in a separate register. This mode must be used with PMAC2-style ASICs or with MACRO transfers. This bit cannot be set to 1 if bit 0 is also set to 1.

If bit 3 (value 8) of **Motor[*x*].PhaseCtrl** is set to 1, this motor will close its position/velocity servo loop on the phase interrupt. This permits some Power PMAC motors, such as those driving "fast-tool servos" or galvanometers, to close their loops at a substantially higher frequency than other motors in the system. With this setting, the motor will use the register specified by **Motor[*x*].pPhaseEnc** as the source of its feedback for both the outer (position) and inner (velocity) loops, instead of the standard **Motor[*x*].pEnc** and **Motor[*x*].pEnc2**.

In this case, Power PMAC will perform a simple interpolation in the phase interrupt for the motor, calculating a spline between the points from the standard servo-interrupt interpolation. The order of the spline is set by **Motor[*x*].PhaseSplineCtrl**. The element **Sys.PhaseOverServoPeriod** must be set to the ratio of phase to servo periods for this interpolation to work properly.

So if **Motor[*x*].PhaseCtrl** is set to 8, this motor will execute its servo tasks in the phase interrupt, but not execute any phase commutation tasks for the motor. In this case, it is advisable to set **Motor[*x*].pAdc** to 0 to disable reading of current-feedback ADCs. (Reading the ADCs in this case wastes processor time, but does not otherwise hurt calculations. The ADC values read are not used.)

If **Motor[*x*].PhaseCtrl** is set to 9, this motor will execute its servo tasks in the phase interrupt, and execute phase commutation tasks for the motor. In this case, the setting of **Motor[*x*].pAdc** determines whether the current loop is closed by Power PMAC or not. The servo tasks, including loop closure, are performed before the phase tasks, eliminating a phase cycle of "transport delay" that is present in the standard mode of control.

Motor[*x*].PhaseCtrl constitutes bits 24 – 27 of the full-word element **Motor[*x*].Control[0]**.

Motor[*x*].PhaseEncLeftShift

Description: Number of bits to shift phase-position source data left

Range: 0 .. 31

Units: Bits

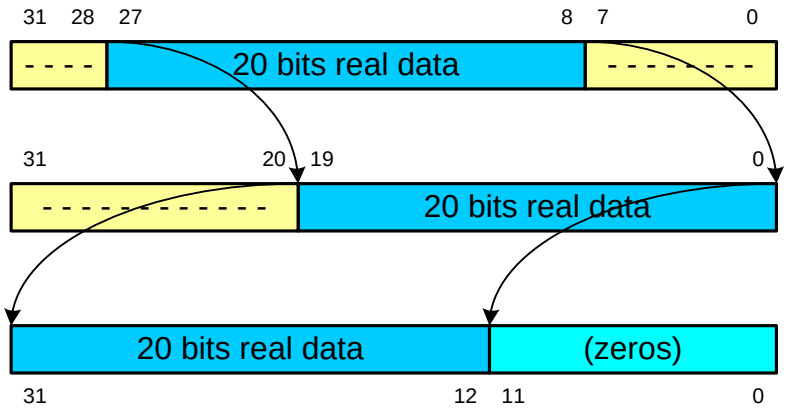
Default: 0

Motor[*x*].PhaseEncLeftShift specifies the number of bits Power PMAC will shift the position value from the 32-bit register whose address is specified by **Motor[*x*].pPhaseEnc** to the left after it has first been shifted right as specified by **Motor[*x*].PhaseEncRightShift**. The purpose of this operation is to leave the most significant bit (MSB) of actual data from the source register in the MSB of the resulting 32-bit value, permitting proper handling of the rollover of source data in the commutation or servo algorithm.

The most common sources of commutation phase position information – the encoder-counter "phase-capture" registers of the DSPGATE1 and DSPGATE3 ASICs – already have their MSB of actual data in the MSB of the register, so no shifting is required, and **Motor[*x*].PhaseEncLeftShift** can be left at its default value of 0 for these sources. Note that shifting operations do not add any computational time to the commutation algorithm.

However, there are some sources where this is not the case, and a shifting operation is required. For example, the Panasonic and Tamagawa serial encoders provide 17 bits of single-turn position information in a single register. In the DSPGATE3 IC of the ACC-24E3 and Power Brick control board, this is bits 0 – 16 of the **Gate3[*i*].Chan[*j*].SerialEncDataA** register, and **Motor[*x*].PhaseEncLeftShift** would need to be set to 15 to move the MSB to bit 31 of the result. In the ACC-84E, this is bits 8 – 24 of the **Acc84E[*i*].Chan[*j*].SerialEncDataA** register, and **Motor[*x*].PhaseEncLeftShift** would need to be set to 7 (plus any value of **Motor[*x*].PhaseEncRightShift**) to move the MSB to bit 31 of the result.

This diagram shows a 12-bit left shift after an 8-bit right shift.



Motor[x].PhaseEncRightShift

Description: Number of bits to shift phase-position source data right

Range: 0 .. 31

Units: Bits

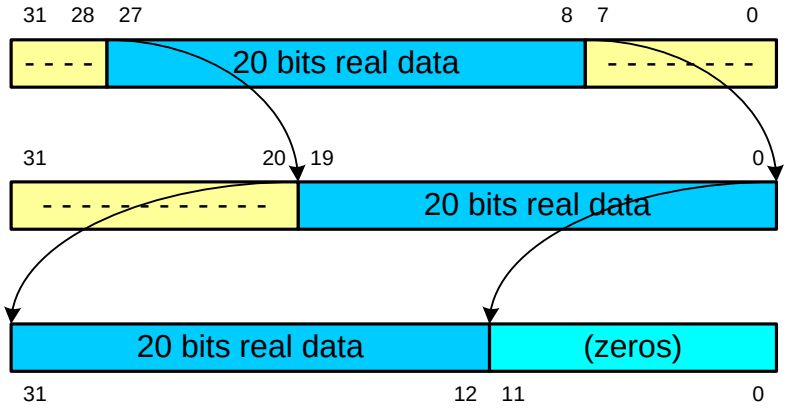
Default: 0

Motor[*x*].PhaseEncRightShift specifies the number of bits Power PMAC will shift the position value from the 32-bit register whose address is specified by **Motor[*x*].pPhaseEnc** to the right as an initial processing step. The purpose of this operation is to leave the least significant bit (LSB) of the actual data in the 32-bit register in bit 0 of the intermediate value, eliminating possible "garbage data" in the low bits of the 32-bit source register. To do this, **Motor[*x*].PhaseEncRightShift** would be set to the bit number of the LSB of actual data in the 32-bit source word.

Note that that commutation lookup "sine" table in Power PMAC has 2048 (2^{11}) entries, so that if the source data has at least 11 bits (2048 "counts") of true data per commutation cycle, the lookup table operation will automatically eliminate lower bits, and this shifting operation is not required. This is most commonly the case. However, shifting operations do not add any computational time to the commutation algorithm.

This operation is followed by a "left-shift" operation specified by **Motor[*x*].PhaseEncLeftShift**. The result must be to leave the most significant bit of actual data from the source register in the MSB of the resulting 32-bit value, permitting proper handling of the rollover of source data.

This diagram shows an 8-bit right shift followed by a 12-bit left shift.



Motor[x].PhaseFindingDac

Description: Phasing search output magnitude

Range: Non-negative floating-point

Units: LSBs of 16-bit output

Default: 0.0

Legacy I-variable alias: I*x*73

Motor[*x*].PhaseFindingDac specifies the magnitude of the output used in a power-on phasing-search move for a synchronous motor commutated by Power PMAC. A synchronous motor (such as a permanent-magnet brushless servo motor) requires a method of establishing the phase reference on power-on – either by reading an absolute position sensor, or by a phasing-search move. This parameter must be set greater than 0 for a phasing search move; it must be equal to 0 for a read of an absolute sensor.

Note that despite the name of this element, it is not required that D/A converters be used for output.

Presently, two phasing-search methods are supported: the "four-guess" method (a refinement of Turbo PMAC's "two-guess" method), and the "stepper motor" method. In the four-guess method, **Motor[*x*].PhaseFindingDac** specifies the magnitude of the torque command input to the commutation algorithm that is used for each of the four guesses of the commutation phase angle, and held for the period specified by **Motor[*x*].PhaseFindingTime**.

In the stepper-motor method, **Motor[*x*].PhaseFindingDac** specifies the magnitude of the phase current that is forced into phases, ramped up over the period specified by **Motor[*x*].PhaseFindingTime** to lock the motor like a stepper motor at a known commutation phase angle.

Motor[x].PhaseFindingTime

Description: Phasing search time

Range: 0 .. 32,767

Units: Real-time interrupt periods

Default: 0

Legacy I-variable alias: I*x*74

Motor[*x*].PhaseFindingTime specifies the duration of each step in a phasing-search move, also specifying whether a phasing-search move is done and if so, which method is used. The units are real-time interrupt (RTI) periods, which are (**Sys.RTIntPeriod** + 1) servo cycles in length.

If **Motor[*x*].PhaseFindingTime** is set to its default value of 0, or a value of 1 to 3, no phasing-search move will be performed.

If **Motor[*x*].PhaseFindingTime** is set to a value of 4 to 255, the "four-guess" phasing-search method will be used, and this parameter specifies the number of servo cycles each "guess" is maintained. In this method, Power PMAC makes four guesses as to the correct rotor phase angle (each 90 degrees apart), commands an open-loop output for each guess of the magnitude specified by **Motor[*x*].PhaseFindingDac**, holds this guess for **Motor[*x*].PhaseFindingTime**, evaluates the motor movement for each guess, and computes the rotor phase angle that best matches these responses.

If **Motor[*x*].PhaseFindingTime** is set to a value of 256 or greater, the "stepper-motor" phasing-search method will be used, and this parameter specifies the number of servo cycles each phase-current forcing is held. This is a two-stage process, and the time specified is for each stage of the process. The time should be great enough for the motor to be able to settle reliably to within the required tolerance (usually 1 or 2 degrees) for proper phasing.

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*].PhaseFindingTime** 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*].PhaseFindingTime** if **Sys.MotorsPerRtInt** > 0 is given by the equation:

$$N = ceil\left(\frac{Sys.ServoMotors + 1}{Sys.MotorsPerRtInt}\right)$$

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*].PhaseFindingTime** should be divided by *N* compared to cases where motor status updates are performed every real-time interrupt.

Motor[*x*].PhaseLoadEncLeftShift

Description: Number of bits to shift phase load position source data left

Range: 0 .. 31

Units: Bits

Default: 0

Motor[*x*].PhaseLoadEncLeftShift specifies the number of bits Power PMAC will shift the position value from the 32-bit register whose address is specified by **Motor[*x*].pPhaseLoadEnc** to the left after it has first been shifted right as specified by **Motor[*x*].PhaseLoadEncRightShift**. The purpose of this operation is to leave the most significant bit (MSB) of actual data from the source register in the MSB of the resulting 32-bit value, permitting proper handling of the rollover of source data in the servo algorithm executing under the phase interrupt.

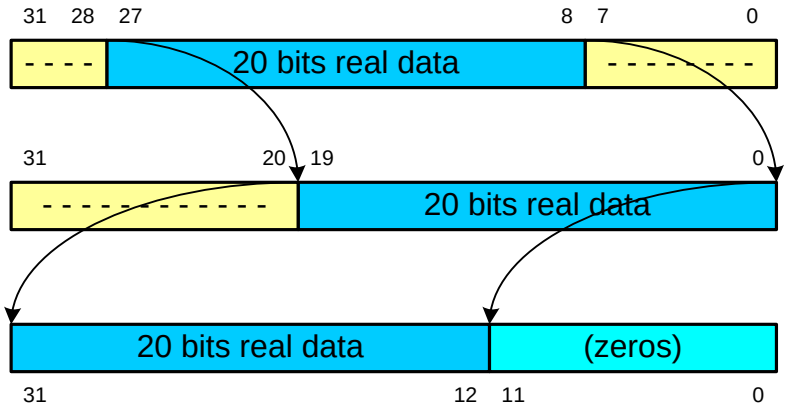
Motor[*x*].PhaseLoadEncLeftShift is only used if bit 3 (value 8) of **Motor[*x*].PhaseCtrl** is set to 1 to specify closure of the servo loop under the phase interrupt, and if **Motor[*x*].pPhaseLoadEnc** is set to a non-zero value to specify a separate load position feedback.

If **Motor[*x*].PhaseLoadEncRightShift** has been set greater than 0 to shift out low bits of undefined data, then **Motor[*x*].PhaseLoadEncLeftShift** must be set to a value at least as great as the right-shift value, to leave the MSB of real data in bit 31 of the resulting 32-bit value. For example, if the "phase capture" register of a DSPGATE1 Servo IC channel is used, which has real data in bits 8 – 31 of the 32-bit bus, the data should first be right-shifted 8 bits, then left shifted 8 bits so the MSB of the real data ends up in bit 31 of the result, and the low 8 bits of the result are all 0.

In other cases, the MSB of real data is not found in bit 31 of the source register, so the left shift value must be greater than the right shift. For example, the Panasonic and Tamagawa serial encoder interfaces in the DSPGATE3 IC provide 17 bits of real data in bits 0 – 16 of the source register. No right shift is required, but **Motor[*x*].PhaseLoadEncLeftShift** should be set to 15 to leave the MSB of real data in bit 31.

The Panasonic and Tamagawa serial encoder interfaces in the FPGA-based ACC-84E provide 17 bits of real data in bits 8 – 24 of the source register. After a right shift of 8 bits, **Motor[*x*].PhaseLoadEncRightShift** should be set to 15 to leave the MSB of real data in bit 31.

This diagram shows a 12-bit left shift after an 8-bit right shift.



Motor[*x*].PhaseLoadEncRightShift

Description: Number of bits to shift phase load position source data right

Range: 0 .. 31

Units: Bits

Default: 0

Motor[*x*].PhaseLoadEncRightShift specifies the number of bits Power PMAC will shift the position value from the 32-bit register whose address is specified by **Motor[*x*].pPhaseLoadEnc**

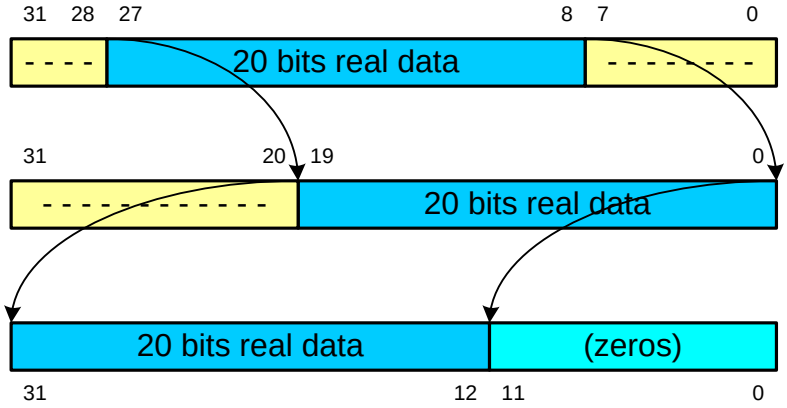
to the right as an initial processing step. The purpose of this operation is to leave the least significant bit (LSB) of the actual data in the 32-bit register in bit 0 of the intermediate value, eliminating possible "garbage data" in the low bits of the 32-bit source register. To do this, **Motor[*x*].PhaseLoadEncRightShift** would be set to the bit number of the LSB of actual data in the 32-bit source word.

For example, the "phase capture" encoder counter register in a DSPGATE1 IC is found in bits 8 – 31 of the 32-bit bus, with the low 8 bits on the bus undefined when the processor reads it. In this case **Motor[*x*].PhaseLoadEncRightShift** should be set to 8 to "shift out" these undefined low 8 bits.

This operation is followed by a "left-shift" operation specified by **Motor[*x*].PhaseLoadEncLeftShift**. The result must be to leave the most significant bit of actual data from the source register in the MSB (bit 31) of the resulting 32-bit value, permitting proper handling of the rollover of source data.

Motor[*x*].PhaseLoadEncRight is only used if bit 3 (value 8) of **Motor[*x*].PhaseCtrl** is set to 1 to specify closure of the servo loop under the phase interrupt, and if **Motor[*x*].pPhaseLoadEnc** is set to a non-zero value to specify a separate load position feedback.

This diagram shows an 8-bit right shift followed by a 12-bit left shift.



Motor[*x*].PhaseMode

Description: Commutation mode-control bits

Range: 0 .. 15

Units: Bit field

Default: 0

Motor[*x*].PhaseMode specifies how certain phase commutation tasks are performed. It is a 3-bit value, and is only used when commutation is enabled (**Motor[*x*].PhaseCtrl** bit 0 or bit 2 = 1) and digital current-loop closure is enabled (**Motor[*x*].pAdc** > 0).

Bit 0 (value 1) specifies whether "third-harmonic injection" is used or not in computing the direct-PWM voltage-command outputs. If it is set to the default value of 0, third-harmonic injection is enabled. If it is set to 1, third-harmonic injection is disabled.

Third-harmonic injection is a technique for 3-phase motors that reduces the required bus voltage for given motor phase voltages by about 15%. This permits significantly higher velocities and/or current levels before bus saturation is reached. Therefore, bit 0 should be set to 0 for direct-PWM control of 3-phase motors (**Motor[*x*].PhaseOffset** = +/-683).

However, third-harmonic injection does not reduce the required bus voltages for 2-phase motors, and it introduces significant torque ripple. Therefore, bit 0 should be set to 1 for direct-PWM control of 2-phase motors (**Motor[*x*].PhaseOffset** = +/-512) to disable this feature.

Bit 1 (value 2) of **Motor[*x*].PhaseMode** specifies whether the "direct current" loop integrator is active or not. If it is set to the default value of 0, this integrator is active. This setting should be used for multi-phase motors (e.g. brushless servo motors and AC induction motors), which need full direct current loop capability (even if only to force the direct current level to zero). If **PhaseMode** bit 3 is 1, bit 1 is not used.

If bit 1 is set to 1, the direct current loop integrator is not active. This setting should be used for direct-PWM control of DC brush motors (with **Motor[*x*].PhasePosSf** = 0.0), for which only a "quadrature current" loop is actually closed, and integrating direct current "errors" from noise and offset could interfere with proper control.

Bit 2 (value 4) specifies whether the Power PMAC phase routine computes the PWM voltage command values for 3-level drives or not. If it is set to the default value of 0, it computes the PWM voltage command values for the standard 2-level drives, with one value per phase.

If bit 2 is set to 1, the Power PMAC phase routine computes the PWM voltage command values to directly control 3-level drives (such as the Power Brick LV Ultra), with two values per phase. In this case, it uses two channels of the Servo IC – the channel specified by **Motor[*x*].pDac**, and the next higher-numbered channel on the IC (these cannot be split between ICs).

Note that for 3-level drives with a single-channel cable connection (such as the G3 Ultra drives), Power PMAC computes only a single value per phase, and the drive splits this into two values for phase. In this case bit 2 of **Motor[*x*].PhaseMode** should be set to 0.

Bit 3 (value 8) specifies whether the Power PMAC closes the current loop in the forward path for the motor or not when it is doing "direct-PWM" control. If it is set to the default value of 0, it does close the current loop here, comparing the desired current values to the measured current values, multiplying the difference by **Motor[*x*].IpfGain**, to compute the PWM voltage commands based on these errors.

If bit 3 is set to 1, the measured current values are still calculated, but they are not subtracted from the desired values before the desired values are multiplied by **Motor[*x*].IpfGain**, so the PWM voltage commands are computed based on the "desired current" values only – **Motor[*x*].IqCmd** from the position/velocity servo loop, and **Motor[*x*].IdCmd**. This mode of operation is known as "voltage-mode" PWM control instead of "current-mode" PWM control.

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.

In this case of "voltage-mode direct-PWM" control, **Motor[*x*].IpfGain** is simply used as a scale factor multiplying **IqCmd** and **IdCmd**. Typically, **IpfGain** is set to 1.0 so the loop just passes through the input commands. At this setting, the maximum command out of the servo loop provides the maximum available voltage.

Gain term **Motor[*x*].IiGain** is not used in this mode. It is possible to use the actual current values multiplied by **Motor[*x*].IpbGain**, but this is not common, so usually **IpbGain** will be set to 0.0. Actual current values can be measured, and used for "I^2T" integrated current protection. However, it is not required that current be measured to use this mode.

Voltage-mode direct-PWM control can be used in ultra-high-precision applications, when eliminating the noise from current sensing is more important than the increased bandwidth from current-loop closure. It can also be used in low-cost applications without any current-sensing capabilities.

Bit 3 of **Motor[*x*].PhaseMode** is new in V2.1 firmware, released 1st quarter 2016.

Motor[*x*].PhaseMode constitutes bits 4 – 7 of the full-word element **Motor[*x*].Control[0]**.

Motor[*x*].PhaseOffset

Description: Offset between Phases A & B

Range: -1024 .. 1023

Units: 1/2048 commutation cycle

Default: -683

Legacy I-variable alias: I*x*72

Motor[*x*].PhaseOffset specifies the angular distance between the phases of a multi-phase motor that is commutated by Power PMAC. The units of **Motor[*x*].PhaseOffset** are 1/2048 of a commutation cycle. The usual values to be used are:

3-phase: +/-683 (+/- 120°e)

2- or 4-phase: +/-512 (+/- 90°e)

When performing digital current-loop closure for a (single-phase) brush DC motor, **Motor[*x*].PhaseOffset** should be set to +/-512, as for a 2- or 4-phase motor.

For a given number of phases, the proper choice of the two possible values is determined by the polarity match between the output commands and the feedback, as detailed below.

Motor[*x*].PhaseOffset is used slightly differently depending on whether Power PMAC is performing current-loop calculations as well as commutation. Both cases are explained below:

1. Power PMAC performing commutation, but not current loop: When Power PMAC *is not* performing digital current loop closure for this motor the output direction sense determined by this parameter and the motor and amplifier phase wiring must match the feedback direction sense as determined by the encoder-decode variable (typically **Gate*n*[*i*].Chan[*j*].EncCtrl**) and the encoder wiring. If the direction senses do not match proper commutation and servo control will be impossible; the motor will lock into a given position.

For these systems, changing between the two values for a given number of phases has the same effect as exchanging motor leads, which changes the motor's direction of rotation for a given sign of a servo torque command. Once this commutation/feedback polarity has been properly matched, the servo/feedback polarity will automatically be properly matched (assuming that the servo loop is using the same feedback sensor).

2. Power PMAC performing commutation and current loop: When Power PMAC *is* performing digital current loop closure for this motor, the output direction sense determined by this parameter must match the polarity of the phase current sensors and the analog-to-digital conversion (ADC) circuitry that brings this data into Power PMAC. It is independent of motor or amplifier phase wiring, encoder wiring, and Turbo PMAC encoder-decode direction sense.

For these systems with a Power PMAC digital current loop, if the phase-current ADC registers report a positive value for current flowing *into* the phase (i.e. the PWM voltage command value and the current feedback value have the same sign), **Motor[*x*].PhaseOffset** must be set to a negative value (usually -683 for a 3-phase motor, or -512 for a 2- or 4-phase motor). If the phase-current ADC registers report a positive value for current flowing *out of* the phase (i.e. the PWM voltage command value and the current feedback value have opposite signs), **Motor[*x*].PhaseOffset** must be set to a positive value (usually 683 for a 3-phase motor, or 512 for a 2- or 4-phase motor).

Caution

Do not attempt to close the digital current loops on a Power PMAC motor (enabling the motor with either an open or closed position loop) until you are sure of the proper sense of the **Motor[*x*].PhaseOffset** setting. A **Motor[*x*].PhaseOffset** setting of the wrong sense will cause positive feedback in the current loop, leading to saturation of the PMAC outputs and possible damage to the motor and/or amplifier.

For systems with Power PMAC digital current loop closure, the commutation/feedback polarity match is independent of the servo/feedback polarity. Once **Motor[*x*].PhaseOffset** has been set for proper commutation/feedback polarity, the proper position-loop servo/feedback polarity must still be established.

Note that bit 0 (value 1) of **Motor[*x*].PhaseMode** should be set to 0 to enable "third-harmonic injection" if **Motor[*x*].PhaseOffset** is set to +/-683 for a 3-phase motor. This bit must be set to 1 if **Motor[*x*].PhaseOffset** is set to +/-512 for a 1-, 2-, or 4-phase motor, because third-harmonic injection is not appropriate for these motors.

Motor[*x*].PhasePosSf

Description: Commutation angle scale factor

Range: Positive floating-point (double-precision)

Units: 1/2048 commutation cycle per 32-bit source-register LSB

Default: 0.008

Legacy I-variable alias: I*x*71

Motor[*x*].PhasePosSf defines the size of a commutation cycle by multiplying the data in the 32-bit source register for commutation position feedback as specified by **Motor[*x*].pPhaseEnc** into the units of a commutation cycle. This multiplication is done after any shifting of the source data is performed as specified by **Motor[*x*].PhaseEncRightShift** and **Motor[*x*].PhaseEncLeftShift** (not common). Power PMAC divides a commutation cycle into 2048 parts, so the units of the result (in **Motor[*x*].PhasePos**) are 1/2048 of a commutation cycle.

One commutation cycle is equivalent to a single north-south magnetic pole-pair of a motor, so a common 4-pole rotary brushless or induction motor will have two commutation cycles per revolution, and one commutation unit is equivalent to 1/4096 of a mechanical revolution.

Power PMAC's commutation algorithm always reads the commutation position register specified by **Motor[*x*].pPhaseEnc** as a full 32-bit register, even if there is not real data in all 32 bits. For example, the PMAC2-style "DSPGATE1" Servo ICs have 24-bit phase position registers that appear in the high 24 bits of the 32-bit Power PMAC data bus. The low 8 bits have no real data. This means that a single encoder count is equivalent to 256 LSBs of the 32-bit numerical value that is read by the processor.

Note that **Motor[*x*].PhasePosSf** can virtually always be expressed as the ratio of two integers, even though it will not be an integer itself. Users may find it easier to enter it as the ratio of two integers, especially if the resulting value does not have an exact representation as a floating-point value, or an extremely long representation.

When performing direct-PWM control of a brush DC motor, Power PMAC must execute the commutation algorithm in order to close the current loop for the motor. In this case, **Motor[*x*].PhasePosSf** should be set to 0.0 in order to effectively disable the AC nature of the commutation for the DC motor.

Examples

A 4-pole rotary brushless motor has a 500-line encoder that is fed into a channel of a PMAC2-style "DSPGATE1" Servo IC, which performs "times-4" decoding to produce 2000 counts per mechanical revolution. A 4-pole motor has 2 commutation cycles per mechanical revolution, so there are 1000 counts per commutation cycle. The count data appears in the high 24 bits of the

32-bit data bus, so each count is equivalent to 256 LSBs, yielding 256,000 LSBs per commutation cycle. The scale factor can be calculated as:

$$Motor[x].PhasePosSf = \frac{2048\,units/comm-cyc}{256{,}000\,LSBs/comm-cyc} = 0.008\left(\frac{units}{LSB}\right)$$

A linear brushless motor has a 60.96-mm (2.4") pole-pair spacing. It has a sinusoidal encoder with a 10-micron (0.01-mm) pitch that is fed into a PMAC3-style "DSPGATE3" IC, which performs interpolation resulting in 16,384 LSBs per line of the encoder. The interpolated value is provided in a 32-bit register with the interpolated LSB of position in the LSB of the register. There are 60.96/0.01=6,096 encoder lines per commutation cycle, and so 6,096*16,384=99,876,864 LSBs per commutation cycle. The scale factor can be calculated as:

$$Motor[x].PhasePosSf = \frac{2048\,units/comm-cyc}{99{,}876{,}864\,LSBs/comm-cyc} = 0.000020505249\left(\frac{units}{LSB}\right)$$

This value is best entered as 2,048 / 99,876,864 to let Power PMAC calculate the exact value.

A 4-pole rotary brushless motor has a serial encoder for which the 17-bit single-turn data appears in bits 8 to 24 of the "A" data register of an ACC-84E channel, and the multi-turn data appears in the "B" data register. A net left shift of 7 bits using **Motor[*x*].PhaseEncShiftRight** and **Motor[*x*].PhaseEncShiftLeft** is required to leave the most significant bit of the single-turn data in bit 31 so Power PMAC can handle rollover of the data properly. This means that there are 2^{32} LSBs of the register per mechanical revolution, and 2^{31} LSBs per commutation cycle. The scale factor can be computed as:

$$Motor[x].PhasePosSf = \frac{2^{11}\,units/comm-cyc}{2^{31}\,LSBs/comm-cyc} = \frac{1}{2^{20}} = \frac{1}{1{,}048{,}576}\left(\frac{units}{LSB}\right)$$

This value is best entered as 1 / 1,048,576 to let Power PMAC calculate the exact value.

Motor[*x*].PhaseSplineCtrl

Description: Phase interrupt spline interpolation order

Range: 0 .. 3

Units: Polynomial order

Default: 0

Motor[*x*].PhaseSplineCtrl specifies the order of the spline interpolation performed in the phase interrupt if the motor has been instructed to perform servo calculations in the phase interrupt by setting bit 3 (value 8) of **Motor[*x*].PhaseCtrl** to 1. **Motor[*x*].PhaseSplineCtrl** is not used if servo calculations are not performed in the phase interrupt.

When servo calculations are performed in the phase interrupt, only a simple additional stage of interpolation is performed in the phase interrupt – the standard interpolation calculations are still

performed in the lower-frequency servo interrupt. The Power PMAC phase-interrupt routines perform a simple spline interpolation between the points calculated in the servo interrupts. The following spline orders can be specified:

- **Motor[*x*].PhaseSplineCtrl** = 0: 0th order – No interpolation between points
- **Motor[*x*].PhaseSplineCtrl** = 1: 1st order – Constant-vel interpolation between points
- **Motor[*x*].PhaseSplineCtrl** = 2: 2nd order – Constant-accel interpolation between points
- **Motor[*x*].PhaseSplineCtrl** = 3: 3rd order – Constant-jerk interpolation between points

Higher-order splines are smoother, but add more of a "tail" to the trajectory, extending its total time. The use of **Motor[*x*].PhaseSplineCtrl** permits the user to find the optimal tradeoff between smoothness and quickness.

Note that if **Motor[*x*].PreFilterEna** is set greater than 0 when servo calculations are done in the phase interrupt, 3rd-order spline interpolation is used in the phase interrupt regardless of the setting of **Motor[*x*].PhaseSplineCtrl**.

Global saved setup element **Sys.PhaseOverServoPeriod** must be set properly to reflect the ratio of the phase and servo periods for this phase spline interpolation to work properly.

Motor[*x*].pLimits

Description: Overtravel limit (input) flag pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware
(= 0 – disabled – if no hardware available to assign)

Motor[*x*].pLimits specifies which register the selected motor uses for its hardware overtravel-limit input flag signals. It contains the address of this register.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Most commonly, **Motor[*x*].pLimits** is set to the address of a channel status register in a Servo IC. For example:

Motor[1].pLimits = Gate1[4].Chan[0].Status.a

Motor[2].pLimits = Gate3[0].Chan[1].Status.a

If the motor is controlled through the MACRO ring, **Motor[*x*].pLimits** is typically set to the address of a MACRO-node input flag register (input register 3 of the node). For example:

Motor[3].pLimits = Gate2[0].Macro[4][3].a

Motor[4].pLimits = Gate3[0].MacroInA[5][3].a

If the motor is controlled through an EtherCAT network, **Motor[*x*].pLimits** is typically set to the address of an EtherCAT holding register that is mapped to the drive status register or I/O device input register containing the limit input bits. For example:

Motor[5].pLimits = **ECAT[0].IO[30].Data.a**

In the specified register, Power PMAC usually uses the bit whose number is contained in **Motor[*x*].LimitBits** to read the positive-limit signal; the negative-limit signal is read in the next higher-numbered bit. However, if **LimitBits** is set to a value greater than 31, other combinations are possible (new in V2.1 firmware, released 1st quarter 2016).

If **Motor[*x*].LimitBits** is in the range of 96 – 127 or 224 – 255, **Motor[*x*].pLimits** specifies the register for the positive-end limit input only, and **Motor[*x*].pAuxFault** is used to specify the register for the negative-end limit. This functionality is new in V2.1 firmware, released 1st quarter 2016.

Note that it is possible to specify the address of the specific bit within a register if it has its own element name (e.g. **Motor[5].pLimits = Gate1[6].Chan[0].PlusLimit.a**), but the value will report back as the address of the full-word element (e.g. **Gate1[6].Chan[0].Status.a**).

On a **$$$***** re-initialization command, Power PMAC automatically assigns values of **Motor[*x*].pLimits** based on the Servo ICs and MACRO ICs found.

If **Motor[*x*].pLimits** is set to 0, this motor will not use any automatic hardware overtravel-limit function (but software overtravel limits may still be active). On re-initialization, if no hardware is present for auto-assignment to the motor, this element will automatically be set to 0, disabling the hardware overtravel limit input function for the motor.

Motor[*x*].pMasterEnc

Description: Master position source pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: **EncTable[0].a**

Legacy I-variable alias: I*x*05

Motor[*x*].pMasterEnc specifies which register the selected motor uses for its master position value in its position-following (electronic-gearing) function. It contains the address of this register. In this way, it determines which sensor (or pseudo-sensor) is used as a master. Note that despite the element name, it is not required that encoders be used for master position.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

Motor[*x*].pMasterEnc must be set to the address of an encoder conversion table entry so that the following algorithm uses the processed result of that entry. The command to do this is of the type:

Motor[*x*].pMasterEnc = EncTable[*i*].a

For example:

Motor[1].pMasterEnc = EncTable[0].a

Every servo cycle, if following is active for the motor (**Motor[*x*].MasterCtrl** bit 0 = 1), the motor will read the specified register, which should contain the *change in position* during the servo cycle. It will take this value, multiply it by **Motor[*x*].MasterPosSf**, and add it to the previous cycle's net desired position value, so the motor will track the change in this master position value.

By default, each motor's **pMasterEnc** is set to the address of **EncTable[0]**, which by default is a zero value.

Motor[*x*].pMotorNode

Description: Cyclic network command source pointer

Range: Legitimate addresses

Units: Data structure element addresses, 0

Default: **Sys.pushm** (non-functional)

Motor[*x*].pMotorNode specifies which register the motor gets its (first) cyclic network command value from if motor is set up to take these commands. It contains the address of this register. It is only used if **Motor[*x*].MotorMode** is set to a value greater than 0 to tell the motor to accept cyclic network commands. The meaning of the command is dependent on the exact value of **Motor[*x*].MotorMode**.

The value of this element is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address. Most commonly, **Motor[*x*].pMotorNode** is set to the address of the first MACRO input register for a servo node in a PMAC2-style or PMAC3-style MACRO IC.

For a PMAC2-style "DSPGATE2" MACRO IC, such as in a UMAC ACC-5E, the setting would be of the form:

Motor[*x*].pMotorNode = **Gate2[*i*].Macro[*j*][0].a**, where ***i*** is the IC index (0 – 15) and ***j*** is the node number (where 0, 1, 4, 5, 8, 9, 12, and 13 are standardly used as "servo nodes" for passing this type of data). Instead of **Gate2**, the "alias" name of the actual device (e.g. **Acc5E**) can be used instead. When queried, the appropriate alias name will be used in reporting the value of this element (e.g. **Acc5E[0].Macro[8][0].a**).

For a PMAC3-style "DSPGATE3" MACRO IC, such as in a Power Brick, or UMAC ACC-5E3, the setting would be of the form:

Motor[*x*].pMotorNode = **Gate3[*i*].MacroIn*α*[*j*][0].a**, where ***i*** is the IC index (0 – 15) and ***j*** is the node number (where 0, 1, 4, 5, 8, 9, 12, and 13 are standardly used as "servo nodes" for passing this type of data), and ***α*** is "A" or "B". Instead of **Gate3**, the "alias" name of the actual device (e.g. **PowerBrick**) can be used instead. When queried, the appropriate alias name will be used in reporting the value of this element (e.g. **PowerBrick[1].MacroInA[1][0].a**).

Motor[*x*].Pn*i*

Description: Pre-filter polynomial i^{th}-order numerator term (i = 0 to 4)

Range: Floating-point

Units: none (unit-less z-transform coefficient)

Default: 0.0

The **Motor[*x*].Pn*i*** terms are double-precision coefficients of the 4th-order polynomial numerator term acting on the commanded trajectory position before it is sent the servo algorithm. The transfer function of this filter is:

$$P(z) = 1 - \left(1 - z^{-1}\right)\frac{Pn_0 + Pn_1 z^{-1} + Pn_2 z^{-2} + Pn_3 z^{-3} + Pn_4 z^{-4}}{1 + Pd_1 z^{-1} + Pd_2 z^{-2} + Pd_3 z^{-3} + Pd_4 z^{-4}}$$

To use any of these terms, **Motor[*x*].PreFilterEna** must be set to a value greater than 0 to enable the pre-filter for this motor. The value of **Motor[*x*].PreFilterEna** specifies the update period of the filter in servo interrupt periods.

In standard filter designs, the **Pn4** term is always set to 0, so the numerator does not have a higher effective order than the denominator.

The sum of all **Pn*i*** terms minus the sum of all **Pd*i*** terms must equal exactly 1.0 in order for the filter not to have any net scaling effect on the trajectory "signal" (i.e. to have a "DC gain" of 1.0). Typically, **Pn1** and higher, and **Pd1** and higher are set to obtain the desired dynamics, then **Pn0** is set to 1.0 minus the sum of the higher-order **Pn*i*** terms plus the sum of the **Pd*i*** terms so there is no net scaling due to the filter.

Motor[x].PosReportMode

Description: Position reporting mode control

Range: 0 .. 1

Units: Boolean

Default: 0

Motor[*x*].PosReportMode specifies the position reporting mode for the motor. If set to the default value of 0, the components of net desired motor position that are in "offset mode", moving the reference position for computed trajectories for the motor and associated axes, are subtracted out when the position is reported in response to a query command of the motor. In this mode, the motor position is reported relative to the programming reference position.

If **Motor[*x*].PosReportMode** is set to 1, the offset-mode components of motor position are not subtracted out when the position is reported in response to a query command of the motor. In this mode, the motor position is reported relative to the base motor reference ("home") position.

The master-position component of the position-following function in **Motor[*x*].ActiveMasterPos** is an offset-mode component if bit 1 of **Motor[*x*].MasterCtrl** is set to 1, putting the following function in offset mode. In this case, **Motor[*x*].PosReportMode** determines whether it is subtracted out of the reported position value or not.

Motor[*x*].ActiveMasterPos is not an offset-mode component if bit 1 of **Motor[*x*].MasterCtrl** is set to 0, putting the following function in normal mode. In this case, it is not subtracted out of the reported position, regardless of the value of **Motor[*x*].PosReportMode**.

The motor position component in **Motor[*x*].CompDesPos**, which usually comes from a compensation table or a cam table, is always an offset-mode component, so **Motor[*x*].PosReportMode** determines whether it is subtracted out of the reported position value or not.

Motor[*x*].PosReportMode controls how the offset-mode position components are used in the reporting of motor desired positions (**d**, **dread**), actual positions (**p**, **pread**), and target positions (**t**, **tread**).

Motor[x].PosSf

Description: Outer (position) loop scale factor

Range: Positive floating-point (double-precision)

Units: Motor units per source unit

Default: 1.0

Legacy I-variable alias: I*x*08

Motor[*x*].PosSf specifies the scale factor by which the actual position value read at the register specified by **Motor[*x*].pEnc** is multiplied before being used in actual-position calculations in the outer (usually position) loop of the motor. This scale factor effectively defines what a motor unit is for software purposes. This motor unit does not have to match the units of the source data, although with the default scale factor of 1.0, it will.

The source of position data is almost always the result register of an encoder conversion table (ECT) entry. Each entry has its own output scale factor, so there is a lot of flexibility as to what units are used in each stage of the process. However, the most common practice is for the floating-point output of the ECT entry to be scaled in "counts" of a counter, or LSBs of a data word or A/D converter, even if there is fractional resolution from some interpolation technique.

Usually, **Motor[*x*].PosSf** will be left at the default value of 1.0, so the motor units are in "counts" or "LSBs" of the feedback device, or it will be set so that the motor units become the engineering units of millimeters, inches, degrees, or revolutions. In the first case, the units of the matching axis are made to be engineering units by a non-unity scale factor (e.g. **#1->1000X**); in the second case, there can be a unity scale factor in the axis definition (e.g. **#1->X**).

All saved setup elements for the motor that deal with position or its derivatives (velocity, acceleration, jerk) use the motor units that are defined by **Motor[*x*].PosSf**, so a change in this element will require counteracting changes in all of those elements.

Caution

This scale factor is effectively a gain term in the feedback algorithm, so changing this scale factor requires counteracting servo gain changes, particularly in the **Motor[*x*].Servo.Kp** proportional gain term.

Motor[*x*].PosUnit

Description: Motor selected outer (position) loop units

Range: 0 .. 15

Units: Enumeration

Default: 0

Motor[*x*].PosUnit expresses the position unit selected by the user for the motor. Power PMAC makes no automatic use of this element; the element is intended for use with the IDE automatic motor setup routines to store and later retrieve user unit information for the motor. It is most commonly set from the IDE setup routines.

Motor[*x*].PosUnit is an enumeration, with each of the 16 possible values representing a different position unit. The following table gives each of the values and its meaning as interpreted by the IDE motor setup routines:

Motor[*x*].PosUnit	Selected Unit	Motor[*x*].PosUnit	Selected Unit
0	*None selected*	8	Mil (in/1000)
1	Feedback unit (ct)	9	Revolution
2	Meter (m)	10	Radian (rad)
3	Millimeter (mm)	11	Degree (deg)
4	Micrometer (μm)	12	Gradian (grad)
5	Nanometer (nm)	13	Arcminute (')
6	Picometer (pm)	14	Arcsecond (")
7	Inch (in)	15	*Reserved*

Note that the actual scaling used by Power PMAC for the motor's outer (position) loop is determined by **Motor[*x*].PosSf**, which multiplies the output from the encoder conversion table (ECT) entry used for outer-loop feedback, and by **EncTable[*n*].ScaleFactor** for the ECT entry, which multiplies the raw position feedback value. The scaling of the units of an axis related to the motor relative to motor units is determined by the coefficients of the axis definition statement of the kinematic subroutines.

Motor[*x*].PosUnit is new in V2.1 firmware, released 1st quarter 2016.

Motor[*x*].Pos2Sf

Description: Inner (velocity) loop scale factor

Range: Positive floating-point (double-precision)

Units: Motor units per LSB of source data

Default: 1.0

Legacy I-variable alias: I*x*09

Motor[*x*].Pos2Sf specifies the scale factor by which the actual position value read at the register specified by **Motor[*x*].pEnc2** is multiplied before being used in actual-position calculations in the inner (usually velocity) loop of the motor. This scale factor effectively defines what a motor unit is for software purposes. This motor unit does not have to match the units of the source data, although with the default scale factor of 1.0, it will.

The source of position data is almost always the result register of an encoder conversion table (ECT) entry. Each entry has its own output scale factor, so there is a lot of flexibility as to what units are used in each stage of the process. However, the most common practice is for the floating-point output of the ECT entry to be scaled in "counts" of a counter, or LSBs of a data word or A/D converter, even if there is fractional resolution from some interpolation technique.

Usually, **Motor[*x*].Pos2Sf** will be left at the default value of 1.0, so the inner-loop units are in "counts" or "LSBs" of the feedback device, or it will be set so that the motor units become the engineering units of millimeters, inches, degrees, or revolutions. Almost always, the choice of units will match those of the outer loop as set by **Motor[*x*].PosSf**.

This scale factor is effectively a gain term in the feedback algorithm, so changing this scale factor requires counteracting servo gain changes, particularly in the **Motor[*x*].Servo.Kvfb** and **Kvifb** derivative gain terms.

Motor[*x*].Pos2Unit

Description: Motor selected inner (velocity) loop units

Range: 0 .. 15

Units: Enumeration

Default: 0

Motor[*x*].Pos2Unit expresses the position unit selected by the user for the inner (velocity) loop of the motor. Power PMAC makes no automatic use of this element; the element is intended for use with the IDE automatic motor setup routines to store and later retrieve user unit information for the motor. It is most commonly set from the IDE setup routines.

Motor[*x*].Pos2Unit is an enumeration, with each of the 16 possible values representing a different position unit. The following table gives each of the values and its meaning as interpreted by the IDE motor setup routines:

Motor[*x*].Pos2Unit	Selected Unit	Motor[*x*].Pos2Unit	Selected Unit
0	*None selected*	8	Mil (in/1000)
1	Feedback unit (ct)	9	Revolution
2	Meter (m)	10	Radian (rad)
3	Millimeter (mm)	11	Degree (deg)
4	Micrometer (μm)	12	Gradian (grad)
5	Nanometer (nm)	13	Arcminute (')
6	Picometer (pm)	14	Arcsecond (")
7	Inch (in)	15	*Reserved*

Note that the actual scaling used by Power PMAC for the motor's inner (velocity) loop is determined by **Motor[*x*].Pos2Sf**, which multiplies the output from the encoder conversion table (ECT) entry used for inner-loop feedback, and by **EncTable[*n*].ScaleFactor** for the ECT entry, which multiplies the raw position feedback value.

Motor[*x*].Pos2Unit is new in V2.1 firmware, released 1st quarter 2016.

Motor[x].PowerOnMode

Description: Power-on/reset action control

Range: 0 .. 7

Units: Bit field

Default: 0

Legacy I-variable alias: I*x*80

Motor[*x*].PowerOnMode determines the power-on behavior of the motor. It consists of 3 independent control bits.

- Bit 0 (value 1) determines the motor state after a phase-referencing command – either the on-line **$** command or the setting of **Motor[*x*].PhaseFindingStep** to 1. If it is set to 0, the motor is killed, even it if had to be enabled for a phasing-search move. If it is set to 1, the motor is enabled and the position loop is closed. This applies to phase referencing done both by absolute position reads and phasing search moves. For motors not commutated by Power PMAC, if the bit is 0, a phase-referencing command will not enable the motor and close its loop; if the bit is 1, a phase-referencing command will enable the motor and close the loop.

- Bit 1 (value 2) determines whether a phase-referencing command is automatically issued for the motor at power-on/reset. If it is set to 0, the command is not automatically issued by Power PMAC, and the user's application must issue the command. If it is set to 1, the command is automatically issued at power-on/reset, so no explicit action in the user application software is required to execute this function. For motors not commutated by PMAC, if both this bit and bit 0 are set to 1, the motor will automatically be enabled on power-on/reset.

WARNING

An unsuccessful phase referencing can lead to a dangerous runaway condition when the loop is closed. If the phase referencing (read or search) is specified to execute immediately on power-on/reset, it is essential that whatever sensors and/or amplifiers that are used be in proper operation at that time.

- Note that an unsuccessful phase referencing can lead to a dangerous runaway condition when the loop is closed. If the phase referencing (read or search) is specified to execute immediately on power-on/reset, it is essential that whatever sensors and/or amplifiers that are used be in proper operation at that time.

- Bit 2 (value 4) determines whether an absolute servo position read command is automatically issued for the motor at power-on/reset. If it is set to 0, the command is not automatically issued by Power PMAC, and the user's application must issue the command (on-line **hmz** or program **homez**). If it is set to 1, the command is automatically issued at power-on/reset, so no explicit action in the user application

software is required to execute this function. If **Motor[*x*].pAbsPos** is set to the default value of 0, disabling the absolute position read function, the setting of this bit does not matter.

Motor[*x*].pPhaseEnc

Description: Commutation ongoing position pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Auto-configured based on hardware

Legacy I-variable alias: I*x*83

Motor[*x*].pPhaseEnc specifies the 32-bit register the motor uses for its commutation position feedback value. It contains the address of this register. In this way, it determines which feedback device is used to provide rotor-angle feedback. Note that despite the element name, it is not required that encoders be used for this feedback.

If the motor's servo loop is closed in the phase interrupt, as specified by setting bit 3 (value 8) of **Motor[*x*].PhaseCtrl** to 1, then this element also specifies the register that is used for the outer (position) and inner (velocity) servo loops. However, if **Motor[*x*].pPhaseLoadEnc** is set to a non-zero value, it specifies the register that is used for the outer (position) loop feedback, and **Motor[*x*].pPhaseEnc** specifies only the register used for the inner (velocity) loop feedback.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

The data in this register must be in fixed-point (integer) format, and to be useful, it must be updated every phase cycle. Data from the encoder conversion table is not suitable on these accounts.

Note that if the source data from the register specified here can "roll over", the resulting data from the use of this register must be found in the most-significant bit (MSB) of the 32-bit bus. If this is not true of the data in the source register, **Motor[*x*].PhaseEncRightShift** and **Motor[*x*].PhaseEncLeftShift** can be used.

Most commonly, **Motor[*x*].pPhaseEnc** is set to the address of the incremental encoder "phase capture" register for a channel in a Servo IC, or of the serial encoder data register (latched on the phase clock) of a Servo IC or FPGA, or of the position-feedback register (register[0]) for a node in a MACRO IC. For example:

```
Motor[1].pPhaseEnc = Gate1[4].Chan[0].PhaseCapt.a
Motor[2].pPhaseEnc = Gate3[1].Chan[1].PhaseCapt.a
Motor[3].pPhaseEnc = Gate3[2].Chan[2].SerialEncDataA.a
Motor[4].pPhaseEnc = Acc84E[0].Chan[3].SerialEncDataA.a
Motor[5].pPhaseEnc = Gate2[0].Macro[4][0].a
```

Motor[6].pPhaseEnc = Gate3[1].MacroInA[5][0].a

Motor[*x*].pPhaseLoadEnc

Description: Servo-in-phase separate load position pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0 (disabled)

Motor[*x*].pPhaseLoadEnc specifies the 32-bit register the motor will use as a separate outer-loop servo feedback position when closing the servo loop in the phase interrupt – when bit 3 (value 8) of **Motor[*x*].PhaseCtrl** is set to 1. It contains the address of this register. If it is set to 0, the register specified by **Motor[*x*].pPhaseEnc** is used for both the inner-loop and outer-loop servo feedback positions, as was always the case in firmware versions older than 1.6, when **Motor[*x*].pPhaseLoadEnc** was introduced. This provides backward compatibility.

Motor[*x*].pPhaseLoadEnc provides the capability for dual feedback, even when the servo loop is closed under the phase interrupt, as for fast-tool servo axes. In this case, **Motor[*x*].pPhaseLoadEnc** is usually used for feedback on the load, closing the position loop with that for accuracy, and **Motor[*x*].pPhaseEnc** is usually used for feedback on the motor, closing the velocity loop with that for stability.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

The data in this register must be in fixed-point (integer) format, and to be useful, it must be updated every phase cycle. Data from the encoder conversion table is not suitable on these accounts.

Note that if the source data from the register specified here can "roll over", the resulting data from the use of this register must be found in the most-significant bit (MSB) of the 32-bit bus. If this is not true of the data in the source register, **Motor[*x*].PhaseLoadEncRightShift** and **Motor[*x*].PhaseLoadEncLeftShift** can be used.

Most commonly, **Motor[*x*].pPhaseLoadEnc** is set to the address of the incremental encoder "phase capture" register for a channel in a Servo IC, or of the serial encoder data register (latched on the phase clock) of a Servo IC or FPGA, or of the position-feedback register (register[0]) for a node in a MACRO IC. For example:

Motor[1].pPhaseLoadEnc = Gate1[4].Chan[1].PhaseCapt.a
Motor[2].pPhaseLoadEnc = Gate3[1].Chan[3].PhaseCapt.a
Motor[3].pPhaseLoadEnc = Gate3[2].Chan[0].SerialEncDataA.a
Motor[4].pPhaseLoadEnc = Acc84E[0].Chan[1].SerialEncDataA.a
Motor[5].pPhaseLoadEnc = Gate2[0].Macro[8][0].a
Motor[6].pPhaseLoadEnc = Gate3[0].MacroInA[1][0].a

Motor[x].PreFilterEna

Description: Trajectory pre-filter enable/update-period control

Range: 0 .. 255

Units: none

Default: 0

Motor[*x*].PreFilterEna controls whether the trajectory pre-filter is enabled or not, and if enabled, what the update period of the filter is. If it is set to the default value of 0, the pre-filter is disabled. If it is set to a value greater than 0, the pre-filter is enabled, and the value specifies the update period of the filter in servo interrupt periods. Often, periods of many servo cycles are desired when the filter is used to compensate for low-frequency dynamics. When the filter period is greater than one servo cycle, the filter output is then re-interpolated to the servo update period using a cubic spline algorithm.

The trajectory pre-filter employs polynomial terms **Motor[*x*].Pn0** to **Pn4** and **Motor[*x*].Pd1** to **Pd4** to adjust the commanded position values from mathematically computed trajectories before they are sent to the servo loop. This pre-filter can be used to do things like remove frequency content at the mechanical resonant frequencies of the system.

When the servo loop is opened on the motor (in either an enabled or disabled state), Power PMAC automatically sets **Motor[*x*].PreFilterEna** to the negative of the value it was set to with the loop closed. When the loop is closed again, this ensures that the filter is properly initialized. On enabling, Power PMAC automatically sets **Motor[*x*].PreFilterEna** back to the positive value.

Motor[x].ProgJogPos

Description: Variable jog position/distance

Range: Floating-point

Units: Motor units

Default: 0.0

Motor[*x*].ProgJogPos specifies the end position or move distance for on-line "variable jog" commands for the motor. If a **`j=*`** command is given, the value of **Motor[*x*].ProgJogPos** is used as the end position for the jog move. If a **`j:*`** command is given, the value of **Motor[*x*].ProgJogPos** is used as the distance from the present commanded position for the jog move. If a **`j^*`** command is given, the value of **Motor[*x*].ProgJogPos** is used as the distance from the present actual position for the jog move.

Motor[x].pSineTable

Description: Pointer to commutation current lookup table

Range: Legitimate addresses

Units: Data structure element addresses

Default: **Sys.SineTable[0].a**

Motor[*x*].pSineTable specifies the starting address of the 2048-entry lookup "sine" table for current feedback values in Power PMAC digital current-loop algorithm. This table is used to transform the actual phase-current feedback inputs into "direct" and "quadrature" components for current-loop closure.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

In the large majority of cases, **Motor[*x*].pSineTable** will specify the address of the standard sine table that is provided by Power PMAC. If use of a different table is desired, it should be loaded into 2048 consecutive registers in the user shared-memory buffer (each register containing a four-byte single-precision floating-point value, with the set covering 360 degrees), and **Motor[*x*].pSineTable** set to the starting address of this table.

For example, if the custom table is loaded into user shared memory buffer using **Sys.Fdata[4096]** (for the entry at zero degrees) through **Sys.Fdata[6143]** (for the entry at 359.824 degrees), **Motor[*x*].pSineTable** would be set to **Sys.pushm**+16384. (Each **Fdata** element occupies four bytes, so 4096 * 4 = 16,384.)

Motor[x].pVoltSineTable

Description: Pointer to commutation voltage lookup table

Range: Legitimate addresses

Units: Data structure element addresses

Default: **Sys.SineTable[0].a**

Motor[*x*].pVoltSineTable specifies the starting address of the 2048-entry lookup table for phase output values in the Power PMAC commutation algorithm, whether or not Power PMAC is closing the current loop for the motor. This table is used to transform the calculated "direct" and "quadrature" output commands into individual phase commands. Note that the table used to transform actual current feedback inputs into "direct" and "quadrature" components if Power PMAC is closing the current loop is specified by the separate element **Motor[*x*].pSineTable**.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

In the large majority of cases, **Motor[*x*].pVoltSineTable** will specify the address of the standard sine table that is provided by Power PMAC. If use of a different table is desired, it should be loaded into 2048 consecutive registers in the user buffer (each register containing a four-byte single-precision floating-point value, with the set covering 360 degrees), and **Motor[*x*].pVoltSineTable** set to the starting address of this table.

For example, if the custom table is loaded into user shared memory buffer using **Sys.Fdata[6144]** (for the entry at zero degrees) through **Sys.Fdata[8191]** (for the entry at 359.824 degrees), **Motor[*x*].pVoltSineTable** would be set to **Sys.pushm**+24576. (Each **Fdata** element occupies four bytes, so 6144 * 4 = 24,576.)

Motor[*x*].PwmDbComp

Description: PWM deadband compensation voltage offset magnitude

Range: -32,768 .. 32,767

Units: PWM output circuit units

Default: 0

Motor[*x*].PwmDbComp specifies the magnitude of the voltage offset applied to each phase PWM voltage command output value in direct-PWM mode to compensate for the deadband effect created by the turn-on/turn-off times of the power transistors and the resultant required deadtime. If it is set to its default value of 0, these deadband-compensation calculations are not performed.

If the magnitude of the commanded current for a phase is greater than that of saved setup element **Motor[*x*].PwmDbI**, compensation is applied to phase voltages. If the phase current is positive with at least this magnitude, twice the value of **Motor[*x*].PwmDbComp** is added to the value of the phase voltage command in **Motor[*x*].IaVolts**, **IbVolts**, or **IcVolts**, and the value of **Motor[*x*].PwmDbComp** is subtracted from the other two phase voltage commands. If the phase current is negative with at least this magnitude, twice the value of **Motor[*x*].PwmDbComp** is subtracted from the value of the phase voltage command, and the value of **Motor[*x*].PwmDbComp** is added to the other two phase voltage commands. This calculation is done for all three motor phases, based on each motor's measured phase current.

Motor[*x*].PwmDbComp is expressed in the units of the PWM output circuit for the servo interface used. If the PMAC2-style DSPGATE1 ASIC is used to generate the PWM outputs, these units are determined by the value of saved setup element **Gate1[*i*].PwmPeriod**. That is, a +100% voltage command (phase voltage command equals $+V_{DCbus}$) is generated by a command value equal to +**Gate1[*i*].PwmPeriod**, and a -100% voltage command (phase voltage command equals $-V_{DCbus}$) is generated by a command value equal to -**Gate1[*i*].PwmPeriod**.

If the PMAC3-style DPSGATE3 ASIC is used to generate the PWM outputs, the units are fixed in a +/-16,384 range, regardless of the PWM period/frequency. That is, a +100% voltage command (phase voltage command equals $+V_{DCbus}$) is generated by a command value of +16,383, and a -100% voltage command (phase voltage command equals $-V_{DCbus}$) is generated by a command value of -16,384.

Motor[*x*]. PwmDbComp is only used if commutation is enabled for the motor and **Motor[*x*].pAdc** > 0 to activate digital current loop execution.

Motor[*x*].PwmDbI

Description: PWM deadband compensation current threshold

Range: -32,768 .. 32,767

Units: 16-bit current LSBs

Default: 0

Motor[*x*].PwmDbI specifies the minimum magnitude of the commanded current in a phase for the voltage offset in **Motor[*x*].PwmDbComp** to be applied to compensate for the deadband effect at the zero-crossing in direct-PWM mode. Every phase cycle, the values of the commanded currents in each of the motor phases are compared to **Motor[*x*].PwmDbI**.

If the magnitude of the commanded current for a phase is greater than **Motor[*x*].PwmDbI**, offsets based on the value of **Motor[*x*].PwmDbComp** are applied to the motor phase voltage output values to try to minimize the distortion to the physical voltage and current waveforms from the zero-crossing deadband. The direction of the offsets applied is dependent on the sign of the current value. However, if the magnitude of the commanded current for the phase is less than or equal to **Motor[*x*].PwmDbI**, no offsets are applied to phase voltage output values as a result of this phase's current.

Motor[*x*]. PwmDbI is only used if commutation is enabled for the motor and **Motor[*x*].pAdc** > 0 to activate digital current loop execution.

Motor[*x*].PwmSf

Description: PWM output scale factor

Range: 0 .. 32,767

Units: PWM counter units

Default: 7636

Legacy I-variable alias: I*x*66

Motor[*x*].PwmSf multiplies the outputs of the commutation or digital current-loop algorithms (which are interpreted here as normalized values with a range of -1.0 to +1.0) before they are written to the output registers. As such, it determines both the scaling of the outputs, and the maximum magnitude of the outputs.

If the motor is not performing digital current-loop closure (**Motor[*x*].pAdc** = 0), typically the maximum permitted value of 32,767 is desired here to permit usage of the full range of output D/A converters (treated here as having signed 16-bit resolution, even if the actual resolution is

different). Smaller values can be used, but it must be kept in mind that they reduce the effective gain of the system.

If the motor is performing digital current-loop closure (**Motor[*x*].pAdc** > 0), **Motor[*x*].PwmSf** is usually set in relationship to the numerical range of the PWM generation circuitry. In a PMAC2-style "DSPGATE1" Servo IC, this range is controlled by the setup element **Gate1[*i*].PwmPeriod**. In a PMAC3-style "DSPGATE3" IC, the numerical range of the PWM generation circuitry is always +/-16,384, so **Motor[*x*].PwmSf** should be set relative to this value.

Setting **Motor[*x*].PwmSf** equal to the full-range magnitude (**Gate1[*i*].PwmPeriod** or 16,384) permits the outputs just to get to 0% and 100% duty cycle at the peaks of their waveforms (50% duty cycle is a net zero command).

If the direct-PWM amplifier requires that the waveform never get all the way to 0% or 100% duty cycle (usually to keep a charge pump active), then **Motor[*x*].PwmSf** must be set less than the full-range magnitude, typically less than 95%. Refer to the specific amplifier manual for details.

Otherwise, **Motor[*x*].PwmSf** can be set equal to or greater than the full-range magnitude. Values greater than the full-range magnitude will drive the waveforms into saturation over part of the commutation cycle when the input commands are large, but this can still generate useful additional torque. Fundamentally, for large commands, the waveforms are transformed from sinusoidal to trapezoidal. **Motor[*x*].PwmSf** values up to 17% larger than **Gate1[*i*].PwmPeriod** can be useful. At the default **Gate1[*i*].PwmPeriod** value of 6527, a value of **Motor[*x*].PwmSf** of 7636 gives this margin.

In a PMAC3-style "DSPGATE3" IC, the numerical range of the PWM generation circuitry is always +/-16,384, so **Motor[*x*].PwmSf** should be set relative to this value.

Motor[*x*].RapidSpeedSel

Description: Rapid mode speed select

Range: 0 .. 1

Units: none

Default: 1

Legacy I-variable alias: I*x*90

Motor[*x*].RapidSpeedSel specifies which setup data structure element is used to determine the speed of a **`rapid`**-mode move for the motor. If it is set to 0, **Motor[*x*].JogSpeed** is used. If it is set to the default value of 1, **Motor[*x*].MaxSpeed** is used.

Note that if **Coord[*x*].RapidVelCtrl** is set to 1 for the coordinate system to which the motor belongs, in a multi-axis rapid move, if this motor has a smaller distance-to-speed ratio than other motors used in the move, its commanded velocity will be reduced so that its move time matches that of the motor with the largest distance-to-speed ratio.

Motor[*x*].RapidSpeedSel constitutes bit 9 of the full-word element **Motor[*x*].Control[0]**.

Motor[x].ServoCaptTimeOffset

Description: Offset from servo feedback sampling to interrupt

Range: 32-bit integers

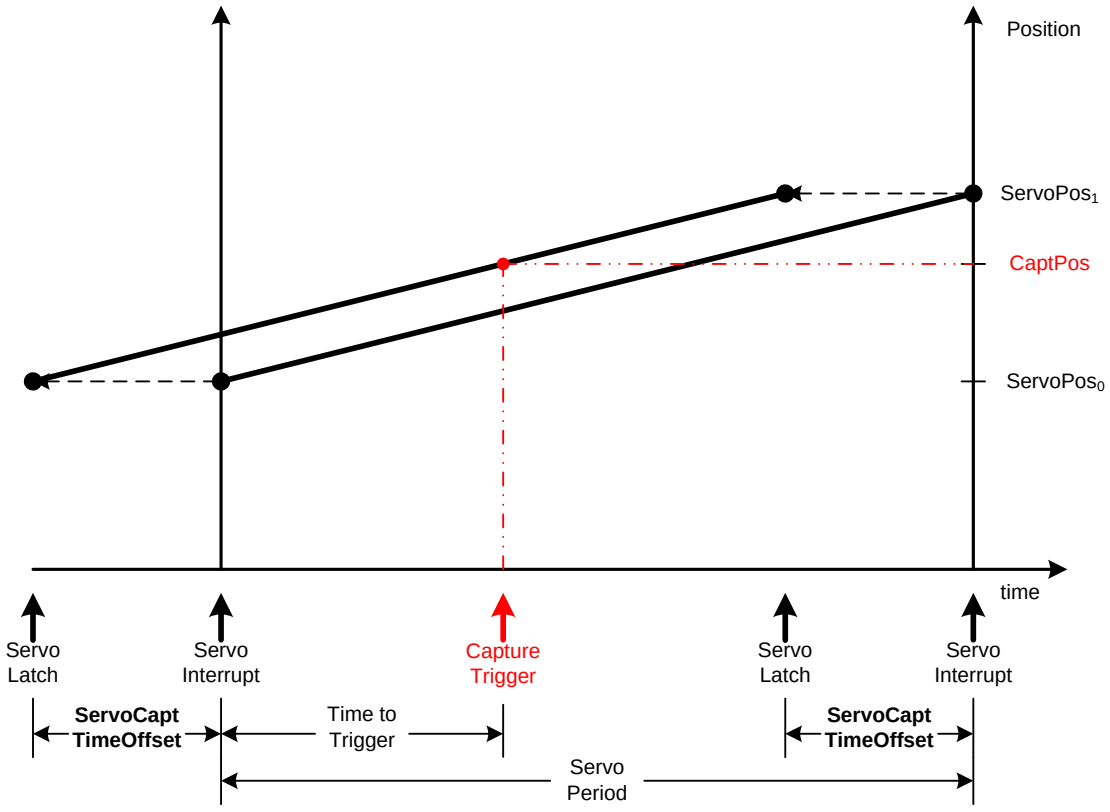
Units: 1/65536 servo cycle

Default: 0

Motor[x].ServoCaptTimeOffset specifies the time before the servo interrupt (falling edge of the servo clock) at which the (outer-loop) position feedback is sampled (latched or strobed). This value is used in the calculations for timer-assisted software capture (specified by setting **Motor[x].CaptureMode** to 3). It does not need to be set for any other purpose.

Many types of feedback, particularly those with a serial data interface to the Power PMAC, such as serial encoders and those feedback types using serial analog-to-digital converters, must be sampled significantly before the servo interrupt to provide time to transmit the data to the Power PMAC so that it is ready for use by servo-interrupt tasks. If the delay between the sampling and the interrupt is not taken into account, then the calculations using the timer value latched by the capture trigger in this mode will not be accurate.

This diagram shows graphically how the delay specified by this element is used to compute the captured position value in timer-assisted software capture.



Most Power PMAC serial A/D-converter interfaces strobe the analog signal on the rising edge of the phase clock and transmit the data back to the Power PMAC so it is ready on the falling edge of the phase clock. The phase clock has a frequency *n* times that of the servo clock, where *n* is a positive integer, and the falling edge of the servo clock (which generates the processor interrupt) is always coincident with a falling edge of the phase clock.

For serial A/D feedback sampled on the rising edge of the phase clock, as with an ACC-28E or ACC-59E3, and with the default setting of 4 phase-clock cycles per servo-clock cycle (**Gate*n*[*i*].ServoClockDiv** at the default value of 3), the feedback is sampled 1/8 of a servo cycle before the servo interrupt, so **Motor[*x*].ServoCaptTimeOffset** should be set to 65,536/8 = 8,192.

Serial encoders connected through the ACC-84E UMAC serial-encoder interface board or the DSPGATE3 ASIC used on the ACC-24E3 UMAC axis-interface board and the Power Brick control board can be sampled on the rising or falling edge of either the phase clock or servo clock. In general, the clock edge that provides the minimum delay from sampling to interrupt while providing sufficient time for the data to be transmitted to the Power PMAC to be ready for interrupt calculations is used.

If the encoder is sampled on the falling edge of the phase clock, there is a full phase-clock cycle delay before it is used in the next servo interrupt. With the default setting of 4 phase-clock cycles per servo-clock cycle, the feedback is sampled ¼ of a servo cycle before the servo interrupt, so **Motor[*x*].ServoCaptTimeOffset** should be set to 65,536/4 = 16,384.

The servo clock signal is derived by dividing down from the phase clock, with its falling edge coincident with one of the phase clock falling edges. It is low for at most one phase clock cycle. At the default setting of 4 phase-clock cycles per servo-clock cycle, it is low for 1 phase-clock cycle and high for 3. If the encoder is sampled on the rising edge of the servo clock, it is sampled ¾ of a servo cycle before the servo interrupt, so **Motor[*x*].ServoCaptTimeOffset** should be set to 65,536*3/4 = 49,152.

Note that it is possible to delay the actual sampling of the serial encoder from the specified clock edge. If this is done, the calculation of **Motor[*x*].ServoCaptTimeOffset** must take this into account.

Parallel-data servo feedback, as from interferometers through an ACC-14E I/O card, is typically sampled on the falling edge of the servo interrupt, in which case **Motor[*x*].ServoCaptTimeOffset** should be set to 0.

Motor[*x*].ServoCtrl

Description: Control flag to activate servo tasks

Range: 0 .. 15

Units: Bit field

Default: 0

Legacy I-variable alias: I*x*00

Motor[*x*].ServoCtrl determines what tasks are done during the servo loop, and how they are done. It is a 4-bit value, potentially permitting 16 modes of operation, but only a few modes are presently implemented.

If **Motor[*x*].ServoCtrl** is set to the default value of 0, no calculations are performed for the motor, not even position monitoring, so a position query command would not reflect position changes. Any Power PMAC motor not used at all in an application should be de-activated, so Power PMAC does not waste time doing calculations for that motor.

If **Motor[*x*].ServoCtrl** is set to 1, the motor is activated in standard mode. In this mode, position monitoring, servo, and trajectory calculations are performed for the motor. An "activated" motor may be "enabled" – either in open or closed-loop mode – or "disabled" (killed), depending on commands or events.

If **Motor[*x*].ServoCtrl** is set to 8, the motor is activated in "gantry following" mode. In this mode, position monitoring and servo calculations are performed for the motor, but it does not do its own trajectory calculations. Instead, it uses the trajectory calculations of the motor specified by **Motor[*x*].CmdMotor**. This feature permits multiple motors to use the same command trajectory, whether from an axis program move, or a jog move of the "gantry leader" motor. The gantry leader motor should be activated in standard mode (**Motor[*x*].ServoCtrl** = 1) Note that the "gantry following" motors have zero lag to the "gantry leader" motor (not even 1 servo cycle).

WARNING

Do not try to de-activate an active and enabled motor by setting **Motor[*x*].ServoCtrl** to 0! The motor outputs would be left enabled with the last command level on them. This could lead to a dangerous runaway condition

Motor[*x*].ServoCtrl constitutes bits 28 – 31 of the full-word element **Motor[*x*].Control[0]**.

Motor[*x*].SlewMasterPosSf

Description: Position following ratio slew rate

Range: Non-negative floating-point

Units: (Motor units per master unit) per servo cycle

Default: 0.0 (slew rate control disabled)

Motor[*x*].SlewMasterPosSf specifies the rate of change of the position-following ratio on enabling and disabling of the position-following function, and on changes in the desired ratio in **Motor[*x*].MasterPosSf**. If **Motor[*x*].SlewMasterPosSf** is set to the default value of 0.0, slew rate control is disabled, and changes in the following ratio take place immediately as step changes.

If **Motor[*x*].SlewMasterPosSf** is set to a value greater than 0.0, slew rate control is enabled, and each servo cycle, the actual position-following ratio used in that servo cycle, found in status

element **Motor[*x*].ActiveMasterPosSf**, will change by the magnitude of **Motor[*x*].SlewMasterPosSf** until the desired value in **Motor[*x*].MasterPosSf** is reached, or in the case of disabling following, until 0.0 is reached.

If it is desired to enable or disable following, or to change the following ratio, while the master signal is changing ("on-the-fly" changes), it is important to set **Motor[*x*].MasterPosSf** to an appropriate positive value so the resulting motor acceleration or deceleration is properly controlled.

Motor[*x*].SlipGain

Description: Motor commutation command/slip constant

Range: Non-negative floating point

Units: Commutation angle units per 16-bit torque command unit

Default: 0.0

Motor[*x*].SlipGain is the constant of proportionality between the servo-loop output command and the "slip" in the commutation angle. The slip is the advance in the rotor magnetic field angle relative to the measured physical angle of the rotor. **Motor[*x*].SlipGain** can be used in two very different ways for different types of motor control.

For the control of asynchronous induction motors, **SlipGain** is automatically calculated as an intermediate value each phase cycle using the values of saved setup elements **Motor[*x*].DtOverRotorTc** and **Motor[*x*].IdCmd**. In this case, it is used as an internal status element that can generally be ignored by the user, except possibly in analyzing the operation of advanced techniques such as field weakening, and the saved value of **SlipGain** is automatically overwritten.

For the open-loop "direct microstepping" control of synchronous AC motors (typically those marketed as "stepper" motors), **Motor[*x*].SlipGain** is set directly by the user. For this control mode, **Motor[*x*].DtOverRotorTc** should be set to 0.0, which disables the automatic internal calculation of **SlipGain**. In direct microstepping control, there is no measured commutation angle feedback, so the "slip" generated from this parameter is all that advances the commutation angle to drive the motor. In this case, the output of the (simulated) servo loop is a velocity command; when it is multiplied by **SlipGain**, the commutation angle is advanced as much as needed to command that velocity in the motor.

The setting of **Motor[*x*].SlipGain** in direct microstepping must translate the velocity command from the simulated servo loop to the appropriate advance of the commutation angle. In the recommended setup, where the motor's servo units are individual microsteps (2048 per commutation cycle), **SlipGain** should be set to 1 / (Phase-cycles-per-servo-cycle). At the default setting of 4 phase cycles per servo cycle, this value will be 1/4, or 0.25.

Motor[x].SoftLimitOffset

Description: Distance between calculation-time and execution-time software limits

Range: Floating-point

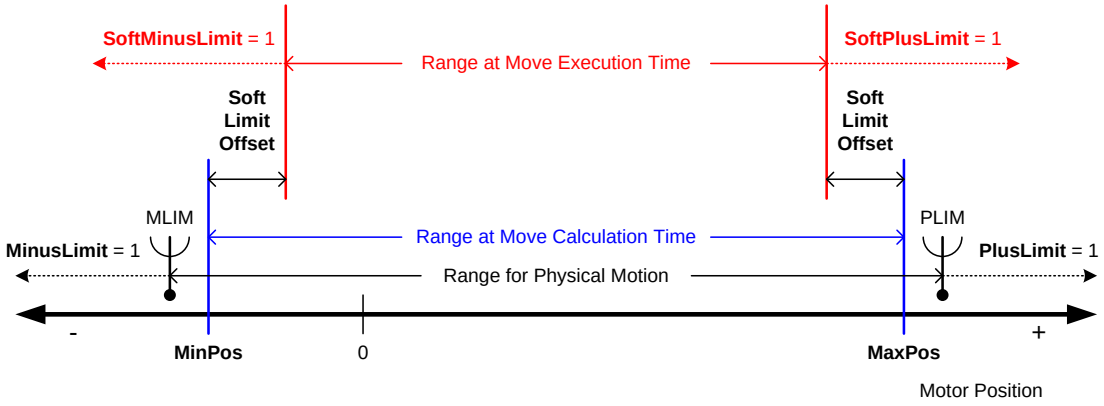
Units: Motor units

Default: 0.0

Legacy I-variable alias: I*x*41

Motor[*x*].SoftLimitOffset specifies the signed distance between the software position limits used at move calculation time and those used during move execution. At move calculation time, Power PMAC uses **Motor[*x*].MinPos** at the negative end, and **Motor[*x*].MaxPos** at the positive end. If the endpoint of the move is past the limit, the move will not be executed, or its endpoint will be changed to the limit position, depending on the type of move. Segmented moves (**`linear`**, **`circle`**, and **`pvt`** mode moves with **Coord[*x*].SegMoveTime** > 0) are checked as each segment is calculated, not at the initial move calculation time.

This diagram shows the relative positions of the execution-time software limits compared to the calculation-time limits for the typical case where the execution-time limits are "inside".



At move-execution time, Power PMAC continually checks motor instantaneous net desired position – from both computed trajectories and master following (and from actual position in open-loop mode) – against (**Motor[*x*].MinPos - Motor[*x*].SoftLimitOffset**) at the negative end, and (**Motor[*x*].MaxPos** + **Motor[*x*].SoftLimitOffset**) at the positive end. If one of these thresholds is exceeded (not just met), the appropriate motor status bit **Motor[*x*].SoftMinusLimit** or **Motor[*x*].SoftPlusLimit** is automatically set.

If **Motor[*x*].SoftLimitOffset** is zero or positive, so the thresholds at move execution time are "outside" the move-calculation thresholds, and one of these thresholds is exceeded at move-execution time, all motors in the coordinate system are aborted, brought to a controlled stop as specified by **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**.

If **Motor[*x*].SoftLimitOffset** is negative, so the thresholds at move execution time are "inside" the move-calculation thresholds, no motors in the coordinate system are aborted when one of this motor's move-execution threshold is exceeded. However, the appropriate status bit is still set to

indicate that the motor is presently "in the limit". Users who want these status bits set when a move that was limited at move calculation time actually reaches the limit will want to set **Motor[*x*].SoftLimitOffset** to a very small negative value (e.g. equal to 1 feedback "count").

Note that no software position limit checks are performed, either at move calculation time or move execution time, unless **Motor[*x*].MaxPos** is greater than **Motor[*x*].MinPos**.

Motor[*x*].Stime

Description: Servo cycle period extension

Range: 0 .. 2^{31}-1

Units: Servo interrupt periods

Default: 0

Legacy I-variable alias: I*x*60

Motor[*x*].Stime specifies the number of servo interrupt periods that are skipped between successive position/velocity-loop closures for the motor. This permits an extension of the servo update time for the motor beyond a single servo interrupt period. The servo loop will be closed every (**Motor[*x*].Stime** + 1) servo interrupts. With the default value of zero, the loop will be closed every servo interrupt. An extended servo update time can be useful for motors with slow dynamics, and/or limited feedback resolution. It can also be useful if the control loop is used for a slow process-control function, instead of an actual motor.

If the servo loop is closed under the phase interrupt, as specified by setting bit 3 (value 8) of **Motor[*x*].PhaseCtrl** to 1, then **Motor[*x*].Stime** specifies the number of phase interrupt periods that are skipped between successive position/velocity-loop closures for the motor.

Caution

Changing the value of **Motor[*x*].Stime** changes the dynamics of the servo loop, possibly resulting in unpredictable performance. The servo loop should always be (re)tuned after a change.

Other update times, including trajectory update (interpolation) and commutation update, are not affected by **Motor[*x*].Stime**. The global setup element **Sys.ServoPeriod** does *not* need to be changed with **Motor[*x*].Stime**.

Motor[*x*].TraceSize

Description: Motor move trace buffer length

Range: Non-negative integers

Units: Move segments or sections

Default: 0

Motor[*x*].TraceSize, if set to a value greater than the default of 0, specifies the length of the move trace buffer for the motor. This buffer stores the equations of motion for already executed moves. If the moves are not segmented, each section of a programmed move with different equations (e.g. constant acceleration, constant speed sections) occupies one slot in the buffer. If the moves are segmented, each segment of **Coord[*x*].SegMoveTime** occupies one slot in the buffer.

The presence of a trace buffer permits reverse execution of already executed moves with negative "time base" values for the coordinate system to which the motor is assigned. Reverse execution can proceed through a continuous motion sequence back to the beginning of the stored trace buffer. A continuous motion sequence can include zero-velocity moves and **`delay`** times, but not **`dwell`** times or breaks in the sequence for logical reasons (e.g. end of program).

Path reversal through motor trace buffers is similar to path reversal through lookahead buffer history, but it is better suited to "proportional" backward/forward control instead of "state change" backward/forward control.

All motors assigned to positioning axes in a coordinate system should have identically sized trace buffers to permit full reversal of programmed paths using negative time base in the coordinate system.

Motor[*x*].WarnFeLimit

Description: Warning (trigger) following error limit

Range: Non-negative floating-point

Units: Motor units

Default: 1000

Legacy I-variable alias: I*x*12

Motor[*x*].WarnFeLimit sets the magnitude of the following error for the specified motor at which warning flags are set. When the magnitude of the following error exceeds **Motor[*x*].WarnFeLimit**, status bits are set for the motor and for the coordinate system to which it is assigned (if any). The coordinate system status bit is the logical OR of the status bits for all motors assigned to that coordinate system.

Setting this parameter to zero disables the warning following error limit function. If this parameter is set greater than **Motor[*x*].FatalFeLimit**, the warning status bit will never go true, because the fatal limit will disable the motor first.

If bit 1 (value 2) of **Motor[*x*].CaptureMode** is set to 1, the motor can be "triggered" for homing search moves, jog-until-trigger moves, and motion program move-until-trigger moves when the following error exceeds **Motor[*x*].WarnFeLimit**. This is known as "torque-mode" triggering, because the trigger will occur at a torque level corresponding to this error limit.

Motor Servo Loop Term Elements

The setup elements in the section are part of the **Motor[*x*].Servo** sub-structure. They are used to compute the servo output command based on the commanded trajectory and the feedback values. These values are usually set using the Tuning utility in the Integrated Development Environment (IDE) program, either using the auto-tuning or interactive tuning facility, or both.

Motor[*x*].Servo.BreakPosErr

Description: Servo gain-break error size

Range: Non-negative floating-point

Units: Motor units

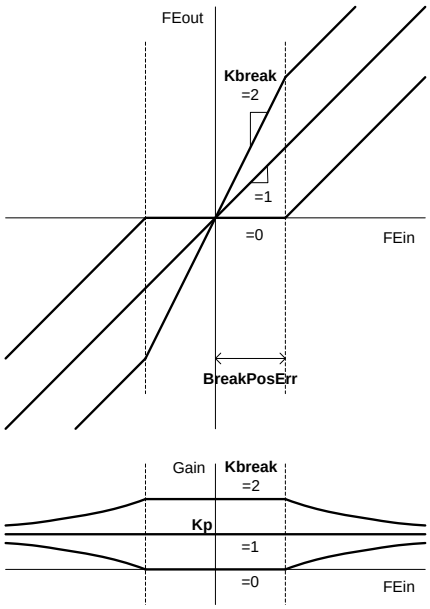
Default: 0.0

Legacy I-variable alias: I*x*65

Motor[*x*].Servo.BreakPosErr defines the size of the position error band, measured from zero error in either direction, within which there will be changed (or no) control effort, for the feature known as "gain break" or "deadband compensation". The gain-break zone is typically used to create a "deadband" algorithmically or to compensate for a physical deadband effect.

Motor[*x*].Servo.Kbreak controls the effective gain within this band, effectively multiplying the standard proportional gain term **Motor[*x*].Servo.Kp**. Outside the gain-break zone, the gain asymptotically approaches **Motor[*x*].Servo.Kp**. When **Motor[*x*].Servo.BreakPosErr** is set to its default value of 0.0, there is no special band, and **Motor[*x*].Servo.Kbreak** is not used.

The following diagram shows the effect of **Motor[*x*].Servo.BreakPosErr**:



Motor[*x*].Servo.BreakPosErr is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[*x*].Servo.EstMinDac

Description: Minimum servo command level for active adaptation

Range: 0 .. 32,767

Units: 16-bit command units

Default: 0

Motor[*x*].Servo.EstMinDac specifies the minimum servo-loop command output value for which the Power PMAC will use the servo cycle's data to estimate the plant gain for its adaptive control algorithms. It is only used if the adaptive servo control algorithm is selected. It is in units of a 16-bit output device, which has a range of +/-32K, and despite the name of this element, the output device does not need to be a D/A converter.

The recursive plant-gain estimation calculations will only be performed for those servo cycles where the desired acceleration value is non-zero and the servo command output magnitude is at least this value. When the command output magnitude is very low, plant acceleration will not be very well related to command level, due to offset, noise, and disturbance effects, so plant-gain estimation calculations will not be robust. Typical values for **Motor[*x*].Servo.EstMinDac** are 1 to 5% of the **Motor[*x*].MaxDac** servo output limit.

Motor[*x*].Servo.EstMinDac is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is not used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Motor[*x*].Servo.EstTime

Description: Number of servo cycles in active adaptation

Range: Non-negative integer

Units: Servo cycles

Default: 0 (disabled)

Motor[*x*].Servo.EstTime specifies the number of servo cycles of data to be used in the recursive algorithm to calculate the present estimated plant gain in the adaptive servo control algorithm. It is only used if the adaptive servo control algorithm is selected. At the default value of 0, the estimation/adaptation algorithm is disabled, even if the adaptive control algorithm is selected.

The algorithm will use servo-command output and position feedback data from the past consecutive **Motor[*x*].Servo.EstTime** servo cycles to estimate the ratio between commanded torque output and resulting measured acceleration. Only those servo cycles where the desired acceleration value is non-zero and the servo-command output magnitude is at least as large as

Motor[*x*].EstMinDac are used. The resulting plant gain value is stored in status element **Motor[*x*].Servo.EstGain**, and is used to calculate the servo algorithms gain-adjustment value **Motor[*x*].Servo.GainFactor.**

Lower values of **Motor[*x*].Servo.EstTime** will provide faster response to plant inertia changes. However, they will be more sensitive to errors from measurement and quantization noise, and physical disturbances. Generally, this should be set to a value of at least 50 servo cycles, and the most common range is 100 to 500 at the default servo frequency of 2.26 kHz.

Motor[*x*].Servo.EstTime is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is not used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Motor[*x*].Servo.Kafb

Description: PID acceleration feedback gain term

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit per servo cycle per servo cycle) of actual acceleration

Default: 0.0

Motor[*x*].Servo.Kafb adds an amount to the control output proportional to the desired acceleration of the motor. It is intended to reduce tracking error due to inertial lag.

Motor[*x*].Servo.Kafb is the servo loop's second-derivative gain term, providing a contribution subtracted from the control output proportional to the actual acceleration feedback (this cycle's actual velocity minus last cycle's actual velocity, using the "**pEnc2**" feedback source) of the motor. It acts effectively as an electronic inertia. The higher **Motor[*x*].Servo.Kafb** is, the heavier the inertia effect is.

Motor[*x*].Servo.Kafb is simply multiplied by the actual acceleration to compute its contribution to the servo output, scaled as a 16-bit signed output. That is, an actual acceleration of 1.0 motor units per servo cycle per servo cycle, when multiplied by a **Motor[*x*].Servo.Kafb** of 1.0, contributes a value of 1.0 LSBs of a 16-bit output. (This would be 4.0 LSBs of an 18-bit DAC.)

Because double digital differentiation introduces a lot of quantization noise, this term should not be used unless the actual velocity is filtered first with the "E" polynomial filter elements **Motor[*x*].Servo.Ke1** and **Motor[*x*].Servo.Ke2**.

Motor[*x*].Servo.Kafb is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[x].Servo.Kaff

Description: PID acceleration feedforward gain term

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit per servo cycle per servo cycle) of command acceleration

Default: 0.0

Legacy I-variable alias: I*x*35

Motor[*x*].Servo.Kaff adds an amount to the control output proportional to the desired acceleration of the motor. It is intended to reduce tracking error due to inertial lag.

Motor[*x*].Servo.Kaff is simply multiplied by the net desired acceleration to compute its contribution to the servo output, scaled as a 16-bit signed output. That is, a desired acceleration of 1.0 motor units per servo cycle per servo cycle, when multiplied by a **Motor[*x*].Servo.Kaff** of 1.0, contributes a value of 1.0 LSBs of a 16-bit output. (This would be 4.0 LSBs of an 18-bit DAC.)

Note that this contribution is added in to the *output* of the integral gain term. (There are velocity feedforward terms for both the input and the output of the integrator.)

If the servo update time is changed, **Motor[*x*].Servo.Kaff** must be changed by the inverse square to keep the same effect. For instance, if the servo update time is cut in half, **Motor[*x*].Servo.Kaff** must be quadrupled to keep the same effect.

For equivalent action to a Turbo PMAC servo algorithm, this gain can be set as:

$$Motor[x].Servo.Kaff = Ixx35 * \frac{Ixx08 * Ixx30}{2^{26} * Motor[x].PosSf}$$

Motor[*x*].Servo.Kaff is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[x].Servo.Ka*i*

Description: Desired-position polynomial n^{th}-order numerator term (i = 0 to 7)

Range: Floating-point

Units: none (unit-less z-transform coefficient)

Default: 1.0 (**Ka0**), 0.0 (**Ka1** – **Ka7**)

The **Motor[*x*].Servo.Ka*i*** terms are double-precision coefficients of a 2nd-order or 7th-order "A" numerator polynomial filter acting on the net desired position into the servo algorithm. The transfer function of this polynomial is:

$$A(z) = Ka_0 + Ka_1 z^{-1} + Ka_2 z^{-2} + Ka_3 z^{-3} + Ka_4 z^{-4} + Ka_5 z^{-5} + Ka_6 z^{-6} + Ka_7 z^{-7}$$

The sum of all **Ka*i*** terms used must equal exactly 1.0 in order for the filter not to have any net scaling effect on the signal (i.e. to have a "DC gain" of 1.0). Typically, **Ka1** and higher are set to obtain the desired dynamics, then **Ka0** is set to 1.0 minus the sum of these terms so there is no net scaling due to the filter.

The **Motor[*x*].Servo.Ka*i*** terms are used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. They are *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

If **Motor[*x*].Servo.SwPoly7** is set to the default value of 0, only the **Ka1** and **Ka2** terms will be used in this section, resulting in a 2nd-order polynomial. If **Motor[*x*].Servo.SwPoly7** is set to 1, all 7 **Ka*i*** terms will be used in this section, resulting in a 7th-order polynomial, capable of more effect, but taking longer to execute.

Motor[*x*].Servo.Kb*i*

Description: Actual-position polynomial i^{th}-order numerator term (i = 0 to 7)

Range: Floating-point

Units: none (unit-less z-transform coefficient)

Default: 1.0 (**Kb0**), 0.0 (**Kb1** – **Kb7**)

The **Motor[*x*].Servo.Kb*i*** terms are double-precision coefficients of a 2nd-order or 7th-order "B" numerator polynomial filter acting on the net actual outer-loop position into the servo algorithm. The transfer function of this polynomial is:

$$B(z) = Kb_0 + Kb_1 z^{-1} + Kb_2 z^{-2} + Kb_3 z^{-3} + Kb_4 z^{-4} + Kb_5 z^{-5} + Kb_6 z^{-6} + Kb_7 z^{-7}$$

The sum of all **Kb*i*** terms used must equal exactly 1.0 in order for the filter not to have any net scaling effect on the signal (i.e. to have a "DC gain" of 1.0). Typically, **Kb1** and higher are set to obtain the desired dynamics, then **Kb0** is set to 1.0 minus the sum of these terms so there is no net scaling due to the filter.

The **Motor[*x*].Kb*i*** terms are used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. They are *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

If **Motor[*x*].Servo.SwPoly7** is set to the default value of 0, only the **Kb1** and **Kb2** terms will be used in this section, resulting in a 2nd-order polynomial. If **Motor[*x*].Servo.SwPoly7** is set to 1, all 7 **Kb*i*** terms will be used in this section, resulting in a 7th-order polynomial, capable of more effect, but taking longer to execute.

Motor[*x*].Servo.Kbreak

Description: Servo "gain break" relative gain

Range: Non-negative floating-point

Units: Fraction of proportional gain **Kp**

Default: 0.0

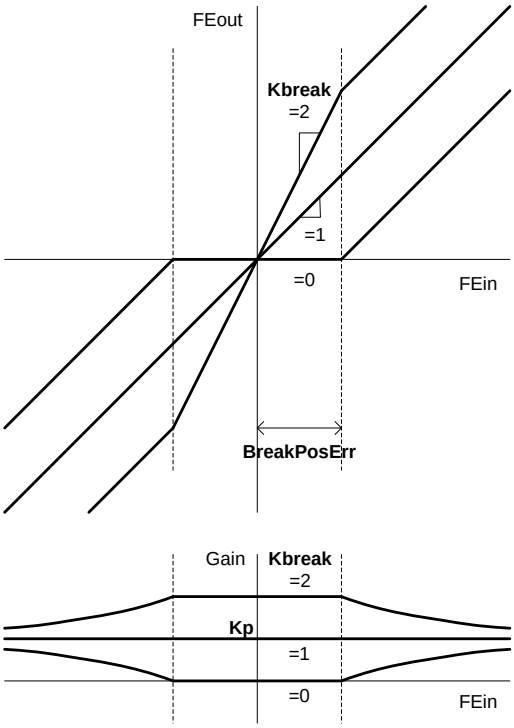
Legacy I-variable alias: I*x*64

Motor[*x*].Servo.Kbreak controls the servo loop's proportional gain inside the "gain-break" zone, the band of position error of magnitude less than **Motor[*x*].Servo.BreakPosErr**. The gain-break zone is typically used to create a "deadband" algorithmically or to compensate for a physical deadband effect.

Motor[*x*].Servo.Kbreak is expressed relative to the standard proportional gain term **Motor[*x*].Servo.Kp** – that is, a value of 1.0 for **Motor[*x*].Servo.Kbreak** provides the same gain inside the gain-break zone as outside (which would defeat the purpose of the zone); a value of 0.0 would create a deadband in this zone; a value of 2.0 would provide double the gain inside the zone (which could compensate for a physical deadband).

Outside the gain-break zone, the gain asymptotically approaches the standard proportional gain term **Motor[*x*].Servo.Kp**.

The following diagram shows the effect of common settings of **Motor[*x*].Servo.Kbreak**:



Motor[*x*].Servo.Kbreak is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

Motor[*x*].Servo.Kc*i*

Description: Servo-error polynomial i^{th}-order numerator term (i = 1 to 7)

Range: Floating-point

Units: none (unit-less z-transform coefficient)

Default: 0.0

Legacy I-variable alias: I*x*36 (**Kc1**), I*x*37 (**Kc2**)

The **Motor[*x*].Servo.Kc*i*** terms are double-precision coefficients of a 2nd-order or 7th-order "C" numerator polynomial filter acting on the error in the outer (position) loop of the servo algorithm. The transfer function of this polynomial is:

$$C(z) = 1 + Kc_1 z^{-1} + Kc_2 z^{-2} + Kc_3 z^{-3} + Kc_4 z^{-4} + Kc_5 z^{-5} + Kc_6 z^{-6} + Kc_7 z^{-7}$$

Note that the "DC gain" of this filter is equal to the sum of 1 plus all of the coefficients used, and this gain is part of the overall servo-loop gain.

The **Motor[*x*].Servo.Kc*i*** terms are used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. They are *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

If **Motor[*x*].Servo.SwPoly7** is set to the default value of 0, only the **Kc1** and **Kc2** terms will be used in this section, resulting in a 2nd-order polynomial. If **Motor[*x*].Servo.SwPoly7** is set to 1, all 7 **Kc*i*** terms will be used in this section, resulting in a 7th-order polynomial, capable of more effect, but taking longer to execute.

Motor[*x*].Servo.Kd*i*

Description: Servo-error polynomial i^{th}-order denominator term (i = 1 to 7)

Range: Floating-point

Units: none (unit-less z-transform coefficient)

Default: 0.0

Legacy I-variable alias: I*x*38 (**Kd1**), I*x*39 (**Kd2**)

The **Motor[*x*].Servo.Kd*i*** terms are double-precision coefficients of a 2nd-order or 7th-order "D" denominator polynomial filter acting on the error in the outer (position) loop of the servo algorithm. The transfer function of this polynomial is:

$$D(z) = \frac{1}{1 + Kd_1 z^{-1} + Kd_2 z^{-2} + Kd_3 z^{-3} + Kd_4 z^{-4} + Kd_5 z^{-5} + Kd_6 z^{-6} + Kd_7 z^{-7}}$$

Note that the "DC gain" of this filter is equal to 1 divided by the sum of 1 plus all of the coefficients used, and this gain is part of the overall servo-loop gain.

The **Motor[*x*].Servo.Kd*i*** terms are used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. They are *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

If **Motor[*x*].Servo.SwPoly7** is set to the default value of 0, only the **Kd1** and **Kd2** terms will be used in this section, resulting in a 2nd-order polynomial. If **Motor[*x*].Servo.SwPoly7** is set to 1, all 7 **Kd*i*** terms will be used in this section, resulting in a 7th-order polynomial, capable of more effect, but taking longer to execute.

Motor[*x*].Servo.Ke*i*

Description: Inner-loop velocity polynomial i^{th}-order denominator term (i = 1 to 2)

Range: Floating-point

Units: none (unit-less z-transform coefficient)

Default: 0.0

The **Motor[*x*].Servo.Ke*i*** terms are double-precision coefficients of a 2nd-order "E" denominator polynomial filter acting on the actual position in the inner (velocity) loop of the servo algorithm. The transfer function of this polynomial is:

$$E(z) = \frac{1}{1 + Ke_1 z^{-1} + Ke_2 z^{-2}}$$

Note that the "DC gain" of this filter is equal to 1 divided by the sum of 1 plus all of the coefficients used, and this gain is part of the overall inner-loop gain. To retain the overall DC gain when these terms are changed, there will have to be counteracting changes in the **Kvfb** and/or **Kvifb** gains.

The **Motor[*x*].Servo.Ke*i*** terms are used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. They are *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

Motor[x].Servo.Kfi

Description: Desired-velocity polynomial i^{th}-order denominator term (i = 1 to 2)

Range: Floating-point

Units: none (unit-less z-transform coefficient)

Default: 0.0

The **Motor[*x*].Servo.Kf*i*** terms are double-precision coefficients of a 2nd-order "F" denominator polynomial filter acting on the net-desired velocity in the feedforward path of the servo algorithm. The transfer function of this polynomial is:

$$F(z) = \frac{1}{1 + Kf_1 z^{-1} + Kf_2 z^{-2}}$$

These terms are usually used to apply a low-pass filter to the feedforward signal when a master encoder with measurement and quantization noise is used in the generation of the motor's net desired trajectory (as in position-following and external time-base modes). Feedforward gains will magnify these high-frequency noise components, resulting in rough operation.

Generally, these terms will be left at zero for a purely mathematically generated trajectory, as there is no noticeable noise in the command trajectory in this case.

Note that the "DC gain" of this filter is equal to 1 divided by the sum of 1 plus the two coefficients, and this gain is part of the overall feedforward path.

The **Motor[*x*].Servo.Kf*i*** terms are used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. They are *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

Motor[x].Servo.Kfff

Description: Servo friction feedforward gain term

Range: Floating-point

Units: LSBs of 16-bit output

Default: 0.0

Legacy I-variable alias: I*x*68

Motor[*x*].Servo.Kfff adds a bias term to the servo loop output of the motor that is proportional to the *sign* of the net commanded velocity. That is, if the commanded velocity is positive, **Motor[*x*].Servo.Kfff** is added to the output. If the commanded velocity is negative, **Motor[*x*].Servo.Kfff** is subtracted from the output. If the commanded velocity is zero, no value is added to or subtracted from the output.

This parameter is intended primarily to help overcome errors due to mechanical friction. It can be thought of as a "friction feedforward" term. Because it is a feedforward term that does not utilize any feedback information, it has no direct effect on system stability. It can be used to correct the error resulting from friction, especially on turnaround, without the time constant and potential stability problems of integral gain.

The contribution of this term will be added directly to the servo output if the software switch **Motor[*x*].SwFffInt** is set to its default value of 0. It will be added instead to the input of the servo-loop integrator if **Motor[*x*].SwFffInt** is set to 1.

Motor[*x*].Servo.Kfff is scaled for a 16-bit signed output. That is, a **Motor[*x*].Servo.Kfff** of 1.0 contributes a value of +/-1.0 LSBs of a 16-bit output. (This would be +/-4.0 LSBs of an 18-bit DAC.)

For equivalent action to a Turbo PMAC servo algorithm, **Motor[*x*].Servo.Kfff** should be set to the same value as Ixx68 in Turbo PMAC.

Motor[*x*].Servo.Kfff is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

Motor[*x*].Servo.Ki

Description: PID integral gain term

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit * servo cycle) of integrated error

Default: 0.001

Legacy I-variable alias: I*x*33

Motor[*x*].Servo.Ki is the servo loop's main integral gain term, providing a control output proportional to the time integral of the (possibly modified) position error. The magnitude of the integrator's output is limited by **Motor[*x*].Servo.MaxInt**. With **Motor[*x*].Servo.MaxInt** at a value of 0.0, the contribution of the integrator to the servo output is zero, regardless of the value of **Motor[*x*].Servo.Ki**.

No further errors are added to the integrator if the output saturates (if output magnitude equals **Motor[*x*].MaxDac**), and, if **Motor[*x*].Servo.SwZvInt** = 1, when a move is being commanded (when desired velocity is not zero). In both of these cases, the contribution of the integrator to the output remains constant.

If the servo update time is changed, **Motor[*x*].Servo.Ki** must be changed proportionately in the same direction to keep the same effect. For instance, if the servo update time is cut in half, **Motor[*x*].Servo.Ki** must be cut in half to keep the same effect.

If velocity gain terms **Motor[*x*].Servo.Kvfb** (feedback) and **Motor[*x*].Servo.Kvff** (feedforward), which add to the *output* of the integrator, are used, the integrator is effectively in the position

loop. However, if velocity gain terms **Motor[*x*].Servo.Kvifb** (feedback) and **Motor[*x*].Servo.Kviff** (feedforward), which add to the *input* of the integrator, are used, the integrator is effectively in the velocity loop. In general, a position-loop integrator is better for trajectory tracking; a velocity-loop integrator is better for disturbance rejection.

For equivalent action to a Turbo PMAC servo algorithm, this gain can be set as:

$$Motor[x].Servo.Ki = \frac{Ixx33}{2^{23} * Ixx08}$$

Motor[*x*].Servo.Ki is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[*x*].Servo.Kp

Description: PID proportional gain term

Range: Floating-point

Units: LSBs of 16-bit output per motor unit of position error

Default: 4

Legacy I-variable alias: I*x*30

Motor[*x*].Servo.Kp is the servo loop's main proportional gain term, providing a control output proportional to the position error (commanded position minus actual position) of the motor. It acts effectively as an electronic spring. The higher **Motor[*x*].Servo.Kp** is, the stiffer the "spring" is. Too low a value will result in sluggish performance. Too high a value can cause a "buzz" from constant over-reaction to errors.

Motor[*x*].Servo.Kp is simply multiplied by the position error to compute its contribution to the servo output, scaled as a 16-bit signed output. That is, a position error of 1.0 motor units, when multiplied by a **Motor[*x*].Servo.Kp** of 1.0, contributes a value of 1.0 LSBs of a 16-bit output. (This would be 4.0 LSBs of an 18-bit DAC.)

For equivalent action to a Turbo PMAC servo algorithm, this gain can be set as:

$$Motor[x].Servo.Kp = Ixx30 * \frac{Ixx08}{2^{19} * Motor[x].PosSf}$$

Motor[*x*].Servo.Kp is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[*x*].Servo.Kvfb

Description: PID velocity feedback (“derivative”) gain term

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit per servo cycle) of actual velocity

Default: 40

Legacy I-variable alias: I*x*31

Motor[*x*].Servo.Kvfb is the servo loop’s main derivative gain term, providing a contribution subtracted from the control output proportional to the actual velocity feedback (this cycle’s actual position minus last cycle’s actual position, using the “**Pos2**” feedback source) of the motor. It acts effectively as an electronic damper. The higher **Motor[*x*].Servo.Kvfb** is, the heavier the damping effect is.

Motor[*x*].Servo.Kvfb is simply multiplied by the actual velocity to compute its contribution to the servo output, scaled as a 16-bit signed output. That is, an actual velocity of 1.0 motor units per servo cycle, when multiplied by a **Motor[*x*].Servo.Kvfb** of 1.0, contributes a value of 1.0 LSBs of a 16-bit output. (This would be 4.0 LSBs of an 18-bit DAC.)

Note that this contribution is subtracted from the *output* of the integral gain term. The contribution of the similar term **Motor[*x*].Servo.Kvifb** is subtracted from the *input* of the integral gain term. In general, subtracting the contribution from the output of the integrator provides superior trajectory-tracking performance; subtracting it from the input of the integrator provides better disturbance rejection.

If the motor is driving a properly tuned velocity-loop amplifier, the amplifier will provide sufficient damping, and **Motor[*x*].Servo.Kvfb** should be set to zero. If the motor is driving a current-loop (torque) amplifier, or if Power PMAC is commutating the motor, the amplifier will provide no damping, and **Motor[*x*].Servo.Kvfb** must be greater than zero to provide damping for stability.

If the servo update time is changed, whether through changing the frequency of the global servo interrupt or the motor’s own period with **Motor[*x*].Stime**, **Motor[*x*].Servo.Kvfb** must be changed proportionately in the opposite direction to keep the same damping effect. For instance, if the servo update time is cut in half, **Motor[*x*].Servo.Kvfb** must be doubled to keep the same effect.

For equivalent action to a Turbo PMAC servo algorithm with Ixx96 bit 1 (value 2) = 0, this gain can be set as:

$$Motor[x].Servo.Kvfb = Ixx31 * \frac{Ixx09 * Ixx30}{2^{26} * Motor[x].Pos2Sf}$$

If Turbo PMAC Ixx96 bit 1 = 1, the Power PMAC **Kvifb** gain should be used instead.

Motor[*x*].Servo.Kvfb is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[x].Servo.Kvff

Description: PID velocity feedforward gain term

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit per servo cycle) of command velocity

Default: 40

Legacy I-variable alias: I*x*32

Motor[*x*].Servo.Kvff adds an amount to the control output proportional to the desired velocity of the motor. It is intended to reduce tracking error due to the damping introduced by **Motor[*x*].Servo.Kvfb**, analog tachometer feedback, or physical damping effects.

Motor[*x*].Servo.Kvff is simply multiplied by the net desired velocity to compute its contribution to the servo output, scaled as a 16-bit signed output. That is, a desired velocity of 1.0 motor units per servo cycle, when multiplied by a **Motor[*x*].Servo.Kvff** of 1.0, contributes a value of 1.0 LSBs of a 16-bit output. (This would be 4.0 LSBs of an 18-bit DAC.)

Note that this contribution is added in to the *output* of the integral gain term. The contribution of the similar term **Motor[*x*].Servo.Kviff** is added to the *input* of the integral gain term. In general, adding the contribution to the output of the integrator provides superior trajectory-tracking performance; adding it to the input of the integrator provides better disturbance rejection.

If the motor is driving a current-loop (torque) amplifier, **Motor[*x*].Servo.Kvff** will usually be equal to (or slightly greater than) **Motor[*x*].Servo.Kvfb** to minimize tracking error. If the motor is driving a velocity-loop amplifier, **Motor[*x*].Servo.Kvff** will typically be substantially greater than **Motor[*x*].Servo.Kvfb** to minimize tracking error.

If the servo update time is changed, **Motor[*x*].Servo.Kvff** must be changed proportionately in the opposite direction to keep the same effect. For instance, if the servo update time is cut in half, **Motor[*x*].Servo.Kvff** must be doubled to keep the same effect.

For equivalent action to a Turbo PMAC servo algorithm with Ixx96 bit 1 (value 2) = 0, this gain can be set as:

$$Motor[x].Servo.Kvff = Ixx32 * \frac{Ixx09 * Ixx30}{2^{26} * Motor[x].Pos2Sf}$$

If Turbo PMAC Ixx96 bit 1 = 1, the Power PMAC **Kviff** gain should be used instead.

Motor[*x*].Servo.Kvff is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[x].Servo.Kvifb

Description: Servo velocity feedback (into integrator) gain term

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit per servo cycle) of actual velocity

Default: 0

Motor[*x*].Servo.Kvifb is the servo loop's secondary derivative gain term, providing a control output proportional to the actual velocity feedback (this cycle's actual position minus last cycle's actual position, using the "**Pos2**" feedback source) of the motor. It acts effectively as an electronic damper. The higher **Motor[*x*].Servo.Kvifb** is, the heavier the damping effect is.

Motor[*x*].Servo.Kvifb is simply multiplied by the actual velocity to compute its contribution to the servo output, scaled as a 16-bit signed output. That is, an actual velocity of 1.0 motor units per servo cycle, when multiplied by a **Motor[*x*].Servo.Kvifb** of 1.0, contributes a value of 1.0 LSBs of a 16-bit output. (This would be 4.0 LSBs of an 18-bit DAC.)

Note that this contribution is subtracted from the *input* of the integral gain term. The contribution of the similar term **Motor[*x*].Servo.Kvfb** is subtracted from the *output* of the integral gain term. In general, subtracting the contribution from the output of the integrator provides superior trajectory-tracking performance; subtracting it from the input of the integrator provides better disturbance rejection.

If the motor is driving a properly tuned velocity-loop amplifier, the amplifier will provide sufficient damping, and **Motor[*x*].Servo.Kvifb** should be set to zero. If the motor is driving a current-loop (torque) amplifier, or if Power PMAC is commutating the motor, the amplifier will provide no damping, and **Motor[*x*].Servo.Kvifb** must be greater than zero to provide damping for stability.

If the servo update time is changed, whether through changing the frequency of the global servo interrupt or the motor's own period with **Motor[*x*].Stime**, **Motor[*x*].Servo.Kvifb** must be changed proportionately in the opposite direction to keep the same damping effect. For instance, if the servo update time is cut in half, **Motor[*x*].Servo.Kvifb** must be doubled to keep the same effect.

For equivalent action to a Turbo PMAC servo algorithm with Ixx96 bit 1 (value 2) = 1, this gain can be set as:

$$Motor[x].Servo.Kvifb = Ixx31 * \frac{Ixx09 * Ixx30}{2^{26} * Motor[x].Pos2Sf}$$

If Turbo PMAC Ixx96 bit 1 = 0, the Power PMAC **Kvfb** gain should be used instead.

Motor[*x*].Servo.Kvifb is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[x].Servo.Kviff

Description: Servo velocity feedforward (into integrator) gain term

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit per servo cycle) of command velocity

Default: 0

Motor[*x*].Servo.Kviff adds an amount to the control output proportional to the desired velocity of the motor. It is intended to reduce tracking error due to the damping introduced by **Motor[*x*].Servo.Kvifb**, analog tachometer feedback, or physical damping effects.

Motor[*x*].Servo.Kviff is simply multiplied by the net desired velocity to compute its contribution to the servo output, scaled as a 16-bit signed output. That is, a desired velocity of 1.0 motor units per servo cycle, when multiplied by a **Motor[*x*].Servo.Kviff** of 1.0, contributes a value of 1.0 LSBs of a 16-bit output. (This would be 4.0 LSBs of an 18-bit DAC.)

Note that this contribution is added in to the in*put* of the integral gain term. The contribution of the similar term **Motor[*x*].Servo.Kvff** is added to the *output* of the integral gain term. In general, adding the contribution to the output of the integrator provides superior trajectory-tracking performance; adding it to the input of the integrator provides better disturbance rejection.

If the motor is driving a current-loop (torque) amplifier, **Motor[*x*].Servo.Kviff** will usually be equal to (or slightly greater than) **Motor[*x*].Servo.Kvifb** to minimize tracking error. If the motor is driving a velocity-loop amplifier, **Motor[*x*].Servo.Kviff** will typically be substantially greater than **Motor[*x*].Servo.Kvifb** to minimize tracking error.

If the servo update time is changed, **Motor[*x*].Servo.Kviff** must be changed proportionately in the opposite direction to keep the same effect. For instance, if the servo update time is cut in half, **Motor[*x*].Servo.Kviff** must be doubled to keep the same effect.

For equivalent action to a Turbo PMAC servo algorithm with Ixx96 bit 1 (value 2) = 1, this gain can be set as:

$$Motor[x].Servo.Kviff = Ixx32 * \frac{Ixx09 * Ixx30}{2^{26} * Motor[x].Pos2Sf}$$

If Turbo PMAC Ixx96 bit 1 = 0, the Power PMAC **Kvff** gain should be used instead.

Motor[*x*].Servo.Kviff is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[x].Servo.Kxig

Description: Cross-coupling integral gain

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit * servo cycle) of integrated error difference

Default: 0.0

Motor[*x*].Servo.Kxig is the integral gain term for the cross-coupling of gantry motors, providing a control effort for this motor proportional to the time integral of the difference in position errors between the leader and follower motors. It acts to drive this difference in error toward zero. Its contribution adds to the control effort of the servo terms based on the motor's own commanded and actual positions.

This gain term is only used if **Motor[*x*].Ctrl** for the lower-numbered of the two motors in the cross-coupled pair (typically the leader motor) is set to **Sys.GantryXCtrl**. The follower motor must be numbered one higher than the leader motor. **Motor[*x*].ExtraMotors** must be set to 1 for the leader motor. Each motor in the dual-motor algorithm has its own **Kxig** term that affects its own servo output; in general, these terms for the two motors will be very similar, if not identical.

Motor[*x*].Servo.Kxpg

Description: Cross-coupling proportional gain

Range: Floating-point

Units: LSBs of 16-bit output per motor unit of error difference

Default: 0.0

Motor[*x*].Servo.Kxpg is the proportional gain term for the cross-coupling of gantry motors, providing a control effort for this motor proportional to the difference in position errors between the leader and follower motors. It acts to drive this difference in error toward zero. Its contribution adds to the control effort of the servo terms based on the motor's own commanded and actual positions.

This gain term is only used if **Motor[*x*].Ctrl** for the lower-numbered of the two motors in the cross-coupled pair (typically the leader motor) is set to **Sys.GantryXCtrl**. The follower motor must be numbered one higher than the leader motor. **Motor[*x*].ExtraMotors** must be set to 1 for the leader motor. Each motor in the dual-motor algorithm has its own **Kxpg** term that affects its own servo output; in general, these terms for the two motors will be very similar, if not identical.

Motor[*x*].Servo.Kxvg

Description: Cross-coupling derivative gain

Range: Floating-point

Units: LSBs of 16-bit output per (motor unit per servo cycle) of error difference

Default: 0.0

Motor[*x*].Servo.Kxvg is the derivative gain term for the cross-coupling of gantry motors, providing a control effort for this motor proportional to the difference in the rate of change of position errors between the leader and follower motors. It acts to drive this difference in error toward zero. Its contribution adds to the control effort of the servo terms based on the motor's own commanded and actual positions.

This gain term is only used if **Motor[*x*].Ctrl** for the lower-numbered of the two motors in the cross-coupled pair (typically the leader motor) is set to **Sys.GantryXCtrl**. The follower motor must be numbered one higher than the leader motor. **Motor[*x*].ExtraMotors** must be set to 1 for the leader motor. Each motor in the dual-motor algorithm has its own **Kxvg** term that affects its own servo output; in general, these terms for the two motors will be very similar, if not identical.

Motor[*x*].Servo.MaxDR

Description: Adaptive servo closed-loop damping ratio at minimum plant gain

Range: Non-negative floating-point

Units: None

Default: 0.0 (disabled)

Motor[*x*].Servo.MaxDR specifies the closed-loop damping ratio (ζ) the adaptive servo algorithm should attempt to create for the motor when the plant gain estimated by the algorithm is at the maximum value (which means minimum inertia) the algorithm will compensate for (as set by **Motor[*x*].Servo.MinGainFactor**). However, if **Motor[*x*].Servo.MinDR** is set to its default value of 0.0, the adaptive algorithm does not try to adjust the closed-loop damping ratio (or natural frequency), instead trying to keep it constant at the nominal value.

If **Motor[*x*].Servo.MinDR** is set to a value greater than 0.0 to enable automatic adjustment of damping ratio, **Motor[*x*].Servo.MaxDR**, **Motor[*x*].Servo.MinW**, and **Motor[*x*].MaxW** should be set as well to define the span of change of natural frequency and damping ratio. These terms are normally set through the interactive window in the IDE's tuning control.

WARNING

If **Motor[*x*].Servo.MinDR** and **Motor[*x*].Servo.MinW** are set to values greater than 0.0, enabling gain-scheduled adaptive control, **Motor[*x*].Servo.MaxDR** must not be left at its default value of 0.0, which would specify no damping at minimum system inertia. This could lead to dangerous oscillations of the system.

When automatic damping-ratio adjustment is active, the adaptive servo algorithm sets its internal gains as a function of the estimated plant gain to vary the closed-loop damping ratio linearly between the value of **MaxW** at the maximum plant gain (minimum inertia) and the value of **MinW** at the minimum plant gain (maximum inertia). Automatic natural-frequency adjustment can provide a trade-off between the desire to keep performance constant as inertia changes and the physical limitations of the system.

Motor[*x*].Servo.MaxDR is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is not used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Motor[*x*].Servo.MaxGainFactor

Description: Maximum permitted relative gain

Range: Non-negative floating-point

Units: none (ratio)

Default: 1.0

Motor[*x*].Servo.MaxGainFactor specifies the maximum relative servo gain that the adaptive control algorithm will use, expressed as a ratio to the standard saved servo gain values. It will limit the change at the highest system inertia (lowest plant gain) expected to be seen. It is only used if the adaptive servo control algorithm is selected. For example, if it is set to 3.0, the resulting gains will not go above 3 times the standard saved gain values.

Typical values for **Motor[*x*].Servo.MaxGainFactor** are in the range of 1.0 to 10.0. This parameter is an important protection against possible spurious results in the adaptive control algorithm.

Motor[*x*].Servo.MaxGainFactor is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is not used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Motor[*x*].Servo.MaxInt

Description: Servo maximum integrated error magnitude

Range: Non-negative floating-point

Units: Integrator output units (LSBs of 16-bit output)

Default: 28,000

Legacy I-variable alias: I*x*63

Motor[*x*].Servo.MaxInt limits the magnitude of the integrated position error (the *output* of the integrator) for the PID servo algorithm, which can be useful for "anti-windup" protection, when the servo loop output saturates. The default value of **Motor[*x*].Servo.MaxInt** provides essentially no limitation. (The integral gain **Motor[*x*].Servo.Ki** controls how fast the error is integrated.)

A value of zero in **Motor[*x*].Servo.MaxInt** forces a zero output of the integrator, effectively disabling the integration function in the PID filter. This can be useful during periods when you are applying a constant force and are expecting a steady-state position error. (In contrast, setting **Motor[*x*].Servo.Ki** to 0 prevents further inputs to the integrator, but maintains the output.)

For equivalent action to a Turbo PMAC servo algorithm, this term can be set as:

$$Motor[x].Servo.MaxInt = Ixx63 * \frac{Ixx08 * Ixx30}{2^{23}}$$

Motor[*x*].Servo.MaxInt is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[*x*].Servo.MaxPosErr

Description: Servo-error input magnitude limit

Range: Non-negative floating-point

Units: Motor units

Default: 10,000

Legacy I-variable alias: I*x*67

Motor[*x*].Servo.MaxPosErr specifies the magnitude of the largest position error that will be allowed into the servo filter. This is intended to keep extreme conditions from upsetting the stability of the filter. However, if it is set too low, it can limit the response of the system to legitimate command trajectories or disturbances (this situation can particularly be noticed on very fine resolution systems).

If pure velocity control is desired for the motor, **Motor[*x*].Servo.MaxPosErr** can be set to 0, effectively disabling the position loop.

This parameter is not to be confused with **Motor[*x*].FatalFeLimit** or **Motor[*x*].WarnFeLimit**, the following error limit parameters. Those parameters take action outside the servo loop based on the real (before limiting) following error.

Motor[*x*].Servo.MaxPosErr is used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**.

Motor[x].Servo.MaxW

Description: Adaptive servo closed-loop natural frequency at maximum plant gain

Range: Non-negative floating-point

Units: Radians per second

Default: 0.0 (disabled)

Motor[*x*].Servo.MaxW specifies the closed-loop natural frequency (ω) the adaptive servo algorithm should attempt to create for the motor when the plant gain estimated by the algorithm is at the maximum value (which means minimum inertia) the algorithm will compensate for (as set by **Motor[*x*].Servo.MinGainFactor**). However, if **Motor[*x*].Servo.MinW** is set to its default value of 0.0, the adaptive algorithm does not try to adjust the closed-loop natural frequency (or damping ratio), instead trying to keep it constant at the nominal value.

If **Motor[*x*].Servo.MinW** is set to a value greater than 0.0 to enable automatic adjustment of natural frequency, **Motor[*x*].Servo.MaxW**, **Motor[*x*].Servo.MinDR**, and **Motor[*x*].MaxDR** should be set as well to define the span of change of natural frequency and damping ratio. These terms are normally set through the interactive window in the IDE's tuning control.

WARNING

If **Motor[*x*].Servo.MinDR** and **Motor[*x*].Servo.MinW** are set to values greater than 0.0, enabling gain-scheduled adaptive control, **Motor[*x*].Servo.MaxW** must not be left at its default value of 0.0, which would specify no control effort at minimum system inertia. This could lead to dangerous conditions in the system.

When automatic natural-frequency adjustment is active, the adaptive servo algorithm sets its internal gains as a function of the estimated plant gain to vary the closed-loop natural frequency linearly between the value of **MaxW** at the maximum plant gain (minimum inertia) and the value of **MinW** at the minimum plant gain (maximum inertia). Automatic natural-frequency adjustment can provide a trade-off between the desire to keep performance constant as inertia changes and the physical limitations of the system.

Note

The units of **Motor[*x*].Servo.MaxW** are radians per second. A frequency expressed in radians per second has a value that is a factor of 2π greater than the same frequency expressed in Hertz (cycles per second).

Motor[*x*].Servo.MaxW is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Note

The natural frequency adjustment algorithms use the value of **Sys.ServoPeriod** to convert time units between seconds and servo cycles. The value of **Sys.ServoPeriod** must therefore be correct for accurate adjustment calculations.

Motor[*x*].Servo.MinDR

Description: Adaptive servo closed-loop damping ratio at minimum plant gain

Range: Non-negative floating-point

Units: None

Default: 0.0 (disabled)

Motor[*x*].Servo.MinDR specifies the closed-loop damping ratio (ζ) the adaptive servo algorithm should attempt to create for the motor when the plant gain estimated by the algorithm is at the minimum value (which means maximum inertia) the algorithm will compensate for (as set by **Motor[*x*].Servo.MaxGainFactor**). However, if it is set to its default value of 0.0, the adaptive algorithm does not try to adjust the closed-loop damping ratio (or natural frequency), instead trying to keep it constant at the nominal value.

If **Motor[*x*].Servo.MinDR** is set to a value greater than 0.0 to enable automatic adjustment of damping ratio, **Motor[*x*].Servo.MaxDR**, **Motor[*x*].Servo.MinW**, and **Motor[*x*].MaxW** should be set as well to define the span of change of natural frequency and damping ratio. These terms are normally set through the interactive window in the IDE's tuning control.

When automatic damping-ratio adjustment is active, the adaptive servo algorithm sets its internal gains as a function of the estimated plant gain to vary the closed-loop damping ratio linearly between the value of **MaxW** at the maximum plant gain (minimum inertia) and the value of **MinW** at the minimum plant gain (maximum inertia).

Motor[*x*].Servo.MinDR is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Motor[*x*].Servo.MinGainFactor

Description: Minimum permitted relative gain

Range: Non-negative floating-point

Units: none (ratio)

Default: 1.0

Motor[*x*].Servo.MinGainFactor specifies the minimum relative servo gain that the adaptive control algorithm will use, expressed as a ratio to the standard saved servo gain values. It will limit the change at the lowest system inertia (highest plant gain) expected to be seen. It is only used if the adaptive servo control algorithm is selected. For example, if it is set to 0.25, the resulting gains will not go below ¼ of the standard saved gain values.

Typical values for **Motor[*x*].Servo.MinGainFactor** are in the range of 0.1 to 1.0. This parameter is an important protection against possible spurious results in the adaptive control algorithm.

Motor[*x*].Servo.MinGainFactor is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Motor[*x*].Servo.MinW

Description: Adaptive servo closed-loop natural frequency at minimum plant gain

Range: Non-negative floating-point

Units: Radians per second

Default: 0.0 (disabled)

Motor[*x*].Servo.MinW specifies the closed-loop natural frequency (ω) the adaptive servo algorithm should attempt to create for the motor when the plant gain estimated by the algorithm is at the minimum value (which means maximum inertia) the algorithm will compensate for (as set by **Motor[*x*].Servo.MaxGainFactor**). However, if it is set to its default value of 0.0, the adaptive algorithm does not try to adjust the closed-loop natural frequency (or damping ratio), instead trying to keep it constant at the nominal value.

If **Motor[*x*].Servo.MinW** is set to a value greater than 0.0 to enable automatic adjustment of natural frequency, **Motor[*x*].Servo.MaxW**, **Motor[*x*].Servo.MinDR**, and **Motor[*x*].MaxDR** should be set as well to define the span of change of natural frequency and damping ratio. These terms are normally set through the interactive window in the IDE's tuning control.

When automatic natural-frequency adjustment is active, the adaptive servo algorithm sets its internal gains as a function of the estimated plant gain to vary the closed-loop natural frequency linearly between the value of **MaxW** at the maximum plant gain (minimum inertia) and the value of **MinW** at the minimum plant gain (maximum inertia). Automatic natural-frequency adjustment can provide a trade-off between the desire to keep performance constant as inertia changes and the physical limitations of the system.

Note

The units of **Motor[*x*].Servo.MinW** are radians per second. A frequency expressed in radians per second has a value that is a factor of 2π greater than the same frequency expressed in Hertz (cycles per second).

Motor[*x*].Servo.MinW is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Note

The natural frequency adjustment algorithms use the value of **Sys.ServoPeriod** to convert time units between seconds and servo cycles. The value of **Sys.ServoPeriod** must therefore be correct for accurate adjustment calculations.

Motor[*x*].Servo.NominalGain

Description: Plant reference gain

Range: Non-negative floating-point

Units: (Motor units per second^2) / 16-bit command unit

Default: 0.0

Motor[*x*].Servo.NominalGain specifies the reference "gain" of the physical plant for the motor to servo loop output commands in the adaptive servo control algorithm. It is only used if the adaptive servo control algorithm is selected. This gain is expressed as the resulting acceleration of the motor in motor units per second^2 divided by the servo-loop output command in 16-bit units (+/-32,768). This value not only includes motor and load inertia gain, but servo-command signal gain, amplifier gain, and feedback sensor and scaling gain.

Of the components of this overall gain, generally only the inertia value will vary (the inertia "gain" being inversely proportional to the inertia). It does not matter what specific inertia value within the range of possible inertias is used, but limiting terms **Motor[*x*].Servo.MinGainFactor** and **Motor[*x*].Servo.MaxGainFactor** must be set relative to the value chosen for this, and the standard saved servo-loop gain terms should be set for the desired performance at this plant gain value. The effective servo-loop gains used will be based on the ratio between this value and the latest estimated plant gain in **Motor[*x*].Servo.EstGain**.

While the value for **Motor[*x*].Servo.NominalGain** can be determined analytically, it is usually easier and more accurate to determine it experimentally. The "position loop autotune" feature of the IDE's servo tuning control will report the value it calculates for the plant based on its excitation as "*Estimated Gain*". This reported value can be used directly for **Motor[*x*].Servo.NominalGain**.

Motor[*x*].Servo.NominalGain is used when **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**. It is not used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.PidCtrl**, or **Sys.PosCtrl**.

Motor[x].Servo.OutDbOff

Description: Servo output-deadband-off error size

Range: Non-negative floating-point

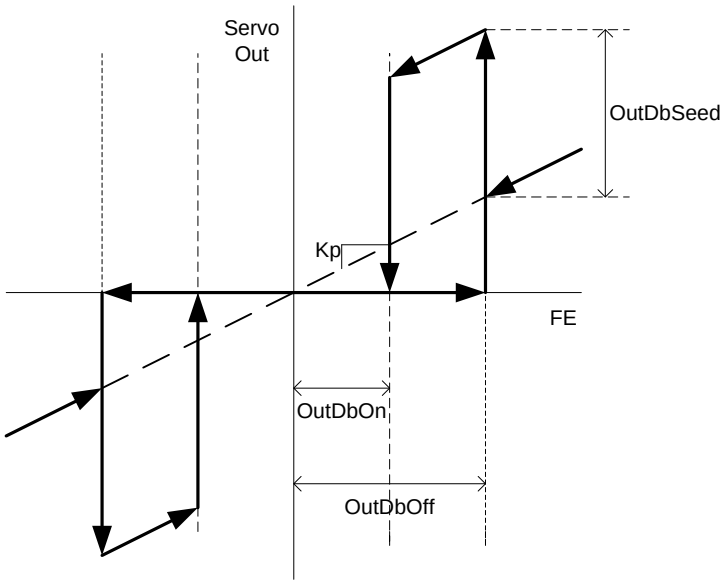
Units: Motor units

Default: 0.0

Motor[*x*].Servo.OutDbOff defines the size of the position error band, measured from zero error in either direction, where the output deadband will be de-activated as the magnitude of the position error becomes greater than this size. Outside this band, the servo output will generally be non-zero. If **OutDbOff** is set to its default value of 0.0, the output deadband function is not active.

The output deadband has hysteresis, so once outside this band, the magnitude of the position error must subsequently become less than (the usually smaller) **Motor[*x*].Servo.OutDbOn** to create an output deadband (zero servo output) again.

The following diagram shows the relationship and effect of these elements:



Motor[*x*].Servo.OutDbOff is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

Motor[x].Servo.OutDbOn

Description: Servo output-deadband-on error size

Range: Non-negative floating-point

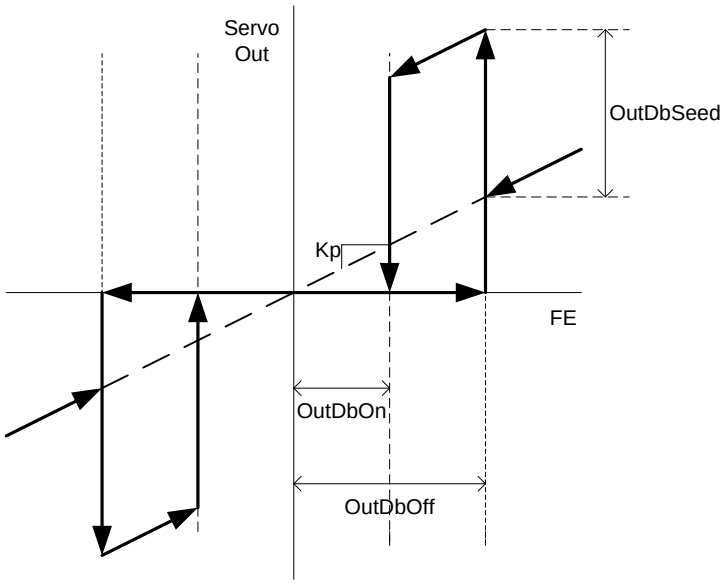
Units: Motor units

Default: 0.0

Motor[*x*].Servo.OutDbOn defines the size of the position error band, measured from zero error in either direction, where the output deadband will be activated as the magnitude of the position error falls beneath this size when the commanded velocity of the motor is zero (**Motor[*x*].DesVelZero** = 1). Within this band, the servo output will be exactly zero.

The output deadband has hysteresis, so once within this band, the magnitude of the position error must subsequently become greater than (the usually larger) **Motor[*x*].Servo.OutDbOff** to enable a non-zero servo output again.

The following diagram shows the relationship and effect of these elements:



Motor[*x*].Servo.OutDbOn is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

Motor[*x*].Servo.OutDbSeed

Description: Servo output-deadband integrator seed

Range: Non-negative floating-point

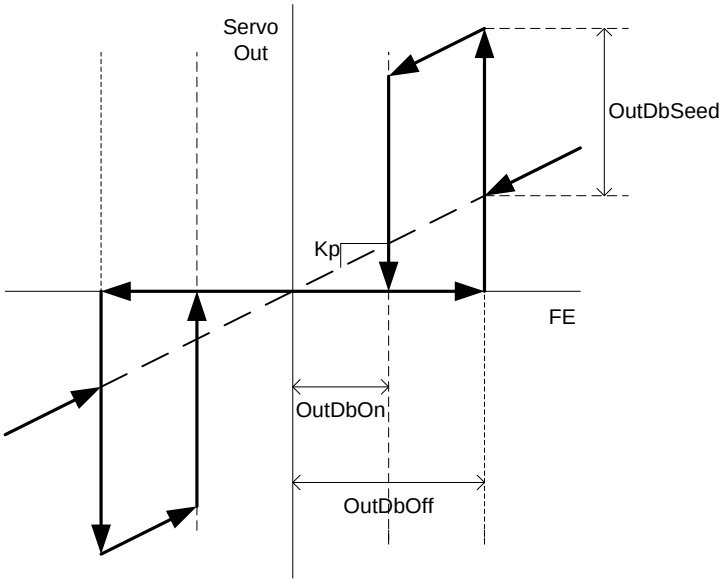
Units: LSBs of 16-bit output

Default: 0.0

Motor[*x*].Servo.OutDbSeed defines the magnitude of the integrator "seed" used when the position error exceeds the **Motor[*x*].Servo.OutDbOff** threshold and servo control is re-activated.

That is, a value of this magnitude (and of the appropriate sign) is added into the integrator output register at this time, overwriting whatever value happened to be there. This value can be used to improve the settling to final position.

The following diagram shows the relationship and effect of these elements:



Motor[*x*].Servo.OutDbSeed is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

Motor[x].Servo.pAccOut

Description: Servo acceleration/torque feedforward auxiliary output pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0 (disabled)

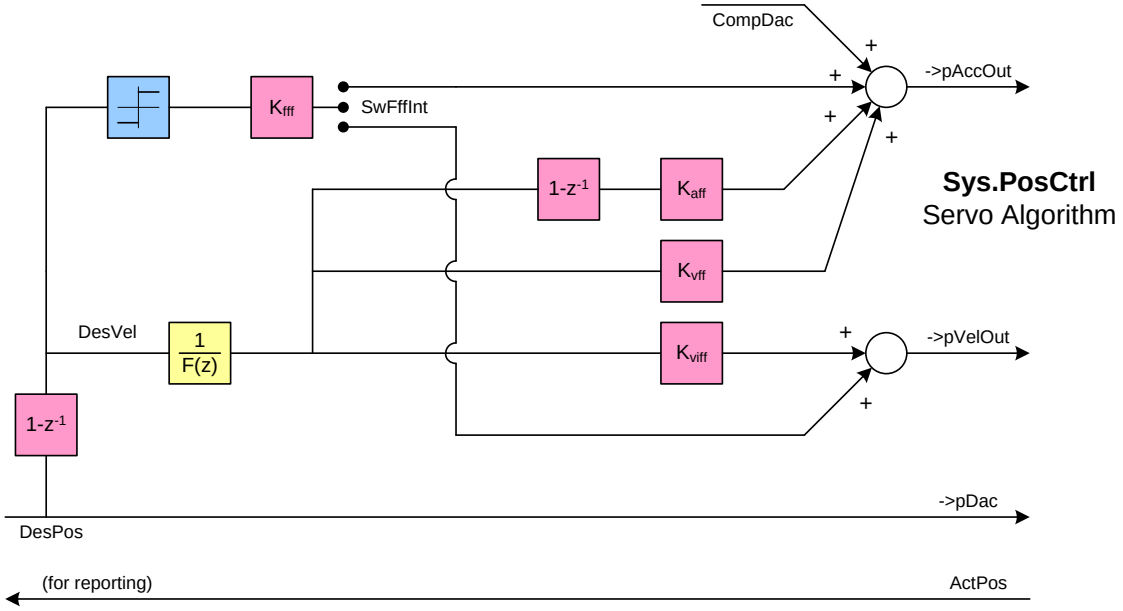
Motor[*x*].Servo.pAccOut specifies which register the motor uses for an auxiliary acceleration/torque feedforward output from its servo algorithm. It contains the address of this register. If **Motor[*x*].Servo.pAccOut** is set to its default value of 0, this motor will not have any auxiliary acceleration/torque feedforward output command value.

The acceleration/torque feedforward command written to the **pAccOut** register will contain the sum of several terms: the commanded acceleration multiplied by the acceleration feedforward gain (**Kaff**); the commanded velocity multiplied by a velocity feedforward gain (**Kvff**), if **Motor[*x*].Servo.SwFffInt** is set to its default value of 0, the sign of the commanded velocity multiplied by the friction feedforward gain (**Kfff**), and the torque offset value in **Motor[*x*].CompDac** (usually from a torque compensation table).

The commanded velocity value can be filtered through the "F(z)" 2nd-order filter (with elements **Motor[*x*].Servo.Kf1** and **Kf2**) before the feedforward values are computed. This can provide smoothing when the commanded trajectory is slaved to a measured signal, as in position following or external time base.

When using the **Sys.PosCtrl** servo algorithm, which outputs the commanded position value to the register specified by **Motor[*x*].pDac**, the use of **Motor[*x*].pAccOut** does not affect the value written to the main **pDac** register.

The following block diagram shows how the auxiliary outputs work in the **Sys.PosCtrl** algorithm:



In other servo algorithms that perform real servo loop closure, the use of **Motor[*x*].Servo.pAccOut** causes the terms listed above to be used for the value written to the **pAccOut** register *instead of* for the value written to the main **pDac** register. (The separate **pAccOut** register cannot be used in the cross-coupled **Sys.GantryXCtrl** servo algorithm.)

The main use of the separate **pAccOut** register is for networked servo drives operating in cyclic-position or cyclic-velocity mode, but which support an additional cyclic "torque offset" command value. This auxiliary value is supported by many EtherCAT servo drives operating under the DS402 standard. In this case, **Motor[*x*].Servo.pAccOut** is set to **ECAT[*i*].IO[*j*].Data.a**, with ***i*** and ***j*** selected to map to the "offset torque ($60b2_h$)" register in the drive.

The auxiliary is also supported by some servo drives with MACRO interfaces operating in cyclic velocity mode. In this case, **Motor[*x*].Servo.pAccOut** is set to something like **Gate1[*i*].Macro[*j*][1].a** or **Gate3[*i*].MacroOutA[*j*][1].a**.

Motor[*x*].Servo.pAccOut is new in version 2.0 firmware, released 1st quarter 2015.

Motor[x].Servo.pVelOut

Description: Servo velocity feedforward auxiliary output pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: 0 (disabled).

Motor[*x*].Servo.pVelOut specifies which register the motor uses for an auxiliary velocity feedforward output from its **Sys.PosCtrl** position-output servo algorithm. It contains the address of this register. If **Motor[*x*].Servo.pVelOut** is set to its default value of 0, this motor will not have any auxiliary velocity feedforward output command value.

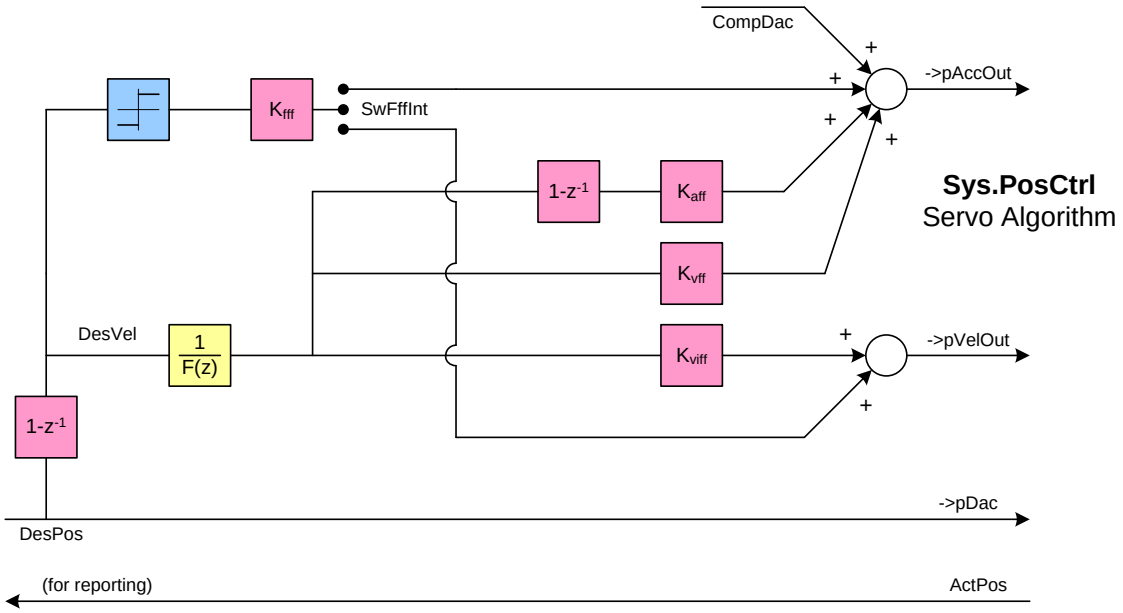
The velocity feedforward command written to the **pVelOut** register will contain the sum of two terms: the commanded velocity multiplied by a velocity feedforward gain (**Kviff**), if **Motor[*x*].Servo.SwFffInt** is set to 1, the sign of the commanded velocity multiplied by the friction feedforward gain (**Kfff**).

The commanded velocity value can be filtered through the "F(z)" 2nd-order filter (with elements **Motor[*x*].Servo.Kf1** and **Kf2**) before the feedforward values are computed. This can provide smoothing when the commanded trajectory is slaved to a measured signal, as in position following or external time base.

The main use of the separate **pVelOut** register is for networked servo drives operating in cyclic-position mode, but which support an additional cyclic "velocity offset" command value. This auxiliary value is supported by many EtherCAT servo drives operating under the DS402 standard. In this case, **Motor[*x*].pVelOut** is set to **ECAT[*i*].IO[*j*].Data.a**, with ***i*** and ***j*** selected to map to the "offset velocity ($60b1_h$)" register in the drive.

Motor[*x*].Servo.pVelOut is only used if **Motor[*x*].Ctrl** is set to **Sys.PosCtrl** to select the position-output servo algorithm, which primarily just processes the desired position to a position output format.

The following block diagram shows how the auxiliary outputs work in the **Sys.PosCtrl** algorithm:



Motor[*x*].Servo.pVelOut is new in version 2.0 firmware, released 1st quarter 2015.

Motor[*x*].Servo.SwFffInt

Description: Software switch to select friction feedforward command destination

Range: 0 .. 1

Units: Boolean

Default: 0

The **Motor[*x*].Servo.SwFffInt** (**Sw**itch **F**riction **f**eed**f**orward **Int**egrator) term selects the destination of the output of the servo loop's "friction feedforward" term. If **Servo.SwFffInt** is set to the default value of 0, the friction feedforward command is added directly to the servo-loop output, after the integrator. If **Servo.SwFffInt** is set to 1, it is added to the input of the servo loop's integrator.

When the friction feedforward, velocity feedforward, and velocity feedback terms are added directly to the servo output, the integrator is effectively in the position loop, which is generally better for tracking command trajectories. When the friction feedforward, velocity feedforward, and velocity feedback terms are added to the input of the integrator, the integrator is effectively in the velocity loop, which is generally better for rejecting disturbances.

Motor[*x*].Servo.SwFffInt is used when **Motor[*x*].Ctrl** is set to **Sys.PosCtrl**, **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**.

Motor[x].Servo.SwPoly7

Description: Software switch to select 2nd or 7th-order polynomials

Range: 0 .. 1

Units: Boolean

Default: 0

The **Motor[*x*].Servo.SwPoly7** (**Sw**itch **Poly**nomial **7**th order) term selects whether the polynomial blocks "A", "B", "C", and "D" of the standard servo algorithm are 2nd-order or 7th-order expressions. If it is set to the default value of 0, they are executed as 2nd-order expressions and only the first two terms are used (even if the others are set to non-zero values). If it set to 1, they are executed as 7th-order expressions.

Generally, **Motor[*x*].Servo.SwPoly7** is only set to 1 for motors with difficult dynamics (e.g. multiple resonances) that require high-order polynomial calculations to control well. In most systems, this variable is better left at the default value of 0 to save calculation time.

Motor[*x*].Servo.SwPoly7 is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

Motor[x].Servo.SwZvInt

Description: Software switch to select integrator mode

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: I*x*34

The **Motor[*x*].Servo.SwZvInt** (**Sw**itch **Z**ero-**v**elocity **Int**egration mode) term selects whether the position-error integration term (**Ki**) is on all the time, or only when the desired velocity for the motor is equal to zero. If it is set to 0, position-error integration is performed all the time

If it is set to 1, position-error integration is performed only when the the desired velocity is exactly equal to 0. When the desired velocity is not exactly equal to zero (i.e. during a commanded move), the *input* to the integrator is turned off, which means the *output* control effort from the integrator is kept constant during this period (but is generally not zero). This same action takes place whenever the total control output saturates at the **Motor.MaxDac** value.

Motor[*x*].Servo.SwZvInt is usually set initially using the Tuning utility in the Integrated Development Environment (IDE) program. When performing the interactive feedforward tuning portion of that utility, it is important to set this to 1 so the dynamic behavior of the system may be observed without integrator action.

Motor[*x*].Servo.SwZvInt is used when **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, **Sys.LegacyCtrl**, **Sys.GantryXCtrl**, or **Sys.AdaptiveCtrl**. It is *not* used when **Motor[*x*].Ctrl** is set to **Sys.PidCtrl** or **Sys.PosCtrl**.

MuxIo. Saved Data Structure Elements

The **MuxIo.** data structure comprises the multiplexed I/O ("thumbwheel port") settings for the Power PMAC. The elements in this structure provide access to multiplexed I/O devices which use the serial thumbwheel-multiplexer protocol, such as ACC-34A and ACC-34AA.

MuxIo.ClockPeriod

Description: Serial multiplexed data clock period

Range: 0 … 5,000

Units: Microseconds

Default: 0

MuxIo.ClockPeriod specifies the period between consecutive serial clock pulses that transfer data into or out of the multiplexed I/O device.

If **MuxIo.ClockPeriod** is left at its default value of 0, the data will be clocked in/out as fast as possible by background tasks in Power PMAC. For example, on a 1 GHz Power PMAC CPU, the clock frequency can be as fast as 1.25MHz.

Increasing the **MuxIo.ClockPeriod** will inversely reduce the clock frequency, which may be necessary for slower multiplexed devices or longer transmit lines.

Note that the **MuxIo.ClockPeriod** setting can affect the update period at which all enabled ports' data can be clocked in/out. If **MuxIo.UpdatePeriod** is set to a number greater than 0, allowing cyclic update of input/output data image words, but smaller than the minimum amount of time required for one full scan of all enabled ports, **MuxIo.UpdatePeriod** will be automatically recalculated and set based upon the **MuxIo.ClockPeriod** according to the following equation:

*MuxIo.UpdatePeriod = Number of Enabled Ports * 40 * MuxIo.ClockPeriod/1000 *1.5*

MuxIo.Enable

Description: Multiplexed I/O background task enable

Range: 0 .. 1

Units: Boolean

Default: 0

MuxIo.Enable controls whether the background multiplexed I/O task is activated or not. If it is set to 0, serial scanning of multiplexed I/O is not enabled.

If **MuxIo.Enable** is set to 1 and its previous value was 0, serial scan of multiplexed I/O task is enabled. At this moment, the existing multiplexed port address values **MuxIo.pOut** and

MuxIo.pIn are read and used until **MuxIo.Enable** is set to 0. Since in a typical application these values are not changed once setup, this should not be of any concern to users.

If **MuxIo.UpdatePeriod** is set to a value greater than 0, setting **MuxIo.Enable** to 1 will start the cyclic scan of all enabled ports.

If **MuxIo.UpdatePeriod** is set to a value of 0, setting **MuxIo.Enable** to 1 will only start the multiplexed I/O background task which monitors the **MuxIo.PortA[*n*].Enable** and **MuxIo.PortB[*n*].Enable** flags every 50 milliseconds where ***n*** is the address of multiplexed I/O device selected by DIP switchs on the device.

MuxIo.InBit

Description: Bit number of input signal in **pIn** register.

Range: 0 .. 31

Units: Bit number (little-endian)

Default: 0

MuxIo.InBit determines which bit of the 32-bit register whose address is specified by **MuxIo.pIn** Power PMAC will use as input signal for multiplexed I/O port.

For the JTHW port available on the ACC-5E, where the DAT0 line is usually used as the input signal for multiplexed I/O, **MuxIo.InBit** should be set to 8, as the DAT0 bit is located at bit 8 of the **Gate2[*i*].MuxData** register. If the alternate DAT7 line is used for the input signal, **MuxIo.InBit** should be set to 15 instead.

For the JIO port available on the ACC-5E3, where the GPIO00 line is usually used as the input signal for multiplexed I/O, **MuxIo.InBit** should be set to 0, as the GPIO00 bit is located at bit 0 of the **Gate3[*i*].GpioData[0]** register. If the alternate GPIO07 line is used for the input signal, **MuxIo.InBit** should be set to 7 instead.

For the JTHW port available on the Power Clipper, where the DAT0 line is usually used as the input signal for multiplexed I/O, **MuxIo.InBit** should be set to 0, as the DAT0 bit is located at bit 0 of the **Gate3[*i*].GpioData[0]** register. If the alternate GPIO07 line is used for the input signal, **MuxIo.InBit** should be set to 7 instead.

MuxIo.OutBit

Description: Bit number of the first output signal in **pOut** register.

Range: 0 .. 24

Units: Bit number (little-endian)

Default: 0

MuxIo.OutBit determines which bit of the 32-bit register whose address is specified by **MuxIo.pOut** Power PMAC will use as the first of eight consective output signals for multiplexed I/O port.

For the JTHW port available on the ACC-5E, where the SEL0 line is usually used as the first output signal for multiplexed I/O, **MuxIo.OutBit** should be set to 16, as the SEL0 bit is located at bit 16 of the **Gate2[*i*].MuxData** register.

For the JIO port available on the ACC-5E3, where the GPIO08 line is usually used as the output signal for multiplexed I/O, **MuxIo.OutBit** should be set to 8, as the GPIO08 bit is located at bit 8 of the **Gate3[*i*].GpioData[0]** register.

For the JTHW port available on the Power Clipper, where the SEL0 line is usually used as the output signal for multiplexed I/O, **MuxIo.OutBit** should be set to 8, as the SEL0 bit is located at bit 8 of the **Gate3[*i*].GpioData[0]** register.

MuxIo.pIn

Description: Multiplexed port input register pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Sys.pushm

MuxIo.pIn specifies which register the multiplexed I/O background task looks to for its input bit. It contains the address of this register.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

In the specified register, Power PMAC uses the bit whose number is contained in **MuxIo.InBit** to read the input signal on multiplexed port.

For the JTHW port available on the ACC-5E, where the DAT0 line is usually used as input signal for multiplexed I/O, **MuxIo.pIn** should be set to **Gate2[*i*].MuxData.a**. The DAT0 signal is in bit 8 of this register.

For the JIO port available on the ACC-5E3, where the GPIO00 line is usually used as input signal for multiplexed I/O, **MuxIo.pIn** should be set to **Gate3[*i*].GpioData[0].a**. The GPIO00 signal is in bit 0 of this register.

For the JTHW port available on the Power Clipper, where the DAT0 line is usually used as input signal for multiplexed I/O, **MuxIo.pIn** should be set to **Gate3[*i*].GpioData[0].a**. The DAT0 signal is in bit 0 of this register.

MuxIo.pOut

Description: Multiplexed port output register pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Sys.pushm

MuxIo.pOut specifies which register the multiplexed I/O background task uses for writing its output bits. It contains the address of this register.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

In the specified register, Power PMAC uses 8 consecetive bits whose first bit number is contained in **MuxIo.OutBit** to write the output signals on multiplexed port.

For the JTHW port available on the ACC-5E, where SEL0 to SEL7 lines are usually used as output signals for multiplexed I/O, **MuxIo.pOut** should be set to **Gate2[*i*].MuxData.a**. The SEL0 signal is in bit 16 of this register.

For the JIO port available on the ACC-5E3, where GPIO08 to GPIO15 lines are usually used as output signals for multiplexed I/O, **MuxIo.pOut** should be set to **Gate3[*i*].GpioData[0].a**. The SEL0 signal is in bit 8 of this register.

For the JTHW port available on the Power Clipper, where SEL0 to SEL7 lines are usually used as output signals for multiplexed I/O, **MuxIo.pOut** should be set to **Gate3[*i*].GpioData[0].a**. The SEL0 signal is in bit 8 of this register.

MuxIo.UpdatePeriod

Description: Multiplexed I/O cyclic update period

Range: Positive integer

Units: Milliseconds

Default: 0

MuxIo.UpdatePeriod tells Power PMAC software how much time there is between scans of all enabled ports on all multiplexed I/O addresses. It is expressed in milliseconds as an integer value; numerically it is the reciprocal of the multiplexed I/O update frequency in kilohertz (kHz).

If **MuxIo.UpdatePeriod** is set to the default value of 0, writing or reading data to the multiplexed addresses are performed on-demand basis. In this case, every 50 milliseconds, all ports on all addresses are checked and if any of the **MuxIo.Port*α*[*n*].Enable** (where ***α*** is **A** or **B**) flags is set to 1, then the **MuxIo.Port*α*[*n*].Data** value is read from or written to, depending on the **MuxIo.Port*α*[*n*].Dir** setting, where ***n*** is the address of the multiplexed I/O device on the port,

selected by DIP switches on the device. After the data transfer is completed, the **MuxIo.Port*α*[*n*].Enable** flag is automatically cleared (set to 0).

Since the serial multiplexed port data transfer is performed as a background task in Power PMAC firmware and it is asynchronous to other tasks such as PLCs or motion programs, users must explicitly check for the **MuxIo.Port*α*[*n*].Enable** flags to be cleared after setting them to 1 in any PLC or motion program. In this approach, in order to keep the multiplexed devices from shutting down in a watchdog state due to inactivity, it is necessary for user to keep updating the **MuxIo.Port*α*[*n*].Enable** flags by setting them to 1, even if I/O transfers are not actually required.

If **MuxIo.UpdatePeriod** is set to an interger value greater than 0, the Power PMAC will perform a cyclic scan of all enabled ports on all multiplexed addresses. In this case, all ports on all addresses are checked, and if any of the **MuxIo.Port*α*[*n*].Enable** flags are set to 1, then the **MuxIo.Port*α*[*n*].Data** value is read from or written to depending on **MuxIo.Port*α*[*n*].Dir** setting.

Unlike the case where **MuxIo.UpdatePeriod** was set to 0, in this case the **MuxIo.Port*α*[*n*].Enable** flags are not cleared after each data transfer and the same tasks will be repeated after **MuxIo.UpdatePeriod** milliseconds. In this approach, the data image words **MuxIo.Port*α*[*n*].Data** are updated periodically as long as **MuxIo.Port*α*[*n*].Enable** is set to a value of 1.

MuxIo.PortA[*n*].AutoParityCheck

Description: Automatic check for parity match for input data on Port A at address *n*

Range: 0 .. 1

Units: Boolean

Default: 0

MuxIo.PortA[*n*].AutoParityCheck specifies whether parity check is performed on input words or not. The index ***n*** (= 0 to 31) is the address of multiplexed I/O device as selected by DIP switches on the device.

If **MuxIo.PortA[*n*].AutoParityCheck** is set to the default value of 0, the input word is always stored in **MuxIo.PortA[*n*].Data** as received. Although parity for each received input word is stored in **MuxIo.PortA[*n*].Parity** and result of its comparison with calculated parity word is stored in **MuxIo.PortA[*n*].ParityStatus**, no automatic action, such as rejection of the data, is taken. This method is useful if the user decides to take some other action other than rejecting the data and keeping the previous value in **MuxIo.PortA[*n*].Data**.

If **MuxIo.PortA[*n*].AutoParityCheck** is set to a value of 1, the input word is stored in **MuxIo.PortA[*n*].Data** only if the received parity value, which is stored in **MuxIo.PortA[*n*].Parity**, matches with calculated parity value for the received input word. The result of this comparison is stored in **MuxIo.PortA[*n*].ParityStatus**. In case of a mismatch, the received input word is rejected and previous value of **MuxIo.PortA[*n*].Data** is kept.

MuxIo.PortB[*n*].AutoParityCheck

Description: Automatic check for parity match for input data on Port B at address *n*

Range: 0 .. 1

Units: Boolean

Default: 0

MuxIo.PortB[*n*].AutoParityCheck specifies whether parity check is performed on input words or not. The index ***n*** (= 0 to 31) is the address of multiplexed I/O device as selected by DIP switches on the device.

If **MuxIo.PortB[*n*].AutoParityCheck** is set to the default value of 0, the input word is always stored in **MuxIo.PortB[*n*].Data** as received. Although parity for each received input word is stored in **MuxIo.PortB[*n*].Parity** and result of its comparison with calculated parity word is stored in **MuxIo.PortB[*n*].ParityStatus**, no automatic action, such as rejection of the data, is taken. This method is useful if the user decides to take some other action other than rejecting the data and keeping the previous value in **MuxIo.PortB[*n*].Data**.

If **MuxIo.PortB[*n*].AutoParityCheck** is set to a value of 1, the input word is stored in **MuxIo.PortB[*n*].Data** only if the received parity value, which is stored in **MuxIo.PortB[*n*].Parity**, matches with calculated parity value for the received input word. The result of this comparison is stored in **MuxIo.PortB[*n*].ParityStatus**. In case of a mismatch, the received input word is rejected and previous value of **MuxIo.PortB[*n*].Data** is kept.

MuxIo.PortA[*n*].Dir

Description: Direction of multiplexed port A

Range: 0 .. 1

Units: Bit field

Default: 0

MuxIo.PortA[*n*].Dir determines whether the port is used as an input or output port. The index ***n*** (= 0 to 31) is the address of multiplexed I/O device as selected by DIP switches on the device.

If **MuxIo.PortA[*n*].Dir** is set to default value of 0, the corresponding port acts as an input port, so the data is read from multiplexed I/O device and copied into **MuxIo.PortA[*n*].Data** image word in the Power PMAC.

If **MuxIo.PortA[*n*].Dir** is set to a value of 1, the corresponding port acts as an output port, so the data is read from **MuxIo.PortA[*n*].Data** image word in the Power PMAC and copied into multiplexed I/O device.

For ACC-34A and ACC-34AA, where port A is for inputs and port B if for outputs, **MuxIo.PortA[*n*].Dir** is set to a value of 0 and **MuxIo.PortB[*n*].Dir** is set to a value of 1.

MuxIo.PortB[*n*].Dir

Description: Direction of multiplexed port B

Range: 0 .. 1

Units: Bit field

Default: 0

MuxIo.PortB[*n*].Dir determines whether the port is used as an input or output port. The index ***n*** (= 0 to 31) is the address of multiplexed I/O device as selected by DIP switches on the device.

If **MuxIo.PortB[*n*].Dir** is set to default value of 0, the corresponding port acts as an input port, so the data is read from multiplexed I/O device and copied into **MuxIo.PortB[*n*].Data** image word in the Power PMAC.

If **MuxIo.PortB[*n*].Dir** is set to a value of 1, the corresponding port acts as an output port, so the data is read from **MuxIo.PortB[*n*].Data** image word in the Power PMAC and copied into multiplexed I/O device.

For ACC-34A and ACC-34AA, where port A is for inputs and port B if for outputs, **MuxIo.PortA[*n*].Dir** is set to a value of 0 and **MuxIo.PortB[*n*].Dir** is set to a value of 1.

MuxIo.PortA[*n*].Enable

Description: Data transfer for address *n* multiplexed port A enabled

Range: 0 .. 1

Units: Boolean

Default: 0

MuxIo.PortA[*n*].Enable determines whether data transfer for multiplexed port A on the device with address ***n*** is perfomed by Power PMAC firmware or not. The index ***n*** (= 0 to 31) is the address of multiplexed I/O device as selected by DIP switches on the device.

If **MuxIo.PortA[*n*].Enable** is set to a value of 1, the **MuxIo.PortA[*n*].Data** value is read from or written to, depending on the **MuxIo.PortA[*n*].Dir** setting.

Depending on the **MuxIo.UpdatePeriod** setting, the **MuxIo.PortA[*n*].Enable** flag is either cleared automatically by Power PMAC firmware after each data transfer or left unmodified for next data transfer cycle.

If **MuxIo.UpdatePeriod** is set to the default value of 0, writing or reading data to the multiplexed addresses are performed on-demand basis. In this case, every 50 milliseconds, all ports on all addresses are checked and if any of the **MuxIo.PortA[*n*].Enable** flags are set to 1, then the **MuxIo.PortA[*n*].Data** value is read from or written to depending on the **MuxIo.PortA[*n*].Dir**

setting. After the data transfer is completed, the **MuxIo.PortA[*n*].Enable** flag is automatically cleared (set to 0).

Since the serial multiplexed port data transfer is performed as a background task in Power PMAC firmware and it is asynchronous to other tasks such as PLCs or motion programs, users must explicitly check for the **MuxIo.PortA[*n*].Enable** flags to be cleared after setting them high in any PLC or motion program.

In this approach, in order to keep the multiplexed devices from shutting down in a watchdog state due to inactivity, it is necessary for the user to keep updating the **MuxIo.PortA[*n*].Enable** flags by setting them to 1, even if data transfer is not required.

If **MuxIo.UpdatePeriod** is set greater than 0, the Power PMAC will perform a cyclic scan of all enabled ports on all multiplexed addresses. In this case, all ports on all addresses are checked and if any of the **MuxIo.PortA[*n*].Enable** flags are set to 1, then the **MuxIo.PortA[*n*].Data** value is read from or written to depending on the **MuxIo.PortA[*n*].Dir** setting.

Unlike the case where **MuxIo.UpdatePeriod** was set to 0, in this case the **MuxIo.PortA[*n*].Enable** flags are not cleared after each data transfer and the same tasks will be repeated after **MuxIo.UpdatePeriod** milliseconds. In this approach, the data image words **MuxIo.PortA[*n*].Data** are updated periodically as long as **MuxIo.PortA[*n*].Enable** is set to a value of 1.

MuxIo.PortB[*n*].Enable

Description: Data transfer for address *n* multiplexed port B enabled

Range: 0 .. 1

Units: Boolean

Default: 0

MuxIo.PortB[*n*].Enable determines whether data transfer for multiplexed port B on the device with address ***n*** is perfomed by Power PMAC firmware or not. The index ***n*** (= 0 to 31) is the address of multiplexed I/O device as selected by DIP switches on the device.

If **MuxIo.PortB[*n*].Enable** is set to a value of 1, the **MuxIo.PortB[*n*].Data** value is read from or written to, depending on the **MuxIo.PortB[*n*].Dir** setting.

Depending on the **MuxIo.UpdatePeriod** setting, the **MuxIo.PortB[*n*].Enable** flag is either cleared automatically by Power PMAC firmware after each data transfer or left unmodified for next data transfer cycle.

If **MuxIo.UpdatePeriod** is set to the default value of 0, writing or reading data to the multiplexed addresses are performed on-demand basis. In this case, every 50 milliseconds, all ports on all addresses are checked and if any of the **MuxIo.PortB[*n*].Enable** flags are set to 1, then the **MuxIo.PortB[*n*].Data** value is read from or written to depending on the **MuxIo.PortB[*n*].Dir** setting. After the data transfer is completed, the **MuxIo.PortB[*n*].Enable** flag is automatically cleared (set to 0).

Since the serial multiplexed port data transfer is performed as a background task in Power PMAC firmware and it is asynchronous to other tasks such as PLCs or motion programs, users must explicitly check for the **MuxIo.PortB[*n*].Enable** flags to be cleared after setting them high in any PLC or motion program.

In this approach, in order to keep the multiplexed devices from shutting down in a watchdog state due to inactivity, it is necessary for the user to keep updating the **MuxIo.PortB[*n*].Enable** flags by setting them to 1, even if data transfer is not required.

If **MuxIo.UpdatePeriod** is set greater than 0, the Power PMAC will perform a cyclic scan of all enabled ports on all multiplexed addresses. In this case, all ports on all addresses are checked and if any of the **MuxIo.PortB[*n*].Enable** flags are set to 1, then the **MuxIo.PortB[*n*].Data** value is read from or written to depending on the **MuxIo.PortB[*n*].Dir** setting.

Unlike the case where **MuxIo.UpdatePeriod** was set to 0, in this case the **MuxIo.PortB[*n*].Enable** flags are not cleared after each data transfer and the same tasks will be repeated after **MuxIo.UpdatePeriod** milliseconds. In this approach, the data image words **MuxIo.PortB[*n*].Data** are updated periodically as long as **MuxIo.PortB[*n*].Enable** is set to a value of 1.

PowerBrick[*i*]. Saved Data Structure Elements

The **PowerBrick[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the Power Brick servo interface board are listed under the **Gate3[*i*]** data structure, above.

Sys. Saved Data Structure Elements

The **Sys.** data structure comprises the global settings for the Power PMAC. The elements in this structure affect the entire Power PMAC, and are not limited to a particular motor, coordinate system, or ASIC.

Note that while global settings in Turbo PMAC were governed by I-variables in the I0 – I99 range, that range of I-variables have been assigned to Motor 0 in Power PMAC. The "legacy" I-variable numbers for these elements are 8000 greater than the comparable numbers in Turbo PMAC.

Sys.AbortAllBit

Description: Bit number of global abort input in **pAbortAll** register

Range: 0 .. 31

Units: Bit number (little-endian)

Default: Power Brick: 31

Other Power PMACs: 0

Sys.AbortAllBit determines which bit of the 32-bit register whose address is specified by **Sys.pAbortAll** Power PMAC will use to detect the global abort input. The bit number is specified in the "little-endian" convention, where bit *n* has a value of 2^n in the 32-bit word. The setting of **Sys.AbortAllBit** does not matter if **Sys.pAbortAll** is set to 0 to disable the global input function.

When the specified bit is found in its "1" state (no polarity control is provided) for the cumulative number of scans specified by **Sys.AbortAllLimit**, the global status bit **Sys.AbortAll** is set to 1, and each coordinate system reacts according to the setting of its saved setup element **Coord[*x*].AbortAllMode**, which can specify controlled or uncontrolled stops, or can tell the coordinate system to ignore the input.

Sys.AbortAllLimit

Description: Global-abort maximum accumulated number of true detections

Range: 0 .. 255

Units: Scans

Default: 0

Sys.AbortAllLimit specifies the maximum number of accumulated scans the Power PMAC can find the global abort input in its "abort" state without issuing the global abort command. If the number of accumulated scans in the abort state exceeds this value, each coordinate system reacts according to the setting of its saved setup element **Coord[*x*].AbortAllMode**, which can specify

controlled or uncontrolled stops, or can tell the coordinate system to ignore the input. Power PMAC checks the state of the specified global-abort bit every real-time interrupt period.

The global abort input detection circuits can temporarily sense a fault due to electrical noise or similar factors. The ability to confirm a true abort input by requiring multiple consecutive scans detecting the abort state can be valuable in eliminating false trips.

The register for the global abort bit is specified by **Sys.pAbortAll**. The bit within this register is loss state of this bit is specified by **Sys.AbortAllBit**. If **Sys.pAbortAll** is set to 0 to disable the global-abort detection function, the setting of **Sys.AbortAllLimit** does not matter. The accumulated number of scans with the specified bit found in the "abort" state is found in status element **Sys.AbortAllCount**. This bit is incremented each time the bit is found in the abort state, and decremented (if greater than zero) each time the bit is not found in the abort state.

Sys.BgSleepTime

Description: Background cycle pause period

Range: 0, 250 .. 10,000

Units: Microseconds

Default: 0 (specifies 1,000 microseconds)

Sys.BgSleepTime specifies the "sleep" period between consecutive scheduled Power PMAC background cycles during which the Power PMAC application will try to execute any of its background tasks. This period permits the general-purpose operating system (GPOS) to execute its applications. Power PMAC interrupt tasks can execute during this period as well.

In a single scheduled background cycle, Power PMAC will execute one scan of one active background Script PLC program, one scan of all active background C PLC programs, and perform "housekeeping" status and safety checks.

If **Sys.BgSleepTime** is set to its default value of 0, a value of 1,000 microseconds (1 msec) is used, matching the fixed value of earlier firmware versions. The valid range for this element is 250 to 10,000 microseconds (0.25 to 10 milliseconds). A command to set it to a value less than 250 µsec will cause it to be set to 250 µsec; a command to set it to a value greater than 10,000 µsec will cause it to be set to 10,000 µsec.

This element gives the user flexibility in deciding the amounts of background time to be allotted to scheduled background cycle tasks and to GPOS tasks including independent C applications.

Sys.BgWDTReset

Description: Background watchdog timer threshold

Range: 0 .. 2^{32}-1

Units: Background software cycles

Default: 0 (specifies 10 cycles)

Sys.BgWDTReset specifies the maximum number of background cycles that can execute without a "real-time interrupt" (RTI) occurring before Power PMAC will create a "soft" watchdog timer fault. If it is set to its default value of 0, a value of 10 background cycles is used. This monitoring feature is intended to detect a failure or overload of the interrupt hardware or software that could prevent safe control, and provide a shutdown mechanism.

An RTI is a "software interrupt" that should occur every (**Sys.RtIntPeriod** + 1) servo periods. The effective default value of 10 background cycles should be suitable for the great majority of Power PMAC applications. However, applications with unusual timing requirements may require a different setting.

If **Sys.BgWDTReset** is set to too large a value, it may not catch an interrupt-task failure before a "hard" watchdog trip occurs that completely shuts down the Power PMAC.

In a soft watchdog timer trip, all motion programs are aborted, all motors are killed in software (open-loop, zero output, amplifier disabled), and all of the interface ASICs are locked into their "reset" state, which turns off all discrete outputs, including amplifier enables, and forces all continuously variable command outputs (DAC, PWM, PFM) to their zero command value. However, the Power PMAC processor remains active and can execute PLC programs and communicate with the host computer.

The number of background cycles that have executed since the most recent RTI at any given time is stored in the read-only element **Sys.BgWdTimer**. If a fault of this type occurs, the read-only element **Sys.WDTFault** is set to 2 (**RtWdFault**) to indicate a fault from the failure of real-time routines.

Sys.BufIoEnable

Description: Buffered PLC-style I/O transfer enabled control

Range: 0 .. 1

Units: Boolean

Default: 0

Sys.BufIoEnable specifies whether PLC-style buffered input and output transfers are performed or not, and if performed, in which cycle they are done. If set to its default value of 0, these buffered transfers are not performed.

If **Sys.BufIoEnable** is set to 1, Power PMAC will perform regular cyclic operations of copying "input" registers specified by **BufIo[*i*].pIn** into memory input holding registers, calculating and showing differences from the previous scan. It will also copy memory output holding registers to "output" registers specified by **BufIo[*i*].pOut** at the end of the scan.

These cyclic operations can be both in the real-time interrupt and in background. Input registers specified by **BufIo[*i*].pIn** with index ***i*** from 0 through (**Sys.MaxRtBufIn** - 1) could be read each

real-time interrupt cycle. Output registers specified by **BufIo[*i*].pOut** with index ***i*** from 0 through (**Sys.MaxRtBufOut** - 1) could be written to each real-time interrupt cycle.

Input registers specified by **BufIo[*i*].pIn** with index ***i*** from **Sys.MaxRtBufIn** to 63 could be read each background cycle. Output registers specified by **BufIo[*i*].pOut** with index ***i*** from **Sys.MaxRtBufIn** to 63 could be written to each background cycle.

Using this buffered I/O functionality provides several potential advantages. First, for users with experience with typical PLC devices, the actual input/output process is the same as for those PLCs, with all of the inputs read in to holding registers at the start of a scan and all of the outputs written out from holding registers at the end of a scan, with all of the scan's processing done in between.

Second, the reading process automatically detects and stores changes in any input bit, making it much easier to write algorithms that act on these changes. Third, it makes it easier to write efficient code, because the user algorithms are accessing internal fast memory, often multiple times per scan, with the slow access to each actual I/O register automatically occurring only once per scan.

Fourth, if input filtering is enabled by**BufIo[*i*].InScans**, this function automatically provides "debouncing" of the inputs by requiring multiple consecutive scans in the new state of the input before the holding register in memory shows the new state to the user algorithms. This saves the user from needing to write the debouncing algorithms himself.

Finally, input values can be overridden using **BufIo[*i*].ForceInOn** and **BufIo[*i*].ForceInOff**, allowing for simulation and debugging without changing any user code. Similarly, output values can be overridden using **BufIo[*i*].ForceOutOn** and **BufIo[*i*].ForceOutOff**.

Even if this functionality is enabled, there is nothing preventing the user from directly reading or writing to I/O registers. However, if a different task writes directly to an output register that is also written to by this buffered I/O functionality, those changes can be undone by the buffered output write operations.

Sys.BufIoEnable is new in V2.1 firmware, released 1st quarter 2016.

Sys.BusCtrl[*n*]

Description: I/O bus timing control

Range: $0000 .. $FFFF

Units: Bit field

Default: $6666 (***n*** = 0 .. 9, 14 .. 15)
$6747 (***n*** = 10 .. 13)

Sys.BusCtrl[*n*] specifies aspects of the timing of read and write operations for memory-mapped I/O on the Power PMAC. There is a separate element for each of the 16 possible "chip-select" lines (***n*** = 0 to 15) used for I/O access. A table showing all of the chip-select lines, what they access, and their index ***n***, is given in the "*I/O Address Offsets*" chapter of the software reference.

The value of the index ***n*** corresponds to the first hexadecimal digit of the address offset (from the I/O base address in **sys.piom**) of I/O that uses that chip-select line.

The following table summarizes the I/O whose access timing is controlled by each **Sys.BusCtrl[*n*]** element for those I/O presently supported.

I/O Type	**BusCtrl[*n*]** Index	Base Address Offset	I/O Type	**BusCtrl[*n*]** Index	Base Address Offset
PMAC2-Style Servo ICs	6	$600000	General-Purpose I/O	11	$B00000
PMAC2-Style Servo ICs	7	$700000	General-Purpose I/O	12	$C00000
PMAC2-Style MACRO ICs	8	$800000	General-Purpose I/O	13	$D00000
PMAC3-Style ICs	9	$900000	Shared Memory	14	$E00000
General-Purpose I/O	10	$A00000	Shared Memory	15	$F00000

Sys.BusCtrl[*n*] is a 16-bit value, with the low 8 bits (last 2 hex digits) governing write operations for the chip-select line, and the high 8 bits (first 2 hex digits) governing read operations for the chip-select line.

The low 5 bits (bits 0 – 4) of each 8-bit field specify the number of added wait states ($00 to $1F, 0 to 31 decimal) for the chip-select and read or write lines. At 0 wait states, the read or write line is taken low for 40 nanoseconds (nsec) and the chip-select line is held low for 50 nsec after the start of the read or write pulse. Each wait state adds 10 nsec to the end of the low period for both the chip-select and read or write lines. At the factory default values of 6 added wait states, the read or write line is taken low for 100 nsec, and the chip-select line is held low for 110 nsec after the start of the read or write pulse.

Bits 5 and 6 of each 8-bit field control the delay of the read or write line going low after the chip-select line goes low. With a value of 0 in these bits, there is a 30-nsec delay; with a value of 1, there is a 20-nsec delay; with a value of 2, there is a 10-nsec delay; with a value of 3 (the default), there is no delay.

Bit 7 of the write field, if set to 1, takes the write line high 20 nsec earlier than it would if this bit were set to 0. It does not affect the length of the chip-select line pulse. Bit 7 of the read field is reserved for future use.

Caution

The values of the **Sys.BusCtrl[*n*]** elements used for Delta Tau hardware should not be reduced from the factory defaults without explicit instructions from Delta Tau. These elements have user access for special situations and custom hardware. Note that there are "borderline" settings that work under most situations but do not have adequate safety margins for robust operation, and intermittent failures can lead to very dangerous results.

This table shows a common configuration of **Sys.BusCtrl[*n*]**, with the components, bits, and resulting hex digit values.

Hex Digit ($)	6				7				4				7			
Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit Value	-	1	1	0	0	1	1	1	0	1	0	0	0	1	1	1
Component:		*Read Setup*			*Read Wait States*				*W.H. Write Setup*				*Write Wait States*			

Sys.CamEnable

Description: Number of cam tables to enable

Range: 0 .. 255

Units: Tables

Default: 0

Sys.CamEnable specifies the maximum number of cam tables that can be enabled. If set to a value greater than 0, the cam tables with indices from 0 through (**Sys.CamEnable** - 1) will be checked each servo cycle. If the individual table's **CamTable[*m*].Enable** is greater than 0, that table will be executed that cycle. If the individual table's **CamTable[*m*].Enable** is set to 0, that table will be prepared for proper re-enabling. The value of **CamTable[*m*].Enable** is not saved, and is always set to 0 on power-up/reset.

Sys.CompEnable

Description: Number of compensation tables to enable

Range: 0 .. 256

Units: Tables

Default: 0

Legacy I-variable alias: I8051 (I51 in Turbo PMAC)

Sys.CompEnable specifies the number of compensation tables that are enabled (if present). If set to a value of *n*, compensation tables from **CompTable[0]** to **CompTable[*n*-1]** are enabled; any higher-numbered compensation tables are not enabled . When a compensation table is enabled, Power PMAC will compute the correction(s) from it every servo interrupt.

Sys.CompMotor

Description: Number of motor before which comp tables are updated

Range: 0 .. 255

Units: Motor number

Default: 0

Sys.CompMotor specifies when the active cam tables and then compensation tables are calculated in the servo cycle. The corrections for all active compensation tables are computed immediately before the servo-loop closure of the motor whose number is contained in **Sys.CompMotor**.

Most users will leave **Sys.CompMotor** at the default value of 0, so the table corrections are calculated before any motor's servo loop is closed (but after all motor's command trajectories are updated, so the corrections are based on this cycle's desired positions).

However, users who wish to utilize compensation tables for cascading servo loops may want to set **Sys.CompMotor** so that the compensation tables execute after any outer-loop motors close their servo loops, but before any inner-loop motors closes theirs. This eliminates a delay of a servo cycle in getting the command from the outer loop into the inner loop. Users should realize that for the standard use of the compensation tables based on motor desired positions, this adds a servo cycle of delay for motors numbered less than **Sys.CompMotor**.

Sys.CpuTimerIntr

Description: Processor internal timer-based interrupt enabled

Range: 0 .. 1

Units: Boolean

Default: 0 (any clock-generating IC found)
1 (no clock-generating ICs found)

Sys.CpuTimerIntr controls whether the Power PMAC processor uses external signals for its phase and servo interrupts, or creates its own interrupts from an internal timer.

If **Sys.CpuTimerIntr** is set to the default value of 0, it expects to receive external clock signals for its phase and servo interrupt signals. Typically, these signals come from one of the **Gate*n*[*i*]** "DSPGATE1", "DSPGATE2", or "DSPGATE3" Servo or MACRO ICs in the system. The (single) IC with **Gate*n*[*i*].PhaseServoDir** set to 0 to specify output of the phase and servo clock signals is the source of these interrupts for the processor. These clock signals keep the ASIC hardware and Power PMAC software fully synchronized.

In this case, if the CPU does not detect the presence of the clock signals by the end of the power-on/reset initialization procedure, it will set the global status bit **Sys.NoClocks** to 1. Background tasks such as host communications can still operate, but no foreground (interrupt-based) tasks will run, and no motors can be enabled. The watchdog timer is disabled. If the clock signals are present during the initialization procedure, but subsequently lost, the watchdog timer will trip (hardware or software trip, depending on settings).

If **Sys.CpuTimerIntr** is set to 1, the Power PMAC CPU will generate its own phase and servo interrupts from an internal timer. Both interrupts will occur at the same frequency, with a period

set by saved setup element **Sys.ServoPeriod** (in milliseconds). This setting is for the case where there are no "DSPGATE*n*" Servo or MACRO ICs in the system to generate hardware clock signals, as when the Power PMAC is used for simulation or data acquisition, or when motors are controlled through a network such as EtherCAT that does not utilize one of these ASICs.

Note that when the CPU is generating its own interrupts, it cannot output these signals to keep any of the ASICs synchronized to it.

If no clock-generating servo or MACRO ICs are found on re-initialization, then **Sys.CpuTimerIntr** is automatically set to 1 so that the processor will immediately start generating its own timer-based interrupts.

Sys.EcatType

Description: EtherCAT stack software select control

Range: 0 .. 1

Units: Enumeration

Default: 0 (PMAC configurations with servo or MACRO ASICs)
1 (µPower PMAC, Motion Core IPC)

Sys.EcatType specifies which EtherCAT stack software is used to manage the EtherCAT network communications. It is only used at power-on/reset, so if the user wants to change the stack used, the value of **Sys.EcatType** must be changed, a **save** command must be issued, and the controller must be reset with a **$$$** command or the controller power must be cycled.

If **Sys.EcatType** is 0 at power-on/reset, the IgH Etherlab stack is used. The high-priority tasks of this stack execute in the processor's "kernel space", so execute efficiently and with lower latencies. This permits higher network update frequencies. It must be used if "local" motors are to be controlled as well. However, this stack has a smaller feature set and more limited interactive setup software.

If **Sys.EcatType** is 1 at power-on/reset, the Acontis EtherCAT stack is used. This stack supports a broader array of EtherCAT peripheral devices, and has more extensive setup support through the "EC-Engineer" development software. However, it executes in the processor "user space" and so cannot support as high network update frequencies as the Etherlab stack.

Sys.FirstEnc

Description: Number of first encoder conversion table entry to process

Range: 0 .. **Sys.MaxEncoders** – 1

Units: Entry number

Default: 0

Sys.FirstEnc specifies the number of the first encoder conversion table (ECT) entry to process each servo interrupt. Power PMAC will start executing the table entries starting with the entry **EncTable[*n*]** that has this index number (***n*** = **Sys.FirstEnc**) and will continue until it finds an entry with method **Type** = 0 (end of table).

The primary reason to change **Sys.FirstEnc** from its default value of 0 is to facilitate experimentation with different conversion methods during development. Different sections of the table can contain different conversion methods and parameters, and by changing **Sys.FirstEnc**, it is easy to switch back and forth between "sub-tables". In a finished application, it is recommended that only the entries actually used be left in the table and **Sys.FirstEnc** be set to 0 to start at the beginning of the table.

Sys.I[*i*]

Description: I-variable array element

Range: Definition-dependent

Units: Definition-dependent

Power-on default: Definition-dependent

Sys.I[*i*] is the "***i***th" I-variable array element. I-variables are pre-defined saved setup element variables that match the function of Turbo PMAC saved setup variables. Index values ***i*** can range from 0 to 8,191. This array provides an alternate method for accessing I-variables.

Sys.MaxCoords

Description: Maximum number of coordinate systems that can be used

Range: 1 .. 128

Units: Coordinate systems

Default: 16

Sys.MaxCoords specifies the maximum number of coordinate systems that may be used on this Power PMAC. The coordinate systems that can be used are C.S.0 through C.S.(***Sys.MaxCoords*** – *1*). Data structures **Coord[0]** through **Coord[*Sys.MaxCoords* – *1*]** may be used, and the saved setup elements for these data structures are copied to non-volatile flash memory on a **`save`** command.

When the value of **Sys.MaxCoords** is increased, no assumptions should be made about the values of the saved setup elements for the newly usable coordinate systems.

If the user attempts to reduce the value of **Sys.MaxCoords** with a Script command so that a coordinate system that has a motor assigned to it would be left unusable, the command is rejected with an error. The number of the highest-numbered coordinate system with a motor assigned to it

is equal to the value of read-only element **Sys.Coords** minus 1, so it is not possible to set **Sys.MaxCoords** to less than the present value of **Sys.Coords**.

Setting **Sys.MaxCoords** larger than necessary (> **Sys.Coords**) increases the computational load slightly and makes the backup files larger than necessary.

Sys.MaxEcats

Description: Number of EtherCAT networks that can be activated

Range: 0 … 8

Units: Units

Default: 0

Sys.MaxEcats specifies the number of EtherCAT networks that can be enabled on the Power PMAC. The networks represented by data structures **ECAT[*i*]** with index values ***i*** from 0 through **Sys.MaxEcats -** 1 can be enabled.

Sys.MaxMotors

Description: Maximum number of motors that can be used

Range: 1 .. 256

Units: Motors

Default: 32

Sys.MaxMotors specifies the maximum number of motors that may be used on this Power PMAC. The motors that can be used are Motor 0 through Motor (***Sys.MaxMotors*** *– 1*). Data structures **Motor[0]** through **Motor[*Sys.MaxMotors – 1*]** may be used, and the saved setup elements for these data structures are copied to non-volatile flash memory on a **`save`** command.

Note

If **Sys.MaxMotors** is set to a value *N*, then Motor *N* cannot be activated; only Motors 0 through *N-1* can be activated. At the default value of 32, Motors 0 through 31 can be activated, but not Motor 32.

When the value of **Sys.MaxMotors** is increased, no assumptions should be made about the values of the saved setup elements for the newly usable motors.

If the user attempts to reduce the value of **Sys.MaxMotors** with a Script command so that a motor that is activated (**Motor[*x*].ServoCtrl** > 0) or has a phase task enabled (**Motor[*x*].PhaseCtrl** > 0) would be left unusable, the command is rejected with an error (#70). The number of the highest-numbered motor that is activated is equal to the value of read-only

element **Sys.ServoMotors** minus 1, and the number of the highest-numbered motor that has a phase task enabled is equal to the value of read-only element **Sys.PhaseMotors** minus 1 so it is not possible to set **Sys.MaxMotors** to less than the present value of either **Sys.ServoMotors** or **Sys.PhaseMotors**.

Setting **Sys.MaxMotors** larger than necessary (> **Sys.ServoMotors** and **Sys.PhaseMotors**) increases the computational load slightly and makes the backup files larger than necessary.

Sys.MaxRtBufIn

Description: Number of possible buffered real-time input transfers

Range: 0 .. 64

Units: Structure index number

Default: 0

Sys.MaxRtBufIn specifies the number of buffered input registers that can be processed in the real-time interrupt. The index numbers for these buffered inputs range from 0 to (**MaxRtBufIn** – 1). Buffered input registers that can be processed in background have index values that range from **MaxRtBufIn** to 63.

If **Sys.MaxRtBufIn** is set to 0, no buffered input registers can be processed in the real time interrupt. If **Sys.MaxRtBufIn** is set to 64, no buffered input registers can be processed in background.

For example, if **Sys.MaxRtBufIn** is set to 8, **BufIo[0].pIn** through **BufIo[7].pIn** can be used to specify registers to be read in the real-time interrupt. **BufIo[8].pIn** through **BufIo[8].pIn** can be used to specify registers to be read in background.

In operation, each real-time interrupt, Power PMAC will start at **BufIo[0].pIn** and read the specified input registers until it finds a **BufIo[*i*].pIn** that is set to 0, or until it reads all of the input registers up to **BufIo[*MaxRtBufIn-1*].pIn**, whichever comes first.

Each background cycle, Power PMAC will start at **BufIo[*MaxRtBufIn*].pIn** and read the specified input registers until it finds a **BufIo[*i*].pIn** that is set to 0, or until it reads all of the input registers up to **BufIo[63].pIn**, whichever comes first.

Sys.MaxRtBufIn does not need to be set to the same value as corresponding buffered output control element **Sys.MaxRtBufOut**.

Sys.MaxRtBufIn is new in V2.1 firmware, released 1st quarter 2016.

Sys.MaxRtBufOut

Description: Number of possible buffered real-time output transfers

Range: 0 .. 64

Units: Structure index number

Default: 0

Sys.MaxRtBufOut specifies the number of buffered output registers that can be processed in the real-time interrupt. The index numbers for these buffered outputs range from 0 to (**MaxRtBufOut** – 1). Buffered output registers that can be processed in background have index values that range from **MaxRtBufOut** to 63.

If **Sys.MaxRtBufOut** is set to 0, no buffered output registers can be processed in the real time interrupt. If **Sys.MaxRtBufOut** is set to 64, no buffered output registers can be processed in background.

For example, if **Sys.MaxRtBufOut** is set to 8, **BufIo[0].pOut** through **BufIo[7].pOut** can be used to specify registers to be written to in the real-time interrupt. **BufIo[8].pOut** through **BufIo[8].pOut** can be used to specify registers to be written to in background.

In operation, each real-time interrupt, Power PMAC will start at **BufIo[0].pOut** and write to the specified output registers until it finds a **BufIo[*i*].pOut** that is set to 0, or until it reads all of the output registers up to **BufIo[*MaxRtBufOut-1*].pOut**, whichever comes first.

Each background cycle, Power PMAC will start at **BufIo[*MaxRtBufOut*].pOut** and write the specified output registers until it finds a **BufIo[*i*].pOut** that is set to 0, or until it reads all of the input registers up to **BufIo[63].pOut**, whichever comes first.

Sys.MaxRtBufOut does not need to be set to the same value as corresponding buffered input control element **Sys.MaxRtBufIn**.

Sys.MaxRtBufOut is new in V2.1 firmware, released 1st quarter 2016.

Sys.MaxRtPlc

Description: Highest number of foreground PLC

Range: 0 .. 3

Units: PLC program numbers

Default: 0

Sys.MaxRtPlc specifies the highest numbered PLC program that will be executed on the real-time interrupt. All PLC programs with higher numbers will be executed in background. At the default value of 0, only PLC 0 will be executed as a foreground task on the real-time interrupt; all other PLCs will be executed in background. At the maximum permitted value of 3, PLCs 0 – 3 will be executed in foreground, PLCs 4 – 31 will be executed in background.

Note

Any PLC program whose execution priority would be changed by a new value of **Sys.MaxRtPlc** must be disabled at the time of the change. Otherwise the new value for **Sys.MaxRtPlc** will not be accepted.

Sys.MaxTimedUnderflow

Description: Maximum excess negative time base excursion

Range: Non-negative floating-point

Units: Nominal milliseconds

Default: 0.0 (limit disabled)

Sys.MaxTimedUnderflow, if set to a value greater than the default of 0.0, specifies the maximum permitted "time" past the beginning of the present move section or segment for any motor without a move trace buffer (**Motor[*x*].TraceSize** = 0) that is permitted when reversing using negative time base values. If interpolation using negative time base values carries the motor further than **Sys.MaxTimedUnderflow** milliseconds back past the beginning of this section, motion is stopped with an "abort" deceleration on all motors in the coordinate system, with **Coord[*x*].ErrorStatus** set to 20.

Without this protection active, execution under conditions of negative time base can continue indefinitely past the beginning of the present section or segment, which could lead to unpredictable results.

Sys.MotorsPerRtInt

Description: Number of motors status checked each real-time interrupt

Range: 0 .. 255

Units: Motors

Default: 0 (all checked each RTI)

Sys.MotorsPerRtInt specifies how many motors are checked for their status and safety updates each real-time interrupt (RTI). At the default value of 0, all motors are checked each RTI, operation compatible with older firmware versions that did not have this element.

If **Sys.MotorsPerRtInt** is set to a value greater than 0, Power PMAC will check only this number of motors each RTI. This makes it possible to spread these checks out over multiple RTI cycles, reserving more processor time for other calculations. In some applications with very high RTI frequencies required for features such as high block rates in motion programs, high segmentation rates with kinematics and/or lookahead, performing these status and safety checks for every

motor each RTI cycle is not necessary and can take too much processor time, especially when I/O access is required, as for reading safety input flags. For these applications, it can be useful to set **Sys.MotorsPerRtInt** to a value greater than 0 to set a longer time interval between successive status and safety updates for individual motors.

Power PMAC performs the safety and status checks for motors in a cycle starting at Motor 0 and ending at the highest numbered active motor, whose number can be found in status element **Sys.ServoMotors**. This means there is a cycle of (**Sys.ServoMotors** + 1) motors. At the motor's turn in the cycle, Power PMAC first checks to see if the motor is active (**Motor[*x*].ServoCtrl** > 0). If it is active, it proceeds through the safety checks and status updates for that motor. These checks and updates typically take between 0.5 and 1.0 microseconds per motor each cycle.

If **Sys.MotorsPerRtInt** is greater than 0, then in a single RTI cycle, Power PMAC will only do this checking for the specified number of motors. If the total numbers in the cycle (**Sys.ServoMotors** + 1) is not evenly divisible by **Sys.MotorsPerRtInt**, then in the last RTI of the full cycle, fewer motors will be checked. For example, if **Sys.ServoMotors** is 6, there are 7 motors in the full cycle (0 through 6, inclusive). If **Sys.MotorsPerRtInt** is set to 3, in the first RTI of the full cycle, Motors 0, 1, and 2 will be checked, in the second RTI, Motors 3, 4, and 5 will be checked, and in the third RTI, Motor 6 alone will be checked. In this case, the full cycle is 3 RTI periods, and each motor is checked every third RTI.

Safety and status checks performed at the real-time interrupt level include hardware and software overtravel limits, amplifier faults, I^2T integrated current, encoder-loss detection, brake control, backlash compensation, and capture-trigger monitoring for triggered moves. (Following-error and in-position monitoring are performed at the higher servo-interrupt level and are not affected by **Sys.MotorsPerRtInt**.)

A few of the checks and updates done at the RTI level have calculations dependent on the time period between consecutive checks. If **Sys.MotorsPerRtInt** is set greater than 0 to extend the time between consecutive checks, user calculations must be modified to take this extension into account.

Caution

Documentation of these time-dependent safety and status checks, and automatic setup software for them, particularly if written before the implementation of this element, may not reflect the necessary modifications of user calculations for correct operation of these functions with the checking period is extended by **Sys.MotorsPerRtInt**. If **Sys.MotorsPerRtInt** is used to extend the check period, the user should review the calculations for these checks manually to ensure they are correct.

The time extension factor *N* required to modify these calculations is given by the equation:

$$N = ceil\left(\frac{Sys.ServoMotors + 1}{Sys.MotorsPerRtInt}\right)$$

In this equation, "*ceil*" is the "ceiling" function, indicating a rounding up to the next integer (if necessary).

If backlash compensation is used for a motor, the backlash correction is introduced and removed as the direction changes at a rate specified by **Motor[*x*].BlSlewRate**, expressed in "motor units per RTI". If **Sys.MotorsPerRtInt** is greater than 0, this is technically "motors per *N* RTI periods", so the value of **Motor[*x*].BlSlewRate** should be *increased* by a factor of *N* compared to the case where there was no extension.

If automatic brake control is used for a motor, the delay periods **Motor[*x*].BrakeOffDelay** and **Motor[*x*].BrakeOnDelay** are expressed in milliseconds. Power PMAC tracks the delay in the RTI check update, adding to the time elapsed a value of one RTI period. If **Sys.MotorsPerRtInt** is greater than 0, the actual time elapsed since the last check is actually *N* RTI periods, so **Motor[*x*].BrakeOffDelay** and **Motor[*x*].BrakeOnDelay** should be *reduced* by a factor of *N* compared to the case where there was no extension.

If the "I^2T" integrated current limiting protection is used for a motor, the integrated current shutdown limit **Motor[*x*].I2tTrip** is expressed in the square of current units times seconds. Power PMAC accumulates the time in the RTI check update, using for the time elapsed since the last check a value of one RTI period. If **Sys.MotorsPerRtInt** is greater than 0, the actual time elapsed since the last check is actually *N* RTI periods, so **Motor[*x*].I2tTrip** should be *reduced* by a factor of *N* compared to the case where there was no extension.

If a phasing-search move is used to establish a phase reference for a PMAC-commutated synchronous motor, **Motor[*x*].PhaseFindingTime** specifies the time for each stage of the search in RTI periods. If **Sys.MotorsPerRtInt** is greater than 0, this time will actually be *N* RTI periods, so **Motor[*x*].PhaseFindingTime** should be *reduced* by a factor of *N* compared to the case where there was no extension. However, the numerical value of this element still controls the type of search, so it is important not to change the mode when rescaling.

Sys.NoShortCmds

Description: Short-form on-line command-disable control

Range: 0 .. 1

Units: Boolean

Default: 0

Sys.NoShortCmds controls whether "short-form" on-line "action" commands, particularly single-letter commands, will be permitted or not. If it is set to its default value of 0, short-form commands of this type are permitted. If it is set to 1, they are not permitted, and will be rejected with an error for erroneous syntax (Error 20). In this mode, the equivalent "long-form" on-line command must be used instead. Note that long-form commands may also be used when the bit is set to 0.

While the short-form commands are very convenient, especially for interactive terminal-mode work, it can be easy to issue them accidentally, especially when using user-declared variable names – if the name used in a command does not match one in the project database, Power PMAC will try to interpret the letters in the name as individual commands. This can lead to unpredictable and possibly dangerous results.

The following table shows the short-form "action" commands that are disabled by setting **Sys.NoShortCmds** to 1, and the equivalent long-form commands.

Short-Form Command	Long-Form Command	Short-Form Command	Long-Form Command
a	abort	j=={constant}	jog=={constant}
b	begin	j:{constant}	jog:{constant}
h	hold	j^{constant}	jog^{constant}
hm	home	j=*	jog=*
hmz	homez	j:*	jog:*
j+	jog+	j^*	jog^*
j-	jog-	k	kill
j/	jog/	q	pause
j=	jog=	r	run
j={constant}	jog={constant}	s	step

Sys.pAbortAll

Description: Global abort input register pointer

Range: Legitimate addresses

Units: Data structure element addresses

Default: Power Brick: **Gate3[*i*].GpioData[0].a**
Other Power PMACs: 0 (global abort input disabled)

Sys.pAbortAll specifies which register Power PMAC looks to for its "global abort" input bit. It contains the address of this register. If **Sys.pAbortAll** is set to the default value of 0, there is no automatic global abort input function on the Power PMAC.

The value of this variable is usually set by assigning it to the address of the data structure element representing the register to be used, using the "**.a**" suffix for that element. The user does not need to know the numerical value of this address.

In the specified register, Power PMAC uses the bit whose number is contained in **Sys.AbortAllBit** to read the input status. The cumulative number of scans detecting the specified input state before a global abort is issue is specified by **Sys.AbortAllLimit**.

When the specified bit is found in its "1" state (no polarity control is provided) for the specified cumulative number of scans, the global status bit **Sys.AbortAll** is set to 1, and each coordinate system reacts according to the setting of its saved setup element **Coord[*x*].AbortAllMode**, which can specify controlled or uncontrolled stops, or can tell the coordinate system to ignore the input.

Sys.PhaseCycleExt

Description: Phase cycle extension period

Range: 0 .. 255

Units: Phase interrupt periods

Default: 0

Legacy I-variable alias: I8007 (I7 in Turbo PMAC)

Sys.PhaseCycleExt specifies the number of phase interrupt periods that are skipped between consecutive executions of the motor "phase" software tasks (commutation and current-loop closure for selected motors). These tasks are executed every (**Sys.PhaseCycleExt** + 1) phase interrupt periods.

The phase interrupt period is set by one of the Servo or MACRO ICs in the system, using parameters **Gate1[*i*].PwmPeriod** and **Gate1[*i*].PhaseClockDiv** for a PMAC2-style "DSPGATE1" Servo IC, **Gate2[*i*].PwmPeriod** and **Gate2[*i*].PhaseClockDiv** for a PMAC2-style "DSPGATE2" MACRO IC, or **Gate3[*i*].PhaseFreq** for a PMAC3-style "DSPGATE3" machine-interface IC.

Most Power PMAC users will leave **Sys.PhaseCycleExt** at the default value of 0, so that the motor phase algorithms are executed every phase interrupt period. There are two reasons to extend the motor phase update cycle by setting **Sys.PhaseCycleExt** greater than 0.

First, if the Power PMAC is doing direct PWM control of motors over the MACRO ring, it is advisable to set **Sys.PhaseCycleExt** to 1 so that the MACRO ring, which operates on the hardware phase clock, cycles twice per software phase cycle. This will eliminate one phase cycle delay in the closing of the current loops, which permits higher gains and higher performance. For example, the hardware phase clock could be set to 18 kHz, but with **Sys.PhaseCycleExt** = 1, the current loop would be closed at a reasonable 9 kHz.

Second, if many multiplexed A/D converters from ACC-36 boards are used for servo feedback, **Sys.PhaseCycleExt** can be set greater than zero to ensure that each A/D converter is processed once per servo cycle. One pair of multiplexed ADCs is processed each hardware phase clock cycle.

For example, if 8 pairs of multiplexed ADCs needed to be processed each 440 μsec (2.25 kHz) servo cycle, and the software phase update were desired to be at 220 μsec (4.5 kHz), the phase clock update would be set to 18 kHz (18/8 = 2.25) to get through all 8 ADC pairs each servo cycle, **Sys.PhaseCycleExt** would be set to 3 (18/[3+1] = 4.5) to get the software phase update at 4.5 kHz, and the servo cycle clock divider would be set to divide-by-8 (**Gate*n*[*i*].ServoClockDiv** = 7).

There must be an integer number of software phase updates in a servo clock period for smooth operation. For example, if the servo clock frequency is ¼ the Phase clock frequency (**Gate*n*[*i*].ServoClockDiv** = 3), the legitimate values of **Sys.PhaseCycleExt** are 0, which provides 4 software phase updates per servo clock period; 1, which provides 2 updates per period; and 3, which provides 1 update per period. Note that this rule means that the software phase update period must never be longer than the servo clock period.

Sys.PhaseOverServoPeriod

Description: Ratio of phase update period to servo update period

Range: 0.0 .. 1.0

Units: none

Default: 0.25

Sys.PhaseOverServoPeriod specifies the ratio of the phase update period to the servo update period as a floating-point number. This value is only used if a motor is executing its servo algorithms in the phase update (enabled by setting bit 3 of **Motor[*x*].PhaseCtrl** to 1), a specialized feature for very-high-bandwidth actuators such as galvanometers. In this case, it tells the phase update's interpolation algorithm what fraction of a servo cyle to interpolate each phase cycle.

The actual ratio between phase and servo periods is most commonly set by the saved setup element **Gate*n*[*i*].ServoClockDiv** for the Servo or MACRO IC that is providing these clock signals for the Power PMAC system. **Sys.PhaseOverServoPeriod** must be set by the user to inform the phase update algorithms what this ratio is. It can be calculated as:

$$Sys.PhaseOverServoPeriod = \frac{1}{Gaten[i].ServoClockDiv + 1}$$

For the default value of 3 for **Gate*n*[*i*].ServoClockDiv**, 1/(3+1) yields a value of 0.25 for **Sys.PhaseOverServoPeriod**.

Sys.PreCalc

Description: Move pre-calculation buffer time

Range: 0 .. 15

Units: Servo clock cycles

Default: 1

Sys.PreCalc specifies the minimum distance ahead in a continuous program trajectory Power PMAC will calculate. That is, when Power PMAC is calculating motion equations from a motion program, it will continue calculating until it has equations that cover a time period of one real-time interrupt (RTI) period plus **Sys.PreCalc** servo cycles ahead of the already calculated point in the trajectory that is ready for execution – that is, (**Sys.RtIntPeriod** + 1) + **Sys.PreCalc** servo cycles ahead.

The vast majority of Power PMAC applications can use the default value of 1 for **Sys.PreCalc**. Only those applications that have many small programmed moves and move segments whose time is near a single servo cycle may need to increase this value to ensure that adequate pre-calculation is always performed.

If **Sys.PreCalc** is too small for an application, "run-time errors" can occur during the program execution when the pre-calculation fails to stay ahead of move execution (interpolation). The actual failure typically occurs due to a burst of small moves requiring a lot of calculation and/or a missed RTI cycle due to an overload of calculations. In this case, the coordinate system status element **Coord[*x*].RunTimeError** is set to 1 (**Coord[*x*].ErrorStatus** is set to 16).

If **Sys.PreCalc** is too large for an application, it increases the delay between calculation and execution, which can be problematic if there is conditional execution based on the present system state. In addition, the additional pre-calculation distance can overflow the 16-segment buffer. In this case, the coordinate system status element **Coord[*x*].BufferError** is set to 1 (**Coord[*x*].ErrorStatus** is set to 2).

In firmware version 1.6 (released 1st quarter 2014) and newer, the coordinate system status element **Coord[*x*].BufferWarn** can be used to optimize the setting of **Sys.PreCalc** in high block-rate applications. It should be set high enough that **BufferWarn** is sometimes set to 1 (meaning that another move block could not be loaded into the equation buffer), but not so high that **BufferWarn** is set to 2 (which would mean that it had to postpone a move calculation request until the next RTI to avoid a buffer overflow).

Sys.RtIntPeriod

Description: Real-time interrupt period

Range: 0 .. 255

Units: Servo clock cycles - 1

Default: 0

Legacy I-variable alias: I8008 (I8 in Turbo PMAC)

Sys.RtIntPeriod specifies the period of the "real-time interrupt" for Power PMAC. The real-time interrupt occurs every (**Sys.RtIntPeriod** + 1) servo interrupts. At the default value of 0, a real-time interrupt occurs every servo interrupt.

Every real-time interrupt, Power PMAC checks to see if motion-program calculations are required, either for a new programmed move or for a new move segment derived from the programmed move. Also, a scan of each active foreground PLC program is executed.

The real-time interrupt period must be smaller than the smallest move segment time (**Coord[*x*].SegMoveTime**) in any coordinate system. Otherwise, Power PMAC may not be able to compute the motion equations for a segment in time, and the motion program will fail with a "run-time error".

Power PMAC decrements the watchdog timer counter on the real-time interrupt. If **Sys.RtIntPeriod** is set so large that this decrementing does not happen at least 50 times per second, the watchdog timer may trip, shutting down the Power PMAC.

Sys.SendFileMode

Description: Sent-message formatting mode

Range: 0 .. 31

Units: bit field

Default: 0

Sys.SendFileMode specifies how text strings transmitted using the **sendn** command are terminated. **Sys.SendFileMode** is comprised of 5 independent bits – bits 0 to 4 – each bit *n* corresponding to the port *n* used by the **sendn** command.

If a bit *n* is set to the default value of 0, no extra characters are appended to a text string sent to that port; only characters explicitly specified in the **sendn** command are sent. If a bit *n* is set to 1, the characters "**-n:**" (hyphen, port number digit, colon) are automatically appended to the beginning of the text string, and a "null" character (ASCII byte value of 0) is automatically appended to the end of the text string transmitted from each individual **sendn** command.

Appending a null character can make it easier for the receiving software to ensure that it can separate individual messages. For this reason, the "Unsolicited Messages" window in the IDE automatically sets these bits to 1. However, it does not permit the direct concatenation of strings from separate **sendn** commands.

Sys.ServoPeriod

Description: Servo update period for interpolation

Range: Positive floating-point

Units: Milliseconds

Default: 0.44274211

Legacy I-variable alias: I8010 (I10 in Turbo PMAC)

Sys.ServoPeriod tells Power PMAC software how much time there is between servo interrupts (which is controlled by hardware circuitry), so that the interpolation software knows how much time to increment each servo interrupt. It is also used by features such as the "I^2T" integrated current limiting and adaptive servo control algorithms to compute elapsed time in seconds. It is expressed in milliseconds as a floating-point value; numerically it is the reciprocal of the servo interrupt frequency in kilohertz (kHz).

If the servo-interrupt period is determined by a PMAC2-style Servo IC (DSPGATE1), it is controlled by the settings of **Gate1[*i*].PwmPeriod**, **Gate1[*i*].PhaseClockDiv**, and **Gate1[*i*].ServoClockDiv**, where *i* is the number of the IC that controls the servo clock for the system. (Typically ***i*** = 4 for a Power UMAC system.) It can be calculated by the following equation:

$$Sys.ServoPeriod = \frac{(2 * PwmPeriod + 3)(PhaseClockDiv + 1)(ServoClockDiv + 1)}{117{,}964.8}$$

If the servo-interrupt period is determined by a PMAC2-style MACRO IC (DSPGATE2), it is controlled by the settings of **Gate2[*i*].PwmPeriod**, **Gate2[*i*].PhaseClockDiv**, and **Gate2[*i*].ServoClockDiv**, where ***i*** is the number of the IC that controls the servo clock for the system. (Typically ***i*** = 0 for a Power UMAC system.) It can be calculated by the same equation as above.

If the servo-interrupt period is determined by a PMAC3-style machine-interface IC (DSPGATE3, it is controlled by the settings of **Gate3[*i*].PhaseFreq** and **Gate3[*i*].ServoClockDiv**, where ***i*** is the number of the IC that controls the servo clock for the system. (Typically *i* = 0 for a Power UMAC system.) It can be calculated by the following equation:

$$Sys.ServoPeriod = 1000 * \frac{(Gate3[i].ServoClockDiv + 1)}{Gate3[i].PhaseFreq}$$

If the Power PMAC processor generates its own phase and servo interrupts internally with saved setup element **Sys.CpuTimerIntr** set to 1, **Sys.ServoPeriod** also specifies the physical time period for these interrupts. This time period is only set at power-on/reset, so the settings must be saved and the Power PMAC reset to effect a change.

Sys.SimConfigOk

Description: Simulated configuration download enable

Range: 0 .. 1

Units: Boolean

Default: 0

Sys.SimConfigOk determines whether it is permissible to download address settings for a hardware configuration that is not present. At the default setting of 0, if a command to set the value of an addressing saved setup element (e.g. **Motor[5].pDac**) to hardware that is not present in the system (e.g. **Acc24E2A[6].Chan[0].Dac[0].a**) is sent to the Power PMAC, the command will be rejected with an error.

However, if **Sys.SimConfigOk** is set to 1, then a command to set the value of an addressing saved setup element to hardware that is not present in the system will be accepted without error. In this case, if the value of this element is queried, Power PMAC will report the numerical offset value of the address of non-existent hardware (e.g. **Sys.piom+$700040**) instead of using the element name (e.g. **Acc24E2A[6].Chan[0].Dac[0].a**).

Setting **Sys.SimConfigOk** to 1 can be useful for testing software project configurations on processor-only simulations. It can also be useful when a variety of system configurations are produced in a modular fashion – the full system configuration can be downloaded even to a partial system without error. In these cases, it is recommended to include the command `Sys.SimConfigOk=1` in the project configuration file `pp_disable.txt` so that this setting

can be guaranteed to be made before any of the hardware setup elements are sent to Power PMAC.

Sys.WDTReset

Description: Foreground watchdog timer threshold

Range: 0 .. 2^{32}-1

Units: Real-time interrupt periods

Default: 0 (specifies 5000 cycles)

Sys.WDTReset specifies the maximum number of "real-time interrupt" (RTI) cycles that can execute without a background software cycle finishing before Power PMAC will create a "soft" watchdog timer fault. If it is set to its default value of 0 (or any value less than 100), a value of 5000 RTI cycles is used. This monitoring feature is intended to detect a software overload of the processor that could prevent safe control, and provide a shutdown mechanism.

Background software executes in the time available between tasks that execute under the phase, servo, and real-time interrupts. It performs its various tasks in a cyclic manner, with each cycle executing a single scan of one active background script PLC program, a single scan of all active background C PLC programs, and various "housekeeping" tasks. By its nature, the cycle time for these background tasks is not precisely predictable, but these tasks should always execute in a reasonably timely fashion.

The effective default value of 5000 RTI cycles should be suitable for the great majority of Power PMAC applications. However, applications with unusual timing requirements may require a different setting.

In a soft watchdog timer trip, all motion programs are aborted, all motors are killed in software (open-loop, zero output, amplifier disabled), and all of the interface ASICs are locked into their "reset" state, which turns off all discrete outputs, including amplifier enables, and forces all continuously variable command outputs (DAC, PWM, PFM) to their zero command value. However, the Power PMAC processor remains active and can execute PLC programs and communicate with the host computer.

Each background cycle, the read-only element **Sys.WdTimer** is set to the value of **Sys.WDTReset** (or to 5000 if **Sys.WDTReset** is less than 100). Each RTI cycle, **Sys.WdTimer** is decremented by 1. If the value of **Sys.WdTimer** ever reaches 0, Power PMAC executes a soft watchdog timer trip. If a fault of this type occurs, the read-only element **Sys.WDTFault** is set to 1 (**BgWdFault**) to indicate a fault from failure of background routines.

Sys.ZeroVelSetPoint

Description: Desired-velocity-zero threshold

Range: Non-negative floating-point

Units: Motor position units per servo cycle

Default: 0.0

Sys.ZeroVelSetPoint specifies the maximum magnitude of any motor's net desired velocity (**Motor[*x*].DesVel**) that will be considered close enough to zero to cause the desired-velocity-zero status bit (**Motor[*x*].DesVelZero**) to be set to 1. Since this status bit must be set to 1 before the motor can be considered to be "in position" (**Motor[*x*].InPos** = 1), a motor cannot be "in position" if the magnitude of desired velocity is greater than this threshold.

The main reason to set a value for **Sys.ZeroVelSetPoint** greater than the default value of 0.0 is the use of a trajectory pre-filter as a low-pass filter for a motor (particularly if it is an "infinite impulse response" filter). With this type of filter, when the programmed trajectory for the motor is finished, the velocity output from the filter asymptotically approaches zero, and the exact time when the numeric representation of this value becomes exactly 0.0 depends of the details of the processor's floating-point math implementation, and can vary depending on the motor position.

At the default value for **Sys.ZeroVelSetPoint** of 0.0, **Motor[*x*].DesVel** must be exactly equal to 0.0 for the **DesVelZero** status bit to be set to 1. If there is no trajectory pre-filter in use for a motor, this setting is fine, because the **DesVel** value is forced exactly to 0.0 at the end of a commanded trajectory.

Typical non-zero values of **Sys.ZeroVelSetPoint** to ensure timely setting of the **DesVelZero** status bit as a trajectory pre-filter output converges to the final position are usually in the range of 10^{-10} (**`1E-10`**) to 10^{-20} (**`1E-20`**). These values are small enough that the status bit would not be set on any realistic very slow commanded move.

Sys.ZeroVelSetPoint is new in V2.0.2 firmware, released 2nd quarter 2015. At the default value of 0.0, operation is completely compatible with older firmware versions.

POWER PMAC NON-SAVED SETUP DATA STRUCTURE ELEMENTS

This chapter documents data structure elements that may be useful to write to in an application, but whose values are not stored to flash memory during a save command, or copied from flash memory during a power-up/reset.

Acc5E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc5E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate2[*i*]** data structure. The data structure elements for the ACC-5E MACRO interface board are listed under the **Gate2[*i*]** data structure, below.

Acc5E3[*i*]. Non-Saved Setup Data Structure Elements

The **Acc5E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-5E3 MACRO interface board are listed under the **Gate3[*i*]** data structure, below.

Acc5EP3[*i*]. Non-Saved Setup Data Structure Elements

The **Acc5EP3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-5EP3 Etherlite MACRO interface board are listed under the **Gate3[*i*]** data structure, below.

Acc11C[*i*]. Non-Saved Setup Data Structure Elements

The **Acc11C[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-11C digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc11E[i]. Non-Saved Setup Data Structure Elements

The **Acc11E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-11E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Note that the ACC-11E cannot be auto-identified by the Power UMAC CPU, so to use this data structure, the user must manually set **GateIo[*i*].PartNum** to 603307, **GateIo[*i*].PartType** to 8, issue a **`save`** command, and reset the controller, before this structure can be used. This structure name is new in V2.0 firmware, released 1st quarter 2015.

Acc14E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc14E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-14E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc24C2[*i*]. Non-Saved Setup Data Structure Elements

The **Acc24C2[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24C2 digital servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24C2A[*i*]. Non-Saved Setup Data Structure Elements

The **Acc24C2A[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24C2A analog servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2[*i*]. Non-Saved Setup Data Structure Elements

The **Acc24E2[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2 digital servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2A[*i*]. Non-Saved Setup Data Structure Elements

The **Acc24E2A[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2A analog servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2S[*i*]. Non-Saved Setup Data Structure Elements

The **Acc24E2S[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2S stepper interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E3[*i*]. Non-Saved Setup Data Structure Elements

The **Acc24E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-24E3 servo interface board are listed under the **Gate3[*i*]** data structure, below.

Acc36E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc36E[*i*]** data structure provides access to the hardware registers of the ACC-36E 16-channel 12-bit analog-to-digital converter board.

Acc36E[*i*].ConvertCode

Description: ADC selection and conversion code

Range: $000000 .. $00F00F

Units: Bit field

Power-on default: 0

Acc36E[*i*].ConvertCode specifies which pair of multiplexed analog-to-digital converters (ADCs) on the ACC-36E board with board index ***i*** will be selected next, and whether the conversions will be in unipolar or bipolar mode. It is a 24-bit value, represented by 6 hexadecimal digits.

Legitimate values of **Acc36E[*i*].ConvertCode** take the form $00*m*00*n*, where *m* and *n* can take any hex value from 0 to F. The *m* value determines which of the "high" analog inputs ANAI08 to ANAI15 is to be read, and how it is to be converted, according to the following formulas:

```
m = ANAI# - 8        // 0V to +20V unipolar input, unsigned result
m = ANAI#            // -10V to +10V bipolar input, signed result
```

The result of this conversion will be found in status element **Acc36E[*i*].ADCuHigh** for an unsigned result, or in **Acc36E[*i*].ADCsHigh** for a signed result, once the status bit **Acc36E[*i*].ADCRdyHigh** is set to 1 (about 5 microseconds after the convert code is set).

The *n* value determines which of the "low" analog inputs ANAI08 to ANAI15 is to be read, and how it is to be converted, according to the following formulas:

```
m = ANAI#            // 0V to +20V unipolar input, unsigned result
m = ANAI# + 8        // -10V to +10V bipolar input, signed result
```

The result of this conversion will be found in status element **Acc36E[*i*].ADCuLow** for an unsigned result, or in **Acc36E[*i*].ADCsLow** for a signed result, once the status bit **Acc36E[*i*].ADCRdyLow** is set to 1 (about 5 microseconds after the convert code is set).

It is not possible to read back a value that has been written to **Acc36E[*i*].ConvertCode**. An attempt to read it will actually read the value in status element **Acc36E[*i*].ADCHighLow**.

Acc36E[*i*].ConvertCode will be written to every phase cycle by the automatic ADC de-multiplexing algorithms if addressed by saved setup element **AdcDemux.Address[*i*]** and enabled by **AdcDemux.Enable**. In this case, the user should not attempt to write directly to this element.

In the Script environment, **Acc36E[*i*].ConvertCode** is a 24-bit element in the high 24 bits of the 32-bit data bus. In the C environment, it is a 32-bit element with the low 8 bits as "don't care"

values, so in C, the value of this element is 256 times that of the Script environment (so its format in C is 0x00*m*00*n*00).

Acc51C[*i*]. Non-Saved Setup Data Structure Elements

The **Acc51C[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-51C sine-encoder interface board are listed under the **Gate1[*i*]** data structure, below.

Acc51E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc51E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-51E sine-encoder interface board are listed under the **Gate1[*i*]** data structure, below.

Acc58E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc58E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-58E resolver-to-digital converter board are listed under the **Gate1[*i*]** data structure, below.

Acc59E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc59E[*i*]** data structure provides access to the hardware registers of the ACC-59E 8-channel 12-bit analog-to-digital converter, digital-to-analog converter board.

Acc59E[*i*].ConvertCode

Description: ADC selection and conversion code

Range: $000 .. $00F

Units: Bit field

Power-on default: 0

Acc59E[*i*].ConvertCode specifies which of the multiplexed analog-to-digital converters (ADCs) on the ACC-59E board with board index ***i*** will be selected next, and whether the conversion will be in unipolar or bipolar mode. It is a 12-bit value, represented by 3 hexadecimal digits.

Legitimate values of **Acc59E[*i*].ConvertCode** take the form $00*m*, where *m* can take any hex value from 0 to F. The *m* value determines which of the analog inputs ANAI1 to ANAI8 is to be read, and how it is to be converted, according to the following formulas:

```
m = ANAI# - 1        // 0V to +20V unipolar input, unsigned result
m = ANAI# + 7        // -10V to +10V bipolar input, signed result
```

The result of this conversion will be found in status element **Acc59E[*i*].ADCu** for an unsigned result, or in **Acc59E[*i*].ADCs** for a signed result, once the status bit **Acc59E[*i*].ADCRdy** is set to 1 (about 5 microseconds after the convert code is set).

It is not possible to read back a value that has been written to **Acc59E[*i*].ConvertCode**. An attempt to read it will actually read the value in status element **Acc59E[*i*].ADCu** or **ADCs**.

Acc59E[*i*].ConvertCode will be written to every phase cycle by the automatic ADC de-multiplexing algorithms if addressed by saved setup element **AdcDemux.Address[*i*]** and enabled by **AdcDemux.Enable**. In this case, the user should not attempt to write directly to this element.

In the Script environment, **Acc36E[*i*].ConvertCode** is a 12-bit element in the low 12 bits of the 24-bit data bus. In the C environment, it is a 32-bit element with the low 8 bits and high 12 bits as "don't care" values, so in C, the value of this element is 256 times that of the Script environment (so its format in C is 0x00000*m*00).

Acc59E[*i*].DAC[*j*]

Description: DAC command value

Range: 0 .. 4095

Units: 12-bit DAC LSBs

Power-on default: 0

Acc59E[*i*].DAC[*j*] specifies the value to be written to the 12-bit digital-to-analog converter with index ***j*** (corresponding to the DAC*n* output pins, where ***j*** = *n* - 1) on the ACC-59E board with board index ***i***. It is an unsigned 12-bit value, with a value of 0 corresponding to a 0V output on the DAC*j*+ and DAC*j*- pins, and a value of 4095 corresponding to a +10V output on the DAC*j*+ pin and a -10V output on the DAC*j*- pin.

Acc59E[*i*].DAC[*j*] cannot be accessed from the C environment. C programs must use the dual-DAC element **Acc59E[*i*].DacHighLow[*j*]** instead.

Acc59E[*i*].DACHighLow[*j*]

Description: DAC pair combined command value

Range: 0 .. 16,777,215

Units: Dual12-bit DAC LSBs

Power-on default: 0

Acc59E[*i*].DACHighLow[*j*] specifies the value to be written to the 12-bit digital-to-analog converter with index ***j*** (corresponding to the DAC*n* output pins, where ***j*** = *n* - 1) and the 12-bit analog converter with index ***j+4*** (corresponding to the DAC{*n*+4} output pins) on the ACC-59E board with board index ***i***. It is an unsigned 24-bit value, with the low 12 bits specifying the unsigned value to be written to the DAC with index ***j*** and the high 12 bits specifying the unsigned value to be written to the DAC with index ***j+4***.

For each DAC, a value of 0 corresponds to a 0V output on the DAC*j*+ and DAC*j*- pins, and a value of 4095 corresponds to a +10V output on the DAC*j*+ pin and a -10V output on the DAC*j*- pin. To assemble a value of **Acc59E[*i*].DACHighLow[*j*]**, multiply the command value for the DAC with index ***j+4*** by 4096 and add it to the value for the DAC with index ***j***.

In the Script environment, **Acc59E[*i*].DACHighLow[*j*]** is a 24-bit value in the high 24 bits of the 32-bit data bus. In the C environment, it is a 32-bit value with the low 8 bits as “don’t care” bits, so the value of this element in the C environment should be 256 times larger than the value in the Script environment.

Acc59E3[*i*]. Non-Saved Setup Data Structure Elements

The **Acc59E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-59E3 A/D and D/A-converter board are listed under the **Gate3[*i*]** data structure, below.

Acc65E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc65E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-65E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc66E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc66E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-66E digital input board are listed under the **GateIo[*i*]** data structure, below.

Acc67E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc67E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-67E digital output board are listed under the **GateIo[*i*]** data structure, below.

Acc68E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc68E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-68E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc72EX[*i*]. Non-Saved Setup Data Structure Elements

Power PMAC has full support for the ACC-72EX fieldbus interface card and all its specific fieldbus communications options. Due to the built-in data structures for accessing ACC-72EX dual ported RAM from Power PMAC, no additional software is required for memory mapping and/or identification..

This section of the manual covers Power PMAC's built in data structures for the ACC-72EX in addition to providing examples for header files, start-up and handshaking PLCs.

All of the interactions with ACC-72EX can be achieved through data structures defined specifically for ACC-72EX in Power PMAC firmware. The following structures allow access to the DPRAM in bit, byte, 2-byte and 4-byte wide access modes. The bit-wise read and write is only supported through **Acc72EX[*i*].Udata16[*j*]** data structure.

Acc72EX[*i*].Data8[*j*]

Description: ACC-72EX DPRAM "unsigned 8-bit integer" data array element

Range: 0 .. 2^8-1

Units: address dependent

Power-on default: address dependent

Acc72EX[*i*].Data8[*j*] is the "***j***th" unsigned 8-bit integer data array element in the **Acc72EX[*i*]** dual ported RAM. Each of these elements occupies one byte in the DPRAM, and is located starting at ***j*** addresses past the beginning of the buffer (which is located at the address in **Acc72EX[*i*].a**). This array is defined based upon the Hilscher ComX memory map.

Index values ***j*** in the square brackets can be integer constants in the range 0 to 524,287, or local L-variables. No expressions or non-integer constants are permitted. The size of the DPRAM is dependent on the ACC-72EX communication option and installed Hilscher ComX module.

Acc72EX[*i*].Data8[*j*] is located in the same registers as **Acc72EX[*i*].Idata16[*j/2*]**, **Acc72EX[*i*].Udata16[*j/2*]**, **Acc72EX[*i*].Idata32[*j/4*]**, **Acc72EX[*i*].Fdata[*j/4*]**, **Acc72EX[*i*].Idata32[*j/4*]**, and **Acc72EX[*i*].Udata32[*j/4*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

Acc72EX[*i*].Fdata[*j*]

Description: ACC-72EX DPRAM "32-bit floating-point" data array element

Range: -2^{31} .. 2^{31}-1

Units: address dependent

Power-on default: address dependent

Acc72EX[*i*].Fdata[*j*] is the "***j***th" 32-bit floating-point data array element in the **Acc72EX[*i*]** dual ported RAM. Each of these elements occupies four bytes in the DPRAM, and is located starting at ***4*j*** addresses past the beginning of the buffer (which is located at the address in **Acc72EX[*i*].a**). This array is defined based upon the Hilscher ComX memory map.

Index values ***j*** in the square brackets can be integer constants in the range 0 to 131,072, or local L-variables. No expressions or non-integer constants are permitted. The size of the DPRAM is dependent on the ACC-72EX communication option and installed Hilscher ComX module.

Acc72EX[*i*].Fdata[*j*] is located in the same registers as **Acc72EX[*i*].Data8[4**j*]** to **Acc72EX[*i*].Data8[*4*j+3*]**, **Acc72EX[*i*].Idata16[2**j*]** to **Acc72EX[*i*].Idata16[2**j+1*]**, **Acc72EX[*i*].Udata16[2**j*]** to **Acc72EX[*i*].Udata16[2**j+1*]**, **Acc72EX[*i*].Idata32[*j*]**,and **Acc72EX[*i*].Udata32[*j*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

Acc72EX[*i*].Idata16[*j*]

Description: ACC-72EX DPRAM "signed 16-bit integer" data array element

Range: -2^{15} .. $2^{15}-1$

Units: address dependent

Power-on default: address dependent

Acc72EX[*i*].Idata16[*j*] is the "***j***th" signed 16-bit integer data array element in the **Acc72EX[*i*]** dual ported RAM. Each of these elements occupies two bytes in the DPRAM, and is located starting at ***2*j*** addresses past the beginning of the buffer (which is located at the address in **Acc72EX[*i*].a**). This array is defined based upon the Hilscher ComX memory map.

Index values ***j*** in the square brackets can be integer constants in the range 0 to 262,143, or local L-variables. No expressions or non-integer constants are permitted. The size of the DPRAM is dependent on the ACC-72EX communication option and installed Hilscher ComX module.

Acc72EX[*i*].Idata16[*j*] is located in the same registers as **Acc72EX[*i*].Data8[2**j*]** to **Acc72EX[*i*].Data8[*2*j+1*]**, **Acc72EX[*i*].Udata16[*j*]**, **Acc72EX[*i*].Fdata[*j/2*]**, **Acc72EX[*i*].Idata32[*j/2*]**, and **Acc72EX[*i*].Udata32[*j/2*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

Acc72EX[*i*].Idata32[*j*]

Description: ACC-72EX DPRAM "signed 32-bit integer" data array element

Range: -2^{31} .. $2^{31}-1$

Units: address dependent

Power-on default: address dependent

Acc72EX[*i*].Idata32[*j*] is the "***j***th" signed 32-bit integer data array element in the **Acc72EX[*i*]** dual ported RAM. Each of these elements occupies four bytes in the DPRAM, and is located starting at ***4*j*** addresses past the beginning of the buffer (which is located at the address in **Acc72EX[*i*].a**). This array is defined based upon the Hilscher ComX memory map.

Index values ***j*** in the square brackets can be integer constants in the range 0 to 131,072, or local L-variables. No expressions or non-integer constants are permitted. The size of the DPRAM is dependent on the ACC-72EX communication option and installed Hilscher ComX module.

Acc72EX[*i*].Idata32[*j*] is located in the same registers as **Acc72EX[*i*].Data8[4**j*]** to **Acc72EX[*i*].Data8[*4*j+3*]**, **Acc72EX[*i*].Idata16[2**j*]** to **Acc72EX[*i*].Idata16[2**j+1*]**, **Acc72EX[*i*].Udata16[2**j*]** to **Acc72EX[*i*].Udata16[2**j+1*]** , **Acc72EX[*i*].Fdata[*j*]** and **Acc72EX[*i*].Udata32[*j*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

Acc72EX[*i*].Udata16[*j*]

Description: ACC-72EX DPRAM "unsigned 16-bit integer" data array element

Range: 0 .. 2^{16}-1

Units: address dependent

Power-on default: address dependent

Acc72EX[*i*].Udata16[*j*] is the "***j***th" unsigned 16-bit integer data array element in the **Acc72EX[*i*]** dual ported RAM. Each of these elements occupies two bytes in the DPRAM, and is located starting at ***2*j*** addresses past the beginning of the buffer (which is located at the address in **Acc72EX[*i*].a**). This array is defined based upon the Hilscher ComX memory map.

Index values ***j*** in the square brackets can be integer constants in the range 0 to 262,143, or local L-variables. No expressions or non-integer constants are permitted. The size of the DPRAM is dependent on the ACC-72EX communication option and installed Hilscher ComX module.

Acc72EX[*i*].Udata16[*j*] is located in the same registers as **Acc72EX[*i*].Data8[2**j*]** to **Acc72EX[*i*].Data8[*2*j+1*], Acc72EX[*i*].Idata16[*j*]**, **Acc72EX[*i*].Fdata[*j/2*]**, **Acc72EX[*i*].Fdata[*j/2*]**, **Acc72EX[*i*].Idata32[*j/2*]**, and **Acc72EX[*i*].Udata32[*j/2*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

Individual bits of this element can be accessed for read or write purposes in the Script environment through the use of the syntax **Acc72EX[*i*].Udata16[*j*].*k***, where ***k*** is an integer constant in the range 0 to 15 representing the bit number.

Consecutive sets of bits of this element can be accessed for read or write purposes in the Script environment through the use of the syntax **Acc72EX[*i*].Udata16[*j*].*k*.*l***, where ***k*** is an integer constant in the range 0 to 15 representing the starting (low) bit number, and ***l*** is an integer constant in the range 1 to 16, not to exceed (16 – ***k***), representing the number of consecutive bits to be accessed.

Udata16 is the only ACC-72EX data format that can be accessed in this bit-wise format.

Acc72EX[*i*].Udata32[*j*]

Description: ACC-72EX DPRAM "unsigned 32-bit integer" data array element

Range: 0 .. 2^{32}-1

Units: address dependent

Power-on default: address dependent

Acc72EX[*i*].Udata32[*j*] is the "***j***th" unsigned 32-bit integer data array element in the **Acc72EX[*i*]** dual ported RAM. Each of these elements occupies four bytes in the DPRAM, and is located starting at ***4*j*** addresses past the beginning of the buffer (which is located at the address in **Acc72EX[*i*].a**). This array is defined based upon the Hilscher ComX memory map.

Index values ***j*** in the square brackets can be integer constants in the range 0 to 262,143, or local L-variables. No expressions or non-integer constants are permitted. The size of the DPRAM is dependent on the ACC-72EX communication option and installed Hilscher ComX module.

Acc72EX[*i*].Udata32[*j*] is located in the same registers as **Acc72EX[*i*].Data8[4**j*]** to **Acc72EX[*i*].Udata8[*4*j+3*]**, **Acc72EX[*i*].Idata16[2**j*]** to **Acc72EX[*i*].Idata16[2**j+1*]**, **Acc72EX[*i*].Udata16[2**j*]** to **Acc72EX[*i*].Udata16[2**j+1*]**, **Acc72EX[*i*].Fdata[*j*]**, and **Acc72EX[*i*].Idata32[*j*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

Acc84B[*i*]. Non-Saved Setup Data Structure Elements

The **Acc84B[*i*]** data structure provides access to the registers of the serial encoder interface board for Power Brick systems. Its elements are documented under the equivalent **Acc84E[*i*]** data structure.

Acc84C[*i*]. Non-Saved Setup Data Structure Elements

The **Acc84C[*i*]** data structure provides access to the registers of the serial encoder interface board for Compact UMAC systems. Its elements are documented under the equivalent **Acc84E[*i*]** data structure.

Acc84E[*i*]. Non-Saved Setup Data Structure Elements

The **Acc84E[*i*]** data structure provides access to the registers of the serial encoder interface board for UMAC systems. There are identical **Acc84C[*i*]** and **Acc84S[*i*]** data structures for equivalent interfaces in other form factors. The non-saved setup data structure elements in this section configure the optional second four channels on an ACC-84 board. Presently, this option is only available on the ACC-84C Compact UMAC board.

Acc84E[*i*].AuxSerialEncCtrl

Description: Accessory control word for second IC serial encoder configuration

Range: $0 .. $FFFFFF

Units: Bit field

Power-on default: Protocol specific

Acc84E[*i*].AuxSerialEncCtrl specifies the multi-channel setup for serial encoder interfaces on the optional second encoder-interface IC on an ACC-84*x* board. Presently, only the ACC-84C Compact UMAC board supports this second IC.

Acc84E[*i*].AuxSerialEncCtrl settings are equivalent to those for saved setup element **Acc84E[*i*].SerialEncCtrl**, which specifies the setting for the first encoder IC on the board. However, the value of **AuxSerialEncCtrl** is not saved, so will need to be configured by the user application after every power-on/reset.

Refer to the description for **Acc84E[*i*].SerialEncCtrl** for details on the setting of this element. This element is new in V2.0 firmware, released 4th quarter 2014.

Acc84E[*i*].AuxChan[*j*].SerialEncCmd

Description: Accessory channel control word for second IC serial encoder commands

Range: $0 .. $FFFFFF

Units: Bit field

Power-on default: Protocol specific

Acc84E[*i*].AuxChan[*j*].SerialEncCmd specifies the command information for the serial encoder interface of the channel on the optional second encoder-interface IC on an ACC-84*x* board. Presently, only the ACC-84C Compact UMAC board supports this second IC. Note that the specific protocol, trigger timing, and clock frequency are determined by the multi-channel element **Acc84E[*i*].AuxSerialEncCtrl**.

Acc84E[*i*].AuxChan[*j*].SerialEncCmd settings are equivalent to those for saved setup element **Acc84E[*i*].Chan[*j*].SerialEncCmd**, which specifies the setting for the channel on the first encoder IC on the board. However, the value of **Acc84E[*i*].AuxChan[*j*].SerialEncCmd** is not saved, so will need to be configured by the user application after every power-on/reset.

Refer to the description for **Acc84E[*i*].Chan[*j*].SerialEncCmd** for details on the setting of this element. This element is new in V2.0 firmware, released 4th quarter 2014.

Acc84S[*i*]. Non-Saved Setup Data Structure Elements

The **Acc84S[*i*]** data structure provides access to the registers of the serial encoder interface board for Power Clipper embedded systems. Its elements are documented under the equivalent **Acc84E[*i*]** data structure.

Note that the ACC-84S cannot be auto-identified by the Power Clipper CPU, so to use this data structure, the user must manually set **GateIo[*i*].PartNum** to 603936, **GateIo[*i*].PartType** to 8, issue a **`save`** command, and reset the controller, before this structure can be used. This structure name is new in V2.0 firmware, released 1st quarter 2015.

BrickAC. Non-Saved Data Structure Elements

The Power Brick AC is one class of the Power Brick family of intelligent amplifiers. It can support multiple types of brush, brushless and AC-induction motors, operating from an AC line input. Its registers can be accessed through **BrickAC.** data structure.

Note

The **BrickAC.** data structure elements documented in this section are software elements that are distinct from the **PowerBrick[*i*].** hardware data structure elements that form the control and status registers for the ASIC(s) in the Power Brick AC. (The **PowerBrick[*i*].** data structure is an "alias" for the **Gate3[*i*].** hardware data structure.)

BrickAC. Multi-Channel Non-Saved Setup Elements

Some aspects of the Brick AC amplifier are common to all channels on the board. The non-saved setup elements in this section affect all channels.

BrickAC.Config

Description: Amplifier configuration/initialization control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickAC.Config acts as a flag for the Power PMAC firmware which controls the initialization of Power Brick AC amplifier based upon the **BrickAC.** saved setup elements. The amplifier stage is not automatically configured at power-up, so the configuration process *must* be commanded explicitly by the user application before the amplifier stage can be used.

Setting **BrickAC.Config** to 1 in a Script command starts the initialization process as a background task on Power PMAC CPU. The element stays at the set value until either the initialization process is successfully completed, in which case the value of **BrickAC.Config** is set to 0, or until a configuration error is detected, in which case the **BrickAC.Config** value is set to a negative value indicating the error in the process.

The following list shows the error codes which can be encountered:

Error Code	Description
-1	The assigned value is not accepted. Only a value of 1 or 0 can be assigned by user to this data structure.
-2	The **BrickAC.Monitor** was was called while either the **BrickAC.Reset** or **BrickAC.Config** process was active.
-3	The configuration process was attempted on incompatible hardware. No amplifier hardware with the matching Power Brick part number was detected.
-4	No Power Brick hardware was detected. This error is generated if incompatible output stage is detected.
-7	The configuration process attempted used on incompatible hardware. No DPSGATE3 interface ASIC was detected.

If **BrickAC.Config** is set to 1 in an on-line command, there will be a text response indicating whether the configuration completed correctly or not, and if not, what the error was.

It is strongly recommended for users to confirm the pass/fail status of the initialization process whenever **BrickAC.Config** is set to a value of 1.

Note

While setting **BrickAC.Config** to 1 as part of the standard system initialization process after power-up will load the configuration parameters into the amplifier control circuitry, it is recommended instead to set **BrickAC.Reset** to 1, which will not only load the configuration parameters, but clear any faults that may have occurred due to power-on transient conditions.

Note

Setting **BrickAC.Config** to 1 to start the amplifier configuration process automatically stops the amplifier monitoring process, and the monitoring process does *not* automatically resume when the configuration is completed. **BrickAC.Monitor** must be set to 1 again in the user Script application to resume the monitoring process.

Example

```
BrickAC.Config = 1
while (BrickAC.Config > 0) {}
if (BrickAC.Config < 0)
{
        // take necessary action in case of a fault
}
// continue with script process
```

BrickAC.Monitor

Description: Amplifier status monitoring update control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickAC.Monitor acts as a flag for the Power PMAC firmware which controls the execution of Power Brick AC amplifier status monitoring background task. This task updates the **BrickAC.** status elements at constant period set by saved setup element **BrickAC.MonitorPeriod**.

If **BrickAC.Monitor** is set to its power-on default value of 0, there is no updating of the **BrickAC.** status elements. In this mode none of these element values are updated and they maintain their last updated value until next reset or power cycle.

Setting **BrickAC.Monitor** equal to 1 in a Script command starts the background **BrickAC.** status update task at a period set by **BrickAC.MonitorPeriod**. The element stays at the set value until either the user application sets the value to 0, which stops the update process, or the user application commands an initialization or reset process by setting **BrickAC.Config** or **BrickAC.Reset** to a value of 1.

If an error occurs during the monitor process, the **BrickAC.Monitor** value is set to a negative value indicating an error in the process. The following table shows the errors that can be reported. It is strongly recommended for users to confirm the pass/fail status of the monitoring initialization process whenever **BrickAC.Monitor** is set to a value of 1.

Error Code	Description
-1	The assigned value is not accepted. Only a value of 1 or 0 can be assigned by user to this data structure.
-2	The **BrickAC.Monitor** was was called while either the **BrickAC.Reset** or **BrickAC.Config** process was active.
-3	The configuration process was attempted on incompatible hardware. No amplifier hardware with the matching Power Brick part number was detected.
-4	No Power Brick hardware was detected. This error is generated if incompatible output stage is detected.
-6	Packed data mode is detected (**PowerBrick[*i*].Chan[*j*].PackInData** > 0). This error is only generated if the monitor process is requested.
-7	The configuration process attempted used on incompatible hardware. No DPSGATE3 interface ASIC was detected.

Note

The monitored data in the Power Brick AC amplifier is provided to the controller in the low bits of the **Gate3[*i*].Chan[*j*].AdcAmp[*k*]** registers, below the current feedback values. This data cannot be read if two phases are "packed" into one register, so it is essential that **Gate3[*i*].Chan[*j*].PackInData** and **Gate3[*i*].Chan[*j*].PackOutData** are set to 0, disabling packed data and allowing the full registers to be read by the CPU.

Note

The monitoring process is automatically halted when either **BrickAC.Config** or **BrickAC.Reset** is set to 1 to update the amplifier configuration or reset the amplifier state, respectively, with **BrickAC.Monitor** set to 0. The monitoring process is *not* automatically resumed when the configuration or reset process is finished, so it must be explicitly restarted when one of these other processes is finished.

Example

```
BrickAC.Monitor = 1
MyEndTime = Sys.Time + 0.1            // 100 msec period
while (BrickAC.Monitor > 0 || MyEndTime < Sys.Time) {}
if (BrickAC.Monitor != 0)
{
        // take necessary action in case of a fault
}
// continue with script process
```

BrickAC.Reset

Description: Amplifier reset/fault-clear control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickAC.Reset acts as a flag for the Power PMAC firmware which controls the reset process of Power Brick AC amplifier. This reset process clears any latched faults, and loads the configuration into the active amplifier-control circuits based upon the **BrickAC** saved setup elements.

Setting **BrickAC.Reset** equal to 1 in a Script command starts the reset process as a background task on Power PMAC CPU. The value stays at this set value until either the reset process is completed, in which case the value of **BrickAC.Reset** is set to 0, or an error occurs in which case

the **BrickAC.Reset** value is set to a negative value indicating an error in the process. Please refer to **BrickAC.Config** for detailed information on the error code list.

It is strongly recommended for users to confirm the pass/fail status of the reset process whenever **BrickAC.Reset** is set to a value of 1.

Note

Setting **BrickAC.Reset** to 1 to start the amplifier configuration process automatically stops the amplifier monitoring process, and the monitoring process does *not* automatically resume when the configuration is completed. **BrickAC.Monitor** must be set to 1 again in the user Script application to resume the monitoring process.

Example

```
BrickAC.Reset = 1
while (BrickAC.Reset > 0) {}
if (BrickAC.Reset < 0)
{
        // take necessary action in case of a fault
}
// continue with script process
```

BrickLV. Non-Saved Data Structure Elements

The Power Brick LV is one class of the Power Brick family of intelligent amplifiers. It can support multiple types of brush, brushless and stepper motors, operating from a low-voltage DC input. Its registers can be accessed through **BrickLV.** data structure.

Note

The **BrickLV.** data structure elements documented in this section are software elements that are distinct from the **PowerBrick[*i*].** hardware data structure elements that form the control and status registers for the ASIC(s) in the Power Brick LV. (The **PowerBrick[*i*].** data structure is an "alias" for the **Gate3[*i*].** hardware data structure.)

BrickLV. Multi-Channel Setup Elements

Some aspects of the Brick LV amplifier are common to all channels on the board. The setup elements in this section affect all channels.

BrickLV.Config

Description: Amplifier configuration/initialization control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickLV.Config acts as a flag for the Power PMAC firmware which controls the initialization of Power Brick AC amplifier based upon the **BrickLV.** saved setup elements. The amplifier stage is not automatically configured at power-up, so the configuration process *must* be commanded explicitly by the user application before the amplifier stage can be used.

Setting **BrickLV.Config** to 1 in a Script command starts the initialization process as a background task on Power PMAC CPU. The element stays at the set value until either the initialization process is successfully completed, in which case the value of **BrickLV.Config** is set to 0, or until a configuration error is detected, in which case the **BrickLV.Config** value is set to a negative value indicating the error in the process.

The following list shows the error codes which can be encountered:

Error Code	Description
-1	The assigned value is not accepted. Only a value of 1 or 0 can be assigned by user to this data structure.
-2	The **BrickLV.Monitor** was was called while either the **BrickLV.Reset** or **BrickLV.Config** process was active.
-3	The configuration process was attempted on incompatible hardware. No amplifier hardware with the matching Power Brick part number was detected.
-4	No Power Brick hardware was detected. This error is generated if incompatible output stage is detected.
-7	The configuration process attempted used on incompatible hardware. No DPSGATE3 interface ASIC was detected.

If **BrickLV.Config** is set to 1 in an on-line command, there will be a text response indicating whether the configuration completed correctly or not, and if not, what the error was.

It is strongly recommended for users to confirm the pass/fail status of the initialization process whenever **BrickLV.Config** is set to a value of 1.

Note

While setting **BrickLV.Config** to 1 as part of the standard system initialization process after power-up will load the configuration parameters into the amplifier control circuitry, it is recommended instead to set **BrickLV.Reset** to 1, which will not only load the configuration parameters, but clear any faults that may have occurred due to power-on transient conditions.

Note

Setting **BrickLV.Config** to 1 to start the amplifier configuration process automatically stops the amplifier monitoring process, and the monitoring process does *not* automatically resume when the configuration is completed. **BrickLV.Monitor** must be set to 1 again in the user Script application to resume the monitoring process.

Example

```
BrickLV.Config = 1
while (BrickLV.Config > 0) {}
if (BrickLV.Config != 0)
{
        // take necessary action in case of a fault
}
// continue with script process
```

BrickLV.Monitor

Description: Amplifier status monitoring update control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickLV.Monitor acts as a flag for the Power PMAC firmware which controls the execution of Power Brick AC amplifier status monitoring background task. This task updates the **BrickLV.** status elements at constant period set by saved setup element **BrickLVC.MonitorPeriod**.

If **BrickLV.Monitor** is set to its power-on default value of 0, there is no updating of the **BrickLV.** status elements. In this mode none of these element values are updated and they maintain their last updated value until next reset or power cycle.

Setting **BrickLV.Monitor** equal to 1 in a Script command starts the background **BrickLV.** status update task at a period set by **BrickLV.MonitorPeriod**. The element stays at the set value until either the user application sets the value to 0, which stops the update process, or the user application commands an initialization or reset process by setting **BrickLV.Config** or **BrickLV.Reset** to a value of 1.

If an error occurs during the monitor process, the **BrickLV.Monitor** value is set to a negative value indicating an error in the process. The following table shows the errors that can be reported. It is strongly recommended for users to confirm the pass/fail status of the monitoring initialization process whenever **BrickLV.Monitor** is set to a value of 1.

Error Code	Description
-1	The assigned value is not accepted. Only a value of 1 or 0 can be assigned by user to this data structure.
-2	The **BrickLV.Monitor** was was called while either the **BrickLV.Reset** or **BrickLV.Config** process was active.
-3	The configuration process was attempted on incompatible hardware. No amplifier hardware with the matching Power Brick part number was detected.
-4	No Power Brick hardware was detected. This error is generated if incompatible output stage is detected.
-6	Packed data mode is detected (**PowerBrick[*i*].Chan[*j*].PackInData** > 0). This error is only generated if the monitor process is requested.
-7	The configuration process attempted used on incompatible hardware. No DPSGATE3 interface ASIC was detected.

Note

The monitored data in the Power Brick AC amplifier is provided to the controller in the low bits of the **Gate3[*i*].Chan[*j*].AdcAmp[*k*]** registers, below the current feedback values. This data cannot be read if two phases are "packed" into one register, so it is essential that **Gate3[*i*].Chan[*j*].PackInData** and **Gate3[*i*].Chan[*j*].PackOutData** are set to 0, disabling packed data and allowing the full registers to be read by the CPU.

Note

The monitoring process is automatically halted when either **BrickLV.Config** or **BrickLV.Reset** is set to 1 to update the amplifier configuration or reset the amplifier state, respectively, with **BrickLV.Monitor** set to 0. The monitoring process is *not* automatically resumed when the configuration or reset process is finished, so it must be explicitly restarted when one of these other processes is finished.

Example

```
BrickLV.Monitor = 1
MyEndTime = Sys.Time + 0.1            // 100 msec period
while (BrickLV.Monitor > 0 || MyEndTime < Sys.Time) {}
if (BrickLV.Monitor < 0)
{
        // take necessary action in case of a fault
}
// continue with script process
```

BrickLV.Reset

Description: Amplifier reset/fault-clear control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickLV.Reset acts as a flag for the Power PMAC firmware which controls the reset process of Power Brick LV amplifier. This reset process clears any latched faults, and loads the configuration into the active amplifier-control circuits based upon the **BrickLV** saved setup elements.

Setting **BrickLV.Reset** equal to 1 in a Script command starts the reset process as a background task on Power PMAC CPU. The value stays at this set value until either the reset process is completed, in which case the value of **BrickLV.Reset** is set to 0, or an error occurs in which case

the **BrickLV.Reset** value is set to a negative value indicating an error in the process. Please refer to **BrickLV.Config** for detailed information on the error code list.

It is strongly recommended for users to confirm the pass/fail status of the reset process whenever **BrickLV.Reset** is set to a value of 1.

Note

Setting **BrickLV.Reset** to 1 to start the amplifier configuration process automatically stops the amplifier monitoring process, and the monitoring process does *not* automatically resume when the configuration is completed. **BrickLV.Monitor** must be set to 1 again in the user Script application to resume the monitoring process.

Example

```
BrickLV.Reset = 1
while (BrickLV.Reset > 0) {}
if (BrickLV.Reset < 0)
{
        // take necessary action in case of a fault
}
// continue with script process
```

BufIo[*i*]. Buffered I/O Non-Saved Setup Data Structure Elements

The **BufIo[*i*]** data structure contains several elements the user can write to for the buffered input/output functionality. The values of these elements are not saved to non-volatile memory.

BufIo[*i*].ForceInOn

Description: Input override "on" register

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Power-on default: $0

BufIo[*i*].ForceInOn is a 32-bit register corresponding to the input register specified by **BufIo[*i*].pIn** that permits the user to override the value of one or more bits in the actual input word, forcing that bit value to 1 in the holding register **BufIo[*i*].In**. The primary uses of this forcing functionality are simulation in the initial software development, before installation on the full machine, and later debugging of the software.

This functionality is enabled if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pIn** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this input. If input filtering is enabled by setting **BufIo[*i*].InScans** greater than 0, **BufIo[*i*].In** will reflect changes in the filtered forced bits, not necessarily the raw forced bits for the scan.

In operation, each bit in the actual input register is compared to the corresponding bit in **BufIo[*i*].ForceInOn**. If the bit of **BufIo[*i*].ForceInOn** is 0, the bit from the input word is passed on unchanged with a value of either 0 or 1. If the bit of **BufIo[*i*].ForceInOn** is 1, the bit in the resulting word that is passed on will be set to 1, regardless of the value in the actual input word.

This comparison is done before a similar comparison to **BufIo[*i*].ForceInOff** is performed (which means that the "forcing off" function can override the "forcing on" function) and before the filtering function, if that is enabled. If filtering is enabled, a bit of **BufIo[*i*].ForceInOn** would have to be set to 1 for multiple scans before its effect is seen in the resulting **BufIo[*i*].In**.

An individual bit of **BufIo[*i*].ForceInOn** can be accessed with the syntax **BufIo[*i*].ForceInOn.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].ForceInOn** can be accessed with the syntax **BufIo[*i*].ForceInOn.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].ForceInOn is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].ForceInOff

Description: Input override "off" register

Range: $0 .. $FFFFFFFF (0 .. 2^{32} – 1)

Units: Bit field

Power-on default: $0

BufIo[*i*].ForceInOff is a 32-bit register corresponding to the input register specified by **BufIo[*i*].pIn** that permits the user to override the value of one or more bits in the actual input word, forcing that bit value to 0 in the holding register **BufIo[*i*].In**. The primary uses of this forcing functionality are simulation in the initial software development, before installation on the full machine, and later debugging of the software.

This functionality is enabled if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pIn** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this input. If input filtering is enabled by setting **BufIo[*i*].InScans** greater than 0, **BufIo[*i*].In** will reflect changes in the filtered forced bits, not necessarily the raw forced bits for the scan.

In operation, each bit in the actual input register is compared to the corresponding bit in **BufIo[*i*].ForceInOff**. If the bit of **BufIo[*i*].ForceInOff** is 0, the bit from the input word is passed on unchanged with a value of either 0 or 1. If the bit of **BufIo[*i*].ForceInOff** is 1, the bit in the resulting word that is passed on will be set to 0, regardless of the value in the actual input word.

Note that setting a bit to 1 in **BufIo[*i*].ForceInOff** means that the value of the bit in the resulting word is 0.

This comparison is done after a similar comparison to **BufIo[*i*].ForceInOn** is performed (which means that the "forcing off" function can override the "forcing on" function) and before the filtering function, if that is enabled. If filtering is enabled, a bit of **BufIo[*i*].ForceInOff** would have to be set to 1 for multiple scans before its effect is seen in the resulting **BufIo[*i*].In**.

An individual bit of **BufIo[*i*].ForceInOff** can be accessed with the syntax **BufIo[*i*].ForceInOff.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].ForceInOff** can be accessed with the syntax **BufIo[*i*].ForceInOff.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].ForceInOff is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].ForceOutOn

Description: Output override "on" register

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Power-on default: $0

BufIo[*i*].ForceOutOn is a 32-bit register corresponding to the output register specified by **BufIo[*i*].pOut** that permits the user to override the value of one or more bits in the output holding word **BufIo[*i*].Out**, forcing that bit value to 1 in the actual output register addressed by **BufIo[*i*].pOut**. The primary uses of this forcing functionality are simulation in the initial software development, before installation on the full machine, and later debugging of the software.

This functionality is enabled if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pOut** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this output.

In operation, each bit in the output holding register **BufIo[*i*].Out** is compared to the corresponding bit in **BufIo[*i*].ForceOutOn**. If the bit of **BufIo[*i*].ForceOutOn** is 0, the bit from the output holding word is passed on unchanged with a value of either 0 or 1. If the bit of **BufIo[*i*].ForceOutOn** is 1, the bit in the resulting word that is passed on will be set to 1, regardless of the value in the output holding register.

This comparison is done before a similar comparison to **BufIo[*i*].ForceOutOff** is performed (which means that the "forcing off" function can override the "forcing on" function).

An individual bit of **BufIo[*i*].ForceOutOn** can be accessed with the syntax **BufIo[*i*].ForceOutOn.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].ForceOutOn** can be accessed with the syntax **BufIo[*i*].ForceOutOn.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].ForceOutOn is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].ForceOutOff

Description: Output override "off" register

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Power-on default: $0

BufIo[*i*].ForceOutOff is a 32-bit register corresponding to the output register specified by **BufIo[*i*].pOut** that permits the user to override the value of one or more bits in the output holding word **BufIo[*i*].Out**, forcing that bit value to 0 in the actual output register addressed by **BufIo[*i*].pOut**. The primary uses of this forcing functionality are simulation in the initial software development, before installation on the full machine, and later debugging of the software.

This functionality is enabled if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pOut** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this output.

In operation, each bit in bit in the output holding register **BufIo[*i*].Out** is compared to the corresponding bit in **BufIo[*i*].ForceOutOff**. If the bit of **BufIo[*i*].ForceOutOff** is 0, the bit from the output holding word is passed on unchanged with a value of either 0 or 1. If the bit of **BufIo[*i*].ForceOutOff** is 1, the bit in the resulting word that is passed on will be set to 0, regardless of the value in the actual input word.

Note that setting a bit to 1 in **BufIo[*i*].ForceOutOff** means that the value of the bit in the resulting word is 0.

This comparison is done after a similar comparison to **BufIo[*i*].ForceOutOn** is performed (which means that the "forcing off" function can override the "forcing on" function.

An individual bit of **BufIo[*i*].ForceOutOff** can be accessed with the syntax **BufIo[*i*].ForceOutOff.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].ForceOutOff** can be accessed with the syntax **BufIo[*i*].ForceOutOff.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].ForceOutOff is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].Out

Description: Buffered output holding register

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Power-on default: $0

BufIo[*i*].Out is the buffered 32-bit output holding register for the output register specified by **BufIo[*i*].pOut**. Its contents will automatically be copied to the matching output register if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pOut** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this output. User PLC programs will write to this holding register in memory, not the actual output register.

If these conditions are true, then at the end of each scan, Power PMAC will copy the contents of **BufIo[*i*].Out** to the output register at the address specified by **BufIo[*i*].pOut**.

An individual bit of **BufIo[*i*].Out** can be accessed with the syntax **BufIo[*i*].Out.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].Out** can be accessed with the syntax **BufIo[*i*].Out.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].Out is new in V2.1 firmware, released 1st quarter 2016.

CamTable[*m*]. Non-Saved Setup Data Structure Elements

This section describes setup elements in the electronic cam table data structure whose values are not copied to flash memory on a **save** command.

CamTable[*m*].Disable

Description: Cam table gradual disable control

Range: 0 .. 255

Units: Servo cycles

Power-on default: 0

CamTable[*m*].Disable permits the user to disable the action of the cam table in a gradual and controlled fashion. When **CamTable[*m*].Disable** is set to a value greater than 0 for an enabled table, the disabling process is automatically started.

Each subsequent servo cycle, the position output of the cam table in **Motor[*x*].CompDesPos** for the target motor is reduced toward 0.0 by the magnitude of **CamTable[*m*].SlewPosOffset**. This will create actual motion of the target motor. When this position value reaches zero, the value of **CamTable[*m*].Disable** is then decremented by 1 each servo cycle until it reaches 0.

When the value of **CamTable[*m*].Disable** reaches 0, **CamTable[*m*].Enable** is automatically set to 0 to complete the process. The possibility of multiple servo cycles in this second stage of the process permits some settling time before the state change that permits the user to detect the end of the entire process and proceed with the next function of the machine. Even if real settling time is not required, it is recommended for the user to set **CamTable[*m*].Disable** to at least 2 to assure proper convergence.

Note that the table "torque" outputs (if any) in **Motor[*x*].CompDac** for the target motor, and any discrete outputs in the register specified by **CamTable[*m*].pOut** are not affected by this disabling process. If the user wishes to clear these, that should be done explicitly at the end of the disabling process.

The action of the table can also be disabled simply by setting **CamTable[*m*].Enable** directly to 0, but that action will leave the present value of **Motor[*x*].CompDesPos** for the target motor unchanged, and the disabling process will create no motion.

CamTable[*m*].Disable is new in V2.0.2 firmware, released 2nd quarter 2015.

CamTable[*m*].Enable

Description: Cam table enable control

Range: 0 .. 3

Units: Bit field

Power-on default: 0

CamTable[*m*].Enable specifies whether the cam table calculations will be performed or not. If it is set to its power-on/reset default value of 0, the table calculations will not be executed. If it is set to a value of 1, 2, or 3, the table calculations will be executed. **CamTable[*m*].Enable** is automatically set to 0 on power-up/reset for all tables. An explicit command is then required to enable a table.

For a "returning" table, the operation of the table is identical for any of the non-zero values. For a "non-returning" table, the specific non-zero value determines how the table establishes synchronization when first enabled.

In the servo cycle where Power PMAC first sees a non-zero value in **CamTable[*m*].Enable** for a non-returning table, enabling the table it determines the direction the calculated cam position value will slew from the initial position to the synchronized position. The three possibilities are:

1: Slew always in the positive direction to synchronization

2: Slew always in the negative direction to synchronization

3: Slew in the shortest direction to synchronization

If **CamTable[*m*].Enable** has been set greater than 0, enabling the table, when it is set to 0, its position, torque, and direct-output command values are simply left with their last values.

CamTable[*m*].Enable is only used if global saved setup element **Sys.CamEnable** is set to a value greater than ***m***. If this is the case, **CamTable[*m*].Enable** is checked every servo cycle.

CamTable[*m*].PosOffset

Description: Cam table desired slave position offset

Range: Floating-point

Units: Target motor position units

Power-on default: 0.0

CamTable[*m*].PosOffset specifies the commanded target-motor position offset for the cam table. This is the steady-state value to be added to the position value calculated from the table **PosData[*i*]** entries each servo cycle after being multiplied by **CamTable[*m*].PosSf** and written to the target motor's **Motor[*x*].CompDesPos** position register.

The main use of **CamTable[*m*].PosOffset** is for Power PMAC to use automatically to specify which cycle of a non-returning table to select when the table is enabled. In the servo cycle when the table is enabled, Power PMAC will automatically set **PosOffset** to *N**(**PosData[*N*x]-PosData[0]**), where the difference between the last and first points is the size of the table cycle, and *N* is based on the cycle number chosen. (For a returning table, *N* can always be considered to be zero, so **PosOffset** will be set to 0.) This overwrites any value that is present in the element at that time (such as a user-specified offset from a previous period when the table was enabled).

If **CamTable[*m*].Enable** is set to 1 to enable the table, *N* is chosen so that the target motor will always move in the positive direction to establish synchronization. If it is set to 2, *N* is chosen so it will always move in the negative direction to establish synchronization. If it is set to 3, *N* is chosen so it will move in the direction that has the shortest direction to establish synchronization.

The actual value added in a given servo cycle to the value calculated from table entries is found in status element **CamTable[*m*].ActivePosOffset**. If the value of **CamTable[*m*].PosOffset** is changed, the value of **ActivePosOffset** will change each servo cycle by the value of saved setup element **CamTable[*m*].SlewPosOffset** until the new desired value is reached. This allows for controlled changes in the actual position offset used.

It is also possible for the user to specify the value of **CamTable[*m*].PosOffset** directly when the table is enabled, changing the offset. However, if the table is then disabled and re-enabled, Power PMAC will automatically set the value of **PosOffset** as described above, overwriting any value the user may have previously set. Generally, if the user wants to create a different effective offset, this will be done by setting saved setup element **CamTable[*m*].PosBias** (new in V2.0 firmware released 1st quarter 2015), or by commanding trajectory motion, such as through a jogging move.

Cid[*j*]. Non-Saved Setup Data Structure Elements

This section describes useful setup elements within the card identification data structure whose values are not copied to flash memory on a **save** command.

The index values ***j*** can have a range of 0 to 63. These map to the index values of the different types of accessories according to the following table:

Cid[*j*]	**Gate*xx*[*i*]**	**Cid[*j*]**	**Gate*xx*[*i*]**	**Cid[*j*]**	**Gate*xx*[*i*]**	**Cid[*j*]**	**Gate*xx*[*i*]**
Cid[0]	*{reserved}*	Cid[16]	*{reserved}*	Cid[32]	*{reserved}*	Cid[48]	*{reserved}*
Cid[1]	*{reserved}*	Cid[17]	*{reserved}*	Cid[33]	*{reserved}*	Cid[49]	*{reserved}*
Cid[2]	Gate1[4]	Cid[18]	Gate1[8]	Cid[34]	Gate1[12]	Cid[50]	Gate1[16]
Cid[3]	Gate1[6]	Cid[19]	Gate1[10]	Cid[35]	Gate1[14]	Cid[51]	Gate1[18]
Cid[4]	Gate2[0]	Cid[20]	Gate2[1]	Cid[36]	Gate2[2]	Cid[52]	Gate2[3]
Cid[5]	Gate2[4]	Cid[21]	Gate2[5]	Cid[37]	Gate2[6]	Cid[53]	Gate2[7]
Cid[6]	Gate2[8]	Cid[22]	Gate2[9]	Cid[38]	Gate2[10]	Cid[54]	Gate2[11]
Cid[7]	Gate2[12]	Cid[23]	Gate2[13]	Cid[39]	Gate2[14]	Cid[55]	Gate2[15]
Cid[8]	Dpr[0]	Cid[24]	Dpr[2]	Cid[40]	Dpr[4]	Cid[56]	Dpr[6]
Cid[9]	Dpr[1]	Cid[25]	Dpr[3]	Cid[41]	Dpr[5]	Cid[57]	Dpr[7]
Cid[10]	Gate1[5]	Cid[26]	Gate1[9]	Cid[42]	Gate1[13]	Cid[58]	Gate1[17]
Cid[11]	Gate1[7]	Cid[27]	Gate1[11]	Cid[43]	Gate1[15]	Cid[59]	Gate1[19]
Cid[12]	GateIo[0]	Cid[28]	GateIo[4]	Cid[44]	GateIo[8]	Cid[60]	GateIo[12]
Cid[13]	GateIo[1]	Cid[29]	GateIo[5]	Cid[45]	GateIo[9]	Cid[61]	GateIo[13]
Cid[14]	GateIo[2]	Cid[30]	GateIo[6]	Cid[46]	GateIo[10]	Cid[62]	GateIo[14]
Cid[15]	GateIo[3]	Cid[31]	GateIo[7]	Cid[47]	GateIo[11]	Cid[63]	GateIo[15]

Cid[*j*].PartCtrl[*k*]

Description: Card identification IC information

Range: $00 .. $FF

Units: Bit field

Power-on default: Card dependent

Cid[*j*].PartCtrl[*k*] is an 8-bit element that contains both setup and status information about the card. The index ***k*** can range from 0 to 3.

Bits 0 – 3 of **PartCtrl[0]** and **PartCtrl[1]**, and bits 0 – 4 of **PartCtrl[2]** and **PartCtrl[3]** contain factory-set information about the card. Power PMAC automatically processes this information into **Cid[*j*]** status elements **Cid[*j*].num**, **Cid[*j*].opt**, **Cid[*j*].rev**, and **Cid[*j*].ven**, and into status elements for accessories such as **PartNum**, **PartOpt**, **PartRev**, and **PartType**.

Bit 4 of **PartCtrl[1]** is the bank-select control bit. The value written into this bit determines whether "Bank 0" information or "Bank 1" information is accessed when using **PartCtrl[*k*]**.

The function of the upper bits of **PartCtrl[*k*]** is card-dependent.

Cid[*j*].PartData[*k*]

Description: Card identification IC information

Range: $00 .. $FF

Units: Bit field

Power-on default: Card dependent

Cid[*j*].PartData[*k*] is a 16-bit element that contains setup information for the card. The index ***k*** can range from 0 to 3.

The function of the bits of **PartData[*k*]** is card-dependent. On the ACC-5E, these bits set the direction of the byte-wide buffer ICs for the digital I/O on the board.

Clipper[*i*]. Non-Saved Data Structure Elements

The **Clipper[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the Power Clipper controller board are listed under the **Gate3[*i*]** data structure, below.

Coord[*x*]. Non-Saved Setup Data Structure Elements

This section describes useful setup elements within the coordinate-system data structure whose values are not copied to flash memory on a **save** command.

Coord[*x*].DesTimeBase

Description: Desired time base value

Range: Floating-point

Units: Milliseconds

Power-on default: = **Sys.ServoPeriod**

Coord[*x*].DesTimeBase specifies the user's target "time base" value for the coordinate system if it is specified to use this element by setting saved addressing setup element **Coord[*x*].pDesTimeBase** to the address of this element (**Coord[*x*].DesTimeBase.a**). This is the factory default addressing.

The value in **Coord[*x*].DesTimeBase** can be set using coordinate-system-specific on-line **%** commands. In this case, the value is set according to the equation:

$$Coord[x].DesTimeBase = \frac{\%value}{100} * Sys.ServoPeriod$$

For example, a **%50** command would set **Coord[*x*].DesTimeBase** to **Sys.ServoPeriod** / 2.

In a buffered program command, a value is assigned directly to **Coord[*x*].DesTimeBase**. For example, to set the time base value to values of 0 to 150% from a 4-bit switch that produces numbers from 0 to 15, the following buffered program command could be used:

Coord[*x*].DesTimeBase = **Sys.ServoPeriod** * SwitchValue / 10;

When **Coord[*x*].DesTimeBase** is set directly, values are not limited to the 0 – 200% range as they are with **%*{constant}*** on-line commands.

The actual time base value used each servo cycle to increment the time elapsed in the trajectory is in status element **Coord[*x*].TimeBase**. If the value in **Coord[*x*].DesTimeBase** differs from this value, **TimeBase** is incremented by the magnitude of saved setup element **Coord[*x*].TimeBaseSlew** toward the value of **DesTimeBase** until they match.

Power PMAC provides full support for negative time base values, permitting reverse trajectory execution through motor "trace" buffers.

Coord[x].InvTimeMode

Description: Inverse time feedrate mode control

Range: 0 .. 3

Units: Enumeration

Power-on default: 0

Non-saved setup element **Coord[*x*].InvTimeMode** determines the feedrate specification mode for linear and circle mode blended moves in the coordinate system. It does this by specifying how the value in the **F** (feedrate) program command is interpreted. In any of the modes, the negative of the value in the **F** command is stored in the element **Coord[*x*].Tm**, with the sign indicating it came from an **F** command and not a **tm** (move time) command.

If **Coord[*x*].InvTimeMode** is set to its power-on default value of 0, inverse time mode is disabled, and the value in the **F** program command is interpreted as a vector speed in axis length units per coordinate-system time units for both linear and circle mode moves. This is typically known as "feedrate" mode.

The axis units are determined by the axis definitions in the coordinate system, with only the "vector feedrate axes" are involved in the vector speed calculations; these are X, Y, and Z by default, but this can be changed with the **frax** command. The time units are set by saved setup element **Coord[*x*].FeedTime**; if this is set to the default of 1000, the time units are seconds, or if it is set to 60,000, the time units are minutes. In this mode, the time for a move is calculated as the vector distance of the feedrate axes divided by the specified vector speed from the most recent **F** command.

If **Coord[*x*].InvTimeMode** is set to 1, inverse time mode is enabled, and the value in the **F** program command is interpreted as the inverse of the commanded move time for both linear and circle mode moves. In both move modes, the value can be considered to be the velocity divided by the distance for an axis. The time units are set by saved setup element **Coord[*x*].FeedTime**; if this is set to the default of 1000, the time units are seconds, or if it is set to 60,000, the time units are minutes. In this mode, the value from the most recent **F** command is divided into the value of **Coord[*x*].FeedTime** and the result is used as the commanded move time in milliseconds.

If **Coord[*x*].InvTimeMode** is set to 2, inverse time mode is enabled, and the value in the **F** program command is interpreted as the inverse of the commanded move time for linear mode moves. In this move mode, the value can be considered to be the velocity divided by the distance for an axis. In this case, the value from the most recent **F** command is divided into the value of **Coord[*x*].FeedTime** and the result is used the commanded move time in milliseconds.

With **Coord[*x*].InvTimeMode** set to 2, for circle mode moves the value in the **F** command can be considered to be the velocity divided by the X/Y/Z circle radius. (If the radius varies during the move, the starting radius is used.) In this case, the value in the **F** command is divided into the value of **Coord[*x*].FeedTime** and this intermediate result is multiplied by the angle subtended by the arc move (in radians). This result is then used as the commanded move time in milliseconds.

If **Coord[*x*].InvTimeMode** is set to 3, operation is the same as when it is set to 2, except the time for circle mode moves is based on the XX/YY/ZZ circle radius.

When **Coord[*x*].InvTimeMode** is changed between a zero and non-zero value, the mode will change for the next feedrate-based move (even if the most recent **F** command was executed before the mode was changed). The setting of **Coord[*x*].InvTimeMode** does not affect the interpretation of **tm** (move time) command values for either linear or circle mode moves, for which the value in the command is directly stored in the element **Coord[*x*].Tm**.

Inverse time mode is commonly implemented in the G93 code in RS-274 CNC code. The G94 code is commonly used to specify feedrate mode.

Some controllers require a new **F** command for every programmed move while in inverse-time mode. Power PMAC does not require this, but it is strongly suggested that the user do so. Unexpected action can occur when an old move time is used for a new move.

Cutter radius compensation may not be used with the coordinate system in inverse time mode.

Examples

The following examples show how move times are computed for simple linear and circle mode moves in feedrate and inverse time modes.

```
Coord[1].FeedTime = 60000;                  // Time units of minutes
frax(X,Y,Z);                                // Vector feedrate axes

Coord[1].InvTimeMode = 0;                   // Feedrate mode (G94)
inc; linear; X10 Y10 Z5 F50;                // (G91 G01)
```

Move distance = sqrt($10^2 + 10^2 + 5^2$) = 15.00; Move time = 15.00 / 50 = 0.300 min = 18,000 msec

```
Coord[1].InvTimeMode = 0;                   // Feedrate mode (G94)
inc; circle1; X10 Y10 Z5 I10 F50;           // (G91 G02)
```

Move distance = sqrt($(10*Pi/2)^2+5^2$) = 16.48; Move time = 16.48 / 50 = 0.329 min = 19,781 msec

```
Coord[1].InvTimeMode = 1;                   // Inverse time mode (G93)
inc; linear; X10 Y10 Z5 F50;                // (G91 G01)
```

Move time = 60,000 / 50 = 12,000 msec

```
Coord[1].InvTimeMode = 1;                   // Inverse time mode (G94)
inc; circle1; X10 Y10 Z5 I10 F50;           // (G91 G02)
```

Move time = 60,000 / 50 = 12,000 msec

```
Coord[1].InvTimeMode = 2;                   // Inverse time mode (G94)
inc; circle1; X10 Y10 Z5 I10 F50;           // (G91 G02)
```

Move time = (60,000 / 50) * (Pi / 2) = 18,849 msec

Coord[x].OnceNoBlend

Description: Non-modal blend-disable control flag

Range: 0 .. 255

Units: Programmed moves

Power-on default: 0

Coord[*x*].OnceNoBlend permits the disabling of blending at the end of the next linear, circle, or spline-mode move(s). If it is set to a value greater than 0 when one of these moves is calculated, commanded motion will be brought to a stop at the end of the move. Once the move is calculated, **Coord[*x*].OnceNoBlend** is automatically decremented by 1.

If **Coord[*x*].OnceNoBlend** is set to 0, blending is not disabled by this element, but can still be disabled at the end of the next move by modal elements such as **Coord[*x*].NoBlend** and **Coord[*x*].CornerBlendBp**.

If **Coord[*x*].OnceNoBlend** is set greater than 0, blending at the end of the next move is disabled regardless of the setting of modal elements **Coord[*x*].NoBlend** and **Coord[*x*].CornerBlendBp**. If **Coord[*x*].InPosTimeOut** is set to its default value of 0, the commanded trajectory for the following move will start immediately after the commanded trajectory for this move is ended, regardless of the actual positions of the motors. If **Coord[*x*].InPosTimeOut** is greater than 0, all motors in the coordinate system must be "in position" at the end of the move before the following move is started.

Setting **Coord[*x*].OnceNoBlend** is mainly useful in CNC-style programs, where the **G09** non-modal "exact stop" code is used to disable blending at the end of the following commanded move. The subroutine implementing this **G09** code can simply set **Coord[*x*].OnceNoBlend** to 1.

Coord[*x*].OnceNoBlend is new in version 2.2 firmware, released 3rd quarter 2016. At its default value of 0, operation is compatible with earlier firmware versions.

Example

```
open subprog 1000                    // G-code subprogram
local ThisCs;                        // Number of executing coord sys
…
N9000:                               // G09 (exact stop) subroutine
ThisCs = Ldata.Coord;                // Set to number of executing C.S.
Coord[ThisCs].OnceNoBlend = 1;       // Next move not blended at end
return;
…
```

Coord[*x*].Q[*i*]

Description: Q-variable array element

Range: Floating-point

Units: User-determined

Power-on default: 0

Coord[*x*].Q[*i*] is the "***i***th" Q-variable array element for the coordinate system. Q-variables are coordinate-system-specific global double-precision floating-point user variables. Index values ***i*** can range from 0 to 8,191 in each coordinate system. This array provides an alternate method for accessing Q-variables.

Coord[*x*].SegOverride

Description: Segmentation feedrate override command value

Range: Non-negative floating-point

Units: Ratio to real-time

Power-on default: 1.0

Coord[*x*].SegOverride specifies the user's commanded value for "feedrate override" at the segmentation, or coarse-interpolation, stage of trajectory generation for segmented moves (**`linear`**, **`circle`**, and **`pvt`**, with **Coord[*x*].SegMoveTime** > 0). It is a normalized value (not a percentage), so a value of 1.0 commands "real time" execution.

In segmented moves, every **Coord[*x*].SegMoveTime** milliseconds of actual time, Power PMAC computes the next segment's position. In doing so, it advances the numerical time of the motion equations by the product of **Coord[*x*].SegMoveTime** and the override value. For example, if **Coord[*x*].SegMoveTime** is set to 5.0 msec, and the override value is set to 1.5, each segment point computed and executed 5.0 msec apart advances the motion equations by 5.0 * 1.5 = 7.5 msec, so the move is executed at 150% of the programmed speed.

When the commanded value **Coord[*x*].SegOverride** is changed, the internal value **Coord[*x*].ActSegOverride** that is actually used to compute the time advance in the move equations for a segment is changed by the amount in saved setup element **Coord[*x*].SegOverrideSlew** each segment until the new commanded value is reached. This prevents step changes in the resulting velocity.

Coord[x].CC3Data[i]. 3D Cutter Compensation Elements

The **Coord[*x*].CC3Data[*i*].** substructure contains elements specifying the tool-nose geometry for 3D cutter (tool) radius compensation.

Coord[*x*].CC3Data[*i*].CutRadius

Description: 3D cutter compensation tool-tip segment local radius

Range: Non-negative floating-point

Units: Linear axis user units

Power-on default: 0.0

Coord[*x*].CC3Data[*i*].CutRadius specifies, for the tool-tip geometry to be used for three-dimensional cutter radius compensation in Coordinate System ***x***, the radius of the arc section of the table with index ***i***. It is expressed in the units of the base X, Y, and Z axes for the coordinate system, almost always millimeters or inches. It is not rescaled if the axis units are rescaled through use of a transformation matrix.

There can be up to 16 arc sections in the tool-tip table, with indices ***i*** from 0 to 15. Values of **Coord[*x*].CC3Data[*i*].NdotT** and **Coord[*x*].CC3Data[*i*].CutRadius** must be specified for each defined arc section. As these values are entered, Power PMAC will automatically set values for the **ToolOffset** and **ToolRadius** elements for these arc sections.

A value of 0.0 for **Coord[*x*]. CC3Data[*i*].CutRadius** can be specified, defining an arc section of zero size. Specifying this zero value creates an instant change ("corner") in the angle of the tool tip.

For more details, refer to the section *Three-Dimensional Cutter Radius Compensation* in the chapter *Power PMAC Move Mode Trajectories* in the Power PMAC User's Manual.

Coord[*x*].CC3Data[*i*].NdotT

Description: 3D cutter compensation tool-tip segment start angle cosine

Range: -1.0 .. 1.0

Units: Angle cosine

Power-on default: 0.0 (***i*** = 0 .. 15), 1.0 (***i*** = 16)

Coord[*x*].CC3Data[*i*].NdotT specifies, for the tool-tip geometry to be used for three-dimensional cutter radius compensation in Coordinate System ***x***, the cosine of the starting angle of the arc section of the table with index ***i***. This angle is defined as the angle between the tool's axis of rotation leading away from the tool tip and the radial line for the arc section furthest from the tool tip leading from the tool surface toward the tool center.

The cosine of this angle is equivalent for this point on the tool tip to the dot product of the part surface normal vector ***N*** specified with the **nxyz** command in the part program and the tool-tip orientation vector ***T*** specified with the **txyz** command in the part program – hence the element name **NdotT**.

The value of this element must be a valid cosine value, and so must be within the range of -1.0 to +1.0; an attempt to enter a value outside of this range using a script command will be rejected with an error. The value of this element for an arc section ***i*** must be greater than that for any previous arc sections (with lesser index values).

The value of **Coord[*x*].CC3Data[0].NdotT** specifies the starting angle for the first arc section of the table. A value of 0.0 in this element specifies a smooth (tangent) blend between the shaft of the tool cylinder and the start of the tool tip. Non-zero values are possible here, but imply a sharp "corner" between the shaft and the start of the tool tip. A negative value implies a "bulbous" tool tip that has a larger peak radius than the shaft.

There can be up to 16 arc sections in the tool-tip table, with indices ***i*** from 0 to 15. Values of **Coord[*x*].CC3Data[*i*].NdotT** and **Coord[*x*].CC3Data[*i*].CutRadius** must be specified for each defined arc section. As these values are entered, Power PMAC will automatically set values for the **ToolOffset** and **ToolRadius** elements for these arc sections.

A table with ***n*** defined arc sections, with indices of 0 to ***n***-1, must have a setting for **Coord[*x*].CC3Data[*n*].NdotT** of 1.0. This cosine value specifies a 0° angle (parallelism) between the ending radial line of arc section ***n***-1 and the tool center of rotation, which specifies that the tool tip at the end of this section is perpendicular to the center of rotation. In general, this means a "smooth" tool-center tip. If a pointed tool-center tip is desired (an unusual case), a last arc section with a value of 0.0 for **Coord[*x*].CC3Data[*i*].CutRadius** must be specified.

Coord[*x*].CC3Data[16].NdotT, which specifies the end angle of the last possible arc section (***i*** = 15) is set by default to 1.0 to provide the proper end to any table, and the user is not permitted to change this value.

For more details, refer to the section *Three-Dimensional Cutter Radius Compensation* in the chapter *Power PMAC Move Mode Trajectories* in the Power PMAC User's Manual.

Coord[*x*].CC3Data[0].ToolOffset

Description: 3D cutter compensation first-segment axial offset

Range: Floating-point

Units: Linear axis user units

Power-on default: 0.0

Coord[*x*]. CC3Data[0].ToolOffset specifies, for the tool-tip geometry to be used for three-dimensional cutter radius compensation in Coordinate System ***x***, the signed axial distance from the tool center point to the starting point of the first defined arc section of the tool tip. This starting point, defined by **Coord[*x*].CC3Data[0].ToolOffset** and

Coord[*x*].CC3Data[0].ToolRadius, is the reference point for the other sections of the tool-tip table.

Coord[*x*]. CC3Data[0].ToolOffset is expressed in the units of the base X, Y, and Z axes for the coordinate system, almost always millimeters or inches. It is not rescaled if the axis units are rescaled through use of a transformation matrix. If it is a positive value, the reference point for the table is "above" the tool center point (farther away from the tool tip than the center point). If it is a negative value, the reference point is "below" the tool center point (closer to the tool tip than the center point). In 3D cutter radius compensation, the motion program path is written as if the tool center point were in contact with the surface.

Power PMAC does have terms for **Coord[*x*].CC3Data[*i*].ToolOffset** for subsequent arc sections of the tool tip with index values ***i*** greater than 0, but these are calculated automatically by the Power PMAC as the user enters values for **Coord[*x*].CC3Data[*i*].NdotT** and **Coord[*x*]. CC3Data[*i*].CutRadius**, and the user should not attempt to overwrite these values.

For more details, refer to the section *Three-Dimensional Cutter Radius Compensation* in the chapter *Power PMAC Move Mode Trajectories* in the Power PMAC User's Manual.

Coord[*x*].CC3Data[0].ToolRadius

Description: 3D cutter compensation first-segment shaft radius

Range: Positive floating-point

Units: Linear axis user units

Power-on default: 0.0

Coord[*x*].CC3Data[0].ToolRadius specifies, for the tool-tip geometry to be used for three-dimensional cutter radius compensation in Coordinate System ***x***, the perpendicular distance from the tool centerline (axis of rotation) to the starting point of the first defined arc section of the tool tip. It is usually the shaft radius for the tool. This starting point, defined by **Coord[*x*].CC3Data[0].ToolOffset** and **Coord[*x*].CC3Data[0].ToolRadius**, is the reference point for the other sections of the tool-tip table.

Coord[*x*].CC3Data[0].ToolRadius is expressed in the units of the base X, Y, and Z axes for the coordinate system, almost always millimeters or inches. It is not rescaled if the axis units are rescaled through use of a transformation matrix.

Power PMAC does have terms for **Coord[*x*].CC3Data[*i*].ToolRadius** for subsequent arc sections of the tool tip with index values ***i*** greater than 0, but these are calculated automatically by the Power PMAC as the user enters values for **Coord[*x*].CC3Data[*i*].NdotT** and **Coord[*x*]. CC3Data[*i*].CutRadius**, and the user should not attempt to overwrite these values.

For more details, refer to the section *Three-Dimensional Cutter Radius Compensation* in the chapter *Power PMAC Move Mode Trajectories* in the Power PMAC User's Manual.

Coord[x].Ldata. Non-Saved Local Data Elements

The **Coord[*x*].Ldata.** substructure contains elements local to various coordinate system computations. PLC programs and communications threads have identical local-data substructures, so these substructures are documented under the common **Ldata.** section in this chapter.

DPR[*i*]. Non-Saved Setup Data Structure Elements

This section describes read/write elements registers for shared-memory accessories (mostly from third-party suppliers) for the Power PMAC.

DPR[*i*].Data8[*j*]

Description: Shared memory "unsigned 8-bit integer" element, Turbo-compatible addressing

Range: 0 .. 255

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].Data8[*j*] is the "***j***th" unsigned 8-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered for compatibility with Turbo PMAC addressing, and so are "non-linear" with respect to Power PMAC's addressing scheme. This element is typically used for accessories originally designed for the Turbo PMAC.

DPR[*i*].Data8[*j*] and **DPR[*i*].Data8[*j+1*]** are the two halves of **DPR[*i*].Udata16[*j/2*]**.

DPR[*i*].Idata16[*j*]

Description: Shared memory "signed 16-bit integer" element, Turbo-compatible addressing

Range: -2^{15} .. $2^{15}-1$

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].Idata16[*j*] is the "***j***th" signed 16-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered for compatibility with Turbo PMAC addressing, and so are "non-linear" with respect to Power PMAC's addressing scheme. This element is typically used for accessories originally designed for the Turbo PMAC. If access is desired using "linear" addressing for Power PMAC, **DPR[*i*].LinIdata16[*j*]** should be used instead.

The difference in these two addressing schemes is in the low 4 bits of the index ***j***, which can be expressed as the remainder of ***j*** when divided by 16 (***j*** % 16). The relationship between the two schemes can be seen in the following table.

Turbo-compatible ***j*** % 16	Power PMAC Linear ***j*** % 16	Turbo-compatible ***j*** % 16	Power PMAC Linear ***j*** % 16	Turbo-compatible ***j*** % 16	Power PMAC Linear ***j*** % 16	Turbo-compatible ***j*** % 16	Power PMAC Linear ***j*** % 16
0	0	4	2	8	4	12	6
1	8	5	10	9	12	13	14
2	1	6	3	10	5	14	7
3	9	7	11	11	13	15	15

The Power PMAC "word" addressing offset from the base address of the dual-port memory accessory is equivalent to the linear index ***j***. The Power PMAC "byte" offset is 4 times this value. The 16 bits are accessed from the middle 16 bits of the 32-bit bus.

DPR[*i*].Idata32[*j*]

Description: Shared memory "signed 32-bit integer" element, Turbo-compatible addressing

Range: -2^{31} .. $2^{31}-1$

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].Idata32[*j*] is the "***j***th" signed 32-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered for compatibility with Turbo PMAC addressing, and so are "non-linear" with respect to Power PMAC's addressing scheme. This element is typically used for accessories originally designed for the Turbo PMAC. If access is desired using "linear" addressing for Power PMAC, **DPR[*i*].LinIdata32[*j*]** should be used instead.

The difference in these two addressing schemes is in the low 4 bits of the index ***j***, which can be expressed as the remainder of ***j*** when divided by 16 (***j*** % 16). The relationship between the index ***j*** for this Turbo-compatible addressing and the Power PMAC "word" addressing offsets from the base address of the dual-port memory accessory of the two registers used is shown in the following table. The Power PMAC "byte" offsets are 4 times these values.

Turbo-compatible ***j*** % 16	Power PMAC Word Offsets	Turbo-compatible ***j*** % 16	Power PMAC Word Offsets	Turbo-compatible ***j*** % 16	Power PMAC Word Offsets	Turbo-compatible ***j*** % 16	Power PMAC Word Offsets
0	0 & 8	4	4 & 12	8	16 & 24	12	20 & 28
1	1 & 9	5	5 & 13	9	17 & 25	13	21 & 29
2	2 & 10	6	6 & 14	10	18 & 26	14	22 & 30
3	3 & 11	7	7 & 15	11	19 & 27	15	23 & 31

By comparison, the word addressing offsets used for **LinIdata32[*j*]** are 2****j*** and 2****j***+1.

DPR[*i*].LinIdata16[*j*]

Description: Shared memory "signed 16-bit integer" element, linear addressing

Range: -2^{15} .. $2^{15}-1$

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].LinIdata16[*j*] is the "***j***th" signed 16-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered "linearly" in Power PMAC's addressing space (and so are not arranged conveniently for Turbo PMAC addressing – **DPR[*i*].Idata16[*j*]** should be used instead for accessories originally designed for the Turbo PMAC).

The Power PMAC "word" addressing offset from the base address of the dual-port memory accessory is equivalent to the linear index ***j***. The Power PMAC "byte" offset is 4 times this value. The 16 bits are accessed from the middle 16 bits of the 32-bit bus.

DPR[*i*].LinIdata32[*j*]

Description: Shared memory "signed 32-bit integer" element, linear addressing

Range: -2^{31} .. $2^{31}-1$

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].LinIdata32[*j*] is the "***j***th" signed 32-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered "linearly" in Power PMAC's addressing space (and so are not arranged conveniently for Turbo PMAC addressing – **DPR[*i*].Idata32[*j*]** should be used instead for accessories originally designed for the Turbo PMAC).

The 32 bits of data in this element are found at two consecutive word addresses in the Power PMAC I/O space, with 16 bits at each address accessed from the middle 16 bits of the 32-bit bus.

The Power PMAC "word" addressing offsets from the base address of the dual-port memory accessory areequivalent to 2****j*** and 2****j*** +1. The Power PMAC "byte" offsets are 4 times these values.

DPR[*i*].LinUdata16[*j*]

Description: Shared memory "unsigned 16-bit integer" element, linear addressing

Range: 0 .. 2^{16}-1

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].LinUdata16[*j*] is the "***j***th" unsigned 16-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered "linearly" in Power PMAC's addressing space (and so are not arranged conveniently for Turbo PMAC addressing – **DPR[*i*].Udata16[*j*]** should be used instead for accessories originally designed for the Turbo PMAC).

The Power PMAC "word" addressing offset from the base address of the dual-port memory accessory is equivalent to the linear index ***j***. The Power PMAC "byte" offset is 4 times this value. The 16 bits are accessed from the middle 16 bits of the 32-bit bus.

DPR[*i*].LinUdata32[*j*]

Description: Shared memory "unsigned 32-bit integer" element, linear addressing

Range: 0 .. 2^{32}-1

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].LinUdata32[*j*] is the "***j***th" unsigned 32-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered "linearly" in Power PMAC's addressing space (and so are not arranged conveniently for Turbo PMAC addressing – **DPR[*i*].Udata32[*j*]** should be used instead for accessories originally designed for the Turbo PMAC).

The 32 bits of data in this element are found at two consecutive word addresses in the Power PMAC I/O space, with 16 bits at each address accessed from the middle 16 bits of the 32-bit bus.

The Power PMAC "word" addressing offsets from the base address of the dual-port memory accessory areequivalent to 2****j*** and 2****j*** +1. The Power PMAC "byte" offsets are 4 times these values.

DPR[*i*].Udata16[*j*]

Description: Shared memory "unsigned 16-bit integer" element, Turbo-compatible addressing

Range: 0 .. 2^{16}-1

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].Udata16[*j*] is the "***j***th" unsigned 16-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered for compatibility with Turbo PMAC addressing, and so are "non-linear" with respect to Power PMAC's addressing scheme. This element is typically used for accessories originally designed for the Turbo PMAC. If access is desired using "linear" addressing for Power PMAC, **DPR[*i*].LinUdata16[*j*]** should be used instead.

The difference in these two addressing schemes is in the low 4 bits of the index ***j***, which can be expressed as the remainder of ***j*** when divided by 16 (***j*** % 16). The relationship between the two schemes can be seen in the following table.

Turbo-compatible ***j*** % 16	Power PMAC Linear ***j*** % 16	Turbo-compatible ***j*** % 16	Power PMAC Linear ***j*** % 16	Turbo-compatible ***j*** % 16	Power PMAC Linear ***j*** % 16	Turbo-compatible ***j*** % 16	Power PMAC Linear ***j*** % 16
0	0	4	2	8	4	12	6
1	8	5	10	9	12	13	14
2	1	6	3	10	5	14	7
3	9	7	11	11	13	15	15

The Power PMAC "word" addressing offset from the base address of the dual-port memory accessory is equivalent to the linear index ***j***. The Power PMAC "byte" offset is 4 times this value. The 16 bits are accessed from the middle 16 bits of the 32-bit bus.

DPR[*i*].Udata32[*j*]

Description: Shared memory "unsigned 32-bit integer" element, Turbo-compatible addressing

Range: 0 .. 2^{32}-1

Units: Address dependent

Power-on default: Address dependent

DPR[*i*].Udata32[*j*] is the "***j***th" unsigned 32-bit integer data array element in a generic (usually third-party) dual-port memory accessory. The index values ***j*** are ordered for compatibility with Turbo PMAC addressing, and so are "non-linear" with respect to Power PMAC's addressing scheme. This element is typically used for accessories originally designed for the Turbo PMAC. If access is desired using "linear" addressing for Power PMAC, **DPR[*i*].LinIdata32[*j*]** should be used instead.

The difference in these two addressing schemes is in the low 4 bits of the index ***j***, which can be expressed as the remainder of ***j*** when divided by 16 (***j*** % 16). The relationship between the index ***j*** for this Turbo-compatible addressing and the Power PMAC "word" addressing offsets from the base address of the dual-port memory accessory of the two registers used is shown in the following table. The Power PMAC "byte" offsets are 4 times these values.

Turbo-compatible ***j*** % 16	Power PMAC Word Offsets	Turbo-compatible ***j*** % 16	Power PMAC Word Offsets	Turbo-compatible ***j*** % 16	Power PMAC Word Offsets	Turbo-compatible ***j*** % 16	Power PMAC Word Offsets
0	0 & 8	4	4 & 12	8	16 & 24	12	20 & 28
1	1 & 9	5	5 & 13	9	17 & 25	13	21 & 29
2	2 & 10	6	6 & 14	10	18 & 26	14	22 & 30
3	3 & 11	7	7 & 15	11	19 & 27	15	23 & 31

By comparison, the word addressing offsets used for **LinIdata32[*j*]** are 2****j*** and 2****j***+1.

ECAT[*i*]. Non-Saved Setup Data Structure Elements

The following EtherCAT data structures contain the data in Slave I/O registers. This information cannot be saved via the **save** command.

ECAT[*i*].Enable

Description: EtherCAT network enable

Range: 0 .. 3 (**Sys.EcatType** = 0)
0 .. 1 (**Sys.EcatType** = 1)

Units: Bit field

Power-on default: 0

ECAT[*i*].Enable controls whether data updates are enabled for the EtherCAT network with index ***i***. When **Sys.EcatType** = 0 (IgH Etherlab stack), it can take 4 values:

- 0: Cyclic updates and background updates are both disabled
- 1: Cyclic updates are enabled, background updates are disabled
- 2: Cyclic updates are disabled, background updates are enabled
- 3: Cyclic updates and background updates are both enabled

When **Sys.EcatType** = 1 (Acontis stack), it can take 2 values (there are no separate cyclic and background updates):

- 0: Network updates are disabled
- 1: Network updates are enabled

ECAT[*i*].Enable is automatically set to 0 on power-on/reset. The user application should not set it to a non-zero value until it is certain that all connected EtherCAT slave devices have had adequate time to initialize properly and are ready for network communications from the Power PMAC.

With **Sys.EcatType** = 0, cyclic data updates are required to perform servo control over the EtherCAT network. I/O control can be done either with cyclic updates (high-priority, a.k.a. HP) or background (low-priority, a.k.a. LP) updates. Generally, the user should only use a setting of 1 or 2. If cyclic servo control is being performed, it is recommended that all I/O control be performed with cyclic updates (i.e. set this parameter equal to 1 and use cyclic updates for both servo control and I/O). A setting of 3 is generally not recommended, because background updates can interfere with the precise timing of the cyclic updates.

The only situation wherein setting **ECAT[*i*].Enable** = 3 is appropriate is when there is a large amount of non-servo data operating over EtherCAT and it is influencing the duty cycle used for servo. An EtherCAT receive/send cycle usually requires about 10 μsec of processor time.

Therefore, if any data needs to be exchanged at the servo rate, all data should be exchanged at the servo rate, since exchanging at the background rate and the servo rate results in about 20 µsec of processor time being used for EtherCAT. However, if there is a large quantity of non-servo EtherCAT data exchanged in the servo period, and it results in an excessive amount of time used in the servo period, you may wish to place the large quantity of I/O data in the background by using **ECAT[*i*].Enable** = 3.

In firmware versions older than V2.1 (released 1st quarter 2016), **ECAT[*i*].Enable** was a saved setup element, and so was automatically set at power-on/reset to the value it had at the time of the most recent **save** command. In those firmware versions, **ECAT[*i*].Enable** should be set to 0 before using a save command so that all EtherCAT devices can initialize properly before the network is enabled by the user application.

ECAT[*i*].IOBuffer[*m*]

Description: I/O buffer for I/O bit lengths > 32

Range: $0 ... $F

Units: Device dependent

If **ECAT[*i*].IO[*k*].BitLength** > 32, Power PMAC treats the I/O data as a string and places it byte-wise into each element, one byte per element, of **ECAT[*i*].IOBuffer[*m*]**, where ***m* = ECAT[*i*].IO[k].Data + *j***, where ***j*** takes values from 0 to (**ECAT[*i*].IO[*k*].BitLength/8** – 1**)**. For example, if **ECAT[*i*].IO[*k*].BitLength** = 128 and **ECAT[*i*].IO[*k*].Data** = 1000, the I/O data will be placed byte-wise in **ECAT[*i*].IOBuffer[0], ECAT[*i*].IOBuffer[1], … ECAT[*i*].IOBuffer[1015]**. In this case, **ECAT[*i*].IO[*k*].Data** is used as a user-specified offset which determines the starting index of **ECAT[*i*].IOBuffer[*m*]** array elements that will store the I/O data.

EtherCAT Cyclic I/O Non-Saved Elements

The following structures are related to I/O devices for use in the High Priority interrupt. These devices fall into a category different from Slave amplifiers.

ECAT[*i*].IO[*k*].Data

Description: EtherCAT slave device cyclic I/O data

Range: $0 ... $FFFFFFFF

Units: Device dependent

ECAT[*i*].IO[*k*].Data contains the contents of the data transfer for the cyclic inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***. If **ECAT[*i*].IO[*k*].Input** for the bank is 0, the contents that the user has written to this element are copied to the slave device for the bank

each network update cycle. If **ECAT[*i*].IO[*k*].Input** for the bank is 1, the contents of this element are copied from the slave device for the bank each network update cycle.

Starting in V2.0.2 firmware (released 2nd quarter 2015), individual bits of this element can be accessed for read or write purposes in the Script environment through the use of the syntax **ECAT[*i*].IO[*k*].Data.*m***, where ***m*** is an integer constant in the range 0 to 31 representing the bit number.

Consecutive sets of bits of this element can be accessed for read or write purposes in the Script environment through the use of the syntax **ECAT[*i*].IO[*k*].Data.*m*.*n***, where ***m*** is an integer constant in the range 0 to 31 representing the starting (low) bit number, and ***n*** is an integer constant in the range 1 to 32, not to exceed (32 – ***k***), representing the number of consecutive bits to be accessed.

ECAT[*i*].IO[*k*].Offset

Description: EtherCAT I/O data memory offset

Range: $0 ... $FFFFFFFF

Units: Bytes

This parameter indicates the memory offset of **ECAT[*i*].IO[*k*].Data** from the beginning of the process data buffer. It is used for internal diagnostics only.

EtherCAT Low-Priority I/O Module Non-Saved Elements

The following structures are related Slave I/O to be used in background (i.e. in the Low-Priority EtherCAT interrupt).

ECAT[i].LPIO[k].Data

Description: EtherCAT slave device background I/O data

Range: $0 ... $FFFFFFFF

Units: Device dependent

ECAT[*i*].LPIO[*k*].Data contains the contents of the data transfer for the background (low-priority) inputs and/or outputs of I/O bank ***k*** for the EtherCAT network with index ***i***. If **ECAT[*i*].LPIO[*k*].Input** for the bank is 0, the contents that the user has written to this element are copied to the slave device for the bank periodically in background. If **ECAT[*i*].LPIO[*k*].Input** for the bank is 1, the contents of this element are copied from the slave device for the bank periodically in background.

Starting in V2.0.2 firmware (released 2nd quarter 2015), individual bits of this element can be accessed for read or write purposes in the Script environment through the use of the syntax

ECAT[*i*].LPIO[*k*].Data.*m*, where ***m*** is an integer constant in the range 0 to 31 representing the bit number.

Consecutive sets of bits of this element can be accessed for read or write purposes in the Script environment through the use of the syntax **ECAT[*i*].LPIO[*k*].Data.*m.n***, where ***m*** is an integer constant in the range 0 to 31 representing the starting (low) bit number, and ***n*** is an integer constant in the range 1 to 32, not to exceed (32 – ***k***), representing the number of consecutive bits to be accessed.

Gate1[*i*]. (PMAC2-Style Servo IC) Non-Saved Setup Data Structure Elements

This section describes setup elements in the "DSPGATE1" Servo ASIC whose values are not copied to flash memory on a **save** command.

The index value ***i*** for an IC can range from 4 to 19 (0 – 3 are reserved). It is determined by the hardware address DIP switch setting for the board. Note that channel index values ***j*** can range from 0 to 3, representing hardware channel numbers 1 to 4, respectively.

Gate1[*i*].Chan[*j*].AmpEna

Description: IC channel amplifier-enable output flag value

Range: 0 .. 1

Units: Boolean

Power-on default: 0

Gate1[*i*].Chan[*j*].AmpEna is a single-bit value that controls the state of the amplifier-enable output for the channel.

If this output flag is used for the automatic amplifier-enable function for a motor as specified by **Motor[*x*].pAmpEnable** and **Motor[*x*].AmpEnableBit**, user application code should generally not write to this element.

This hardware output bit should not be confused with the software motor status bit **Motor[*x*].AmpEna**.

Gate1[*i*].Chan[*j*].AmpEna is bit 14 of the 24-bit element **Gate1[*i*].Chan[*j*].Ctrl** (bit 22 for C access). C-language software must use the full word element with masking and shifting as needed.

Gate1[*i*].Chan[*j*].CompA

Description: Servo IC channel compare position A

Range: 0 .. 16,777,215

Units: Encoder counts

Power-on default: 0

Gate1[*i*].Chan[*j*].CompA specifies the raw encoder position at which a position-compare event will take place, toggling the EQUn compare output for the channel. It is in units of "counts" of the encoder decode circuit for the channel, with **Gate1[*i*].Chan[*j*].EncCtrl** defining how that decode is done. The position is relative to the power-on/reset position. If this encoder is the

position feedback encoder for a motor, its position is offset from the motor position by the amount in **Motor[*x*].HomePos**, which is the encoder position at the motor home (zero) position.

If **Gate1[*i*].Chan[*j*].OneOverTEna** is set to 1, then "hardware 1/T" sub-count interpolation is enabled for the channel, and bits 0 – 11 of **Gate1[*i*].Chan[*j*].TimeSinceCts** holds the fractional-count value for the **CompA** position.

When the encoder position of **Gate1[*i*].Chan[*j*].CompA** is passed, the value of **Gate1[*i*].Chan[*j*].CompB** is changed by the amount of **Gate1[*i*].Chan[*j*].CompAdd** (in the direction of motion). When the encoder position of **Gate1[*i*].Chan[*j*].CompB** is passed, the value of **Gate1[*i*].Chan[*j*].CompA** is changed by the amount of **Gate1[*i*].Chan[*j*].CompAdd** (in the direction of motion). This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs.

In the Script environment, it is possible to write values from -8,388,608 (-2^{23}) to +16,777,215 ($+2^{24}-1$) to this element. (An attempt to write a value outside this range will cause the element value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 24-bit value with a range of -2^{23} to $2^{23}-1$ or an unsigned 24-bit value with a range of 0 to $2^{24}-1$. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{24} greater than that value.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate1[*i*].Chan[*j*].CompAdd

Description: Servo IC channel compare position auto-increment

Range: 0 .. 16,777,215

Units: Encoder counts

Power-on default: 0

Gate1[*i*].Chan[*j*].CompAdd specifies the magnitude of the value that is automatically added to or subtracted from (depending on the direction of motion) the **CompA** and **CompB** compare position registers for the channel as the position of the other register is passed. This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs. It is in units of "counts" of the encoder decode circuit for the channel, with **Gate1[*i*].Chan[*j*].EncCtrl** defining how that decode is done.

When the encoder position of **Gate1[*i*].Chan[*j*].CompA** is passed, the value of **Gate1[*i*].Chan[*j*].CompB** is changed by the amount of **Gate1[*i*].Chan[*j*].CompAdd** (in the direction of motion). When the encoder position of **Gate1[*i*].Chan[*j*].CompB** is passed, the value of **Gate1[*i*].Chan[*j*].CompA** is changed by the amount of **Gate1[*i*].Chan[*j*].CompAdd** (in the direction of motion).

In the Script environment, it is possible to write values from -8,388,608 (-2^{23}) to +16,777,215 ($+2^{24}-1$) to this element. (An attempt to write a value outside this range will cause the element value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 24-bit value with a range of -2^{23} to $2^{23}-1$ or an unsigned 24-bit value with a range

of 0 to 2^{24}-1. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{24} greater than that value.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate1[*i*].Chan[*j*].CompB

Description: Servo IC channel compare position B

Range: 0 .. 16,777,215

Units: Encoder counts

Power-on default: 0

Gate1[*i*].Chan[*j*].CompB specifies the raw encoder position at which a position-compare event will take place, toggling the EQUn compare output for the channel. It is in units of "counts" of the encoder decode circuit for the channel, with **Gate1[*i*].Chan[*j*].EncCtrl** defining how that decode is done. The position is relative to the power-on/reset position. If this encoder is the position feedback encoder for a motor, its position is offset from the motor position by the amount in **Motor[*x*].HomePos**, which is the encoder position at the motor home (zero) position.

If **Gate1[*i*].Chan[*j*].OneOverTEna** is set to 1, then "hardware 1/T" sub-count interpolation is enabled for the channel, and bits 0 – 11 of **Gate1[*i*].Chan[*j*].TimeBetweenCts** holds the fractional-count value for the **CompB** position.

When the encoder position of **Gate1[*i*].Chan[*j*].CompB** is passed, the value of **Gate1[*i*].Chan[*j*].CompA** is changed by the amount of **Gate1[*i*].Chan[*j*].CompAdd** (in the direction of motion). When the encoder position of **Gate1[*i*].Chan[*j*].CompA** is passed, the value of **Gate1[*i*].Chan[*j*].CompB** is changed by the amount of **Gate1[*i*].Chan[*j*].CompAdd** (in the direction of motion). This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs.

In the Script environment, it is possible to write values from -8,388,608 (-2^{23}) to +16,777,215 ($+2^{24}$-1) to this element. (An attempt to write a value outside this range will cause the element value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 24-bit value with a range of -2^{23} to 2^{23}-1 or an unsigned 24-bit value with a range of 0 to 2^{24}-1. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{24} greater than that value.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate1[*i*].Chan[*j*].Dac[*k*]

Description: Servo IC channel D/A converter register command value

Range: -8,388,608 .. 8,388,607

Units: Signed 24-bit DAC units

Power-on default: 0

Gate1[*i*].Chan[*j*].Dac[*k*] specifies the command value for the digital-to-analog converter register for the specified IC, channel, and phase. The phase index ***k*** can take a value of 0 or 1, corresponding to the A or B phase for the channel, respectively.

If the phase is configured for DAC mode output by setting bit 0 of **Gate1[*i*].Chan[*j*].OutputMode** to 1, then Each phase cycle, this 24-bit value is shifted out, MSB first. Typically, the data for an *n*-bit DAC should be placed in the high *n* bits of this 24-bit element.

Gate1[*i*].Chan[*j*].Dac[*k*] shares a register with the PWM command value **Gate1[*i*].Chan[*j*].Pwm[*k*]**. If the register is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

In the script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate1[*i*].Chan[*j*].EquWrite

Description: Servo IC channel compare initial-state write

Range: 0 .. 3

Units: Bit field

Power-on default: 0

Gate1[*i*].Chan[*j*].EquWrite permits the forcing of the present state of the compare output for the channel (EQU*n*). It is a two-bit value. Bit 1 (value 2) is the output state to be forced. Bit 0 (value 1) is the "forcing bit". Setting the forcing bit to 1 causes the state bit to be placed on the output; when this is done, the forcing bit is automatically cleared to 0.

To force the channel's compare output to 1, **EquWrite** should be set to 3 (and it will then report back as 2). To force the channel's compare output to 0, **EquWrite** should be set to 1 (and it will then report back as 0). The value of the output can be read at any time in the status bit **Gate1[*i*].Chan[*j*].Equ**.

Once the output value is set, when the channel's encoder position passes either the position of **Gate1[*i*].Chan[*j*].CompA** is **Gate1[*i*].Chan[*j*].CompB**, the value of the output will toggle to the opposite state.

Gate1[*i*].Chan[*j*].EquWrite constitutes bits 11 – 12 of the full-word element **Gate1[*i*].Chan[*j*].Ctrl** (bits 19 – 20 of the 32-bit element in C). It must be accessed through the full-word element in C.

Gate1[*i*].Chan[*j*].Pfm

Description: Servo IC channel pulse-frequency modulation register command value

Range: -8,388,608 .. 8,388,607

Units: Signed 24-bit PFM units

Power-on default: 0

Gate1[*i*].Chan[*j*].Pfm specifies the command value for the pulse-frequency modulation register for the specified IC and channel. Each channel has a single PFM register that generates a pulse-and-direction output signal pair.

If the phase is configured for PFM mode output by setting bit 1 of **Gate1[*i*].Chan[*j*].OutputMode** to 1, then each PFMCLK cycle, the value in **Gate1[*i*].Chan[*j*].Pfm** is added into a 24-bit accumulator. Each time the accumulator rolls over, an output pulse is generated. If it rolls over in the positive direction, the direction output signal is set for "plus"; if it rolls over in the negative direction, the direction output is set for "minus". The output pulse frequency is proportional to the value of the element.

The frequency of the PFMCLK signal is set by 3 bits of saved setup element **Gate1[*i*].HardwareClockCtrl**. The default frequency is 9.83 MHz. The output frequency for a given command value can be computed as:

$$f_{out} = \frac{Gate1[i].Chan[j].Pfm}{16{,}777{,}216} * f_{PFMCLK}$$

The resulting output frequency for a given PFMCLK frequency and 16-bit servo output value (in the high 16 bits) can be computed as:

$$f_{out} = \frac{ServoOut}{65{,}536} * f_{PFMCLK}$$

The width of each pulse, in PFMCLK cycles, is determined by the dual-use saved setup element **Gate1[*i*].PwmDeadTime**. The IC will inhibit any further pulses until there has been an "off" time equivalent to the pulse width, so the duty cycle of the pulse train cannot exceed 50%.

Gate1[*i*].Chan[*j*].Pfm shares a register with the pulse-width modulation command value **Gate1[*i*].Chan[*j*].Pwm[2]**. If the register is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

In the script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate1[*i*].Chan[*j*].Pwm[*k*]

Description: Servo IC channel pulse-width modulation register command value

Range: -8,388,608 .. 8,388,607

Units: Signed 24-bit PWM units

Power-on default: 0

Gate1[*i*].Chan[*j*].Pwm[*k*] specifies the command value for the pulse-width modulation register for the specified IC, channel, and phase. The phase index ***k*** can take a value of 0, 1 or 2, corresponding to the A, B, or C phase for the channel, respectively.

If the phase is configured for PWM mode output by setting bit 0 (for A and B phases) or bit 1 (for C phase) of **Gate1[*i*].Chan[*j*].OutputMode** to 0, then each phase cycle, the high 16 bits of this value are automatically loaded into the PWM generation circuit for the phase, where their value is digitally compared to the running PWM up/down counter for the IC. (The low 8 bits of this element are not used by the PWM circuit.)

This PWM counter increments between +**Gate1[*i*].PwmPeriod**+1 and –**Gate1[*i*].PwmPeriod**-2 each PWM cycle, and the digital PWM outputs for the phase toggle as the counter passes this command value in each direction. The net duty cycle of the phase PWM signal is proportional to this value of **Gate1[*i*].Chan[*j*].Pwm[*k*]**. When the value in the high 16 bits is greater than or equal to +**Gate1[*i*].PwmPeriod**+1 (element value 256 times this), the duty cycle will be 100% (top signal always on, bottom signal always off). When the value in the high 16 bits is less than or equal to –**Gate1[*i*].PwmPeriod**-2 (element value 256 times this), the duty cycle will be 0% (top signal always off, bottom signal always on).

Gate1[*i*].Chan[*j*].Pwm[*k*] shares a register with the D/A-converter command value **Gate1[*i*].Chan[*j*].Dac[*k*]** for ***k*** = 0 or 1, or with the PFM command value **Gate1[*i*].Chan[*j*].Pfm** for ***k*** = 2. If the register is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

In the script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate2[*i*]. (PMAC2-Style MACRO IC) Non-Saved Setup Data Structure Elements

This section describes setup elements in the "DSPGATE2" MACRO ASIC whose values are not copied to flash memory on a **save** command. The index value ***i*** for an IC can range from 0 to 15. This IC is found on the UMAC ACC-5E.

Gate2[*i*]. Multi-Channel Non-Saved Setup Elements

The non-saved setup elements in this section configure signals that are not-specific to a servo channel. These include those for the general-purpose I/O and MACRO functionality.

Gate2[*i*].DispData

Description: MACRO IC display port data word

Range: $0 .. $FFF

Units: User-determined

Power-on default: $0

Gate2[*i*].DispData is a 12-bit value that represents the state of the 8 I/O points DISP0 – DISP7 on the "display" port of the IC in the low 8 bits, and the 4 "control" lines CTRL0 – CTRL3 in the high 4 bits. The display port is present on the JDISP connector of the ACC-5E board, but the 4 CTRL*n* lines are not presently used in any Power PMAC device. The port is configured with saved setup elements **Gate2[*i*].DispDir** and **Gate2[*i*].DispPol**, whose default values set up the port as non-inverting outputs.

If an I/O point is set to an output because the matching bit of **Gate2[*i*].DispDir** is set to 1, then the bit value of **Gate2[*i*].DispData** controls the output state. If an I/O point is set to an input because the matching bit of **Gate2[*i*].DispDir** is set to 0, then the bit value of **Gate2[*i*].DispData** simply reports the input state. In this case, writing a value to the bit has no effect on either the output state or the subsequently reported bit value.

There is no automatic display function for this port in the Power PMAC, as there was in older PMAC controllers, so this port is always available for general-purpose I/O.

In the C-language environment, this is a 32-bit element, with the real data in bits 8 – 19.

In the Script environment, an individual I/O point in the I/O bank can be referenced with the syntax **Gate2[*i*].DispData.*k***, where ***k*** is an integer constant in the range 0 to 11 representing the bit number in the register. This is particularly useful to be able to assign an M-variable pointer to the I/O point, e.g.:

```
ptr SolenoidOn->Gate2[0].DispData.3
```

Similarly, a set of I/O points occupying consecutive bits in the I/O bank can be referenced with the syntax **Gate2[*i*].DispData.*k.l***, where ***k*** is an integer constant in the range 0 to 11 representing the starting (low) bit number in the register, and ***l*** is an integer constant in the range 1 to 12, not to exceed (12 – ***k***), representing the number of consecutive bits to be accessed.

Gate2[*i*].HighIoData

Description: MACRO IC I/O port high data word

Range: $0 .. $FF

Units: User-determined

Power-on default: $0

Gate2[*i*].HighIoData is an 8-bit value that represents the state of the high 8 I/O points on the 32-bit I/O port of the IC. This port is present on the JI/O connector of the ACC-5E board. The port is configured with saved setup elements **Gate2[*i*].HighIoMode**, **Gate2[*i*].HighIoDir**, and **Gate2[*i*].HighIoPol**, whose default values set up the port as non-inverting general-purpose inputs.

If an I/O point is set to an output because the matching bit of **Gate2[*i*].HighIoDir** is set to 1, then the bit value of **Gate2[*i*].HighIoData** controls the output state. If an I/O point is set to an input because the matching bit of **Gate2[*i*].HighIoDir** is set to 0, then the bit value of **Gate2[*i*].HighIoData** simply reports the input state. In this case, writing a value to the bit has no effect on either the output state or the subsequently reported bit value.

In the C-language environment, this is a 32-bit element, with the real data in bits 8 – 15.

In the Script environment, an individual I/O point in the I/O bank can be referenced with the syntax **Gate2[*i*].HighIoData.*k***, where ***k*** is an integer constant in the range 0 to 7 representing the bit number in the register. This is particularly useful to be able to assign an M-variable pointer to the I/O point, e.g.:

```
ptr ContactSwitch->Gate2[1].HighIoData.6
```

Similarly, a set of I/O points occupying consecutive bits in the I/O bank can be referenced with the syntax **Gate2[*i*].HighIoData.*k.l***, where ***k*** is an integer constant in the range 0 to 7 representing the starting (low) bit number in the register, and ***l*** is an integer constant in the range 1 to 8, not to exceed (8 – ***k***), representing the number of consecutive bits to be accessed.

Gate2[*i*].LowIoData

Description: MACRO IC I/O port low data word

Range: $0 .. $FFFFFF

Units: User-determined

Power-on default: $0

Gate2[*i*].LowIoData is a 24-bit value that represents the state of the low 24 I/O points on the 32-bit I/O port of the IC. This port is present on the JI/O connector of the ACC-5E board, with I/O00 – I/O23 represented in this element. The port is configured with saved setup elements **Gate2[*i*].LowIoMode**, **Gate2[*i*].LowIoDir**, and **Gate2[*i*].LowIoPol**, whose default values set up the port as non-inverting general-purpose inputs.

If an I/O point is set to an output because the matching bit of **Gate2[*i*].LowIoDir** is set to 1, then the bit value of **Gate2[*i*].LowIoData** controls the output state. If an I/O point is set to an input because the matching bit of **Gate2[*i*].LowIoDir** is set to 0, then the bit value of **Gate2[*i*].LowIoData** simply reports the input state. In this case, writing a value to the bit has no effect on either the output state or the subsequently reported bit value.

In the C-language environment, this is a 32-bit element, with the real data in bits 8 – 31.

In the Script environment, an individual I/O point in the I/O bank can be referenced with the syntax **Gate2[*i*].LowIoData.*k***, where ***k*** is an integer constant in the range 0 to 7 representing the bit number in the register. This is particularly useful to be able to assign an M-variable pointer to the I/O point, e.g.:

```
ptr OverTemp->Gate2[1].LowIoData.20
```

Similarly, a set of I/O points occupying consecutive bits in the I/O bank can be referenced with the syntax **Gate2[*i*].LowIoData.*k*.*l***, where ***k*** is an integer constant in the range 0 to 23 representing the starting (low) bit number in the register, and ***l*** is an integer constant in the range 1 to 24, not to exceed (24 – ***k***), representing the number of consecutive bits to be accessed.

Gate2[*i*].Macro[*j*][*k*]

Description: MACRO Node j Register k input/output data

Range: -2^{23} .. $2^{23}-1$

Units: User-determined

Power-on default: $0

Gate2[*i*].Macro[*j*][*k*] is the input/output data register ***k*** of node ***j*** of the MACRO bank in the IC. The data register index ***k*** has a range of 0 to 3. The node index ***j*** has a range of 0 to 15. This bank has a master number on the MACRO ring that is set by bits 20 – 23 of saved setup element **Gate2[*i*].MacroEnable**. The output data in all four registers for the node ***j*** is automatically sent, and the input data in all four registers is received every phase cycle if bit ***j*** of **Gate2[*i*].MacroEnable** is set to 1.

These are 24-bit elements in the Script environment. **Gate2[*i*].Macro[*j*][0]** has real ring data in all 24 bits, while **Gate2[*i*].Macro[*j*][1]**, **Gate2[*i*].Macro[*j*][2]**, and **Gate2[*i*].Macro[*j*][3]** only have real ring data in the high 16 bits.

Note that the output and input registers share an element – a write operation to the element accesses the output register, and this value will be sent out across the ring; a read operation from the element accesses the input register, getting a value that has been received from across the ring. This means that it is not possible to read back an output value that has been written to one of these elements.

When the node is used for automatic servo control, saved setup element **Motor[*x*].pDac** will probably be set to **Gate2[*i*].Macro[*j*][0].a**, and **Motor[*x*].pEncCtrl** will be probably be set to **Gate2[*i*].Macro[*j*][3].a**. In this case, automatic Power PMAC tasks will write to these registers, and in general, user application code should not write to these registers.

In the Script environment, an individual I/O point in a MACRO data register can be referenced with the syntax **Gate2[*i*].Macro[*j*][*k*].*m***, where ***m*** is an integer constant in the range 0 to 23 representing the bit number in the register.

Similarly, a set of I/O points occupying consecutive bits in a MACRO register can be referenced with the syntax **Gate2[*i*].Macro[*j*][*k*].*m*.*n***, where ***m*** is an integer constant in the range 0 to 23 representing the starting (low) bit number in the register, and ***n*** is an integer constant in the range 1 to 24, not to exceed (24 – ***k***), representing the number of consecutive bits to be accessed.

In the C-language environment, these are 32-bit elements, with the real data in the high 24 or 16 bits.

Gate2[*i*].MuxData

Description: MACRO IC I/O multiplexer port data word

Range: $0 .. $FFFF

Units: User-determined

Power-on default: $0

Gate2[*i*].MuxData is a 16-bit value that represents the state of the 16 I/O points on the multiplexer port of the IC. This port is present on the JTHW connector of the ACC-5E board, with DAT0 – DAT7 represented in the low byte of this element, and SEL0 – SEL7 represented in the high byte. The port is configured with saved setup elements **Gate2[*i*].MuxMode**, **Gate2[*i*].MuxDir**, and **Gate2[*i*].MuxPol**, whose default values set up the port as non-inverting general-purpose inputs.

If an I/O point is set to an output because the matching bit of **Gate2[*i*].MuxDir** is set to 1, then the bit value of **Gate2[*i*].MuxData** controls the output state. If an I/O point is set to an input because the matching bit of **Gate2[*i*].MuxDir** is set to 0, then the bit value of **Gate2[*i*].MuxData** simply reports the input state. In this case, writing a value to the bit has no effect on either the output state or the subsequently reported bit value.

The I/O points on this port can be used automatically for multiplexed I/O on Delta Tau's ACC-34 family of remote I/O boards through the **MuxIO** data structure. If using the automatic features, the **Gate2[*i*].MuxData** element should not be used to write to this register manually, and reading this element is not necessary or generally useful.

In the C-language environment, this is a 32-bit element, with the real data in bits 8 – 23.

In the Script environment, an individual I/O point in the I/O bank can be referenced with the syntax **Gate2[*i*].MuxData.*k***, where ***k*** is an integer constant in the range 0 to 15 representing the bit number in the register. This is particularly useful to be able to assign an M-variable pointer to the I/O point, e.g.:

```
ptr HoldingClamp->Gate2[3].MuxData.14
```

Similarly, a set of I/O points occupying consecutive bits in the I/O bank can be referenced with the syntax **Gate2[*i*].MuxData.*k*.*l***, where ***k*** is an integer constant in the range 0 to 15 representing the starting (low) bit number in the register, and ***l*** is an integer constant in the range 1 to 16, not to exceed (16 – ***k***), representing the number of consecutive bits to be accessed.

Gate2[*i*]. Channel-Specific Non-Saved Setup Elements

The non-saved setup elements in this section are used to configure the use of the inputs and outputs for the selected servo channel on the IC. Note that channel index values ***j*** can range from 0 to 1, representing hardware channel numbers 1 to 2, respectively.

The quadrature encoder inputs have dedicated pins as "handwheel" inputs on the ACC-5E. The Phase C (**Pfm**, **Pwm[2]**) outputs have dedicated pins as "pulse" outputs. Other digital channel signals share pins with general-purpose digital I/O as controlled by saved setup elements **Gate2[*i*].LowIoMode**, **Gate2[*i*].HighIoMode**, and **Gate2[*i*].MuxMode**.

Gate2[*i*].Chan[*j*].AmpEna

Description: IC channel amplifier-enable output flag value

Range: 0 .. 1

Units: Boolean

Power-on default: 0

Gate2[*i*].Chan[*j*].AmpEna is a single-bit value that controls the state of the amplifier-enable output for the channel.

If this output flag is used for the automatic amplifier-enable function for a motor as specified by **Motor[*x*].pAmpEnable** and **Motor[*x*].AmpEnableBit**, user application code should generally not write to this element.

This hardware output bit should not be confused with the software motor status bit **Motor[*x*].AmpEna**.

Gate2[*i*].Chan[*j*].AmpEna is bit 14 of the 24-bit element **Gate2[*i*].Chan[*j*].Ctrl** (bit 22 for C access). C-language software must use the full word element with masking and shifting as needed.

Gate2[*i*].Chan[*j*].CompA

Description: MACRO IC channel compare position A

Range: 0 .. 16,777,215

Units: Encoder counts

Power-on default: 0

Gate2[*i*].Chan[*j*].CompA specifies the raw encoder position at which a position-compare event will take place, toggling the EQUn compare output for the channel. It is in units of "counts" of the encoder decode circuit for the channel, with **Gate2[*i*].Chan[*j*].EncCtrl** defining how that decode is done. The position is relative to the power-on/reset position. If this encoder is the position feedback encoder for a motor, its position is offset from the motor position by the amount in **Motor[*x*].HomePos**, which is the encoder position at the motor home (zero) position.

If **Gate2[*i*].Chan[*j*].OneOverTEna** is set to 1, then "hardware 1/T" sub-count interpolation is enabled for the channel, and bits 0 – 11 of **Gate2[*i*].Chan[*j*].TimeSinceCts** holds the fractional-count value for the **CompA** position.

When the encoder position of **Gate2[*i*].Chan[*j*].CompA** is passed, the value of **Gate2[*i*].Chan[*j*].CompB** is changed by the amount of **Gate2[*i*].Chan[*j*].CompAdd** (in the direction of motion). When the encoder position of **Gate2[*i*].Chan[*j*].CompB** is passed, the value of **Gate2[*i*].Chan[*j*].CompA** is changed by the amount of **Gate2[*i*].Chan[*j*].CompAdd** (in the direction of motion). This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs.

In the Script environment, it is possible to write values from -8,388,608 (-2^{23}) to +16,777,215 ($+2^{24}-1$) to this element. (An attempt to write a value outside this range will cause the element value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 24-bit value with a range of -2^{23} to $2^{23}-1$ or an unsigned 24-bit value with a range of 0 to $2^{24}-1$. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{24} greater than that value.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate2[*i*].Chan[*j*].CompAdd

Description: MACRO IC channel compare position auto-increment

Range: 0 .. 16,777,215

Units: Encoder counts

Power-on default: 0

Gate2[*i*].Chan[*j*].CompAdd specifies the magnitude of the value that is automatically added to or subtracted from (depending on the direction of motion) the **CompA** and **CompB** compare

position registers for the channel as the position of the other register is passed. This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs. It is in units of "counts" of the encoder decode circuit for the channel, with **Gate2[*i*].Chan[*j*].EncCtrl** defining how that decode is done.

When the encoder position of **Gate2[*i*].Chan[*j*].CompA** is passed, the value of **Gate2[*i*].Chan[*j*].CompB** is changed by the amount of **Gate2[*i*].Chan[*j*].CompAdd** (in the direction of motion). When the encoder position of **Gate2[*i*].Chan[*j*].CompB** is passed, the value of **Gate2[*i*].Chan[*j*].CompA** is changed by the amount of **Gate2[*i*].Chan[*j*].CompAdd** (in the direction of motion).

In the Script environment, it is possible to write values from -8,388,608 (-2^{23}) to +16,777,215 ($+2^{24}-1$) to this element. (An attempt to write a value outside this range will cause the element value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 24-bit value with a range of -2^{23} to $2^{23}-1$ or an unsigned 24-bit value with a range of 0 to $2^{24}-1$. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{24} greater than that value.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate2[*i*].Chan[*j*].CompB

Description: MACRO IC channel compare position B

Range: 0 .. 16,777,215

Units: Encoder counts

Power-on default: 0

Gate2[*i*].Chan[*j*].CompB specifies the raw encoder position at which a position-compare event will take place, toggling the EQUn compare output for the channel. It is in units of "counts" of the encoder decode circuit for the channel, with **Gate2[*i*].Chan[*j*].EncCtrl** defining how that decode is done. The position is relative to the power-on/reset position. If this encoder is the position feedback encoder for a motor, its position is offset from the motor position by the amount in **Motor[*x*].HomePos**, which is the encoder position at the motor home (zero) position.

If **Gate2[*i*].Chan[*j*].OneOverTEna** is set to 1, then "hardware 1/T" sub-count interpolation is enabled for the channel, and bits 0 – 11 of **Gate2[*i*].Chan[*j*].TimeBetweenCts** holds the fractional-count value for the **CompB** position.

When the encoder position of **Gate2[*i*].Chan[*j*].CompB** is passed, the value of **Gate2[*i*].Chan[*j*].CompA** is changed by the amount of **Gate2[*i*].Chan[*j*].CompAdd** (in the direction of motion). When the encoder position of **Gate2[*i*].Chan[*j*].CompA** is passed, the value of **Gate2[*i*].Chan[*j*].CompB** is changed by the amount of **Gate2[*i*].Chan[*j*].CompAdd** (in the direction of motion). This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs.

In the Script environment, it is possible to write values from -8,388,608 (-2^{23}) to +16,777,215 ($+2^{24}-1$) to this element. (An attempt to write a value outside this range will cause the element

value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 24-bit value with a range of -2^{23} to $2^{23}-1$ or an unsigned 24-bit value with a range of 0 to $2^{24}-1$. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{24} greater than that value.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate2[*i*].Chan[*j*].Dac[*k*]

Description: MACRO IC channel D/A converter register command value

Range: -8,388,608 .. 8,388,607

Units: Signed 24-bit DAC units

Power-on default: 0

Gate2[*i*].Chan[*j*].Dac[*k*] specifies the command value for the digital-to-analog converter register for the specified IC, channel, and phase. The phase index ***k*** can take a value of 0 or 1, corresponding to the A or B phase for the channel, respectively.

If the phase is configured for DAC mode output by setting bit 0 of **Gate2[*i*].Chan[*j*].OutputMode** to 1, then Each phase cycle, this 24-bit value is shifted out, MSB first. Typically, the data for an *n*-bit DAC should be placed in the high *n* bits of this 24-bit element.

Gate2[*i*].Chan[*j*].Dac[*k*] shares a register with the PWM command value **Gate2[*i*].Chan[*j*].Pwm[*k*]**. If the register is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

In the script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate2[*i*].Chan[*j*].EquWrite

Description: MACRO IC channel compare initial-state write

Range: 0 .. 3

Units: Bit field

Power-on default: 0

Gate2[*i*].Chan[*j*].EquWrite permits the forcing of the present state of the compare output for the channel (EQU*n*). It is a two-bit value. Bit 1 (value 2) is the output state to be forced. Bit 0 (value 1) is the "forcing bit". Setting the forcing bit to 1 causes the state bit to be placed on the output; when this is done, the forcing bit is automatically cleared to 0.

To force the channel's compare output to 1, **EquWrite** should be set to 3 (and it will then report back as 2). To force the channel's compare output to 0, **EquWrite** should be set to 1 (and it will then report back as 0). The value of the output can be read at any time in the status bit **Gate2[*i*].Chan[*j*].Equ**.

Once the output value is set, when the channel's encoder position passes either the position of **Gate2[*i*].Chan[*j*].CompA** is **Gate2[*i*].Chan[*j*].CompB**, the value of the output will toggle to the opposite state.

Gate2[*i*].Chan[*j*].EquWrite constitutes bits 11 – 12 of the full-word element **Gate2[*i*].Chan[*j*].Ctrl** (bits 19 – 20 of the 32-bit element in C). It must be accessed through the full-word element in C.

Gate2[*i*].Chan[*j*].Pfm

Description: MACRO IC channel pulse-frequency modulation register command value

Range: -8,388,608 .. 8,388,607

Units: Signed 24-bit PFM units

Power-on default: 0

Gate2[*i*].Chan[*j*].Pfm specifies the command value for the pulse-frequency modulation register for the specified IC and channel. Each channel has a single PFM register that generates a pulse-and-direction output signal pair.

If the phase is configured for PFM mode output by setting bit 1 of **Gate2[*i*].Chan[*j*].OutputMode** to 1, then each PFMCLK cycle, the value in **Gate2[*i*].Chan[*j*].Pfm** is added into a 24-bit accumulator. Each time the accumulator rolls over, an output pulse is generated. If it rolls over in the positive direction, the direction output signal is set for "plus"; if it rolls over in the negative direction, the direction output is set for "minus". The output pulse frequency is proportional to the value of the element.

The frequency of the PFMCLK signal is set by 3 bits of saved setup element **Gate2[*i*].HardwareClockCtrl**. The default frequency is 9.83 MHz. The output frequency for a given command value can be computed as:

$$f_{out} = \frac{Gate2[i].Chan[j].Pfm}{16{,}777{,}216} * f_{PFMCLK}$$

The resulting output frequency for a given PFMCLK frequency and 16-bit servo output value (in the high 16 bits) can be computed as:

$$f_{out} = \frac{ServoOut}{65{,}536} * f_{PFMCLK}$$

The width of each pulse, in PFMCLK cycles, is determined by the dual-use saved setup element **Gate2[*i*].PwmDeadTime**. The IC will inhibit any further pulses until there has been an "off" time equivalent to the pulse width, so the duty cycle of the pulse train cannot exceed 50%.

Gate2[*i*].Chan[*j*].Pfm shares a register with the pulse-width modulation command value **Gate2[*i*].Chan[*j*].Pwm[2]**. If the register is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

In the script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate2[*i*].Chan[*j*].Pwm[*k*]

Description: MACRO IC channel pulse-width modulation register command value

Range: -8,388,608 .. 8,388,607

Units: Signed 24-bit PWM units

Power-on default: 0

Gate2[*i*].Chan[*j*].Pwm[*k*] specifies the command value for the pulse-width modulation register for the specified IC, channel, and phase. The phase index ***k*** can take a value of 0, 1 or 2, corresponding to the A, B, or C phase for the channel, respectively.

If the phase is configured for PWM mode output by setting bit 0 (for A and B phases) or bit 1 (for C phase) of **Gate2[*i*].Chan[*j*].OutputMode** to 0, then each phase cycle, the high 16 bits of this value are automatically loaded into the PWM generation circuit for the phase, where their value is digitally compared to the running PWM up/down counter for the IC. (The low 8 bits of this element are not used by the PWM circuit.)

This PWM counter increments between +**Gate2[*i*].PwmPeriod**+1 and –**Gate2[*i*].PwmPeriod**-2 each PWM cycle, and the digital PWM outputs for the phase toggle as the counter passes this command value in each direction. The net duty cycle of the phase PWM signal is proportional to this value of **Gate2[*i*].Chan[*j*].Pwm[*k*]**. When the value in the high 16 bits is greater than or equal to +**Gate2[*i*].PwmPeriod**+1 (element value 256 times this), the duty cycle will be 100% (top signal always on, bottom signal always off). When the value in the high 16 bits is less than or equal to –**Gate2[*i*].PwmPeriod**-2 (element value 256 times this), the duty cycle will be 0% (top signal always off, bottom signal always on).

Gate2[*i*].Chan[*j*].Pwm[*k*] shares a register with the D/A-converter command value **Gate2[*i*].Chan[*j*].Dac[*k*]** for ***k*** = 0 or 1, or with the PFM command value **Gate2[*i*].Chan[*j*].Pfm** for ***k*** = 2. If the register is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

In the script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

In the C-language environment, this is a 32-bit element, with the real data in the high 24 bits.

Gate3[*i*]. (PMAC3-Style Interface IC) Non-Saved Setup Data Structure Elements

This section describes setup elements in the "DSPGATE3" machine-interface ASIC whose values are not copied to flash memory on a **save** command.

The index value ***i*** for an IC can range from 0 to 15. It is determined by the hardware address DIP switch setting for the board.

Gate3[*i*]. Multi-Channel Non-Saved Setup Elements

The non-saved setup elements in this section configure signals that are not-specific to a servo channel. These include those for the general-purpose I/O and MACRO functionality.

Gate3[*i*].GpioData[*j*]

Description: IC general-purpose I/O bank data values

Range: $0 .. $FFFFFFFF

Units: Bit field

Power-on default: $0

Gate3[*i*].GpioData[*j*] is a 32-bit value that represents the state of the 32 I/O points of I/O bank "*j*" for the IC. I/O bank 0 has dedicated pins on the IC. I/O banks 1, 2, and 3 share pins with dedicated servo functions – to be used as general-purpose I/O, the matching bit of saved setup element **Gate3[*i*].GpioMode[*j*]** must be set to 1. For all banks, the purpose of any general-purpose I/O point depends on the hardware implementation and the application.

If an I/O point is set to an output because the matching bit of **Gate3[*i*].GpioDir[*j*]** is set to 1, then the value written to that bit of **Gate3[*i*].GpioData[*j*]** controls the output state. Note that reading back the bit value in **Gate3[*i*].GpioData[*j*]** does not necessarily reflect the value written to the output. However, if **Gate3[*i*].GpioData[*j*]** was written to in the Script environment, this same value was also automatically written to the holding memory register in status element **Gate3[*i*].GpioOutData[*j*]**, so the output bit value can be read from that element.

If an I/O point is set to an input because the matching bit of **Gate3[*i*].GpioDir[*j*]** is set to 0, then the bit value of **Gate3[*i*].GpioData[*j*]** simply reports the input state. In this case, writing a value to the bit has no effect on either the output state or the subsequently reported bit value.

An individual I/O point in the I/O bank can be referenced with the syntax **Gate3[*i*].GpioData[*j*].*k***, where ***k*** is an integer constant in the range 0 to 31 representing the bit number in the register. This is particularly useful to be able to assign an M-variable pointer to the I/O point, e.g.:

```
ptr CoolingAirOn->Gate3[1].GpioData[0].13
```

Similarly, a set of I/O points occupying consecutive bits in the I/O bank can be referenced with the syntax **Gate3[*i*].GpioData.*k.l***, where ***k*** is an integer constant in the range 0 to 31 representing the starting (low) bit number in the register, and ***l*** is an integer constant in the range 1 to 32, not to exceed (32 – ***k***), representing the number of consecutive bits to be accessed.

Note that Power PMAC automatically uses the output bit values in the holding register **Gate3[*i*].GpioOutData[*j*]** when setting a bit value of the actual I/O register **Gate3[*i*].GpioData[*j*]** so it does not inadvertently change other output bit values.

Gate3[*i*].IntCtrl

Description: IC interrupt control and status register

Range: $0 .. $00FFFFFF

Units: Bit field

Power-on default: $0

Gate3[*i*].IntCtrl is the control and status register for the "programmable interrupt controller" in the DSPGATE3 IC, which permits the IC to interrupt the processor on a position-capture or position-compare event on any of the four servo channels in the IC. It 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 compare flag)
- Bit 5: **Chan[1].Equ** (2nd channel compare flag)
- Bit 6: **Chan[2].Equ** (3rd channel compare flag)
- Bit 7: **Chan[3].Equ** (4th channel compare flag)

The primary use of this register is in conjunction with the user-written C interrupt-service routine "`CaptCompISR`", whose execution is triggered by the interrupt from the IC. Of course, one or more of these interrupts must be enabled first from a separate routine in order for the ISR to

execute. The ISR can use the "interrupt source" byte to determine which interrupt event occurred if multiple sources were enabled, and it will write to the "interrupt status" byte to prepare for the next interrupt event.

Gate3[*i*].MacroOutA[*j*][*k*]

Description: MACRO Bank A Node j Register k output data

Range: -2^{31} .. $2^{31}-1$

Units: User-determined

Power-on default: $0

Gate3[*i*].MacroOutA[*j*][*k*] is the output data register ***k*** of node ***j*** of MACRO bank A in the IC. The data register index ***k*** has a range of 0 to 3. The node index ***j*** has a range of 0 to 15. This bank has a master number on the MACRO ring that is set by bits 28 – 31 of saved setup element **Gate3[*i*].MacroEnableA**. The output data in all four registers for the node ***j*** is automatically transmitted every phase cycle if bit (***j*** + 8) of **Gate3[*i*].MacroEnableA** is set to 1.

In the original MACRO protocol, only the high 24 bits of **Gate3[*i*].MacroOutA[*j*][0]** are transmitted over the ring, and only the high 16 bits of **Gate3[*i*].MacroOutA[*j*][1]**, **Gate3[*i*].MacroOutA[*j*][2]**, and **Gate3[*i*].MacroOutA[*j*][3]** are transmitted over the ring. In the new MACRO2 protocol, all 32 bits of all 4 node registers are transmitted over the ring.

When the node is used for automatic servo control, saved setup element **Motor[*x*].pDac** will probably be set to **Gate3[*i*].MacroOutA[*j*][0].a**, and **Motor[*x*].pEncCtrl** will be probably be set to **Gate3[*i*].MacroOutA[*j*][3].a**. In this case, automatic Power PMAC tasks will write to these registers, and in general, user application code should not write to these registers.

In the Script environment, an individual I/O point in a MACRO output data register can be referenced with the syntax **Gate3[*i*].MacroOutA[*j*][*k*].*m***, where ***m*** is an integer constant in the range 0 to 31 representing the bit number in the full 32-bit register.

Similarly, a set of I/O points occupying consecutive bits in a MACRO output register can be referenced with the syntax **Gate3[*i*].MacroOutA[*j*][*k*].*m*.*n***, where ***m*** is an integer constant in the range 0 to 31 representing the starting (low) bit number in the full 32-bit register, and ***n*** is an integer constant in the range 1 to 32, not to exceed (32 – ***k***), representing the number of consecutive bits to be accessed.

In the DSPGATE3 IC, it is possible to read back what has been written to these registers, as the corresponding inputs are in separate registers **Gate3[*i*].MacroInA[*j*][*k*]**. In the older DSPGATE2 ICs, the separate output and input values share a register, so it is not possible to read back the output values written to the registers.

Gate3[*i*].MacroOutB[*j*][*k*]

Description: MACRO Bank B Node j Register k output data

Range: -2^{31} .. $2^{31}-1$

Units: User-determined

Power-on default: $0

Gate3[*i*].MacroOutB[*j*][*k*] is the output data register ***k*** of node ***j*** of MACRO bank B in the IC. The data register index ***k*** has a range of 0 to 3. The node index ***j*** has a range of 0 to 15. This bank has a master number on the MACRO ring that is set by bits 28 – 31 of saved setup element **Gate3[*i*].MacroEnableB**. The output data in all four registers for the node ***j*** is automatically transmitted every phase cycle if bit (***j*** + 8) of **Gate3[*i*].MacroEnableB** is set to 1.

In the original MACRO protocol, only the high 24 bits of **Gate3[*i*].MacroOutB[*j*][0]** are transmitted over the ring, and only the high 16 bits of **Gate3[*i*].MacroOutB[*j*][1]**, **Gate3[*i*].MacroOutB[*j*][2]**, and **Gate3[*i*].MacroOutB[*j*][3]** are transmitted over the ring. In the new MACRO2 protocol, all 32 bits of all 4 node registers are transmitted over the ring.

When the node is used for automatic servo control, saved setup element **Motor[*x*].pDac** will probably be set to **Gate3[*i*].MacroOutB[*j*][0].a**, and **Motor[*x*].pEncCtrl** will be probably be set to **Gate3[*i*].MacroOutB[*j*][3].a**. In this case, automatic Power PMAC tasks will write to these registers, and in general, user application code should not write to these registers.

In the Script environment, an individual I/O point in a MACRO output data register can be referenced with the syntax **Gate3[*i*].MacroOutB[*j*][*k*].*m***, where ***m*** is an integer constant in the range 0 to 31 representing the bit number in the full 32-bit register.

Similarly, a set of I/O points occupying consecutive bits in a MACRO output register can be referenced with the syntax **Gate3[*i*].MacroOutB[*j*][*k*].*m*.*n***, where ***m*** is an integer constant in the range 0 to 31 representing the starting (low) bit number in the full 32-bit register, and ***n*** is an integer constant in the range 1 to 32, not to exceed (32 – ***k***), representing the number of consecutive bits to be accessed.

In the DSPGATE3 IC, it is possible to read back what has been written to these registers, as the corresponding inputs are in separate registers **Gate3[*i*].MacroInB[*j*][*k*]**. In the older DSPGATE2 ICs, the separate output and input values share a register, so it is not possible to read back the output values written to the registers.

Gate3[i]. Channel-Specific Non-Saved Setup Elements

The non-saved setup elements in this section are used to configure the use of the inputs and outputs for the selected servo channel on the IC. Note that channel index values ***j*** can range from 0 to 3, representing hardware channel numbers 1 to 4, respectively.

Gate3[*i*].Chan[*j*].AmpEna

Description: IC channel amplifier-enable output flag (A) value

Range: 0 .. 1

Units: Boolean

Power-on default: 0

Gate3[*i*].Chan[*j*].AmpEna is a single-bit value that controls the state of output flag A for the channel. The function of this output flag (if any) can vary with the hardware implementation, but it is almost always used as the amplifier-enable output for the channel.

If this output flag is used for the automatic amplifier-enable function for a motor as specified by **Motor[*x*].pAmpEnable** and **Motor[*x*].AmpEnableBit**, user application code should generally not write to this element.

This hardware output bit should not be confused with the software motor status bit **Motor[*x*].AmpEna**.

Gate3[*i*].Chan[*j*].AmpEna is bit 8 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. C-language software must use the full word element with masking and shifting as needed.

Gate3[*i*].Chan[*j*].CompA

Description: IC channel compare position A

Range: 0 .. $2^{32} - 1$

Units: 1/4096 encoder count

Power-on default: 0

Gate3[*i*].Chan[*j*].CompA specifies the raw encoder position at which a position-compare event will take place, toggling the EQUn compare output for the channel. It is in units of 1/4096 of a "count" (i.e. with 12 bits of fraction) of the encoder decode circuit for the channel, with **Gate3[*i*].Chan[*j*].EncCtrl** defining how that decode is done. The position is relative to the power-on/reset position. If this encoder is the position feedback encoder for a motor, its position is offset from the motor position by the amount in **Motor[*x*].HomePos**, which is the encoder position at the motor home (zero) position.

The present (most recent servo cycle) position of the encoder can be found in status register **Gate3[*i*].Chan[*j*].ServoCapt**. If **Gate3[*i*].Chan[*j*].AtanEna** is set to its default value of 0, this register has 8 bits of fractional count value, and so is not in the same units as **CompA**. If **AtanEna** is set to 1, **ServoCapt** has 12 bits of fraction (computed as the arctangent of the encoder sine and cosine ADC values), and so is in the same units as **CompA**.

If **Gate3[*i*].Chan[*j*].TimerMode** is set to the default value of 0, then "hardware 1/T" sub-count interpolation is enabled for the channel, and the fractional bits (0 – 11) of **Gate3[*i*].Chan[*j*].CompA** are compared to the fractional count value automatically computed by the 1/T circuitry every encoder SCLK cycle. In this case, status register **Gate3[*i*].Chan[*j*].TimerA** holds the present encoder position with 12 bits of timer-based fraction, and so is in the same units as **CompA**.

When the encoder position of **Gate3[*i*].Chan[*j*].CompA** is passed, the value of **Gate3[*i*].Chan[*j*].CompB** is changed by the amount of **Gate3[*i*].Chan[*j*].CompAdd** (in the direction of motion). When the encoder position of **Gate3[*i*].Chan[*j*].CompB** is passed, the value of **Gate2[*i*].Chan[*j*].CompA** is changed by the amount of **Gate3[*i*].Chan[*j*].CompAdd** (in the direction of motion). This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs.

The act of writing to the **CompA** element automatically enables auto-incrementing on the next compare event. The act of writing to the **EquWrite** element automatically disables auto-incrementing on the next compare event (but it is enabled on subsequent events).

In the Script environment, it is possible to write values from -2^{31} to $+2^{31}-1$ to this element. (An attempt to write a value outside this range will cause the element value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 32-bit value with a range of -2^{31} to $2^{31}-1$ or an unsigned 32-bit value with a range of 0 to $2^{32}-1$. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{32} greater than that value.

Gate3[*i*].Chan[*j*].CompAdd

Description: IC channel compare position auto-increment

Range: 0 .. $2^{32} - 1$

Units: 1/4096 encoder count

Power-on default: 0

Gate3[*i*].Chan[*j*].CompAdd specifies the magnitude of the value that is automatically added to or subtracted from (depending on the direction of motion) the **CompA** and **CompB** compare position registers for the channel as the position of the other register is passed. This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs. It is in units of 1/4096 of a "count" (i.e. 12 bits of fraction) of the encoder decode circuit for the channel, with **Gate3[*i*].Chan[*j*].EncCtrl** defining how that decode is done.

When the encoder position of **Gate3[*i*].Chan[*j*].CompA** is passed, the value of **Gate3[*i*].Chan[*j*].CompB** is changed by the amount of **Gate3[*i*].Chan[*j*].CompAdd** (in the

direction of motion). When the encoder position of **Gate3[*i*].Chan[*j*].CompB** is passed, the value of **Gate3[*i*].Chan[*j*].CompA** is changed by the amount of **Gate3[*i*].Chan[*j*].CompAdd** (in the direction of motion).

Forcing a value into the internal compare state (even if the same as the present value) with **Gate3[*i*].Chan[*j*].EquWrite** inhibits the auto-increment operation on the first compare event that follows. This means that the initial **CompA** and **CompB** values do not have to "bracket" the present position, making it much easier to specify a sequence while the encoder is actively counting, but the direction of motion must be known ahead of time.

Writing to either the **CompA** or **CompB** registers (re-) enables the auto-increment operation on the first subsequent compare event. This requires that the values in **CompA** and **CompB** be on opposite sides of the present position, but it permits an auto-incrementing sequence in either direction from the present position.

In the Script environment, it is possible to write values from -2^{31}) to $+2^{31}-1$ to this element. (An attempt to write a value outside this range will cause the element value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 32-bit value with a range of -2^{31} to $2^{31}-1$ or an unsigned 32-bit value with a range of 0 to $2^{32}-1$. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{32} greater than that value.

Gate3[*i*].Chan[*j*].CompB

Description: IC channel compare position B

Range: 0 .. $2^{32} - 1$

Units: 1/4096 encoder count

Power-on default: 0

Gate3[*i*].Chan[*j*].CompB specifies the raw encoder position at which a position-compare event will take place, toggling the EQUn compare output for the channel. It is in units of 1/4096 of a "count" (i.e. with 12 bits of fraction) of the encoder decode circuit for the channel, with **Gate3[*i*].Chan[*j*].EncCtrl** defining how that decode is done. The position is relative to the power-on/reset position. If this encoder is the position feedback encoder for a motor, its position is offset from the motor position by the amount in **Motor[*x*].HomePos**, which is the encoder position at the motor home (zero) position.

The present (most recent servo cycle) position of the encoder can be found in status register **Gate3[*i*].Chan[*j*].ServoCapt**. If **Gate3[*i*].Chan[*j*].AtanEna** is set to its default value of 0, this register has 8 bits of fractional count value, and so is not in the same units as **CompB**. If **AtanEna** is set to 1, **ServoCapt** has 12 bits of fraction (computed as the arctangent of the encoder sine and cosine ADC values), and so is in the same units as **CompB**.

If **Gate3[*i*].Chan[*j*].TimerMode** is set to the default value of 0, then "hardware 1/T" sub-count interpolation is enabled for the channel, and the fractional bits (0 – 11) of **Gate3[*i*].Chan[*j*].CompB** are compared to the fractional count value automatically computed by the 1/T circuitry every encoder SCLK cycle. In this case, status register

Gate3[*i*].Chan[*j*].TimerA holds the present encoder position with 12 bits of timer-based fraction, and so is in the same units as **CompB**.

When the encoder position of **Gate3[*i*].Chan[*j*].CompB** is passed, the value of **Gate3[*i*].Chan[*j*].CompA** is changed by the amount of **Gate3[*i*].Chan[*j*].CompAdd** (in the direction of motion). When the encoder position of **Gate3[*i*].Chan[*j*].CompA** is passed, the value of **Gate3[*i*].Chan[*j*].CompB** is changed by the amount of **Gate3[*i*].Chan[*j*].CompAdd** (in the direction of motion). This permits the automatic incrementing of the compare positions for a continuous pulse train of outputs.

The act of writing to the **CompB** element automatically enables auto-incrementing on the next compare event. The act of writing to the **EquWrite** element automatically disables auto-incrementing on the next compare event (but it is enabled on subsequent events).

In the Script environment, it is possible to write values from -2^{31} to $+2^{31}-1$ to this element. (An attempt to write a value outside this range will cause the element value to saturate at the end of this legal range.) That is, for writing purposes, it can be treated as either a signed 32-bit value with a range of -2^{31} to $2^{31}-1$ or an unsigned 32-bit value with a range of 0 to $2^{32}-1$. However, for reading purposes, it is always reported as an unsigned value, so if it is set to a negative value, the value reported will be 2^{32} greater than that value.

Gate3[*i*].Chan[*j*].Dac[*k*]

Description: IC channel D/A converter register command value

Range: -2,147,483,648 .. 2,147,483,647

Units: Signed 32-bit DAC units

Power-on default: 0

Gate3[*i*].Chan[*j*].Dac[*k*] specifies the command value for the digital-to-analog converter register for the specified IC, channel, and phase. The phase index ***k*** can take a value of 0, 1, or 2, corresponding to the A, B, or C phase for the channel, respectively.

If the phase is configured for DAC mode output by setting bit ***k*** of **Gate3[*i*].Chan[*j*].OutputMode** to 1, then each phase cycle, this 32-bit value is shifted out, MSB first. Typically, the data for an *n*-bit DAC field should be placed in the high *n* bits of this 32-bit element. (Note that the data for the 18-bit DACs optionally available on the ACC-24E3 analog amplifier interface board are part of a 24-bit "field". The bits used by the DAC are in the low 18 bits of the 24-bit field, but the 24-bit field is in the high 24 bits of the 32-bit register.)

Gate3[*i*].Chan[*j*].Dac[*k*] shares a register with the PWM command value **Gate3[*i*].Chan[*j*].Pwm[*k*]**. If the register is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

If saved setup element **Gate3[*i*].Chan[*j*].PackOutData** is set to 1, 16-bit values for the 4 phases of the channel are "packed" into 2 of these 32-bit elements, so the processor can command all 4 phases with just 2 write operations. In this case, the high 16 bits of the command value for the C phase are written into the low 16 bits of the **Dac[0]** element (with the command value for the A phase in the high 16 bits), and the 16-bit command value for the D phase is written into the low 16 bits of **Dac[1]** (with the command value for the B phase in the high 16 bits), so the value in **Dac[2]** is not used. The resulting command value for each phase is limited to 16 bits in this mode (even if the output device has higher resolution), with lower bits automatically set to 0.

In the Script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

Gate3[*i*].Chan[*j*].EquWrite

Description: IC channel compare initial-state write

Range: 0 .. 3

Units: Bit field

Power-on default: 0

Gate3[*i*].Chan[*j*].EquWrite permits the forcing of the present value of the internal compare state for the channel. It is a two-bit value. Bit 1 (value 2) is the internal state value to be forced. Bit 0 (value 1) is the "forcing bit". Setting the forcing bit to 1 causes the state bit to be placed in the channel's active internal value; when this is done, the forcing bit is automatically cleared to 0.

To force the channel's compare state to 1, **EquWrite** should be set to 3 (and it will then report back as 2). To force the channel's compare state to 0, **EquWrite** should be set to 1 (and it will then report back as 0). The value of the internal state can be read at any time in the status bit **Gate3[*i*].Chan[*j*].Equ**.

This internal compare state value for the channel can be used in the creation of the compare output state for any of the four channels on the IC as controlled by the 4-bit saved setup elements **Gate3[*i*].Chan[*j*].EquOutMask** and **Gate3[*i*].Chan[*j*].EquOutPol**. (Note that here the channel index "*j*" refers to the output channel; the bit number in each element refers to the "supplying" channel.)

The act of writing to the **EquWrite** element automatically disables auto-incrementing on the next compare event (but it is enabled on subsequent events). The act of writing to the **CompA** or **CompB** element automatically enables auto-incrementing on the next compare event.

Once the internal state value is set, when the channel's encoder position passes either the position of **Gate3[*i*].Chan[*j*].CompA** is **Gate3[*i*].Chan[*j*].CompB**, the value of the state will toggle to the opposite value.

Gate3[*i*].Chan[*j*].EquWrite constitutes bits 06 – 07 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. It must be accessed through the full-word element in C.

Gate3[*i*].Chan[*j*].OutFlagB

Description: IC channel output flag B value

Range: 0 .. 1

Units: Boolean

Power-on default: 0

Gate3[*i*].Chan[*j*].OutFlagB is a single-bit value that controls the state of output flag B for the channel. The function of this output flag (if any) varies with the hardware implementation, but it is often used as a brake-control output for the channel.

Gate3[*i*].Chan[*j*].OutFlagB is bit 9 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. C-language software must use the full-word element with masking and shifting as needed.

Gate3[*i*].Chan[*j*].OutFlagC

Description: IC channel output flag C value

Range: 0 .. 1

Units: Boolean

Power-on default: 0

Gate3[*i*].Chan[*j*].OutFlagC is a single-bit value that controls the state of output flag C for the channel. The function of this output flag (if any) varies with the hardware implementation.

Gate3[*i*].Chan[*j*].OutFlagC is bit 10 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. C-language software must use the full-word element with masking and shifting as needed.

Gate3[*i*].Chan[*j*].OutFlagD

Description: IC channel output flag D value

Range: 0 .. 1

Units: Boolean

Power-on default: 0

Gate3[*i*].Chan[*j*].OutFlagD is a single-bit value that controls the state of output flag B for the channel. The function of this output flag (if any) varies with the hardware implementation, but it is often used to enable pulse-and-direction output drivers.

Gate3[*i*].Chan[*j*].OutFlagD is bit 11 of the full-word element **Gate3[*i*].Chan[*j*].OutCtrl**. C-language software must use the full-word element with masking and shifting as needed.

Gate3[*i*].Chan[*j*].Pwm[*k*]

Description: IC channel pulse-width modulation register command value

Range: -2,147,483,648 .. 2,147,483,647

Units: Signed 32-bit PWM units

Power-on default: 0

Gate3[*i*].Chan[*j*].Pwm[*k*] specifies the command value for the pulse-width modulation register for the specified IC, channel, and phase. The phase index ***k*** can take a value of 0, 1, 2, or 3, corresponding to the A, B, C, or D phase for the channel, respectively. While these are 32-bit registers, only the high 16 bits are actually used in PWM generation.

If saved setup element **Gate3[*i*].Chan[*j*].PackOutData** is set to 1, 16-bit values for the 4 phases are "packed" into 2 of these 32-bit elements, so the processor can command all 4 phases with just 2 write operations. In this case, the 16-bit command value for the C phase is written into the low 16 bits of **Pwm[0]** (with the command value for the A phase in the high 16 bits), and the 16-bit command value for the D phase is written into the low 16 bits of **Pwm[1]** (with the command value for the B phase in the high 16 bits), so **Pwm[2]** and **Pwm[3]** are not used.

If the phase is configured for PWM mode output by setting bit ***k*** of **Gate3[*i*].Chan[*j*].OutputMode** to 0, then each phase cycle, the values in these elements are automatically loaded into the PWM generation circuits for the phases, "unpacking" if necessary, where the active 16 bits are each individually compared to the running PWM up/down counter for the channel.

This PWM counter increments between +16,383 and -16,384 each PWM cycle, regardless of PWM frequency, and the digital PWM outputs for the phase toggle as the counter passes this command value in each direction. The net duty cycle of the phase PWM signal is proportional to this value of **Gate3[*i*].Chan[*j*].Pwm[*k*]**. When the value in the high 16 bits is greater than or equal to +16,383 (element value of +1,073,676,288), the duty cycle will be 100% (top signal always on, bottom signal always off). When the value in the high 16 bits is less than or equal to -16,384 (element value of -1,073,741,824), the duty cycle will be 0% (top signal always off, bottom signal always on).

Gate3[*i*].Chan[*j*].Pwm[*k*] shares a register with the D/A-converter command value **Gate3[*i*].Chan[*j*].Dac[*k*]** for ***k*** = 0 to 2, or with the PFM command value **Gate3[*i*].Chan[*j*].Pfm** for ***k*** = 3. (Note that the value of the **Pfm** element is saved for applications that need a constant pulse-frequency output, so the value of **Pwm[3]** is effectively saved as well.)

If one of these registers is specified for a motor command output with **Motor[*x*].pDac** set to the address of this register, the register will be written to automatically every servo or phase cycle, and it is not available for general purpose use. However, if this is not the case, it is available for general-purpose use, with the value settable at any time by writing a value to this register.

In the Script environment, if a value outside the legal range is written to this element, the value in the element will saturate at the maximum legal magnitude for that sign.

GateIo[*i*]. Non-Saved Setup Data Structure Elements

This section describes writeable elements in the "IOGATE" digital I/O ASIC whose values are not copied to flash memory on a **save** command.

GateIo[*i*].CtrlReg

Description: IOGATE control register

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Power-on default: (From **GateIo[*i*].Init.CtrlReg)**

GateIo[*i*].CtrlReg is an 8-bit element in the IOGATE digital I/O interface IC that controls the setup and operation of the IC. On power-up/reset it is automatically used to configure the IC based on the saved settings of the **GateIo[*i*].Init** sub-structure. At the end of this automatic configuration process, the saved value of software element **GateIo[*i*].Init.CtrlReg** is automatically copied into this element.

Bit ***j*** (***j*** = 0 to 5) of **GateIo[*i*].CtrlReg** controls the direction of the 8 I/O points associated with **GateIo[*i*].DataReg[*j*]** for the IC. A value of 0 in the control bit permits a write operation to the data register, enabling the output function for each line in the register. Enabling the output function does not prevent the use of any or all of the lines as inputs, as long as the outputs are off (non-conducting).

A value of 1 in the control bit does not permit a write operation to the data register, disabling the output, reserving the register for inputs. This setting is strongly recommended when using these lines for inputs, as a write operation to the data register cannot then disable the inputs.

Bits 6 and 7 of **GateIo[*i*].CtrlReg** determine what information is accessed in **GateIo[*i*].DataReg[*j*]** and **GateIo[*i*].IntrReg**. When both bits 6 and 7 are set to 0, the actual input/output information in these registers is accessed. For actual operation of the IC, it is essential that these two bits are both set to 0.

When either or both of bits 6 and 7 of **GateIo[*i*].CtrlReg** is set to 1, setup registers for **GateIo[*i*].DataReg[*j*]** and **GateIo[*i*].IntrReg** are accessed. This setup is typically done only at power-on/reset, or in the initial configuration of the card.

GateIo[*i*].DataReg[*j*]

Description: IOGATE data register

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Power-on default: (From **GateIo[*i*].Init.DataReg0[*j*]**)

GateIo[*i*].DataReg[*j*] is an 8-bit element in the IOGATE digital I/O interface IC that represents information about the 8-bit data register with index ***j***. What information about this register this element represents depends on the setup and configuration of the IC.

An individual I/O point in the register can be referenced with the syntax **GateIo[*i*].DataReg[*j*].*k***, where ***k*** is an integer constant in the range 0 to 7 representing the bit number in the register. This is particularly useful to be able to assign an M-variable pointer to the I/O point, e.g.:

```
ptr ConveyorOn->GateIo[2].DataReg[3].5
```

Data Value

If the present values of bits 6 and 7 of **GateIo[*i*].CtrlReg** are both 0, this element represents information about the actual data for the 8 I/O lines for the register. This is the operational (as opposed to configuration) state of the register.

For read operations, depending on how the register was configured, this element can provide any of 4 types of information about the data state of each I/O line associated with the register.

1. The present high/low voltage state of the pin, whether the pin is used as an input or an output. The match between voltage state and resulting bit value depends on whether the bit was set up as inverting or non-inverting.

2. The most recent value written into the bit of the register by the processor, regardless of the state of the pin.

3. The high/low voltage state of the pin latched on the most recent phase or servo clock edge (depending on how the card hardware has been configured).

4. The value obtained through a Gray-code-to-binary conversion from the high-low voltage state of the pin latched on the most recent phase or servo clock edge (depending on how the card hardware has been configured).

For write operations, this element can control the output state of each I/O line associated with the register, provided that the register is presently configured to enable outputs by the value of **GateIo[*i*].CtrlReg**.

Inversion Control Value

If the present value of bit 6 of **GateIo[*i*].CtrlReg** is 1 and bit 7 is 0, the value in **GateIo[*i*].DataReg[*j*]** represents the inversion control state of the 8 lines for the register.

Each bit specifies the inversion of the corresponding I/O point, matched in numerical order. A value of 0 specifies an inverting I/O point for the matching bit. That is, for an output, a value of 0

produces a low (conducting) output from the IC itself, and a value of 1 produces a high (non-conducting) output. For an input, a line pulled low into the IC produces a 1 value, and a line pulled high or permitted to float high produces a 0 value.

A value of 1 in a bit of **GateIo[*i*].DataReg[*j*]** specifies a non-inverting I/O point for the matching bit. That is, for an output, a value of 0 produces a high (non-conducting) output from the IC itself, and a value of 1 produces a low (conducting) output. For an input, a line into the IC pulled low produces a 0 value, and a line pulled high or permitted to float high produces a 1 value.

For IOGATE accessory boards that use isolators and driver ICs to interface to the field wiring (e.g. ACC-65E, ACC-66E, ACC-67E, and ACC-68E), the inverting setting means that the "conducting" state, whether sinking or sourcing, corresponds to a "1" in the bit of the data register for both inputs and outputs.

For IOGATE accessory boards in which the IC pins connect directly to the field wiring and have pull-up resistors on the board (e.g. ACC-14E), the inverting setting on an output means that the sinking transistor on the IC pin is turned on when a "1" is written to the bit of the data register, pulling the output line low. The transistor is turned off when a "0" is written to the bit of the data register, permitting the external resistor to pull the line low. The inverting setting on an input means that when the input pin is actively pulled low, a "1" can be read from the bit of the data register. When the input pin is actively pulled high, or allowed to be pulled up by the external resistor, a "0" can be read from the bit of the data register.

On re-initialization, this value for the element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

On a power-up/reset, this value for the element is automatically copied from the saved value of **GateIo[*i*].Init.DataReg64[*j*]**.

Register Read Control Value

If the present value of bit 6 of **GateIo[*i*].CtrlReg** is 0 and bit 7 is 1, the value in **GateIo[*i*].DataReg[*j*]** represents the read control state of the 8 lines for the register.

The register read control value, along with the register latch control value, determines what is reported for the 8 I/O points in a read operation on the data value for the register. Each bit specifies the reporting of the corresponding I/O point, matched in numerical order.

The action of a bit of the register read control value is dependent on the setting of the matching bit of the register latch control value for the same index ***j***. If the matching bit of the register latch control value is 0, selecting unlatched inputs, the bit of the register read control value determines whether the pin value is read, or the value in the writeable register is read. A value of 0 in the bit of the register read control value selects the pin value to be read from the matching bit of the register data value; a value of 1 in the bit selects the writeable register value.

If the matching bit of the register latch control value is 1, selecting latched inputs, the bit of the register read control value determines whether the directly latched data is read, or the value that is the result of a Gray-code-to-binary conversion. A value of 0 in the bit of the register latch control value selects the directly latched value to be read from the matching bit of register data value; a value of 1 in the bit selects the value that is the result of a Gray-code-to-binary conversion from the latched value.

On re-initialization, the value of this element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

On a power-up/reset, this value for the element is automatically copied from the saved value of **GateIo[*i*].Init.DataReg128[*j*]**.

Register Latch Control Value

If the present value of both bits 6 and 7 of **GateIo[*i*].CtrlReg** is 1, the value in **GateIo[*i*].DataReg[*j*]** represents the read control state of the 8 lines for the register.

The register latch control value, along with the register read control value, determines what is reported for the 8 I/O points as in a read operation on the data value for the register. Each bit specifies the reporting of the corresponding I/O point, matched in numerical order. The 4 possibilities for each bit are described in the section above on the register read control value.

On re-initialization, the value of this element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

On a power-up/reset, this value for the element is automatically copied from the saved value of **GateIo[*i*].Init.DataReg192[*j*]**.

GateIo[*i*].IntrReg

Description: IOGATE interrupt register

Range: $00 .. $FF (0 .. 255)

Units: Bit field

Power-on default: n/a (input)

GateIo[*i*].IntrReg is an 8-bit element in the IOGATE digital I/O interface IC that represents information about the 8-bit interrupt register. What information about this register this element represents depends on the setup and configuration of the IC. Note that for the present implementations of products using the IOGATE, this element is not of much practical use.

Data Value

If the present values of bits 6 and 7 of **GateIo[*i*].CtrlReg** are both 0, this element represents information about the actual data for the 8 "E*n*" input lines for the register. This is the operational (as opposed to configuration) state of the register. In this mode, the register is read only. In the present implementations of products using the IOGATE, these lines are all connected to the phase or servo clock signal, depending on the user hardware configuration of the board.

Inversion Control Value

If the present value of bit 6 of **GateIo[*i*].CtrlReg** is 1 and bit 7 is 0, the value in **GateIo[*i*].DataReg[*j*]** represents the inversion control state of the 8 lines for the register.

The inversion control value determines the inversion of the 8 the "E*n*" line inputs to the IOGATE IC. Each bit specifies the inversion of the corresponding "E*n*" line, matched in numerical order. A value of 0 specifies an inverting I/O point for the matching bit. A low level into the IC produces a 1 value, and a high level produces a 0 value. A value of 1 specifies a non-inverting I/O point for the matching bit. A low level produces a 0 value, and a high level produces a 1 value.

On re-initialization, this value for the element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

On a power-up/reset, this value for the element is automatically copied from the saved value of **GateIo[*i*].Init.IntrReg64**.

Mask Control Value

If the present value of bit 6 of **GateIo[*i*].CtrlReg** is 0 and bit 7 is 1, the value in **GateIo[*i*].IntrReg** represents the mask control state of the 8 lines for the register.

The mask control value determines which of the "E*n*" line inputs to the IOGATE IC can create an "interrupt" output from the IC. Each bit specifies the use of the corresponding E*n* line, matched in numerical order. A value of 0 disables the use for the matching E*n* line. A value of 1 enables the use of the matching E*n* line. A falling edge of the E*n* input will create a falling edge of the IC's "INT" output.

On present Power PMAC I/O accessories, this feature does not have significant use. The E*n* inputs to the IC can only come from the system servo or phase clock signals (as selected by jumpers), which can be used to latch the input data. The "INT" output from the IC cannot interrupt the processor, but it can be read in the "ID" chip for the accessory.

On re-initialization, this value for the element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

On a power-up/reset, this value for the element is automatically copied from the saved value of **GateIo[*i*].Init.DataReg128**.

Edge/Level Control

If the present value of both bits 6 and 7 of **GateIo[*i*].CtrlReg** is 1, the value in **GateIo[*i*].IntrReg[*j*]** represents the edge/level control state of the 8 lines for the register.

The edge/level control value determines which of the "E*n*" line inputs to the IOGATE IC can provide a level-triggered interrupt and which can provide an edge-triggered interrupt on the "interrupt" output from the IC. Each bit specifies the use of the corresponding E*n* line, matched in numerical order. A value of 0 specifies a level-triggered interrupt for the matching E*n* line. A value of 1 specifies an edge-triggered interrupt the matching E*n* line. This setting is only relevant if the input is enabled for interrupts.

On present Power PMAC I/O accessories, this feature does not have significant use. The E*n* inputs to the IC can only come from the system servo or phase clock signals (as selected by jumpers), which can be used to latch the input data. The "INT" output from the IC cannot interrupt the processor, but it can be read in the "ID" chip for the accessory.

On re-initialization, this value for the element is automatically set by the Power PMAC for any digital I/O accessories utilizing the IOGATE IC that it finds in the auto-detection process. The value set provides for the most common use of the particular accessory.

On a power-up/reset, this value for the element is automatically copied from the saved value of **GateIo[*i*].Init.DataReg192**.

Gather. Non-Saved Setup Data Structure Elements

This section describes useful setup elements within the data gathering structure. These values are not copied to flash memory on a **save** command. In most uses, the values of these elements will be set automatically by the IDE's plotting, tuning, and "scope" functions. The IDE will store values of these elements in a separate file in flash memory on the Power PMAC for subsequent use.

Data gathering can be performed under the servo interrupt or the phase interrupt. There is an equivalent set of elements for both environments.

Gather.Addr[*i*]

Description: Servo-interrupt data gathering source address

Range: Power PMAC addresses

Units: Valid addresses

Power-on default: **Sys.ServoCount.a**

Gather.Addr[*i*] specifies the source address for the ***i***th register for data gathering under the servo interrupt. The legal values of ***i*** range from 0 to 127, so up to 128 registers can be gathered each sample.

The value of **Gather.Addr[*i*]** is usually set by specifying the name of the data structure element at the register followed by the "**.a**" (address of) suffix. This permits the address to be specified without the user needing to know the numerical address of the register. It is also possible to specify the numerical value of the address directly. In either case, when the value of **Gather.Addr[*i*]** is queried, Power PMAC will report the numerical value of the address.

Power PMAC will interpret the data at the address specified by **Gather.Addr[*i*]** according to the setting of **Gather.Type[*i*]**, which Power PMAC sets automatically when **Gather.Addr[*i*]** is set to the address of an element with a script command. If **Gather.Addr[*i*]** is set directly to a number in the script environment, or set in C, the user must explicitly set **Gather.Type[*i*]** for the proper data format.

The data at the specified address will be gathered each sample period when gathering is enabled if **Gather.Items** is greater than ***i***.

Gather.Enable

Description: Servo-interrupt data gathering activation control

Range: 0 .. 3

Units: none

Power-on default: 0

Gather.Enable controls whether the data gathering function in the servo interrupt is activated or not. If it is set to 0, gathering under the servo interrupt is not enabled, and when it is enabled, data will be stored starting at the beginning of the buffer, overwriting any previous data in the buffer.

If **Gather.Enable** is set to 1, gathering under the servo interrupt is not enabled, and when it is enabled, data will be stored in the buffer starting where gathering last stopped, so new gathered data is appended to the end of the buffer. Setting this element to 1 once the addresses and number of items to gather have been specified can be useful, because it causes Power PMAC to calculate the number of words required to store each sample (**Gather.LineLength**) and the number of samples that can be stored (**Gather.MaxLines**).

If **Gather.Enable** is set to 2, "finite" gathering under the servo interrupt is enabled, and will proceed according to the rules established by the values of other setup elements. When the number of samples gathered equals **Gather.MaxSamples**, or the gathering buffer is filled, gathering is automatically disabled and **Gather.Enable** is set to 0.

If **Gather.Enable** is set to 3, "indefinite" gathering under the servo interrupt is enabled, and will proceed according to the rules established by the values of other setup elements. Gathering will continue until the user stops it by changing the value of **Gather.Enable** from 3. **Gather.MaxSamples** is not used in this mode, and the buffer is used in "rotary" fashion, with new samples overwriting older samples as necessary.

Gather.Enable may not be set to a value greater than 0 if **Gather.PhaseEnable** has a value greater than 0.

Example

```
open prog 73
Gather.Enable = 2;                    // Start gathering
abs; linear;                          // Specify move mode
X25 Y15 F10 ta100 td200 ts50;         // Command move
dwell 100;                            // Stop pre-calc, allow settling
Gather.Enable = 0;                    // Stop gathering
```

Gather.Items

Description: Servo-interrupt data gathering number of items per sample

Range: 0 .. 128

Units: none

Power-on default: 0

Gather.Items specifies the number of items to be gathered each sample when the data gathering function in the servo interrupt is enabled. The items at the addresses specified by **Gather.Addr[0]** through **Gather.Addr[*Items-1*]** will be gathered each sample.

Gather.MaxSamples

Description: Servo-interrupt data gathering maximum number of samples

Range: Non-negative integers

Units: none

Power-on default: 0

Gather.MaxSamples specifies the maximum number of samples that can be gathered when the data gathering function in the servo interrupt is enabled in "finite" mode (**Gather.Enable** = 2). If data gathering is still enabled when this many samples have been stored in the buffer, Power PMAC will automatically disable gathering by setting **Gather.Enable** to 0.

Gather.MaxSamples is not used when gathering in "indefinite" mode (**Gather.Enable** = 3).

Gather.Period

Description: Servo-interrupt data gathering sampling period

Range: 0 .. 65,535

Units: Servo interrupt periods

Power-on default: 0

Gather.Period specifies the interval between consecutive samples when the data gathering function in the servo interrupt is enabled, in servo interrupt periods ("servo cycles"). If **Gather.Period** is set to 1, gathering is performed every servo cycle; if it is set to 2; every second servo cycle, and so on. If it is set to its default value of 0, gathering is done on a "single-shot" basis, not cyclically.

Gather.PhaseAddr[*i*]

Description: Phase-interrupt data gathering source address

Range: Power PMAC addresses

Units: Valid addresses

Power-on default: **Sys.PhaseCount.a**

Gather.PhaseAddr[*i*] specifies the source address for the ***i***th register for data gathering under the phase interrupt. The legal values of ***i*** range from 0 to 7, so up to 8 registers can be gathered each sample.

The value of **Gather.PhaseAddr[*i*]** is usually set by specifying the name of the data structure element at the register followed by the "**.a**" (address of) suffix. This permits the address to be

specified without the user needing to know the numerical address of the register. It is also possible to specify the numerical value of the address directly. In either case, when the value of **Gather.PhaseAddr[*i*]** is queried, Power PMAC will report the numerical value of the address.

Power PMAC will interpret the data at the address specified by **Gather.PhaseAddr[*i*]** according to the setting of **Gather.PhaseType[*i*]**, which Power PMAC sets automatically when **Gather.PhaseAddr[*i*]** is set to the address of an element with a script command. If **Gather.PhaseAddr[*i*]** is set directly to a number in the script environment, or set in C, the user must explicitly set **Gather.PhaseType[*i*]** for the proper data format.

The data at the specified address will be gathered each sample period when gathering is enabled if **Gather.PhaseItems** is greater than ***i***.

Gather.PhaseEnable

Description: Phase-interrupt data gathering activation control

Range: 0 .. 3

Units: none

Power-on default: 0

Gather.PhaseEnable controls whether the data gathering function in the phase interrupt is activated or not. If it is set to 0, gathering under the phase interrupt is not enabled, and when it is enabled, data will be stored starting at the beginning of the buffer, overwriting any previous data in the buffer.

If **Gather.PhaseEnable** is set to 1, gathering under the phase interrupt is not enabled, and when it is enabled, data will be stored in the buffer starting where gathering last stopped, so new gathered data is appended to the end of the buffer. Setting this element to 1 once the addresses and number of items to gather have been specified can be useful, because it causes Power PMAC to calculate the number of words required to store each sample (**Gather.PhaseLineLength**) and the number of samples that can be stored (**Gather.PhaseMaxLines**).

If **Gather.PhaseEnable** is set to 2, "finite" gathering under the phase interrupt is enabled, and will proceed according to the rules established by the values of other setup elements. When the number of samples gathered equals **Gather.PhaseMaxSamples**, or the gathering buffer is filled, gathering is automatically disabled and **Gather.PhaseEnable** is set to 0.

If **Gather.PhaseEnable** is set to 3, "indefinite" gathering under the phase interrupt is enabled, and will proceed according to the rules established by the values of other setup elements. Gathering will continue until the user stops it by changing the value of **Gather.PhaseEnable** from 3. **Gather.PhaseMaxSamples** is not used in this mode, and the buffer is used in "rotary" fashion, with new samples overwriting older samples as necessary.

Gather.PhaseEnable may not be set to a value greater than 0 if **Gather. Enable** has a value greater than 0.

Gather.PhaseItems

Description: Phase-interrupt data gathering number of items per sample

Range: 0 .. 8

Units: none

Power-on default: 0

Gather.PhaseItems specifies the number of items to be gathered each sample when the data gathering function in the phase interrupt is enabled. The items at the addresses specified by **Gather.PhaseAddr[0]** through **Gather.PhaseAddr[*PhaseItems-1*]** will be gathered each sample.

Gather.PhaseMaxSamples

Description: Phase-interrupt data gathering maximum number of samples

Range: Non-negative integers

Units: none

Power-on default: 0

Gather.PhaseMaxSamples specifies the maximum number of samples that can be gathered when the data gathering function in the phase interrupt is enabled in "finite" mode (**Gather.PhaseEnable** = 2). If data gathering is still enabled when this many samples have been stored in the buffer, Power PMAC will automatically disable gathering by setting **Gather.PhaseEnable** to 0.

Gather.PhaseMaxSamples is not used when gathering in "indefinite" mode (**Gather.PhaseEnable** = 3).

Gather.PhasePeriod

Description: Phase-interrupt data gathering sampling period

Range: 0 .. 65,535

Units: Phase interrupt periods

Power-on default: 0

Gather.PhasePeriod specifies the interval between consecutive samples when the data gathering function in the phase interrupt is enabled, in phase interrupt periods ("phase cycles"). If **Gather.PhasePeriod** is set to 1, gathering is performed every phase cycle; if it is set to 2; every second phase cycle, and so on. If it is set to its default value of 0, gathering is done on a "single-shot" basis, not cyclically.

Gather.PhaseType[*i*]

Description: Phase-interrupt data gathering source format

Range: 0 .. 65,535

Units: Enumeration

Power-on default: 0

Gather.PhaseType[*i*] specifies the data format rule that Power PMAC will use to interpret the data at the ***i***th register for data gathering under the phase interrupt. The legal values of ***i*** range from 0 to 7. When gathered data is uploaded to the host computer using the `gather_csv` utility program, that program uses the value of this element to derive the reported value from the gathered data. The IDE's plotting and tuning functions use this utility, as do most user programs that upload gathered data.

In the Power PMAC script environment, when **Gather.PhaseAddr[*i*]** is set to the address of an element, Power PMAC automatically sets **Gather.PhaseType[*i*]** for the format appropriate for that element. However, if the address is specified directly by a number in the script environment, or specified in any way with a C command, the user must specify **Gather.PhaseType[*i*]** explicitly.

The following "full-word" formats are supported:

- 0: 32-bit integer, unsigned
- 1: 32-bit integer, signed
- 2: 24-bit integer, unsigned (in high 24 bits of 32-bit bus)
- 3: 24-bit integer, signed (in high 24 bits of 32-bit bus)
- 4: 32-bit floating-point (single-precision)
- 5: 64-bit floating-point (double-precision)

It is also possible to format the data to create a value from any set of consecutively numbered bits from a 32-bit integer register. In this case, **Gather.PhaseType[*i*]** is split into 3 fields:

- Bits 0 – 2: = 6 for unsigned, = 7 for signed
- Bits 6 – 10: = 32 - (# of bits)
- Bits 11 – 15: = starting bit number

Bits 3 – 5 are reserved for future use. They should be left at 0.

For example, to report bit 20 of a register only as a 0 or 1, bits 0 – 2 would form a value of 6 (110) for unsigned, bits 6 – 10 would form a value of 31 (11111) for one bit, and bits 11 – 15 would form a value of 20 (10100) for starting bit 20. The resulting value of **Gather.PhaseType[*i*]** would be 1010,0111,1100,0110 (base 2), or $A7C6 (base 16), or 49,250 (base 10).

In another example, to report bits 8 to 15 of a register as a signed value, bits 0 – 2 would form a value of 7 (111) for signed, bits 6 – 10 would form a value of 24 (11000) for 8 bits, and bits 11 – 15 would form a value of 8 (01000) for starting bit 8. The resulting value of

Gather.PhaseType[*i*] would be 0100,0110,0000,0111 (base 2), or $4607 (base 16), or 17,927 (base 10).

Gather.PhaseUserBufSize

Description: Phase data gathering user buffer storage length

Range: 0 .. 4,294,967,295

Units: Bytes

Power-on default: 0

Gather.PhaseUserBufSize specifies the length of the phase-interrupt data gathering storage buffer in the user shared memory buffer, in bytes. It is only used if **Gather.PhaseUserBufStart** is set to the address in the user shared memory buffer (not the default).

Gathering storage will start at the address specified by **Gather.PhaseUserBufStart**. As long as it is enabled, it will continue until it reaches the end of the buffer as defined by **Gather.PhaseUserBufSize**. If **Gather.PhaseEnable** is set to 2, gathering will automatically be disabled at this point. If **Gather.PhaseEnable** is set to 3, the storage will wrap around to the beginning of the buffer and continue, overwriting earlier stored data.

It is the user's responsibility to avoid conflicts in the user shared memory buffer.

Gather.PhaseUserBufSize is new in V2.1 firmware, released 1st quarter 2016.

Gather.PhaseUserBufStart

Description: Phase data gathering user buffer storage start address

Range: 0, valid user buffer addresses

Units: Power PMAC memory addresses

Power-on default: 0 (no gathering to user buffer)

Gather.PhaseUserBufStart specifies the address of the beginning of the phase-interrupt data gathering storage buffer in an alternate location in the user shared memory buffer. However, if it is set to the default value of 0, the data gathering storage buffer is located in its standard location.

The standard phase data gathering storage buffer is limited to 1 MB of size, and this cannot be changed. If storage is moved to the user shared memory buffer, much greater storage is possible. The default size of the user shared memory buffer is also 1 MB, but it can be made much larger, limited only by the total available active memory of the Power PMAC.

The size of the user shared memory buffer is defined in the `pp_proj.ini` configuration file for a project, and usually set in the Project Properties window of the IDE.

The value of **Gather.PhaseUserBufStart** is usually set by specifying the name of the buffer data structure element at the register followed by the "**.a**" (address of) suffix. This permits the address to be specified without the user needing to know the numerical address of the register. It is possible to specify the numerical value of the address directly.

A typical assignment command is:

Gather.PhaseUserBufStart = **Sys.Ddata[16384].a**

This will start the phase gather storage 16,384 * 8 = 131,072 bytes from the start of the user shared memory buffer, because each **Ddata** element is 8 bytes long.

Storage from the phase data gathering will start at this address, and can continue for the length of the buffer as defined by **Gather.PhaseUserBufSize**. It is the user's responsibility to avoid conflicts in the user shared memory buffer.

Gather.PhaseUserBufStart is new in V2.1 firmware, released 1st quarter 2016. At its default value of 0, operation is compatible with older firmware versions. IDE tuning and plotting routines require gathering into the standard buffer, not the user shared memory buffer. In versions of the IDE released before this firmware, the user must make sure that **Gather.PhaseUserBufStart** is set to 0 for these routines to work properly.

Gather.Type[*i*]

Description: Servo-interrupt data gathering source format

Range: 0 ..65,535

Units: Enumeration

Power-on default: 0

Gather.Type[*i*] specifies the data format rule that Power PMAC will use to interpret the data at the ***i***th register for data gathering under the servo interrupt. The legal values of ***i*** range from 0 to 127. When gathered data is uploaded to the host computer using the `gather_csv` utility program, that program uses the value of this element to derive the reported value from the gathered data. The IDE's plotting and tuning functions use this utility, as do most user programs that upload gathered data.

In the Power PMAC script environment, when **Gather.Addr[*i*]** is set to the address of an element, Power PMAC automatically sets **Gather.Type[*i*]** for the format appropriate for that element. However, if the address is specified directly by a number in the script environment, or specified in any way with a C command, the user must specify **Gather.Type[*i*]** explicitly.

The following "full-word" formats are supported:

- 0: 32-bit integer, unsigned
- 1: 32-bit integer, signed
- 2: 24-bit integer, unsigned (in high 24 bits of 32-bit bus)
- 3: 24-bit integer, signed (in high 24 bits of 32-bit bus)

- 4: 32-bit floating-point (single-precision)
- 5: 64-bit floating-point (double-precision)

It is also possible to format the data to create a value from any set of consecutively numbered bits from a 32-bit integer register. In this case, **Gather.Type[*i*]** is split into 3 fields:

- Bits 0 – 2: = 6 for unsigned, = 7 for signed
- Bits 6 – 10: = 32 - (# of bits)
- Bits 11 – 15: = starting bit number

Bits 3 – 5 are reserved for future use. They should be left at 0.

For example, to report bit 20 of a register only as a 0 or 1, bits 0 – 2 would form a value of 6 (110) for unsigned, bits 6 – 10 would form a value of 31 (11111) for one bit, and bits 11 – 15 would form a value of 20 (10100) for starting bit 20. The resulting value of **Gather.Type[*i*]** would be 1010,0111,1100,0110 (base 2), or $A7C6 (base 16), or 49,250 (base 10).

In another example, to report bits 8 to 15 of a register as a signed value, bits 0 – 2 would form a value of 7 (111) for signed, bits 6 – 10 would form a value of 24 (11000) for 8 bits, and bits 11 – 15 would form a value of 8 (01000) for starting bit 8. The resulting value of **Gather.Type[*i*]** would be 0100,0110,0000,0111 (base 2), or $4607 (base 16), or 17,927 (base 10).

Gather.UserBufSize

Description: Servo data gathering user buffer storage length

Range: 0 .. 4,294,967,295

Units: Bytes

Power-on default: 0

Gather.UserBufSize specifies the length of the servo-interrupt data gathering storage buffer in the user shared memory buffer, in bytes. It is only used if **Gather.UserBufStart** is set to the address in the user shared memory buffer (not the default).

Gathering storage will start at the address specified by **Gather.UserBufStart**. As long as it is enabled, it will continue until it reaches the end of the buffer as defined by **Gather.UserBufSize**. If **Gather.Enable** is set to 2, gathering will automatically be disabled at this point. If **Gather.Enable** is set to 3, the storage will wrap around to the beginning of the buffer and continue, overwriting earlier stored data.

It is the user's responsibility to avoid conflicts in the user shared memory buffer.

Gather.UserBufSize is new in V2.1 firmware, released 1st quarter 2016.

Gather.UserBufStart

Description: Servo data gathering user buffer storage start address

Range: 0, valid user buffer addresses

Units: Power PMAC memory addresses

Power-on default: 0 (no gathering to user buffer)

Gather.UserBufStart specifies the address of the beginning of the servo-interrupt data gathering storage buffer in an alternate location in the user shared memory buffer. However, if it is set to the default value of 0, the data gathering storage buffer is located in its standard location.

The standard servo data gathering storage buffer is limited to 1 MB of size, and this cannot be changed. If storage is moved to the user shared memory buffer, much greater storage is possible. The default size of the user shared memory buffer is also 1 MB, but it can be made much larger, limited only by the total available active memory of the Power PMAC.

The size of the user shared memory buffer is defined in the `pp_proj.ini` configuration file for a project, and usually set in the Project Properties window of the IDE.

The value of **Gather.UserBufStart** is usually set by specifying the name of the buffer data structure element at the register followed by the "**.a**" (address of) suffix. This permits the address to be specified without the user needing to know the numerical address of the register. It is possible to specify the numerical value of the address directly.

A typical assignment command is:

Gather.UserBufStart = **Sys.Idata[4096].a**

This will start the servo gather storage 4096 * 4 = 16,384 bytes from the start of the user shared memory buffer, because each **Idata** element is 4 bytes long.

Storage from the servo data gathering will start at this address, and can continue for the length of the buffer as defined by **Gather.UserBufSize**. It is the user's responsibility to avoid conflicts in the user shared memory buffer.

Gather.UserBufStart is new in V2.1 firmware, released 1st quarter 2016. At its default value of 0, operation is compatible with older firmware versions. IDE tuning and plotting routines require gathering into the standard buffer, not the user shared memory buffer. In versions of the IDE released before this firmware, the user must make sure that **Gather.UserBufStart** is set to 0 for these routines to work properly.

Ldata. Non-Saved Setup Data Structure Elements

Each coordinate system, PLC program, and communications thread has its own **Ldata** (local data) data structure. From within the executing program or thread, the elements in this structure can be accessed directly as **Ldata.*{element}***. Externally, they must be accessed using the higher-level structure – e.g. **Coord[*x*].Ldata.*{element}*** or **Plc[*i*].Ldata.*{element}***.

Ldata.C[*i*]

Description: Local C-variable value

Range: floating-point

Units: User-specified

Power-on default: 0.0

Ldata.C[*i*] is an alternate way of accessing the local variable **C*i*** for the coordinate system, PLC program, or thread. C-variables are used to pass axis positions (***i*** = 0 to 31) and velocities (***i*** = 32 to 63) to and from user-written kinematics algorithms. These local data elements are generally only used within a **Coord[*x*]** data structure.

For positions, the index ***i*** for each axis is:

Axis	Index	Axis	Index	Axis	Index	Axis	Index
A	0	Z	8	HH	16	SS	24
B	1	AA	9	LL	17	TT	25
C	2	BB	10	MM	18	UU	26
U	3	CC	11	NN	19	VV	27
V	4	DD	12	OO	20	WW	28
W	5	EE	13	PP	21	XX	29
X	6	FF	14	QQ	22	YY	30
Y	7	GG	15	RR	23	ZZ	31

For velocities, the index ***i*** for each axis is:

Axis	Index	Axis	Index	Axis	Index	Axis	Index
A	32	Z	40	HH	48	SS	56
B	33	AA	41	LL	49	TT	57
C	C4	BB	42	MM	50	UU	58
U	C5	CC	43	NN	51	VV	59
V	C6	DD	44	OO	52	WW	60
W	C7	EE	45	PP	53	XX	61
X	C8	FF	46	QQ	54	YY	62
Y	C9	GG	47	RR	55	ZZ	63

In the Script environment, the direct access using **C*i*** is faster and simpler, and so almost always used from within the program or thread. This access through the **Ldata** structure is primarily used for "external" access, usually for debugging purposes.

Ldata.Control

Description: Variable type for vector and matrix functions

Range: 0 .. 2

Units: Enumeration

Power-on default: 0

Ldata.Control specifies which variable type is used for the Script vector and matrix functions in the program or communications thread. The function arguments that specify a variable number specify the number of the type of variable determined by this element.

If **Ldata.Control** is set to the power-on default value of 0, the vector and matrix functions will use "global" P-variables.

If **Ldata.Control** is set to 1, the vector and matrix functions will use "local" L-variables.

If **Ldata.Control** is set to 2, the vector and matrix functions will use "user buffer" Ddata elements (**Sys.Ddata[*i*]**).

In all cases, the variables are double-precision floating-point variables.

For example, the vector function **`sum(6,3,1)`**, which calculates the sum of 3 variables spaced one apart starting with variable 6 will return the sum of global variables P6, P7, and P8 if **Ldata.Control** is 0; it will return the sum of local variables L6, L7, and L8 of the program or thread if **Ldata.Control** is 1; it will return the sum of (global) user buffer elements **Sys.Ddata[6]**, **Sys.Ddata[7]**, and **Sys.Ddata[8]** if **Ldata.Control** is 2.

Ldata.coord

Description: Addressed coordinate system for program or thread

Range: 0 .. 127

Units: Enumeration

Power-on default: 0 (PLCs and communications threads)
Coordinate system number (coordinate systems)

Ldata.coord specifies the modally addressed coordinate system for the program or thread. It determines which coordinate system is affected by program direct commands such as **`run`** and **`abort`**, which coordinate system is queried by commands such as **`pread`** and **`tread`**, and which coordinate system's set of Q-variables (csglobal) is accessed.

Ldata.coord can be set by the user in PLC programs and host communications threads. Note that the power-on default value is 0, specifying C.S. 0, which for most users is not used, so it is

important that this element value be explicitly set before trying to access coordinate-system-specific information.

The value of **Ldata.coord** for a coordinate system is fixed to the number of that coordinate system and cannot be changed by the user. This means that motion programs running in that coordinate system automatically access that coordinate system's information.

In a communications thread, the value of **Ldata.coord** for the thread can be set by the on-line ***&{constant}*** command.

Ldata.D[*i*]

Description: Local D-variable value

Range: floating-point

Units: User-specified

Power-on default: 0.0

Ldata.D[*i*] is an alternate way of accessing the local (non-stack) variable **D*i*** for the coordinate system, PLC program, or thread. Index values ***i*** can range from 0 to 54. In the Script environment, the direct access using **D*i*** is faster and simpler, and so almost always used from within the program or thread. This access through the **Ldata** structure is primarily used for "external" access, usually for debugging purposes.

Ldata.GoBack

Description: Number of jumps back before program suspension/abort

Range: 0 .. 255

Units: Enumeration

Power-on default: 10 (**Coord[*x*].Ldata.GoBack**)
0 (**Plc[*x*].Ldata.GoBack**)

Ldata.GoBack specifies how many times program execution can "jump back" before execution will be suspended or aborted. Execution can jump back (**GoBack** + 1) times without suspension or aborting.

A "jump back" is cause either by the end of a **`while`** or **`do..while`** loop, or by a **`goto`** command to a previous line in the program. Returns from subroutines or subprograms do not count for this purpose.

In a coordinate system, **Ldata.GoBack** is used for forward and inverse kinematic subroutine execution. (This is distinct from saved setup element **Coord[*x*].GoBack**, which is used in the execution of the motion program itself.) Some kinematic algorithms are iterative in nature, and attempt to converge to a solution. This parameter can be used to stop execution in the case where

the algorithm does not converge. In this case, the kinematic subroutine and the motion program that called it are stopped with an error if there are too many jumps back. If it is desired to change this value from the default, the value should be set inside the kinematics routine so it is executed for every scan of the routine.

In a Script PLC program, if there are too many jumps back within the program, the scan of the PLC program is ended, and execution of this PLC will resume at its place in the next cycle (RTI or background). If it is desired to change this value from the default, the value should be set inside the looping construct, as the value is reset to default each time the PLC is entered.

Ldata.L[*i*]

Description: Local L-variable value

Range: floating-point

Units: User-specified

Power-on default: 0.0

Ldata.L[*i*] is an alternate way of accessing the local stack variable **L*i*** for the coordinate system, PLC program, or thread. Index values ***i*** can range from 0 to 16,383, and are referenced to the base of the stack for the top-level program even if a subprogram with a different base is presently executing. Status element **Ldata.Lindex** shows the base index for the presently executing program or subprogram.

In the Script environment, the direct access using **L*i*** is faster and simpler, and so almost always used from within the program or thread. This access through the **Ldata** structure is primarily used for "external" access, usually for debugging purposes.

Ldata.motor

Description: Addressed motor for program or thread

Range: 0 .. 255

Units: Enumeration

Power-on default: 0

Ldata.motor specifies the modally addressed motor for the program or thread. It determines which motor is affected by program direct commands such as **`jog`** and **`home`**.

Ldata.motor can be set by the user in motion programs (for coordinate systems), PLC programs, and host communications threads. Note that the power-on default value is 0, specifying Motor 0, which for most users is not used, so it is important that this element value be explicitly set before trying to command motors.

In a communications thread, the value of **Ldata.motor** for the thread can be set by the on-line ***#{constant}*** command.

Ldata.R[*i*]

Description: Local R-variable value

Range: floating-point

Units: User-specified

Power-on default: 0.0

Ldata.R[*i*] is an alternate way of accessing the local stack pass/return variable **R*i*** for the coordinate system, PLC program, or thread. Index values ***i*** can range from 0 to 255. **R0** for the executing routine is equivalent to **L**(**Ldata.Lsize**) for the routine, which is equivalent to **Ldata.L[Ldata.Lindex + Ldata.Lsize]** for the entire stack

In the Script environment, the direct access using **R*i*** is faster and simpler, and so almost always used from within the program or thread. This access through the **Ldata** structure is primarily used for "external" access, usually for debugging purposes.

Motor[*x*]. Non-Saved Setup Data Structure Elements

This section describes useful setup elements within the motor data structure whose values are not copied to flash memory on a **save** command.

Motor[*x*].CapturePos

Description: Motor position capture monitoring control

Range: 0 .. 1

Units: Boolean

Power-on default: 0

Motor[*x*].CapturePos specifies the state of the motor's position-capture monitoring function. Setting it to 1 causes Power PMAC to monitor the motor's status for a position-capture trigger event every real-time interrupt. If the motor is set up for hardware position capture in one of the Servo ICs, it will do a "dummy" read of the captured position register to re-arm the trigger circuitry. When it detects the trigger event, it reads the sensor position captured by the event, converts it to motor position, and stores the result in status element **Motor[*x*].CapturedPos**. It then automatically sets **Motor[*x*].CapturePos** back to 0, signifying completion of the task.

Note that this position-capture monitoring function does not affect the motor motion in any way. It can even work on an open-loop or killed motor, but the motor must be activated with **Motor[*x*].ServoCtrl** greater than 0. It should *not* be used when the motor is searching for a trigger in a move-until-trigger function such as a homing search move because of the possibility of interference.

The specification of the capture trigger condition and processing of the captured position value is the same as for move-until-trigger functions. This specification is covered in depth in the section on triggered motor moves in the User's Manual chapter *Executing Individual Motor Moves*.

Motor[*x*].CaptureMode specifies the overall functionality of the position-capture function for the motor. When the capture is specified to happen on an input trigger, **Motor[*x*].pCaptFlag** and **Motor[*x*].CaptFlagBit** determine where the motor will look for the trigger bit. If this bit is from one of the Servo ICs, **Gate*n*[*i*].Chan[*j*].CaptCtrl** and **Gate*n*[*i*].Chan[*j*].CaptFlagSel** determine which states of which inputs (flags and index) will cause a trigger.

If a hardware position capture is specified, **Motor[*x*].CaptPosRightShift**, **Motor[*x*].CaptPosLeftShift**, and **Motor[*x*].CaptPosRound** specify how the hardware-captured sensor position is processed to match feedback position.

Motor[x].MacroFlags

Description: Holding register for MACRO control flags

Range: 0 .. $FF

Units: Bit field

Power-on default: 0

Motor[*x*].MacroFlags is a holding register for up to 8 MACRO handshaking control flags for the motor. It comprises the high 8 bits (bits 24 – 31) of the 32-bit element **Motor[*x*].MacroCtrl**.

When the motor is controlled over the MACRO ring, automatic motor routines will access this element in both read and write modes, for purposes such as setting up remote position capture. It is also possible for the user's application to access this register as well, but it is important not to interfere with the automatic routines.

Motor[x].PhaseFindingStep

Description: Motor commutation phase-referencing state

Range: 0 .. 15

Units: Enumeration

Power-on default: 0

Motor[*x*].PhaseFindingStep specifies the state of the phase-referencing algorithm for a Power-PMAC-commutated synchronous motor. Setting it to 1 initiates the phase-referencing algorithm, whether a phasing-search move or an absolute phasing read. This is the equivalent of issuing the on-line **$** command, but can be done directly from within a program.

During operation of a phase-finding search, Power PMAC will set **Motor[*x*].PhaseFindingStep** to values of 1 to 4 during the guesses of a "four-guess" search, or to 5 and 6 during the two steps of a "stepper-motor" search.

Power PMAC will automatically set **Motor[*x*].PhaseFindingStep** back to 0 when it is done with the phase referencing. This can be used to detect when the referencing is complete. Dependent on the referencing method, it may set it to other non-zero values during the referencing process. Note that if Power PMAC considers the phase referencing to be unsuccessful, it will leave the status bit **Motor[*x*].PhaseFound** at 0 at the end of the referencing process, and not permit the enabling of the motor.

Setting **Motor[*x*].PhaseFindingStep** to 8 causes Power PMAC to perform the current-loop auto-nulling procedure if **Motor[*x*].CurrentNullPeriod** > 0) without performing a phase referencing. It will be set back to 0 automatically when the nulling is complete.

Motor[x].PhaseTableBias

Description: Motor commutation phase angle offset

Range: -2048 .. 2047

Units: 1/2048 commutation cycle

Power-on default: 0

Motor[x].PhaseTableBias specifies a user-settable offset term for the rotor phase angle in the commutation algorithm (with or without digital current loop). It is automatically added to the value computed from the rotor angle sensor (and possibly from slip calculations with an induction motor) after it has been scaled into the internal commutation angle units of 1/2048 of a cycle.

This term is intended for special adjustments to the standard algorithm, as for custom "phase advance" algorithms and for interactive testing of the accuracy of phase referencing algorithms.

Motor[x].VaBias

Description: Motor direct PWM first-phase output offset

Range: -32,768 .. 32,767

Units: PWM output units

Power-on default: 0.0

Motor[x].VaBias specifies a user-settable offset term for the first-phase command output of the digital current loop. It is in units of the PWM output registers, as scaled by **Motor[x].PwmSf**. It is automatically added to the value computed for the phase by the "direct PWM" current-loop algorithm. The net first phase command is written to the register whose address is specified by **Motor[x].pDac**. This is normally the Phase A PWM output command.

This term is intended for special adjustments to the standard algorithm, such as compensating for higher harmonics while generating to an AC grid.

There is a comparable offset term for the second phase (**Motor[x].VbBias**), but none for the third phase, because a subsequent built-in algorithm (often called "third-harmonic" injection) automatically "centers" the three phase voltage commands for optimal use of the available bus voltage, while maintaining the relative voltages between the phases.

Motor[x].VbBias

Description: Motor direct PWM second-phase output offset

Range: -32,768 .. 32,767

Units: PWM output units

Power-on default: 0.0

Motor[*x*].VbBias specifies a user-settable offset term for the second-phase command output of the digital current loop. It is in units of the PWM output registers, as scaled by **Motor[*x*].PwmSf**. It is automatically added to the value computed for the phase by the "direct PWM" current-loop algorithm. The net second phase command is written to the next higher-addressed register after the one specified by **Motor[*x*].pDac**. This is normally the Phase B PWM output command.

This term is intended for special adjustments to the standard algorithm, such as compensating for higher harmonics while generating to an AC grid.

There is a comparable offset term for the first phase (**Motor[*x*].VaBias**), but none for the third phase, because a subsequent built-in algorithm (often called "third-harmonic" injection) automatically "centers" the three phase voltage commands for optimal use of the available bus voltage, while maintaining the relative voltages between the phases.

MuxIo. Non-Saved Data Structure Elements

The **MuxIo.** data structure comprises the multiplexed I/O ("thumbwheel port") settings for the Power PMAC. The elements in this structure provide access to multiplexed IO devices which use the serial thumbwheel-multiplexer protocol, such as ACC-34A and ACC-34AA.

MuxIo.PortA[*n*].Data

Description: Multiplexed I/O image word for Port A

Range: $00000000 .. $FFFFFFFF (0 .. 4,294,967,295)

Units: Bit field

Power-on default: 0

Legacy I-variable alias: none

MuxIo.PortA[*n*].Data is the image word in Power PMAC memory for the multiplexed I/O port A of the device with address ***n*** as selected by the DIP switches on the device. The value is read from or written to, depending on the **MuxIo.PortA[*n*].Dir** setting. The period at which these image words are accessed depends on **MuxIo.UpdatePeriod** setting.

An individual I/O point in the I/O bank can be referenced with the syntax **MuxIo.PortA[*n*]. Data.*k*** where ***k*** is an integer constant in the range 0 to 31 representing the bit number in the register. This is particularly useful to be able to assign an M-variable pointer to the I/O point, e.g.:

```
ptr ErrorLight->MuxIo.PortB[1].Data.18
```

Similarly, a set of I/O points occupying consecutive bits in the I/O bank can be referenced with the syntax **MuxIo.PortA[*n*].Data.*k.l*** where ***k*** is an integer constant in the range 0 to 31 representing the starting (low) bit number in the register, and ***l*** is an integer constant in the range 1 to 32, not to exceed (32 – ***k***), representing the number of consecutive bits to be accessed.

MuxIo.PortB[*n*].Data

Description: Multiplexed I/O image word for Port B

Range: $00000000 .. $FFFFFFFF (0 .. 4,294,967,295)

Units: Bit field

Power-on default: 0

Legacy I-variable alias: none

MuxIo.PortB[*n*].Data is the image word in Power PMAC memory for the multiplexed I/O port B of the device with address ***n*** as selected by the DIP switches on the device. The value is read

from or written to, depending on the **MuxIo.PortB[*n*].Dir** setting. The period at which these image words are accessed depends on **MuxIo.UpdatePeriod** setting.

An individual I/O point in the I/O bank can be referenced with the syntax **MuxIo.PortB[*n*]. Data.*k*** where ***k*** is an integer constant in the range 0 to 31 representing the bit number in the register. This is particularly useful to be able to assign an M-variable pointer to the I/O point, e.g.:

```
ptr ErrorLight->MuxIo.PortB[1].Data.18
```

Similarly, a set of I/O points occupying consecutive bits in the I/O bank can be referenced with the syntax **MuxIo.PortB[*n*].Data.*k.l***, where ***k*** is an integer constant in the range 0 to 31 representing the starting (low) bit number in the register, and ***l*** is an integer constant in the range 1 to 32, not to exceed (32 – ***k***), representing the number of consecutive bits to be accessed.

Plc[*i*]. Non-Saved Data Structure Elements

The **Plc[*i*]** data contains elements pertaining to Script program PLC*i*.

Plc[*i*].Ldata. Non-Saved Local Data Elements

The **Plc[*i*].Ldata.** substructure contains elements local to various PLC program computations. Coordinate systems and communications threads have identical local-data substructures, so these substructures are documented under the common **Ldata.** section in this chapter.

PowerBrick[*i*]. Non-Saved Data Structure Elements

The **PowerBrick[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the Power Brick servo interface board are listed under the **Gate3[*i*]** data structure, above.

Sys. Global Non-Saved Setup Data Structure Elements

This section describes global setup elements whose values are not copied to flash memory on a **save** command.

Sys.Cdata[*i*]

Description: User shared memory "character" (byte) data array element

Range: 0 .. 255

Units: User set

Power-on default: 0

Sys.Cdata[*i*] is the "***i***th" unsigned 8-bit integer data array element in the user shared memory buffer. Each of these elements occupies 1 byte in the user shared memory buffer, and is located starting at ***i*** addresses past the beginning of the buffer (which is located at the address in **Sys.pushm**). This array is for general-purpose use in applications; it has no automatic use.

Index values ***i*** in the square brackets can be integer constants in the range 0 to 16,777,215, or local L-variables. No expressions or non-integer constants are permitted. The size of the user shared memory buffer is set as a directive in the `pp_proj.ini` file, 1 megabyte (1,048,576 bytes) by default, which makes the maximum valid index value 262,143.

Values can be assigned to a **Sys.Cdata[*i*]** element as a literal character inside single quotes (e.g. **`Sys.Cdata[170] = 'A'`**) or as a numeric integer value (e.g. **`Sys.Cdata[170] = 65`** or **`Sys.Cdata[170] = 65 + LowerCase*32`**). When queried, Power PMAC will report the value as an integer constant.

Consecutive sets of **Sys.Cdata[*i*]** elements can be used in Power PMAC Script string manipulation functions.

Sys.Cdata[*i*] is located in the same registers as **Sys.Ddata[*i/8*]**, **Sys.Fdata[*i/4*]**, **Sys.Idata[*i/4*]**, **Sys.Udata[*i/4*]**, and **Sys.Uhex[*i/4*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

In C, this element should be accessed as a pointer to a `char` variable, offset from the `pushm` base address by ***i*** addresses (e.g. `MyCharPtr = (char *) pushm + i`).

Sys.Ddata[*i*]

Description: User shared memory "double" data array element

Range: Double-precision floating-point

Units: User set

Power-on default: 0.0

Sys.Ddata[*i*] is the "***i***th" double-precision floating-point data array element in the user shared memory buffer. Each of these elements occupies 8 bytes in the user shared memory buffer, and is located starting at 8***i*** addresses past the beginning of the buffer (which is located at the address in **Sys.pushm**). This array is for general-purpose use in applications; it has no automatic use.

Index values ***i*** in the square brackets can be integer constants in the range 0 to 16,777,215, or local L-variables. No expressions or non-integer constants are permitted. The size of the user shared memory buffer is set as a directive in the `pp_proj.ini` file, 1 megabyte (1,048,576 bytes) by default, which makes the maximum valid index value 131,071.

Sys.Ddata[*i*] is located in the same registers as **Sys.Cdata[*i/8*]** to **Sys.Cdata[*i/8+7*]**, **Sys.Fdata[*2*i*]**, **Sys.Fdata[*2*i+1*]**, **Sys.Idata[*2*i*]**, **Sys.Idata[*2*i+1*]**, **Sys.Udata[*2*i*]**, and **Sys.Udata[*2*i+1*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

In C, this element should be accessed as a pointer to a `double` variable, offset from the `pushm` base address by ***i*** addresses (e.g. `MyDoublePtr = (double *) pushm + i`).

Sys.Fdata[*i*]

Description: User shared memory "float" data array element

Range: Single-precision floating-point

Units: User set

Power-on default: 0.0

Sys.Fdata[*i*] is the "***i***th" single-precision floating-point data array element in the user shared memory buffer. Each of these elements occupies 4 bytes in the user shared memory buffer, and is located starting at 4***i*** addresses past the beginning of the buffer (which is located at the address in **Sys.pushm**). This array is for general-purpose use in applications; it has no automatic use.

Index values ***i*** in the square brackets can be integer constants in the range 0 to 16,777,215, or local L-variables. No expressions or non-integer constants are permitted. The size of the user shared memory buffer is set as a directive in the `pp_proj.ini` file, 1 megabyte (1,048,576 bytes) by default, which makes the maximum valid index value 262,143.

Sys.Fdata[*i*] is located in the same registers as **Sys.Ddata[*i/2*]**, **Sys.Idata[*i*]**, and **Sys.Udata[*i*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

Sys.Fdata[*i*] is located in the same registers as **Sys.Cdata[*i/4*]** to **Sys.Cdata[*i/4+3*]**, **Sys.Ddata[*i/2*]**, **Sys.Idata[*i*]**, **Sys.Udata[*i*]**, and **Sys.Uhex[*i*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

In C, this element should be accessed as a pointer to a `float` variable, offset from the `pushm` base address by ***i*** addresses (e.g. `MyFloatPtr = (float *) pushm + i`).

Sys.Idata[*i*]

Description: User shared memory "signed integer" data array element

Range: -2^{31} .. $2^{31}-1$

Units: User set

Power-on default: 0

Sys.Idata[*i*] is the "***i***th" signed 32-bit integer data array element in the user shared memory buffer. Each of these elements occupies 4 bytes in the user shared memory buffer, and is located starting at 4***i*** addresses past the beginning of the buffer (which is located at the address in **Sys.pushm**). This array is for general-purpose use in applications; it has no automatic use.

Index values ***i*** in the square brackets can be integer constants in the range 0 to 16,777,215, or local L-variables. No expressions or non-integer constants are permitted. The size of the user shared memory buffer is set as a directive in the `pp_proj.ini` file, 1 megabyte (1,048,576 bytes) by default, which makes the maximum valid index value 262,143.

Sys.Idata[*i*] is located in the same registers as **Sys.Ddata[*i/2*]**, **Sys.Fdata[*i*]**, and **Sys.Udata[*i*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

Sys.Idata[*i*] is located in the same registers as **Sys.Cdata[*i/4*]** to **Sys.Cdata[*i/4+3*]**, **Sys.Ddata[*i/2*]**, **Sys.Fdata[*i*]**, **Sys.Udata[*i*]**, and **Sys.Uhex[*i*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

In C, this element should be accessed as a pointer to a `int` variable, offset from the `pushm` base address by ***i*** addresses (e.g. `MyIntPtr = (int *) pushm + i`).

Sys.ioIdata[*j*]

Description: I/O address space "signed integer" data array element

Range: -2^{31} .. $2^{31}-1$

Units: address dependent

Power-on default: address dependent

Sys.ioIdata[*j*] provides an alternate method to access registers in Power PMAC's I/O space, where various interface ASICs and shared-memory ICs can reside. Most commonly, these registers are accessed through pre-defined structures and elements for the individual ICs (e.g. **Gate3[*i*]**, **Acc84E[*i*]**), but it can be useful to access all of these registers in a uniform manner.

The index ***j*** specifies the word offset from the start of Power PMAC's I/O space. Since Power PMAC uses 32-bit (4-byte) words, this offset value is equal to ¼ of the byte address offset values

documented in the chapter *Power PMAC I/O Address Offsets*. These index values can be decimal integer constants in the range 0 to 4,194,303, or local L-variables. No expressions are permitted.

The full 32-bit data bus is accessed when this element is used, even if there is real hardware in only part of the register. The 32-bit value is treated as a signed integer when using this element.

Note

When reading a **Sys.ioIdata[*j*]** element, a numerical value will be returned even if there if there is no actual hardware at that address. This is unlike using an IC element for the same address; that will return "not-a-number" (*nan*).

Example

The first ACC-24E3 in a system is usually assigned to **Gate3[0]**, which has a byte address offset of $900000. The third channel (**Chan[2]**) on this IC has a byte offset from the base address of the IC of $100. The encoder phase capture register for this channel has a byte offset of $004 from the start of the channel registers.

The total byte offset of this register from the start of the I/O space is therefore $900104. The word offset of this register is ¼ of this: $280041, or 2,621,505 decimal. So this register could be accessed as a 32-bit signed integer with **Sys.ioIdata[2621505]**.

Sys.ioUdata[*j*]

Description: I/O address space "unsigned integer" data array element

Range: 0 .. 2^{32}-1

Units: address dependent

Power-on default: address dependent

Sys.ioUdata[*j*] provides an alternate method to access registers in Power PMAC's I/O space, where various interface ASICs and shared-memory ICs can reside. Most commonly, these registers are accessed through pre-defined structures and elements for the individual ICs (e.g. **Gate3[*i*]**, **Acc84E[*i*]**), but it can be useful to access all of these registers in a uniform manner.

The index ***j*** specifies the word offset from the start of Power PMAC's I/O space. Since Power PMAC uses 32-bit (4-byte) words, this offset value is equal to ¼ of the byte address offset values documented in the chapter *Power PMAC I/O Address Offsets*. These index values can be decimal integer constants in the range 0 to 4,194,303, or local L-variables. No expressions are permitted.

The full 32-bit data bus is accessed when this element is used, even if there is real hardware in only part of the register. The 32-bit value is treated as an unsigned integer when using this element.

Note

When reading a **Sys.ioUdata[*j*]** element, a numerical value will be returned even if there if there is no actual hardware at that address. This is unlike using an IC element for the same address; that will return "not-a-number" (*nan*).

Example

The first ACC-24E2*x* in a system is usually assigned to **Gate1[4]**, which has a byte address offset of $600000. The second channel (**Chan[1]**) on this IC has a byte offset from the base address of the IC of $040. The status register for this channel has a byte offset of $020 from the start of the channel registers.

The total byte offset of this register from the start of the I/O space is therefore $600060. The word offset of this register is ¼ of this: $180018, or 1,572,888 decimal. So this register could be accessed as a 32-bit unsigned integer with **Sys.ioUdata[1572888]**. Note that this is a 24-bit register in the ASIC, mapped into the high 24 bits of the 32-bit data bus, so the low 8 bits of the value should not be used.

Sys.Lock[*i*]

Description: Process *i* locking control bit

Range: 0 .. 1

Units: Boolean

Power-on default: 0

Sys.Lock[*i*] is a single-bit element that permits the control and detection of the "locking" status for user execution process ***i*** in Script program operation. There are 32 of these process locks, with indices ***i*** ranging from 0 to 31.

A read access to **Sys.Lock[*i*]** returns the present value of the status bit. A value of 1 indicates that a task (in normal use, a different task) has control of process ***i***.

A returned value of 0 indicates that no task has control of this thread. If the returned value is 0, the act of reading the bit automatically and immediately sets the bit to 1 in an "atomic" operation, permitting this task to robustly take control of the thread.

A write access to set **Sys.Lock[*i*]** to 0 is permitted. In normal use, this should only be done by the task that presently has control of the thread. However, it is possible for any task to do this, allowing a supervisory task to release the thread in the case of an error.

A write access to set **Sys.Lock[*i*]** to 1 is not permitted. The only way to set the bit to 1 is through a read access.

A Script program can wait for control of a process, take control, and then release control, with code such as:

```
while (Sys.Lock[4] == 1) {}          // Wait until process 4 free
{Code that uses control of process 4}
Sys.Lock[4] = 0;                     // Release process 4 control
```

The program will be stuck in the while loop as long as the bit is set to 1, under the assumption that another task has locked the process. This example has an "empty" loop that does nothing while it is waiting for thread control, but of course it is possible to do other tasks inside the waiting loop. When the loop finds that **Sys.Lock[4]** is 0, the act of reading it for the conditional comparison automatically sets it to 1 so any other task looking at the bit will see that the process is locked while this program performs tasks that use control of the process. When it is done with these tasks, it releases control of the process.

In other cases, the Script program may use code such as:

```
if (Sys.Lock[15] == 0) {             // Process 15 free?
  {Code that uses control of process 15}
  Sys.Lock[15] = 0;                  // Release process 15 control
}
```

If the program finds that **Sys.Lock[15]** is 0, the act of reading it for the conditional comparison automatically sets it to 1 so any other task looking at the bit will see that the process is locked while this program performs tasks that use control of the process. When it is done with these tasks, it releases control of the process. If the program finds that **Sys.Lock[15]** is 1, it will skip over these tasks, as it realizes that another program has locked this execution process.

The present state of **Sys.Lock[*i*]** can be read without the possibility of changing its state by reading the 32-bit status element **Sys.Lock** and assessing the value of bit ***i*** of that element.

Sys.M[*i*]

Description: M-variable array element

Range: Definition-dependent

Units: User-determined

Power-on default: Definition-dependent

Sys.M[*i*] is the "***i***th" M-variable array element. M-variables are global pointer user variables that can address memory and I/O registers. Index values ***i*** can range from 0 to 65,535. This array provides an alternate method for accessing M-variables.

This element cannot be directly accessed in C. In C, the targeted register should be addressed directly with a pointer variable.

Sys.P[*i*]

Description: P-variable array element

Range: Floating-point

Units: User-determined

Power-on default: 0

Sys.P[*i*] is the "***i***th" P-variable array element. P-variables are global double-precision floating-point user variables. Index values ***i*** can range from 0 to 65,535. This array provides an alternate method for accessing P-variables.

In C, this element should be accessed as `pshm->P[i]`.

Sys.Udata[*i*]

Description: User shared memory "unsigned integer" data array element

Range: 0 .. 2^{32}-1

Units: User set

Power-on default: 0

Sys.Udata[*i*] is the "***i***th" unsigned 32-bit integer data array element in the user shared memory buffer. Each of these elements occupies 4 bytes in the user shared memory buffer, and is located starting at 4****i*** addresses past the beginning of the buffer (which is located at the address in **Sys.pushm**). This array is for general-purpose use in applications; it has no automatic use.

Sys.Udata[*i*] is identical to **Sys.Uhex[*i*]**, except that when its value is queried in the Script environment, Power PMAC reports its value as a decimal number (e.g. `256`) instead of a decimal number (e.g. `$100`).

Index values ***i*** in the square brackets can be integer constants in the range 0 to 16,777,215, or local L-variables. No expressions or non-integer constants are permitted. The size of the user shared memory buffer is set as a directive in the `pp_proj.ini` file, 1 megabyte (1,048,576 bytes) by default, which makes the maximum valid index value 262,143.

Sys.Udata[*i*] is located in the same registers as **Sys.Cdata[*i/4*]** to **Sys.Cdata[*i/4+3*]**, **Sys.Ddata[*i/2*]**, **Sys.Fdata[*i*]**, **Sys.Idata[*i*]**, and **Sys.Uhex[*i*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

In C, this element should be accessed as a pointer to a `unsigned int` variable, offset from the `pushm` base address by ***i*** addresses (e.g. `MyUintPtr = (int *) pushm + i`).

Sys.Uhex[*i*]

Description: User shared memory "unsigned hex integer" data array element

Range: 0 .. 2^{32}-1

Units: User set

Power-on default: 0

Sys.Uhex[*i*] is the "***i***th" unsigned 32-bit integer data array element in the user shared memory buffer. Each of these elements occupies 4 bytes in the user shared memory buffer, and is located starting at 4***i*** addresses past the beginning of the buffer (which is located at the address in **Sys.pushm**). This array is for general-purpose use in applications; it has no automatic use.

Sys.Uhex[*i*] is identical to **Sys.Udata[*i*]**, except that when its value is queried in the Script environment, Power PMAC reports its value as a hexadecimal number (e.g. `$100`) instead of a decimal number (e.g. `256`).

Index values ***i*** in the square brackets can be integer constants in the range 0 to 16,777,215, or local L-variables. No expressions or non-integer constants are permitted. The size of the user shared memory buffer is set as a directive in the `pp_proj.ini` file, 1 megabyte (1,048,576 bytes) by default, which makes the maximum valid index value 262,143.

Sys.Uhex[*i*] is located in the same registers as **Sys.Cdata[*i/4*]** to **Sys.Cdata[*i/4+3*]**, **Sys.Ddata[*i/2*]**, **Sys.Fdata[*i*]**, **Sys.Idata[*i*]**, and **Sys.Udata[*i*]**. It is the user's responsibility to prevent possible multiple uses of the same register.

In C, this element should be accessed as a pointer to a `unsigned int` variable, offset from the `pushm` base address by ***i*** addresses (e.g. `MyUintPtr = (int *) pushm + i`).

Sys.WpKey

Description: Write-protect key value for DSPGATE3 ASICs

Range: 0 .. $FFFFFFFF

Units: none

Power-on default: 0

Sys.WpKey specifies the value that is automatically copied to the write-protect-key register **Gate3[*i*].WpKey** of DSPGATE3 machine-interface ASIC ***i*** when a script command performs a write operation to a write-protected register in that ASIC. **Sys.WpKey** must be set to a value of $AAAAAAAA for the script-language write operation to the write-protected register to be successful. After such a write operation, the write-protect-key register in the ASIC is automatically cleared to 0, but the value of this element is not changed.

This element is not used when C-language commands write to a write-protected register in such an ASIC. In C, the program must explicitly write the key value to the **Gate3[*i*].WpKey** register before each write operation to a write-protected register in the ASIC.

This element permits the machine developer to change these write-protected registers easily during the project development, but prevent unauthorized users from making changes later on.

Tdata[*i*]. Non-Saved Setup Data Structure Elements

This section describes useful setup elements within the transformation-matrix structure whose values are not copied to flash memory on a **save** command.

Tdata[*i*].Bias[*m*]

Description: Transformation matrix offset term

Range: floating-point

Units: (Untransformed) axis units

Power-on default: 0.0

Tdata[*i*].Bias[*m*] is the offset term in transformation matrix ***i*** for the axis with index ***m***. Its value is added to the programmed (transformed) axis position to get the base (untransformed) axis position. The transformation-initialization program command **tiniti** causes the bias terms for all axes in the transformation matrix ***i*** to be set to 0.0.

There are 256 transformation matrices, with index values ***i*** from 0 to 255. There are 32 axes, with index values ***m*** from 0 to 31. The following table shows the axis index ***m*** for each of the 32 axis names.

Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*
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

When any term in the active (selected) transformation matrix is changed, or a new transformation matrix is selected for the coordinate system, a **pmatch** command should be executed for the coordinate system before the next programmed move is commanded (one is executed automatically on a command to start motion program execution).

For more details, refer to the section *Axis Transformation Matrices* in the chapter *Setting Up Coordinate Systems* in the Power PMAC User's Manual.

Tdata[*i*].Diag[*m*]

Description: Transformation matrix scaling term

Range: floating-point

Units: Untransformed axis units per transformed axis unit

Power-on default: 1.0

Tdata[*i*].Diag[*m*] is the (diagonal) scaling term in transformation matrix ***i*** for the axis with index ***m***. Its value is multiplied by the programmed (transformed) axis position and the product is added to other terms to get the base (untransformed) axis position. The transformation-initialization program command **`tiniti`** causes the diagonal terms for all axes in the transformation matrix ***i*** to be set to 1.0.

There are 256 transformation matrices, with index values ***i*** from 0 to 255. There are 32 axes, with index values ***m*** from 0 to 31. The following table shows the axis index ***m*** for each of the 32 axis names.

Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*	Axis	*m*
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

When any term in the active (selected) transformation matrix is changed, or a new transformation matrix is selected for the coordinate system, a **`pmatch`** command should be executed for the coordinate system before the next programmed move is commanded (one is executed automatically on a command to start motion program execution).

For more details, refer to the section *Axis Transformation Matrices* in the chapter *Setting Up Coordinate Systems* in the Power PMAC User's Manual.

Tdata[*i*].UUVVWW[*m*]

Description: Transformation matrix UU/VV/WW-axis cross-coupling term

Range: floating-point

Units: Untransformed axis units per transformed axis unit

Power-on default: 0.0

Tdata[*i*].UUVVWW[*m*] is one of the (off-diagonal) cross-coupling terms in transformation matrix ***i*** for the UU/VV/WW axis triplet. There are 6 off-diagonal terms in the 3x3 cross-coupling sub-matrix for this axis triplet. The value of an off-diagonal term is multiplied by the transformed position of one axis of the triplet, and the product is added into the value of the untransformed position of another axis of the triplet. The cross-coupling terms are primarily used for rotations in the 3D axis space.

There are 256 transformation matrices, with index values ***i*** from 0 to 255.

The complete sub-matrix for the triplet can be seen in the following matrix equation, where the axis names on the left are the untransformed positions, and the axis names on the right (with the "prime" marker) are the transformed positions:

$$\begin{bmatrix} UU \\ VV \\ WW \end{bmatrix} = \begin{bmatrix} Diag[26] & UUVVWW[0] & UUVVWW[1] \\ UUVVWW[2] & Diag[27] & UUVVWW[3] \\ UUVVWW[4] & UUVVWW[5] & Diag[28] \end{bmatrix} \begin{bmatrix} UU' \\ VV' \\ WW' \end{bmatrix} + \begin{bmatrix} Bias[26] \\ Bias[27] \\ Bias[28] \end{bmatrix}$$

The transformation-initialization program command **tiniti** causes the off-diagonal terms for all axis triplets in the transformation matrix ***i*** to be set to 0.0.

When any term in the active (selected) transformation matrix is changed, or a new transformation matrix is selected for the coordinate system, a **pmatch** command should be executed for the coordinate system before the next programmed move is commanded (one is executed automatically on a command to start motion program execution).

For more details, refer to the section *Axis Transformation Matrices* in the chapter *Setting Up Coordinate Systems* in the Power PMAC User's Manual.

Tdata[*i*].UVW[*m*]

Description: Transformation matrix U/V/W-axis cross-coupling term

Range: floating-point

Units: Untransformed axis units per transformed axis unit

Power-on default: 0.0

Tdata[*i*].UVW[*m*] is one of the (off-diagonal) cross-coupling terms in transformation matrix ***i*** for the U/V/W axis triplet. There are 6 off-diagonal terms in the 3x3 cross-coupling sub-matrix for this axis triplet. The value of an off-diagonal term is multiplied by the transformed position of one axis of the triplet, and the product is added into the value of the untransformed position of another axis of the triplet. The cross-coupling terms are primarily used for rotations in the 3D axis space.

There are 256 transformation matrices, with index values ***i*** from 0 to 255.

The complete sub-matrix for the triplet can be seen in the following matrix equation, where the axis names on the left are the untransformed positions, and the axis names on the right (with the "prime" marker) are the transformed positions:

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} Diag[3] & UVW[0] & UVW[1] \\ UVW[2] & Diag[4] & UVW[3] \\ UVW[4] & UVW[5] & Diag[5] \end{bmatrix} \begin{bmatrix} U' \\ V' \\ W' \end{bmatrix} + \begin{bmatrix} Bias[3] \\ Bias[4] \\ Bias[5] \end{bmatrix}$$

The transformation-initialization program command **tiniti** causes the off-diagonal terms for all axis triplets in the transformation matrix ***i*** to be set to 0.0.

When any term in the active (selected) transformation matrix is changed, or a new transformation matrix is selected for the coordinate system, a **pmatch** command should be executed for the

coordinate system before the next programmed move is commanded (one is executed automatically on a command to start motion program execution).

For more details, refer to the section *Axis Transformation Matrices* in the chapter *Setting Up Coordinate Systems* in the Power PMAC User's Manual.

Tdata[*i*].XXYYZZ[*m*]

Description: Transformation matrix XX/YY/ZZ-axis cross-coupling term

Range: floating-point

Units: Untransformed axis units per transformed axis unit

Power-on default: 0.0

Tdata[*i*].XXYYZZ[*m*] is one of the (off-diagonal) cross-coupling terms in transformation matrix ***i*** for the XX/YY/ZZ axis triplet. There are 6 off-diagonal terms in the 3x3 cross-coupling sub-matrix for this axis triplet. The value of an off-diagonal term is multiplied by the transformed position of one axis of the triplet, and the product is added into the value of the untransformed position of another axis of the triplet. The cross-coupling terms are primarily used for rotations in the 3D axis space.

There are 256 transformation matrices, with index values ***i*** from 0 to 255.

The complete sub-matrix for the triplet can be seen in the following matrix equation, where the axis names on the left are the untransformed positions, and the axis names on the right (with the "prime" marker) are the transformed positions:

$$\begin{bmatrix} XX \\ YY \\ ZZ \end{bmatrix} = \begin{bmatrix} Diag[29] & XXYYZZ[0] & XXYYZZ[1] \\ XXYYZZ[2] & Diag[30] & XXYYZZ[3] \\ XXYYZZ[4] & XZYYZZ[5] & Diag[31] \end{bmatrix} \begin{bmatrix} XX' \\ YY' \\ ZZ' \end{bmatrix} + \begin{bmatrix} Bias[29] \\ Bias[30] \\ Bias[31] \end{bmatrix}$$

The transformation-initialization program command **`tiniti`** causes the off-diagonal terms for all axis triplets in the transformation matrix ***i*** to be set to 0.0.

When any term in the active (selected) transformation matrix is changed, or a new transformation matrix is selected for the coordinate system, a **`pmatch`** command should be executed for the coordinate system before the next programmed move is commanded (one is executed automatically on a command to start motion program execution).

For more details, refer to the section *Axis Transformation Matrices* in the chapter *Setting Up Coordinate Systems* in the Power PMAC User's Manual.

Tdata[*i*].XYZ[*m*]

Description: Transformation matrix X/Y/Z-axis cross-coupling term

Range: floating-point

Units: Untransformed axis units per transformed axis unit

Power-on default: 0.0

Tdata[*i*].XYZ[*m*] is one of the (off-diagonal) cross-coupling terms in transformation matrix ***i*** for the X/Y/Z axis triplet. There are 6 off-diagonal terms in the 3x3 cross-coupling sub-matrix for this axis triplet. The value of an off-diagonal term is multiplied by the transformed position of one axis of the triplet, and the product is added into the value of the untransformed position of another axis of the triplet. The cross-coupling terms are primarily used for rotations in the 3D axis space.

There are 256 transformation matrices, with index values ***i*** from 0 to 255.

The complete sub-matrix for the triplet can be seen in the following matrix equation, where the axis names on the left are the untransformed positions, and the axis names on the right (with the "prime" marker) are the transformed positions:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} Diag[6] & XYZ[0] & XYZ[1] \\ XYZ[2] & Diag[7] & XYZ[3] \\ XYZ[4] & XYZ[5] & Diag[8] \end{bmatrix} \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} + \begin{bmatrix} Bias[6] \\ Bias[7] \\ Bias[8] \end{bmatrix}$$

The transformation-initialization program command **`tiniti`** causes the off-diagonal terms for all axis triplets in the transformation matrix ***i*** to be set to 0.0.

When any term in the active (selected) transformation matrix is changed, or a new transformation matrix is selected for the coordinate system, a **`pmatch`** command should be executed for the coordinate system before the next programmed move is commanded (one is executed automatically on a command to start motion program execution).

For more details, refer to the section *Axis Transformation Matrices* in the chapter *Setting Up Coordinate Systems* in the Power PMAC User's Manual.

UserAlgo Non-Saved Setup Data Structure Elements

This section describes useful setup elements within the user-algorithm control structure whose values are not copied to flash memory on a **save** command.

UserAlgo.BgCplc[*i*]

Description: Background C PLC program *i* enable

Range: 0 .. 1

Units: Boolean

Power-on default: 0

UserAlgo.BgCplc[*i*] controls whether the C-language background PLC program CPLC*i* is enabled or not. If it is set to its power-on default value of 0, the program is disabled and will not be executed even if loaded into the Power PMAC. If it is set to 1, the program is enabled and will be executed each Power PMAC background software cycle.

The value of this element can be set to 1 with the **enable bgcplc *i*** command, and set to 0 with the **disable bgcplc *i*** command.

If it is desired to have this program executing immediately on power-on/reset, a command setting the value of this element to 1 should be included in the `pp_startup.txt` project configuration file.

UserAlgo.CaptCompIntr

Description: Capture/compare interrupt service routine enable

Range: 0 .. 1

Units: Boolean

Power-on default: 0

UserAlgo.CaptCompIntr controls whether the C-language capture/compare interrupt service routine is enabled or not. If it is set to its power-on default value of 0, the program is disabled and will not be executed even if loaded into the Power PMAC. If it is set to 1, the program is enabled and will be executed each time a PMAC3-style DSPGATE3 IC sends a capture or compare interrupt to the processor.

If it is desired to have this program executing immediately on power-on/reset, a command setting the value of this element to 1 should be included in the `pp_startup.txt` project configuration file.

UserAlgo.CFunc

Description: CfromScript function enable

Range: 0 .. 1

Units: Boolean

Power-on default: 0

UserAlgo.CFunc controls whether the C-language function "CfromScript" is enabled from background programs or not. If it is set to its power-on default value of 0, the function is disabled in background and will not be executed even if loaded into the Power PMAC. If it is set to 1, the function is enabled and will be executed each time it is called from a Power PMAC Script program.

If the "CfromScript" function is called from a foreground Script program, including foreground PLC programs and kinematics routines executed by motion programs, it is not necessary that **UserAlgo.CFunc** be set to 1. If the function is called from a background Script program, including background PLC programs and kinematics routines executed by on-line query commands or background PLC program commands, it is necessary that **UserAlgo.CFunc** be set to 1.

If it is desired to have this program able to execute immediately on power-on/reset, a command setting the value of this element to 1 should be included in the `pp_startup.txt` project configuration file.

UserAlgo.RtiCplc

Description: Foreground C PLC program enable

Range: 0 .. 1

Units: Boolean

Power-on default: 0

UserAlgo.RtiCplc controls whether the C-language foreground PLC program is enabled or not. If it is set to its power-on default value of 0, the program is disabled and will not be executed even if loaded into the Power PMAC. If it is set to 1, the program is enabled and will be executed each Power PMAC real-time interrupt software cycle.

The value of this element can be set to 1 with the **`enable rticplc`** command, and set to 0 with the **`disable rticplc`** command.

If it is desired to have this program executing immediately on power-on/reset, a command setting the value of this element to 1 should be included in the `pp_startup.txt` project configuration file.

POWER PMAC STATUS DATA STRUCTURE ELEMENTS

This chapter documents data structure elements that the user will typically only read. Most are "write-protected" in the Script environment. Those few elements for which there can be a valid purpose in writing to them are noted.

Acc5E[*i*]. Status Data Structure Elements

The **Acc5E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate2[*i*]** data structure. The data structure elements for the ACC-5E MACRO interface board are listed under the **Gate2[*i*]** data structure, below.

Acc5E3[*i*]. Status Data Structure Elements

The **Acc5E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-5E3 MACRO interface board are listed under the **Gate3[*i*]** data structure, below.

Acc5EP3[*i*]. Status Data Structure Elements

The **Acc5EP3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-5EP3 Etherlite MACRO interface board are listed under the **Gate3[*i*]** data structure, below.

Acc11C[*i*]. Status Data Structure Elements

The **Acc11C[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-11C digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc11E[*i*]. Status Data Structure Elements

The **Acc11E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-11E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Note that the ACC-11E cannot be auto-identified by the Power PMAC CPU, so to use this data structure, the user must manually set **GateIo[*i*].PartNum** to 603307, **GateIo[*i*].PartType** to 8, issue a save command, and reset the controller, before being able to use this structure. This structure is new in V2.0 firmware, released 1st quarter 2015.

Acc14E[*i*]. Status Data Structure Elements

The **Acc14E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-14E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc24C2[*i*]. Status Data Structure Elements

The **Acc24C2[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24C2 digital servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24C2A[*i*]. Status Data Structure Elements

The **Acc24C2A[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24C2A analog servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2[*i*]. Status Data Structure Elements

The **Acc24E2[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2 digital servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2A[*i*]. Status Data Structure Elements

The **Acc24E2A[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2A analog servo interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E2S[*i*]. Status Data Structure Elements

The **Acc24E2S[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-24E2S stepper interface board are listed under the **Gate1[*i*]** data structure, below.

Acc24E3[*i*]. Status Data Structure Elements

The **Acc24E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-24E3 servo interface board are listed under the **Gate3[*i*]** data structure, below.

Acc28E[*i*]. Status Data Structure Elements

The ACC-28E is a 4-channel 16-bit A/D converter board for UMAC systems. This section documents the read-only status elements for this accessory.

Acc28E[*i*].AdcSdata[*j*]

Description: ACC-28E signed ADC reading of index ***j***

Range: -32,768 .. 32,767

Units: 16-bit ADC units

Acc28E[*i*].AdcSdata[*j*] contains the most recent digital value produced by the analog-to-digital converter with channel index ***j*** on the ACC-28E with board index ***i***. The channel index ***j*** can take a value from 0 to 3, corresponding to hardware inputs ADC1 to ADC4, respectively. Note that the channel index ***j*** is one less than the corresponding hardware input line number.

Acc28E[*i*].AdcSdata[*j*] treats the digital number as a signed 16-bit value, with -32,768 corresponding to a -10V input, 0 corresponding to a 0V input, and 32,767 corresponding to a +10V input. To treat the digital number as an unsigned 16-bit value, use **Acc28E[*i*].AdcUdata[*j*]** instead.

Note

The ADC data from an ACC-28E is always present in the register as an unsigned 16-bit value (range 0 to 65,535) in the high 16 bits of the 32-bit bus. When accessed through this element, Power PMAC subtracts 32,768 from the value in the register so it appears to the user as a signed 16-bit value. When accessed through data gathering or a C program, this subtraction does not automatically occur.

Acc28E[*i*].AdcSdata[*j*] will report as “nan” (not-a-number) if no board with this index is present.

Acc28E[*i*].AdcUdata[*j*]

Description: ACC-28E unsigned ADC reading of index ***j***

Range: 0 .. 65,535

Units: 16-bit ADC units

Acc28E[*i*].AdcUdata[*j*] contains the most recent digital value produced by the analog-to-digital converter with channel index ***j*** on the ACC-28E with board index ***i***. The channel index ***j*** can take a value from 0 to 3, corresponding to hardware inputs ADC1 to ADC4, respectively. Note that the channel index ***j*** is one less than the corresponding hardware input line number.

Acc28E[*i*].AdcUdata[*j*] treats the digital number as an unsigned 16-bit value, with 0 corresponding to a -10V input, 32,768 corresponding to a 0V input, and 65,535 corresponding to a +10V input. To treat the digital number as a signed 16-bit value, use **Acc28E[*i*].AdcSdata[*j*]** instead.

Acc28E[*i*].AdcUdata[*j*] will report as "nan" (not-a-number) if no board with this index is present.

Acc28E[*i*].PartNum

Description: Hardware part number for ACC-28E ADC board

Range: Non-negative integer

Units: Enumeration

Acc28E[*i*].PartNum contains the six-digit part number of the ACC-28E A/D converter board. It is derived from an identification IC present on the board, which is read automatically at power-up/reset. It will report as 0 if no ACC-28E with this index is present.

The part number for the ACC-28E is 603404.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc28E[*i*].PartOpt

Description: Optional hardware configuration code for ACC-28E ADC board

Range: Non-negative integer

Units: Enumeration

Acc28E[*i*].PartOpt contains a code indicating what optional hardware is present on the ACC-28E board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It will report as "nan" (not-a-number) if no board with this index is present.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc28E[*i*].PartRev

Description: Hardware revision number for ACC-28E ADC board

Range: Non-negative integer

Units: Enumeration

Acc28E[*i*].PartRev contains the revision number of the ACC-28E ADC board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc28E[*i*].PartType

Description: Hardware part type for ACC-28E ADC board

Range: Non-negative integer

Units: Bit field

Acc28E[*i*].PartType contains an integer representing the type(s) of hardware interface of the ACC-28E ADC board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no board with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present on the board. The interface types that could be present in general are:

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

The value of **PartType** for all ACC-28E ADC boards is 4 (analog I/O interface only).

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc36E[*i*]. Status Data Structure Elements

The ACC-36E is a 16-channel 12-bit multiplexed A/D converter board for UMAC systems. This section documents the read-only status elements for this accessory.

Acc36E[*i*].ADCHighLow

Description: ACC-36E combined-pair ADC reading

Range: 0 .. 16,777,215

Units: 12-bit ADC units in low and high halves

Acc36E[*i*].ADCHighLow contains the most recent combined digital value produced by the selected pair of multiplexed analog-to-digital converters on the ACC-36E with board index ***i***. It is a 24-bit value, with the high and low halves each containing the value reported from the selected 12-bit ADC. Which of the 8 pairs of ADCs on the ACC-36E is read, and whether each half is a signed or unsigned 12-bit value depends of the value of **Acc36E[*i*].ConvertCode** that triggered the conversion.

The values of the individual multiplexed ADCs can be read in elements **Acc36E[*i*].ADCsHigh**, **Acc36E[*i*].ADCsLow**, **Acc36E[*i*].ADCuHigh**, **Acc36E[*i*].ADCuLow**. In actual operation, most users will read the automatically demultiplexed values in **AdcDemux.ResultHigh[*j*]** and **AdcDemux.ResultLow[*j*]**.

Acc36E[*i*].ADCHighLow will report as "nan" (not-a-number) if no board with this index is present.

Acc36E[*i*].ADCRdyHigh

Description: ACC-36E "high" ADC ready status bit

Range: 0 .. 1

Units: Boolean

Acc36E[*i*].ADCRdyHigh contains the status of the most recently specified conversion for the selected "high" A/D converter. Its value is 0 if the conversion is ongoing and the result is not ready yet; its value is 1 if the conversion is complete and the result is ready to be read in **Acc36E[*i*].ADCsHigh** or **Acc36E[*i*].ADCuHigh**.

Acc36E[*i*].ADCRdyHigh will report as "nan" (not-a-number) if no board with this index is present.

Acc36E[*i*].ADCRdyLow

Description: ACC-36E "low" ADC ready status bit

Range: 0 .. 1

Units: Boolean

Acc36E[*i*].ADCRdyLow contains the status of the most recently specified conversion for the selected "low" A/D converter. Its value is 0 if the conversion is ongoing and the result is not ready yet; its value is 1 if the conversion is complete and the result is ready to be read in **Acc36E[*i*].ADCsLow** or **Acc36E[*i*].ADCuLow**.

Acc36E[*i*].ADCRdyLow will report as "nan" (not-a-number) if no board with this index is present.

Acc36E[*i*].ADCsHigh

Description: ACC-36E signed multiplexed "high" ADC reading

Range: -2048 .. 2047

Units: 12-bit ADC units

Acc36E[*i*].ADCsHigh contains the most recent digital value produced by the multiplexed analog-to-digital converter in the high half of the data bus on the ACC-36E with board index ***i***. It is a signed 12-bit value. Which of the 8 "high" ADCs on the ACC-36E is read, and whether the result is a signed or unsigned 12-bit value depends of the value of **Acc36E[*i*].ConvertCode** that triggered the conversion.

In actual operation, most users will read the automatically demultiplexed values in **AdcDemux.ResultHigh[*j*]**.

Acc36E[*i*].ADCsHigh will report as "nan" (not-a-number) if no board with this index is present.

Acc36E[*i*].ADCsLow

Description: ACC-36E signed multiplexed "low" ADC reading

Range: 0 .. 4095

Units: 12-bit ADC units

Acc36E[*i*].ADCsLow contains the most recent digital value produced by the multiplexed analog-to-digital converter in the low half of the data bus on the ACC-36E with board index ***i***. It is a signed 12-bit value. Which of the 8 "low" ADCs on the ACC-36E is read, and whether the result is a signed or unsigned 12-bit value depends of the value of **Acc36E[*i*].ConvertCode** that triggered the conversion.

In actual operation, most users will read the automatically demultiplexed values in **AdcDemux.ResultLow[*j*]**.

Acc36E[*i*].ADCsLow will report as "nan" (not-a-number) if no board with this index is present.

Acc36E[*i*].ADCuHigh

Description: ACC-36E unsigned multiplexed "high" ADC reading

Range: 0 .. 4095

Units: 12-bit ADC units

Acc36E[*i*].ADCuHigh contains the most recent digital value produced by the multiplexed analog-to-digital converter in the high half of the data bus on the ACC-36E with board index ***i***. It is an unsigned 12-bit value. Which of the 8 "high" ADCs on the ACC-36E is read, and whether the result is a signed or unsigned 12-bit value depends of the value of **Acc36E[*i*].ConvertCode** that triggered the conversion.

In actual operation, most users will read the automatically demultiplexed values in **AdcDemux.ResultHigh[*j*]**.

Acc36E[*i*].ADCuHigh will report as "nan" (not-a-number) if no board with this index is present.

Acc36E[*i*].ADCuLow

Description: ACC-36E unsigned multiplexed "low" ADC reading

Range: 0 .. 4095

Units: 12-bit ADC units

Acc36E[*i*].ADCuLow contains the most recent digital value produced by the multiplexed analog-to-digital converter in the low half of the data bus on the ACC-36E with board index ***i***. It is an unsigned 12-bit value. Which of the 8 "low" ADCs on the ACC-36E is read, and whether the result is a signed or unsigned 12-bit value depends of the value of **Acc36E[*i*].ConvertCode** that triggered the conversion.

In actual operation, most users will read the automatically demultiplexed values in **AdcDemux.ResultLow[*j*]**.

Acc36E[*i*].ADCuLow will report as "nan" (not-a-number) if no board with this index is present.

Acc36E[*i*].PartNum

Description: Hardware part number for ACC-36E ADC board

Range: Non-negative integer

Units: Enumeration

Acc36E[*i*].PartNum contains the six-digit part number of the ACC-36E A/D converter board. It is derived from an identification IC present on the board, which is read automatically at power-up/reset. It will report as 0 if no ACC-36E with this index is present.

The part number for the ACC-36E is 603483.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc36E[*i*].PartOpt

Description: Optional hardware configuration code for ACC-36E ADC board

Range: Non-negative integer

Units: Enumeration

Acc36E[*i*].PartOpt contains a code indicating what optional hardware is present on the ACC-36E board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It will report as "nan" (not-a-number) if no board with this index is present.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc36E[*i*].PartRev

Description: Hardware revision number for ACC-36E ADC board

Range: Non-negative integer

Units: Enumeration

Acc36E[*i*].PartRev contains the revision number of the ACC-36E ADC board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc36E[*i*].PartType

Description: Hardware part type for ACC-36E ADC board

Range: Non-negative integer

Units: Bit field

Acc36E[*i*].PartType contains an integer representing the type(s) of hardware interface of the ACC-36E ADC board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no board with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present on the board. The interface types that could be present in general are:

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

The value of **PartType** for all ACC-36E ADC boards is 4 (analog I/O interface only).

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc51C[*i*]. Status Data Structure Elements

The **Acc51C[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-51C sine-encoder interface board are listed under the **Gate1[*i*]** data structure, below.

Acc51E[*i*]. Status Data Structure Elements

The **Acc51E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-51E sine-encoder interface board are listed under the **Gate1[*i*]** data structure, below.

Acc53E[i]. Status Data Structure Elements

The ACC-53E is an 8-channel SSI-encoder interface board for UMAC systems. This section documents the read-only status elements for this accessory.

Acc53E[*i*].EncData[*j*]

Description: ACC-53E encoder position of channel index ***j***

Range: 0 .. 16,777,215

Units: Encoder LSBs

Acc53E[*i*].EncData[*j*] contains the most recent numerical value read from the SSI encoder with channel index ***j*** on the ACC-53E with board index ***i***. The channel index ***j*** can take a value from 0 to 7, corresponding to encoders 1 to 8, respectively. Note that the channel index ***j*** is one less than the corresponding encoder number.

Acc53E[*i*].EncData[*j*] will report as "nan" (not-a-number) if no board with this index is present.

Acc53E[*i*].PartNum

Description: Hardware part number for ACC-53E SSI board

Range: Non-negative integer

Units: Enumeration

Acc53E[*i*].PartNum contains the six-digit part number of the ACC-53E SSI-encoder interface board. It is derived from an identification IC present on the board, which is read automatically at power-up/reset. It will report as 0 if no ACC-53E with this index is present.

The part number for the ACC-53E is 603360.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc53E[*i*].PartOpt

Description: Optional hardware configuration code for ACC-53E SSI board

Range: Non-negative integer

Units: Enumeration

Acc53E[*i*].PartOpt contains a code indicating what optional hardware is present on the ACC-53E board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It will report as "nan" (not-a-number) if no board with this index is present.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc53E[*i*].PartRev

Description: Hardware revision number for ACC-53E SSI board

Range: Non-negative integer

Units: Enumeration

Acc53E[*i*].PartRev contains the revision number of the ACC-53E SSI board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc53E[*i*].PartType

Description: Hardware part type for ACC-53E SSI board

Range: Non-negative integer

Units: Bit field

Acc53E[*i*].PartType contains an integer representing the type(s) of hardware interface of the ACC-53E SSI board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no board with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present on the board. The interface types that could be present in general are:

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

The value of **PartType** for the ACC-53E SSI board is 9 (servo and digital I/O interface).

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc58E[*i*]. Status Data Structure Elements

The **Acc58E[*i*]** data structure name is an alias in the Script environment for the underlying **Gate1[*i*]** data structure. The data structure elements for the ACC-58E resolver-to-digital converter board are listed under the **Gate1[*i*]** data structure, below.

Acc59E[i]. Status Data Structure Elements

The ACC-59E is an 8-channel 12-bit multiplexed A/D and D/A converter board for UMAC systems. This section documents the read-only status elements for this accessory.

Acc59E[*i*].ADCRdy

Description: ACC-59E ADC ready status bit

Range: 0 .. 1

Units: Boolean

Acc59E[*i*].ADCRdy contains the status of the most recently specified conversion for the selected A/D converter. Its value is 0 if the conversion is ongoing and the result is not ready yet; its value is 1 if the conversion is complete and the result is ready to be read in **Acc59E[*i*].ADCs** or **Acc59E[*i*].ADCu**.

Acc59E[*i*].ADCRdy will report as "nan" (not-a-number) if no board with this index is present.

Acc59E[*i*].ADCs

Description: ACC-59E multiplexed signed ADC reading

Range: -2048 .. 2047

Units: 12-bit ADC units

Acc59E[*i*].ADCs contains the most recent digital value produced by the multiplexed analog-to-digital converter on the ACC-59E with board index ***i***. It is a signed 12-bit value. Which of the 8 ADCs on the ACC-59E is read, and whether the result is a signed or unsigned 12-bit value depends of the value of **Acc59E[*i*].ConvertCode** that triggered the conversion.

In actual operation, most users will read the automatically demultiplexed values in **AdcDemux.ResultHigh[*j*]**.

Acc59E[*i*].ADCs will report as "nan" (not-a-number) if no board with this index is present.

Acc59E[*i*].ADCu

Description: Acc-59E multiplexed unsigned ADC reading

Range: 0 .. 4095

Units: 12-bit ADC units

Acc59E[*i*].ADCu contains the most recent digital value produced by the multiplexed analog-to-digital converter on the ACC-59E with board index ***i***. It is an unsigned 12-bit value. Which of the 8 "high" ADCs on the ACC-36E is read, and whether the result is a signed or unsigned 12-bit value depends of the value of **Acc59E[*i*].ConvertCode** that triggered the conversion.

In actual operation, most users will read the automatically demultiplexed values in **AdcDemux.ResultHigh[*j*]**.

Acc59E[*i*].ADCu will report as "nan" (not-a-number) if no board with this index is present.

Acc59E[*i*].PartNum

Description: Hardware part number for ACC-59E ADC/DAC board

Range: Non-negative integer

Units: Enumeration

Acc59E[*i*].PartNum contains the six-digit part number of the ACC-59E A/D converter board. It is derived from an identification IC present on the board, which is read automatically at power-up/reset. It will report as 0 if no ACC-59E with this index is present.

The part number for the ACC-59E is 603494.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc59E[*i*].PartOpt

Description: Optional hardware configuration code for ACC-59E ADC/DAC board

Range: Non-negative integer

Units: Enumeration

Acc59E[*i*].PartOpt contains a code indicating what optional hardware is present on the ACC-59E board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It will report as "nan" (not-a-number) if no board with this index is present.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc59E[*i*].PartRev

Description: Hardware revision number for ACC-59E ADC/DAC board

Range: Non-negative integer

Units: Enumeration

Acc59E[*i*].PartRev contains the revision number of the ACC-59E ADC/DAC board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc59E[*i*].PartType

Description: Hardware part type for ACC-59E ADC/DAC board

Range: Non-negative integer

Units: Bit field

Acc59E[*i*].PartType contains an integer representing the type(s) of hardware interface of the ACC-59E ADC board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no board with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present on the board. The interface types that could be present in general are:

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

The value of **PartType** for all ACC-59E ADC boards is 4 (analog I/O interface only).

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc59E3[*i*]. Status Data Structure Elements

The **Acc59E3[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the ACC-59E3 A/D and D/A-converter board are listed under the **Gate3[*i*]** data structure, below.

Acc65E[*i*]. Status Data Structure Elements

The **Acc65E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-65E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc66E[*i*]. Status Data Structure Elements

The **Acc66E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-66E digital input board are listed under the **GateIo[*i*]** data structure, below.

Acc67E[*i*]. Status Data Structure Elements

The **Acc67E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-67E digital output board are listed under the **GateIo[*i*]** data structure, below.

Acc68E[*i*]. Status Data Structure Elements

The **Acc68E[*i*]** data structure name is an alias in the Script environment for the underlying **GateIo[*i*]** data structure. The data structure elements for the ACC-68E digital I/O board are listed under the **GateIo[*i*]** data structure, below.

Acc72EX[*i*]. Status Data Structure Elements

The ACC-72EX is a fieldbus interface board for UMAC systems that can support a variety of different protocols. This section documents the read-only status elements for this accessory.

Acc72EX[*i*].PartNum

Description: Hardware part number for ACC-72EX fieldbus board

Range: Non-negative integer

Units: Enumeration

Acc72EX[*i*].PartNum contains the six-digit part number of the ACC-72EX fieldbus interface board. It is derived from an identification IC present on the board, which is read automatically at power-up/reset. It will report as 0 if no ACC-72EX with this index is present.

The part number for the ACC-72EX is 603958.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc72EX[*i*].PartOpt

Description: Optional hardware configuration code for ACC-72EX fieldbus board

Range: Non-negative integer

Units: Enumeration

Acc72EX[*i*].PartOpt contains a code indicating what optional hardware is present on the ACC-72EX board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It will report as “nan” (not-a-number) if no board with this index is present.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc72EX[*i*].PartRev

Description: Hardware revision number for ACC-72EX fieldbus board

Range: Non-negative integer

Units: Enumeration

Acc72EX[*i*].PartRev contains the revision number of the ACC-72EX fieldbus interface board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc72EX[*i*].PartType

Description: Hardware part type for ACC-84E serial-encoder board

Range: Non-negative integer

Units: Bit field

Acc72EX[*i*].PartType contains an integer representing the type(s) of hardware interface of the ACC-72EX fieldbus interface board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no board with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present on the board. The interface types that could be present in general are:

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

The value of **PartType** for the ACC-72EX fieldbus interface board is 8 (digital I/O interface).

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc84B[*i*]. Status Data Structure Elements

The **Acc84B[*i*]** data structure for the serial-encoder add-in board for the Power Brick product line is equivalent to the **Acc84E[*i*]** data structure that is used for the rack-mounted Power UMAC serial-encoder interface board that is documented below. It is *not* an alias for the **Acc84E[*i*]** data structure name, as it can only be used for the Brick products.

Acc84C[*i*]. Status Data Structure Elements

The **Acc84C[*i*]** data structure for the serial-encoder board for the Compact UMAC product line is equivalent to the **Acc84E[*i*]** data structure that is used for the rack-mounted Power UMAC serial-encoder interface board that is documented below. It is *not* an alias for the **Acc84E[*i*]** data structure name, as it can only be used for the Compact UMAC products.

Acc84E[*i*]. Status Data Structure Elements

The ACC-84E is a 4-channel serial-encoder interface board for UMAC systems that can support a variety of different protocols. This section documents the read-only status elements for this accessory.

Acc84E[*i*]. Multi-Channel Status Elements

Some aspects of the serial-encoder interface are common for all channels.

Acc84E[*i*].PartNum

Description: Hardware part number for ACC-84E serial-encoder board

Range: Non-negative integer

Units: Enumeration

Acc84E[*i*].PartNum contains the six-digit part number of the ACC-84E serial-encoder interface board. It is derived from an identification IC present on the board, which is read automatically at power-up/reset. It will report as 0 if no ACC-84E with this index is present.

The part number for the ACC-84E is 603927. (The part number for the ACC-84B is 604048; the part number for the ACC-84C is 603929; the part number for the ACC-84S is 603936.)

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc84E[*i*].PartOpt

Description: Optional hardware configuration code for ACC-84E serial-encoder board

Range: Non-negative integer

Units: Enumeration

Acc84E[*i*].PartOpt contains a code indicating what optional hardware is present on the ACC-84E board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It will report as “nan” (not-a-number) if no board with this index is present.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc84E[*i*].PartRev

Description: Hardware revision number for ACC-84E serial-encoder board

Range: Non-negative integer

Units: Enumeration

Acc84E[*i*].PartRev contains the revision number of the ACC-84E serial-encoder interface board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc84E[*i*].PartType

Description: Hardware part type for ACC-84E serial-encoder board

Range: Non-negative integer

Units: Bit field

Acc84E[*i*].PartType contains an integer representing the type(s) of hardware interface of the ACC-84E serial-encoder interface board. It is derived from an identification IC on the board, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no board with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present on the board. The interface types that could be present in general are:

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

The value of **PartType** for the ACC-84E serial-encoder interface board is 9 (servo and digital I/O interface).

This element is not actually a hardware register on the board; it is a software register associated with the board. It is used mainly by system setup utilities in the IDE.

Acc84E[*i*]. Single-Channel Status Elements

Some aspects of the serial-encoder interface can be read individually for each channel.

Acc84E[*i*].Chan[*j*].SerialEncDataA

Description: ACC-84E encoder position least significant word

Range: $0 .. $FFFFFF (0 .. 16,777,215)

Units: Encoder LSBs

Acc84E[*i*].Chan[*j*].SerialEncDataA contains the least significant word (the lowest position bits) from the most recent value read from the serial encoder with channel index ***j*** on the ACC-84E with board index ***i***. The channel index ***j*** can take a value from 0 to 3, corresponding to encoders 1 to 4, respectively. Note that the channel index ***j*** is one less than the corresponding encoder number.

The contents of this element are protocol-specific, but virtually always, the least significant bit of this element contains the value of the least significant bit from the encoder reading. When the encoder is used for position feedback or master position, the encoder conversion table will read this register every servo cycle to process the position data for use by a motor's servo algorithm.

More significant position data and status information can be found in **Acc84E[*i*].Chan[*j*].SerialEncDataB** in most protocols.

When used in the Script environment, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is a 24-bit element. When used in the C environment, it is a 32-bit element, with real data in the high 24 bits, so its value in the C environment is 256 times greater than its value in the Script environment.

Note that when the *register* for **Acc84E[*i*].Chan[*j*].SerialEncDataA** is used for automatic servo or phase functions, as when **EncTable[*n*].pEnc**, **Motor[*x*].pAbsPos**, **Motor[*x*].pPhaseEnc**, or **Motor[*x*].pAbsPhasePos** is set to **Acc84E[*i*].Chan[*j*].SerialEncDataA.a**, the entire 32-bit register is used, with real data typically starting in bit 8. Shifting or masking operations are then used to isolate the real position data to be used.

Acc84E[*i*].Chan[*j*].SerialEncDataA will report as "nan" (not-a-number) if no board with this index is present.

BiSS-B/C Protocol

For a BiSS-B/C encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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

Component: *Single/Multi-Turn Position*

Bits P*n* represent the bits of single-turn and multi-turn position.

EnDat 2.1/2.2 Protocol

For an EnDat 2.1/2.2 encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	-	-
	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*Single/Multi-Turn Position*																									

Bits P*n* represent the bits of single-turn and multi-turn position.

Matsushita Protocol

For a Matsushita encoder with 17 bits per revolution, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows (if single turn data has more resolution, higher bits include data):

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	M	M	M	M	M	M	M	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	-	-
	6	5	4	3	2	1	0	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*Multi-Turn Position*							*Single-Turn Position*																		

Bits S*n* represent the bits of single-turn position. Bits M*n* represent the bits of multi-turn position.

Mitsubishi Protocol

For a Mitsubishi HG-☐ encoder with 18 bits per revolution, **Acc84E[*i*].Chan[*j*].SerialEncDataA** can be configured in three different modes, depending on the setting of *SerialEncCmdWord* in **Acc84E[*i*].Chan[*j*].SerialEncCmd**:

Single-Turn Data in Lower 18 Bits of 24-Bit Word

$02 –single-turn data
$BA – for clearing alarms and reporting single-turn data
$2A – for reporting single-turn and multi-turn data

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	-	-	-	-	-	-	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	-	-
							17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*Single-Turn Position*																									

Single-Turn Data in Lower 20 Bits of 24-Bit Word

$A2 – for reporting single-turn and multi-turn data

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	-	-	-	-	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	-	-	-	-
					17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
Component:	*Single-Turn Position*																									

Single-Turn Data in Upper 20 Bits of 24-Bit Word

$32 – for reporting single-turn and multi-turn data

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	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	-	-	-	-	-	-	-	-
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
Component:	*Single-Turn Position*																									

This method of reporting provides the best contiguous data between single-turn and multi-turn data which can be very useful in both Turbo and Power PMAC Power-on Servo Position retrieval process using built-in functionality.

Mitutoyo Protocol

For a Mitutoyo encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	-	-
	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*Single/Multi-Turn Position*																									

Bits P*n* represent the bits of single-turn and multi-turn position.

Panasonic Protocol

For a Panasonic encoder with 17 bits per revolution, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	-	-	-	-	-	-	-	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	-	-
								16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*Single-Turn Position*																									

Bits S*n* represent the bits of single-turn position.

SSI Protocol

For an SSI encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	-	-
	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*Single/Multi-Turn Position*																									

Bits P*n* represent the bits of single-turn and multi-turn position.

Tamagawa FA-Coder Protocol

For a Tamagawa FA-Coder encoder with 17 bits per revolution, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	-	-	-	-	-	-	-	S 16	S 15	S 14	S 13	S 12	S 11	S 10	S 9	S 8	S 7	S 6	S 5	S 4	S 3	S 2	S 1	S 0	-	-
Component:	*Single-Turn Position*																									

Bits S*n* represent the bits of single-turn position.

Yaskawa Sigma II/III/V Protocol

For the Yaskawa Sigma II/III/V protocol, the data format in this element depends on the particular type of the encoder and its reporting mode.

For an absolute Yaskawa Sigma II encoder with 17 bits per revolution in position-reporting (P1) mode, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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	M 2	M 1	M 0	S 16	S 15	S 14	S 13	S 12	S 11	S 10	S 9	S 8	S 7	S 6	S 5	S 4	S 3	S 2	S 1	S 0	-	-	-	-	-	-
Component:	*Multi-Turn Pos*			*Single-Turn Position*																						

Bits S*n* represent the bits of single-turn position; bits M*n* represent bits of the multi-turn position.

For an incremental Yaskawa Sigma II encoder with 17 bits per revolution in position-reporting (P1) mode, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

Hex Digit ($)																									-	-
Script 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
C 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	-	S 16	S 15	S 14	S 13	S 12	S 11	S 10	S 9	S 8	S 7	S 6	S 5	S 4	S 3	S 2	S 1	S 0	-	-	U	V	W	Z	-	-
Component:	*Single-Turn Position*																				*Hall & Index*					

Bits S*n* represent the bits of single-turn position; U, V, and W represent the commutation "Hall" sensor signal states, and Z represents the encoder's "zero" (index) pulse marker signal state.

For an absolute Yaskawa Sigma III or V encoder with 20 bits per revolution in position-reporting (P1) mode, **Acc84E[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	S 19	S 18	S 17	S 16	S 15	S 14	S 13	S 12	S 11	S 10	S 9	S 8	S 7	S 6	S 5	S 4	S 3	S 2	S 1	S 0	-	-	-	-	-	-
Component:	*Single-Turn Position*																									

Bits S*n* represent the bits of single-turn position.

Acc84E[*i*].Chan[*j*].SerialEncDataB

Description: ACC-84E encoder position second word

Range: $0 .. $FFFFFF (0 .. 16,777,215)

Units: (Protocol-specific)

Acc84E[*i*].Chan[*j*].SerialEncDataB contains the second word from the most recent value read from the serial encoder with channel index ***j*** on the ACC-84E with board index ***i***. The channel index ***j*** can take a value from 0 to 3, corresponding to encoders 1 to 4, respectively. Note that the channel index ***j*** is one less than the corresponding encoder number.

The contents of this element are protocol-specific, but virtually always will contain position data (if any) of higher significance than the data in **Acc84E[*i*].Chan[*j*].SerialEncDataA**. It may also contain status and error bits.

When the encoder is used for position feedback, any error bits in this register can be used by the Encoder Conversion Table entry processing the feedback if the type=12 method is used to reject that cycle's position value as possibly erroneous and to substitute a computed value extrapolated from previous cycles' data.

When used in the Script environment, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is a 24-bit element. When used in the C environment, it is a 32-bit element, with real data in the high 24 bits, so its value in the C environment is 256 times greater than its value in the Script environment.

Note that when the *register* for **Acc84E[*i*].Chan[*j*].SerialEncDataB** is used for automatic functions, as for absolute power-on position, or encoder loss detection, the entire 32-bit register is accessed, and specific bits must be specified considering it as a 32-bit value.

Acc84E[*i*].Chan[*j*].SerialEncDataB will report as "nan" (not-a-number) if no board with this index is present.

BiSS-B/C Protocol

For a BiSS-B/C encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E	E	S	S	S	S	S	S	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	-	-
	1	0	5	4	3	2	1	0	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24		
Component:	*TE*	*CE*											*Single/Multi-Turn Position*													

Bits P*n* represent the bits of single-turn and multi-turn position. Bits S*n* represent encoder-specific status bits. Bits E*n* represent the error bits (E0 is a CRC error detected by the IC, and E2 is a timeout error detected by the IC).

EnDat2.1/2.2 Protocol

For an EnDat 2.1/2.2 encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E 5	E 4	E 3	E 2	E 1	E 0	-	-	P 39	P 38	P 37	P 36	P 35	P 34	P 33	P 32	P 31	P 30	P 29	P 28	P 27	P 26	P 25	P 24	-	-
Component:	*TE*	*CE*	*CE1*	*CE2*	*EB1*	*EB2*											*Single/Multi-Turn Position*									

Bits P*n* represent the bits of single-turn and multi-turn position. Bits E*n* represent the error bits. (E0 is "error bit 1" reported by the encoder [2.2 only], E1 is "error bit 2" reported by the encoder, E2 is a CRC error detected by the IC for the 2nd additional information word [2.2 only], E3 is a CRC error detected by the IC for the 1st additional information word [2.2 only], E4 is a CRC error detected by the IC for the position information word, and E5 is a timeout error detected by the IC).

Matsushita Protocol

For a Matsushita encoder with 16 bits of multi-turn count, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	M 14	M 13	M 12	M 11	M 10	M 9	M 8	M 7	-	-
Component:																	*Multi-Turn Position*									

Bits M*n* represent the bits of multi-turn position.

Mitsubishi Protocol

For a Mitsubishi HG-☐ encoder with 16 bits of multi-turn count, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	-	-	-	-	-	-	-	-	M 15	M 14	M 13	M 12	M 11	M 10	M 9	M 8	M 7	M 6	M 5	M 4	M 3	M 2	M 1	M 0	-	-
Component:									*Multi-Turn Position*																	

Bits M*n* represent the bits of multi-turn position.

Mitutoyo Protocol

For a Mitutoyo encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	P 31	P 30	P 29	P 28	P 27	P 26	P 25	P 24	-	-
Component:																	*Single/Multi-Turn Position*									

Bits P*n* represent the bits of single-turn and multi-turn position.

Panasonic Protocol

For a Panasonic encoder with 16 bits of multi-turn count, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	-	-	-	-	-	-	-	-	M 15	M 14	M 13	M 12	M 11	M 10	M 9	M 8	M 7	M 6	M 5	M 4	M 3	M 2	M 1	M 0	-	-
Component:	*Multi-Turn Position*																									

Bits M*n* represent the bits of multi-turn position. Multi-turn position is only reported if the *SerialEncCmdWord* component of **Acc84E[*i*].Chan[*j*].SerialEncCmd** is set to $2A, in which case the encoder ID and alarm code are not reported.

SSI Protocol

For an SSI encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E 0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	P 31	P 30	P 29	P 28	P 27	P 26	P 25	P 24	-	-
Component:	*PE*																*Single/Multi-Turn Position*									

Bits P*n* represent the bits of single-turn and multi-turn position. Bit E0 represents the parity error bit.

Tamagawa FA-Coder Protocol

For a Tamagawa FA-Coder encoder with 16 bits of multi-turn count, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	-	-	-	-	-	-	-	-	M 15	M 14	M 13	M 12	M 11	M 10	M 9	M 8	M 7	M 6	M 5	M 4	M 3	M 2	M 1	M 0	-	-
Component:	*Multi-Turn Position*																									

Bits M*n* represent the bits of multi-turn position.

Yaskawa Sigma II/III/V Protocol

For the Yaskawa Sigma II/III/V protocol, the data format in this element depends on the particular type of the encoder and its reporting mode.

For an absolute Yaskawa Sigma II encoder with 17 bits per revolution and 16 bits of turns count in position-reporting (P1) mode, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E 2	E 1	E 0	-	-	-	-	-	-	-	-	M 15	M 14	M 13	M 12	M 11	M 10	M 9	M 8	M 7	M 6	M 5	M 4	M 3	-	-
Component:	*TE*	*CE*	*EB*	*Multi-Turn Position*																						

Bits M*n* represent the bits of multi-turn position. Bits E*n* represent the error bits (E0 is a coding error reported by the encoder, E1 is a CRC error detected by the IC, and E2 is a timeout error detected by the IC).

For an incremental Yaskawa Sigma II encoder with 17 bits per revolution in position-reporting (P1) mode, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E	E	E	-	-	-	-	-	-	-	-	-	-	C	C	C	C	C	C	C	C	C	C	C	-	-
	2	1	0											16	15	14	13	12	11	10	9	8	7	6		
Component:	*TE*	*CE*	*EB*											*Compensation Position*												

Bits C*n* represent the bits of "compensation" position, captured on the first index pulse. Bits E*n* represent the error bits (E0 is a coding error reported by the encoder, E1 is a CRC error detected by the IC, and E2 is a timeout error detected by the IC).

For an absolute Yaskawa Sigma III or V encoder with 20 bits per revolution in position-reporting (P1) mode, **Acc84E[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E	E	E	-	-	-	-	-	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	-	-
	2	1	0						15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*TE*	*CE*	*EB*											*Multi-Turn Position*												

Bits M*n* represent the bits of multi-turn position. Bits E*n* represent the error bits (E0 is a coding error reported by the encoder, E1 is a CRC error detected by the IC, and E2 is a timeout error detected by the IC).

Bits P*n* represent the bits of single-turn and multi-turn position. Bits E*n* represent the error bits (E0 is a CRC error detected by the IC, and E1 is a timeout error detected by the IC).

Acc84E[*i*].Chan[*j*].SerialEncDataC

Description: ACC-84E encoder position third word

Range: $0 .. $FFFFFF (0 .. 16,777,215)

Units: (Protocol-specific)

Acc84E[*i*].Chan[*j*].SerialEncDataC contains the third word from the most recent value read from the serial encoder with channel index ***j*** on the ACC-84E with board index ***i***. The channel index ***j*** can take a value from 0 to 3, corresponding to encoders 1 to 4, respectively. Note that the channel index ***j*** is one less than the corresponding encoder number.

The contents of this element are protocol specific, but may contain encoder ID data, alarm codes, plus status and error bits. Not all of the serial encoder protocols provide data for this element.

When the encoder is used for position feedback, any error bits in this register can be used by the Encoder Conversion Table entry processing the feedback if the type=12 method is used to reject that cycle's position value as possibly erroneous and to substitute a computed value extrapolated from previous cycles' data.

When used in the Script environment, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is a 24-bit element. When used in the C environment, it is a 32-bit element, with real data in the high 24 bits, so its value in the C environment is 256 times greater than its value in the Script environment.

Note that when the *register* for **Acc84E[*i*].Chan[*j*].SerialEncDataC** is used for automatic functions, as for encoder loss detection, the entire 32-bit register is accessed, and specific bits must be specified considering it as a 32-bit value.

Acc84E[*i*].Chan[*j*].SerialEncDataC will report as "nan" (not-a-number) if no board with this index is present.

EnDat2.1/2.2 Protocol

For the EnDat2.2 protocol, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is used for the first additional information word if this is requested of the encoder with an MRS code in **Acc84E[*i*].Chan[*j*].SerialEncCmd**. This register is never used with the EnDat2.1 protocol.

For an EnDat 2.2 encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	W	R	B	A	A	A	A	A	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	-	-
	0	0	0	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*WN*	*RM*	*BY*	*MRS Acknowledge*									*Additional Information 1 Word*													

Bits I*n* represent bits of the requested first additional information word. Bits A*n* represent bits of the acknowledgement of the MRS code in **Acc84E[*i*].Chan[*j*].SerialEncCmd**. The value of the acknowledgement code is $40 (64) less than the value of the commanded MRS code. For example, if the MRS code is $42 (66), the acknowledgement code is $02 (2).

Bit B0 is the "busy" status bit. Bit R0 is the "RM" (reference mark) status bit. Bit W0 is the "warning" status bit.

The following table provides details of the information and acknowledgement words for each of the MRS command codes for the 1st additional information word.

Serial Encoder Data Register C: Additional Information 1				
Command code	**Acknowledgement of MRS Code Bit[20:16], decimal**	**Information/Type**	**Byte 1 [I15:I8]**	**Byte 2 [I7:I0]**
$42	$02 (2)	Position Value 2 Word 1 LSB	MSB data	LSB data
$43	$03 (3)	Position Value 2 Word 2	MSB data	LSB data
$44	$04 (4)	Position Value 2 Word 3 MSB	MSB data	LSB data
$47	$07 (7)	MRS Code	MRS Code	Any
$49	$09 (9)	Test values word 1 LSB	MSB data	LSB data
$4A	$0A (10)	Test values word 2	MSB data	LSB data
$4B	$0B (11)	Test values word 3 MSB	MSB data	LSB data
$4C	$0C (12)	Temperature sensor 1	MSB data	LSB data
$4D	$0D (13)	Temperature sensor 2	MSB data	LSB data
$4F	$0F (15)	Stop additional information 1	Any	Any

Matsushita Protocol

For the Matsushita protocol, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is configured as follows:

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	E 1	E 0	-	-	-	-	-	-	A 7	A 6	A 5	A 4	A 3	A 2	A 1	A 0	S 3	S 2	S 1	S 0	I 3	I 2	I 1	I 0	-	-
Component:	*TE CE*								*Alarm Code*								*Status Code*				*Encoder ID*					

Bits I*n* represent bits of the returned encoder ID code (fixed at $11). Bits A*n* represent bits of the alarm code. Bits S*n* represent bits of the status code. Bits E*n* represent bits of the error code (E0 is a CRC error detected by the IC, bit E1 is a timeout error detected by the IC).

The A*n* alarm code bits have the following meanings:

- A0: Overspeed error
- A1: Preset status error
- A2: Count error 1
- A3: Count error 2
- A4: (*reserved*)
- A5: Overflow error
- A6: System down (undervoltage) error
- A7: Battery alarm error

The S*n* status code bits have the following meanings:

- S0: ea0 – Logical OR of overspeed, system down, battery alarm
- S1: ea1 – Logical OR of count error 1, count error 2
- S2: (*reserved*)
- S3: Preset status

The encoder ID and alarm code values are only reported if the *SerialEncCmdWord* component of **Acc84E[*i*].Chan[*j*].SerialEncCmd** is set to $52, in which case multi-turn data is not reported.

Mitsubishi Protocol

For a Mitsubishi HG-☐ encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is configured as follows:

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	E	E	-	-	S	S	S	S	A	A	A	A	A	A	A	A	I	I	I	I	I	I	I	I	-	-
	1	0			7	6	5	4	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Component:	*TE CE*				*Status Code*				*Alarm Code*								*Encoder ID*									

Bits I*n* represent bits of the returned encoder ID code. Bits A*n* represent bits of the alarm code. Bits S*n* represent bits of the status code. Bits E*n* represent bits of the error code (E0 is a CRC error detected by the IC, bit E1 is a timeout error detected by the IC).

The A*n* alarm code bits have the following meanings:

- A0: CPU alarm
- A1: (*reserved*)
- A2: Data alarm (data per revolution error)
- A3: Thermal shutdown alarm
- A4: Thermal warning
- A5: Multi-revolution count alarm
- A6: Multi-revolution backup alarm
- A7: Battery disconnected warning

The S*n* status code bits have the following meanings:

- S4: ca1 – Delimiter error in request frame
- S5: ca0 – Parity error in request frame
- S6: ea1 – logical OR of CPU, data, thermal, mult-rev count, multi-rev backup alarms
- S7: ea0 – Logical OR of thermal warning, battery disconnected warning

Mitutoyo Protocol

For a Mitutoyo encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E	E	-	-	S	S	S	S	A	A	A	A	A	A	A	A	I	I	I	I	I	I	I	I	-	-
	1	0			3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Component:	*TE CE*				*Status Code*				*Alarm Code*								*Encoder ID*									

Bits I*n* represent bits of the returned encoder ID code; bits A*n* represent bits of the alarm code; bits S*n* represent bits of the status code; bits E*n* represent bits of the error code (E0 is a CRC error detected by the IC, and E1 is a timeout error detected by the IC).

The A*n* alarm code bits have the following meanings:

- A0: Initialization error
- A1: Mismatch of optical and capacitive sensors
- A2: Optical sensor error
- A3: Capacitive sensor error
- A4: CPU error (AT303); CPU/ROM/RAM error (AT503)

- A5: EEPROM error
- A6: ROM/RAM error (AT303); Communication error (AT503)
- A7: Overspeed error

The S*n* status field bits have the following meanings:

- S0: Fatal (unrecoverable) encoder error
- S1: (*reserved*)
- S2: Illegal command code from controller
- S3: (*reserved*)

Panasonic Protocol

For a Panasonic encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E 1	E 0	-	-	S 7	S 6	S 5	S 4	A 7	A 6	A 5	A 4	A 3	A 2	A 1	A 0	I 7	I 6	I 5	I 4	I 3	I 2	I 1	I 0	-	-
Component:	*TE CE*				*Status Code*				*Alarm Code*								*Encoder ID*									

Bits I*n* represent bits of the returned encoder ID code; bits A*n* represent bits of the alarm code; bits S*n* represent bits of the status code; bits E*n* represent bits of the error code (E0 is a CRC error detected by the IC, and E1 is a timeout error detected by the IC).

The A*n* alarm code bits have the following meanings:

- A0: Overspeed error
- A1: Full resolution status; = 1 when over 100rpm and reporting reduced resolution
- A2: Count error
- A3: Counter overflow
- A4: (reserved)
- A5: Multi-revolution error
- A6: System undervoltage error (< 2.5V)
- A7: Battery low (< 3.1V)

The encoder ID and alarm code values are only reported if the *SerialEncCmdWord* component of **Acc84E[*i*].Chan[*j*].SerialEncCmd** is set to $52, in which case multi-turn data is not reported.

Tamagawa FA-Coder Protocol

For a Tamagawa FA-Coder encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	E 1	E 0	-	-	S 7	S 6	S 5	S 4	A 7	A 6	A 5	A 4	A 3	A 2	A 1	A 0	I 7	I 6	I 5	I 4	I 3	I 2	I 1	I 0	-	-
Component:	*TE CE*				*Status Code*				*Alarm Code*								*Encoder ID*									

Bits I*n* represent bits of the returned encoder ID code; bits A*n* represent bits of the alarm code; bits S*n* represent bits of the status code; bits E*n* represent bits of the error code (E0 is a CRC error detected by the IC, and E1 is a timeout error detected by the IC).

Yaskawa Sigma II/III/V Protocol

For a Yaskawa Sigma II/III/V encoder in position-reporting (P1) mode, **Acc84E[*i*].Chan[*j*].SerialEncDataC** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	-	-	-	-	-	-	-	-	A7	A6	A5	A4	A3	A2	A1	A0	T7	T6	T5	T4	T3	T2	T1	T0	-	-
Component:									*Alarm Code*								*Temperature*									

Bits T*n* represent bits of the returned temperature value (in degrees C); bits A*n* represent bits of the alarm code.

For an absolute encoder, the alarm-code bits have the following meanings:

- A0: Battery-backed turns data lost
- A1: Power-on error self-detected
- A2: Battery low-voltage warning
- A3: Absolute position error
- A4: Over-speed error
- A5: Over-temperature error
- A6: Encoder reset in progress
- A7: (reserved)

For an incremental encoder, the alarm-code bits have the following meanings:

- A0: (reserved)
- A1: Power-on error self-detected
- A2: (reserved)
- A3: Revolution count (index to index) incorrect
- A4: (reserved)
- A5: (reserved)
- A6: Position reference (index) not found yet
- A7: (reserved)

Acc84E[*i*].Chan[*j*].SerialEncDataD

Description: ACC-84E encoder position 4th word

Range: $0 .. $FFFFFF (0 .. 16,777,215)

Units: (Protocol-specific)

Acc84E[*i*].Chan[*j*].SerialEncDataD contains the fourth word (if any) from the most recent value read from the serial encoder with channel index ***j*** on the ACC-84E with board index ***i***. The channel index ***j*** can take a value from 0 to 3, corresponding to encoders 1 to 4, respectively. Note that the channel index ***j*** is one less than the corresponding encoder number.

The contents of this element are protocol-specific. Presently, only one of the protocols supported provides any information for this element.

EnDat2.1/2.2 Protocol

For the EnDat2.2 protocol, **Acc84E[*i*].Chan[*j*].SerialEncDataD** is used for the second additional information word if this is requested of the encoder with an MRS code in **Acc84E[*i*].Chan[*j*].SerialEncCmd**. This register is never used with the EnDat2.1 protocol.

For an EnDat 2.2 encoder, **Acc84E[*i*].Chan[*j*].SerialEncDataD** is configured as follows:

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	7-4	3-0
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	-	-
Bit Value	W	R	B	A	A	A	A	A	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	-	-
	0	0	0	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Component:	*WN*	*RM*	*BY*	*MRS Acknowledge*					*Additional Information 2 Word*																	

Bits I*n* represent bits of the requested first additional information word. Bits A*n* represent bits of the acknowledgement of the MRS code in **Acc84E[*i*].Chan[*j*].SerialEncCmd**. The value of the acknowledgement code is $40 (64) less than the value of the commanded MRS code. For example, if the MRS code is $53 (83), the acknowledgement code is $13 (19).

Bit B0 is the "busy" status bit. Bit R0 is the "RM" (reference mark) status bit. Bit W0 is the "warning" status bit.

The following table provides details of the information and acknowledgement words for each of the MRS command codes for the 2nd additional information word.

Serial Encoder Data Register D: Additional Information 2									
Command code	**Acknowledgement of MRS Code Bit[20:16], decimal**	**Information/Type**	**Byte 1 [15:8]**						**Byte 2 [7:0]**
$51	$11 (17)	Commutation	U	V	W	Not assigned			Not assigned
			15	14	13	[12:8]			
$52	$12 (18)	Acceleration	MSB data						LSB data
$53	$13 (19)	Commutation & Acceleration	U	U	W	MSB Acceleration data			LSB acceleration data
			15	14	13	[12:8]			
$54	$14 (20)	Limit position signals	L1	L2	Not assigned				Not assigned
			15	14	[13:8]				
$55	21	Limit position signals & Acceleration	L1	L2		MSB Acceleration data			LSB acceleration data
			15	14	13	[12:8]			
$5F	31	Stop additional information 2	Any						Any

When used in the Script environment, **Acc84E[*i*].Chan[*j*].SerialEncDataD** is a 24-bit element. When used in the C environment, it is a 32-bit element, with real data in the high 24 bits, so its value in the C environment is 256 times greater than its value in the Script environment.

Acc84E[*i*].Chan[*j*].SerialEncDataD will report as “nan” (not-a-number) if no board with this index is present.

Acc84S[*i*]. Status Data Structure Elements

The **Acc84S[*i*]** data structure for the serial-encoder board for the Power Clipper product line is equivalent to the **Acc84E[*i*]** data structure that is used for the rack-mounted Power UMAC serial-encoder interface board that is documented above. It is *not* an alias for the **Acc84E[*i*]** data structure name, as it can only be used for the Power Clipper products.

Note that the ACC-84S cannot be auto-identified by the Power PMAC CPU, so to use this data structure, the user must manually set **GateIo[*i*].PartNum** to 603936, **GateIo[*i*].PartType** to 9, issue a save command, and reset the controller, before being able to use this structure. This structure is new in V2.0 firmware, released 1st quarter 2015.

AdcDemux. Status Elements

The **AdcDemux** status elements contain the results of the process of the automatic de-multiplexing the multiplexed analog-to-digital converters on certain Power PMAC accessories. These elements can be read as if they were individual A/D converters.

AdcDemux.ResultHigh[*i*]

Description: ADC de-multiplexing high-word result value

Range: -2048 .. 2047, or 0 .. 4095

Units: 12-bit ADC units

AdcDemux.ResultHigh[*i*] elements contain the de-multiplexed result value from the "high" multiplexed 12-bit A/D converter on an ACC-36E board. The index value ***i*** for these elements can take a value from 0 to 15.

The corresponding saved setup element **AdcDemux.Address[*i*]** specifies the I/O address of the A/D converter board, and the corresponding saved setup element **AdcDemux.ConvertCode[*i*]** specifies which of the multiplexed analog inputs is used to create this value, and whether it is a signed or unsigned value.

AdcDemux.ResultLow[*i*]

Description: ADC de-multiplexing low-word result value

Range: -2048 .. 2047, or 0 .. 4095

Units: 12-bit ADC units

AdcDemux.ResultLow[*i*] elements contain the de-multiplexed result value from the "low" multiplexed 12-bit A/D converter on an ACC-36E or ACC-59E board. The index value ***i*** for these elements can take a value from 0 to 15.

The corresponding saved setup element **AdcDemux.Address[*i*]** specifies the I/O address of the A/D converter board, and the corresponding saved setup element **AdcDemux.ConvertCode[*i*]** specifies which of the multiplexed analog inputs is used to create this value, and whether it is a signed or unsigned value.

BrickAC. Status Data Structure Elements

The Power Brick AC is one class of the Power Brick family of intelligent amplifiers. It can support multiple types of brush, brushless and AC induction motors, operating from an AC-line input. Its registers can be accessed through **BrickAC.** data structure.

The Power Brick AC status data structure elements are updated only if **BrickAC.Monitor** control register is set to 1. The update period can be controlled by **BrickAC.MonitorPeriod** parameter.

Note

The amplifier will check for fault conditions regardless of whether the monitor process is enabled. It will take appropriate action to shut down either the entire power stage or the particular channel when a fault condition is detected, and set the "amplifier fault" line(s) back to the controller stage to the "true" state. Without the monitoring process active, the amplifier fault line is the only notification that the controller stage receives.

Note

The monitored data in the Power Brick AC amplifier is provided to the controller in the low bits of the **Gate3[*i*].Chan[*j*].AdcAmp[*k*]** registers, below the current feedback values. This data cannot be read if two phases are "packed" into one register, so it is essential that **Gate3[*i*].Chan[*j*].PackInData** and **Gate3[*i*].Chan[*j*].PackOutData** are set to 0, disabling packed data and allowing the full registers to be read by the CPU.

Note

The **BrickAC.** data structure elements documented in this section are software elements that are distinct from the **PowerBrick[*i*].** hardware data structure elements that form the control and status registers for the ASIC(s) in the Power Brick AC. (The **PowerBrick[*i*].** data structure is an "alias" for the **Gate3[*i*].** data structure.)

BrickAC. Multi-Channel Status Elements

Some of the status data structures for the Power Brick AC return values that are common for all channels.

BrickAC.BusOverVoltage

Description: DC bus overvoltage flag

Range: 0 .. 1

Units: Boolean

The **BrickAC.BusOverVoltage** status bit indicates whether the amplifier has detected an overvoltage condition on the DC bus or not. It is set to 0 when the measured DC bus voltage is 435 VDC or less. It is set to 1 when the bus voltage has exceeded 435 VDC. This is a latching fault in the amplifier power stage and it can only be reset if the bus power is cycled to the amplifier. Please refer to the hardware reference manual for proper power cycling procedure.

This status bit is only updated if **BrickAC.Monitor** is set to 1.

If this fault is detected, the amplifier-fault lines for all channels are set to the "true" state, causing a software fault condition on all Power PMAC motors commanding these channels. After the fault is cleared, the motors will require a command to be re-enabled.

Note

The amplifier will shut down with a fault on all channels when it detects an overvoltage condition regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.BusUnderVoltage

Description: DC bus undervoltage flag

Range: 0 .. 1

Units: Boolean

The **BrickAC.BusUnderVoltage** status bit indicates whether the bus voltage is above a minimum threshold or not. It is set to 1 when the amplifier detects an undervoltage condition on the DC bus, which occurs when the bus voltage goes below 100 VDC, corresponding to a supply voltage of about 70 VAC(rms). This status bit is only updated if **BrickAC.Monitor** is set to 1.

If **BrickAC.UnderVoltageWarnOnly** is set to its default value of 0, this is a fault condition, and the amplifier-fault lines for all channels are set to the "true" state, causing a software fault condition on all Power PMAC motors commanding these channels.

If **BrickAC.UnderVoltageWarnOnly** is set to 1, this is only a warning status bit

BrickAC.BusUnderVoltage is a transparent status bit and it will be cleared to 0 as soon as the measured voltage exceeds 110 VDC again. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

Note

The amplifier will shut down with a fault on all channels when it detects an undervoltage condition if **BrickAC.UnderVoltageWarnOnly** is set to 0 regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.BusVoltage

Description: DC bus voltage value

Range: 0 .. 1023

Units: Volts DC

The **BrickAC.BusVoltage** status element contains the DC Bus voltage value. The value is only updated if **BrickAC.Monitor** is set to 1.

BrickAC.LineOk

Description: Power line input presence (for internal use)

Range: 0 .. 1

Units: Boolean

The **BrickAC.LineOk** status bit indicates the state of bus-power line inputs. It is set to 1 when line input power is detected. It is set to 0 within two AC line cycles after detection of complete loss of input phase power. This status bit is only updated if **BrickAC.Monitor** is set to a value greater than 0.

BrickAC.LineOk is primarily for internal use. If it is set to 0, the status bit **BrickAC.PowerFault** will be set to 1, creating a global fault condition in the amplifier.

BrickAC.PhaseInMissing

Description: Missing line input phase(s) when in 3-phase input mode

Range: 0 .. 1

Units: Boolean

The **BrickAC.PhaseInMissing** status bit indicates whether all three phases of the power line input are present or not, if the amplifier is set up to expect a 3-phase input. It is set to 1 if one or more of the three line input phases is missing when the **BrickAC.SinglePhaseIn** parameter is set to 0. It is set to 0 if all three phases are present or if **BrickAC.SinglePhaseIn** is set to 1. This status bit is only updated if **BrickAC.Monitor** is set to a value greater than 0.

BrickAC.PowerBoardId

Description: Power board ID code (for internal use)

Range: 0 .. 15

Units: none

The **BrickAC.PowerBoardId** status element contains the power board ID code, which indicates the power ratings for each channel. This parameter is for Delta Tau internal use. The value is only updated if **BrickAC.Monitor** is set to a value greater than 0.

BrickAC.PowerFault

Description: Bus power supply fault status bit

Range: 0 .. 1

Units: none

The **BrickAC.PowerFault** status element indicates whether the power supplied to the DC bus has a problem or not. All of the following criteria should be met before this flag is set to 0.

1. Input power verified (**BrickAC.LineOK** = 1).
2. Soft-start process is completed.
3. No soft-start IGBT fault has occurred.

If any of these conditions is not met, **BrickAC.PowerFault** is set to 1 and the amplifier becomes non-operational, setting the amplifier-fault signals on all channels to the "true" state. This fault flag is non-latched and it will automatically clear to 0 once all the above conditions are met. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

Note

In standard operation, **BrickAC.PowerFault** will be set to 1 from the time the amplifier logic initializes after the 24VDC power is applied until several seconds after the bus power is applied and the soft-start process has completed. If the Power PMAC attempts to enable any motors that use the amplifier stage during this period, the enabling will fail due to this fault state.

Note

The amplifier will shut down with a fault on all channels when it detects a power fault regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.RegenFault

Description: Regeneration shunt circuitry fault status bit (internal use only)

Range: 0 .. 1

Units: Boolean

The **BrickAC.RegenFault** status bit whether the regeneration shunt circuitry is in a fault condition or not. It is set to 1 if either of the following fault conditions is found:

1. The regen-shunt IGBT is experiencing an under-voltage condition (voltage is less than 12VDC)
2. Regen-shunt desaturation fault is detected. This fault is generated when the shunt resistor is pulling too much current or is shorted.

When no fault is detected, **BrickAC.RegenFault** is set to 0. This status bit is only updated if **BrickAC.Monitor** is set to 1.

The regen fault is a latching fault. Once it is detected, the fault status is latched. This fault can be cleared by setting **BrickAC.Reset** to 1.

Note

The amplifier will shut down with a fault on all channels when it detects a regeneration fault regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.RegenOverLoad

Description: Regeneration shunt circuitry overload status bit

Range: 0 .. 1

Units: Boolean

The**BrickAC.RegenOverLoad** status bit indicates whether the shunt resistor has recently been on continually for more than 2 seconds. It is set to 1 when the resistor has been on for the past two seconds. In this eventuality, it is cleared to 0 automatically after a “cool-down” period. It is set to 0 if it has not recently been on continually for 2 seconds.

This is a warning flag; no fault is generated when it is set to 1. This status bit is only updated if **BrickAC.Monitor** is set to 1.

BrickAC.SoftStartFault

Description: Soft-start circuitry fault status bit (for internal use)

Range: 0 .. 1

Units: Boolean

The **BrickAC.SoftStartFault** status bit indicates whether a fault has been detected in the soft-start circuitry or not. It is set to 1 if either of the following conditions is detected:

1. The soft-start IGBT is experiencing an under-voltage condition (voltage is less than 12VDC)
2. Soft-start desaturation fault is detected. This fault is generated when the Bus capacitors are pulling too much current.

When no fault is detected, **BrickAC.SoftStartFault** is set to 0. This status bit is only updated if **BrickAC.Monitor** is set to 1.

BrickAC.SoftStartFault is primarily for internal use. If it is set to 1, the status bit **BrickAC.PowerFault** will be set to 1, creating a global fault condition in the amplifier. This fault flag is non-latched and it will automatically clear to 0 once all the above conditions are no longer present. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

Note

The amplifier will shut down with a fault on all channels when it detects a soft-start fault regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.STO0

Description: "Safe torque off" STO0 input state

Range: 0 .. 1

Units: Boolean

The **BrickAC.STO0** status bit indicates the status of STO0 "safe torque off" input. It reports as 0 if a 24VDC level is supplied to the STO0 input, or if the adjacent "Disable STO" pin is connected to the "Disable STO Return" pin. It reports as 1 if there is no 24VDC level applied to this input and if "Disable STO" is not connected to "Disable STO Return".

If the 24VDC level is removed from the STO0 input, there is no power supplied to the gate driver circuits that turn on the power transistors, so no electrical power can be supplied to the motors and no torque can be generated. This is known as "safe torque off" mode. This status bit is only updated if **BrickAC.Monitor** is set to 1.

The safe-torque off condition is non-latched and it will automatically clear to 0 once a 24VDC input is supplied to the STO0 input again. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.


Note

The amplifier will shut down with a fault on all channels when it detects a safe-torque-off condition regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.STO1

Description: STO1 disable condition control input state

Range: 0 .. 1

Units: Boolean

The **BrickAC.STO1** status bit indicates the state of the STO1 "disable condition control" input. It reports as 0 if a 24VDC level is supplied to the STO1 input, or if the adjacent "Disable STO" pin is connected to the "Disable STO Return" pin. It reports as 1 if there is no 24VDC level applied to this input and if "Disable STO" is not connected to "Disable STO Return". This status bit is neither a fault nor a warning bit, and it is only updated if **BrickAC.Monitor** is set to 1.

If the STO inputs are configured so that **BrickAC.STO1** would report a 1, when this channel of the amplifier is disabled (except by the safe-torque-off input), the motor leads are unconnected to each other and floating. If the STO inputs are configured so that **BrickAC.STO1** would report a 1, when this channel of the amplifier is disabled for whatever reason, the motor leads are shorted together through the low side of the DC bus, and dynamic braking is possible.


Note

The functionality control of the STO1 input is observed regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.UnderVoltageMasked

Description: Bus undervoltage condition display disabled

Range: 0 .. 1

Units: Boolean

The **BrickAC.UnderVoltageMasked** status bit indicates whether the bus undervoltage fault/warning is masked from display on the amplifier or not. It is set to 0 if the amplifier is

configured to display an undervoltage condition, and to 1 if it is configured not to display an undervoltage condition.

The selection as to whether this condition is displayed or not is controlled by the value of saved setup element **BrickAC.UnderVoltageWarnOnly**, as loaded into the active amplifier control circuitry with **BrickAC.Config** or **BrickAC.Reset**. This status bit is only updated if **BrickAC.Monitor** is set to 1.

Note

The choice as to whether the amplifier will shut down with a fault when it detects an undervoltage condition is determined by the value of **BrickAC.UnderVoltageWarnOnly**. It is independent of the choice as to whether to display an undervoltage condition or not.

BrickAC. Single-Channel Status Elements

Some status flags/registers of the BrickAC are defined individually for each channel. The channel index ***j*** has a range from 0 to 7, corresponding to hardware channel 1 to 8 on the board. The channel index is one less than the corresponding hardware channel number.

BrickAC.Chan[*j*].I2tExcess

Description: Channel I^2T fault/warning flag

Range: 0 .. 1

Units: Boolean

The **BrickAC.Chan[*j*].I2tExcess** status bit indicates whether an excessive integrated current (I^2T) condition is present on the channel or not. It is set to 0 if the integrated current value is not excessive; it is set to 1 if it is excessive. This status flag is only updated if **BrickAC.Monitor** is set to 1.

An excessive I^2T condition is calculated to exist if a current loading on the channel over the continuous rating would produce a greater dissipation than operating at the maximum intermittent rating for over two seconds does.

An excessive I^2T condition will generate a fault if saved setup element **BrickAC.Chan[*j*].I2tWarnOnly** is set to its default value of 0. It will not generate a fault if **BrickAC.Chan[*j*].I2tWarnOnly** is set to 1.

BrickAC.Chan[*j*].I2tExcess is a transparent status bit and it will be cleared to 0 as soon as the integrated current value falls below the threshold again. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The channel will shut down with a fault when it detects an I^2T excess condition if **BrickAC.Chan[*j*].I2tWarnOnly** is set to 0 regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

Note

The integrated current (I^2T) calculations accessed by this element are performed in the amplifier stage of the Power Brick AC. These calculations are separate from those done by the Power PMAC software.

BrickAC.Chan[*j*].IgbtOverTempFault

Description: Channel power device overtemperature fault flag

Range: 0 .. 1

Units: Boolean

The **BrickAC.Chan[*j*].IgbtOverTempFault** status bit indicates whether the calculated junction temperature of the channel's IGBT power device has exceeded its safe threshold or not. It is set to 0 if the calculated junction temperature is 120°C or less. It is set to 1 if this temperature is over 120°C. This status bit is only updated if **BrickAC.Monitor** is set to a value greater than 0.

BrickAC.Chan[*j*].IgbtOverTempFault is a transparent status bit and it will be cleared to 0 as soon as the calculated junction temperature value falls below the threshold again. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

The calculated junction temperature is derived from the measured case temperature, the measured current levels, and the PWM switching frequency for the channel.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The channel will shut down with a fault when it detects an IGBT over-temperature condition regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.Chan[*j*].IgbtTemp

Description: Channel IGBT case temperature

Range: 0 .. 255

Units: Degrees Celsius

The **BrickAC.Chan[*j*].IgbtTemp** status element contains the measured temperature IGBT case temperature for channel ***j*** of Power Brick AC. This value is only updated if **BrickAC.Monitor** is set to 1.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

BrickAC.Chan[*j*].InvalidPwmFreq

Description: Channel invalid PWM frequency flag

Range: 0 .. 1

Units: Boolean

The **BrickAC.Chan[*j*].InvalidPwmFreq** status bit indicates whether the PWM frequency supplied to this channel is valid or not. It is 0 if the frequency is within the valid range for the channel power device (4 kHz to 20 kHz). It is 1 if it is outside the valid frequency range for the device. This status flag is only updated if **BrickAC.Monitor** is set to a value greater than 0.

BrickAC.Chan[*j*].InvalidPwmFreq is a transparent status bit and it will be cleared to 0 as soon as the PWM frequency comes within the valid range again. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

The present measured PWM frequency for the channel can be found in status element **BrickAC.Chan[*j*].PwmFreq**.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The channel will shut down with a fault when it detects an invalid PWM frequency regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.Chan[*j*].OverCurrent

Description: Channel overcurrent fault flag

Range: 0 .. 1

Units: Boolean

The **BrickAC.Chan[*j*].OverCurrent** status bit indicates whether the hardware over-current detector for the channel has sensed an instantaneous overcurrent or short-circuit state for the channel or not. It is set to 0 if it has not detected this state. It is set to 1 if it has detected this state. This status flag is only updated if **BrickAC.Monitor** is set to a value greater than 0.

Over-current fault detection in Power Brick AC is performed in hardware. Once over-current fault is detected, the fault status is latched. This fault can be cleared by setting **BrickAC.Reset** equal to 1. Any motor software fault conditions it creates are also latched, and the motors must explicitly be re-enabled by command after this fault is cleared.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The channel will shut down with a fault when it detects an over-current condition regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BrickAC.Chan[*j*].OverTemp

Description: Channel excessive measured IGBT case temperature warning flag

Range: 0 .. 1

Units: Boolean

The **BrickAC.Chan[*j*].OverTemp** status bit indicates whether an excessive temperature is measured on the channel's IGBT case or not. It is 0 if the measured temperature is 75°C or less. It is 1 if the measured temperature is greater than 75°C. This status flag is only updated if **BrickAC.Monitor** is set to a value greater than 0.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

No fault is automatically generated if this status bit is set to 1; it should be considered a warning. Faults due to excessive temperature are based on the calculated "junction" temperature of the channel's power transistor block itself. The status bit for that fault is **BrickAC.Chan[*j*].IgbtOverTempFault**.

The present measured temperature for the channel's IGBT case can be found in status element **BrickAC.Chan[*j*].IgbtTemp**.

BrickAC.Chan[*j*].PwmFreq

Description: Channel measured PWM frequency

Range: 0 .. 1.024.000

Units: Hertz

The **BrickAC.Chan[*j*].PwmFreq** status element contains the measured PWM frequency for channel ***j*** of Power Brick AC. It should match the frequency commanded by the controller stage. This value has a granularity of 100 Hz and is only updated if **BrickAC.Monitor** is set to 1.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

BrickLV. Status Data Structure Elements

The Power Brick LV is one class of the Power Brick family of intelligent amplifiers. It can support multiple types of brush, brushless and stepper motors, operating from a low-voltage DC input. Its registers can be accessed through **BrickLV.** data structure.

The Power Brick LV status data structure elements are updated only if **BrickLV.Monitor** control register is set to 1. The update period can be controlled by **BrickAC.MonitorPeriod** parameter.

Note

The amplifier will check for fault conditions regardless of whether the monitor process is enabled. It will take appropriate action to shut down either the entire power stage or the particular channel when a fault condition is detected, and set the "amplifier fault" line(s) back to the controller stage to the "true" state. Without the monitoring process active, the amplifier fault line is the only notification that the controller stage receives.

Note

The monitored data in the Power Brick LV amplifier is provided to the controller in the low bits of the **Gate3[*i*].Chan[*j*].AdcAmp[*k*]** registers, below the current feedback values. This data cannot be read if two phases are "packed" into one register, so it is essential that **Gate3[*i*].Chan[*j*].PackInData** and **Gate3[*i*].Chan[*j*].PackOutData** are set to 0, disabling packed data and allowing the full registers to be read by the CPU.

Note

The **BrickLV.** data structure elements documented in this section are software elements that are distinct from the **PowerBrick[*i*].** hardware data structure elements that form the control and status registers for the ASIC(s) in the Power Brick AC. (The **PowerBrick[*i*].** data structure is an "alias" for the **Gate3[*i*].** data structure.)

BrickLV. Multi-Channel Status Elements

Some aspects of the Brick LV amplifier are common to all channels on the board. The setup elements in this section affect all channels.

BrickLV.BusOverVoltage

Description: DC bus overvoltage fault flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.BusOverVoltage** status bit indicates whether the DC bus voltage supplied to Power Brick LV is above a maximum threshold or not. It is set to 0 if the measured DC bus voltage is 80V or less. It is set to 1 if the measured DC bus voltage is greater than 80V.

BrickLV.BusOverVoltage is a fault flag. If this fault is detected, the amplifier-fault lines for all channels are set to the "true" state, causing a software fault condition on all Power PMAC motors commanding these channels. It is a transparent status bit; as soon as the measured voltage no longer exceeds 80V, the value of this bit is cleared to 0. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command. This status bit is only updated if **BrickLV.Monitor** is set to 1.

Note

The amplifier will shut down with a fault on all channels when it detects an overvoltage condition regardless of whether software status bits are updated for the processor (**BrickLV.Monitor** = 1) or not.

BrickLV.BusUnderVoltage

Description: DC bus undervoltage warning flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.BusUnderVoltage** status bit indicates whether the DC bus voltage supplied to Power Brick LV is above a minimum threshold or not. It is set to 0 if the measured DC bus voltage is 12V or more. It is set to 1 if the measured DC bus voltage is less than 12V.

BrickLV.BusUnderVoltage is a warning flag; there is no fault condition generated if it is set to 1. It is a transparent status bit; as soon as the measured voltage reaches 12V again, the value of this bit is cleared to 0. This status bit is only updated if **BrickLV.Monitor** is set to 1.

BrickLV.OverTemp

Description: Power board overtemperature flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.OverTemp** status bit indicates whether the measured temperature of the power board is above a maximum threshold or not. It is set to 0 if the measured board temperature is 70°C or less. It is set to 1 if the measured board temperature is over 70°C.

BrickLV.OverTemp is a fault flag. If this fault is detected, the amplifier-fault lines for all channels are set to the "true" state, causing a software fault condition on all Power PMAC motors commanding these channels. It is a transparent status bit; as soon as the measured temperature no longer exceeds 70°C, the value of this bit is cleared to 0. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command. This status bit is only updated if **BrickLV.Monitor** is set to 1.

Note

The amplifier will shut down with a fault on all channels when it detects an overtemperature condition regardless of whether software status bits are updated for the processor (**BrickLV.Monitor** = 1) or not.

BrickLV. Single-Channel Status Elements

Some status flags/registers of the Brick LV are defined individually for each channel. The channel index ***j*** has a range from 0 to 7, corresponding to hardware channel 1 to 8 on the board. The channel index is one less than the corresponding hardware channel number.

BrickLV.Chan[*j*].ActivePhaseMode

Description: Channel active output phase mode configuration

Range: 0 .. 1

Units: Boolean

The **BrickLV.Chan[*j*].ActivePhaseMode** status bit indicates whether the channel is presently configured for 3-phase output or 2-phase output. It is set to 0 if the channel is configured for 3-phase output on the U, V, and W motor lines. It is set to 1 if the channel is configured for 2-phase output, with one phase on the U and W motor lines, and the other on the V and X motor lines.

The phase configuration is determined by the value of saved setup element **BrickLV.Chan[*j*].TwoPhaseMode**, but the value of this saved element is not copied into the active amplifier control circuits until the amplifier is successfully reset and/or configured by setting **BrickLV.Reset** or **BrickLV.Config** to 1 in a Script command. This status element can be used to confirm whether the configuration was completed successfully or not. It is only updated if **BrickAC.Monitor** is set to 1.

BrickLV.Chan[*j*].I2tExcess

Description: Channel I^2T fault/warning flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.Chan[*j*].I2tExcess** status bit indicates whether an excessive integrated current (I^2T) condition is present on the channel or not. It is set to 0 if the integrated current value is not excessive; it is set to 1 if it is excessive. This status flag is only updated if **BrickLV.Monitor** is set to 1.

An excessive I^2T condition will generate a fault if saved setup element **BrickLV.Chan[*j*].I2tWarnOnly** is set to its default value of 0. It will not generate a fault if **BrickLV.Chan[*j*].I2tWarnOnly** is set to 1.

BrickLV.Chan[*j*].I2tExcess is a transparent status bit and it will be cleared to 0 as soon as the integrated current value falls below the threshold again. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The channel will shut down with a fault when it detects an I^2T excess condition if **BrickLV.Chan[*j*].I2tWarnOnly** is set to 0 regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

Note

The integrated current (I^2T) calculations accessed by this element are performed in the amplifier stage of the Power Brick LV. These calculations are separate from those done by the Power PMAC software.

BrickLV.Chan[*j*].OverCurrent

Description: Channel overcurrent fault flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.Chan[*j*].OverCurrent** status bit indicates whether the hardware over-current detector for the channel has sensed an instantaneous overcurrent or short-circuit state for the channel or not. It is set to 0 if it has not detected this state. It is set to 1 if it has detected this state. This status flag is only updated if **BrickAC.Monitor** is set to a value greater than 0.

Over-current fault detection in Power Brick AC is performed in hardware. Once over-current fault is detected, the fault status is latched. This fault can be cleared by setting **BrickAC.Reset** equal to 1. Any motor software fault conditions it creates are also latched, and the motors must explicitly be re-enabled by command after this fault is cleared.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The channel will shut down with a fault when it detects an over-current condition regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

BufIo[*i*]. Buffered I/O Status Data Structure Elements

The **BufIo[*i*]** data structure contains several status elements for use in the buffered input/output functionality.

BufIo[*i*].FallIn

Description: Buffered input transparent fallen-bit register

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

BufIo[*i*].FallIn is a 32-bit register in which each bit indicates whether the corresponding bit in the buffered input holding register **BufIo[*i*].In** has changed from 1 to 0 in the most recent scan or not. If the bit **BufIo[*i*].In** has not changed since the previous scan, or it has changed from 0 to 1, this bit in **BufIo[*i*].FallIn** will have a value of 0 this scan. If the bit in **BufIo[*i*].In** has changed from 1 to 0 since the previous scan, this bit in **BufIo[*i*].FallIn** will have a value of 1 this scan.

This functionality is enabled if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pIn** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this input. If input filtering is enabled by setting **BufIo[*i*].InScans** greater than 0, **BufIo[*i*].FallIn** will reflect changes in the filtered bits, not necessarily the raw input bits.

BufIo[*i*].FallIn is useful for triggering actions easily on changes in input states, as the user algorithms do not have to store previous states and perform comparisons explicitly. The user code must execute once for each input scan, because a bit in **BufIo[*i*].FallIn** will only be true for a single input scan on a change.

The "transparent" status of bits in **BufIo[*i*].FallIn** is contrasted with the "latched" status of bits in **BufIo[*i*].FallInLatch**, which stay true when an edge is found until cleared by the user. Different programming styles will prefer the use of on or the other of these elements.

An individual bit of **BufIo[*i*].FallIn** can be accessed with the syntax **BufIo[*i*].FallIn.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].FallIn** can be accessed with the syntax **BufIo[*i*].FallIn.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].FallIn cannot be written to by user Script code. It is new in V2.1 firmware, released 1st quarter 2016.

Example

```
ptr JogPlusButtonState-> BufIo[2].In.13
ptr JogPlusButtonPressed->BufIo[2].FallIn.13
…
if (JogPlusButtonPressed) jog+1
```

BufIo[*i*].FallInLatch

Description: Buffered input latched fallen-bit register

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

BufIo[*i*].FallInLatch is a 32-bit register in which each bit indicates whether the corresponding bit in the buffered input holding register **BufIo[*i*].In** has changed from 1 to 0 since the bit in **BufIo[*i*].FallInLatch** was last cleared or not. If the bit **BufIo[*i*].In** has not changed since the last clearing, or it has changed from 0 to 1, this bit in **BufIo[*i*].FallInLatch** will have a value of. If the bit in **BufIo[*i*].In** has changed from 1 to 0 since the last clearing, this bit in **BufIo[*i*].FallInLatch** will have a value of 1.

This functionality is enabled if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pIn** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this output. If input filtering is enabled by setting **BufIo[*i*].InScans** greater than 0, **BufIo[*i*].FallInLatch** will reflect changes in the filtered bits, not necessarily the raw input bits.

BufIo[*i*].FallInLatch is useful for triggering actions easily on changes in input states, as the user algorithms do not have to store previous states and perform comparisons explicitly. The use of **BufIo[*i*].FallInLatch** is different from the corresponding "transparent" **BufIo[*i*].FallIn**, in which the bit is only set for the single scan when the edge is found.

Application code using a bit of **BufIo[*i*].FallInLatch** may want to clear the bit explicitly before starting to look for the edge. Also, it may be useful for the code to clear the bit after the edge is found, so that the action taken on the edge is not repeated.

An individual bit of **BufIo[*i*].FallInLatch** can be accessed with the syntax **BufIo[*i*].FallInLatch.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].FallInLatch** can be accessed with the syntax **BufIo[*i*].FallInLatch.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].FallInLatch is new in V2.1 firmware, released 1st quarter 2016.

Example

```
ptr JogPlusButtonState-> BufIo[2].In.13
ptr JogPlusButtonPressed-> BufIo[2].FallInLatch.13
…
if (JogPlusButtonPressed) {
  jog+1;
  JogPlusButtonPressed = 0;
}
```

BufIo[*i*].In

Description: Buffered input register

Range: $0 .. $FFFFFFFF (0 .. 2^{32} – 1)

Units: Bit field

BufIo[*i*].In is the buffered 32-bit input holding register for the input register specified by **BufIo[*i*].pIn**. It will contain a copy (possibly filtered) of the contents of the input register if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pIn** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this input.

If these conditions are true, then at the beginning of each scan, Power PMAC will read the input register at the address specified by **BufIo[*i*].pIn** to copy into **BufIo[*i*].In**. However, if filtering is enabled by a setting of **BufIo[*i*].InScans** greater than 0, a new state of any bit in the input register must persist for (**InScans** + 1) consecutive scans before the change is reflected in the bit of **BufIo[*i*].In**.

If bits of **BufIo[*i*].ForceInOn** or **BufIo[*i*].ForceInOff** are set to 1, the value of the corresponding bits of **BufIo[*i*].In** will be determined by these forcing elements, not the actual input register.

Each scan, the bits of **BufIo[*i*].In** are compared to the matching bits of the previous scan's values. Any bits that changed from 0 to 1 since the previous scan will have the corresponding bit of **BufIo[*i*].RiseIn** set to 1; any bits that changed from 1 to 0 since the previous scan will have the corresponding bit of **BufIo[*i*].FallIn** set to 1.

An individual bit of **BufIo[*i*].In** can be accessed with the syntax **BufIo[*i*].In.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].In** can be accessed with the syntax **BufIo[*i*].In.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].In cannot be written to by user Script code. It is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].LastOut

Description: Previous scan's buffered output register

Range: $0 .. $FFFFFFFF (0 .. 2^{32} – 1)

Units: Bit field

BufIo[*i*].LastOut is a 32-bit register that contains the contents of the previous scan's **BufIo[*i*].Out** buffered outputs. At the end of the current scan, Power PMAC will compare the contents of **BufIo[*i*].Out**, which may have been changed by the user, to the contents of **BufIo[*i*].LastOut**. Only if there is a difference between the two will Power PMAC write to the output register specified by **BufIo[*i*].pOut** at the end of the scan. This can make the I/O processes more efficient by eliminating unneeded I/O write accesses.

BufIo[*i*].LastOut is for internal use by Power PMAC. It cannot be written to by user Script code. It is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].RawIn[*j*]

Description: Raw buffered input register for filtering

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

BufIo[*i*].RawIn[*j*] is the raw buffered 32-bit input holding register that is "***j***" scans old for the input register specified by **BufIo[*i*].pIn** if filtering is enabled for that input register by setting **BufIo[*i*].InScans** greater than 0. It will contain a copy of the contents of the input register if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pIn** is non-zero for all index values less than or equal to ***i***.

The index ***i*** can take a value from 0 to 63. The index ***j*** can take a value from 0 to 3. Only those elements required for the filtering of **BufIo[*i*].InScans** are actually used.

BufIo[*i*].RawIn[*j*] is intended for internal use, to hold the input data from multiple scans for the purpose of filtering the input bits for the holding register **BufIo[*i*].In** to eliminate the effects of input "bounce" and electrical noise. The user may access it for debugging purposes, but it is doubtful that it will be useful in the actual application. It cannot be written to by user Script code.

BufIo[*i*].RawIn[*j*] is new in V2.1 firmware, released 1st quarter 2016.

BufIo[*i*].RiseIn

Description: Buffered input risen-bit register

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

BufIo[*i*].RiseIn is a 32-bit register in which each bit indicates whether the corresponding bit in the buffered input holding register **Sys.BufIn[*i*]** has changed from 0 to 1 in the most recent scan or not. If the bit in **Sys.BufIn[*i*]** has not changed since the previous scan, or it has changed from 1 to 0, this bit in **BufIo[*i*].RiseIn]** will have a value of 0 this scan. If the bit in **Sys.BufIn[*i*]** has changed from 0 to 1 since the previous scan, this bit in **BufIo[*i*].RiseIn** will have a value of 1 this scan.

This functionality is enabled if **Sys.BufIoEnable** is set greater than 0 and **Sys.pBufIn[*i*]** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this input. If input filtering is enabled by setting **Sys.BufInScans** greater than 0, **BufIo[*i*].RiseIn** will reflect changes in the filtered bits, not necessarily the raw input bits.

BufIo[*i*].RiseIn is useful for triggering actions easily on changes in input states, as the user algorithms do not have to store previous states and perform comparisons explicitly. The user code

must execute once for each input scan, because a bit in **BufIo[*i*].RiseIn** will only be true for a single input scan on a change.

The "transparent" status of bits in **BufIo[*i*].RiseIn** is contrasted with the "latched" status of bits in **BufIo[*i*].RiseInLatch**, which stay true when an edge is found until cleared by the user. Different programming styles will prefer the use of on or the other of these elements.

An individual bit of **BufIo[*i*].RiseIn** can be accessed with the syntax **BufIo[*i*].RiseIn.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].RiseIn** can be accessed with the syntax **BufIo[*i*].RiseIn.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].RiseIn cannot be written to by user Script code. It is new in V2.1 firmware, released 1st quarter 2016.

Examples

```
ptr CS1InPosState->BufIo[4].In.11
ptr CS1InPosRiseEdge->BufIo[4].RiseIn.11
…
if (CS1InPosRiseEdge) LaserOn=1
```

BufIo[*i*].RiseInLatch

Description: Buffered input latched risen-bit register

Range: $0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

BufIo[*i*].RiseInLatch is a 32-bit register in which each bit indicates whether the corresponding bit in the buffered input holding register **BufIo[*i*].In** has changed from 0 to 1 since the bit in **BufIo[*i*].RiseInLatch** was last cleared or not. If the bit **BufIo[*i*].In** has not changed since the last clearing, or it has changed from 1 to 0, this bit in **BufIo[*i*].RiseInLatch** will have a value of. If the bit in **BufIo[*i*].In** has changed from 0 to 1 since the last clearing, this bit in **BufIo[*i*].RiseInLatch** will have a value of 1.

This functionality is enabled if **Sys.BufIoEnable** is set greater than 0 and **BufIo[*i*].pIn** is non-zero for all index values less than or equal to ***i*** for the cycle (real-time interrupt or background) used for this input. If input filtering is enabled by setting **BufIo[*i*].InScans** greater than 0, **BufIo[*i*].RiseInLatch** will reflect changes in the filtered bits, not necessarily the raw input bits.

BufIo[*i*].RiseInLatch is useful for triggering actions easily on changes in input states, as the user algorithms do not have to store previous states and perform comparisons explicitly. The use of **BufIo[*i*].RiseInLatch** is different from the corresponding "transparent" **BufIo[*i*].RiseIn**, in which the bit is only set for the single scan when the edge is found.

Application code using a bit of **BufIo[*i*].RiseInLatch** may want to clear the bit explicitly before starting to look for the edge. Also, it may be useful for the code to clear the bit after the edge is found, so that the action taken on the edge is not repeated.

An individual bit of **BufIo[*i*].RiseInLatch** can be accessed with the syntax **BufIo[*i*].RiseInLatch.*j*** – where ***j*** is the bit number (0 to 31) – in an on-line command, an expression, or a pointer (M) variable definition.

Similarly, multiple consecutive bits of **BufIo[*i*].RiseInLatch** can be accessed with the syntax **BufIo[*i*].RiseInLatch.*j*.*k*** – where ***j*** is the starting (lowest) bit number and ***k*** is the number of bits – in any of these methods.

BufIo[*i*].RiseInLatch is new in V2.1 firmware, released 1st quarter 2016.

Example

```
ptr CS1InPosState->BufIo[4].In.11
ptr CS1InPosRiseEdge->BufIo[4].RiseInLatch.11
…
if (CS1InPosRiseEdge) {
  LaserOn=1;
  CS1InPosRiseEdge = 0;
}
```

CamTable[*m*]. Status Data Structure Elements

CamTable[*m*].ActivePosOffset

Description: Cam table present slave position offset

Range: Floating-point

Units: Target motor position units

CamTable[*m*].ActivePosOffset contains the value that is added to the target motor's position value calculated from the table **PosData[*i*]** entries in the present servo cycle, after that value has been multiplied by the **CamTable[*m*].PosSf** scale factor. If it is different from the sum of the values in non-saved setup element **CamTable[*m*].PosOffset** and saved setup element **CamTable[*m*].PosBias**, it will change each servo cycle by the magnitude of saved setup element **CamTable[*m*].SlewPosOffset** until the user-specified value is reached.

When the table is enabled, **CamTable[*m*].ActivePosOffset** is automatically set to the difference between the present value in the target motor's **Motor[*x*].CompDesPos** register and the value calculated by the table for this register. Then each following servo cycle, it is changed by the magnitude of **CamTable[*m*].SlewPosOffset** until it reaches the value automatically calculated for **CamTable[*m*].PosOffset** for the table. This allows for controlled synchronization of the table on enabling.

If the user changes the value of **CamTable[*m*].PosOffset** or **CamTable[*m*].PosBias** while the table is enabled, **ActivePosOffset** will change at this same controlled rate each servo cycle until it matches the value of the sum of **PosOffset** and **PosBias**. The gradual change in **CamTable[*m*].ActivePosOffset** permits the user to offset the position of the target motor for the table in a controlled fashion while the table is enabled. For comparable controlled changes to the reference source motor position, **CamTable[*m*].ActiveX0** is used.

CamTable[*m*].ActiveX0

Description: Cam table present slave position offset

Range: Floating-point

Units: User-defined table position units

CamTable[*m*].ActiveX0 contains the present actual starting (minimum) position of the source motor for the table. If it is different from the user-specified value in saved setup element **CamTable[*m*].X0**, it will change each servo cycle by the magnitude of saved setup element **CamTable[*m*].SlewX0** until the user-specified value is reached.

The gradual change in **CamTable[*m*].ActiveX0** permits the user to offset the reference position of the source motor for the table in a controlled fashion while the table is enabled. For comparable controlled changes to the target motor position, **CamTable[*m*].ActivePosOffset** is used.

Cid[*j*]. Card ID Status Elements

The **Cid[*j*]** card identification status data structure elements contain information about the accessory "cards" present in the Power PMAC system. (Note that they do not actually have to be separate printed circuit cards, although they often are.) This information is automatically detected by Power PMAC on power-up/reset.

The index values ***j*** can have a range of 0 to 63. These map to the index values of the different types of accessories according to the following table:

Cid[*j*]	**Gate*xx*[*i*]**	**Cid[*j*]**	**Gate*xx*[*i*]**	**Cid[*j*]**	**Gate*xx*[*i*]**	**Cid[*j*]**	**Gate*xx*[*i*]**
Cid[0]	*{reserved}*	Cid[16]	*{reserved}*	Cid[32]	*{reserved}*	Cid[48]	*{reserved}*
Cid[1]	*{reserved}*	Cid[17]	*{reserved}*	Cid[33]	*{reserved}*	Cid[49]	*{reserved}*
Cid[2]	Gate1[4]	Cid[18]	Gate1[8]	Cid[34]	Gate1[12]	Cid[50]	Gate1[16]
Cid[3]	Gate1[6]	Cid[19]	Gate1[10]	Cid[35]	Gate1[14]	Cid[51]	Gate1[18]
Cid[4]	Gate2[0]	Cid[20]	Gate2[1]	Cid[36]	Gate2[2]	Cid[52]	Gate2[3]
Cid[5]	Gate2[4]	Cid[21]	Gate2[5]	Cid[37]	Gate2[6]	Cid[53]	Gate2[7]
Cid[6]	Gate2[8]	Cid[22]	Gate2[9]	Cid[38]	Gate2[10]	Cid[54]	Gate2[11]
Cid[7]	Gate2[12]	Cid[23]	Gate2[13]	Cid[39]	Gate2[14]	Cid[55]	Gate2[15]
Cid[8]	Dpr[0]	Cid[24]	Dpr[2]	Cid[40]	Dpr[4]	Cid[56]	Dpr[6]
Cid[9]	Dpr[1]	Cid[25]	Dpr[3]	Cid[41]	Dpr[5]	Cid[57]	Dpr[7]
Cid[10]	Gate1[5]	Cid[26]	Gate1[9]	Cid[42]	Gate1[13]	Cid[58]	Gate1[17]
Cid[11]	Gate1[7]	Cid[27]	Gate1[11]	Cid[43]	Gate1[15]	Cid[59]	Gate1[19]
Cid[12]	GateIo[0]	Cid[28]	GateIo[4]	Cid[44]	GateIo[8]	Cid[60]	GateIo[12]
Cid[13]	GateIo[1]	Cid[29]	GateIo[5]	Cid[45]	GateIo[9]	Cid[61]	GateIo[13]
Cid[14]	GateIo[2]	Cid[30]	GateIo[6]	Cid[46]	GateIo[10]	Cid[62]	GateIo[14]
Cid[15]	GateIo[3]	Cid[31]	GateIo[7]	Cid[47]	GateIo[11]	Cid[63]	GateIo[15]

Cid[*j*].dir

Description: Card clock buffer direction

Range: 0 .. 1

Units: Boolean

Cid[*j*].dir contains the direction of the clock buffer circuit of the accessory residing at the address mapped to **Cid[*j*]**. It is set to 0 if the accessory is receiving the system phase and servo clocks from the central system through the buffer. It is set to 1 if the accessory is transmitting its phase and servo clocks to the central system through the buffer.

The direction sense of the clock buffer must match the direction sense of the clock generation circuits in the Servo or MACRO IC of the accessory. This direction sense is determined by saved setup element **Gate*n*[*i*].PhaseServoDir** for the IC, which is set to 3 if it expects to input and use external servo clocks (this setting works with **Cid[*j*].dir** = 0), or to 0 if it generates its own phase and servo clocks and outputs them (this setting works with **Cid[*j*].dir** = 1).

For cards utilizing the PMAC3-style DSPGATE3 IC (e.g. ACC-24E3, ACC-5E3), the direction of the clock buffer is controlled directly from the ASIC and does not use this element.

On re-initialization, Power PMAC automatically determines which IC will provide the system clock signals and sets the IC clock generation circuits and the clock buffer directions accordingly. On a standard power-up/reset, Power PMAC uses the saved values of **Gate*n*[*i*].PhaseServoDir** to determine the settings of the corresponding **Cid[*j*].dir** elements, which it makes automatically.

It is possible to change the clock source from the default selected at re-initialization. The multiple commands needed to do this must be done in a single command line, and the new IC should be made to output its clocks before the old IC is made to input its clocks. Otherwise, the temporary loss of system clocks by the processor may trip the watchdog timer. For example, the command line:

```
Gate1[4].PhaseServoDir=0 Cid[2].dir=1 Gate2[0].PhaseServoDir=3 Cid[4].dir=0
```

makes the first PMAC2-style Servo IC the source of the system clocks, telling its buffer to transmit the clocks, while setting the first PMAC2-style MACRO IC to input the system clocks, telling its buffer to receive the clocks. These settings must be saved to be used subsequently.

Cid[*j*].num

Description: Card part number

Range: 600000 .. 609999

Units: Enumeration

Cid[*j*].num contains the 6-digit part number for the hardware design of the accessory residing at the address mapped to **Cid[*j*]**.

Cid[*j*].opt

Description: Card option code

Range: 0 .. 4095

Units: Bit field

Cid[*j*].opt contains the option code for the hardware design of the accessory residing at the address mapped to **Cid[*j*]**. The option code is specific to the particular accessory, and is usually a bit field representing the presence or absence of several individual options.

Cid[*j*].rev

Description: Card hardware revision number

Range: 0 .. 15

Units: Enumeration

Cid[*j*].rev contains the design revision number for the hardware design of the accessory residing at the address mapped to **Cid[*j*]**. The higher the revision number, the newer the revision is.

Cid[*j*].ven

Description: Card vendor number

Range: 0 .. 255

Units: Enumeration

Cid[*j*].ven contains the identification number of the vendor of the accessory residing at the address mapped to **Cid[*j*]**. Delta Tau's vendor identification number is 1.

Clipper[*i*]. Non-Saved Data Structure Elements

The **Clipper[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the Power Clipper controller board are listed under the **Gate3[*i*]** data structure, below.

Coord[*x*]. Coordinate System Status Elements

The coordinate-system data structures in Power PMAC provide many status elements that may be of interest to users for developing or debugging an application, or in the actual execution of the application. This section documents the key coordinate-system status elements.

Coord[*x*].ActSegOverride

Description: Present "segmentation feedrate override" value

Range: Floating-point

Units: none (ratio)

Coord[*x*].ActSegOverride contains the present "segmentation override" value used in the most recent coarse-interpolation calculations for segmented moves (**`linear`**, **`circle`**, and **`pvt`**, with **Coord[*x*].SegMoveTime** > 0). It is a normalized value (not a percentage), so a value of 1.0 commands "real time" execution.

In segmented moves, every **Coord[*x*].SegMoveTime** milliseconds of actual time, Power PMAC computes the next segment's position. In doing so, it advances the numerical time of the motion equations by the product of **Coord[*x*].SegMoveTime** and the override value. For example, if **Coord[*x*].SegMoveTime** is set to 5.0 msec, and the override value is set to 1.5, each segment point computed and executed 5.0 msec apart advances the motion equations by 5.0 * 1.5 = 7.5 msec, so the move is executed at 150% of the programmed speed.

When the commanded value **Coord[*x*].SegOverride** is changed, the internal value **Coord[*x*].ActSegOverride** that is actually used to compute the time advance in the move equations for a segment is changed by the amount in saved setup element **Coord[*x*].SegOverrideSlew** each segment until the new commanded value is reached. This prevents step changes in the resulting velocity.

Coord[*x*].AddedDwellDis

Description: "Added corner dwell disabled" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].AddedDwellDis** status bit is set to 1 if blending has been disabled at a corner because the angle is sharper than that specified by **Coord[*x*].CornerBlendBp**, but there is no added dwell at the corner because the angle is not as sharp as that specified by **Coord[*x*].CornerDwellBp**. It is 0 otherwise. It is bit 0 of 32-bit element **Coord[*x*].Status[1]**.

Coord[x].AmpEna

Description: "Amplifier enabled" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].AmpEna** status bit is set to 1 if all motors in the coordinate system are enabled (in closed-loop or open-loop control). Note that there do not need to be active amplifier-enable output signals in this case. This bit is 0 if any motor in the coordinate system is disabled. It is bit 12 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].AmpFault

Description: "Amplifier fault error" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].AmpFault** status bit is 1 if any motor in the coordinate system has been disabled because of an amplifier fault error, even if the amplifier fault signal condition is no longer present, or because of a calculated "I^2T" integrated current fault (in which case bit 21 is also set). It is 0 at all other times, becoming 0 again when the motor is re-enabled. It is bit 24 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].AmpWarn

Description: "Amplifier warning" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].AmpWarn** status bit is set to 1 if, for any motor in the coordinate system, **Motor[*x*].AmpFaultLevel** bit 1 (value 2) is set to 1, requiring two consecutive readings of the amplifier fault bit in its specified fault state to trigger an error, and there has been one reading of the amplifier fault bit in its fault state. It is bit 19 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].AuxFault

Description: "Auxiliary fault error" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].AuxFault** status bit is 1 if any motor in the coordinate system has been disabled because of an "auxiliary fault" error, even if the auxiliary fault signal condition is no longer present. It is 0 at all other times, becoming 0 again when the motor is re-enabled. It is bit 17 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].BlockActive

Description: "Single-step block active" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].BlockActive** status bit is set to 1 if the coordinate system is presently calculating between a **`bstart`** (block-start) and a **`bstop`** (block-stop) program command in "single-step" mode. It is 0 otherwise. It is primarily for internal use. It is bit 27 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].BlockRequest

Description: "Block request flag set" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].BlockRequest** status bit is set to 1 if any motor in the coordinate system has just entered a new move section, or has just been commanded to start a program, and is requesting that the equations for the next upcoming programmed move for the motion queue be calculated. It is 0 otherwise. It is primarily for internal use. It is bit 9 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].BufferWarn

Description: Move equation buffer near overflow warning status

Range: 0 .. 2

Units: Enumeration

Coord[*x*].BufferWarn indicates the potential overflow status of the move equation buffer for the coordinate system. It can be useful in optimizing the setup for applications with very high move block rates (thousands of blocks per second) that can require a very high calculation load in a single real-time interrupt.

If there is no potential overflow, **Coord[*x*].BufferWarn** reports as 0. If the move equation buffer is full enough that one more move section would overflow the buffer, it reports as 1. If the most recent RTI has requested move equations that would have overflowed the buffer, it reports as 2.

When this state occurs, Power PMAC will delay the request by one RTI period; if it still would overfl0w the buffer in the next period, the program is stopped with a run-time error.

Saved setup element **Sys.PreCalc** tells each coordinate system the minimum time ahead in servo cycles it should try to fill the move equation buffer when executing motion programs. In very high block-rate applications, the user may have to set this greater than the default value of 1 to ensure that the move equation buffer always has move sections ready for the active interpolation engine.

Coord[*x*].BufferWarn can be useful to optimize the setting of **Sys.PreCalc** in these high block-rate equations. **Sys.PreCalc** should be set high enough that **Coord[*x*].BufferWarn** occasionally reports as 1, but not so high that it reports as 2.

If **Sys.PreCalc** is set too low, the motion program can stop with a run-time error (**Coord[*x*].RunTimeError** = 1) because the active interpolation engine does not get the next move equations in time. If **Sys.PreCalc** is set too high, the motion program can stop with a buffer error (**Coord[*x*].ErrorStatus** = 2) because the move equation buffer has overflowed.

Coord[*x*].CC3Active

Description: "3D cutter comp active" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].CC3Active** status bit is set to 1 if three-dimensional cutter radius compensation is presently active. It is 0 otherwise. It is bit 2 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].CCAddedArc

Description: "Cutter comp added arc move" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].CCAddedArc** status bit is set to 1 if the presently calculated move is an automatically added arc move around an outside corner in 2D cutter radius compensation. It is 0 otherwise. It is bit 7 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].CCMode

Description: "Cutter comp mode" status element

Range: 0 .. 3

Units: Enumeration

Coord[*x*].CCMode is a 2-bit status element indicating the present mode of cutter radius compensation for the coordinate system.

- 0: Cutter compensation disabled (**ccmode0**)
- 1: 2D cutter compensation enabled left (**ccmode1**)
- 2: 2D cutter compensation enabled right (**ccmode2**)
- 3: 3D cutter compensation enabled (**ccmode3**)

Coord[*x*].CCMode forms bits 24 and 25 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].CCMoveType

Description: "Cutter comp move type" status element

Range: 0 .. 3

Units: Enumeration

Coord[*x*].CCMoveType is a 2-bit status element indicating the type of move being calculated with 2D cutter radius compensation active. Its possible values and the move types specified are:

- 0: Dwell
- 1: Zero-distance or out-of-plane move
- 2: Linear-mode move
- 3: Circle-mode move

Coord[*x*].CCMoveType forms bits 4 and 5 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].CCOffReq

Description: "Cutter comp turn-off request" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].CCOffReq** status bit is set to 1 if the coordinate system has been told to turn off cutter compensation by a **ccmode0** command but has not yet computed the lead-out move that removes compensation. It is 0 otherwise. It is bit 6 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].cdata

Description: Circle move mode data

Range: 0 .. 255

Units: Bit field

Coord[*x*].cdata contains modal information about how circle moves execute in the coordinate system. It is an 8-bit value, with each bit having a specific meaning:

- Bit 0: XX/YY/ZZ direction mode (0 = clockwise, 1 = counterclockwise)
- Bit 1: II vector specification mode (0 = incremental, 1 = absolute)
- Bit 2: JJ vector specification mode (0 = incremental, 1 = absolute)
- Bit 3: KK vector specification mode (0 = incremental, 1 = absolute)
- Bit 4: X/Y/Z direction mode (0 = clockwise, 1 = counterclockwise)
- Bit 5: I vector specification mode (0 = incremental, 1 = absolute)
- Bit 6: J vector specification mode (0 = incremental, 1 = absolute)
- Bit 7: K vector specification mode (0 = incremental, 1 = absolute)

The values of bits 0 and 4 are set by **`circlen`** program commands; the values of the other bits are set by **`abs({vector list})`** and **`inc({vector list})`** program commands.

Coord[*x*].CdPos[*j*]

Description: Axis move commanded position

Range: Floating-point

Units: Axis position units

Coord[*x*].CdPos[*j*] contains the most recently calculated commanded move target position for the specified axis. If the coordinate system is currently executing a continuous move sequence, the move for which this contains data may be one or more moves ahead of the presently executing move. (The axis target position for the presently executing move can be found in **Coord[*x*].TPExec.Pos[*j*]** if a target position buffer has been set up.

Axis index values *j* range from 0 to 31, with the axis name for each index value shown in the following table:

Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*
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

Coord[x].Cflags

Description: Array of "conditional execution" status bits

Range: $0 .. $FFFFFFFF

Units: Bit field

Coord[*x*].Cflags is a 32-bit status word containing the status for each of the possible 32 conditional execution flags for the coordinate system. A bit is 1 if the corresponding flag is set. It is 0 if the corresponding flag is cleared. At power-on/reset, all flags are cleared.

Bit *n* of **Coord[*x*].Cflags** is used to determine the action of the conditional-execution buffered program commands **cexec*n*** and **cskip*n*** in the coordinate system.

Values in **Coord[*x*].Cflag** are usually set using the buffered program commands **cset*n*** and **cclr*n***, but it is possible to set and clear bits of **Coord[*x*].Cflags** directly.

Coord[x].ClosedLoop

Description: "Closed-loop mode" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].ClosedLoop** status bit is set to 1 if all motors in the coordinate system are in closed-loop control. It is zero if any motor in the coordinate system is in open-loop mode (enabled or disabled). It is bit 13 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].ContMotion

Description: "Continuous motion request" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].ContMotion** status bit is set to 1 if the coordinate system is executing a sequence of programmed moves that it is blending together without stops in between. It is 0 if it is not currently executing such a sequence, for whatever reason. It is bit 26 of 32-bit element **Coord[*x*].Status[1]**.

Coord[x].Csolve

Description: "Valid axis definitions for PMATCH calculations" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].Csolve** status bit is set to 1 if the axis definition statements for this coordinate system are satisfactory to calculate axis positions in a coordinate-system "PMATCH" (position match) function (at motion-program start, **pmatch** command execution, axis position/velocity/following-error query). It is 0 otherwise.

If the coordinate system axes are assigned using axis definition statements, this bit must be 1 in order to run a motion program. However, if there is a forward kinematic subroutine for the coordinate system to calculate axis positions from motor positions, this **Csolve** status bit is not evaluated in the decision as to whether a motion program can be run in the coordinate system.

Coord[*x*].Csolve is bit 31 of 32-bit element **Coord[*x*].Status[1]**.

Coord[x].DesVelZero

Description: "Desired velocities zero" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].DesVelZero** status bit is set to 1 if all motors in the coordinate system are in closed-loop control and the net commanded velocity is exactly zero (i.e. are trying to hold position), or if they are in open-loop mode (enabled or disabled) and the actual velocity is exactly zero. It is zero either if any motor in the coordinate system is in closed-loop mode with non-zero commanded velocity, or if is in open-loop mode (enabled or disabled) with non-zero actual velocity. It is bit 14 of 32-bit element **Coord[*x*].Status[0]**.

Saved setup element **Sys.ZeroVelSetPoint** can be used to set a threshold for "desired velocity zero" that is not exactly 0.0. It is typically used with a trajectory pre-filter that creates an "infinite impulse response" filter that gradually decays to zero velocity at the end of a programmed move, so it does not have to wait for an exact 0.0 value (which is obtained only by numerical underflow) to be reached. **Sys.ZeroVelSetPoint** is new in V2.0.2 firmware, released 2nd quarter 2015.

Coord[x].EncLoss

Description: "Encoder loss error" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].EncLoss** status bit is 1 if any motor in the coordinate system has been disabled because of an "encoder loss" error, even if the encoder loss signal condition is no longer present. It is 0 at all other times, becoming 0 again when the motor is re-enabled. It is bit 18 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].EndDelayActive

Description: Automatically-added delay time in- progress status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].EndDelayActive** status bit is 1 if the coordinate system is presently executing an automatically added delay time at the end of a move sequence, as specified by saved setup element **Coord[*x*].EndDelay**. It is 0 otherwise. It is bit 17 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].EndDelayActive is new in V2.1 firmware, released 1st quarter 2016.

Coord[x].ErrorStatus

Description: "Error code" status element

Range: 0 .. 255

Units: Enumeration

Coord[*x*].ErrorStatus is an 8-bit status element indicating the type of error that has occurred in the coordinate system. Its possible values and the move types specified are:

Value	Error Name	Description
0	NoError	Normal execution
1	**SyncBufferError**	Error in synchronous variable assignment buffer
2	**BufferError**	Error in motion program buffer
3	CCMoveTypeError	Illegal move mode or command while cutter compensation active (rapid, pvt, spline, lin-to-pvt, new normal, pmatch, pclr, pset, pload)
4	**LinToPvtError**	Error in automatic linear-to-PVT-mode conversion
5	CCLeadOutMoveError	Illegal cutter compensation lead-out move (circle mode or length less than cutter radius)
6	CCLeadInMoveError	Illegal cutter compensation lead-in move (circle mode or length less than cutter radius)
7	CCBufSizeError	Insufficient size for cutter compensation move buffer (not enough to find next in-plane move)
8	**PvtError**	Illegally specified PVT-mode 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
12	CC3NdotTError	3D cutter compensation vector calculation error ("NdotT" value less than 1st entry in tool-tip table)
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
16	**RunTimeError**	Insufficient move calculation time
17	InPosTimeOutError	Exceeded specified time limit for in-position
18	LHNumMotorsError	Mismatch between # of motors used and # in lookahead buffer
19	TraceBufferError	Error in reversal through trace buffer
20	TimedUnderflowError	Too much timebase reversal without trace buffer
21	FeedRateError	Stopped from illegal feedrate (with **FProtect** = 1)
22–31	(*Reserved for future use*)	
32	**SoftLimitError**	Stopped from exceeding software position limit
33–63	(*Reserved for future use*)	
64	**RadiusError** bit 0	Radius error in X/Y/Z-space circle move
65–127	(*Reserved for future use*)	
128	**RadiusError** bit 1	Radius error in XX/YY/ZZ-space circle move
129–254	(*Reserved for future use*)	
255	(User-set error)	Application can set to cause program abort

If **Coord[*x*].ErrorStatus** is equal to a power of 2 (only a single bit set), the error that has occurred is related to the name of the status element for the matching bit, shown in bold in the

table. However, if it is not equal to a power of 2 (more than a single bit set), the error indicated is a function of the combined value of all 8 bits.

While generally used as a read-only status word, the user's application can set the value of this element to 255 to cause an abort of motion program. This is usually done in the forward or inverse kinematic subroutine if an error has been found in the routine.

Coord[*x*].ErrorStatus forms bits 0 through 7 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].FeedHold

Description: "Feed hold state" status element

Range: 0 .. 3

Units: Bit field

Coord[*x*].FeedHold is a 2-bit status element indicating the state of the feed hold operation in the coordinate system.

Bit 1 (value 2) is set to 1 if the coordinate system is presently decelerating to a stop due to a "feed hold" (**h**, **hold**) command, or a "quick stop" (****, **lh**) command outside of lookahead, or re-accelerating out of this condition. It is 0 otherwise. It is bit 29 of 32-bit element **Coord[*x*].Status[1]**.

Bit 0 (value 1) is set to 1 if the coordinate system is presently using the "feed hold time base" rather than its normally specified time base. This is the case during deceleration and full stop due to a "feed hold" (**h**, **hold**) command, or a "quick stop" (****, **lh**) command outside of lookahead. It is 0 otherwise, including during re-acceleration from a feed-hold condition. It is bit 28 of 32-bit element **Coord[*x*].Status[1]**.

The 2-bit **Coord[*x*].FeedHold** value will be 3 as the coordinate system is decelerating into a hold state, 1 when it is frozen in the hold state, 2 when it is accelerating out of the hold state, and 0 during normal operation.

Coord[*x*].FeFatal

Description: "Fatal following error" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].TriggerMove** status bit is 1 if any motor in the coordinate system has been disabled because the magnitude of the following error for the motor has exceeded its fatal following error limit as set by **Motor[*x*].FatalFeLimit**. It is 0 at all other times, becoming 0 again when the motor is re-enabled. It is bit 26 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].FeWarn

Description: "Warning following error" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].FeWarn** status bit is 1 if the magnitude of the following error for any motor in the coordinate system exceeds its warning following error limit as set by **Motor[*x*].WarnFeLimit**. It is 0 if the magnitude of the following error is less than this limit, or if the motor has been disabled due to exceeding its fatal following error limit. It is bit 27 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].FRAxes

Description: Array of "vector feedrate" axis status bits

Range: $0 .. $FFFFFFFF

Units: Bit field

Coord[*x*].FRAxes is a 32-bit status word containing the "feedrate axis" status for each of the possible 32 axes in the coordinate system. A bit is 1 if the corresponding axis is used in the vector feedrate calculations for a linear or circle mode move specified by feedrate, where the move time is calculated by dividing the vector distance for the move of all of these axes by the commanded feedrate. A bit is 0 if the corresponding axis not used in these vector feedrate calculations. At power-on/reset, only the X, Y, and Z axes are feedrate axes.

The following table shows for each axis name the bit number and the bit value in **Coord[*x*].FRAxes**:

Axis Name	Bit #	Bit Value (Hex)	Axis Name	Bit #	Bit Value (Hex)
A	0	$1	HH	16	$10000
B	1	$2	LL	17	$20000
C	2	$4	MM	18	$40000
U	3	$8	NN	19	$80000
V	4	$10	OO	20	$100000
W	5	$20	PP	21	$200000
X	6	$40	QQ	22	$400000
Y	7	$80	RR	23	$800000
Z	8	$100	SS	24	$1000000
AA	9	$200	TT	25	$2000000
BB	10	$400	UU	26	$4000000
CC	11	$800	VV	27	$8000000
DD	12	$1000	WW	28	$10000000
EE	13	$2000	XX	29	$20000000
FF	14	$4000	YY	30	$40000000
GG	15	$8000	ZZ	31	$80000000

Values in **Coord[*x*].FRAxes** are usually set using the buffered program command **`frax`**, but it is possible to set and clear bits of **Coord[*x*].FRAxes** directly.

Coord[x].FR2Axes

Function: Secondary feedrate-axis specification

Description: Array of "secondary vector feedrate" axis status bits

Range: $0 .. $FFFFFFFF

Units: Bit field

Coord[*x*].FR2Axes is a 32-bit status word containing the "secondary feedrate axis" status for each of the possible 32 axes in the coordinate system. A bit is 1 if the corresponding axis is used in the secondary vector feedrate calculations for a linear or circle mode move specified by feedrate, where the move time is calculated by dividing the vector distance for the move of all of these axes by the commanded feedrate. This time is compared to the time computed in the same manner for the primary vector feedrate axes, and the larger of the two times is used.

A bit is 0 if the corresponding axis not used in these secondary vector feedrate calculations. At power-on/reset, there are no secondary vector feedrate axes, so the initial value of **Coord[*x*].FR2Axes** is $0.

The following table shows for each axis name the bit number and the bit value in **Coord[*x*].FR2Axes**:

Axis Name	Bit #	Bit Value (Hex)	Axis Name	Bit #	Bit Value (Hex)
A	0	$1	HH	16	$10000
B	1	$2	LL	17	$20000
C	2	$4	MM	18	$40000
U	3	$8	NN	19	$80000
V	4	$10	OO	20	$100000
W	5	$20	PP	21	$200000
X	6	$40	QQ	22	$400000
Y	7	$80	RR	23	$800000
Z	8	$100	SS	24	$1000000
AA	9	$200	TT	25	$2000000
BB	10	$400	UU	26	$4000000
CC	11	$800	VV	27	$8000000
DD	12	$1000	WW	28	$10000000
EE	13	$2000	XX	29	$20000000
FF	14	$4000	YY	30	$40000000
GG	15	$8000	ZZ	31	$80000000

Values in **Coord[*x*].FR2Axes** are usually set using the buffered program command **`frax2`**, but it is possible to set and clear bits of **Coord[*x*].FR2Axes** directly.

Coord[*x*].FR2Axes is new in V2.1 firmware, released 1st quarter 2016.

Coord[x].HomeComplete

Description: "Position references established" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].HomeComplete** status bit is set to 1 if a position reference is properly established for all motors assigned to position axes in the coordinate system, either with a homing-search move, or an absolute position read. It is automatically set to 0 at power-up/reset, and at the beginning of a homing-search move for a motor. It is bit 15 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].HomeInProgress

Description: "Home search move in progress" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].HomeInProgress** status bit is set to 1 at the beginning of a homing search move for any motor in the coordinate system. It is set to 0 when the pre-specified trigger condition is found and the post-trigger move is started, or when the move ends without a trigger being found. It is bit 30 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].I2tFault

Description: "Integrated current (I^2T) fault" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].I2tFault** status bit is set to 1 when any motor in the coordinate system has been disabled from exceeding its integrated current limit as set by **Motor[*x*].I2tTrip**. The amplifier fault bit **Coord[*x*].AmpFault** will also be set in this case. It will be 0 at all other times, becoming 0 when the motor is re-enabled. (Note that if the amplifier faults due to its own integrated current fault calculations, this bit will not be set.) It is bit 21 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].IncAxes

Description: Array of "incremental mode" axis status bits

Range: $0 .. $FFFFFFFF

Units: Bit field

Coord[*x*].IncAxes is a 32-bit status word containing the incremental/absolute mode status for each of the possible 32 axes in the coordinate system. A bit is 1 if the corresponding axis is in incremental mode, and a move command value specifies the distance from the present commanded position for the axis. A bit is 0 if the corresponding axis is in absolute mode, and a move command value specifies the destination position relative to the programming origin for the axis.

The following table shows for each axis name the bit number and the bit value in **Coord[*x*].IncAxes**:

Axis Name	Bit #	Bit Value (Hex)	Axis Name	Bit #	Bit Value (Hex)
A	0	$1	HH	16	$10000
B	1	$2	LL	17	$20000
C	2	$4	MM	18	$40000
U	3	$8	NN	19	$80000
V	4	$10	OO	20	$100000
W	5	$20	PP	21	$200000
X	6	$40	QQ	22	$400000
Y	7	$80	RR	23	$800000
Z	8	$100	SS	24	$1000000
AA	9	$200	TT	25	$2000000
BB	10	$400	UU	26	$4000000
CC	11	$800	VV	27	$8000000
DD	12	$1000	WW	28	$10000000
EE	13	$2000	XX	29	$20000000
FF	14	$4000	YY	30	$40000000
GG	15	$8000	ZZ	31	$80000000

Values in **Coord[*x*].IncAxes** are usually set using the buffered program commands **`inc`** and **`abs`**, but it is possible to set and clear bits of **Coord[*x*].IncAxes** directly.

Coord[*x*].InPos

Description: "In-position" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].InPos** status bit is set to 1 when all of the conditions for "in position" are satisfied for every motor in the coordinate system: the motor is closed-loop, the desired velocity is zero, the move timer is not active (no move, dwell, or delay being executed, the magnitude of the following error is less than or equal to **Motor[*x*].InPosBand**, and all of these conditions have been true for (**Motor[*x*].InPosTime** – 1) consecutive servo cycles. It is 0 otherwise. It is bit 11 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].LHMotorSlots

Description: Number of motor columns needed in lookahead buffer

Range: 0 .. 256

Units: Motors

Coord[*x*].LHMotorSlots contains the number of Power PMAC motors that need a "column" in the lookahead buffer so lookahead calculations can operate on the motor. It is equal to the number of motors assigned to axes in this coordinate system. A motor counts toward this total if it has been directly assigned to one or more axes in an axis definition statement in the coordinate system (e.g. **#4->1000C**), if it has been defined as an inverse kinematic axis in the coordinate system (e.g. **#5->I**), or if it has been defined as a spindle in the coordinate system (e.g. **#4->S**). It does not count toward this total if it has been give the "null" definition in the coordinate system (e.g. **#6->0**).

Note that while a spindle axis does not need lookahead calculations done for it, a table slot is reserved for it so that the table does not have to be deleted and recreated when the motor is redefined as a positioning axis in the coordinate system.

Coord[*x*].LHSize

Description: Number of segments per motor in lookahead buffer

Range: Non-negative integer

Units: Move segments

Coord[*x*].LHSize contains the number of move segments reserved for each motor in the coordinate system's lookahead buffer. Its value is set by the value in the **define lookahead *{constant}*** command for the coordinate system. It is 0 if no lookahead buffer has been defined for the coordinate system. This value is distinct from the value of saved setup element **Coord[*x*].LHDistance**, which tells the coordinate system how many segments ahead of the executing point to keep stored in this buffer.

Note that the value of this element is stored to flash memory on a **save** command. However, it is not possible to write to it directly in the Script environment.

Coord[*x*].LHStatus

Description: Special lookahead buffer status byte

Range: 0 .. 255

Units: Bit field

The **Coord[*x*].LHStatus** status element combines 8 status bits concerning the operation of the special lookahead buffer into a single 8-bit element. Each of the individual bits is separately

accessible by its own name. The function of each bit is documented in the description of the individual bit. The eight individual status bits that form **Coord[*x*].LHStatus** are:

- Bit 0 (value 1): **Coord[*x*].LookAheadActive**
- Bit 1 (value 2): **Coord[*x*].LookAheadFlush**
- Bit 2 (value 4): **Coord[*x*].LookAheadReCalc**
- Bit 3 (value 8): **Coord[*x*].LookAheadChange**
- Bit 4 (value 16): **Coord[*x*].LookAheadStop**
- Bit 5 (value 32): **Coord[*x*].LookAheadDir**
- Bit 6 (value 64): **Coord[*x*].LookAheadLookBack**
- Bit 7 (value 128): **Coord[*x*].LookAheadWrap**

Coord[*x*].LHStatus forms bits 8 through 15 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].LimitStop

Description: "Stopped on hardware position limit" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LimitStop** status bit is 1 if any motor in the coordinate system has stopped or is stopping because it hit either its positive or negative hardware overtravel limit, even if it is presently not in that limit. It is 0 at all other times, including when into a limit, but moving out of it. It is bit 25 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].LinToPvtBuf

Description: "Linear-to-PVT move buffered" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LinToPvtBuf** status bit is 1 if a linear mode move that has been converted to a PVT move (because **Coord[*x*].SegLinToPvt** > 0) is presently buffered and awaiting execution. It is 0 otherwise. This bit is primarily for internal use. It is bit 30 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].LinToPvtError

Description: "Linear-to-PVT mode error" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LinToPvtError** status bit is set 1 if the motion program in the coordinate system has been aborted due to an error in executing a linear-to-PVT-mode move (in which case none of the other 7 bits of **Coord[*x*].ErrorStatus** will be set, and **Coord[*x*].ErrorStatus** will equal 4). If another of these bits is also set, then it simply indicates that bit 2 (value 4) of the error code is 1. Refer to a list of the error codes under **Coord[*x*].ErrorStatus**. It is 0 at all other times. It is bit 2 of 32-bit element **Coord[*x*].Status[0]**. It is bit 2 of 8-bit element **Coord[*x*].ErrorStatus**.

Coord[*x*].LookAheadActive

Description: "Lookahead buffer active" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LookAheadActive** status bit is set to 1 if the coordinate system is presently executing within the lookahead buffer. It is 0 otherwise. It is bit 8 of 32-bit element **Coord[*x*].Status[1]**. It is bit 0 of 8-bit element **Coord[*x*].LHStatus**.

Coord[*x*].LookAheadChange

Description: "Lookahead execution change" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LookAheadChange** status bit is set to 1 if the execution mode of the lookahead buffer is presently changing between any two of the modes of forward, reverse, and stopped. It is 0 otherwise. It is bit 11 of 32-bit element **Coord[*x*].Status[1]**. It is bit 3 of 8-bit element **Coord[*x*].LHStatus**.

Coord[*x*].LookAheadDir

Description: "Lookahead direction" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LookAheadDir** status bit is set to 1 if the lookahead buffer is presently executing in the reverse direction. It is 0 otherwise. It is bit 13 of 32-bit element **Coord[*x*].Status[1]**. It is bit 5 of 8-bit element **Coord[*x*].LHStatus**.

Coord[*x*].LookAheadFlush

Description: "Lookahead buffer wraparound" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LookAheadFlush** status bit is set to 1 if the lookahead buffer has reached the end of a sequence and is no longer adding new segments to the buffer (but is still executing existing segments). It is 0 otherwise. It is bit 9 of 32-bit element **Coord[*x*].Status[1]**. It is bit 1 of 8-bit element **Coord[*x*].LHStatus**.

Coord[*x*].LookAheadLookBack

Description: "Lookahead buffer look back" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LookAheadLookBack** status bit is set to 1 if the lookahead algorithm is presently calculating "backwards" in the buffer to ensure that a controlled stop is possible at the end of the most recently added move. It is 0 otherwise. It is bit 14 of 32-bit element **Coord[*x*].Status[1]**. It is bit 6 of 8-bit element **Coord[*x*].LHStatus**.

Coord[*x*].LookAheadReCalc

Description: "Lookahead buffer recalculation" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LookAheadReCalc** status bit is set to 1 if the lookahead calculations are working within the already existing buffer, as on reversal or "recovery" from reversal, rather than adding new segments to the buffer. It is 0 otherwise. It is bit 10 of 32-bit element **Coord[*x*].Status[1]**. It is bit 2 of 8-bit element **Coord[*x*].LHStatus**.

Coord[*x*].LookAheadStop

Description: "Lookahead stop" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LookAheadStop** status bit is set to 1 if the execution of the lookahead buffer is stopped or decelerating to a stop due to a "quick-stop" (`\, lh\`) command. It is 0 otherwise. It is bit 12 of 32-bit element **Coord[*x*].Status[1]**. It is bit 4 of 8-bit element **Coord[*x*].LHStatus**.

Coord[*x*].LookAheadWrap

Description: "Lookahead buffer wraparound" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].LookAheadWrap** status bit is set to 1 if the lookahead buffer has filled and "wrapped around" in the present continuous move sequence. This means that it is not possible to reverse through the entire sequence. It is 0 otherwise. It is bit 15 of 32-bit element **Coord[*x*].Status[1]**. It is bit 7 of 8-bit element **Coord[*x*].LHStatus**.

Coord[*x*].MinusLimit

Description: "Hardware negative limit set" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].MinusLimit** status bit is set to 1 when any motor in the coordinate system is presently in its negative hardware limit. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously. It is bit 29 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].Motors[*i*]

Description: Motor number assigned to axes in coordinate system

Range: 0 .. 255

Units: Motor numbers

Coord[*x*].Motors[*i*] contains the number of the "***i***th" motor assigned to a positioning axis in this coordinate system. **Coord[*x*].Motors[0]** contains the number of the lowest-numbered motor assigned to an axis in this coordinate system, **Coord[*x*].Motors[1]** contains the number of the next-lowest-numbered motor, and so on. These values assigned up until ***i*** = **Coord[*x*].NumMotors** – 1. These elements do not report motors with the "null" definition or a "spindle" definition in this coordinate system.

Coord[*x*].MoveMode

Description: "Programmed move mode" status element

Range: 0 .. 3

Units: Enumeration

Coord[*x*].MoveMode is a 2-bit status element indicating the type of move most recently calculated in the coordinate system. Its possible values and the move types specified are:

- 0: Blended (linear or circle mode move)
- 1: Spline mode move
- 2: PVT mode move
- 3: Rapid mode move

If **Coord[*x*].MoveMode** is 0, other status elements must be used to distinguish between the individual move modes. If the blended move mode is **`linear`**, **Coord[*x*].Omega0[0]** will be exactly equal to 0.0; if it is a circular mode, **Coord[*x*].Omega0[0]** will be a non-zero number. If in a circle mode, bit 4 (value 16) of **Coord[x].cdata** will be 0 for **`circle1`** (clockwise) mode; this bit will be 1 for **`circle2`** (counterclockwise) mode for the X/Y/Z Cartesian space. Bit 0 (value 1) reports in the same way for the XX/YY/ZZ Cartesian space.

Coord[*x*].MoveMode forms bits 22 and 23 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].Ncalc

Description: Synchronizing line label number of most recently calculated line

Range: 0 .. 2^{32} - 1

Units: Enumeration

Coord[*x*].Ncalc contains the value of the line label of the most recently *calculated* motion program line with a synchronizing line label (e.g. 560 if the most recent label were **`N560`**), providing an easy way of monitoring the execution of a motion program. If there is a move command on this line, it does not necessarily mean that this move has started execution; related status element **Coord[*x*].Nsync** can be used for that purpose.

Coord[*x*].Normal[*i*]

Description: Operating plane normal-vector component value

Range: floating-point

Units: 3D unit-vector component

Coord[*x*].Normal[*i*] contains the value of the specified dimensional component of the present unit vector normal to the plane of circular interpolation, 2D tool radius compensation, and corner

angle calculation for the coordinate system. Each coordinate system has two normal vectors, one in X/Y/Z-space, the other in XX/YY/ZZ-space, so there are six normal-vector components in a coordinate system.

The six components are:

- **Coord[*x*].Normal[0]**: I-component in X/Y/Z-space
- **Coord[*x*].Normal[1]**: J-component in X/Y/Z-space
- **Coord[*x*].Normal[2]**: K-component in X/Y/Z-space
- **Coord[*x*].Normal[3]**: II-component in XX/YY/ZZ-space
- **Coord[*x*].Normal[4]**: JJ-component in XX/YY/ZZ-space
- **Coord[*x*].Normal[5]**: KK-component in XX/YY/ZZ-space

The values of these elements are set by **`normal`** program commands. Note that even if the vector specified in the command is not of unit magnitude, the stored component values are those for a unit-magnitude vector in the same direction.

Coord[*x*].Nsync

Description: Synchronizing line label number of most recently calculated line

Range: 0 .. 2^{32} - 1

Units: Enumeration

Coord[*x*].Nsync contains the value of the line label most closely preceding the presently executing or most recently executed move in a motion program (e.g. 740 if the most recent label were **`N740`**), providing an easy way of monitoring the execution of the moves resulting from a motion program. Program calculation may have moved beyond this line; related status element **Coord[*x*].Ncalc** can be used to monitor the calculation point.

Coord[*x*].NumMotors

Description: Number of motors assigned to positioning axes in coordinate system

Range: 0 .. 256

Units: Motors

Coord[*x*].NumMotors contains the number of Power PMAC motors assigned to positioning axes in this coordinate system. A motor counts toward this total if it has been directly assigned to one or more axes in an axis definition statement in the coordinate system (e.g. **`#4->1000C`**), or if it has been defined as an inverse kinematic axis in the coordinate system (e.g. **`#5->I`**). It does not count toward this total if it has been given the "null" definition in the coordinate system (e.g. **`#6->0`**), or if it has been defined as a spindle in the coordinate system (e.g. **`#4->S`**).

Coord[*x*].NXYZ[*i*]

Description: 3D tool radius compensation surface-normal vector component value

Range: floating-point

Units: 3D unit-vector component

Coord[*x*].NXYZ[*i*] contains the value of the specified dimensional component of the present vector normal to the part surface in 3D tool (cutter) radius compensation in X/Y/Z-space. This vector has three components:

- **Coord[*x*].NXYZ[0]**: I-component in X/Y/Z-space
- **Coord[*x*].NXYZ[1]**: J-component in X/Y/Z-space
- **Coord[*x*].NXYZ[2]**: K-component in X/Y/Z-space

The values of these elements are set by **nxyz** program commands. Note that even if the vector specified in the command is not of unit magnitude, the stored component values are those for a unit-magnitude vector in the same direction.

Coord[*x*].PlusLimit

Description: "Hardware positive limit set" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].PlusLimit** status bit is set to 1 when any motor in the coordinate system is presently in its positive hardware limit. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously. It is bit 28 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].ProgActive

Description: Motion program active status

Range: 0 .. 1

Units: Boolean

Coord[*x*].ProgActive contains the present motion-program "activation" status for the coordinate system. It is 1 if a motion program is presently active. An active motion program can be running or suspended (as by a **q** [**pause**] or **s** [**step**] command) in a manner that execution could be resumed. The point of execution could be in a subprogram of the motion program. **Coord[*x*].ProgActive** is 0 if no motion program is active for the coordinate system. Refer to related status bits **Coord[*x*].ProgRunning** and **Coord[*x*].ProgProceeding** for slightly different information.

If a motion program is active, its buffer cannot be opened to clear it or download a new program. An **a** [**abort**] command de-activates a motion program, as does an error condition that automatically stops program execution.

This element is not part of the regularly computed status words for the coordinate system. It is computed from several of those bits "on-demand" through a function. As such, it is not directly accessible in the C environment; instead, it must be accessed through an API function call.

Coord[*x*].ProgProceeding

Description: Motion program advancing status

Range: 0 .. 1

Units: Boolean

Coord[*x*].ProgProceeding contains the present motion-program advancing execution status for the coordinate system. It is 1 if a motion program, or a subprogram called from it, is presently running and advancing toward further program calculations. A motion program can be "active" (**Coord[*x*].ProgActive** = 1) but not "proceeding" if its execution has been suspended (by a **q** [**pause**], **s** [**step**], **\ [lh**], or **h** [**hold**] command) in a manner that execution could be resumed.

Note that if motion has been halted by a **%0** time-base value (by a method other than a "hold"), the program is still technically proceeding and this element will have a value of 1.The difference between **Coord[*x*].ProgProceeding** and the similar **Coord[*x*].ProgRunning** is that **Coord[*x*[.ProgRunning** stays set to 1 during a "hold" and **Coord[*x*].ProgProceeding** is set to 0 when the program is suspended by a "hold".

Coord[*x*].ProgProceeding is 0 if no motion program is presently proceeding toward further calculation for the coordinate system, even if a program is "active" (**Coord[*x*].ProgActive** = 1).

This element is not part of the regularly computed status words for the coordinate system. It is computed from several of those bits "on-demand" through a function. As such, it is not directly accessible in the C environment; instead, it must be accessed through an API function call.

Coord[*x*].ProgRunning

Description: Motion program running status

Range: 0 .. 1

Units: Boolean

Coord[*x*].ProgRunning contains the present motion-program execution status for the coordinate system. It is 1 if a motion program, or a subprogram called from it, is presently running. A motion program can be "active" (**Coord[*x*].ProgActive** = 1) but not "running" if its execution has been

suspended (by a **q** [**pause**], **s** [**step**], or **** [**lh**] command) in a manner that execution could be resumed.

Note that if motion has been stopped by an **h** [**hold**] or **%0** command, the program is still technically running and this element will have a value of 1. The difference between **Coord[*x*].ProgRunning** and the similar **Coord[*x*].ProgProceeding** is that **Coord[*x*[.ProgRunning** stays set to 1 during a "hold" and **Coord[*x*].ProgProceeding** is set to 0 when the program is suspended by a "hold".

Coord[*x*].ProgRunning is 0 if no motion program is presently "running" for the coordinate system, even if a program is "active" (**Coord[*x*].ProgActive** = 1).

This element is not part of the regularly computed status words for the coordinate system. It is computed from several of those bits "on-demand" through a function. As such, it is not directly accessible in the C environment; instead, it must be accessed through an API function call.

Coord[*x*].PvtError

Description: "PVT mode error" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].PvtError** status bit is set 1 if the motion program in the coordinate system has been aborted due to an error in executing a PVT-mode move (in which case none of the other 7 bits of **Coord[*x*].ErrorStatus** will be set, and **Coord[*x*].ErrorStatus** will equal 8). If another of these bits is also set, then it simply indicates that bit 3 (value 8) of the error code is 1. Refer to a list of the error codes under **Coord[*x*].ErrorStatus**. It is 0 at all other times. It is bit 3 of 32-bit element **Coord[*x*].Status[0]**. It is bit 3 of 8-bit element **Coord[*x*].ErrorStatus**.

Coord[*x*].RadiusError

Description: "Circle radius error" status element

Range: 0 .. 3

Units: Bit field

Coord[*x*].RadiusError is a 2-bit status element indicating the presence of absence of circle move radius errors in the coordinate system. It also serves as two bits of the 8-bit status element **Coord[*x*].ErrorStatus**.

Bit 0 (value 1) is set to 1 if a circle move in the X/Y/Z axis space has a radius error exceeding that set by **Coord[*x*].RadiusErrorLimit** (in which case none of the other 7 bits of **Coord[*x*].ErrorStatus** will be set, and **Coord[*x*].ErrorStatus** will equal 64). If another of these bits is also set, then it simply indicates that bit 6 (value 64) of the error code is 1. Refer to a list of

the error codes under **Coord[*x*].ErrorStatus**. The bit is 0 if there is no error code with bit 6 set. It is bit 6 of 32-bit element **Coord[*x*].Status[0]**. It is bit 6 of 8-bit element **Coord[*x*].ErrorStatus**.

Bit 1 (value 2) is set to 1 if a circle move in the XX/YY/ZZ axis space has a radius error exceeding that set by **Coord[*x*].RadiusErrorLimit** (in which case none of the other 7 bits of **Coord[*x*].ErrorStatus** will be set, and **Coord[*x*].ErrorStatus** will equal 128). If another of these bits is also set, then it simply indicates that bit 7 (value 128) of the error code is 1. Refer to a list of the error codes under **Coord[*x*].ErrorStatus**. The bit is 0 if there is no error code with bit 7 set. It is bit 7 of 32-bit element **Coord[*x*].Status[0]**. It is bit 7 of 8-bit element **Coord[*x*].ErrorStatus**.

Coord[*x*].RotEnd

Description: Rotary buffer end address

Range: Power PMAC memory addresses

Units: Power PMAC byte addresses

Coord[*x*].RotEnd contains the memory address of the end of the defined rotary motion program buffer for the coordinate system. The difference between its value and that of **Coord[*x*].RotStart** is the size of the defined rotary buffer in bytes.

If no rotary motion program buffer is defined for the coordinate system, **Coord[*x*].RotEnd** will have a value of 0.

Coord[*x*].RotExec

Description: Rotary buffer execution address

Range: Power PMAC memory addresses

Units: Power PMAC byte addresses

Coord[*x*].RotExec contains the memory address of the next program command in the rotary motion program buffer for the coordinate system to be executed. The difference between the value of **Coord[*x*].RotStore** and its value (added to the size of the buffer if negative) is the size of the portion of the buffer with program commands that have not yet been executed. The remainder of the buffer is available for downloading new program lines.

If no rotary motion program buffer is defined for the coordinate system, **Coord[*x*].RotExec** will have a value of 0.

Coord[x].RotStart

Description: Rotary buffer start address

Range: Power PMAC memory addresses

Units: Power PMAC byte addresses

Coord[*x*].RotStart contains the memory address of the start of the defined rotary motion program buffer for the coordinate system. The difference between the value of **Coord[*x*].RotEnd** and its value is the size of the defined rotary buffer in bytes.

If no rotary motion program buffer is defined for the coordinate system, **Coord[*x*].RotStart** will have a value of 0.

Coord[x].RotStore

Description: Rotary buffer storage addess

Range: Power PMAC memory addresses

Units: Power PMAC byte addresses

Coord[*x*].RotStore contains the address of the next location in memory where a program line downloaded into the rotary motion program buffer for the coordinate system will be stored. The difference between its value and the value of **Coord[*x*].RotExec** (added to the size of the buffer if negative) is the size of the portion of the buffer with program commands that have not yet been executed. The remainder of the buffer is available for downloading new program lines.

If no rotary motion program buffer is defined for the coordinate system, **Coord[*x*].RotStore** will have a value of 0.

Coord[x].RunTimeError

Description: "Run-time error" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].RunTimeError** status bit is set to 1 if the motion program in the coordinate system has been aborted due to lack of calculation time (in which case none of the other 7 bits of **Coord[*x*].ErrorStatus** will be set, and **Coord[*x*].ErrorStatus** will equal 16). If another of these bits is also set, then it simply indicates that bit 4 (value 16) of the error code is 1. Refer to a list of the error codes under **Coord[*x*].ErrorStatus**. It is 0 at all other times. It is bit 4 of 32-bit element **Coord[*x*].Status[0]**. It is bit 4 of 8-bit element **Coord[*x*].ErrorStatus**.

Coord[x].SegEnabled

Description: "Segmented move executing" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].SegEnabled** status bit is set to 1 if the coordinate system is presently executing (or has most recently executed) a segmented move (linear, circle, or pvt move with **Coord[*x*].SegMoveTime** > 0). It is 0 otherwise. It is bit 17 of 32-bit element **Coord[*x*].Status[1]**.

Coord[x].SegMove

Description: "Segmented move in process" status element

Range: 0 .. 3

Units: Bit field

Coord[*x*].SegMove is a 2-bit status element indicating the state of execution of segmented moves in the coordinate system.

Bit 1 (value 2) is set to 1 if the coordinate system is presently executing a PVT move in segmentation mode. It is 0 otherwise. This bit is primarily for internal use. It is bit 21 of 32-bit element **Coord[*x*].Status[1]**.

Bit 0 (value 1) is set to 1 if the coordinate system is presently executing a linear or circle move in segmentation mode. It is 0 otherwise. This bit is primarily for internal use. It is bit 20 of 32-bit element **Coord[*x*].Status[1]**.

Coord[x].SegMoveAccel

Description: "First move segment executing" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].SegMoveAccel** status bit is set to 1 if the coordinate system is presently executing the initial (added) segment from a stop in a segmented move or move sequence. It is 0 otherwise. This bit is primarily for internal use. It is bit 19 of 32-bit element **Coord[*x*].Status[1]**.

Coord[x].SegMoveDecel

Description: "Last move segment executing" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].SegMoveDecel** status bit is set to 1 if the coordinate system is presently executing the final (added) segment to a stop in a segmented move or move sequence. It is 0 otherwise. This bit is primarily for internal use. It is bit 18 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].SegStopReq

Description: "Segmented move stop request" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].SegStopReq** status bit is set to 1 if the coordinate system has been requested to stop during a segmented move sequence. It is 0 otherwise. This bit is primarily for internal use. It is bit 16 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].SharpCornerStop

Description: "Blending disabled for sharp corner" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].SharpCornerStop** status bit is set to 1 if blending has been disabled at a corner because the angle is sharper than that specified by **Coord[*x*].CornerBlendBp**. It is 0 otherwise. It is bit 1 of 32-bit element **Coord[*x*].Status[1]**.

Coord[*x*].SoftLimit

Description: "Stopped on software limit" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].SoftLimit** status bit is set to 1 if coordinated program motion in the coordinate system has stopped because a motor in the coordinate system has stopped due to reaching either its positive or negative software overtravel limit (in which case none of the other 7 bits of **Coord[*x*].ErrorStatus** will be set, and **Coord[*x*].ErrorStatus** will be equal to 32). If another of these bits is also set, then it simply indicates that bit 5 (value 32) of the error code is 1. Refer to a list of the error codes under **Coord[*x*].ErrorStatus**. It is 0 at all other times, including when into a limit, but moving out of it. It is bit 5 of 32-bit element **Coord[*x*].Status[0]**. It is bit 5 of 8-bit element **Coord[*x*].ErrorStatus**.

The **Coord[*x*].SoftLimit** status will not be true if a motor's commanded position has simply "saturated" on one of its software overtravel limits (with saved setup element **Coord[*x*].SoftLimitDis** set to 1), or if an independent motor move, such as a jog, of a motor in the coordinate system has stopped due to reaching one of its software overtravel limits.

Coord[*x*].SoftMinusLimit

Description: "Software negative limit set" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].SoftMinusLimit** status bit is set to 1 when any motor in the coordinate system has reached or exceeded its negative software limit as set by (**Motor[*x*].MinPos** – **Motor[*x*].SoftLimitOffset**). (**Motor[*x*].MinPos** must be less than **Motor[*x*].MaxPos** for this limiting to be active). It is 0 otherwise, even if the motor is still stopped from having hit this limit previously. It is bit 23 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].SoftPlusLimit

Description: "Software positive limit set" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].SoftPlusLimit** status bit is set to 1 when any motor in the coordinate system has reached or exceeded its positive software limit as set by (**Motor[*x*].MaxPos + Motor[*x*].SoftLimit Offset**). (**Motor[*x*].MaxPos** must be greater than **Motor[*x*].MinPos** for this limiting to be active). It is 0 otherwise, even if the motor is still stopped from having hit this limit previously. It is bit 22 of 32-bit element **Coord[*x*].Status[0]**.

Coord[x].Status[0]

Description: Coordinate system first status word

Range: $0 .. $FFFFFFFF

Units: Bit field

Coord[*x*].Status[0] is the first 32-bit status word for the coordinate system, containing many individual global status bits. The following table provides a list of these status bits in the word:

Bit #	Hex Value	Element Name: Coord[*x*].*{element}*	Description
31	$80000000	**TriggerMove**	Trigger search move in progress (any motor in C.S.)
30	$40000000	**HomeInProgress**	Home search move in progress (any motor in C.S.)
29	$20000000	**MinusLimit**	Hardware negative limit set (any motor in C.S.)
28	$10000000	**PlusLimit**	Hardware positive limit set (any motor in C.S.)
27	$8000000	**FeWarn**	Warning following error (any motor in C.S.)
26	$4000000	**FeFatal**	Fatal following error (any motor in C.S.)
25	$2000000	**LimitStop**	Stopped on hardware limit (any motor in C.S.)
24	$1000000	**AmpFault**	Amplifier fault (any motor in C.S.)
23	$800000	**SoftMinusLimit**	Software negative limit set (any motor in C.S.)
22	$400000	**SoftPlusLimit**	Software positive limit set (any motor in C.S.)
21	$200000	**I2tFault**	Integrated current (I^2T) fault (any motor in C.S.)
20	$100000	**TriggerNotFound**	Trigger not found (any motor in C.S.)
19	$80000	**AmpWarn**	Amp warning (any motor in C.S.)
18	$40000	**EncLoss**	Sensor loss error (any motor in C.S.)
17	$20000	**AuxFault**	Auxiliary fault error (any motor in C.S.)
16	$10000	**TimerEnabled**	Move timer enabled
15	$8000	**HomeComplete**	Home complete (all motors in C.S.)
14	$4000	**DesVelZero**	Desired velocity zero (all motors in C.S.)
13	$2000	**ClosedLoop**	Closed-loop mode (all motors in C.S.)
12	$1000	**AmpEna**	Amplifier enabled (all motors in C.S.)
11	$800	**InPos**	In position (all motors in C.S.)
10	$400	-	(*Reserved for future use*)
9	$200	**BlockRequest**	Block request flag set
8	$100	**TimersEnabled**	Timers enabled
7	$80	**RadiusError (bit 1)** **ErrorStatus (bit 7)**	XX/YY/ZZ-axis circle radius error / Error word bit 7
6	$40	**RadiusError (bit 0)** **ErrorStatus (bit 6)**	X/Y/Z-axis circle radius error / Error word bit 6
5	$20	**SoftLimit** **ErrorStatus (bit 5)**	Stopped on software position limit (any motor in C.S.) / Error word bit 5
4	$10	**RunTimeError** **ErrorStatus (bit 4)**	Run time error / Error word bit 4
3	$8	**PvtError** **ErrorStatus (bit 3)**	PVT mode error / Error word bit 3
2	$4	**LinToPvtError** **ErrorStatus (bit 2)**	Linear-to-PVT mode error / Error word bit 2
1	$2	**ErrorStatus (bit 1)**	Buffer error / Error word bit 1
0	$1	**ErrorStatus (bit 0)**	Sync assignment buffer error / Error word bit 0

The value of this element is reported in response to the **&x?** query command. Its contents form the first eight hexadecimal digits (four bits per digit) of the response. The on-line query command **backup Coord[*x*].Status** causes Power PMAC to report the values of all of the individual elements as English text.

The low 8 bits (bits 0 – 7) of **Coord[*x*].Status[0]** form the partial-word status element **Coord[*x*].ErrorStatus**. If only one of these bits is set, that bit can be considered an individual error status bit (e.g. **Coord[*x*].RadiusError**). However, if more than one bit is set, the entire 8 bits must be considered an error code. The most general approach is to always consider this section as an 8-bit error code.

Each element is described in more detail in its own individual specification.

Coord[*x*].Status[1]

Description: Coordinate system second status word

Range: $0 .. $FFFFFFFF

Units: Bit field

Coord[*x*].Status[1] is the second 32-bit status word for the coordinate system, containing many individual global status bits. The following table provides a list of these status bits in the word:

Bit #	Hex Value	Element Name: Coord[*x*].*{element}*	Description
31	$80000000	**Csolve**	Valid coordinate system axis definition for PMATCH
30	$40000000	**LinToPvtBuf**	Linear-to-PVT move buffered
29	$20000000	**FeedHold (bit 1)**	Feed hold accel/decel
28	$10000000	**FeedHold (bit 0)**	Feed hold time base source
27	$8000000	**BlockActive**	Block active
26	$4000000	**ContMotion**	Continuous motion request
25	$2000000	**CCMode (bit 1)**	Cutter comp mode bit 1
24	$1000000	**CCMode (bit 0)**	Cutter comp mode bit 0
23	$800000	**MoveMode (bit 1)**	Move mode bit 1 (=0 for blended and spline, 1 for rapid and pvt modes)
22	$400000	**MoveMode (bit 0)**	Move mode bit 0 (=0 for blended and pvt, 1 for rapid and spline modes)
21	$200000	**SegMove (bit 1)**	Segmented PVT-mode move in progress
20	$100000	**SegMove (bit 0)**	Segmented linear-mode move in progress
19	$80000	**SegMoveAccel**	First segment move
18	$40000	**SegMoveDecel**	Last segment move
17	$20000	**SegEnabled**	Segmentation enabled
16	$10000	**SegStopReq**	Segment stop request
15	$8000	**LookAheadWrap LHStatus (bit 7)**	Lookahead wrap
14	$4000	**LookAheadLookBack LHStatus (bit 6)**	Lookahead lookback
13	$2000	**LookAheadDir LHStatus (bit 5)**	Lookahead direction
12	$1000	**LookAheadStop**	Lookahead stop

		LHStatus (bit 4)	
11	$800	**LookAheadChange LHStatus (bit 3)**	Lookahead change
10	$400	**LookAheadReCalc LHStatus (bit 2)**	Lookahead recalculation
9	$200	**LookAheadFlush LHStatus (bit 1)**	Lookahead flush
8	$100	**LookAheadActive LHStatus (bit 0)**	Lookahead active
7	$80	**CCAddedArc**	Cutter comp added arc
6	$40	**CCOffReq**	Cutter comp turn-off request
5	$20	**CCMoveType (bit 1)**	Cutter comp move type bit 1
4	$10	**CCMoveType (bit 0)**	Cutter comp move type bit 0
3	$8	**EndDelayActive**	Automatically added end delay in progress
2	$4	**CC3Active**	3D cutter comp active
1	$2	**SharpCornerStop**	Blending disabled for sharp corner
0	$1	**AddedDwellDis**	Added dwell disable

The value of this element is reported in response to the **&x?** query command. Its contents form the second eight hexadecimal digits (four bits per digit) of the response. The on-line query command **backup Coord[*x*].Status** causes Power PMAC to report the values of all of the individual elements as English text.

Each element is described in more detail in its own individual specification.

Coord[*x*].T0Spline

Description: Spline-mode move first-section time

Range: Positive floating-point

Units: Milliseconds

Coord[*x*].T0Spline contains the time Power PMAC is using for the first of three sections when calculating spline-mode moves. The value is set by a program **spline*{data}*** command. Related status elements **Coord[*x*].T1Spline** and **Coord[*x*].T2Spline** contain the times for the second and third sections of these moves, respectively.

If the coordinate system is in PVT mode, **Coord[*x*].T0Spline** contains the most recent specified time for a PVT move, as set by a program **pvt*{data}*** command.

Coord[*x*].T1Spline

Description: Spline-mode move second-section time

Range: Positive floating-point

Units: Milliseconds

Coord[*x*].T1Spline contains the time Power PMAC is using for the second of three sections when calculating spline-mode moves. The value is set by a program **spline*{data}*** command. Related status elements **Coord[*x*].T0Spline** and **Coord[*x*].T2Spline** contain the times for the first and third sections of these moves, respectively.

Coord[*x*].T2Spline

Description: Spline-mode move third-section time

Range: Positive floating-point

Units: Milliseconds

Coord[*x*].T2Spline contains the time Power PMAC is using for the third of three sections when calculating spline-mode moves. The value is set by a program **spline*{data}*** command. Related status elements **Coord[*x*].T0Spline** and **Coord[*x*].T1Spline** contain the times for the first and second sections of these moves, respectively.

Coord[*x*].TimeBase

Description: Present "time elapsed in servo cycle" for interpolation calculations

Range: Floating-point

Units: milliseconds

Coord[*x*].TimeBase contains the "time elapsed" (Δt) value used by the coordinate system's trajectory interpolation calculations in the most recent servo cycle. That is, in the latest servo cycle, the interpolation algorithms incremented the time by the value in this element. Note that this value does not have to match the physical time elapsed in the servo cycle; that time is fixed for an application. Varying the value in **Coord[*x*].TimeBase** from that for the physical time elapsed permits "time base control" techniques such as "feedrate override".

The "target" value for the time base comes from the register whose address is specified by **Coord[*x*].pDesTimeBase**. If the value in this register is different from that in **Coord[*x*].TimeBase**, the value in **Coord[*x*].TimeBase** is incremented each servo cycle by the amount in saved setup element **Coord[*x*].TimeBaseSlew** until the target value is reached.

It is possible to write directly to **Coord[*x*].TimeBase**, bypassing slew rate limiting (which always imposes at least a one servo-cycle delay). In this case, it is usually desirable to set **TimeBaseSlew** to 0, so the automatic slewing algorithms do not try to undo a direct setting.

There is full support for negative time base values, which permit reverse execution of a trajectory through the stored "trace" buffers.

Coord[*x*].TimerEnabled

Description: "Move timer enabled" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].TimerEnabled** status bit is set to 1 if any motor in the coordinate system is performing a move or move section of definite time, whether from a programmed axis move or an independent motor jogging or homing move. It is 0 otherwise. It is the logical OR of the **Motor[*x*].Desired.TimerEnabled** status bits for all motors assigned to position axes in the coordinate system. It is bit 16 of 32-bit element **Coord[*x*].Status[0]**.

In an "indefinite" jogging move (`j+` or `j-`) or the pre-trigger portion of a homing-search move, the constant-speed portion of the move is not a section of definite time. However, if software limits for the motor are active (**Motor[*x*].MaxPos** > **Motor[*x*].MinPos**), the constant-speed sections of jogging moves are automatically converted to sections of definite time.

Coord[*x*].TimersEnabled

Description: "Move timers enabled" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].TimersEnabled** status bit is set to 1 if every motor assigned to a positioning axis in the coordinate system is performing a move or move section of definite time. It is 0 otherwise. It is the logical AND of the **Motor[*x*].Desired.TimerEnabled** status bits for all motors assigned to position axes in the coordinate system. It is bit 8 of 32-bit element **Coord[*x*].Status[0]**.

When the coordinate system is executing a commanded move in a motion program, all motors assigned to positioning axes in the coordinate system are performing moves of definite time, even if not explicitly commanded by the move.

Coord[*x*].Tsel

Description: Presently selected transformation matrix for coordinate system

Range: -1 .. 255

Units: Matrix index

Coord[*x*].Tsel contains the value of the index of the presently selected axis transformation matrix for the coordinate system. (A value of -1 indicates that no transformation matrix has been selected.) When it contains a value ***i*** in the range 0 .. 255, the **Tdata[*i*]** data structure elements comprising the transformation matrix with this index are used to transform the programmed axis positions.

The value of **Coord[*x*].Tsel** is set using the buffered program command **`tsel{data}`**. It is not possible to write to this element directly.

Coord[*x*].TriggerMove

Description: "Trigger search move in progress" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].TriggerMove** status bit is set to 1 at the beginning of a move-until-trigger (homing search, jog-until-trigger, program rapid-mode move-until-trigger) for any motor in the coordinate system. It is set to 0 when the pre-specified trigger condition is found and the post-trigger move is started, or when the move ends without a trigger being found. It is bit 31 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].TriggerNotFound

Description: "Trigger not found" status bit

Range: 0 .. 1

Units: Boolean

The **Coord[*x*].TriggerNotFound** status bit is set to 1 if, for any motor in the coordinate system a jog-until-trigger or program rapid-mode move-until-trigger ends without the pre-specified trigger condition being found. This is not an error condition, but subsequent actions will often depend on whether this bit is set or not. It is 0 at all other times, changing back to 0 when the next move is started. It is bit 20 of 32-bit element **Coord[*x*].Status[0]**.

Coord[*x*].TXYZ[*j*]

Description: 3D tool radius compensation tool-orientation vector component value

Range: floating-point

Units: 3D unit-vector component

Coord[*x*].TXYZ[*j*] contains the value of the specified dimensional component of the present tool-orientation vector in 3D tool (cutter) radius compensation in X/Y/Z-space. This vector has three components:

- **Coord[*x*].TXYZ[0]**: I-component in X/Y/Z-space
- **Coord[*x*].TXYZ[1]**: J-component in X/Y/Z-space
- **Coord[*x*].TXYZ[2]**: K-component in X/Y/Z-space

The values of these elements are set by **txyz** program commands. Note that even if the vector specified in the command is not of unit magnitude, the stored component values are those for a unit-magnitude vector in the same direction.

Coord[*x*].TxyzScale

Description: Transformation matrix rescaling compensation

Range: Floating-point

Units: none (ratio)

Coord[*x*].TxyzScale permits the scaling of vector feedrate and 2D tool compensation radius to be different from the transformation matrix scaling of the XYZ Cartesian space in the coordinate system. Its primary use is to maintain the feedrate and tool radius in the "base" (untransformed) units of the X, Y, and Z axes even when the programmed units of these axes have been rescaled with the selected transformation matrix.

Coord[*x*].TxyzScale is only used when a transformation matrix has been selected for the coordinate system with a program **tsel*{data}*** command (so status element **Coord[*x*].Tsel** > -1) and **TxyzScale** is greater than its power-on default value of 0.0.

If saved setup element **Coord[*x*].AutoTxyzScale** is set to 1, Power PMAC will automatically compute **Coord[*x*].TxyzScale** to maintain feedrate and compensation radius in the base units of the coordinate system, even when rescaling has been done through the matrix. (The X, Y, and Z axes must all be rescaled by the same factor for this to work properly.)

Alternatively, the value of this element can be set directly with the buffered program command **txyzscale*{data}***, where ***{data}*** specifies the value to be written to this element.

When this feature is active, the value of the vector feedrate velocity as specified by the program **F*{data}*** command, which would use position units of the transformed axes without this feature, is divided by the value of **Coord[*x*].TxyzScale** before the move is calculated. Similarly, the value of the 2D tool compensation radius as specified by the program **ccr*{data}*** command, which would use position units of the transformed axes without this feature, is divided by the value of **Coord[*x*].TxyzScale** before the compensation offset is calculated.

To maintain vector feedrate and compensation radius in the base units of the X, Y, and Z axes, **Coord[*x*].TxyzScale** should be set to the relative magnitude of the transformed axis units to the base axis units. So if the transformed units are 3 times as big as the base units, **TxyzScale** should be set to 3.0. If the transformed units are half as big as the base units, **TxyzScale** should be set to 0.5.

Coord[*x*].XYZoff[*j*]

Description: 3D tool radius compensation offset vector component value

Range: floating-point

Units: Cartesian axis units

Coord[*x*].XYZoff[*j*] contains the value of the specified dimensional component of the most recently calculated offset vector in 3D tool (cutter) radius compensation in X/Y/Z-space. This vector has three components:

- **Coord[*x*].XYZoff[0]**: X-component
- **Coord[*x*].XYZoff[1]**: Y-component
- **Coord[*x*].XYZoff[2]**: Z-component

The values of these elements are set by the 3D tool radius compensation algorithm based on the declared surface-normal vector, the declared tool-orientation vector, and the tool-tip geometry table. The resulting vector defined by these components is the difference between the programmed point and the commanded "tool center" point for the move.

Coord[x].Ldata Local Data Status Elements

The **Coord[*x*].Ldata** substructure contains elements specifying generalized local data for the coordinate system. This substructure is identical to the **Ldata** substructure for each Script PLC program and each communications thread. Refer to the description of the **Ldata** structure below.

Coord[*x*].CC3Data[*i*]. 3D Cutter Compensation Status Elements

The **Coord[*x*].CC3Data[*i*].** substructure contains elements specifying the tool-nose geometry for 3D cutter (tool) radius compensation. Some of these are calculated automatically by Power PMAC and are read-only status elements in the Script environment.

Coord[*x*].CC3Data[*i*].ToolOffset

Description: 3D cutter compensation segment axial offset

Range: Floating-point

Units: Linear axis user units

Power-on default: 0.0

Coord[*x*]. CC3Data[*i*].ToolOffset contains, for the tool-tip geometry to be used for three-dimensional cutter radius compensation in Coordinate System ***x***, the signed axial distance from the tool center point to the starting point of the "*i*th" defined arc section of the tool tip.

The starting point for the entire tool-tip table, defined by non-saved setup elements directly specified by the user: **Coord[*x*].CC3Data[0].ToolOffset** and **Coord[*x*].CC3Data[0].ToolRadius**, is the reference point for the other sections of the tool-tip table. The comparable elements for subsequent arc sections ($i > 0$) are calculated automatically by PMAC from the user settings of **Coord[*x*].CC3Data[*i*].CutRadius** and **Coord[*x*].CC3Data[*i*].NdotT**, and these elements are read-only status elements for the user.

Coord[*x*]. CC3Data[*i*].ToolOffset is expressed in the units of the base X, Y, and Z axes for the coordinate system, almost always millimeters or inches. It is not rescaled if the axis units are rescaled through use of a transformation matrix. If it is a positive value, the reference point for the table is "above" the tool center point (farther away from the tool tip than the center point). If it is a negative value, the reference point is "below" the tool center point (closer to the tool tip than the center point). In 3D cutter radius compensation, the motion program path is written as if the tool center point were in contact with the surface.

For more details, refer to the section *Three-Dimensional Cutter Radius Compensation* in the chapter *Power PMAC Move Mode Trajectories* in the Power PMAC User's Manual.

Coord[*x*].CC3Data[*i*].ToolRadius

Description: 3D cutter compensation segment shaft radius

Range: Positive floating-point

Units: Linear axis user units

Power-on default: 0.0

Coord[*x*].CC3Data[*i*].ToolRadius contains, for the tool-tip geometry to be used for three-dimensional cutter radius compensation in Coordinate System ***x***, the perpendicular distance from the tool centerline (axis of rotation) to the starting point of the "*i*th" defined arc section of the tool tip.

The starting point for the entire tool-tip table, defined by non-saved setup elements directly specified by the user: **Coord[*x*].CC3Data[0].ToolOffset** and **Coord[*x*].CC3Data[0].ToolRadius**, is the reference point for the other sections of the tool-tip table. The comparable elements for subsequent arc sections ($i > 0$) are calculated automatically by PMAC from the user settings of **Coord[*x*].CC3Data[*i*].CutRadius** and **Coord[*x*].CC3Data[*i*].NdotT**, and these elements are read-only status elements for the user.

Coord[*x*].CC3Data[0].ToolRadius is expressed in the units of the base X, Y, and Z axes for the coordinate system, almost always millimeters or inches. It is not rescaled if the axis units are rescaled through use of a transformation matrix.

For more details, refer to the section *Three-Dimensional Cutter Radius Compensation* in the chapter *Power PMAC Move Mode Trajectories* in the Power PMAC User's Manual.

Coord[*x*].TPData[*i*]. Target Position Buffer Status Elements

The **Coord[*x*].TPData[*i*].** substructure contains elements holding the target position data for programmed moves in the coordinate system between move calculation time and move execution time. This information is held to support the reporting of axis target positions and "distance to go" of the currently executing move, particularly useful in CNC-style applications.

Coord[*x*].TPData[*i*].Ncalc

Description: Target position buffer move synchronizing label value

Range: Non-negative integer

Units: Enumeration

Power-on default: 0

Coord[*x*].TPData[*i*].Ncalc contains, for the "*i*th" programmed move in the target position buffer for Coordinate System ***x***, the value of the synchronizing label (if any) for that move. If there was no synchronizing label for that move, the value of this element will be 0.

Move index values *i* range from 0 to **Coord[*x*].TPSize** – 1. The substructure with *i* = 0 is for the presently executing move. The substructure with *i* = 1 is for the next move in the sequence, and so on.

If saved setup element **Coord[*x*].TPSize** is set to its default value of 0, no target position buffer is used, and the value of this element will report as "nan" (not a number).

Coord[*x*].TPData[*i*].Pos[*j*]

Description: Target position buffer move axis destination

Range: Floating-point

Units: Axis position units

Power-on default: 0.0

Coord[*x*].TPData[*i*].Pos[*j*] contains, for the "*i*th" programmed move in the target position buffer for Coordinate System ***x***, the move destination position for the axis with index "*j*" for that move. Axis index values *j* range from 0 to 31, with the axis name for each index value shown in the following table:

Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*
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

Position values do *not* include and cutter compensation offsets. Use **Coord[*x*].TPData[*i*].XYZPos[*j*]** elements if you want positions with these offsets included.

Move index values *i* range from 0 to **Coord[*x*].TPSize** – 1. The substructure with *i* = 0 is for the presently executing move. The substructure with *i* = 1 is for the next move in the sequence, and so on.

If saved setup element **Coord[*x*].TPSize** is set to its default value of 0, no target position buffer is used, and the value of this element will report as "nan" (not a number).

Typically, these values are not accessed directly by the user. They are used to store these coordinates from move calculation time to move execution time, when they are used by Power PMAC to calculate the values reported for queries of move target position and "distance to go".

Coord[*x*].TPData[*i*].XYZPos[*j*]

Description: Target position buffer move axis position with cutter compensation offset

Range: Floating-point

Units: Axis position units

Power-on default: 0.0

Coord[*x*].TPData[*i*].XYZPos[*j*] contains, for the "*i*th" programmed move in the target position buffer for Coordinate System ***x***, the move destination position including cutter compensation offset for the axis with index "*j*" for that move. Axis index values *j* range from 0 to 2, with *j* = 0 representing the X-axis offset, *j* = 1 representing the Y-axis offset, and *j* = 2 representing the Z-axis offset.

Move index values *i* range from 0 to **Coord[*x*].TPSize** – 1. The substructure with *i* = 0 is for the presently executing move. The substructure with *i* = 1 is for the next move in the sequence, and so on.

If saved setup element **Coord[*x*].TPSize** is set to its default value of 0, no target position buffer is used, and the value of this element will report as "nan" (not a number).

Typically, these values are not accessed directly by the user. They are used to store these coordinates from move calculation time to move execution time, when they are used by Power PMAC to calculate the values reported for queries of move target position and "distance to go".

Coord[*x*].TPExec. Target Position Status Elements

The **Coord[*x*].TPExec.** substructure contains elements holding the target position data for the presently executing programmed move in the coordinate system, having been buffered from move calculation time to move execution time. This information is held to support the reporting of axis target positions and "distance to go" of the currently executing move, particularly useful in CNC-style applications.

Coord[*x*].TPExec.Ncalc

Description: Presently executing move synchronizing label value

Range: Non-negative integer

Units: Enumeration

Power-on default: 0

Coord[*x*].TPExec.Ncalc contains, for the presently executing programmed move from the target position buffer for Coordinate System ***x***, the value of the synchronizing label (if any) for that move. If there was no synchronizing label for that move, the value of this element will be 0.

If saved setup element **Coord[*x*].TPSize** is set to its default value of 0, no target position buffer is used, and the value of this element will report as "nan" (not a number).

Coord[*x*].TPExec.Pos[*j*]

Description: Presently executing move axis destination

Range: Non-negative integer

Units: Axis position units

Power-on default: 0.0

Coord[*x*].TPExec.Pos[*j*] contains, for the presently executing programmed move from the target position buffer for Coordinate System ***x***, the move destination position for the axis with index "*j*" for that move. Axis index values *j* range from 0 to 31, with the axis name for each index value shown in the following table:

Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*	Axis	*j*
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

Position values do *not* include and cutter compensation offsets. Use **Coord[*x*].TPExec.XYZPos[*j*]** elements if you want positions with these offsets included.

If saved setup element **Coord[*x*].TPSize** is set to its default value of 0, no target position buffer is used, and the value of this element will report as "nan" (not a number).

These values are used by Power PMAC to calculate the values reported for queries of move target position and "distance to go".

Coord[*x*].TPExec.XYZPos[*j*]

Description: Presently executing move axis position with cutter compensation offset

Range: Non-negative integer

Units: Axis position units

Power-on default: 0.0

Coord[*x*].TPExec.XYZPos[*j*] contains, for the presently executing programmed move from the target position buffer for Coordinate System ***x***, the move destination position including cutter compensation offset for the axis with index "*j*" for that move. Axis index values *j* range from 0 to 2, with *j* = 0 representing the X-axis offset, *j* = 1 representing the Y-axis offset, and *j* = 2 representing the Z-axis offset.

If saved setup element **Coord[*x*].TPSize** is set to its default value of 0, no target position buffer is used, and the value of this element will report as "nan" (not a number).

These values are used by Power PMAC to calculate the values reported for queries of move target position and "distance to go".

ECAT[*i*]. Status Data Structure Elements

The following structures provide status details for the EtherCAT network.

ECAT[*i*].DCClockDiff

Description Time difference between Power PMAC clock and distributed clock reference

Range: Integers

Units: Nanoseconds

ECAT[*i*].DCClockDiff contains the time difference between the Power PMAC's internal clock and the distribute clock reference from the selected slave device. If the magnitude of this difference exceeds that specified by saved setup element **ECAT[*i*].DCRefBand**, Power PMAC's internal phase clock frequency will be adjusted by the amount specified in **ECAT[*i*].DCRefPlus** or **ECAT[*i*].DCRefMinus**, depending on the direction, to reduce this difference.

ECAT[*i*].Error

Description: EtherCAT network enabling error code

Range: -99 .. 0

Units: Enumeration

ECAT[*i*].Error contains the error code from the most recent attempt to enable the EtherCAT network with index ***i*** from the command to set **ECAT[*i*].Enable** to 1. If **ECAT[*i*].Error** is 0, the attempt was successful and there was no error. If **ECAT[*i*].Error** has a value less than 0, the attempt was unsuccessful, and the value represents the specific error. The error codes presently implemented are:

- -1: EtherCAT master application library not present.
- -2: EtherCAT master kernel library not present.
- -3: Could not access master. EtherCAT adapter may not be properly configured.
- -8: Improperly configured slave device is enabled (**ECAT[*i*].Slave[*j*].Enable** = 1 but other slave settings are incorrect).
- -9: Invalid PDO request in the configuration (**ECAT[*i*].IO[*k*]** is not properly configured).
- -10 Invalid PDI configuration. **ECAT[*i*].LPIO[*k*]** is not properly configured.
- **ECAT[i].LPIO[k].Slave** is assigned to an **ECAT[*i*].Slave** that does not have a matching **ECAT[*i*].LPIO[*k*].Index** for **ECAT[*i*].Slave[*j*].PDO[*k*].Index**.
- -11: Could not start the EtherCAT master.
- -12: Slave device specified for I/O (**ECAT[*i*].IO[*k*].Slave**) not enabled.
- -13: **ECAT[*i*].BGIO[*j*].Slave** is not enabled.

- -14: EtherCAT is not properly licensed. Do not exceed purchased EtherCAT axis count.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LinkUp

Description: Master connection status

Range: 0 .. 1

Units: Boolean

ECAT[*i*].LinkUp indicates whether the EtherCAT master of index ***i*** is connected to a device. It is only valid if **ECAT[*i*].RTStateCheck** is set to 1. The values are as follows:

- 0: Not connected to a device
- 1: Connected to a device

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPDomainOutputState

Description: Present state of EtherCAT background process output exchanges

Range: 0 .. 3

Units: Enumeration

ECAT[*i*].LPDomainOutputState contains a value representing the present state of the background (low-priority) process data output transfers for the EtherCAT network with index ***i***. The value of this element is set only if **ECAT[*i*].LPStateCheck** is set to 1 to enable automatic state checking of the background transfers. The state values can be:

- 0: No registered process data were exchanged
- 1: Some registered process data were exchanged
- 2: (reserved)
- 3: All registered process data were exchanged

If **ECAT[*i*].LPnotLRW** is set to the default value of 0, common input/output exchanges are enabled, and the common state for both input and output exchanges is reported in **ECAT[*i*].LPDomainState** and this element is not used. If **ECAT[*i*].LPnotLRW** is set to 1 to disable common input/output exchanges, the separate state for output exchanges is reported here in **ECAT[*i*].LPDomainOutputState**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPDomainState

Description: Present state of EtherCAT background process I/O exchanges

Range: 0 .. 3

Units: Enumeration

ECAT[*i*].LPDomainState contains a value representing the present state of the background (low-priority) process data transfers for the EtherCAT network with index ***i***. The value of this element is set only if **ECAT[*i*].LPStateCheck** is set to 1 to enable automatic state checking of the background transfers. The state values can be:

- 0: No registered process data were exchanged
- 1: Some registered process data were exchanged
- 2: (reserved)
- 3: All registered process data were exchanged

If **ECAT[*i*].LPnotLRW** is set to the default value of 0, this reported state is valid for both input and output exchanges. If **ECAT[*i*].LPnotLRW** is set to 1 to disable common input/output exchanges, this reported state is valid only for input exchanges, and **ECAT[*i*].LPOutputDomainState** is used for output exchanges.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPRxTime

Description: EtherCAT network background receive time

Range: Non-negative floating-point

Units: Microseconds

ECAT[*i*].LPRxTime contains the time in microseconds that it took to receive the input and status data in the most recent background network cycle (when **ECAT[*i*].Enable** = 2 or 3) on the EtherCAT network with index ***i***. It is primarily intended for diagnostic use.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].LPTxTime

Description: EtherCAT network background transmit time

Range: Non-negative floating-point

Units: Microseconds

ECAT[*i*].LPTxTime contains the time in microseconds that it took to transmit the output and command data in the most recent background network cycle (when **ECAT[*i*].Enable** = 2 or 3) on the EtherCAT network with index ***i***. It is primarily intended for diagnostic use.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].MasterState

Description: Present state of EtherCAT network master

Range: 0, 1, 2, 4, 8

Units: Enumeration

ECAT[*i*].MasterState contains a value representing the present operational state of the master for the EtherCAT network with index ***i***. The value of this element is set only if **ECAT[*i*].RTStateCheck** is set to 1 to enable automatic state checking of the master. The state values can be:

- 1: INIT (initialized)
- 2: PREOP (preoperational)
- 4: SAFEOP (operational in safety mode)
- 8: OP (fully operational)

ECAT[*i*].MaxRxTime

Description: EtherCAT maximum network receive time

Range: Non-negative floating-point

Units: Microseconds

ECAT[*i*].MaxRxTime contains the time in microseconds that it took to receive the input and status data in the longest network cycle (when **ECAT[*i*].Enable** = 1) on the EtherCAT network with index ***i***. It is primarily intended for diagnostic use.

The user can set **ECAT[*i*].MaxRxTime** to 0.0 to start a new evaluation period.

This element is only used if **Sys.EcatType** is set to 1.

ECAT[*i*].MaxTxTime

Description: EtherCAT maximum network transmit time

Range: Non-negative floating-point

Units: Microseconds

ECAT[*i*].MaxTxTime contains the time in microseconds that it took to transmit the output and command data in the longest network cycle (when **ECAT[*i*].Enable** = 1) on the EtherCAT network with index ***i***. It is primarily intended for diagnostic use.

The user can set **ECAT[*i*].MaxTxTime** to 0.0 to start a new evaluation period.

This element is only used if **Sys.EcatType** is set to 1.

ECAT[*i*].RTDomainOutputState

Description: Present state of EtherCAT cyclic process output exchanges

Range: 0 .. 3

Units: Enumeration

ECAT[*i*].RTDomainOutputState contains a value representing the present state of the cyclic (real-time) process data output transfers for the EtherCAT network with index ***i***. The value of this element is set only if **ECAT[*i*].RTStateCheck** is set to 1 to enable automatic state checking of the real-time transfers. The state values can be:

- 0: No registered process data were exchanged
- 1: Some registered process data were exchanged
- 2: (reserved)
- 3: All registered process data were exchanged

If **ECAT[*i*].RTnotLRW** is set to the default value of 0, common input/output exchanges are enabled, and the common state for both input and output exchanges is reported in **ECAT[*i*].RTDomainState** and this element is not used. If **ECAT[*i*].RTnotLRW** is set to 1 to disable common input/output exchanges, the separate state for output exchanges is reported here in **ECAT[*i*].RTDomainOutputState**.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].RTDomainState

Description: Present state of EtherCAT cyclic process I/O exchanges

Range: 0 .. 3

Units: Enumeration

ECAT[*i*].RTDomainState contains a value representing the present state of the cyclic (real-time) process data transfers for the EtherCAT network with index ***i***. The value of this element is set only if **ECAT[*i*].RTStateCheck** is set to 1 to enable automatic state checking of the real-time transfers. The state values can be:

- 0: No registered process data were exchanged

- 1: Some registered process data were exchanged
- 2: (reserved)
- 3: All registered process data were exchanged

If **ECAT[*i*].RTnotLRW** is set to the default value of 0, this reported state is valid for both input and output exchanges. If **ECAT[*i*].RTnotLRW** is set to 1 to disable common input/output exchanges, this reported state is valid only for input exchanges, and **ECAT[*i*].RTOutputDomainState** can be monitored for output exchanges.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].RxTime

Description: EtherCAT network foreground receive time

Range: Non-negative floating-point

Units: Microseconds

ECAT[*i*].RxTime contains the time in microseconds that it took to receive the input and status data in the most recent foreground network cycle (when **ECAT[*i*].Enable** = 1) on the EtherCAT network with index ***i***. It is primarily intended for diagnostic use.

ECAT[*i*].TxTime

Description: EtherCAT network foreground transmit time

Range: Non-negative floating-point

Units: Microseconds

ECAT[*i*].TxTime contains the time in microseconds that it took to transmit the output and command data in the most recent foreground network cycle (when **ECAT[*i*].Enable** = 1) on the EtherCAT network with index ***i***. It is primarily intended for diagnostic use.

EtherCAT Slave Status Elements

ECAT[*i*].Slave[*j*].Online

Description: Status bit indicating whether the slave is online

Range: 0 .. 1

Units: none

Default: 0

Legacy I-variable alias: none

If the slave is physically connected to the EtherCAT network and powered on, this parameter will have the value 1. Otherwise, it will be 0.

This element is only used if **Sys.EcatType** is set to 0.

ECAT[*i*].Slave[*j*].State

Description: The current state of the slave.

Range: 1, 2, 4, 8

Units: none

Default: 0

Legacy I-variable alias: none

The current state of the slave 1 = INIT, 2 = PREOP, 4 = SAFEOP, 8 = OP. This variable is only used if **ECAT[*i*].RTStateCheck** = 1. Otherwise this variable is not monitored.

This element is only used if **Sys.EcatType** is set to 0.

EncTable[*n*]. Encoder Conversion Table Status Elements

The status elements in this section are set automatically by the Power PMAC in the process of executing encoder conversion table entries each servo cycle.

EncTable[*n*].Cos

Description: Encoder table entry cosine input

Range: -32,768 .. 32,767

Units: Signed 16-bit ADC units

EncTable[*n*].Cos is the corrected "cosine" input value for the most recent servo cycle. It is only used if saved setup element **EncTable[*n*].Type** is set to 4 or 6 for software sinusoidal encoder interpolation or direct resolver conversion, respectively. It is the sum of the input value of the second ADC register for the entry and the value of saved setup element **EncTable[*n*].CosBias**. It is primarily for diagnostic purposes.

EncTable[*n*].counter

Description: Encoder table entry state cycle counter

Range: Non-negative integer

Units: Servo cycles

EncTable[*n*].counter contains the number of consecutive servo cycles the source position value has exceeded the user-specified velocity or acceleration limit as set by saved setup element **EncTable[*n*].MaxDelta** (if enabled). When the value of **EncTable[*n*].counter** exceeds that of **EncTable[*n*].index3**, the result is incremented using the value of **EncTable[*n*].MaxDelta**, not **EncTable[*n*].PrevDelta**.

In a "triggered time base" entry, **EncTable[*n*].Counter** is set to 1 when the trigger is armed, and to 0 otherwise.

EncTable[*n*].DeltaPos

Description: Encoder table entry scaled change in position

Range: Floating-point

Units: User set

EncTable[*n*].DeltaPos is the resulting change in source position of the encoder table entry for the most recent servo cycle. It is a floating-point value, scaled into units of the user's choice due to multiplication by the saved setup element **EncTable[*n*].ScaleFactor**.

EncTable[*n*].DeltaPos is the primary result of the encoder table entry. It is implicitly selected for use by the motor servo algorithms when **Motor[*x*].pEnc**, **Motor[*x*].pEnc2**, or **Motor[*x*].pMasterEnc** is set to **EncTable[*n*].a**. It is explicitly used for an "external time base" source when **Coord[*x*].pDesTimeBase** is set to **EncTable[*n*].DeltaPos.a**.

EncTable[*n*].DeltaRes

Description: Encoder table entry tracking filter change in residual

Range: Integer

Units: User set

EncTable[*n*].DeltaRes is the change in the "residual" value of the tracking filter (if enabled) of the encoder table entry for the most recent servo cycle. It is for internal use in the tracking filter calculations.

EncTable[*n*].EncRes

Description: Encoder table entry tracking filter residual

Range: Integer

Units: User set

EncTable[*n*].EncRes is the "residual" value of the tracking filter (if enabled) of the encoder table entry for the most recent servo cycle. It is for internal use in the tracking filter calculations.

EncTable[*n*].PrevEnc

Description: Encoder table entry stored position value

Range: -2^{31}.. $2^{31}-1$

Units: Source LSBs

EncTable[*n*].PrevEnc is the processed source position for the most recent servo cycle, stored by Power PMAC for use in the next servo cycle. It is a 32-bit integer value that has *not* been multiplied by the saved setup element **EncTable[*n*].ScaleFactor**.

EncTable[*n*].Sin

Description: Encoder table entry sine input

Range: -32,768 .. 32,767

Units: Signed 16-bit ADC units

EncTable[*n*].Sin is the corrected "sine" input value for the most recent servo cycle. It is only used if saved setup element **EncTable[*n*].Type** is set to 4 or 6 for software sinusoidal encoder interpolation or direct resolver conversion, respectively. It is the sum of the input value of the first ADC register for the entry and the value of saved setup element **EncTable[*n*].SinBias**. It is primarily for diagnostic purposes.

EncTable[*n*].Status

Description: Encoder table status register

Range: 0 .. 2^{32}-1

Units: Bit field

EncTable[*n*].Status provides status information for certain conversion methods.

Bit 0 of **Status** is set to 1 in any servo cycle where the input position value is determined to be invalid, either because it violates the change limit set by **EncTable[*n*].MaxDelta** (in several methods), or if any of the error bits specified by the mask word of **EncTable[*n*].index6** in the register specified by **EncTable[*n*].pEnc1** in the Type 12 method is set. Bit 0 is set to 0 in any servo cycle where the input position value is determined to be valid.

EncTable[*n*].Status shares a register with **EncTable[*n*].SumOfSqr**. It is new in V2.0 firmware, released 1st quarter 2015.

EncTable[*n*].SumOfSqr

Description: Encoder table sum-of-squares magnitude

Range: 0 .. 2^{32}-1

Units: Square of 16-bit ADC units

EncTable[*n*].SumOfSqr is the "sum of squares" magnitude of the corrected "sine" and "cosine" input values for the most recent servo cycle. It is only used if saved setup element **EncTable[*n*].Type** is set to 4, 6, or 7 for software sinusoidal encoder interpolation or direct resolver conversion, respectively. It is calculated as the sum of the square of **EncTable[*n*].Sin** and the square of **EncTable[*n*].Cos**. It is primarily for diagnostic purposes, useful for determining the magnitude of the input signals. Ideally, it will be virtually constant over a cycle.

EncTable[*n*].SumOfSqr shares a register with **EncTable[*n*].Status**.

Gate1[*i*]. (PMAC2-Style Servo ASIC) Status Elements

The PMAC2-style DSPGATE1 servo ASIC provides 4 channels of servo interface. The status elements in this section permit the processor to access values resulting from the external inputs to this ASIC. The index value ***i*** can range from 4 to 19 (0 – 3 are reserved).

Gate1[*i*]. Multi-Channel Status Elements

The status elements in this section are not specific to any one channel of the ASIC.

Gate1[*i*].PartData[*k*]

Description: Raw part data for DSPGATE1 IC board

Range: $0 .. $FFFF

Units: None

Gate1[*i*].PartData[*k*] contains the raw data from one of the four (***k*** = 0 to 3) registers in an identification IC on the board containing the DSPGATE1 ASIC. It is used to derive the status elements **Gate1[*i*].PartNum**, **Gate1[*i*].PartOpt**, and **Gate1[*i*].PartRev**.

There is no direct access to the element in C. In C, a pointer set to the address of **Sys.piom** + (IC base address) + 0x40, 0x44, 0x48, or 0x4C should be used to access the registers for **Gate1[*i*].PartData[0]** to **PartData[3]**, respectively.

You can query the value of particular range of bits of **Gate1[*i*].PartData[*k*]** using the syntax **Gate1[*i*].PartData[*k*].*{start}*.*{width}*** where ***{start}*** is the number of the low bit in the range, and ***{width}*** is the number of bits in the range.

Gate1[*i*].PartNum

Description: Hardware part number for DSPGATE1 IC board

Range: Non-negative integer

Units: Enumeration

Gate1[*i*].PartNum contains the six-digit part number of the board containing the DSPGATE1 ASIC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. It will report as 0 if no ASIC with this index is present.

Part numbers presently existing are:

- ACC-24C2: 603681
- ACC-24C2A: 603611
- ACC-24E2: 603397

- ACC-24E2A: 603398
- ACC-24E2S: 603441
- ACC-51C: 603680
- ACC-51E: 603438

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate1PartData[*i*].Num**.

Gate1[*i*].PartOpt

Description: Optional hardware configuration code for DSPGATE1 IC board

Range: Non-negative integer

Units: Enumeration

Gate1[*i*].PartOpt contains a code indicating what optional hardware is present on the board containing the DSPGATE1 IC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. It will report as "nan" (not-a-number) if no ASIC with this index is present.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate1PartData[*i*].Opt**.

Gate1[*i*].PartRev

Description: Hardware revision number for DSPGATE1 IC board

Range: Non-negative integer

Units: Enumeration

Gate1[*i*].PartRev contains the revision number of the board containing the DSPGATE1 ASIC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate1PartData[*i*].Rev**.

Gate1[*i*].PartType

Description: Hardware part type for DSPGATE1 IC board

Range: Non-negative integer

Units: Bit field

Gate1[*i*].PartType contains an integer representing the type(s) of hardware interface of the board containing the DSPGATE1 ASIC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no ASIC with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present with the IC. It is possible that multiple interface types could be present.

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

Presently, the value of **PartType** for all DSPGATE1 IC boards is 1 (servo interface only).

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate1PartData[*i*].Type**.

Gate1[*i*]. Channel-Specific Status Elements

The status elements in this channel include register elements containing values resulting from input sensors connected to the ASIC channel, and individual status bits that indicate information about the channel's present state. Channel indices ***j*** can range from 0 to 3, representing hardware channels 1 to 4, respectively.

Gate1[*i*].Chan[*j*].ABC

Description: A, B, and C encoder channel input values

Range: 0 .. 7

Units: Bit field

Gate1[*i*].Chan[*j*].ABC contains the present logical state of the IC's A, B, and C encoder inputs for the channel, latched on the most recent servo interrupt. Bit 2 (value 4) represents the state of the C (index) channel; bit 1 (value 2) represents the state of the B channel; and bit 0 (value 1) represents the value of the A channel.

A value of 1 in a bit corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is a differential input, in which case, a value of 1 is attained when the "+" input has a higher voltage than the "-" input.

Gate1[*i*].Chan[*j*].ABC forms bits 12 – 14 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bits 20 – 22 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].Adc[*k*]

Description: Analog-to-digital converter value

Range: -8,388,608 .. 8,388,607 (Script)
-2^{31} .. $2^{31}-1$ (C)

Units: 24-bit ADC LSBs (Script)
32-bit ADC LSBs (C)

Gate1[*i*].Chan[*j*].Adc[*k*] contains the value for the analog-to-digital converter for the IC channel and phase that was latched on the most recent phase clock cycle. The element **Adc[0]** contains the value from the A/D converter connected to the Phase A input for the channel; **Adc[1]** contains the value from the A/D converter connected to the Phase B input for the channel.

The circuitry and register support up to 24-bit ADCs. If the interface is configured correctly, the data for an *n*-bit ADC should be present in the high *n* bits of this 24-bit element. When used for current feedback in the Power PMAC's "direct-PWM" commutation algorithm, the true data must be present in the highest bits.

If the ADC has "header bits" preceding the true data, a proper setting of saved setup element **Gate1[*i*].AdcStrobe** will cause the header data to be "rolled over" to the low 1 – 4 bits of this element.

Note that C programs and functions must access this element as a 32-bit value, with true data only in the high 24 bits (at most).

Gate1[*i*].Chan[*j*].CountError

Description: Encoder-decode counting error

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].CountError contains the present logical state of the IC's "encoder count error" status flag for the channel. A value of 1 indicates that an illegal quadrature transition has occurred, with both the A and B-channel input states from the encoder changing in the same encoder sample-clock (SCLK) cycle. When this illegal transition occurs, this flag is set and latched to indicate that subsequent reads of the encoder counter value will provide an erroneous value.

Gate1[*i*].Chan[*j*].CountError is forced to 0 during the power-up/reset of the Power PMAC. Once set, it can be cleared by writing a 0 to the bit. (This write capability is new in V2.1 firmware, released 1st quarter 2016.) Clearing the bit value does *not* remove the error in the encoder counter numerical value.

This error bit can also be cleared by setting **Gate1[*i*].Chan[*j*].PosClear** to 1 to zero out the encoder counter value. To do this, **Gate1[*i*].Chan[*j*].AmpEna** for the channel must be set to 0. Note that clearing the position of the encoder counter is potentially a very dangerous operation as the servo loop can react violently to a sudden change in position feedback value.

Gate1[*i*].Chan[*j*].CountError is bit 8 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 16 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].EncLossN

Description: Encoder-loss indicator bit

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].EncLossN indicates whether the quadrature encoder-detection circuitry associated with the ASIC senses a valid encoder signal or not. It is 1 if it sees a valid signal; it is 0 if it does not see a valid signal. The circuitry employs exclusive-or gates on the differential signal pairs, so this detection is only valid for encoders with differential line-driver outputs.

Note that this status bit does not actually reside in the ASIC itself; it comes from circuitry associated with the ASIC on several accessories, including the ACC-24E2, the ACC-24E2A, and the ACC-24E2S.

There is no direct access to the element in C. In C, a pointer set to the address of **Sys.piom** + (IC base address) + 0x40, 0x44, 0x48, or 0x4C should be used to access the registers for **Gate1[*i*].Chan[0]** to **Gate1[*i*].Chan[3]**, respectively. **EncLossN** is in bit 13 at the address.

Gate1[*i*].Chan[*j*].Equ

Description: Position-compare flag output value

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].Equ contains the present logical state of the IC's "EQU" position-compare flag output for the channel. A value of 1 corresponds to a high voltage out of the IC; a value of 0 corresponds to a low voltage out of the IC. On most Delta Tau hardware, this signal controls a sinking output driver, so a 1 state turns on the output driver, conducting current and pulling the output voltage low.

It is not possible to use this element to set the output state directly. It is necessary to use the 2-bit non-saved setup element **Gate1[*i*].Chan[*j*].EquWrite** to force an output value. The flag value will automatically change as the encoder counter value passes the positions set in **Gate1[*i*].Chan[*j*].CompA** and **Gate1[*i*].Chan[*j*].CompB**.

Gate1[*i*].Chan[*j*].Equ is bit 9 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 17 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].Fault

Description: Amplifier fault flag input value

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].Fault contains the present logical state of the IC's FAULT (amplifier fault) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On much Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator. This can also be a differential input, in which case a value of 1 is attained when the FAULT+ input has a higher voltage than the FAULT- input.

Note that this element simply represents the input level into the IC; either a 0 or a 1 state can be interpreted as a fault from the amplifier, depending on the setting of **Motor[*x*].AmpFaultLevel**.

This hardware input element is distinct from the motor software status element **Motor[*x*].AmpFault**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate1[*i*].Chan[*j*].Fault is bit 17 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 25 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].HallState

Description: Captured Hall position state value

Range: 0 .. 7

Units: Enumeration

Gate1[*i*].Chan[*j*].HallState contains the most recently captured position edge in the Hall cycle from the channel's U, V, and W flag inputs, when the channel is set up to capture encoder position on any U, V, or W transition (by **Gate1[*i*].Chan[*j*].CaptCtrl** bits 0 and 1 set to 0, and **Gate1[*i*].Chan[*j*].GatedIndexSel** set to 1).

Valid position edge values in the 6-state Hall cycle range from 0 to 5. The following table shows how these 6 values correspond to the legal input values of the U, V, and W flag inputs:

Gate1[*i*].Chan[*j*].HallState	0	1	2	3	4	5
Gate1[*i*].Chan[*j*].UVW	5	1	3	2	6	4	5
U flag input state	1	0	0	0	1	1	1
V flag input state	0	0	1	1	1	0	0
W flag input state	1	1	1	0	0	0	1

Power PMAC considers the standard zero-degree point in the Hall cycle to be the transition edge 1 (between UVW values 1 and 3). If this is located at the zero-degree point in the commutation cycle, no offset is required from sensor position to commutation angle.

Before any captured transitions have occurred, or if there is a transition to an illegal UVW flag input state, **Gate1[*i*].Chan[*j*].HallState** will report a value of 7.

Gate1[*i*].Chan[*j*].HallState forms bits 0 – 2 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bits 8 – 10 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].HomeCapt

Description: Encoder count value latched on external trigger

Range: 0 .. 16,777,215

Units: Encoder counts

Gate1[*i*].Chan[*j*].HomeCapt contains the hardware encoder count value for the IC channel that was latched on the most recent external trigger. It is a 24-bit value, in units of encoder counts as determined by the encoder decode specified by **Gate1[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction.

The trigger state is selected by saved setup elements **Gate1[*i*].Chan[*j*].CaptCtrl** and **Gate1[*i*].Chan[*j*].CaptFlagSel**.

The act of reading **Gate1[*i*].Chan[*j*].HomeCapt** re-arms the trigger circuitry for the next capture. Because level triggering is used, if the triggering inputs are still in the specified state, an immediate re-triggering will occur, latching a (potentially) new count value into this element

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate1[*i*].Chan[*j*].HomeFlag

Description: HOME flag input value

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].HomeFlag contains the present logical state of the IC's HOME flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

Gate1[*i*].Chan[*j*].HomeFlag is bit 16 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 24 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].MinusLimit

Description: MLIM flag input value

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].MinusLimit contains the present logical state of the IC's MLIM (negative overtravel limit) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

This hardware input element is distinct from the motor software status element **Motor[*x*].MinusLimit**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate1[*i*].Chan[*j*].MinusLimit is bit 18 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 26 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].PhaseCapt

Description: Encoder count value latched on phase interrupt

Range: 0 .. 16,777,215

Units: Encoder counts

Gate1[*i*].Chan[*j*].PhaseCapt contains the hardware encoder count value for the IC channel that was latched on the most recent phase interrupt. It is a 24-bit value, in units of encoder counts as determined by the encoder decode specified by **Gate1[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction. Because the instantaneous counter value is not accessible by the processor, this element contains the most direct accessible representation of the present count value.

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate1[*i*].Chan[*j*].PlusLimit

Description: PLIM flag input value

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].PlusLimit contains the present logical state of the IC's PLIM (positive overtravel limit) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

This hardware input element is distinct from the motor software status element **Motor[*x*].PlusLimit**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate1[*i*].Chan[*j*].PlusLimit is bit 17 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 25 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].PosCapt

Description: Position-capture trigger flags

Range: 0 .. 3

Units: Bit field

Gate1[*i*].Chan[*j*].PosCapt contains the present state of the channel's two trigger flags. Bit 1 (value 2) is the state of the channel's "general" trigger flag. It is set whenever the trigger condition specified by **Gate1[*i*].Chan[*j*].CaptCtrl** and **Gate1[*i*].Chan[*j*].CaptFlagSel** has occurred. Bit 0 (value 1) is the state of the "gated-index" trigger flag. It is set whenever a trigger condition that includes the encoder's index channel "gated" to 1 quadrature state wide (with **Gate1[*i*].Chan[*j*].CaptCtrl** bit 0 = 1 and **Gate1[*i*].Chan[*j*].GatedIndexSel** = 1) has occurred. This trigger is mainly used to confirm that the correct number of counts have accumulated between consecutive index pulses.

Both flags are automatically cleared by the act of reading the register containing the encoder position that was captured at the instant of the trigger: **Gate1[*i*].Chan[*j*].HomeCapt**. Both flags are level-triggered, not edge-triggered, so if the selected inputs are still in the specified trigger state when the flags are cleared, there will be an immediate re-triggering.

Gate1[*i*].Chan[*j*].PosCapt forms bits 10 – 11 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bits 18 – 19 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].ServoCapt

Description: Encoder count value latched on servo interrupt

Range: 0 .. 16,777,215

Units: 2* encoder counts

Gate1[*i*].Chan[*j*].ServoCapt contains the hardware encoder count value for the IC channel that was latched on the most recent servo interrupt, plus the direction of the last count received. It is a 24-bit value. Bit 0 contains the direction of the last count received (0 = up, 1 = down). Bits 1 to 23 contain the latched count value, in units of encoder counts as determined by the encoder decode specified by **Gate1[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction.

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate1[*i*].Chan[*j*].Status

Description: Channel status word

Range: $0 .. $FFFFFF (Script)
$0 .. $FFFFFFFF (C)

Units: Bit field

Gate1[*i*].Chan[*j*].Status is a 24-bit status word (the high 24 bits of a 32-bit word when accessed in a C program) that displays the contents of the IC's hardware status register for the channel. It contains many individual status bits and elements. The individual status bits and elements shown in bold are accessible as separate elements in Script programs and commands, but not in C programs and functions.

The following table provides a list of these status bits in the word:

Script Bit #	Script Hex Value	C Bit #	C Hex Value	Element Name: Gate1[*i*].Chan[*j*].*{element}*	Description
23	$800000	31	$80000000	**T**	T flag input value
20 – 22	$700000	28 – 30	$70000000	**UVW**	U, V, and W flag input values
19	$80000	27	$8000000	**UserFlag**	USER flag input value
18	$40000	26	$4000000	**MinusLimit**	MLIM flag input value
17	$20000	25	$2000000	**PlusLimit**	PLIM flag input value
16	$10000	24	$1000000	**HomeFlag**	HOME flag input value
15	$8000	23	$800000	**Fault**	Amplifier fault flag input value
12 – 14	$7000	20 – 22	$700000	**ABC**	Encoder A, B, and C channel input values
10 – 11	$C00	18 – 19	$C0000	**PosCapt**	Position-captured trigger flags
9	$200	17	$20000	**Equ**	Position-compare output state
8	$100	16	$10000	**CountError**	Encoder decode counting error
4 – 7	$F0	12 – 15	$F000	-	*(Reserved for future use)*
3	$8	11	$800	*DemuxInvalid*	Invalid UVW demultiplexing
0 – 2	$7	8 – 10	$700	**HallState**	Captured Hall state
-	-	0 – 7	$FF	-	*(No hardware present here)*

Each element is described in more detail in its own individual specification.

Gate1[*i*].Chan[*j*].T

Description: T flag input value

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].T contains the present logical state of the IC's T flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, the buffering external to the IC is non-inverting, so this applies to the voltages into the controller as well.

Gate1[*i*].Chan[*j*].T is bit 23 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 31 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].TimeBetweenCts

Description: Time between last two encoder counts

Range: 0 .. 16,777,215

Units: 2 * SCLK cycles

Gate1[*i*].Chan[*j*].TimeBetweenCts normally contains the time between the last two encoder counts for the IC channel that was latched on the most recent servo interrupt, plus a flag indicating whether these two counts were in the same or different directions. It is a 24-bit value. Bit 0 contains the direction-change flag (0 = no change, 1 = change). Bits 1 to 23 contain the latched timer value, in units of encoder sample clock (SCLK) cycles, whose period/frequency is specified by **Gate1[*i*].HardwareClockCtrl**. This timer value is commonly used in the default "software 1/T extension" to estimate fractional-count position.

If saved setup element **Gate1[*i*].Chan[*j*].EncCtrl** is set to 12 to specify "MLDT pulse timing", this element contains the time between the last generated output pulse on the channel and the reception of an "echo pulse" from the MLDT, with units of the internal 117.9648 MHz clock signal. This time is proportional to the distance of the MLDT moving magnet from the end of the sensor.

If saved setup element **Gate1[*i*].Chan[*j*].OneOverTEna** is set to 1 to enable "hardware 1/T extension", this 24-bit element is split into two 12-bit components. Bits 0 to 11 are a writable component representing the fractional-count portion of the "Compare A" position value, whose whole-count portion is in **Gate1[*i*].Chan[*j*].CompA**. Bits 12 to 23 are the fractional-count portion of the servo-captured position value, whose whole-count portion is in **Gate1[*i*].Chan[*j*].ServoCapt**. Both fractional components are in units of 1/4096 of a count.

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate1[*i*].Chan[*j*].TimeSinceCts

Description: Time since last encoder count

Range: 0 .. 16,777,215

Units: SCLK cycles

Gate1[*i*].Chan[*j*].TimeSinceCts normally contains the time since the last encoder count for the IC channel to the most recent servo interrupt. It is a 24-bit value, in units of encoder sample clock (SCLK) cycles, whose period/frequency is specified by **Gate1[*i*].HardwareClockCtrl**. This timer value is commonly used in the default "software 1/T extension" to estimate fractional-count position.

If saved setup element **Gate1[*i*].Chan[*j*].OneOverTEna** is set to 1 to enable "hardware 1/T extension", this 24-bit element is split into two 12-bit components. Bits 0 to 11 are a writable component representing the fractional-count portion of the "Compare B" position value, whose whole-count portion is in **Gate1[*i*].Chan[*j*].CompB**. Bits 12 to 23 are the fractional-count portion of the servo-captured position value, whose whole-count portion is in **Gate1[*i*].Chan[*j*].HomeCapt**. Both fractional components are in units of 1/4096 of a count.

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate1[*i*].Chan[*j*].UserFlag

Description: USER flag input value

Range: 0 .. 1

Units: Boolean

Gate1[*i*].Chan[*j*].UserFlag contains the present logical state of the IC's USER flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

Gate1[*i*].Chan[*j*].UserFlag is bit 19 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 27 in C). In the C environment, it must be accessed through the full-word element.

Gate1[*i*].Chan[*j*].UVW

Description: U, V, and W flags input value

Range: 0 .. 7

Units: Bit field

Gate1[*i*].Chan[*j*].UVW contains the present logical state of the IC's U, V, and W flag inputs for the channel, latched on the most recent servo interrupt. Bit 2 (value 4) represents the state of the U flag; bit 1 (value 2) represents the state of the V flag; and bit 0 (value 1) represents the value of the W flag.

A value of 1 in a bit corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, the buffering external to the IC is non-inverting, so this applies to the voltages into the controller as well.

Gate1[*i*].Chan[*j*].UVW forms bits 20 – 22 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bits 28 – 30 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*]. (PMAC2-Style MACRO ASIC) Status Elements

The PMAC2-style DSPGATE2 MACRO ASIC provides a MACRO-ring interface, plus general-purpose I/O (or 2-channels of servo interface). The status elements in this section permit the processor to access values resulting from the external inputs to this ASIC.

Gate2[*i*]. Multi-Channel Status Elements

Gate2[*i*].LowIoGrayData

Description: Gray-to-binary converted input data word

Range: $0 .. $FFFFFF

Units: None

Gate2[*i*].LowIoGrayData contains the value of the input word on the IO00 – IO23 inputs to the DSPGATE2 IC after they have been processed through a Gray-code-to-binary conversion. If **Gate2[*i*].GrayCodeBitLenCtrl** is set to 0, the input word is converted as a single entity. If it is set greater than 0, the input word is split into two parts that are independently converted.

The unconverted input data is found in **Gate2[*i*].LowIoData**.

Gate2[*i*].PartData[*k*]

Description: Raw part data for DSPGATE2 IC board

Range: $0 .. $FFFF

Units: None

Gate2[*i*].PartData[*k*] contains the raw data from one of the four (***k*** = 0 to 3) registers in an identification IC on the board containing the DSPGATE2 ASIC. It is used to derive the status elements **Gate2[*i*].PartNum**, **Gate2[*i*].PartOpt**, and **Gate2[*i*].PartRev**.

There is no direct access to the element in C. In C, a pointer set to the address of **Sys.piom** + (IC base address) + 0x80, 0x84, 0x88, or 0x8C should be used to access the registers for **Gate2[*i*].PartData[0]** to **PartData[3]**, respectively.

You can query the value of particular range of bits of **Gate2[*i*].PartData[*k*]** using the syntax **Gate2[*i*].PartData[*k*].*{start}*.*{width}*** where ***{start}*** is the number of the low bit in the range, and ***{width}*** is the number of bits in the range.

Gate2[*i*].PartNum

Description: Hardware part number for DSPGATE2 IC board

Range: Non-negative integer

Units: Enumeration

Gate2[*i*].PartNum contains the six-digit part number of the board containing the DSPGATE2 ASIC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. It will report as 0 if no ASIC with this index is present.

Part numbers presently existing are:

- ACC-5E: 603437

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate2PartData[*i*].Num**.

Gate2[*i*].PartOpt

Description: Optional hardware configuration code for DSPGATE2 IC board

Range: Non-negative integer

Units: Enumeration

Gate2[*i*].PartOpt contains a code indicating what optional hardware is present on the board containing the DSPGATE2 IC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. It will report as "nan" (not-a-number) if no ASIC with this index is present.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate2PartData[*i*].Opt**.

Gate2[*i*].PartRev

Description: Hardware revision number for DSPGATE2 IC board

Range: Non-negative integer

Units: Enumeration

Gate2[*i*].PartRev contains the revision number of the board containing the DSPGATE2 ASIC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate2PartData[*i*].Rev**.

Gate2[*i*].PartType

Description: Hardware part type for DSPGATE2 IC board

Range: Non-negative integer

Units: Bit field

Gate2[*i*].PartType contains an integer representing the type(s) of hardware interface of the board containing the DSPGATE2 ASIC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no ASIC with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present with the IC. It is possible that multiple interface types could be present.

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

Presently, the value of **PartType** for all DSPGATE2 IC boards is 2 (MACRO interface only).

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate2PartData[*i*].Type**.

Gate2[*i*]. Channel-Specific Status Elements

The status elements in this channel include register elements containing values resulting from input sensors connected to the ASIC servo channel, and individual status bits that indicate information about the channel's present state. There are two servo channels in the DSPGATE2 IC, with index values ***j*** of 0 and 1.

Presently, only the encoder-input and pulse-output functions of the DSPGATE2 servo channels are supported on any Power PMAC-compatible device.

Gate2[*i*].Chan[*j*].ABC

Description: A, B, and C encoder channel input values

Range: 0 .. 7

Units: Bit field

Gate2[*i*].Chan[*j*].ABC contains the present logical state of the IC's A, B, and C encoder inputs for the channel, latched on the most recent servo interrupt. Bit 2 (value 4) represents the state of the C (index) channel; bit 1 (value 2) represents the state of the B channel; and bit 0 (value 1) represents the value of the A channel.

A value of 1 in a bit corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is a differential input, in which case, a value of 1 is attained when the "+" input has a higher voltage than the "-" input.

Gate2[*i*].Chan[*j*].ABC forms bits 12 – 14 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bits 20 – 22 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].Adc[*k*]

Description: Analog-to-digital converter value

Range: -8,388,608 .. 8,388,607 (Script)
-2^{31} .. $2^{31}-1$ (C)

Units: 24-bit ADC LSBs (Script)
32-bit ADC LSBs (C)

Gate2[*i*].Chan[*j*].Adc[*k*] contains the value for the analog-to-digital converter for the IC channel and phase that was latched on the most recent phase clock cycle. The element **Adc[0]** contains the value from the A/D converter connected to the Phase A input for the channel; **Adc[1]** contains the value from the A/D converter connected to the Phase B input for the channel.

The circuitry and register support up to 24-bit ADCs. If the interface is configured correctly, the data for an *n*-bit ADC should be present in the high *n* bits of this 24-bit element. When used for current feedback in the Power PMAC's "direct-PWM" commutation algorithm, the true data must be present in the highest bits.

If the ADC has "header bits" preceding the true data, a proper setting of saved setup element **Gate2[*i*].AdcStrobe** will cause the header data to be "rolled over" to the low 1 – 4 bits of this element.

Note that C programs and functions must access this element as a 32-bit value, with true data only in the high 24 bits (at most).

Gate2[*i*].Chan[*j*].CountError

Description: Encoder-decode counting error

Range: 0 .. 1

Units: Boolean

Gate2[*i*].Chan[*j*].CountError contains the present logical state of the IC's "encoder count error" status flag for the channel. A value of 1 indicates that an illegal quadrature transition has occurred, with both the A and B-channel input states from the encoder changing in the same encoder sample-clock (SCLK) cycle. When this illegal transition occurs, this flag is set and latched to indicate that subsequent reads of any encoder counter value will provide an erroneous value.

Gate2[*i*].Chan[*j*].CountError is forced to 0 during the power-up/reset of the Power PMAC. Once set, it can be cleared by writing a 0 to the bit. (This write capability is new in V2.1 firmware, released 1st quarter 2016.) Clearing the bit value does *not* remove the error in the encoder counter numerical value.

This error bit can also be cleared by setting **Gate2[*i*].Chan[*j*].PosClear** to 1 to zero out the encoder counter value. To do this, **Gate2[*i*].Chan[*j*].AmpEna** for the channel must be set to 0. Note that clearing the position of the encoder counter is potentially a very dangerous operation as the servo loop can react violently to a sudden change in position feedback value.

Gate2[*i*].Chan[*j*].CountError is bit 8 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bit 16 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].Equ

Description: Position-compare flag output value

Range: 0 .. 1

Units: Boolean

Gate2[*i*].Chan[*j*].Equ contains the present logical state of the IC's "EQU" position-compare flag output for the channel. A value of 1 corresponds to a high voltage out of the IC; a value of 0 corresponds to a low voltage out of the IC. On most Delta Tau hardware, this signal controls a sinking output driver, so a 1 state turns on the output driver, conducting current and pulling the output voltage low.

It is not possible to use this element to set the output state directly. It is necessary to use the 2-bit non-saved setup element **Gate2[*i*].Chan[*j*].EquWrite** to force an output value. The flag value will automatically change as the encoder counter value passes the positions set in **Gate2[*i*].Chan[*j*].CompA** and **Gate2[*i*].Chan[*j*].CompB**.

Gate2[*i*].Chan[*j*].Equ is bit 9 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bit 17 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].Fault

Description: Amplifier fault flag input value

Range: 0 .. 1

Units: Boolean

Gate2[*i*].Chan[*j*].Fault contains the present logical state of the IC's FAULT (amplifier fault) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On much Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator. This can also be a differential input, in which case a value of 1 is attained when the FAULT+ input has a higher voltage than the FAULT- input.

Note that this element simply represents the input level into the IC; either a 0 or a 1 state can be interpreted as a fault from the amplifier, depending on the setting of **Motor[*x*].AmpFaultLevel**.

This hardware input element is distinct from the motor software status element **Motor[*x*].AmpFault**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate2[*i*].Chan[*j*].Fault is bit 17 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bit 25 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].HallState

Description: Captured Hall position state value

Range: 0 .. 7

Units: Enumeration

Gate2[*i*].Chan[*j*].HallState contains the most recently captured position edge in the Hall cycle from the channel's U, V, and W flag inputs, when the channel is set up to capture encoder position on any U, V, or W transition (by **Gate2[*i*].Chan[*j*].CaptCtrl** bits 0 and 1 set to 0, and **Gate2[*i*].Chan[*j*].GatedIndexSel** set to 1).

Valid position edge values in the 6-state Hall cycle range from 0 to 5. The following table shows how these 6 values correspond to the legal input values of the U, V, and W flag inputs:

Gate2[*i*].Chan[*j*].HallState	0	1	2	3	4	5	
Gate2[*i*].Chan[*j*].UVW	5	1	3	2	6	4	5
U flag input state	1	0	0	0	1	1	1
V flag input state	0	0	1	1	1	0	0
W flag input state	1	1	1	0	0	0	1

Power PMAC considers the standard zero-degree point in the Hall cycle to be the transition edge 1 (between UVW values 1 and 3). If this is located at the zero-degree point in the commutation cycle, no offset is required from sensor position to commutation angle.

Before any captured transitions have occurred, or if there is a transition to an illegal UVW flag input state, **Gate2[*i*].Chan[*j*].HallState** will report a value of 7.

Gate2[*i*].Chan[*j*].HallState forms bits 0 – 2 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bits 8 – 10 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].HomeCapt

Description: Encoder count value latched on external trigger

Range: 0 .. 16,777,215

Units: Encoder counts

Gate2[*i*].Chan[*j*].HomeCapt contains the hardware encoder count value for the IC channel that was latched on the most recent external trigger. It is a 24-bit value, in units of encoder counts as determined by the encoder decode specified by **Gate2[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction.

The trigger state is selected by saved setup elements **Gate2[*i*].Chan[*j*].CaptCtrl** and **Gate2[*i*].Chan[*j*].CaptFlagSel**.

The act of reading **Gate2[*i*].Chan[*j*].HomeCapt** re-arms the trigger circuitry for the next capture. Because level triggering is used, if the triggering inputs are still in the specified state, an immediate re-triggering will occur, latching a (potentially) new count value into this element

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate2[*i*].Chan[*j*].HomeFlag

Description: HOME flag input value

Range: 0 .. 1

Units: Boolean

Gate2[*i*].Chan[*j*].HomeFlag contains the present logical state of the IC's HOME flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

Gate2[*i*].Chan[*j*].HomeFlag is bit 16 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bit 24 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].MinusLimit

Description: MLIM flag input value

Range: 0 .. 1

Units: Boolean

Gate2[*i*].Chan[*j*].MinusLimit contains the present logical state of the IC's MLIM (negative overtravel limit) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

This hardware input element is distinct from the motor software status element **Motor[*x*].MinusLimit**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate1[*i*].Chan[*j*].MinusLimit is bit 18 of the full-word element **Gate1[*i*].Chan[*j*].Status** in the Script environment (bit 26 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].PhaseCapt

Description: Encoder count value latched on phase interrupt

Range: 0 .. 16,777,215

Units: Encoder counts

Gate2[*i*].Chan[*j*].PhaseCapt contains the hardware encoder count value for the IC channel that was latched on the most recent phase interrupt. It is a 24-bit value, in units of encoder counts as determined by the encoder decode specified by **Gate2[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction. Because the instantaneous counter value is not accessible by the processor, this element contains the most direct accessible representation of the present count value.

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate2[*i*].Chan[*j*].PlusLimit

Description: PLIM flag input value

Range: 0 .. 1

Units: Boolean

Gate2[*i*].Chan[*j*].PlusLimit contains the present logical state of the IC's PLIM (positive overtravel limit) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

This hardware input element is distinct from the motor software status element **Motor[*x*].PlusLimit**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate2[*i*].Chan[*j*].PlusLimit is bit 17 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bit 25 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].PosCapt

Description: Position-capture trigger flags

Range: 0 .. 3

Units: Bit field

Gate2[*i*].Chan[*j*].PosCapt contains the present state of the channel's two trigger flags. Bit 1 (value 2) is the state of the channel's "general" trigger flag. It is set whenever the trigger condition specified by **Gate2[*i*].Chan[*j*].CaptCtrl** and **Gate1[*i*].Chan[*j*].CaptFlagSel** has occurred. Bit 0 (value 1) is the state of the "gated-index" trigger flag. It is set whenever a trigger condition that includes the encoder's index channel "gated" to 1 quadrature state wide (with **Gate2[*i*].Chan[*j*].CaptCtrl** bit 0 = 1 and **Gate2[*i*].Chan[*j*].GatedIndexSel** = 1) has occurred. This trigger is mainly used to confirm that the correct number of counts have accumulated between consecutive index pulses.

Both flags are automatically cleared by the act of reading the register containing the encoder position that was captured at the instant of the trigger: **Gate2[*i*].Chan[*j*].HomeCapt**. Both flags are level-triggered, not edge-triggered, so if the selected inputs are still in the specified trigger state when the flags are cleared, there will be an immediate re-triggering.

Gate2[*i*].Chan[*j*].PosCapt forms bits 10 – 11 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bits 18 – 19 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].ServoCapt

Description: Encoder count value latched on servo interrupt

Range: 0 .. 16,777,215

Units: 2 * encoder counts

Gate2[*i*].Chan[*j*].ServoCapt contains the hardware encoder count value for the IC channel that was latched on the most recent servo interrupt, plus the direction of the last count received. It is a 24-bit value. Bit 0 contains the direction of the last count received (0 = up, 1 = down). Bits 1 to 23 contain the latched count value, in units of encoder counts as determined by the encoder decode specified by **Gate2[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction.

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate2[*i*].Chan[*j*].Status

Description: Channel status word

Range: $0 .. $FFFFFF (Script)
$0 .. $FFFFFFFF (C)

Units: Bit field

Gate2[*i*].Chan[*j*].Status is a 24-bit status word (the high 24 bits of a 32-bit word when accessed in a C program) that displays the contents of the IC's hardware status register for the channel. It contains many individual status bits and elements. The individual status bits and elements are accessible as separate elements in Script programs and commands, but not in C programs and functions.

The following table provides a list of these status bits in the word:

Script Bit #	Script Hex Value	C Bit #	C Hex Value	Element Name: Gate1[*i*].Chan[*j*].*{element}*	Description
23	$800000	31	$80000000	**T**	T flag input value
20 – 22	$700000	28 – 30	$70000000	**UVW**	U, V, and W flag input values
19	$80000	27	$8000000	**UserFlag**	USER flag input value
18	$40000	26	$4000000	**MinusLimit**	MLIM flag input value
17	$20000	25	$2000000	**PlusLimit**	PLIM flag input value
16	$10000	24	$1000000	**HomeFlag**	HOME flag input value
15	$8000	23	$800000	**Fault**	Amplifier fault flag input value
12 – 14	$7000	20 – 22	$700000	**ABC**	Encoder A, B, and C channel input values
10 – 11	$C00	18 – 19	$C0000	**PosCapt**	Position-captured trigger flags
9	$200	17	$20000	**Equ**	Position-compare output state
8	$100	16	$10000	**CountError**	Encoder decode counting error
4 – 7	$F0	12 – 15	$F000	-	*(Reserved for future use)*
3	$8	11	$800	*DemuxInvalid*	Invalid UVW demultiplexing
0 – 2	$7	8 – 10	$700	**HallState**	Captured Hall state
-	-	0 – 7	$FF	-	*(No hardware present here)*

Each element is described in more detail in its own individual specification.

Gate2[*i*].Chan[*j*].T

Description: T flag input value

Range: 0 .. 1

Units: Boolean

Gate2[*i*].Chan[*j*].T contains the present logical state of the IC's T flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, the buffering external to the IC is non-inverting, so this applies to the voltages into the controller as well.

Gate2[*i*].Chan[*j*].T is bit 23 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bit 31 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].TimeBetweenCts

Description: Time between last two encoder counts

Range: 0 .. 16,777,215

Units: 2 * SCLK cycles

Gate2[*i*].Chan[*j*].TimeBetweenCts normally contains the time between the last two encoder counts for the IC channel that was latched on the most recent servo interrupt, plus a flag indicating whether these two counts were in the same or different directions. It is a 24-bit value. Bit 0 contains the direction-change flag (0 = no change, 1 = change). Bits 1 to 23 contain the latched timer value, in units of encoder sample clock (SCLK) cycles, whose period/frequency is specified by **Gate2[*i*].HardwareClockCtrl**. This timer value is commonly used in the default "software 1/T extension" to estimate fractional-count position.

If saved setup element **Gate2[*i*].Chan[*j*].EncCtrl** is set to 12 to specify "MLDT pulse timing", this element contains the time between the last generated output pulse on the channel and the reception of an "echo pulse" from the MLDT, with units of the internal 117.9648 MHz clock signal. This time is proportional to the distance of the MLDT moving magnet from the end of the sensor.

If saved setup element **Gate2[*i*].Chan[*j*].OneOverTEna** is set to 1 to enable "hardware 1/T extension", this 24-bit element is split into two 12-bit components. Bits 0 to 11 are a writable component representing the fractional-count portion of the "Compare A" position value, whose whole-count portion is in **Gate2[*i*].Chan[*j*].CompA**. Bits 12 to 23 are the fractional-count portion of the servo-captured position value, whose whole-count portion is in **Gate2[*i*].Chan[*j*].ServoCapt**. Both fractional components are in units of 1/4096 of a count.

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate2[*i*].Chan[*j*].TimeSinceCts

Description: Time since last encoder count

Range: 0 .. 16,777,215

Units: SCLK cycles

Gate2[*i*].Chan[*j*].TimeSinceCts normally contains the time since the last encoder count for the IC channel to the most recent servo interrupt. It is a 24-bit value, in units of encoder sample clock (SCLK) cycles, whose period/frequency is specified by **Gate2[*i*].HardwareClockCtrl**. This timer value is commonly used in the default "software 1/T extension" to estimate fractional-count position.

If saved setup element **Gate2[*i*].Chan[*j*].OneOverTEna** is set to 1 to enable "hardware 1/T extension", this 24-bit element is split into two 12-bit components. Bits 0 to 11 are a writable component representing the fractional-count portion of the "Compare B" position value, whose whole-count portion is in **Gate2[*i*].Chan[*j*].CompB**. Bits 12 to 23 are the fractional-count portion of the servo-captured position value, whose whole-count portion is in **Gate2[*i*].Chan[*j*].HomeCapt**. Both fractional components are in units of 1/4096 of a count.

If accessed from a C program, this element is a 32-bit value, with real data in the high 24 bits.

Gate2[*i*].Chan[*j*].UserFlag

Description: USER flag input value

Range: 0 .. 1

Units: Boolean

Gate2[*i*].Chan[*j*].UserFlag contains the present logical state of the IC's USER flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

Gate2[*i*].Chan[*j*].UserFlag is bit 19 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bit 27 in C). In the C environment, it must be accessed through the full-word element.

Gate2[*i*].Chan[*j*].UVW

Description: U, V, and W flags input value

Range: 0 .. 7

Units: Bit field

Gate2[*i*].Chan[*j*].UVW contains the present logical state of the IC's U, V, and W flag inputs for the channel, latched on the most recent servo interrupt. Bit 2 (value 4) represents the state of the U flag; bit 1 (value 2) represents the state of the V flag; and bit 0 (value 1) represents the value of the W flag.

A value of 1 in a bit corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, the buffering external to the IC is non-inverting, so this applies to the voltages into the controller as well.

Gate2[*i*].Chan[*j*].UVW forms bits 20 – 22 of the full-word element **Gate2[*i*].Chan[*j*].Status** in the Script environment (bits 28 – 30 in C). In the C environment, it must be accessed through the full-word element.

Gate3[*i*]. (PMAC3-Style ASIC) Status Elements

The PMAC3-style DSPGATE3 machine-interface ASIC provides 4 channels of servo interface, plus general-purpose I/O and MACRO-ring interface. The status elements in this section permit the processor to access values resulting from the external inputs to this ASIC.

Gate3[*i*]. Multi-Channel Status Elements

The elements in this section are not channel-specific, and so do not need the **Chan[*j*]** sub-structure.

Gate3[*i*].ChipID

Description: DSPGATE3 IC identification number

Range: $0 .. $FFFFFFFF

Units: none

Gate3[*i*].ChipID contains the IC's identification number and revision number value. For the DSPGATE3 IC, this value is $7145300*n*, where *n* is the revision number. This element can be used to confirm that the IC is present and accessible by the processor, and to identify the revision being used.

Gate3[*i*].EEpromData[*k*]

Description: EEPROM data word

Range: $0 .. $FFFFFFFF

Units: none

Gate3[*i*].EEpromData[*k*] contains one of the 4 32-bit words of data received from a 128-bit EEPROM IC connected to the DSPGATE3 IC. The index ***k*** can be a value from 0 to 3. Up to 8 I^2C EEPROM ICs can be connected to a DSPGATE3 IC; one of these is addressed using the **Gate3[*i*].EEpromCtrl** element. The value **Gate3[*i*].EEpromData[*k*]** register does not necessarily give any indication which of the possible EEPROM ICs the data came from.

On power-up/reset, Power PMAC will use the DSPGATE3 IC's EEPROM interface to try to read all 8 of the possible EEPROM ICs connected to the ASIC. These EEPROMs contain data about the hardware configuration in which the ASIC is used. The data is used to set the values in status elements **Gate3[*i*].PartNum**, **Gate3[*i*].PartOpt0** through **Gate3[*i*].PartOpt7**, **Gate3[*i*].PartRev**, and **Gate3[*i*].PartType**.

Gate3[*i*].GpioOutData[*j*]

Description: IC general-purpose I/O bank output holding data values

Range: $0 .. $FFFFFFFF

Units: Bit field

Power-on default: $0

Gate3[*i*].GpioOutData[*j*] is a 32-bit value that represents the commanded output states of the 32 I/O points of I/O bank "*j*" for the IC. It is actually a holding register in memory for these outputs. In the Script environment, when a command writes to the actual I/O bank data register **Gate3[*i*].GpioData[*j*]**, Power PMAC will copy the value to this holding register as well.

Gate3[*i*].GpioOutData[*j*] is useful because the I/O bank pins can be set up so that reading the I/O bank data register itself may not reflect the state of the outputs. Therefore, it is better to use the holding value both to monitor the output values, and to use as a register starting value when setting or clearing particular bits of the register. Power PMAC does this when setting individual bits **Gate3[*i*].GpioData[*j*].*k*** in the actual I/O register of the IC.

Gate3[*i*].MacroInA[*j*][*k*]

Description: MACRO Bank A Node j Register k input data

Range: -2^{31} .. $2^{31}-1$

Units: User-determined

Gate3[*i*].MacroInA[*j*][*k*] is the input data register ***k*** of node ***j*** of MACRO bank A in the IC. The data register index ***k*** has a range of 0 to 3. The node index ***j*** has a range of 0 to 15. This bank has a master number on the MACRO ring that is set by bits 28 – 31 of saved setup element **Gate3[*i*].MacroEnableA**. The input data in all four registers for the node ***j*** is automatically received and latched every phase cycle if bit (***j*** + 8) of **Gate3[*i*].MacroEnableA** is set to 1.

In the original MACRO protocol, only the high 24 bits of **Gate3[*i*].MacroInA[*j*][0]** are received over the ring, and only the high 16 bits of **Gate3[*i*].MacroInA[*j*][1]**, **Gate3[*i*].MacroInA[*j*][2]**, and **Gate3[*i*].MacroInA[*j*][3]** are received over the ring. In the new MACRO2 protocol, all 32 bits of all 4 node registers are received over the ring.

When the node is used for automatic servo control, saved setup element **EncTable[*i*].pEnc** will probably be set to **Gate3[*i*].MacroInA[*j*][0].a**, and **Motor[*x*].pEncStatus** (and other associated address variables) will be probably be set to **Gate3[*i*].MacroInA[*j*][3].a**. In this case, automatic Power PMAC tasks will read from these registers, and in general, user application code does not need to read from these registers.

In the DSPGATE3 IC, these are separately addressed registers from the corresponding output values in **Gate3[*i*].MacroOutA**, so it is not possible to write to these elements. In the older DSPGATE2 ICs, the separate output and input values share a register.

You can query the value of particular range of bits of **Gate3[*i*].MacroInA[*j*][*k*]** using the syntax **Gate3[*i*].MacroInA[*j*][*k*].*{start}*.*{width}*** where ***{start}*** is the number of the low bit in the range, and ***{width}*** is the number of bits in the range. It is also possible to define an M-variable to a range of bits within the element using the "start" and "width" values this way, for example:

```
ptr OverPressure->Gate3[0].MacroInB[2][0].12.1
```

Gate3[*i*].MacroInB[*j*][*k*]

Description: MACRO Bank B Node j Register k input data

Range: -2^{31} .. $2^{31}-1$

Units: User-determined

Gate3[*i*].MacroInB[*j*][*k*] is the input data register ***k*** of node ***j*** of MACRO bank B in the IC. The data register index ***k*** has a range of 0 to 3. The node index ***j*** has a range of 0 to 15. This bank has a master number on the MACRO ring that is set by bits 28 – 31 of saved setup element **Gate3[*i*].MacroEnableB**. The input data in all four registers for the node ***j*** is automatically received and latched every phase cycle if bit (***j*** + 8) of **Gate3[*i*].MacroEnableB** is set to 1.

In the original MACRO protocol, only the high 24 bits of **Gate3[*i*].MacroInB[*j*][0]** are received over the ring, and only the high 16 bits of **Gate3[*i*].MacroInB[*j*][1]**, **Gate3[*i*].MacroInB[*j*][2]**, and **Gate3[*i*].MacroInB[*j*][3]** are received over the ring. In the new MACRO2 protocol, all 32 bits of all 4 node registers are received over the ring.

When the node is used for automatic servo control, saved setup element **EncTable[*i*].pEnc** will probably be set to **Gate3[*i*].MacroInB[*j*][0].a**, and **Motor[*x*].pEncStatus** (and other associated address variables) will be probably be set to **Gate3[*i*].MacroInB[*j*][3].a**. In this case, automatic Power PMAC tasks will read from these registers, and in general, user application code does not need to read from these registers.

In the DSPGATE3 IC, these are separately addressed registers from the corresponding output values in **Gate3[*i*].MacroOutB**, so it is not possible to write to these elements. In the older DSPGATE2 ICs, the separate output and input values share a register.

You can query the value of particular range of bits of **Gate3[*i*].MacroInB[*j*][*k*]** using the syntax **Gate3[*i*].MacroInB[*j*][*k*].*{start}*.*{width}*** where ***{start}*** is the number of the low bit in the range, and ***{width}*** is the number of bits in the range. It is also possible to define an M-variable to a range of bits within the element using the "start" and "width" values this way, for example:

```
ptr OverPressure->Gate3[0].MacroInB[2][0].12.1
```

Gate3[*i*].MacroError

Description: MACRO node error status

Range: $0 .. $FFFFFFFF

Units: Bit field

Gate3[*i*].MacroError contains information about which nodes, if any, have experienced an error in the data packet received over the MACRO ring in the most recent phase cycle. It is a 32-bit value, with one bit for each MACRO node supported by the IC. The bit is set to 0 if there was no

data reception error in the most recent cycle; it is set to 1 if there was a data reception error in the most recent cycle. The bit is always 0 if the node is not enabled.

Bits 0 to 15 represent the error status of Nodes 0 to 15, respectively, of MACRO Bank A in the IC. Bits 16 to 31 represent the error status of Nodes 0 to 15, respectively, of MACRO Bank B in the IC.

Gate3[*i*].MacroError only shows the presence or absence of a MACRO data error; the type of error, if any, can be found in the status portion of saved setup elements **Gate3[*i*].MacroModeA** and **Gate3[*i*].MacroModeB**.

Gate3[*i*].PartNum

Description: Hardware part number for DSPGATE3 IC board

Range: Non-negative integer

Units: Enumeration

Gate3[*i*].PartNum contains the six-digit part number of the board containing the DSPGATE3 ASIC. It is derived from an EEPROM IC connected to the ASIC, which is read automatically at power-up/reset. It will report as 0 if no ASIC with this index is present.

Part numbers presently existing are:

- ACC-24E3 604002
- ACC-5E3 604017
- ACC-5EP3 604035
- ACC-59E3 604027
- Power Brick 604030
- PowerClipper 604050

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate3PartData[*i*].Num**.

Gate3[*i*].PartOpt*n*

Description: Optional hardware configuration code for DSPGATE3 IC board

Range: Non-negative integer

Units: Enumeration

Gate3[*i*].PartOpt*n* contains a code indicating what hardware is connected to the DSPGATE3 IC. It is derived from an EEPROM IC connected to the ASIC, which is read automatically at power-

up/reset. The final digit ***n*** can take a value from 0 to 7, matching the number of the particular EEPROM the information was read from. The location of the EEPROM for each value of ***n*** is:

- 0: Base board
- 1: Feedback interface for first set of two channels
- 2: Output interface for first set of two channels
- 3: (*Reserved for future use*)
- 4: Core circuitry for second set of two channels (if separate)
- 5: Feedback interface for second set of two channels (if separate)
- 6: Output interface for second set of two channels (if separate)
- 7: (*Reserved for future use*)

In general, the value of **Gate3[*i*].PartOpt*n*** will be the same as the product code for the option in the price list part number. It reports as "nan" (not-a-number) if no ASIC with this index is present.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate3PartData[*i*].Opt*n***.

Gate3[*i*].PartRev

Description: Hardware revision number for DSPGATE3 IC board

Range: Non-negative integer

Units: Enumeration

Gate3[*i*].PartRev contains the revision number of the board containing the DSPGATE3 ASIC. It is derived from an EEPROM IC connected to the ASIC, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate3PartData[*i*].Rev**.

Gate3[*i*].PartType

Description: Hardware part type for DSPGATE3 IC board

Range: Non-negative integer

Units: Bit field

Gate3[*i*].PartType contains an integer representing the type(s) of hardware interface of the board containing the DSPGATE3 ASIC. It is derived from an EEPROM IC connected to the ASIC,

which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no ASIC with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present with the IC. It is possible that multiple interface types could be present.

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Analog I/O
- Bit 3 (value 8): Digital I/O

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **Gate3PartData[*i*].Type**.

Gate3[*i*]. Channel-Specific Status Elements

Each servo-interface channel in the ASIC has several read-only registers that contain information about the state of the inputs to the channel. The channel index ***j*** has a range of 0 to 3, corresponding to hardware channel numbers 1 to 4, respectively.

Gate3[*i*].Chan[*j*].ABC

Description: A, B, and C encoder channel input values

Range: 0 .. 7

Units: Bit field

Gate3[*i*].Chan[*j*].ABC contains the present logical state of the IC's A, B, and C encoder inputs for the channel, latched on the most recent servo interrupt. Bit 2 (value 4) represents the state of the C (index) channel; bit 1 (value 2) represents the state of the B channel; and bit 0 (value 1) represents the value of the A channel.

A value of 1 in a bit corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is a differential input, in which case, a value of 1 is attained when the "+" input has a higher voltage than the "-" input.

Gate3[*i*].Chan[*j*].ABC forms bits 4 – 6 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].ABPins

Description: Encoder A+, A-, B+, B- signal levels

Range: 0 .. 15

Units: Bit field

Gate3[*i*].Chan[*j*].ABPins contains the present logical state of the IC's A+, A-, B+, and B- encoder inputs for the channel, latched on the most recent phase interrupt. Bit 3 (value 8) represents the state of the B+ signal; Bit 2 (value 4) represents the state of the B- signal; bit 1 (value 2) represents the state of the A+ signal; and bit 0 (value 1) represents the value of the A- signal.

A value of 1 in a bit corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, these are the individual signals that are part of differential line-driver pairs from the encoder, taken from before the differential line receivers. These signals are the inputs to exclusive-OR gates in the ASIC that are used for "encoder loss" detection.

Gate3[*i*].Chan[*j*].ABPins forms bits 0 – 3 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].AdcAmp[*k*]

Description: Amplifier analog-to-digital converter value

Range: -2^{31} .. $2^{31}-1$

Units: 32-bit ADC LSBs

Gate3[*i*].Chan[*j*].AdcAmp[*k*] contains the value for the "amplifier" analog-to-digital converter for the IC channel, phase, and clock-cycle half that was latched on the most recent phase clock cycle. Each IC channel has 4 **AdcAmp** registers: 2 for the A phase, and 2 for the B phase. For each motor/amplifier phase, one is strobed on the rising edge of the phase clock, and one is strobed on the falling edge. The four elements and what they represent are:

- **AdcAmp[0]** A phase, strobed on rising edge of phase clock
- **AdcAmp[1]** B phase, strobed on rising edge of phase clock
- **AdcAmp[2]** A phase, strobed on falling edge of phase clock
- **AdcAmp[3]** B phase, strobed on falling edge of phase clock

The elements **AdcAmp[0]** and **AdcAmp[1]** are typically used for current feedback from the motor for digital current-loop closure in Power PMAC's "direct PWM" algorithms, because they represent the most recent possible values (1/2-phase cycle old). If the external ADC circuitry can multiplex the analog inputs based on the state of the phase clock signal, then the elements **AdcAmp[2]** and **AdcAmp[3]** can represent completely different values, such as bus voltage and temperature.

This is a 32-bit element, with the circuitry supporting a maximum of 24-bit ADCs. If the interface is configured correctly, the data for an *n*-bit ADC should be present in the high *n* bits of this 32-bit element. When used for current feedback in the Power PMAC's "direct-PWM" commutation algorithm, the true data must be present in the highest bits.

If the ADC has "header bits" preceding the true data, a proper setting of saved setup element **Gate1[*i*].AdcAmpHeaderBits** will cause the header data to be "rolled over" to the lowest bits of this element.

Gate3[*i*].Chan[*j*].AdcEnc[*k*]

Description: Encoder analog-to-digital converter value

Range: -2^{31} .. 2^{31} -1

Units: 32-bit ADC LSBs

Gate3[*i*].Chan[*j*].AdcEnc[*k*] contains the value for the "encoder" analog-to-digital converter for the IC channel, phase, and clock-cycle half that was latched on the most recent phase clock cycle. Each IC channel has 4 **AdcEnc** registers: 2 for the A phase, and 2 for the B phase. For each encoder phase, one is strobed on the rising edge of the phase clock, and one is strobed on the falling edge. The four elements and what they represent are:

- **AdcEnc[0]** A phase, strobed on rising edge of phase clock
- **AdcEnc[1]** B phase, strobed on rising edge of phase clock
- **AdcEnc[2]** A phase, strobed on falling edge of phase clock
- **AdcEnc[3]** B phase, strobed on falling edge of phase clock

The elements **AdcEnc[0]** and **AdcEnc[1]** are typically used for "sine" and "cosine" signals, respectively, from sinusoidal encoders and resolvers, because they represent the most recent possible values (1/2-phase cycle old). If the external ADC circuitry can multiplex the analog inputs based on the state of the phase clock signal, then the elements **AdcAmp[2]** and **AdcAmp[3]** can represent different values, such as analog commutation tracks on a sinusoidal encoder, or a geared-down "multi-turn" resolver.

The values in **Gate3[*i*].Chan[*j*].AdcEnc[0]** and **Gate3[*i*].Chan[*j*].AdcEnc[1]** are automatically used in the generation of **Gate3[*i*].Chan[*j*].Atan** and **Gate3[*i*].Chan[*j*].SumOfSquares**, and, if saved setup element **Gate3[*i*].Chan[*j*].AtanEna** is set to 1, will be used automatically for the sub-cycle position information in **Gate3[*i*].Chan[*j*].PhaseCapt** and **Gate3[*i*].Chan[*j*].ServoCapt**.

This is a 32-bit element, with the circuitry supporting a maximum of 24-bit ADCs. If the interface is configured correctly, the data for an *n*-bit ADC should be present in the high *n* bits of this 32-bit element. When used for position feedback from sinusoidal encoders or resolvers, the true data must be present in the highest bits.

If the ADC has "header bits" preceding the true data, a proper setting of saved setup element **Gate1[*i*].AdcEncHeaderBits** will cause the header data to be "rolled over" to the lowest bits of this element.

Gate3[*i*].Chan[*j*].Atan

Description: Encoder sine/cosine angle

Range: 0 .. 65,535

Units: 1/65,536 cycle

Gate3[*i*].Chan[*j*].Atan contains the angle value automatically computed by the ASIC from the encoder primary analog-to-digital converter values. These values can be found in **Gate3[*i*].Chan[*j*].AdcEnc[0]** (sine) and **Gate3[*i*].Chan[*j*].AdcEnc[1]** (cosine). These values come from the A/D converters that are strobed on the rising edge of the phase clock and whose data is ready on the falling edge of the phase clock (which provides the processor interrupt). The **Atan** value is ready immediately on the phase interrupt (provided there is enough time for 25 ADC clock cycles and 2 additional microseconds in half a phase cycle).

Before computing the arctangent value, it adds the values in saved setup elements **Gate3[*i*].Chan[*j*].AdcOffset[0]** and **Gate3[*i*].Chan[*j*]. AdcOffset[1]** to the respective inputs. (These terms allow the user to compensate for analog bias.) Note that this is a "two-argument" arctangent calculation (often called ATAN2) that provides a range in the result of a full 360°. It utilizes the high 16 bits of the ADC registers as inputs, regardless of the actual resolution of the converters. The number of bits of this element that contain meaningful information depends on the resolution of the converters and the noise level of the analog circuitry. When this value is used for servo feedback, a software tracking filter in the encoder conversion table is often employed to reduce the resulting noise levels.

Gate3[*i*].Chan[*j*].Atan is commonly used as the resolver position value when using the ASIC in a resolver-to-digital conversion process.

If saved setup element **Gate3[*i*].Chan[*j*].AtanEna** is set to 1, the upper 14 bits of this 16-bit value are also copied to the low 14 bits of **Gate3[*i*].Chan[*j*].PhaseCapt** and **Gate3[*i*].Chan[*j*].ServoCapt**, where they represent the fractional-cycle value for sinusoidal encoders. If **AtanEna** is set to 1, **Atan** will contain the processed fractional result of the combination with the whole-count data, which can differ in "direction" from the raw arctangent data.

Gate3[*i*].Chan[*j*].Atan forms bits 16 – 31 of the full-word element **Gate3[*i*].Chan[*j*].AtanSumOfSqr**. C programs and functions must use the full-word element.

Gate3[*i*].Chan[*j*].AtanSumOfSqr

Description: Encoder sine/cosine angle and magnitude register

Range: 32-bit integer

Units: (See individual components)

Gate3[*i*].Chan[*j*].AtanSumOfSqr is a 32-bit read-only element containing two 16-bit elements with information automatically calculated from the primary encoder sine and cosine analog-to-digital converter values for the channel.

Gate3[*i*].Chan[*j*].Atan (bits 16 – 31) contains the angle computed from the arctangent of the two values. **Gate3[*i*].Chan[*j*].SumOfSquares** (bits 0 – 15) contains the "magnitude" of the signal computed as the sum of the squares of the input values.

Script programs and commands will probably use the partial-word elements, but C programs and functions must use this full-word element, and mask and shift as necessary to isolate the individual components

For details on the individual components, refer to the descriptions of these partial-word elements.

Gate3[*i*].Chan[*j*].CountError

Description: Encoder-decode counting error

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].CountError contains the present logical state of the IC's "encoder count error" status flag for the channel. A value of 1 indicates that an illegal quadrature transition has occurred, with both the A and B-channel input states from the encoder changing in the same encoder sample-clock (SCLK) cycle. When this illegal transition occurs, this flag is set and latched to indicate that subsequent reads of any encoder counter value will provide an erroneous value.

Gate3[*i*].Chan[*j*].CountError is forced to 0 during the power-up/reset of the Power PMAC. The user can clear this status bit by writing a 0 value to the bit.

Gate3[*i*].Chan[*j*].CountError is bit 30 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].DemuxInvalid

Description: Invalid demultipexing of UVW from quadrature

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].DemuxInvalid contains the present logical state of the IC's "invalid Hall demultiplexing" flag for the channel. If saved setup element **Gate3[*i*].Chan[*j*].IndexDemuxEna** is set to 1, the IC will derive its U, V, and W flag states from the encoder "C" signal, demultiplexing them and the index pulse based on the 4 quadrature states. After power-up/reset, before this circuitry has seen all 4 quadrature states, **Gate3[*i*].Chan[*j*].DemuxInvalid** is set to 1

to indicate that it cannot provide a valid full set of these flag states. Once it has seen all 4 quadrature states, **Gate3[*i*].Chan[*j*].DemuxInvalid** is set to 0.

Gate3[*i*].Chan[*j*].DemuxInvalid is bit 19 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].Equ

Description: Position-compare internal state value

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].Equ contains the present logical state of the IC's "EQU" position-compare internal state for the channel. This internal state can be used in the generation of the position-compare output signals for any or all of the IC's channels, as specified by saved setup elements **Gate3[*i*].Chan[*j*].EquOutMask** and **Gate3[*i*].Chan[*j*].EquOutPol**.

It is not possible to use this element to set the internal state directly. It is necessary to use the 2-bit non-saved setup element **Gate3[*i*].Chan[*j*].EquWrite** to force an output value. The internal value will automatically change as the encoder counter value passes the positions set in **Gate3[*i*].Chan[*j*].CompA** and **Gate3[*i*].Chan[*j*].CompB**.

Gate3[*i*].Chan[*j*].Equ is bit 24 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].EquOut

Description: Position-compare combined output state

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].EquOut contains the present logical state of the IC's "EQU" position-compare flag output for the channel. A value of 1 corresponds to a high voltage out of the IC; a value of 0 corresponds to a low voltage out of the IC. On most Delta Tau hardware, this signal controls a sinking output driver, so a 1 state turns on the output driver, conducting current and pulling the output voltage low.

The channel's output is formed by the logical combination of the internal compare states of (potentially) all 4 channels on the IC, as specified by saved setup elements **Gate3[*i*].Chan[*j*].EquOutMask** and **Gate3[*i*].Chan[*j*].EquOutPol**.

It is not possible to use this element to set the output state directly. It is necessary to use the 2-bit non-saved setup element **Gate3[*i*].Chan[*j*].EquWrite** to force an output value. The flag value will automatically change as the encoder counter value passes the positions set in **Gate3[*i*].Chan[*j*].CompA** and **Gate3[*i*].Chan[*j*].CompB**.

Gate3[*i*].Chan[*j*].EquOut is bit 25 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].Fault

Description: Amplifier fault flag input value

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].Fault contains the present logical state of the IC's FAULT (amplifier fault) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On much Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator. This can also be a differential input, in which case a value of 1 is attained when the FAULT+ input has a higher voltage than the FAULT- input.

Note that this element simply represents the input level into the IC; either a 0 or a 1 state can be interpreted as a fault from the amplifier, depending on the setting of **Motor[*x*].AmpFaultLevel**.

This hardware input element is distinct from the motor software status element **Motor[*x*].AmpFault**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate3[*i*].Chan[*j*].Fault is bit 7 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].HallState

Description: Captured Hall position state value

Range: 0 .. 7

Units: Enumeration

Gate3[*i*].Chan[*j*].HallState contains the most recently captured position edge in the Hall cycle from the channel's U, V, and W flag inputs, when the channel is set up to capture encoder position on any U, V, or W transition (by **Gate3[*i*].Chan[*j*].CaptCtrl** bits 0 and 1 set to 0, and **Gate3[*i*].Chan[*j*].GatedIndexSel** set to 1).

Valid position edge values in the 6-state Hall cycle range from 0 to 5. The following table shows how these 6 values correspond to the legal input values of the U, V, and W flag inputs:

Gate3[*i*].Chan[*j*].HallState	0	1	2	3	4	5
Gate3[*i*].Chan[*j*].UVW	5	1	3	2	6	4	5
U flag input state	1	0	0	0	1	1	1
V flag input state	0	0	1	1	1	0	0
W flag input state	1	1	1	0	0	0	1

Power PMAC considers the standard zero-degree point in the Hall cycle to be the transition edge 1 (between UVW values 1 and 3). If this is located at the zero-degree point in the commutation cycle, no offset is required from sensor position to commutation angle.

Before any captured transitions have occurred, or if there is a transition to an illegal UVW flag input state, **Gate3[*i*].Chan[*j*].HallState** will report a value of 7.

Gate3[*i*].Chan[*j*].HallState forms bits 16 – 18 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].HomeCapt

Description: Encoder count value latched on external trigger

Range: -2^{31} .. $2^{31}-1$

Units: 1/256 encoder counts

Gate3[*i*].Chan[*j*].HomeCapt contains the extended hardware encoder count value for the IC channel that was latched on the most recent external trigger. It is a 32-bit value, with the high 24 bits having units of encoder counts as determined by the encoder decode specified by **Gate3[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction.

The trigger state is selected by saved setup elements **Gate3[*i*].Chan[*j*].CaptCtrl**, **Gate3[*i*].Chan[*j*].CaptFlagSel** and **Gate3[*i*].Chan[*j*].CaptFlagChan**.

The contents of the low 8 bits, representing fractional count position, are dependent on the setting of **Gate3[*i*].Chan[*j*].TimerMode**. If **TimerMode** is set to the default value of 0, the low 8 bits are computed by the IC's "1/T" sub-count extension algorithm that estimates fractional count values based on count timing. If **TimerMode** is set to a non-zero value, the low 8 bits are fixed at a value representing ½-count. The setting of **Gate3[*i*].Chan[*j*].AtanEna** does not affect the contents of these low bits in the **HomeCapt** element, as it does for the channel's **PhaseCapt** and **ServoCapt** elements.

Note that if **TimerMode** is set to 0, **Gate3[*i*].Chan[*j*].TimerB** element contains the same trigger-latched position information, but with only the high 20 bits containing whole-count information, and the low 12 bits containing 1/T-extended fractional-count position.

The act of reading **Gate3[*i*].Chan[*j*].HomeCapt** re-arms the trigger circuitry for the next capture. Because level triggering is used, if the triggering inputs are still in the specified state, an immediate re-triggering will occur, latching a (potentially) new count value into this element.

Gate3[*i*].Chan[*j*].HomeFlag

Description: HOME flag input value

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].HomeFlag contains the present logical state of the IC's HOME flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

Gate3[*i*].Chan[*j*].HomeFlag is bit 8 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].LossCapt

Description: Encoder-loss latched error

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].LossStatus indicates whether the quadrature encoder-detection circuitry in the ASIC has sensed the loss of a valid encoder signal or not. It is 0 if it has always detected a valid signal; it is 1 if it does not see a valid signal ("encoder loss"). The circuitry employs exclusive-or gates on the differential signal pairs, so this detection is only valid for encoders with differential line-driver outputs.

This bit is automatically set to 0 during the power-on/reset of the Power PMAC. The user can clear this status bit by writing a 0 value to the bit.

Gate3[*i*].Chan[*j*].LossCapt is bit 29 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].LossStatus

Description: Encoder-loss present error

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].LossStatus indicates whether the quadrature encoder-detection circuitry in the ASIC presently senses a valid encoder signal or not. It is 0 if it sees a valid signal; it is 1 if it does

not see a valid signal ("encoder loss"). The circuitry employs exclusive-or gates on the differential signal pairs, so this detection is only valid for encoders with differential line-driver outputs.

Gate3[*i*].Chan[*j*].LossStatus is bit 28 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].MinusLimit

Description: MLIM flag input value

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].MinusLimit contains the present logical state of the IC's MLIM (negative overtravel limit) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

This hardware input element is distinct from the motor software status element **Motor[*x*].MinusLimit**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate3[*i*].Chan[*j*].MinusLimit is bit 10 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].PhaseCapt

Description: Encoder count value latched on phase interrupt

Range: -2^{31} .. $2^{31}-1$

Units: 1/256 encoder counts (digital)
1/4096 encoder counts (analog)

Gate3[*i*].Chan[*j*].PhaseCapt contains the extended hardware encoder count value for the IC channel that was latched on the most recent phase interrupt. It is a 32-bit value, with the high 20 or 24 bits having units of encoder counts as determined by the encoder decode specified by **Gate3[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction. Because the instantaneous counter value is not accessible by the processor, this element contains the most direct accessible representation of the present count value.

If saved setup element **Gate3[*i*].Chan[*j*].AtanEna** is set to the default value of 0, the high 24 bits of this element come from the latched encoder counter, and the contents of the low 8 bits, representing fractional count position, are dependent on the setting of **Gate3[*i*].Chan[*j*].TimerMode**. If **TimerMode** is set to the default value of 0, the low 8 bits are

computed by the IC's "1/T" sub-count extension algorithm that estimates fractional count values based on count timing. If **TimerMode** is set to a non-zero value, the low 8 bits are fixed at a value representing ½-count.

If **Gate3[*i*].Chan[*j*].AtanEna** is set to 1, the high 20 bits of this element come from the latched encoder counter, and the low 12 bits come from the value in **Gate3[*i*].Chan[*j*].Atan**, which contains fractional count information automatically computed as the arctangent of the sine and cosine inputs.

Gate3[*i*].Chan[*j*].PlusLimit

Description: PLIM flag input value

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].PlusLimit contains the present logical state of the IC's PLIM (positive overtravel limit) flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

This hardware input element is distinct from the motor software status element **Motor[*x*].PlusLimit**, which shows the motor state that *may* result from this input, if the motor is set up to use this input.

Gate3[*i*].Chan[*j*].PlusLimit is bit 9 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].PosCapt

Description: Position-capture trigger flags

Range: 0 .. 3

Units: Bit field

Gate3[*i*].Chan[*j*].PosCapt contains the present state of the channel's two trigger flags. Bit 1 (value 2) is the state of the channel's "general" trigger flag. It is set whenever the trigger condition specified by **Gate3[*i*].Chan[*j*].CaptCtrl**, **Gate3[*i*].Chan[*j*].CaptFlagSel**, and **Gate3[*i*].Chan[*j*].CaptFlagChan** has occurred. Bit 0 (value 1) is the state of the "gated-index" trigger flag. It is set whenever a trigger condition that includes the encoder's index channel "gated" to 1 quadrature state wide (with **Gate3[*i*].Chan[*j*].CaptCtrl** bit 0 = 1 and **Gate3[*i*].Chan[*j*].GatedIndexSel** = 1) has occurred. This trigger is mainly used to confirm that the correct number of counts have accumulated between consecutive index pulses.

Both flags are automatically cleared by the act of reading the register containing the encoder position that was captured at the instant of the trigger: **Gate3[*i*].Chan[*j*].HomeCapt**. Both flags

are level-triggered, not edge-triggered, so if the selected inputs are still in the specified trigger state when the flags are cleared, there will be an immediate re-triggering.

Gate3[*i*].Chan[*j*].PosCapt forms bits 20 – 21 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].SerialEncDataA

Description: Serial encoder primary position feedback data

Range: 0 .. 2^{32}-1

Units: Encoder position LSBs

Gate3[*i*].Chan[*j*].SerialEncDataA contains the low-order position feedback data received from the channel's serial encoder interface in the most recent phase or servo clock cycle. This element can hold up to 32 bits of encoder data from one of several different serial encoder protocols. If there are more than 32 bits of data received from the encoder, the higher-order data will be found in **Gate3[*i*].Chan[*j*].SerialEncDataB**.

Quadrature Counter

If the serial encoder interface is not enabled (**Gate3[*i*].Chan[*j*].SerialEncEna** = 0), a digital quadrature encoder can be connected to the serial encoder pins, and this 32-bit element will contain the extended hardware encoder count value for the IC channel that was latched on the most recent servo interrupt, with the high 24 bits having units of encoder counts, and the low 8 bits computed by the IC's "1/T" sub-count extension algorithm that estimates fractional count values based on count timing. The count is always generated using "times-4" decode (4 counter per line).

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

Whole-Count Position *Fractional-Count Position*

SPI Protocol

The DSPGATE3 IC can support up to 32 bits of position data and 12 bits of status data from an SPI encoder, as specified by the channel command word. There is no general specification as to how many of the position bits are "single-turn" data and how many (if any) are "multi-turn". There is no general specification as to what particular status bits mean.

For an SPI encoder, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
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

Single/Multi-Turn Position

Bits P*n* represent the bits of single-turn and multi-turn position.

SSI Protocol

The DSPGATE3 IC can support up to 32 bits of position data and a parity error bit from an SSI encoder, as specified by the channel command word. There is no specification as to how many of the position bits are "single-turn" data and how many (if any) are "multi-turn".

For an SSI encoder, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
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

Single/Multi-Turn Position

Bits P*n* represent the bits of single-turn and multi-turn position.

EnDat2.1/2.2 Protocol

The DSPGATE3 IC can support up to 48 bits of position data from an EnDat encoder, as specified by the channel command word. There is no specification as to how many of the position bits are “single-turn” data and how many (if any) are “multi-turn”.

For an EnDat encoder, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
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

Single/Multi-Turn Position

Bits P*n* represent the bits of single-turn and multi-turn position.

Hiperface Protocol

The DSPGATE3 IC supports 32 bits of position data and 12 bits of status data from a Hiperface encoder.

For a Hiperface encoder, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
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

Single/Multi-Turn Position

Bits P*n* represent the bits of single-turn and multi-turn position.

Yaskawa Sigma I Protocol

For an absolute Sigma I encoder, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
D	D	D	D	D	D	D	D	C	C	C	C	C	C	C	C	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

Multi-Turn Position

Bits A*n* represent the bits of the ASCII code for the “one’s digit” of the turns count; bits B*n* represent bits of the “ten’s digit”; bits C*n* represent bits of the “hundred’s digit”; bits D*n* represent bits of the “thousand’s digit”.

For each of the numeric ASCII digits, the numeric value of the digit can be obtained by subtracting 48 ($30) from the value of the ASCII code.

Yaskawa Sigma II/III/V Protocol

For an absolute Sigma II encoder with 17 bits per revolution, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
M	M	M	M	M	M	M	M	M	M	M	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	-	-	-	-
10	9	8	7	6	5	4	3	2	1	0	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
Multi-Turn Position											*Single-Turn Position*																				

Bits S*n* represent the bits of single-turn position; bits M*n* represent the bits of multi-turn position ("turns count"). Overall position can be read "seamlessly" as a combination of single-turn and multi-turn position values.

For an absolute Sigma III or V encoder with 20 bits per revolution, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
M	M	M	M	M	M	M	M	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	-	-	-	-
7	6	5	4	3	2	1	0	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
Multi-Turn Position								*Single-Turn Position*																							

The meaning of the bits is the same as for the 17-bit-per-revolution absolute encoder.

For an incremental Sigma II encoder with 17 bits per revolution, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
C	C	C	C	C	C	C	C	-	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	-	-	U	V	W	Z
13	12	11	10	9	8	7	6		16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
Compensation Position								*Single-Turn Position*																				*Hall/Index*			

Bits S*n* represent the bits of single-turn position; bits C*n* represent the bits of compensation position (the position found at the index). Once an index has been found (alarm code bit A6 goes to 0), the compensation position can be subtracted from the single-turn position data to get the properly referenced position. Notice that the 11 bits of the compensation position match up with the high 11 bits of the 17-bit single-turn position.

U, V, and W represent the 3-phase "Hall" commutation sensor signals. These can be used for absolute power-on phase referencing as if they were true Hall sensors. Z represents the encoder index (zero) pulse state.

Tamagawa FA-Coder Protocol

For a Tamagawa FA-Coder encoder with 17 bits per revolution and 16 bits of "turns count", **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
M	M	M	M	M	M	M	M	0	0	0	0	0	0	0	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
7	6	5	4	3	2	1	0								16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multi-Turn Position															*Single-Turn Position*																

Bits S*n* represent the bits of single-turn position. Overall position can be read as a combination of single-turn and multi-turn position values. However, note that these values are not directly adjacent to each other, so care must be taken in combining them.

Panasonic Protocol

For a Panasonic serial encoder with 17 bits per revolution and 16 bits of "turns count", **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows for multi-turn position reporting:

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
M	M	M	M	M	M	M	M	0	0	0	0	0	0	0	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
7	6	5	4	3	2	1	0								16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multi-Turn Position																*Single-Turn Position*															

Bits S*n* represent the bits of single-turn position; bits M*n* represent the bits of multi-turn position ("turns count"). Overall position can be read as a combination of single-turn and multi-turn position values. However, note that these values are not directly adjacent to each other, so care must be taken in combining them.

For a Panasonic serial encoder with 17 bits per revolution and 16 bits of "turns count", **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows for single-turn position reporting with alarm code:

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
ID	ID	ID	ID	ID	ID	ID	ID	0	0	0	0	0	0	0	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
7	6	5	4	3	2	1	0								16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Encoder ID Code																*Single-Turn Position*															

Bits S*n* represent the bits of single-turn position. Bits ID0 – ID7 of the encoder ID code are fixed at a value of $11.

Mitutoyo Protocol

For the Mitutoyo encoder, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
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
Absolute Position																															

Bits P*n* represent the bits of absolute position.

Kawasaki Protocol

For the Kawasaki encoder, **Gate3[*i*].Chan[*j*].SerialEncDataA** is configured as follows:

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
M	M	M	M	M	M	M	M	M	M	M	M	C	C	S	S	S	S	S	S	S	S	S	S	S	S	S	S	I	I	I	I
11	10	9	8	7	6	5	4	3	2	1	0	1	0	13	12	11	10	9	8	7	6	5	4	3	2	1	0	3	2	1	0
Multi-Turn Position												*Correction*				*Single-Turn Position*												*Interp Pos*			

Bits I*n* represent bits of interpolated position. Bits S*n* represent the bits of single-turn position. Bits C*n* represent bits of the multi-turn correction data. Bits M*n* represent bits of multi-turn position.

Gate3[*i*].Chan[*j*].SerialEncDataB

Description: Serial encoder secondary position feedback data

Range: 0 .. 2^{32}-1

Units: 2^{32} * encoder position LSBs

Gate3[*i*].Chan[*j*].SerialEncDataB contains the high-order position feedback data, if any exists, from the channel's serial encoder feedback, plus protocol-dependent status and error information. The lowest 32 bits (bits 0 – 31) of the position data are found in **Gate3[*i*].Chan[*j*].SerialEncDataA**. If the feedback has more than 32 bits of data, this data will be found in **Gate3[*i*].Chan[*j*].SerialEncDataB**, starting at the low end (bit 32 of the data will be found in bit 0 of this element, etc.).

The status and error information is found starting at the high end of this element. The amount and meaning of this information is dependent on the particular protocol.

Code	Protocol	Status and Error Information
1	SPI	Bits 31 – 20: Encoder dependent
2	SSI	Bit 31: Parity error
3	EnDat	Bit 31: Timeout error Bit 30: CRC error Bit 29: Error bit
4	Hiperface	Bit 31: Timeout error Bit 30: Checksum error Bit 29: Parity error Bit 28: Error bit Bits 27 – 20: Error code
5	Sigma I	Bit 31: Timeout error Bit 30: Parity error
6	Sigma II	Bit 31: Timeout error Bit 30: CRC error Bit 29: Coding error Bit 28: (*reserved*) Bits 27 – 20: Alarm code
7	Tamagawa	Bit 31: Timeout error Bit 30: CRC error Bits 29 – 28: (*reserved*) Bits 27 – 24: Status field
8	Panasonic	Bit 31: Timeout error Bit 30: CRC error Bits 29 – 28: (*reserved*) Bits 27 – 24: Status field
9	Mitutoyo	Bit 31: Timeout error Bit 30: CRC error Bits 29 – 28: (*reserved*) Bits 27 – 24: Status field
10	Kawasaki	Bit 31: Timeout error Bit 30: CRC error Bit 29: Coding error Bits 28 – 27: (*reserved*) Bit 26: Interpolation error Bit 25: Position error Bit 24: Busy flag
-	(AB quadrature)	Bit 31: Count error

When the encoder is used for position feedback, any error bits in this register can be used by the Encoder Conversion Table entry processing the feedback if the **type** = 12 method is used to reject that cycle's position value as possibly erroneous and to substitute a computed value extrapolated from previous cycles' data.

Quadrature Encoder

For a quadrature encoder, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
E0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

CE

Bit E0 represents the count error bit, set when there has been an illegal quadrature transition.

SPI Protocol

For an SPI encoder, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
F11	F10	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Status Field

Bits F*n* represent bits of the status field.

SSI Protocol

For an SSI encoder, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
E0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

PE

Bit E0 represents the parity error bit.

EnDat2.1/2.2 Protocol

For an EnDat encoder, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
E2	E1	E0	-	-	-	-	-	-	-	-	-	-	-	-	-	P47	P46	P45	P44	P43	P42	P41	P40	P39	P38	P37	P36	P35	P34	P33	P32

TE CE EB

Bits P*n* represent the bits of single-turn and multi-turn position. Bits E*n* represent the error bits (E0 is an error reported by the encoder, E1 is a CRC error detected by the IC, and E2 is a timeout error detected by the IC).

Hiperface Protocol

The DSPGATE3 IC supports 32 bits of position data and 12 bits of status data from a Hiperface encoder.

For a Hiperface encoder, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
E	E	E	E	C	C	C	C	C	C	C	C																				
3	2	1	0	7	6	5	4	3	2	1	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Error Bits				*Error Code*																											

Bits C*n* represent bits of the encoder's error code (only reported on status request). Bits E*n* represent error bits (E0 is an error reported by the encoder, E1 is a parity error detected by the IC, E2 is a checksum error detected by the IC, and E3 is a timeout error).

Yaskawa Sigma I Protocol

The Yaskawa Sigma I absolute encoder protocol provides absolute turns count position data in ASCII text format on specific request. For an absolute Sigma I encoder, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
F	F							P	P	P	P	P	P	P	P	S	S	S	S	S	S	S	S	E	E	E	E	E	E	E	E
1	0	-	-	-	-	-	-	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Err Bits								*"P"*								*Sign (+/-)*								*Multi-Turn Position*							

Bits E*n* represent bits of the "ten-thousand's digit" of the turns count; bits S*n* represent bits of the ASCII code for the plus or minus sign; bits P*n* represent bits of the ASCII code for the letter "P". Bits F*n* represent bits of the error field (F0 is a parity error; F1 is a timeout error).

For each of the five numeric ASCII digits, the numeric value of the digit can be obtained by subtracting 48 ($30) from the value of the ASCII code.

Yaskawa Sigma II/III/V Protocol

For an absolute Sigma II encoder with 17 bits per revolution, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
E	E	E		A	A	A	A	A	A	A	A	T	T	T	T	T	T	T										M	M	M	M	M
2	1	0	-	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	T0	-	-	-	-	-	-	-		15	14	13	12	11
Error Bits				*Alarm Code*								*Temperature*																*Multi-Turn Position*				

Bits M*n* represent the bits of multi-turn position ("turns count"). Overall position can be read "seamlessly" as a combination of single-turn and multi-turn position values. Bits T*n* represents the bits of the reported temperature in degrees C (if provided).

Bits A*n* represent bits of the alarm code for the absolute encoder, with each bit having the following meaning:

- A0: Battery-backed turns data lost
- A1: Power-on error self-detected
- A2: Battery low-voltage warning
- A3: Absolute position error

- A4: Over-speed error
- A5: Over-temperature error
- A6: Encoder reset in progress
- A7: (reserved)

Bits E*n* represent error bits detected by the IC, with each bit having the following meaning:

- E0: Coding error
- E1: CRC error
- E2: Timeout error

For an absolute Sigma III/V encoder with 20 bits per revolution,
Gate3[*i*].Chan[*j*].SerialEncDataB is configured as follows:

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
E	E	E	-	A	A	A	A	A	A	A	A	T	T	T	T	T	T	T	T	-	-	-	-	M	M	M	M	M	M	M	M
2	1	0		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0					15	14	13	12	11	10	9	8
Error Bits				*Alarm Code*								*Temperature*												*Multi-Turn Position*							

The meaning of the bits is the same as for the 17-bit-per-revolution absolute encoder.

For an incremental Sigma II encoder with 17 bits per revolution,
Gate3[*i*].Chan[*j*].SerialEncDataB is configured as follows:

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
E	E	E	-	A	A	A	A	A	A	A	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	C	C	C
2	1	0		7	6	5	4	3	2	1	0																		16	15	14
Error Bits				*Alarm Code*																									*Compensation Pos*		

Bits C*n* represent the bits of compensation position (the position found at the index). Once an index has been found (alarm code bit A6 goes to 0), the compensation position can be subtracted from the single-turn position data to get the properly referenced position. Notice that the 11 bits of the compensation position match up with the high 11 bits of the 17-bit single-turn position.

Bits A*n* represent bits of the alarm code for the incremental encoder, with each bit having the following meaning:

- A0: (reserved)
- A1: Power-on error self-detected
- A2: (reserved)
- A3: Revolution count (index to index) incorrect
- A4: (reserved)
- A5: (reserved)
- A6: Position reference (index) not found yet
- A7: (reserved)

Bits E*n* represent error bits detected by the IC, with each bit having the following meaning:

- E0: Coding error
- E1: CRC error
- E2: Timeout error

Tamagawa FA-Coder Protocol

For a Tamagawa FA-Coder encoder with 17 bits per revolution and 16 bits of "turns count", **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
E	E	-	-	F	F	F	F	A	A	A	A	A	A	A	A	-	-	-	-	-	-	-	-	M	M	M	M	M	M	M	M
1	0			7	6	5	4	7	6	5	4	3	2	1	0									15	14	13	12	11	10	9	8
Err Bits				*Status Field*				*Alarm Code*																*Multi-Turn Position*							

Bits M*n* represent the bits of multi-turn position ("turns count"). Overall position can be read as a combination of single-turn and multi-turn position values. However, note that these values are not directly adjacent to each other, so care must be taken in combining them. Bits A*n* represent bits of the alarm code, bits F*n* represent bits of the status field; and bits E*n* represent error bits. (E0 is CRC error, and E1 is timeout error.)

Panasonic Protocol

For a Panasonic serial encoder with 17 bits per revolution and 16 bits of "turns count", **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows for multi-turn position reporting:

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
E	E	-	-	F	F	F	F	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	M	M	M	M	M	M	M	M
1	0			7	6	5	4																	15	14	13	12	11	10	9	8
Err Bits				*Status Field*																				*Multi-Turn Position*							

Bits M*n* represent the bits of multi-turn position ("turns count"). Overall position can be read as a combination of single-turn and multi-turn position values. However, note that these values are not directly adjacent to each other, so care must be taken in combining them. Bits F*n* represent bits of the status field; and bits E*n* represent error bits. (E0 is CRC error, and E1 is timeout error.)

For a Panasonic serial encoder with 17 bits per revolution and 16 bits of "turns count, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows for single-turn position reporting with alarm code:

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
E	E	-	-	F	F	F	F	-	-	-	-	-	-	-	-	A	A	A	A	A	A	A	A	ID	ID	ID	ID	ID	ID	ID	ID
1	0			7	6	5	4									7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
Err Bits				*Status Field*												*Alarm Code*								*Encoder ID Code*							

Bits ID8 – ID15 represent bits of the encoder ID code. Bits A0 – A7 represent the alarm code reported by the processor, with each bit having the following meaning:

- A0: Overspeed error
- A1: Full resolution status; = 1 when over 100rpm and reporting reduced resolution
- A2: Count error
- A3: Counter overflow
- A4: (reserved)
- A5: Multi-revolution error
- A6: System undervoltage error (< 2.5V)
- A7: Battery low (< 3.1V)

Mitutoyo Protocol

For the Mitutoyo encoder, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
E1	E0	-	-	F3	F2	F1	F0	A7	A6	A5	A4	A3	A2	A1	A0	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0	-	-	-	-	-	-	-	-
Err Bits				*Status Field*				*Alarm Code*								*Encoder ID*															

Bits ID*n* represent bits of the encoder ID (only provided on specific request). Bits A*n* represent bits of the alarm code reported from the encoder, with each bit having the following meaning:

- A0: Initialization error
- A1: Mismatch of optical and capacitive sensors
- A2: Optical sensor error
- A3: Capacitive sensor error
- A4: CPU error (AT303); CPU/ROM/RAM error (AT503)
- A5: EEPROM error
- A6: ROM/RAM error (AT303); Communication error (AT503)
- A7: Overspeed error

Bits F*n* represent bits of the status field reported from the encoder, with each bit having the following meaning:

- F0: Fatal (unrecoverable) encoder error
- F1: (reserved)
- F2: Illegal command code from controller
- F3: (reserved)

Bits E*n* represent bits of the error code detected by the IC, with each bit having the following meaning:

- E0: CRC error
- E1: timeout error

Kawasaki Protocol

For the Kawasaki encoder, **Gate3[*i*].Chan[*j*].SerialEncDataB** is configured as follows:

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
E2	E1	E0	-	-	A2	A1	A0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	M15	M14	M13	M12
Error Bits					*Alarm Code*																							*Multi-Turn Pos*			

Bits M*n* represent bits of multi-turn position. Bits A*n* represent bits of the alarm code reported from the encoder (A0 is interpolator error, A1 is absolute track error, A2 is the busy flag). Bits E*n* represent bits of the error code detected by the IC (E0 is coding error, E1 is CRC error, E2 is timeout error).

Gate3[*i*].Chan[*j*].ServoCapt

Description: Encoder count value latched on servo interrupt

Range: -2^{31} .. $2^{31}-1$

Units: 1/256 encoder counts (digital)
1/4096 encoder counts (analog)

Gate3[*i*].Chan[*j*].ServoCapt contains the extended hardware encoder count value for the IC channel that was latched on the most recent servo interrupt. It is a 32-bit value, with the high 20 or 24 bits having units of encoder counts as determined by the encoder decode specified by **Gate3[*i*].Chan[*j*].EncCtrl**. The counter is set to 0 on power-on/reset and counts from there, so is always referenced to the position at power-on/reset. It can roll over in either direction.

If saved setup element **Gate3[*i*].Chan[*j*].AtanEna** is set to the default value of 0, the high 24 bits of this element come from the latched encoder counter, and the contents of the low 8 bits, representing fractional count position, are dependent on the setting of **Gate3[*i*].Chan[*j*].TimerMode**. If **TimerMode** is set to the default value of 0, the low 8 bits are computed by the IC's "1/T" sub-count extension algorithm that estimates fractional count values based on count timing. If **TimerMode** is set to a non-zero value, the low 8 bits are fixed at a value representing ½-count.

Note that if **TimerMode** is set to 0, **Gate3[*i*].Chan[*j*].TimerA** element contains the same servo-latched position information, but with only the high 20 bits containing whole-count information, and the low 12 bits containing 1/T-extended fractional-count position.

If **Gate3[*i*].Chan[*j*].AtanEna** is set to 1, the high 20 bits of this element come from the latched encoder counter, and the low 12 bits come from the value in **Gate3[*i*].Chan[*j*].Atan**, which contains fractional count information automatically computed as the arctangent of the sine and cosine inputs.

Gate3[*i*].Chan[*j*].SosError

Description: Sine/cosine "sum of squares" error

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].SosError contains the present logical state of the IC's "sum of squares" error status flag for the channel. A value of 1 indicates that the sum of the squares of the encoder ADC inputs (from **Gate3[*i*].Chan[*j*].AdcEnc[0]** and **Gate3[*i*].Chan[*j*].AdcEnc[1]**) for the channel, intended to be from the "sine" and "cosine" channels of a sinusoidal encoder or resolver, is not large enough to be considered a valid signal. A value of 0 indicates a large enough sum of squares to constitute a valid signal.

The IC automatically computes the sum of squares of these two signed ADC registers each phase cycle and puts the result in the unsigned 16-bit element **Gate3[*i*].Chan[*j*].SumOfSquares**. If the highest 4 bits of this element are all 0 in the phase cycle, indicating that the value is less than 1/16 of the maximum possible, so the sine and cosine values are less than ¼ of the maximum possible, **Gate3[*i*].Chan[*j*].SosError** is set to 1. If any of the high 4 of 16 bits is 1 in the cycle, **Gate3[*i*].Chan[*j*].SosError** is set to 0.

Gate3[*i*].Chan[*j*].SosError is bit 31 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].Status

Description: Channel status word

Range: $0 .. $FFFFFFFF

Units: Bit field

Gate3[*i*].Chan[*j*].Status is a 32-bit status word that displays the contents of the IC's hardware status register for the channel. It contains many individual status bits and elements. The individual status bits and elements are accessible as separate elements in Script programs and commands, but not in C programs and functions.

The following table provides a list of these status bits in the word:

Bit #	Hex Value	Element Name: Gate3[*i*].Chan[*j*]. *{element}*	Description
31	$80000000	**SosError**	Sine/cosine "sum of squares" error
30	$40000000	**CountError**	Encoder decode counting error
29	$20000000	**LossCapt**	Encoder loss latched error
28	$10000000	**LossStatus**	Encoder loss present error
26 – 27	$C000000	**-**	*(Reserved for future use)*
25	$2000000	**EquOut**	Position-compare combined output state
24	$1000000	**Equ**	Position-compare internal state
23	$800000	**-**	*(Reserved for future use)*
22	$400000	**TrigState**	Present combined trigger input state
20 – 21	$300000	**PosCapt**	Position-captured trigger flags
19	$80000	**DemuxInvalid**	Invalid demultiplexing of UVW from quadrature
16 – 18	$70000	**HallState**	Captured Hall state
15	$8000	**T**	T flag input value
12 – 14	$7000	**UVW**	U, V, and W flag input values
11	$800	**UserFlag**	USER flag input value
10	$400	**MinusLimit**	MLIM flag input value
9	$200	**PlusLimit**	PLIM flag input value
8	$100	**HomeFlag**	HOME flag input value
7	$80	**Fault**	Amplifier fault flag input value
4 – 6	$70	**ABC**	Encoder A, B, and C channel input values
0 – 3	$F	**ABPins**	Encoder A+, A-, B+, B- Signal levels

Each element is described in more detail in its own individual specification.

Gate3[*i*].Chan[*j*].SumOfSquares

Description: Encoder sine/cosine magnitude value

Range: 0 .. 65,535

Units: (16-bit ADC units)^2 / 32,768

Gate3[*i*].Chan[*j*].SumOfSquares contains the magnitude value automatically computed by the ASIC from the encoder primary analog-to-digital converter values. These values can be found in **Gate3[*i*].Chan[*j*].AdcEnc[0]** (sine) and **Gate3[*i*].Chan[*j*].AdcEnc[1]** (cosine). These values come from the A/D converters that are strobed on the rising edge of the phase clock and whose data is ready on the falling edge of the phase clock (which provides the processor interrupt). The **SumOfSquares** value is ready immediately on the phase interrupt (provided there is enough time for 25 ADC clock cycles and 2 additional microseconds in half a phase cycle).

The calculation producing this value utilizes the high 16 bits of the ADC registers as inputs, regardless of the actual resolution of the converters, treating them as signed quantities. First it adds the values in saved setup elements **Gate3[*i*].Chan[*j*].AdcOffset[0]** and **Gate3[*i*].Chan[*j*]. AdcOffset[1]** to the respective inputs. (These terms allow the user to compensate for analog bias.) Next, it squares each corrected value to a 32-bit result, adds the two values together, then uses the high 16 bits of the result.

If the sine and cosine ADC values use their full range (reaching +/-32,768 at their peak in the high 16 bits of their register), the **SumOfSquares** value will also be about 32,768. A value of **SumOfSquares** higher than 32,768 is indicative of a saturation problem on the analog inputs.

Gate3[*i*].Chan[*j*].SumOfSquares is primarily used as a diagnostic value in the setup and debugging of a sine/cosine feedback device such as a sinusoidal encoder or resolver. Note that if all of the highest 4 bits of this 16-bit value are 0, the channel status bit **Gate3[*i*].Chan[*j*].SosError** is set to 1. This bit can be used as an indicator of sensor loss.

Gate3[*i*].Chan[*j*].SumOfSquares forms bits 0 –15 of the full-word element **Gate3[*i*].Chan[*j*].AtanSumOfSqr**. C programs and functions must use the full-word element.

Gate3[*i*].Chan[*j*].T

Description: T flag input value

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].T contains the present logical state of the IC's T flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, the buffering external to the IC is non-inverting, so this applies to the voltages into the controller as well.

Gate3[*i*].Chan[*j*].T is bit 15 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].TimerA

Description: Primary timer-based value

Range: -2^{31} .. $2^{31}-1$

Units: (Mode dependent)

Gate3[*i*].Chan[*j*].TimerA contains timer-based feedback information in one of four modes.

If saved setup element **Gate3[*i*].Chan[*j*].TimerMode** is set to the default value of 0, the timer circuitry is set up for "1/T" count extension, and the 32-bit **TimerA** element contains the servo-latched encoder count value, with the high 20 bits containing whole-count information, and the low 12 bits containing 1/T-extended fractional-count position. It is equivalent to the value in **Gate3[*i*].Chan[*j*].ServoCapt**, except with 4 more bits of fractional-count resolution (and 4 fewer bits of full-count range).

If **TimerMode** is set to 1, the timer circuitry is set up for MLDT "pulse echo" mode, and the high 24 bits of the **TimerA** element contain a value proportional to the time between the output pulse that occurred at the start of the previous servo cycle and the receipt of the "echo pulse" from the MLDT sensor. This time, in units of 600 MHz clock cycles (1.66667 nanoseconds), is directly proportional to the distance of the MDLT moving magnet from the base.

If **TimerMode** is set to 2, the timer circuitry is set up for trigger input timing, and the high 24 bits of the **TimerA** element contain the time between the last servo interrupt and the receipt of the specified trigger condition. This is in units of cycles of the encoder sample clock (SCLK), whose period is set by **Gate3[*i*].EncClockDiv**. This value can be divided by the value of the **TimerB** element to find the fraction of the servo cycle time where the trigger occurred.

If **TimerMode** is set to 3, the timer circuitry is set up for internal pulse counting, and the high 24 bits of the **TimerA** element contain the accumulated pulse count value from the channel's D-phase pulse-frequency-modulation (PFM) circuitry latched on the most recent servo interrupt. There is no sub-count extension in the low 8 bits, which all read as 0. This value can be used as the simulated feedback when creating pulse-and-direction outputs to a traditional stepper drive, while leaving the encoder counter circuitry available for a true encoder, as for position confirmation.

Gate3[*i*].Chan[*j*].TimerB

Description: Secondary timer-based value

Range: -2^{31} .. $2^{31}-1$

Units: (Mode dependent)

Gate3[*i*].Chan[*j*].TimerB contains timer-based feedback information in one of four modes.

If saved setup element **Gate3[*i*].Chan[*j*].TimerMode** is set to the default value of 0, the timer circuitry is set up for "1/T" count extension, and the 32-bit **TimerB** element contains the trigger-latched encoder count value, with the high 20 bits containing whole-count information, and the low 12 bits containing 1/T-extended fractional-count position. It is equivalent to the value in

Gate3[*i*].Chan[*j*].HomeCapt, except with 4 more bits of fractional-count resolution (and 4 fewer bits of full-count range).

If **TimerMode** is set to 1, the timer circuitry is set up for MLDT "pulse echo" mode, and the high 24 bits of the **TimerB** element contain a value proportional to the time between the output pulse that occurred at the start of the previous servo cycle and the one that occurred at the start of this servo cycle. This time, in units of 600 MHz clock cycles (1.66667 nanoseconds), can be used to confirm the scaling of the **TimerA** element, which contains the measurement of interest in this mode.

If **TimerMode** is set to 2, the timer circuitry is set up for trigger input timing, and the high 24 bits of the **TimerB** element contain the time between consecutive servo interrupts. This is in units of cycles of the encoder sample clock (SCLK), whose period is set by **Gate3[*i*].EncClockDiv**. This value can be divided into the value of the **TimerA** element to find the fraction of the servo cycle time where the trigger occurred.

If **TimerMode** is set to 3, the timer circuitry is set up for internal pulse counting, and the high 24 bits of the **TimerA** element contain the accumulated pulse count value from the channel's D-phase pulse-frequency-modulation (PFM) circuitry latched on the most recent trigger. There is no sub-count extension in the low 8 bits, which all read as 0. This value can be used as the simulated position-capture feedback when creating pulse-and-direction outputs to a traditional stepper drive, while leaving the encoder counter circuitry available for a true encoder, as for position confirmation.

Gate3[*i*].Chan[*j*].TrigState

Description: Position-capture present input trigger state

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].TrigState contains the present state of the channel's input trigger signal for the position-capture function. The trigger signal is a logical combination of the channel's index channel and input flags as specified by **Gate3[*i*].Chan[*j*].CaptCtrl**, **Gate3[*i*].Chan[*j*].CaptFlagSel**, and **Gate3[*i*].Chan[*j*].CaptFlagChan**.

A value of 1 indicates the triggering state; a value of 0 indicates the non-triggering state. The actual triggering for position capture, which sets one or both bits of **Gate3[*i*].Chan[*j*].PosCapt** to 1, occurs at the instant that this value changes from 0 to 1, or when the captured position register **Gate3[*i*].Chan[*j*].HomeCapt** is read while this element is 1, causing an automatic re-triggering (as the circuitry is level-triggered).

Both flags are automatically cleared by the act of reading the register containing the encoder position that was captured at the instant of the trigger: **Gate3[*i*].Chan[*j*].HomeCapt**. Both flags are level-triggered, not edge-triggered, so if the selected inputs are still in the specified trigger state when the flags are cleared, there will be an immediate re-triggering.

Gate3[*i*].Chan[*j*].TrigState is bit 22 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].UserFlag

Description: USER flag input value

Range: 0 .. 1

Units: Boolean

Gate3[*i*].Chan[*j*].UserFlag contains the present logical state of the IC's USER flag input for the channel, latched on the most recent servo interrupt. A value of 1 corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, this is an opto-isolated input, and a value of 0 is attained when the flag is "closed", conducting current through the optical isolator.

Gate3[*i*].Chan[*j*].UserFlag is bit 11 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

Gate3[*i*].Chan[*j*].UVW

Description: U, V, and W flags input value

Range: 0 .. 7

Units: Bit field

Gate3[*i*].Chan[*j*].UVW contains the present logical state of the IC's U, V, and W flag inputs for the channel, latched on the most recent servo interrupt. Bit 2 (value 4) represents the state of the U flag; bit 1 (value 2) represents the state of the V flag; and bit 0 (value 1) represents the value of the W flag.

A value of 1 in a bit corresponds to a high voltage into the IC; a value of 0 corresponds to a low voltage into the IC. On most Delta Tau hardware, the buffering external to the IC is non-inverting, so this applies to the voltages into the controller as well.

Gate3[*i*].Chan[*j*].UVW forms bits 12 – 14 of the full-word element **Gate3[*i*].Chan[*j*].Status**. In the C environment, it must be accessed through the full-word element.

GateIo[*i*]. Status Data Structure Elements

This section documents the status elements for the **GateIo[*i*]** data structure. Note that all of the data registers in this structure can be written to, even if the matching lines are used as inputs, and so are documented as unsaved setup elements.

GateIo[*i*].PartNum

Description: Hardware part number for IOGATE IC board

Range: Non-negative integer

Units: Enumeration

GateIo[*i*].PartNum contains the six-digit part number of the board containing the IOGATE ASIC. It is derived from an EEPROM IC connected to the ASIC, which is read automatically at power-up/reset. It will report as 0 if no ASIC with this index is present.

Part numbers presently existing are:

- ACC-11C: 603603
- ACC-14E: 603474
- ACC-65E: 603575
- ACC-66E: 603576
- ACC-67E: 603577
- ACC-68E: 603595

For auto-identified I/O accessories, this is purely a status element. However, for I/O accessories that cannot be auto-identified by the Power PMAC CPU, the user can write the part number to this element so that the data structure can be used for these accessories. (**GateIo[*i*].PartType** must be set to 8 as well.) This functionality is presently supported for the ACC-11E UMAC I/O board and the ACC-84S Clipper serial-encoder interface board. It is a new feature in V2.0 firmware, released 1st quarter 2015.

The part numbers presently supported for this feature are:

- ACC-11E: 603307
- ACC-84S: 603936

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **CardIOPartData[*i*].Num**.

GateIo[*i*].PartOpt

Description: Optional hardware configuration code for IOGATE IC board

Range: Non-negative integer

Units: Enumeration

GateIo[*i*].PartOpt contains a code indicating what optional hardware is present on the board containing the IOGATE IC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. It will report as "nan" (not-a-number) if no ASIC with this index is present.

For auto-identified I/O accessories, this is purely a status element. However, for I/O accessories that cannot be auto-identified by the Power PMAC CPU, the user can write the part option to this element. It is a new feature in V2.0 firmware, released 1st quarter 2015. However, the write capability for this element does not add any significant functionality.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **CardIOPartData[*i*].Opt**.

GateIo[*i*].PartRev

Description: Hardware revision number for IOGATE IC board

Range: Non-negative integer

Units: Enumeration

GateIo[*i*].PartRev contains the revision number of the board containing the IOGATE ASIC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. This number corresponds to the last digit of the 3-digit part number suffix for the board.

For auto-identified I/O accessories, this is purely a status element. However, for I/O accessories that cannot be auto-identified by the Power PMAC CPU, the user can write the part revision to this element. It is a new feature in V2.0 firmware, released 1st quarter 2015. However, the write capability for this element does not add any significant functionality.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **CardIOPartData[*i*].Rev**.

GateIo[*i*].PartType

Description: Hardware part type for IOGATE IC board

Range: Non-negative integer

Units: Bit field

GateIo[*i*].PartType contains an integer representing the type(s) of hardware interface of the board containing the IOGATE ASIC. It is derived from an identification IC associated with the ASIC, which is read automatically at power-up/reset. It reports as "nan" (not-a-number) if no ASIC with this index is present.

Each bit specifies one type of interface that could be present. The bit is set if that type of interface is present with the IC. It is possible that multiple interface types could be present.

- Bit 0 (value 1): Servo
- Bit 1 (value 2): MACRO
- Bit 2 (value 4): Digital I/O
- Bit 3 (value 8): Analog I/O

Presently, the value of **PartType** for all IOGATE IC boards is 4 (digital I/O interface only).

For auto-identified I/O accessories, this is purely a status element. However, for I/O accessories that cannot be auto-identified by the Power PMAC CPU, the user can write the part type to this element so that the data structure can be used for these accessories. (**GateIo[*i*].PartNum** must be set to the six-digit part number as well.) This functionality is presently supported for the ACC-11E UMAC I/O board and the ACC-84S Clipper serial-encoder interface board. It is a new feature in V2.0 firmware, released 1st quarter 2015.

This element is not actually a hardware register in the ASIC; it is a software register associated with the ASIC. It is used mainly by system setup utilities in the IDE.

In C, this element should be accessed as **CardIOPartData[*i*].Type**.

Gather. Data Gathering Status Elements

The status elements in this section are set automatically by Power PMAC in the process of setting up and executing the data gathering function. Most commonly, the Integrated Development Environment (IDE) plotting and tuning functions read these elements automatically.

Gather.Ddata[*j*]

Description: Servo-interrupt data gathering "double" gathered data point

Range: Double-precision floating-point

Units: User set

Gather.Ddata[*j*] is the "***j***th" 32-bit word representing the starting word of a double-precision (64-bit, 8-byte) floating-point data point in the most recent data set gathered under the servo interrupt. The index ***j*** specifies the number of 4-byte data blocks from the beginning of the gathered data buffer. For double-precision data points, only even-numbered values of ***j*** should be used. There can be, and usually are, multiple data points per sample.

Gather.Ddata[*j*] accesses the same registers as **Gather.Fdata[*j*]** and **Gather.Fdata[*j+1*]**, as **Gather.Idata[*j*]** and **Gather.Idata[*j+1*]**, and as **Gather.Udata[*j*]** and **Gather.Udata[*j+1*]**. If **Gather.Ddata[*j*]** is used to read data that was not gathered as a double-precision floating-point value, a nonsensical value can result.

Gather.Fdata[*j*]

Description: Servo-interrupt data gathering "float" gathered data point

Range: Single-precision floating-point

Units: User set

Gather.Fdata[*j*] is the "***j***th" single-precision (32-bit, 4-byte) floating-point data point in the most recent data set gathered under the servo interrupt. The index ***j*** specifies the number of 4-byte data blocks from the beginning of the gathered data buffer. There can be, and usually are, multiple data points per sample.

Gather.Fdata[*j*] accesses the same registers as **Gather.Idata[*j*]**, as **Gather.Udata[*j*]**, and as **Gather.Ddata[*j*]** (for even values of ***j***) or as **Gather.Ddata[*j-1*]** (for odd values of ***j***). If **Gather.Fdata[*j*]** is used to read data that was not gathered as a single-precision floating-point value, a nonsensical value can result.

Gather.Idata[*j*]

Description: Servo-interrupt data gathering signed-integer gathered data point

Range: 0 .. 2^{32}-1

Units: User set

Gather.Idata[*j*] is the "***j***th" signed long-integer (32-bit, 4-byte) data point in the most recent data set gathered under the servo interrupt. The index ***j*** specifies the number of 4-byte data blocks from the beginning of the gathered data buffer. There can be, and usually are, multiple data points per sample.

Gather.Idata[*j*] accesses the same registers as **Gather.Fdata[*j*]**, as **Gather.Udata[*j*]**, and as **Gather.Ddata[*j*]** (for even values of ***j***) or as **Gather.Ddata[*j-1*]** (for odd values of ***j***). If **Gather.Idata[*j*]** is used to read data that was not gathered as a signed long-integer value, a nonsensical value can result.

Gather.Index

Description: Servo-interrupt data gathering sample index

Range: Non-negative integer

Units: Sample

Gather.Index contains the value of the sample index of the next servo-interrupt data gathering sample. Each time a new sample is gathered (every **Gather.Period** servo cycles when enabled), the value of **Gather. Index** is incremented by 1. When the gather buffer "rolls over" while gathering in "indefinite" mode (**Gather.Enable** = 3), **Index** is reset to 0.

Gather.LineLength

Description: Servo-interrupt data gathering words per sample

Range: Non-negative integer

Units: 32-bit words

Gather.LineLength contains the number of 32-bit words of memory required to store each sample in the servo-interrupt data gathering function. It is calculated automatically as soon as **Gather.Enable** is set to a value greater than 0, based on the values of **Gather.Items** and **Gather.Type[*i*]** (where ***i*** = 0 to **Gather.Items** - 1).

Gather.MaxLines

Description: Servo-interrupt data gathering maximum possible samples

Range: Non-negative integer

Units: Samples

Gather.MaxLines contains the number of samples of gathered data that can be stored in the available memory by the servo-interrupt data gathering function. It is calculated automatically as soon as **Gather.Enable** is set to a value greater than 0, based on the size of the data gathering buffer and the values of **Gather.Items** and **Gather.Type[*i*]** (where ***i*** = 0 to **Gather.Items** - 1).

Gather.PhaseDdata[*j*]

Description: Phase-interrupt data gathering "double" gathered data point

Range: Double-precision floating-point

Units: User set

Gather.PhaseDdata[*j*] is the "***j***th" 32-bit word representing the starting word of a double-precision (64-bit, 8-byte) floating-point data point in the most recent data set gathered under the phase interrupt. The index ***j*** specifies the number of 4-byte data blocks from the beginning of the gathered data buffer. For double-precision data points, only even-numbered values of ***j*** should be used. There can be, and usually are, multiple data points per sample.

Gather.PhaseDdata[*j*] accesses the same registers as **Gather.PhaseFdata[*j*]** and **Gather.PhaseFdata[*j+1*]**, as **Gather.PhaseIdata[*j*]** and **Gather.PhaseIdata[*j+1*]**, and as **Gather.PhaseUdata[*j*]** and **Gather.PhaseUdata[*j+1*]**. If **Gather.PhaseDdata[*j*]** is used to read data that was not gathered as a double-precision floating-point value, a nonsensical value can result.

Gather.PhaseFdata[*j*]

Description: Phase-interrupt data gathering "float" gathered data point

Range: Single-precision floating-point

Units: User set

Gather.PhaseFdata[*j*] is the "***j***th" single-precision (32-bit, 4-byte) floating-point data point in the most recent data set gathered under the phase interrupt. The index ***j*** specifies the number of 4-byte data blocks from the beginning of the gathered data buffer. There can be, and usually are, multiple data points per sample.

Gather.PhaseFdata[*j*] accesses the same registers as **Gather. PhaseIdata[*j*]**, as **Gather.PhaseUdata[*j*]**, and as **Gather.PhaseDdata[*j*]** (for even values of ***j***) or as **Gather.PhaseDdata[*j-1*]** (for odd values of ***j***). If **Gather.PhaseFdata[*j*]** is used to read data that was not gathered as a single-precision floating-point value, a nonsensical value can result.

Gather.PhaseIdata[*j*]

Description: Phase-interrupt data gathering signed-integer gathered data point

Range: 0 .. 2^{32}-1

Units: User set

Gather.PhaseIdata[*j*] is the "***j***th" signed long-integer (32-bit, 4-byte) data point in the most recent data set gathered under the phase interrupt. The index ***j*** specifies the number of 4-byte data blocks from the beginning of the gathered data buffer. There can be, and usually are, multiple data points per sample.

Gather.PhaseIdata[*j*] accesses the same registers as **Gather.PhaseFdata[*j*]**, as **Gather.PhaseUdata[*j*]**, and as **Gather.PhaseDdata[*j*]** (for even values of ***j***) or as **Gather.PhaseDdata[*j-1*]** (for odd values of ***j***). If **Gather.PhaseIdata[*j*]** is used to read data that was not gathered as a signed long-integer value, a nonsensical value can result.

Gather.PhaseIndex

Description: Phase-interrupt data gathering sample index

Range: Non-negative integer

Units: Sample

Gather.PhaseIndex contains the value of the sample index of the next phase-interrupt data gathering sample. Each time a new sample is gathered (every **Gather.PhasePeriod** phase cycles when enabled), the value of **Gather.PhaseIndex** is incremented by 1. When the gather buffer "rolls over" while gathering in "indefinite" mode (**Gather.PhaseEnable** = 3), **PhaseIndex** is reset to 0.

Gather.PhaseLineLength

Description: Phase-interrupt data gathering words per sample

Range: Non-negative integer

Units: 32-bit words

Gather.PhaseLineLength contains the number of 32-bit words of memory required to store each sample in the phase-interrupt data gathering function. It is calculated automatically as soon as **Gather.PhaseEnable** is set to a value greater than 0, based on the values of **Gather.PhaseItems** and **Gather.PhaseType[*i*]** (where ***i*** = 0 to **Gather.PhaseItems** - 1).

Gather.PhaseMaxLines

Description: Phase-interrupt data gathering maximum possible samples

Range: Non-negative integer

Units: Samples

Gather.PhaseMaxLines contains the number of samples of gathered data that can be stored in the available memory by the servo-interrupt data gathering function. It is calculated automatically as soon as **Gather.PhaseEnable** is set to a value greater than 0, based on the size of the data gathering buffer and the values of **Gather.PhaseItems** and **Gather.PhaseType[*i*]** (where ***i*** = 0 to **Gather.PhaseItems** - 1).

Gather.PhaseSamples

Description: Servo-interrupt data gathering number of samples

Range: Non-negative integer

Units: Samples

Gather.PhaseSamples contains the number of individual time samples in the present servo-interrupt data gathering buffer. Each time a new sample is gathered (every **Gather.PhasePeriod** servo cycles when enabled), the value of **Gather.PhaseSamples** is incremented by 1. Unlike the similar **Gather.PhaseIndex**, **Gather.PhaseSamples** does *not* reset to 1 when the buffer rolls over in indefinite gathering mode (**Gather.PhaseEnable** = 3).

Gather.PhaseUdata[*j*]

Description: Phase-interrupt data gathering unsigned-integer gathered data point

Range: -2^{31} .. $2^{31}-1$

Units: User set

Gather.PhaseUdata[*j*] is the "***j***th" unsigned long-integer (32-bit, 4-byte) data point in the most recent data set gathered under the servo interrupt. The index ***j*** specifies the number of 4-byte data blocks from the beginning of the gathered data buffer. There can be, and usually are, multiple data points per sample.

Gather.PhaseUdata[*j*] accesses the same registers as **Gather.PhaseFdata[*j*]**, as **Gather.PhaseIdata[*j*]**, and as **Gather.PhaseDdata[*j*]** (for even values of ***j***) or as **Gather.PhaseDdata[*j-1*]** (for odd values of ***j***). If **Gather.PhaseUdata[*j*]** is used to read data that was not gathered as an unsigned long-integer value, a nonsensical value can result.

Gather.Samples

Description: Servo-interrupt data gathering number of samples

Range: Non-negative integer

Units: Samples

Gather.Samples contains the number of individual time samples in the present servo-interrupt data gathering buffer. Each time a new sample is gathered (every **Gather.Period** servo cycles when enabled), the value of **Gather.Samples** is incremented by 1. Unlike the similar **Gather.Index**, **Gather.Samples** does *not* reset to 1 when the buffer rolls over in indefinite gathering mode (**Gather.Enable** = 3).

Gather.Udata[*j*]

Description: Servo-interrupt data gathering unsigned-integer gathered data point

Range: -2^{31} .. $2^{31}-1$

Units: User set

Gather.Udata[*j*] is the "***j***th" unsigned long-integer (32-bit, 4-byte) data point in the most recent data set gathered under the servo interrupt. The index ***j*** specifies the number of 4-byte data blocks from the beginning of the gathered data buffer. There can be, and usually are, multiple data points per sample.

Gather.Udata[*j*] accesses the same registers as **Gather.Fdata[*j*]**, as **Gather.Idata[*j*]**, and as **Gather.Ddata[*j*]** (for even values of ***j***) or as **Gather.Ddata[*j-1*]** (for odd values of ***j***). If **Gather.Udata[*j*]** is used to read data that was not gathered as an unsigned long-integer value, a nonsensical value can result.

Ldata. Status Data Structure Elements

Each coordinate system, PLC program, and communications thread has its own **Ldata** (local data) data structure. From within the executing program or thread, the elements in this structure can be accessed directly as **Ldata.*{element}***. Externally, they must be accessed using the higher-level structure – **Coord[*x*].Ldata.*{element}*** or **Plc[*i*].Ldata.*{element}***.

Ldata.Cdata[*i*]

Description: Axis calculated move target position or velocity

Range: floating-point

Units: Axis position or velocity units

Ldata.Cdata[*i*] contains the most recent commanded axis move target position (***i*** = 0 to 31) or axis move target velocity (***i*** = 0 to 31) calculated from an axis move command by Power PMAC for the axis specified by the index. These local data elements are only used within a **Coord[*x*]** data structure.

For positions, the index ***i*** for each axis is:

Axis	Index	Axis	Index	Axis	Index	Axis	Index
A	0	Z	8	HH	16	SS	24
B	1	AA	9	LL	17	TT	25
C	2	BB	10	MM	18	UU	26
U	3	CC	11	NN	19	VV	27
V	4	DD	12	OO	20	WW	28
W	5	EE	13	PP	21	XX	29
X	6	FF	14	QQ	22	YY	30
Y	7	GG	15	RR	23	ZZ	31

For velocities, the index ***i*** for each axis is:

Axis	Index	Axis	Index	Axis	Index	Axis	Index
A	32	Z	40	HH	48	SS	56
B	33	AA	41	LL	49	TT	57
C	34	BB	42	MM	50	UU	58
U	35	CC	43	NN	51	VV	59
V	36	DD	44	OO	52	WW	60
W	37	EE	45	PP	53	XX	61
X	38	FF	46	QQ	54	YY	62
Y	39	GG	47	RR	55	ZZ	63

Ldata.CmdCount

Description: Number of text commands issued

Range: Non-negative integer

Units: Commands

Ldata.CmdCount is incremented by 1 for each text command issued with a **cmd"*{text}*"** statement in the coordinate system, PLC program, or communications thread. It is incremented whether or not the text command was valid. It can be used within a program to verify that the text command has actually been issued.

Ldata.CmdCount is automatically set to 0 on power-on/reset. If the user does not write to it, it contains the number of text commands issued since power-on reset.

Ldata.CmdStatus

Description: Status of most recently issued text command

Range: Integer

Units: Enumeration

Ldata.CmdStatus reflects the result of the text command most recently issued with a **cmd"*{text}*"** statement in the coordinate system, PLC program, or communications thread. It is automatically set by Power PMAC to 0 if the command is successfully executed, or to a negative value denoting the error code if there is an error in the command.

The magnitude of a negative value in **Ldata.CmdStatus** corresponds to the error number reported in a text response to an invalid external command. These error numbers are documented in the Command Syntax Summary chapter of the Software Reference Manual.

Many users will find it valuable to set **Ldata.CmdStatus** to a positive value before issuing the text command. This permits the user's program logic to distinguish between the states of non-completion (> 0), successful completion (= 0), and unsuccessful completion (< 0).

Specific values of **Ldata.CmdStatus** denoting errors are:

- -20: Illegal command syntax
- -21 Illegal command parameter
- -22 Reference to non-existent program
- -23 Out-of-range command
- -24 Out-of-order command
- -25 Invalid number in command
- -26 Invalid range in command
- -31 Compilation error
- -32 Break points set
- -33 Buffer in use
- -34 Buffer full

- -35 Invalid label
- -36 Invalid line number
- -37 Invalid break point
- -38 Program running
- -39 Not ready to run
- -40 Buffer not defined
- -41 Buffer already defined
- -42 No motors defined in coordinate system
- -43 Motor not closed loop
- -44 Motor not phased
- -45 Motor not activated
- -46 Motor(s) jogged out of paused position
- -47 Servo request active

Ldata.Lindex

Description: Base L-variable stack index for executing routine

Range: 0 .. 16,383

Units: Index

Ldata.Lindex contains the base index on the overall L-variable stack for the presently executing program or subprogram. That is, variable L0 for the executing routine is L(**Ldata.Lindex**) of the overall stack, L1 for the executing routine is L(**Ldata.Lindex**+1), and so on. Each time a subprogram is called with a **`call`** command or a subroutine is called with a **`callsub`** command, the value of **Ldata.Lindex** is increased by the value of **Ldata.Lsize** for the calling routine.

Ldata.Lsize

Description: L-variable stack offset for executing routine

Range: 0 .. 16,383

Units: Index

Ldata.Lsize contains the L-variable stack offset for the presently executing program or subprogram. Pass/return local variable **R*i*** for the routine is equivalent to internally used **L(Ldata.Lsize** + ***i***) for the routine. If the routine calls a subprogram with a **`call`** command or a subroutine with a **`callsub`** command, the value of **Ldata.Lindex** is increased by the value of **Ldata.Lsize** for the calling routine. This makes **L0** of the called routine equivalent to **L(Ldata.Lsize)** of the calling routine.

The value of **Ldata.Lsize** for the program or subprogram is set by the declared value of ***{stack offset}*** in the **`open`** command that prepares the program buffer for entry. If no value is declared, a value of 256 is automatically used. If the program is downloaded through the IDE's

Project Manager, a value equivalent to the number of local variables declared within the program is used.

Ldata.Status

Description: Script program execution status

Range: 0 .. 255

Units: Bit field

Ldata.Status contains information about the program execution engine status for the coordinate system or PLC program structure. This 8-bit element is divided into two 4-bit sections.

The low 4 bits (**Status** & $0F) contain a code signifying whether a program is running or not, and if not, why not. The following code values are supported:

- = 0: Program running
- = 1: Program stopped on quit or end of program
- = 2: Program stopped on breakpoint
- = 3: Program stopped on single step
- = 4: Program stopped on illegal script operation code
- = 5: Program stopped on execution stack overflow
- = 6: Program stopped on local-variable stack overflow
- = 7: Program stopped on call to non-existent subroutine or subprogram
- = 8: Program stopped on “not assigned” error
- = 9: Program stopped on inverse-kinematic missing equations
- = 10: Program stopped on synchronous variable buffer overflow
- = 11-15: (*Reserved for future use*)

The high 4 bits provide information about how the program is executing:

- Bit 4 (**Status** & $10) = 1 when running in single-step mode
- Bit 5 (**Status** & $20) = 1 when running (continuous or single-step)
- Bit 6 (**Status** & $40) = 1 for motion program, = 0 for PLC program
- Bit 7 (**Status** & $80) = 1 for “pmatch” execution at start of program

Ldata.SystemCmdCount

Description: Number of system text commands issued

Range: Non-negative integer

Units: Commands

Ldata.SystemCmdCount is incremented by 1 for each text command issued with a **`system"{text}"`** statement in the coordinate system, PLC program, or communications

thread. It is incremented whether or not the text command was valid. It can be used within a program to verify that the text command has actually been issued.

Ldata.SystemCmdCount is automatically set to 0 on power-on/reset. If the user does not write to it, it contains the number of text commands issued since power-on reset.

Ldata.SystemCmdStatus

Description: Status of most recently issued system text command

Range: Integer

Units: Enumeration

Ldata.SystemCmdStatus reflects the result of the text command most recently issued with a **`system"{text}"`** statement in the coordinate system, PLC program, or communications thread. It is automatically set by Power PMAC to 0 if the command is successfully executed, or to a negative value denoting the error code if there is an error in the command.

Many users will find it valuable to set **Ldata.SystemCmdStatus** to a positive value before issuing the text command. This permits the user's program logic to distinguish between the states of non-completion (> 0), successful completion (= 0), and unsuccessful completion (< 0).

Macro. Status Data Structure Elements

The **Macro** data structure status elements provide information as to the configuration and state of the MACRO ring.

Macro.ICs

Description: Number of effective MACRO ICs detected

Range: 0 .. 15

Units: ICs

Macro.ICs contains the number of "effective" MACRO ICs that were automatically detected at power-on/reset in the Power PMAC system. Each PMAC2-style "DSPGATE2" IC counts as 1 IC. Each PMAC3-style "DSPGATE3" IC counts as 2 effective ICs because it contains an "A" and a "B" virtual MACRO IC.

Macro.IC3s

Description: Number of PMAC3-style MACRO ICs detected

Range: 0 .. 15

Units: ICs

Macro.IC3s contains the number of PMAC3-style "DSPGATE3" ICs used with MACRO interfaces that were automatically detected at power-on/reset in the Power PMAC system.

Macro.Rings

Description: Number of separate MACRO rings detected

Range: 0 .. 4

Units: Rings

Macro.Rings contains the number of separate MACRO rings that were automatically detected at power-on/reset in the Power PMAC system, or by the most recent **`macrocontrollerinit`** or **`macrocontrollerdetect`** on-line command.

Macro.Station

Description: Power PMAC station number on ring

Range: 0 .. 255

Units: Station numbers

Macro.Station contains the station number of the Power PMAC on the ring. The synchronizing master Power PMAC always has a station number of 0. All non-synchronizing masters have a station number greater than 0.

Macro Ring Test Status Elements

The **Macro.RingTest[*i*]** substructure data elements provide information about the results of the automatic ring testing functionality for MACRO ring *i* controlled by the Power PMAC.

Macro.RingTest[*i*].PwrOnErrCntr

Description: Ring errors detected since power-on/reset

Range: Non-negative integers

Units: Errors

Macro.RingTest[*i*]PwrOnErrCtr contains the total number of ring errors that have been detected on Ring ***i*** since power-on/reset. It is possible to write to this element, for example to clear the error count to zero without resetting the Power PMAC.

Macro.RingTest[*i*].RingBrkStationNum

Description: Station number detecting ring break

Range: 0 .. 254

Units: Station numbers

Macro.RingTest[*i*].RingBrkStationNum contains the number of the station on Ring ***i*** that detected a ring break immediately upstream of it, if a break has been detected.

Macro.Status[*i*]. Status Data Structure Elements

The **Macro.Status[*i*]** substructure data elements provide information about the present state of MACRO ring ***i*** controlled by the Power PMAC.

Macro.Status[*i*].Active

Description: Valid data on ring, confirmed by testing

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].Active is set to if **Macro.TestPeriod** is greater than zero and there were no errors in a period. It will remain 0 until a valid test period has been detected after reset. Once it is set it will remain set until the Power PMAC is reset.

Macro.Status[*i*].AsciiCmdOn

Description: Internal MACRO ASCII handshaking bit

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].AsciiCmdOn is an internal handshaking bit used for ASCII communications over the MACRO ring.

Macro.Status[*i*].AsciiCmdRdy

Description: Internal MACRO ASCII handshaking bit

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].AsciiCmdRdy is an internal handshaking bit used for ASCII communications over the MACRO ring.

Macro.Status[*i*].AsciiCom

Description: Internal MACRO ASCII handshaking bit

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].AsciiCom is set to 1 when ASCII communication is active on this ring.

A valid connection to Station ***n*** (**MacroStation *n*)** will set this bit.

Macro.Status[*i*].AsciiRespRdy

Description: Internal MACRO ASCII handshaking bit

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].AsciiRespRdy is an internal handshaking bit used for ASCII communications over the MACRO ring.

Macro.Status[*i*].BrkDetected

Description: Ring break detected by Power PMAC

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].BrkDetected is set to 1 if the MACRO IC for Ring ***i*** has detected a ring break immediately upstream of it. It is 0 otherwise. The **CLRF** command will clear this fault.

Macro.Status[*i*].BrkMsgSent

Description: Ring break message sent by Power PMAC

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].BrkMsgSent is set to 1 if the MACRO IC for Ring ***i*** has sent a ring break message to the remaining stations downstream of it. Note that if this is the case, the Power PMAC has automatically turned itself into a synchronizing master. It is 0 otherwise.

The **CLRF** command will clear this fault.

Macro.Status[*i*].BrkReceivd

Description:

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].BrkReceivd is set to 1 if the MACRO IC for Ring *i* received a message from an upstream station that the station has detected a ring break. It is 0 otherwise.

Macro.RingTest[*i*].RingBrkStationNum contains the number of the station that detected the Ring Break.

The **CLRF** command will clear this fault.

Macro.Status[*i*].ErrorsFault

Description: Ring error count fault detected

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].ErrorsFault is set to 1 if the maximum permissible ring error count (**Macro.TestMaxErrors**) in a ring test period (**Macro.TestPeriod**) has been exceeded. It is 0 otherwise.

The **CLRF** command will clear this fault.

Macro.Status[*i*].MacroServoSync

Description: MACRO servo synchronization command sent

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].MacroServoSync is set to 1 if the **macroringordersync** command has been sent over Ring *i* so that servo cycles of all devices occur in the same phase cycle. It is 0 otherwise.

Macro.Status[*i*].Master

Description: MACRO ring master

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].Master is set to 1 if the MACRO IC for Ring ***i*** is a master on the ring, whether synchronizing or not. It is 0 if it is a slave on the ring.

Macro.Status[*i*].RingError

Description: MACRO ring error detected

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].RingError is set to 1 if Ring ***i*** has failed in a test period, either due to excessive communications errors (> **Macro.TestMaxErrors)** or not receiving enough synchronization packets (< **Macro.TestReqdSynchs**). It is 0 otherwise.

The **CLRF** command will clear this fault.

Macro.Status[*i*].StatWord

Description: MACRO ring ***i*** status word

Range: $0 .. $FFFFFFFF

Units: Bit field

Macro.Status[*i*].StatWord contains the field of status bits for MACRO ring ***i*** for the Power PMAC. The following table provides a list of these status bits in the word.

Bit #	Hex Value	Element Name: Macro.Status[*i*]. *{element}*	Description
15-31	$FFFF8000	-	(*Reserved for future use*)
14	$4000	**MacroServoSync**	MACRO servo synchronization command sent
13	$2000	**BrkMsgSent**	Power PMAC has detected a ring break and sent a message
12	$1000	**AsciiRespRdy**	Internal MACRO ASCII handshaking bit
11	$800	**AsciiCmdRdy**	Internal MACRO ASCII handshaking bit
10	$400	**AsciiCmdOn**	Internal MACRO ASCII handshaking bit
9	$200	**AsciiCom**	MACRO ASCII communications active
8	$100	**Master**	Power PMAC is a master on the ring
7	$80	**SynchMaster**	Power PMAC is the synchronizing master on the ring
6	$40	**TestEnabled**	Ring test function enabled
5	$20	**RingError**	Ring communications error detected
4	$10	**SynchFault**	Ring fault from too few sync packets detected
3	$8	**ErrorsFault**	Ring fault from too many data errors detected
2	$4	**BrkReceivd**	Message of ring break has been received
1	$2	**BrkDetected**	Upstream ring break detected
0	$1	**Active**	Ring active

Each element is described in more detail in its own individual specification.

Macro.Status[*i*].SynchFault

Description: MACRO synchronization packet error detected

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].SynchFault is set to 1 if the MACRO IC for Ring *i* has detected an insufficient number of "sync packets" (< **Macro.TestReqdSynchs**) in a test period. It is 0 otherwise.

The **CLRF** command will clear this fault.

Macro.Status[*i*].SynchMaster

Description: MACRO synchronizing ring master

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].SynchMaster is set to 1 if the MACRO IC for Ring ***i*** is the synchronizing ring master IC. It is 0 if it is a non-synchronizing master or a slave.

Macro.Status[*i*].TestEnabled

Description: MACRO ring integrity testing enabled

Range: 0 .. 1

Units: Boolean

Macro.Status[*i*].TestEnabled is set to 1 if the MACRO IC for Ring *i* is actively testing for ring communication errors and sync packets. It is 0 otherwise. **Macro.TestPeriod** must be greater than 0 to enable this testing.

Motor Status Data Structure Elements

The motor data structures in Power PMAC provide many elements that may be of interest to users for developing or debugging an application, or in the actual execution of the application. This section documents the key motor status elements.

Motor[*x*].ActiveMasterPos

Description: Accumulated (rate-limited) position following distance

Range: Floating-point

Units: Motor units

Motor[*x*].ActiveMasterPos contains the accumulated active following distance from the motor's position following function. If the following is speed and acceleration-limited (**Motor[*x*].MaxSpeed** and **Motor[*x*].MaxAccel** both greater than 0.0), this can differ from the unlimited following distance in **Motor[*x*].MasterPos** when limiting occurs. This difference is the accumulated "excess" that will be "released" when the following function is no longer limited.

Each servo cycle, the value of **Motor[*x*].ActiveMasterPos** is overwritten based on the value of **Motor[*x*].MasterPos**, so it is not possible to write directly to **Motor[*x*].ActiveMasterPos**.

If the position-following is in "offset" mode (**Motor[*x*].MasterCtrl** bit 1 = 1), this element contains the difference between the "base" programming origin for the motor and the "offset" programming origin actually used.

Motor[*x*].ActiveMasterPosSf

Description: Active (slew-limited) position following ratio

Range: Floating-point

Units: Motor units per master unit

Motor[*x*].ActiveMasterPosSf contains the actual position following ratio that was used in the most recent servo cycle. If saved setup element **Motor[*x*].SlewMasterPosSf** is greater than 0.0, enabling slew rate control of the following ratio, this can be different from the desired ratio in saved setup element **Motor[*x*].MasterPosSf** after enabling or disabling of the position-following function, or after changes in the desired ratio. When position-following is disabled, this value will be equal to 0.0.

Motor[*x*].ActPos

Description: Motor outer-loop feedback position

Range: Floating-point

Units: Motor units

Motor[*x*].ActPos contains the present servo cycle's net feedback position value for the outer servo loop. It is derived from the measured position in **Motor[*x*].Pos** with corrections such as those from position-table compensation and backlash compensation. It is scaled in the motor units and referenced to the power-up position. It will be subtracted from **Motor[*x*].DesPos** to obtain the following error value in **Motor[*x*].PosError**.

To calculate the position relative to the motor zero position, subtract the value of **Motor[x].HomePos**.

Motor[*x*].ActPos2

Description: Motor inner-loop feedback position

Range: Floating-point

Units: Inner-loop units

Motor[*x*].ActPos2 contains the present servo cycle's net feedback position value for the inner (usually velocity) servo loop. It is derived from the measured position in **Motor[*x*].Pos2** with corrections such as those from a compensation table. It is scaled in the units of the inner servo loop, which do not necessarily have to be the same as the motor units used in the outer loop, and referenced to the power-up position.

Motor[*x*].ActVel

Description: Motor actual velocity

Range: Floating-point

Units: Motor units per servo interrupt period

Motor[*x*].ActVel contains the present servo cycle's actual velocity value, the difference of the inner-loop actual position values for this servo cycle and the previous servo cycle. It is scaled in the motor units per servo interrupt period. Note that this value is calculated every servo interrupt period, even if the motor's servo loop is closed less often because **Motor[*x*].Stime** is set greater than 0. It is used in the calculation of the velocity and acceleration feedback terms for the motor's servo algorithm.

Motor[*x*].AmpEna

Description: "Amplifier enabled" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].AmpEna** status bit is set to 1 if the motor is enabled (in closed-loop or open-loop control). Note that there does not need to be an active amplifier-enable output signal in this case. This bit is 0 if the motor is disabled. It is bit 12 of 32-bit element **Motor[*x*].Status[0]**.


Note

This software status bit is distinct from the hardware output bit that actually controls the amplifier-enable output signal.

Motor[*x*].AmpFault

Description: "Amplifier fault error" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].AmpFault** status bit is 1 if the motor has been disabled because of an amplifier fault error, even if the amplifier fault signal condition is no longer present, or because of a calculated "I^2T" integrated current fault (in which case bit 21 is also set). It is 0 at all other times, becoming 0 again when the motor is re-enabled. It is bit 24 of 32-bit element **Motor[*x*].Status[0]**.

Note that this software status bit is distinct from the hardware input bit that reflects the state of the amplifier-fault input signal.


Note

This software status bit is distinct from the hardware input bit that reflects the state of the amplifier-fault input signal.

Motor[*x*].AmpWarn

Description: "Amplifier warning" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].AmpWarn** status bit is set to 1 if **Motor[*x*].AmpFaultLevel** bit 1 (value 2) is set to 1, requiring two consecutive readings of the amplifier fault bit in its specified fault state to trigger an error, and there has been one reading of the amplifier fault bit in its fault state. It is bit 19 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].AuxFault

Description: Auxiliary fault error status bit

Range: 0 .. 1

Units: Boolean

The **Motor[x].AuxFault** status bit is 1 if the motor has been disabled because it detected an "auxiliary fault" error condition as specified by the settings of saved setup elements **Motor[x].pAuxFault**, **Motor[x].AuxFaultBit**, **Motor[x].AuxFaultLevel**, and **Motor[x].AuxFaultLimit**. It is 0 at all other times, becoming 0 again when the motor is re-enabled. It is bit 17 of 32-bit element **Motor[x].Status[0]**.

Note

This software status bit is distinct from the hardware input bit that reflects the state of the auxiliary-fault input signal.

Motor[x].AuxFaultCount

Description: Cumulative number of scans finding auxiliary fault

Range: 0 .. 255

Units: Scans

Motor[x].AuxFaultCount contains the present accumulated number of scans in which the "auxiliary fault" state was detected. Each scan in which the fault state is detected, **Motor[x].AuxFaultCount** is incremented by one, until it exceeds the fault threshold specified by **Motor[x].AuxFaultLimit**. Each scan in which the fault state is not detected, **Motor[x].AuxFaultCount** is decremented by one, until it reaches zero.

The register for the auxiliary fault input bit is specified by **Motor[x].pAuxFault**. The bit within this register is specified by **Motor[x].AuxFaultBit**. The fault state of this bit is specified by **Motor[x].AuxFaultLevel**.

Motor[x].BlCompSize

Description: Motor backlash compensation-table correction

Range: Floating-point

Units: Motor units

Motor[x].BlCompSize contains the present servo cycle's net backlash correction from any compensation tables that use this element as a target register. If the motor is moving in the negative direction, a slew-rate-limited value derived from the sum of this element and saved setup

element **Motor[*x*].BlSize** is subtracted from the raw measured position in **Motor[*x*].Pos** to obtain the corrected value in **Motor[*x*].ActPos** that is used in the servo calculations.

If no compensation tables use this register as a target, it is possible for the user to write directly to this register to create a position offset.

Motor[*x*].BlDir

Description: "Backlash direction" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].BlDir** status bit is 1 if the motor's backlash function is enabled and the motor is executing or has most recently executed a position move in the negative direction. It is 0 otherwise. It is bit 28 of 32-bit element **Motor[*x*].Status[1]**.

Motor[*x*].BlockRequest

Description: "Block request flag set" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].BlockRequest** status bit is set to 1 if the motor has just entered a new move section, and is requesting that the equations for the next upcoming move section for the motion queue be calculated. It is 0 otherwise. It is primarily for internal use. It is bit 9 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].BrakeTimer

Description: Time remaining in brake state transition delay

Range: Floating-point

Units: Seconds

Motor[*x*].BrakeTimer contains the time remaining during a transition of the motor brake state, in seconds. When the motor is not in a transition period, it is set to 0.0.

During the transition for disengaging the brake on enabling the motor, whose period is set by **Motor[*x*].BrakeOffDelay** (which is specified in milliseconds), the value of **Motor[*x*].BrakeTimer** is positive, counting down from the initial value of **BrakeOffDelay**/1000 to 0.0.

During the transition for engaging the brake on disabling the motor, whose period is set by **Motor[*x*].BrakeOnDelay** (which is specified in milliseconds), the value of **Motor[*x*].BrakeTimer** is negative, counting up from the initial value of –**BrakeOnDelay**/1000 to 0.0.

User access to this element is new in V2.0 firmware, released 1st quarter 2015.

Motor[*x*].CapturedPos

Description: Motor monitored captured position

Range: Floating-point

Units: Motor units

Motor[*x*].CapturedPos contains the most recent motor position value calculated from the monitored position-capture function for the motor. This function is activated by setting non-saved setup element **Motor[*x*].CapturePos** to 1. When the capture event for the motor occurs, Power PMAC automatically processes the captured sensor value into a motor position and stores the value in this element. This monitoring function does not affect the motor motion in any way.

The capture trigger and subsequent position processing are specified just as for move-until-trigger functions. This specification is covered in depth in the section on triggered motor moves in the User's Manual chapter *Executing Individual Motor Moves*. The monitored position-capture function should not be used simultaneously with triggered moves. Triggered moves do not affect the value of this element.

This position value is referenced to the motor zero position (not necessarily power-on position). It is not affected by any programming offsets for an axis assigned to the motor.

Motor[*x*].ClosedLoop

Description: "Closed-loop mode" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].ClosedLoop** status bit is set to 1 if the motor is in closed-loop control. It is zero if the motor is in open-loop mode (enabled or disabled). It is bit 13 of 32-bit element **Motor[*x*].Status[0]**.

Motor[*x*].CompDac

Description: Motor servo-loop output ("torque") compensation correction

Range: Floating-point

Units: 16-bit output units

Motor[*x*].CompDac contains the present servo cycle's net servo-loop output correction from any compensation tables that use this element as a target register. This value is added to the raw servo-loop output in **Motor[*x*].ServoOut** to obtain the corrected value that is output to the register specified by **Motor[*x*].pDac** if Power PMAC is not performing commutation for the motor, or that is input to the commutation algorithm as the torque command in **Motor[*x*].IqCmd** if Power PMAC is performing commutation for the motor. The magnitude of the sum of **Motor[*x*].ServoOut** and **Motor[*x*].CompDac** is compared to the value of saved setup element **Motor[*x*].MaxDac**, and limited to that value if necessary.

If no compensation tables use this register as a target, it is possible for the user to write directly to this register to create a position offset.

Motor[*x*].CompDesPos

Description: Motor desired position compensation-table offset

Range: Floating-point

Units: Motor units

Motor[*x*].CompDesPos contains the present servo cycle's net desired position "correction" from any compensation tables that use this element as a target register. This value is added to the trajectory commanded position in **Motor[*x*].Desired.Pos** and the master position value in **Motor[*x*].ActiveMasterPos** to obtain the net desired position value in **Motor[*x*].DesPos** that is used in the servo calculations.

Motor[*x*].CompDesPos is typically used as a target register for a compensation table when the table is used as an "electronic cam" table to create desired motion of this motor as a function of another motor's position. To correct for measurement errors on the motor, it is better to use **Motor[*x*].CompPos**, which is an offset to actual measured position.

If no compensation tables use this register as a target, it is possible for the user to write directly to this register to create a position offset.

Motor[*x*].CompPos

Description: Motor outer-loop actual position compensation-table correction

Range: Floating-point

Units: Motor units

Motor[*x*].CompPos contains the present servo cycle's net outer-loop actual position correction from any compensation tables that use this element as a target register. This value is added to the raw measured position in **Motor[*x*].Pos** to obtain the corrected value in **Motor[*x*].ActPos** that is used in the servo calculations.

If no compensation tables use this register as a target, it is possible for the user to write directly to this register to create a position offset.

Motor[*x*].CompPos2

Description: Motor inner-loop actual position compensation-table correction

Range: Floating-point

Units: Inner-loop units

Motor[*x*].CompPos2 contains the present servo cycle's net inner-loop actual position correction from any compensation tables that use this element as a target register. This value is added to the raw measured position in **Motor[*x*].Pos2** to obtain the corrected value in **Motor[*x*].ActPo2s** that is used in the servo calculations.

If no compensation tables use this register as a target, it is possible for the user to write directly to this register to create an inner-loop offset.

Motor[*x*].Coord

Description: Coordinate system to which motor is assigned

Range: 0 .. 127

Units: Coordinate system number

Motor[*x*].Coord contains the number of the coordinate system to which the motor has been assigned through the use of a coordinate-system-specific axis definition statement. The motor can be defined to a positioning axis, either explicitly or as an inverse-kinematic axis, a spindle axis, or given the null definition in the coordinate system. On re-initialization, all motors are automatically given the null definition in Coordinate System 0, so **Motor[*x*].Coord** will report as 0 for all motors on re-initialization.

Motor[*x*].CoordSf[*i*]

Description: Axis-definition scale factor

Range: Floating-point

Units: Motor units per axis unit

Motor[*x*].CoordSf[*i*] contains the axis-definition scale factor relating Motor ***x*** to the axis with index ***i***, where ***i*** has a range of 0 to 32. The following table shows the axis index value for each of the 32 axis names:

Axis Name	Index *i*	Axis Name	Index *i*	Axis Name	Index *i*	Axis Name	Index *i*
A	0	Z	8	HH	16	SS	24
B	1	AA	9	LL	17	TT	25
C	2	BB	10	MM	18	UU	26
U	3	CC	11	NN	19	VV	27
V	4	DD	12	OO	20	WW	28
W	5	EE	13	PP	21	XX	29
X	6	FF	14	QQ	22	YY	30
Y	7	GG	15	RR	23	ZZ	31

Motor[*x*].CoordSf[32] is the "axis offset" value.

These values are typically set by on-line axis-definition commands. For example, the axis-definition command:

```
#1->866X+500Y-333.333
```

would set **Motor[1].CoordSf[6]** to 866, **Motor[1].CoordSf[7]** to 500, and **Motor[1].CoordSf[32]** to -333.333. All other **Motor[1].CoordSf[*i*]** elements would be set to 0.

Typically, these elements are only read by the user. It is possible to write to them directly. Remember that changing a value changes the relationship between motors and axes, so that before the next programmed axis move, the "pmatch" position-match function should be performed to re-establish the proper relationship. This is automatically done when a motion program is started, but otherwise, an explicit **`pmatch`** command should be executed.

In addition, changing a value between a zero and non-zero value changes the entire logic for matching motor and axis positions properly – this is strongly discouraged.

Motor[*x*].Csolve

Description: "Motor used in PMATCH calculations" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].Csolve** status bit is set to 1 if this motor's position is used to calculate axis positions in a coordinate-system "pmatch" (position match) function (at motion-program start, **pmatch** command execution, axis position/velocity/following-error query). It is 0 when the motor has a "null" definition or a redundant axis definition, and so is not used in these calculations. It is bit 31 of 32-bit element **Motor[*x*].Status[1]**.

The user may write to this bit in order to change which motors are used in the "pmatch" function, as to change which motor of a gantry pair is used. However, great care should be taken in doing this, as it is quite possible to create an invalid function, with dangerous results.

Motor[*x*].CurrentNullTimer

Description: Number of cycles remaining in current auto-null averaging

Range: 0 .. 32,767

Units: Phase interrupt periods

Motor[*x*].CurrentNullTimer contains the number of phase cycles remaining in the current auto-nulling process for the motor. At the beginning of the auto-nulling process, it is set to the magnitude of saved setup element **Motor[*x*].CurrentNullPeriod**, and it is decremented by 1 each phase cycle until the process is completed. While **CurrentNullTimer** is greater than 0, actual current-loop control is disabled, and status elements **Motor[*x*].IaMeas** and **Motor[*x*].IbMeas** contain the accumulated sum of measured phase current during the process instead of the present cycle's value. If **CurrentNullTimer** is equal to 0, no auto-nulling process is presently occurring.

Motor[*x*].CurrentNullTimer is intended primarily as a status element to track the progress of the standard auto-nulling process, it is possible for the user to write a positive value to start the auto-nulling process at a time not supported by the standard commands. In this case, it is the user's full responsibility to make sure that the auto-nulling is occurring with the motor in an appropriate state.

Motor[*x*].DacLimit

Description: "Servo output limited" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].DacLimit** status bit is 1 if the motor's servo output command value is presently saturated to the magnitude of **Motor[*x*].MaxDac**. It is 0 otherwise. It is bit 29 of 32-bit element **Motor[*x*].Status[1]**.

Motor[x].DesPos

Description: Motor net desired position

Range: Floating point

Units: Motor units

Motor[*x*].DesPos contains the present servo cycle's net desired position value, the sum of the calculated trajectory position, the master position from the position following function (**Motor[*x*].ActiveMasterPos**), and the desired position compensation value (**Motor[*x*].CompDesPos**) from cam or compensation tables. It is scaled in the motor units and referenced to the power-up/reset position. The value of **Motor[*x*].ActPos** will be subtracted from this to obtain the following error value in **Motor[*x*].PosError**.

To calculate the position relative to the motor zero position, subtract the value of **Motor[*x*].HomePos**.

Motor[x].DesVel

Description: Motor net desired velocity

Range: Floating point

Units: Motor units per servo interrupt period

Motor[*x*].DesVel contains the present servo cycle's net desired velocity value, the difference of the desired position values for this servo cycle and the previous servo cycle. It is scaled in the motor units per servo interrupt period. Note that this value is calculated every servo interrupt period, even if the motor's servo loop is closed less often because **Motor[*x*].Stime** is set greater than 0. It is used in the calculation of the velocity, acceleration and friction feedforward terms for the motor's servo algorithm.

Motor[x].DesVelZero

Description: "Desired velocity zero" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].DesVelZero** status bit is set to 1 if the motor is in closed-loop control and the net commanded velocity is exactly zero (i.e. it is trying to hold position), or if it is in open-loop mode and the actual velocity is exactly zero. It is zero either if the motor is in closed-loop mode with non-zero commanded velocity, or if it is in open-loop mode (enabled or disabled) with non-zero actual velocity. It is bit 14 of 32-bit element **Motor[*x*].Status[0]**.

Saved setup element **Sys.ZeroVelSetPoint** can be used to set a threshold for "desired velocity zero" that is not exactly 0.0. It is typically used with a trajectory pre-filter that creates an "infinite

impulse response" filter that gradually decays to zero velocity at the end of a programmed move, so it does not have to wait for an exact 0.0 value (which is obtained only by numerical underflow) to be reached. **Sys.ZeroVelSetPoint** is new in V2.0.2 firmware, released 2nd quarter 2015.

Motor[*x*].EncLoss

Description: "Sensor loss" error status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].EncLoss** status bit is 1 if the motor has been disabled because it detected an "encoder loss" condition as specified by the settings of saved setup elements **Motor[*x*].pEncLoss**, **Motor[*x*].EncLossBit**, **Motor[*x*].EncLossLevel**, and **Motor[*x*].EncLossLimit**. It is 0 at all other times, becoming 0 again when the motor is re-enabled. It is bit 18 of 32-bit element **Motor[*x*].Status[0]**.

Note

This software status bit is distinct from the hardware input bit that reflects the state of the encoder-loss input signal.

Motor[*x*].EncLossCount

Description: Cumulative number of scans finding sensor loss

Range: 0 .. 255

Units: Scans

Motor[*x*].EncLossCount contains the present accumulated number of scans in which the "encoder loss" state was detected. Each scan in which the loss state is detected, **Motor[*x*].EncLossCount** is incremented by one, until it exceeds the fault threshold specified by **Motor[*x*].EncLossLimit**. Each scan in which the loss state is not detected, **Motor[*x*].EncLossCount** is decremented by one, until it reaches zero.

The register for the encoder loss bit is specified by **Motor[*x*].pEncLoss**. The bit within this register is specified by **Motor[*x*].EncLossBit**. The loss state of this bit is specified by **Motor[*x*].EncLossLevel**.

Motor[x].FeFatal

Description: "Fatal following error" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].FeFatal** status is 1 if the motor has been disabled because the magnitude of the following error for the motor has exceeded its fatal following error limit as set by **Motor[*x*].FatalFeLimit**. It is 0 at all other times, becoming 0 again when the motor is re-enabled. It is bit 26 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].FeWarn

Description: "Warning following error" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].FeWarn** status bit is 1 if the magnitude of the following error for the motor exceeds its warning following error limit as set by **Motor[*x*].WarnFeLimit**. It is 0 if the magnitude of the following error is less than this limit, or if the motor has been disabled due to exceeding its fatal following error limit. It is bit 27 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].FltrMasterPos[i]

Description: Motor master position history array element

Range: Floating point

Units: Motor units

Motor[*x*].FltrMasterPos[*i*] contains the motor's master position value from one of the past 16 servo cycles. Each servo cycle, Power PMAC copies the value of **Motor[*x*].ActiveMasterPos** into the element **Motor[*x*].FltrMasterPos[*i*]**, where the value of ***i*** is equal the low 4 bits of the servo cycle counter found in **Sys.ServoCount**, so ***i*** can take a value from 0 through 15.

Motor[*x*].FltrMasterPos[*i*] is scaled in the motor units and referenced to the power-up master position. It is used in the calculation of the filtered master velocity value **Motor[*x*].FltrMasterVel** for reporting purposes.

Motor[x].FltrMasterVel

Description: Motor master filtered velocity

Range: Floating point

Units: Motor units per 16 servo interrupt periods

Motor[*x*].FltrMasterVel contains the motor's "filtered" master velocity value for the most recent servo cycle. Each servo cycle, Power PMAC subtracts the master position value from 16 servo cycles previously (stored in **Motor[*x*].FltrMasterPos[*i*]**) from the present cycle's position (in **Motor[*x*].MasterPos**) to compute **Motor[*x*].FltrMasterVel**. Note that this value is calculated every servo interrupt period, even if the motor's servo loop is closed less often because **Motor[*x*].Stime** is set greater than 0.

Motor[x].FltrPos[i]

Description: Motor outer loop feedback position history array element

Range: Floating point

Units: Motor units

Motor[*x*].FltrPos[*i*] contains the net feedback position value for the outer servo loop from one of the past 16 servo cycles. Each servo cycle, Power PMAC copies the value of **Motor[*x*].ActPos** into the element **Motor[*x*].FltrPos[*i*]**, where the value of ***i*** is equal the low 4 bits of the servo cycle counter found in **Sys.ServoCount**, so ***i*** can take a value from 0 through 15.

Motor[*x*].FltrPos[*i*] is scaled in the motor units and referenced to the power-up position. It is used in the calculation of the filtered velocity value **Motor[*x*].FltrVel** for reporting purposes. To calculate the position relative to the motor zero position, subtract the value of **Motor[*x*].HomePos**.

Motor[x].FltrPos2[i]

Description: Motor inner loop feedback position history array element

Range: Floating point

Units: Inner-loop units

Motor[*x*].FltrPos2[*i*] contains the net feedback position value for the inner servo loop from one of the past 16 servo cycles. Each servo cycle, Power PMAC copies the value of **Motor[*x*].ActPos2** into the element **Motor[*x*].FltrPos2[*i*]**, where the value of ***i*** is equal the low 4 bits of the servo cycle counter found in **Sys.ServoCount**, so ***i*** can take a value from 0 through 15.

Motor[*x*].FltrPos2[*i*] is scaled in the inner-loop units, which do not necessarily have to be the same as the motor units used in the outer loop, and referenced to the power-up position. It is used in the calculation of the filtered velocity value **Motor[*x*].FltrVel2** for reporting purposes.

Motor[x].FltrVel

Description: Motor outer loop feedback filtered velocity

Range: Floating point

Units: Motor units per 16 servo interrupt periods

Motor[*x*].FltrVel contains the "filtered" velocity value for the outer servo loop for the most recent servo cycle. Each servo cycle, Power PMAC subtracts the actual position value from 16 servo cycles previously (stored in **Motor[*x*].FltrPos[*i*]**) from the present cycle's position (in **Motor[*x*].ActPos**) to compute **Motor[*x*].FltrVel**. Note that this value is calculated every servo interrupt period, even if the motor's servo loop is closed less often because **Motor[*x*].Stime** is set greater than 0.

Motor[x].FltrVel is used to calculate the reported velocity for the on-line **v** command or the buffered **vread** command. The reported values are rescaled into motor units per millisecond. To convert the value of **Motor[*x*].FltrVel** into these units, divide by 16 and by **Sys.ServoPeriod**.

Motor[x].FltrVel2

Description: Motor inner loop feedback filtered velocity

Range: Floating point

Units: Inner-loop units per 16 servo interrupt periods

Motor[*x*].FltrVel2 contains the "filtered" velocity value for the inner servo loop for the most recent servo cycle. Each servo cycle, Power PMAC subtracts the actual position value from 16 servo cycles previously (stored in **Motor[*x*].FltrPos2[*i*]**) from the present cycle's position (in **Motor[*x*].ActPos2**) to compute **Motor[*x*].FltrVel2**. Note that this value is calculated every servo interrupt period, even if the motor's servo loop is closed less often because **Motor[*x*].Stime** is set greater than 0.

Motor[x].GantryHomed

Description: "Gantry homing complete" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].GantryHomed** status bit is set to 1 if this motor is a follower motor in a leader/follower gantry system, the home triggers for both leader and this follower motor have been found, and the power-on skew between this follower motor and the leader has been fully removed. It is 0 otherwise. It is bit 6 of 32-bit element **Motor[*x*].Status[0]**.

Motor[*x*].HomeComplete

Description: "Position reference established" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].HomeComplete** status bit is set to 1 if a position reference is properly established for the motor, either with a homing-search move, or an absolute position read. It is automatically set to 0 at power-up/reset, and at the beginning of a homing-search move. In a homing-search move, it is set to 1 when the pre-specified trigger condition is found, which is before the post-trigger portion of the move is complete. It is bit 15 of 32-bit element **Motor[*x*].Status[0]**.

Motor[*x*].HomeInProgress

Description: "Homing search move in progress" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].HomeInProgress** status bit is set to 1 at the beginning of a homing search move. It is set to 0 when the pre-specified trigger condition is found and the post-trigger move is started, or when the move ends without a trigger being found. It is bit 30 of 32-bit element **Motor[*x*].Status[0]**.

Motor[*x*].HomePos

Description: Motor reference position

Range: Floating-point

Units: Motor units

Motor[*x*].HomePos contains the motor zero position relative to the power-up/reset position. It is set during the homing search move or absolute position read for the motor. Because many motor position element values are relative to the power-up/reset position, the value of this element can be subtracted from those values to obtain values relative to the motor zero position.

Although normally considered a read-only status element, it is permissible to write to this element, allowing custom position referencing algorithms.

Motor[x].I2tFault

Description: "Integrated current (I^2T) fault" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].I2tFault** status bit is set to 1 when the motor has been disabled from exceeding its integrated current limit as set by **Motor[*x*].I2tTrip**. The amplifier fault status bit (**Motor[*x*].AmpFault**) will also be set in this case. It will be 0 at all other times, becoming 0 when the motor is re-enabled. (Note that if the amplifier faults due to its own integrated current fault calculations, this bit will not be set.) It is bit 21 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].I2tSum

Description: Present integrated current value

Range: Non-negative floating-point

Units: (16-bit DAC/ADC units)^2 * seconds

Motor[*x*].I2tSum contains the present integrated current value for Power PMAC's I^2T integrated current limiting function. If it exceeds the threshold value set by saved setup element **Motor[*x*].I2tTrip**, an I^2T fault occurs, killing the motor and setting motor status bit **Motor[*x*].I2tFault** to 1.

If the present current level is above that set by saved element **Motor[*x*].I2tSet**, the difference of the square of this current level minus the square of **Motor[*x*].I2tSet** multiplied by the time for the scan is added into **Motor[*x*].I2tSum** (until it exceeds the limit and faults the motor). If the present current level is below that set by saved element **Motor[*x*].I2tSet**, the difference of the square of this current level minus the square of **Motor[*x*].I2tSet** multiplied by the time for the scan is subtracted from **Motor[*x*].I2tSum** (until it reaches zero).

Motor[x].IaMeas

Description: Motor A-phase measured current

Range: -32,768.0 .. 32,767.999

Units: 16-bit input units

Motor[*x*].IaMeas contains the present actual current value for the A-phase of the motor. It is calculated only if digital current loop operation is enabled for the motor, obtained from the register specified by saved setup element **Motor[*x*].pAdc** with the offset from saved setup element **Motor[*x*].IaBias** added in. It is in units of a 16-bit ADC, even if a different resolution ADC is used.

Motor[x].IaVolts

Description: Motor A-phase output command

Range: -32,768.0 .. 32,767.999

Units: Output units

Motor[*x*].IaVolts contains the present command output value for the A-phase of the motor, whether digital current loop operation is enabled for the motor or not. It has already been scaled by the value of saved setup element **Motor[*x*].PwmSf**. If digital current loop operation is enabled, it includes the offset from non-saved setup element **Motor[*x*].VaBias** added in. This value is converted to integer and written to the register specified by **Motor[*x*].pDac**.

Motor[x].IbMeas

Description: Motor B-phase measured current

Range: -32,768.0 .. 32,767.999

Units: 16-bit input units

Motor[*x*].IbMeas contains the present actual current value for the B-phase of the motor. It is calculated only if digital current loop operation is enabled for the motor, obtained from the register after the one specified by saved setup element **Motor[*x*].pAdc** with the offset from saved setup element **Motor[*x*].IbBias** added in. It is in units of a 16-bit ADC, even if a different resolution ADC is used.

Motor[x].IbVolts

Description: Motor B-phase output command

Range: -32,768.0 .. 32,767.999

Units: Output units

Motor[*x*].IbVolts contains the present command output value for the B-phase of the motor, whether digital current loop operation is enabled for the motor or not. It has already been scaled by the value of saved setup element **Motor[*x*].PwmSf**. If digital current loop operation is enabled, it includes the offset from non-saved setup element **Motor[*x*].VbBias** added in. This value is converted to integer and written to the register after the one specified by **Motor[*x*].pDac**.

Motor[x].IcVolts

Description: Motor C-phase output command

Range: -32,768.0 .. 32,767.999

Units: Output units

Motor[*x*].IcVolts contains the present command output value for the C-phase of the motor. It is calculated only if digital current loop operation is enabled for the motor. It has already been scaled by the value of saved setup element **Motor[*x*].PwmSf**. This value is converted to integer and written to the register two after the one specified by **Motor[*x*].pDac**.

Motor[x].IdMeas

Description: Motor measured direct (field) current

Range: -32,768.0 .. 32,767.999

Units: 16-bit input units

Motor[*x*].IdMeas contains the present actual "direct" current input to the commutation algorithm. It is calculated only if digital current loop operation is enabled for the motor. It is derived from the phase current measurements **Motor[*x*].IaMeas** and **Motor[*x*].IbMeas** based on the rotor field angle in **Motor[*x*].PhasePos**. It is compared to the commanded direct current value in **Motor[*x*].IdCmd** to drive the current loop. It is in units of a 16-bit ADC, even if a different resolution ADC is used.

Motor[x].IdInt

Description: Motor direct (field) current integrator output

Range: -32,768.0 .. 32,767.999

Units: 16-bit units

Motor[*x*].IdInt contains the present output value of the "direct" current integrator in the commutation algorithm. It is calculated only if digital current loop operation is enabled for the motor. Each phase cycle, the direct current error is multiplied by the current integral gain term **Motor[*x*].IiGain** and the product is added into the value of this element from the previous cycle. The resulting integrated value is then added to the output values from the proportional gain terms to obtain the net direct output command **Motor[*x*].IdVolts** for the cycle.

Motor[x].IdVolts

Description: Motor direct (field) current loop output

Range: -32,768.0 .. 32,767.999

Units: 16-bit input units

Motor[*x*].IdVolts contains the present output value of the "direct" current loop of the commutation algorithm. It is calculated only if digital current loop operation is enabled for the motor. Each phase cycle, the proportional and integral gain terms for the current loop are calculated and combined to obtain this net output term. This is then used in combination with **Motor[*x*].IqVolts** and the rotor field angle in **Motor[*x*].PhasePos** to compute the individual phase output voltages.

Motor[x].Imag

Description: Motor estimated rotor field (magnetization) current

Range: -32,768.0 .. 32,767.999

Units: 16-bit units

Motor[*x*].Imag contains the present estimated rotor magnetization current for an asynchronous (induction) motor. It is derived from the commanded direct current value in **Motor[*x*].IdCmd** as filtered by the rotor time constant value in saved setup element **Motor[*x*].DtOverRotorTc**. It is used in the calculation of the motor slip frequency.

Motor[x].InPos

Description: "In position" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].InPos** status bit is set to 1 when all of the conditions for "in position" are satisified: the motor is closed-loop, the desired velocity is zero, the move timer is not active (no move, dwell, or delay being executed, the magnitude of the following error is less than or equal to **Motor[*x*].InPosBand**, and all of these conditions have been true for (**Motor[*x*].InPosTime** - 1) consecutive servo cycles. It is 0 otherwise. It is bit 11 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].InPosTimer

Description: In-position consecutive scan counter

Range: Non-negative integer

Units: Servo cycles

Motor[*x*].InPosTimer contains the number of additional consecutive servo cycles that the motor will need to meet the instantaneous "in-position" conditions for that motor to be considered "in-position". Each servo cycle, Power PMAC evaluates the motor to see if four conditions related to the in-position function are true:

6.) The motor must be in closed-loop control;
7.) The motor desired velocity must be zero;
8.) The motor must not be executing any move or dwell of definite time;
9.) The magnitude of the following error must be less than or equal to **Motor[*x*].InPosBand**

If all four of these conditions are true, Power PMAC decrements **Motor[*x*].InPosTimer** towards 0, or if it is already 0, Power PMAC sets the motor status bit **Motor[*x*].InPos** to 1, and the motor is considered to be "in position" for subsequent operations.

If any of these four conditions is false, Power PMAC sets **Motor[*x*].InPosTimer** to the value of saved setup element **Motor[*x*].InPosTime**, and sets the motor status bit **Motor[*x*].InPos** to 0.

Motor[x].IqCmd

Description: Motor commanded quadrature (torque) current

Range: -32,768.0 .. 32,767.999

Units: 16-bit output units

Motor[*x*].IqCmd contains the present "quadrature" current command input to the commutation algorithm. This is the output of the motor's servo algorithm, and it represents the desired torque value. If digital current loop operation is enabled for the motor, it is compared to the actual quadrature current value in **Motor[*x*].IqMeas** to drive the current loop.

Motor[x].IqInt

Description: Motor quadrature (torque) current integrator output

Range: -32,768.0 .. 32,767.999

Units: 16-bit units

Motor[*x*].IqInt contains the present output value of the "quadrature" current integrator in the commutation algorithm. It is calculated only if digital current loop operation is enabled for the motor. Each phase cycle, the quadrature current error is multiplied by the current integral gain

term **Motor[*x*].IiGain** and the product is added into the value of this element from the previous cycle. The resulting integrated value is then added to the output values from the proportional gain terms to obtain the net quadrature output command **Motor[*x*].IqVolts** for the cycle.

Motor[*x*].IqMeas

Description: Motor measured quadrature (torque) current

Range: -32,768.0 .. 32,767.999

Units: 16-bit input units

Motor[x].IqMeas contains the present actual "quadrature" current input to the commutation algorithm. It is calculated only if digital current loop operation is enabled for the motor. It is derived from the phase current measurements **Motor[*x*].IaMeas** and **Motor[*x*].IbMeas** based on the rotor field angle in **Motor[*x*].PhasePos**. It is in units of a 16-bit ADC, even if a different resolution ADC is used.

Motor[*x*].IqVolts

Description: Motor quadrature (torque) current loop output

Range: -32,768.0 .. 32,767.999

Units: 16-bit input units

Motor[*x*].IqVolts contains the present output value of the "quadrature" current loop of the commutation algorithm. It is calculated only if digital current loop operation is enabled for the motor. Each phase cycle, the proportional and integral gain terms for the current loop are calculated and combined to obtain this net output term. This is then used in combination with **Motor[*x*].IdVolts** and the rotor field angle in **Motor[*x*].PhasePos** to compute the individual phase output voltages.

Motor[*x*].JogPos

Description: Motor jog end desired position

Range: Floating-point

Units: Motor units

Motor[*x*].JogPos contains the end position of the present or most recently executed "definite" jog move. It is scaled in the motor units and referenced to the power-up/reset position.

To calculate the position relative to the motor zero position, subtract the value of **Motor[*x*].HomePos**.

Motor[x].JogVel

Description: Motor move top desired signed velocity or velocity magnitude

Range: Floating-point

Units: Motor units per millisecond

Motor[x].JogVel contains the top desired velocity magnitude of the present or most recently executed motor jog, homing-search, or rapid-mode move if to a definite position (including indefinite jog moves automatically changed to definite moves by a soft limit).

It contains the top signed velocity value of the present or most recently executed indefinite motor jog or homing-search move (including the pre-trigger portion of a homing-search move).

Motor[x].JogVel is scaled in motor units per millisecond. It is distinct from the saved setup element **Motor[x].JogSpeed**, which is used to specify the magnitude of the top speed of these moves.

Motor[x].LimitStop

Description: "Stopped on limit detected at execution time" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[x].LimitStop** status bit is 1 if the motor has stopped or is stopping because it reached a limit detected at move execution (not move calculation) time. This can be because it hit either its positive or negative hardware overtravel limit (even if it is presently not in that limit), or because it exceeded a software overtravel limit during execution. It is 0 at all other times, including when into a limit, but moving out of it. It is bit 25 of 32-bit element **Motor[x].Status[0]**.

Motor[x].MacroCtrl

Description: MACRO motor command flag holding register

Range: $0 .. $FFFFFFFF

Units: Bit field

Motor[x].MacroCtrl is a 32-bit holding register in memory for MACRO command flags such as amplifier enable and capture reset. If the motor command flags are to be sent over the MACRO ring (**Motor[x].EncCtrl** contains the address of a MACRO IC register), Power PMAC will manipulate bits in this holding register, then copy the entire register to the MACRO IC register.

The high 8 bits (bits 24 – 31) of **Motor[*x*].MacroCtrl** comprise the writeable non-saved element **Motor[*x*].MacroFlags**. Users wanting to transmit customized motor command flags over the MACRO ring should use that element.

Motor[*x*].MacroStatus

Description: MACRO motor status flag holding register

Range: $0 .. $FFFFFFFF

Units: Bit field

Motor[*x*].MacroStatus is a 32-bit holding register in memory for MACRO status flags such as overtravel limits and the home flag. If the motor status flags are to be sent over the MACRO ring (**Motor[*x*].EncStatus** contains the address of a MACRO IC register), Power PMAC will copy the entire register from the MACRO IC register, then manipulate bits in this holding register.

Motor[*x*].MasterPos

Description: Accumulated unlimited position following distance

Range: Floating-point

Units: Motor units

Motor[*x*].MasterPos contains the accumulated unlimited following distance from the motor's position following function. If the following is speed and acceleration-limited (**Motor[*x*].MaxSpeed** and **Motor[*x*].MaxAccel** both greater than 0.0), this can differ from the active following distance in **Motor[*x*].ActiveMasterPos** when limiting occurs. This difference is the accumulated "excess" that will be "released" when the following function is no longer limited.

If the following is speed-limited but not acceleration-limited (**Motor[*x*].MaxSpeed** greater than 0.0, but **Motor[*x*].MaxAccel** equal to 0.0), no "excess" is accumulated when speed limiting occurs, and this value is equal to the active following distance in **Motor[*x*].ActiveMasterPos**.

Generally, **Motor[*x*].MasterPos** will only be read by the user (and for diagnostic purposes only), but it is possible to write to it for custom following, excitation, and offset algorithms for a standard motor.

However, if the motor is set up as a gantry "follower" (**Motor[*x*].ServoCtrl** = 8), then **Motor[*x*].MasterPos** contains the "de-skewing" home offset between this follower motor and the gantry leader motor. In this case, the automatic algorithms will immediate overwrite any value the user writes to it in the application.

Motor[x].MinusLimit

Description: "Hardware negative limit set" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].MinusLimit** status bit is set to 1 when the motor is presently in its negative hardware limit. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously. It is bit 29 of 32-bit element **Motor[*x*].Status[0]**.

Note

This software status bit is different from the hardware input bit for the limit switch (typically **Gate*n*[*i*].Chan[*j*].MinusLimit**).

Motor[x].MotorTa

Description: Motor move acceleration time or inverse rate

Range: Floating point

Units: Milliseconds (if >= 0) or milliseconds^2 per motor unit (if < 0)

Motor[*x*].MotorTa contains the acceleration time or inverse rate used for the most recent motor move. It comes from **Motor[*x*].JogTa** for a jogging or homing move, or it comes from **Motor[*x*].AbortTa** for an abort deceleration. It is primarily for internal use.

Motor[x].MotorTs

Description: Motor move S-curve time or inverse jerk rate

Range: Floating point

Units: Milliseconds (if >= 0) or milliseconds^3 per motor unit (if < 0)

Motor[*x*].MotorTs contains the S-curve time or inverse jerk rate used for the most recent motor move. It comes from **Motor[*x*].JogTs** for a jogging or homing move, or it comes from **Motor[*x*].AbortTs** for an abort deceleration. It is primarily for internal use.

Motor[x].MoveDesPos

Description: Motor programmed move end desired position

Range: Floating point

Units: Motor units

Motor[*x*].MoveDesPos contains the end position of the present or most recently executed programmed move, or if a segmented move, the end position of the present or most recently executed move segment. It is scaled in the motor units and referenced to the power-up/reset position.

To calculate the position relative to the motor zero position, subtract the value of **Motor[*x*].HomePos**.

Motor[*x*].MoveTimer

Description: Motor time from beginning of presently executing section or segment

Range: Floating-point

Units: Milliseconds

Motor[*x*].MoveTimer contains the time elapsed (at +100% time base) from the beginning of the presently executing section or segment of motion for the motor. Each servo cycle, this value is incremented by the present value of **Coord[*x*].TimeBase** for the coordinate system to which the motor is assigned. The total time for this section or segment is contained in **Motor[*x*].Desired.Time**.

Motor[*x*].PhaseFindingEnabled

Description: Phasing search move executing

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].PhaseFindingEnabled** status bit is set to 1 while a phasing-search move is executing for the motor. It is set to 0 otherwise.

Motor[*x*].PhaseFound

Description: “Phase reference established” status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].PhaseFound** status bit is set to 0 on power-up/reset for a motor commutated by Power PMAC (**Motor[*x*].PhaseCtrl** bit 0 or bit 2 = 1) that is synchronous (**Motor[*x*].DtOverRotorTc** = 0.0). It is set to 1 if a phase reference is properly established for

the motor, either with a phasing-search move, or an absolute position read. It is bit 8 of 32-bit element **Motor[*x*].Status[0]**.

Motor[*x*].PhasePos

Description: Motor commutation present phase angle

Range: 0 .. 2047.999

Units: Commutation angle units

Motor[*x*].PhasePos contains the present commutation angle in the units of 1/2048 of a commutation cycle. The closest integer value is used to select the entry in the selected "sine lookup tables" as specified by saved setup elements **Motor[*x*].pSineTable** and **Motor[*x*].pVoltSineTable**.

Although normally considered a read-only status element, it is permissible to write to this element, allowing custom phase referencing algorithms.

WARNING

Writing a value to the **Motor[*x*].PhasePos** element that is significantly different from the true commutation angle can cause torque or force to be applied in the wrong direction, leading to destabilizing positive feedback and dangerous runaway conditions.

Motor[*x*].PhaseVoltOffset

Description: Motor commutation phase-advance offset

Range: -512 .. 512

Units: Commutation angle units

Motor[*x*].PhaseVoltOffset contains the present commutation angle offset due to the velocity dependent "phase advance". It is calculated from the present motor velocity and the saved setup element **Motor[*x*].AdvGain**.

Motor[*x*].PlusLimit

Description: "Hardware positive limit set" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].PlusLimit** status bit is set to 1 when the motor is presently in its positive hardware limit. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously. It is bit 28 of 32-bit element **Motor[*x*].Status[0]**.

Note

This software status bit is different from the hardware input bit for the limit switch (typically **Gate*n*[*i*].Chan[*j*].PlusLimit**).

Motor[*x*].Pos

Description: Motor unprocessed actual outer-loop position

Range: Floating-point

Units: Motor units

Motor[*x*].Pos contains the raw actual outer-loop position for the motor, before offsets such as backlash compensation and position-table compensation are added to obtain the value in **Motor[*x*].ActPos** that is compared with the commanded position to get the following error for the servo loop. Each cycle, the "delta position" value from the encoder conversion table entry addressed by **Motor[*x*].pEnc** is added to **Motor[*x*].Pos** to obtain the new cycle's "raw" motor position.

To calculate the position relative to the motor zero position, subtract the value of **Motor[x].HomePos**.

In most applications, **Motor[*x*].Pos** is simply an intermediate value in the actual position processing that is ignored or at most, occasionally monitored for debugging and analysis. However, with the motor disabled ("killed"), it is possible to write a value to **Motor[*x*].Pos**. This has the same effect as reading an absolute position sensor, and can be used to force a position value from a source not supported by the automatic position-read algorithms. With the motor disabled, the value written into **Motor[*x*].Pos**, possibly modified by backlash and compensation, is copied into the desired position register for the motor as well. An attempt to write to **Motor[*x*].Pos** when the motor is enabled (open-loop or closed-loop) will be rejected with an error.

Motor[*x*].Pos2

Description: Motor unprocessed actual inner-loop position

Range: Floating-point

Units: Inner-loop units

Motor[*x*].Pos2 contains the raw actual inner-loop position for the motor, before offsets such as position-table compensation are added to obtain the value in **Motor[*x*].ActPos2** that is used in the velocity feedback servo calculations. Each cycle, the "delta position" value from the encoder

conversion table entry addressed by **Motor[*x*].pEnc2** is added to **Motor[*x*].Pos2** to obtain the new cycle's "raw" motor position.

Motor[*x*].PosError

Description: Motor servo following error

Range: Floating point

Units: Motor units

Motor[*x*].PosError contains the present servo cycle's position error ("following error") value automatically calculated by subtracting **Motor[*x*].ActPos** from **Motor[*x*].DesPos**. It is scaled in the motor units. If the magnitude of this element exceeds that of saved setup element **Motor[*x*].FatalFeLimit**, the motor will be "killed" automatically.

Motor[*x*].Ppos

Description: Motor trajectory pre-filter input position

Range: Floating-point

Units: Motor units

Motor[*x*].Ppos contains the commanded position value for the motor that is the input to the trajectory pre-filter for the most recent cycle of the filter. It is only used if saved setup element **Motor[*x*].PreFilterEna** is set to a value greater than 0 to enable the filter. The output value from the filter is used to compute the value in **Motor[*x*].DesPos**.

Motor[*x*].PresBlSize

Description: Motor present backlash correction

Range: Floating-point

Units: Motor units

Motor[*x*].PresBlSize contains the present servo cycle's slew-rate limited backlash correction derived from the saved setup element **Motor[*x*].BlSize** and any compensation tables that use the **Motor[*x*].BlcompSize** element as a target register. It is added to the raw measured position in **Motor[*x*].Pos** in the calculation of the corrected value in **Motor[*x*].ActPos** that is used in the servo calculations.

If the motor is moving in the negative direction, this value increases by the value of **Motor[*x*].BlSlewRate** each real-time interrupt until it reaches the sum of the values in **Motor[*x*].BlSize** and **Motor[*x*].BlcompSize**. If the motor is moving in the negative direction, this value decreases by the value of **Motor[*x*].BlSlewRate** each real-time interrupt until it reaches 0.

Motor[*x*].PrevDesPos

Description: Motor previous net desired position

Range: Floating point

Units: Motor units

Motor[*x*].PrevDesPos contains the net desired position value for the servo cycle before the present cycle, the sum of the calculated trajectory position and the master position from the position following function for that cycle. It is scaled in the motor units and referenced to the power-up/reset position. It is used in the calculation of the velocity, acceleration and friction feedforward terms for the motor's servo algorithm.

To calculate the position relative to the motor zero position, subtract the value of **Motor[*x*].HomePos**.

Motor[*x*].PrevMasterPos

Description: Motor previous position-following master position

Range: Floating-point

Units: Motor units

Motor[*x*].PrevMasterPos contains accumulated position-following distance for the servo cycle before the present cycle. It is scaled in the motor units.

Motor[*x*].PrevPhaseEnc

Description: Motor previous phase source position

Range: $0 .. 2^{32} - 1$

Units: Commutation source-register units

Motor[*x*].PrevPhaseEnc contains the contents of the 32-bit register whose address is specified by **Motor[*x*].pPhaseEnc** for the phase cycle before the present cycle. It is used for rollover and commutation frequency calculations.

Motor[*x*].PrevPos2

Description: Motor previous inner loop feedback position

Range: Floating-point

Units: Inner-loop units

Motor[*x*].PrevPos2 contains the net feedback position value for the servo cycle before the present cycle for the inner (usually velocity) servo loop. It uses the value of **Motor[*x*].ActPos2** from the previous servo cycle. It is scaled in the units of the inner servo loop, which do not necessarily have to be the same as the motor units used in the outer loop and referenced to the power-up position. It is used in the calculation of the motor's actual velocity.

Motor[*x*].Pu*i*

Description: Motor trajectory pre-filter input position from *i* cycles ago

Range: Floating-point

Units: Motor units

Motor[*x*].Pu*i* contains the commanded position value for the motor that was the input to the trajectory pre-filter "***i***" cycles previously of the filter. The digit ***i*** can take a value from 1 through 4. The element is only used if saved setup element **Motor[*x*].PreFilterEna** is set to a value greater than 0 to enable the filter. The most recent input value can be found in **Motor[*x*].Ppos**. The output value from the filter is used to compute the value in **Motor[*x*].DesPos**.

Motor[*x*].ResMasterPos

Description: Motor master position residual

Range: Floating-point

Units: Motor units

Motor[*x*].ResMasterPos contains the position-following master position "residual" for the motor. If the value in the double-precision floating-point element **Motor[*x*].MasterPos** is so large that its 52-bit mantissa cannot retain full resolution, **Motor[*x*].ResMasterPos** is used to keep its "residual" value, thus maintaining full accuracy of the position calculations. **Motor[*x*].ResMasterPos** will only have a non-zero value if the magnitude of the value of **Motor[*x*].MasterPos** exceeds a value of about 2^{50}, or 10^{15}, motor units.

Motor[x].ResPos

Description: Motor actual outer-loop position residual

Range: Floating-point

Units: Motor units

Motor[x].ResPos contains the actual outer-loop position "residual" for the motor. If the value in the double-precision floating-point element **Motor[x].ActPos** is so large that its 52-bit mantissa cannot retain full resolution, **Motor[x].ResPos** is used to keep its "residual" value, thus maintaining full accuracy of the position calculations. **Motor[x].ResPos** will only have a non-zero value if the magnitude of the value of **Motor[x].ActPos** exceeds a value of about 2^{50}, or 10^{15}, motor units.

Motor[x].ResPos2

Description: Motor actual inner-loop position residual

Range: Floating-point

Units: Motor units

Motor[x].ResPos2 contains the actual inner-loop position "residual" for the motor. If the value in the double-precision floating-point element **Motor[x].ActPos2** is so large that its 52-bit mantissa cannot retain full resolution, **Motor[x].ResPos2** is used to keep its "residual" value, thus maintaining full accuracy of the position calculations. **Motor[x].ResPos2** will only have a non-zero value if the magnitude of the value of **Motor[x].ActPos2** exceeds a value of about 2^{50}, or 10^{15}, inner-loop units.

Motor[x].ServoOut

Description: Motor servo loop command output

Range: -32,768.0 .. 32,767.999

Units: 16-bit output units

Motor[x].ServoOut contains the present output value of the motor's servo loop. If Power PMAC is not performing phase commutation for the motor (**Motor[x].PhaseCtrl** bits 0 and 2 = 0), the sum of this value and the offset terms **Motor[x].CompDac** and **Motor[x].DacBias** is used to write to the integer register specified by **Motor[x].pDac**. If Power PMAC is performing phase commutation for the motor (**Motor[x].PhaseCtrl** bit 0 or 2 = 1), the sum of this value and the offset term **Motor[x].CompDac** is used as the "quadrature" current command input **Motor[x].IqCmd** to the commutation algorithm.

Motor[x].SoftLimit

Description: "Stopped on software position limit" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].SoftLimit** status bit is 1 if the motor has stopped because it hit either its positive or negative software overtravel limit, or has noted that it needs to stop the present motion at one of these limits, even if it is presently not in that limit. It is 0 at all other times, including when in a limit, but moving out of it. It is bit 30 of 32-bit element **Motor[*x*].Status[1]**.

Note

The action of any indefinite jog command **j+** or **j-** is automatically modified when software overtravel limits are active, causing **Motor[*x*].SoftLimit** to be set to 1 and **Motor[*x*].SoftLimitDir** to be set to the appropriate value as soon as the move starts. If another jog command (e.g. **j/**) whose action is not modified by the limits is issued before a limit is reached, **Motor[*x*].SoftLimit** is immediately cleared to 0.

Motor[x].SoftLimitDir

Description: "Soft limit direction" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].SoftLimitDir** status bit indicates which software overtravel limit caused a motor move to be stopped, modified, or rejected. It is only valid when **Motor[*x*].SoftLimit** is set to 1 to indicate that such an action has been taken. It is bit 27 of 32-bit element **Motor[*x*].Status[1]**.

Motor[*x*].SoftLimitDir is 0 if the positive software overtravel limit, as set by **Motor[*x*].MaxPos**, caused this action. It is 1 if the negative software overtravel limit, as set by **Motor[*x*].MinPos**, caused this action.

Note

The action of any indefinite jog command **j+** or **j-** is automatically modified when software overtravel limits are active, causing **Motor[*x*].SoftLimit** to be set to 1 and **Motor[*x*].SoftLimitDir** to be set to the appropriate value as soon as the move starts. If another jog command (e.g. **j/**) whose action is not modified by the limits is issued before a limit is reached, **Motor[*x*].SoftLimit** is immediately cleared to 0.

Motor[x].SoftMinusLimit

Description: "Software negative limit set" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].SoftMinusLimit** status bit is set to 1 when the motor has reached or exceeded its negative software limit as set by **Motor[*x*].MinPos** (which must be less than **Motor[*x*].MaxPos** to be active) and **Motor[*x*].SoftLimitOffset**. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously. It is bit 23 of 32-bit element **Motor[*x*].Status[0]**.

Note that **Motor[*x*].SoftLimitOffset** must be set to a negative value in order for this status bit to be set when at move execution time the motor's net desired position passes (**Motor[*x*].MinPos** - **Motor[*x*].SoftLimitOffset**).

Motor[x].SoftPlusLimit

Description: "Software positive limit set" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].SoftPlusLimit** status bit is set to 1 when the motor has reached or exceeded its positive software limit as set by **Motor[*x*].MaxPos** (which must be greater than **Motor[*x*].MinPos** to be active) and **Motor[*x*].SoftLimitOffset**. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously. It is bit 22 of 32-bit element **Motor[*x*].Status[0]**.

Note that **Motor[*x*].SoftLimitOffset** must be set to a negative value in order for this status bit to be set when at move execution time the motor's net desired position passes (**Motor[*x*].MaxPos** + **Motor[*x*].SoftLimitOffset**).

Motor[x].SpindleMotor

Description: Spindle motor definition status

Range: 0 .. 3

Units: Enumeration

The **Motor[*x*].SpindleMotor** status element is set to a value greater than 0 when the motor is defined as a "spindle axis" in a coordinate system (**#*x*->S**, **#*x*->S0**, **#*x*->S1**). It is set to 0 if it is not defined as a spindle axis in any coordinate system.

If the motor is defined as a spindle axis, the value of this element depends on the "time base" (override) it uses. The values it can take are:

- 1: Use the specified coordinate system's time base (**#*x*->S**)
- 2: Use Coordinate System 0's time base (**#*x*->S0**)
- 3: Use a fixed 100% time base (**#*x*->S1**)

Motor[*x*].SpindleMotor comprises bits 4 and 5 of 32-bit element **Motor[*x*].Status[0]**.

Motor[*x*].Status[0]

Description: Motor first status word

Range: $0 .. $FFFFFFFF

Units: Bit field

Motor[*x*].Status[0] is the first 32-bit status word for the motor, containing many individual global status bits. The following table provides a list of these status bits in the word:

Bit #	Hex Value	Element Name: Motor[*x*].*{element}*	Description
31	$80000000	**TriggerMove**	Trigger search move in progress
30	$40000000	**HomeInProgress**	Home search move in progress
29	$20000000	**MinusLimit**	Hardware negative limit set
28	$10000000	**PlusLimit**	Hardware positive limit set
27	$8000000	**FeWarn**	Warning following error
26	$4000000	**FeFatal**	Fatal following error
25	$2000000	**LimitStop**	Stopped on hardware position limit
24	$1000000	**AmpFault**	Amplifier fault error
23	$800000	**SoftMinusLimit**	Software negative limit set
22	$400000	**SoftPlusLimit**	Software positive limit set
21	$200000	**I2tFault**	Integrated current (I^2T) fault
20	$100000	**TriggerNotFound**	Trigger not found
19	$80000	**AmpWarn**	Amplifier warning
18	$40000	**EncLoss**	Sensor loss error
17	$20000	**AuxFault**	Auxiliary fault error
16	$10000	-	(*Reserved for future use*)
15	$8000	**HomeComplete**	Home complete
14	$4000	**DesVelZero**	Desired velocity zero
13	$2000	**ClosedLoop**	Closed-loop mode
12	$1000	**AmpEna**	Amplifier enabled
11	$800	**InPos**	In position
10	-	-	(*Reserved for future use*)
9	$200	**BlockRequest**	Block request flag set
8	$100	**PhaseFound**	Phase reference established
7	$80	**TriggerSpeedSel**	Triggered move speed select
6	$40	**GantryHomed**	Gantry homing complete
5	$20	**SpindleMotor** (bit 1)	Spindle axis definition status
4	$10	**SpindleMotor** (bit 0)	Spindle axis definition status
0 – 3	-	-	(*Reserved for future use*)

The value of this element is reported in response to the **#x?** query command. Its contents form the first eight hexadecimal digits (four bits per digit) of the response. The on-line query command **backup Motor[*x*].Status** causes Power PMAC to report the values of all of the individual elements as English text.

Each element is described in more detail in its own individual specification.

Motor[*x*].Status[1]

Description: Motor second status word

Range: $0 .. $FFFFFFFF

Units: Bit field

Motor[*x*].Status[1] is the second 32-bit status word for the motor, containing many individual global status bits. The following table provides a list of these status bits in the word:

Bit #	Hex Value	Element Name: Motor[*x*].*{element}*	Description
31	$80000000	**Csolve**	Motor used in PMATCH calculations
30	$40000000	**SoftLimit**	Stopped on software position limit
29	$20000000	**DacLimit**	Servo output limited
28	$10000000	**BlDir**	Backlash direction
27	$8000000	**SoftLimitDir**	SW limit causing move modification (0=pos, 1=neg)
0 – 26	-	-	(*Reserved*)

The value of this element is reported in response to the **#x?** query command. Its contents form the second eight hexadecimal digits (four bits per digit) of the response. The on-line query command **backup Motor[*x*].Status** causes Power PMAC to report the values of all of the individual elements as English text.

Each element is described in more detail in its own individual specification.

Motor[*x*].TraceCount

Description: Motor trace buffer segments or sections filled

Range: Non-negative integers

Units: Move segments or sections

Motor[*x*].TraceCount contains the number of move segments or sections currently filled in the motor's trace buffer. This number can range from 0 to **Motor[*x*].TraceSize**. Reverse motion through the trace buffer is limited to this number of move segments or sections.

Motor[x].TriggerMove

Description: "Trigger move search in progress" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].TriggerMove** status bit is set to 1 at the beginning of a move-until-trigger (homing search, jog-until-trigger, program rapid-mode move-until-trigger). It is set to 0 when the pre-specified trigger condition is found and the post-trigger move is started, or when the move ends without a trigger being found. It is bit 31 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].TriggerNotFound

Description: "Trigger not found" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].TriggerNotFound** status bit is set to 1 if a jog-until-trigger or program rapid-mode move-until-trigger ends without the pre-specified trigger condition being found. This is not an error condition, but subsequent actions will often depend on whether this bit is set or not. It is 0 at all other times, changing back to 0 when the next move is started. It is bit 20 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].TriggerSpeedSel

Description: "Trigger speed select" status bit

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].TriggerSpeedSel** status bit is set to 1 during a move-until-trigger if the move is done at the velocity specified by **Motor[*x*].MaxSpeed**. It is set to 0 if the move is done at the velocity specified by **Motor[*x*].JogSpeed**. It is for internal use so the processor knows which speed to use for the post-trigger portion of the move. It is bit 7 of 32-bit element **Motor[*x*].Status[0]**.

Motor[x].VxCouple

Description: Motor current-loop cross-coupling frequency

Range: Floating-point

Units: Commutation frequency

Motor[x].VxCouple contains the present commutation frequency used in the cross-coupling of the direct and quadrature current loops. It is calculated only if digital current loop operation is enabled for the motor. It is derived from the present motor velocity and the saved setup element **Motor[*x*].IxCoupleGain**.

Motor[*x*].Desired. Trajectory Substructure Elements

Each motor has a substructure **Motor[*x*].Desired** that contains the key information about the presently executing equations of commanded motion. These equations are derived from the commanded moves for the motor, whether directly with jog and home commands, or from programmed axes assigned to the motors.

Motor[*x*].Desired.Accel

Description: Motor present trajectory equation acceleration coefficient

Range: Floating-point

Units: Motor position units per millisecond^2 / 2

Motor[*x*].Desired.Accel contains the acceleration coefficient for the presently executing cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to one-half of the acceleration at the starting time for the equation for the present section or segment of motion.

Motor[*x*].Desired.Dwell

Description: Motor present trajectory dwell-in-progress flag

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].Desired.Dwell** status bit is set to 1 if the presently executing equations of motion come from a **`dwell`** command. It is set to 0 otherwise, even if the desired velocity is zero for other reasons.

Motor[*x*].Desired.Jerk

Description: Motor present trajectory equation jerk coefficient

Range: Floating-point

Units: Motor position units per millisecond^3 / 6

Motor[*x*].Desired.Jerk contains the jerk (rate of change of acceleration) coefficient for the presently executing cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to one-sixth of the jerk at the starting time for the equation for the present section or segment of motion.

Motor[*x*].Desired.Next

Description: Motor next trajectory equation index

Range: 0 .. 15

Units: Index value

Motor[*x*].Desired.Next contains the index value ***i*** of the substructure **Motor[*x*].New[*i*]** containing the equation of motion for the section or segment immediately following the presently executing section or segment.

Motor[*x*].Desired.Nsync

Description: Motor present synchronizing label value

Range: 0 .. 2^{31}-1

Units: Enumeration

Motor[*x*].Desired.Nsync contains the number of the most recently executed synchronizing line label value (from the delayed synchronous assignment of an **N*{data}*** command) associated with this move.

Motor[*x*].Desired.Pos

Description: Motor present trajectory equation position coefficient

Range: Floating-point

Units: Motor position units

Motor[*x*].Desired.Pos contains the position coefficient for the presently executing cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to the position at the starting time for the equation for the present section or segment of motion.

Motor[*x*].Desired.Time

Description: Motor present trajectory equation time span

Range: Floating-point

Units: Milliseconds

Motor[*x*].Desired.Time contains the time span for the presently executing cubic equation of trajectory commanded motion for the motor. This is the time from beginning to end of the section or segment when executing at 100% time base value.

Motor[*x*].Desired.TimerEnabled

Description: Motor timed move in progress status flag

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].Desired.TimerEnabled** status flag is set to 1 if the motor is presently executing a closed-loop move section of definite time, whether from a motor command or a programmed axis command. It is set to 0 otherwise. Note that the pre-trigger constant-speed section of a homing-search move and the constant-speed section of an indefinite jog command are not of a definite time, so this bit will be 0 during execution of those sections.

Motor[*x*].Desired.Vel

Description: Motor present trajectory equation velocity coefficient

Range: Floating-point

Units: Motor position units per millisecond

Motor[*x*].Desired.Vel contains the velocity coefficient for the presently executing cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to the velocity at the starting time for the equation for the present section or segment of motion.

Motor[*x*].New[*i*]. Trajectory Substructure Elements

Each motor has a substructure **Motor[*x*].New[*i*]** that contains the key information about the soon-to-be-executing buffered equations of commanded motion. These equations are derived from the commanded moves for the motor, whether directly with jog and home commands, or from programmed axes assigned to the motors. There are 16 of these substructures (***i*** = 0 to 15), permitting substantial buffering of upcoming move equations. The buffer is used in a rotary fashion.

Motor[*x*].New[*i*].Accel

Description: Motor buffered trajectory equation acceleration coefficient

Range: Floating-point

Units: Motor position units per millisecond^2 / 2

Motor[*x*].New[*i*].Accel contains the acceleration coefficient for the buffered cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to one-half of the acceleration at the starting time for the equation for this section or segment of motion.

Motor[*x*].New[*i*].Dwell

Description: Motor buffered trajectory dwell-in-progress flag

Range: 0 .. 1

Units: Boolean

The **Motor[*x*].Desired.Dwell** status bit is set to 1 if the buffered equations of motion for this section come from a **`dwell`** command. It is set to 0 otherwise, even if the desired velocity is zero for other reasons.

Motor[*x*].New[*i*].Jerk

Description: Motor buffered trajectory equation jerk coefficient

Range: Floating-point

Units: Motor position units per millisecond^3 / 6

Motor[*x*].New[*i*].Jerk contains the jerk (rate of change of acceleration) coefficient for the presently buffered cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to one-sixth of the jerk at the starting time for the equation for this section or segment of motion.

Motor[*x*].New[*i*].Nsync

Description: Motor buffered trajectory synchronizing label value

Range: 0 .. 2^{31}-1

Units: Enumeration

Motor[*x*].Desired.Nsync contains the number of the synchronizing line label value (from the delayed synchronous assignment of an **N{data}** command) associated with this move section.

Motor[*x*].New[*i*].Pos

Description: Motor buffered trajectory equation position coefficient

Range: Floating-point

Units: Motor position units

Motor[*x*].New[*i*].Pos contains the position coefficient for the buffered cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to the position at the starting time for the equation for this section or segment of motion.

Motor[*x*].New[*i*].Time

Description: Motor buffered trajectory equation time span

Range: Floating-point

Units: Milliseconds

Motor[*x*].New[*i*].Time contains the time span for the buffered cubic equation of trajectory commanded motion for the motor. This is the time from beginning to end of the section or segment when executing at 100% time base value.

Motor[*x*].New[*i*].Vel

Description: Motor buffered trajectory equation velocity coefficient

Range: Floating-point

Units: Motor position units per millisecond

Motor[*x*].New[*i*].Vel contains the velocity coefficient for the buffered cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to the velocity at the starting time for the equation for this section or segment of motion.

Motor[x].Servo. Substructure Status Elements

The elements in this section provide the servo-related information for the motor that is automatically calculated by Power PMAC. These elements are part of the motor's **Servo** sub-structure.

Motor[x].Servo.EstGain

Description: Present estimated plant gain for adaptive control

Range: Non-negative floating-point

Units: (Motor units per second^2) / 16-bit command unit

Motor[*x*].Servo.EstGain contains the present estimated "plant gain" value calculate by the recursive estimation algorithm of the adaptive servo control, based on the results of the last **Motor[*x*].Servo.EstTime** servo cycles. It is only used if the adaptive servo control algorithm is selected. It is expressed as the ratio of the acceleration in motor units per second^2 to 16-bit command output units.

This value is used to compute the compensating adaptive gain value **Motor[*x*].Servo.GainFactor** so the total loop gain remains as constant as possible.

Motor[x].Servo.EstTimer

Description: Number of servo cycles presently used in adaptation

Range: Non-negative integer

Units: Servo cycles

Motor[*x*].Servo.EstTimer contains the value of the number of consecutive servo cycles that the proper conditions have existed for plant gain estimation for the motor in the adaptive servo control algorithm. It is only used if the adaptive servo control algorithm is selected.

The conditions for plant gain estimation are that the motor desired acceleration value be non-zero and the magnitude of the servo command output be greater than **Motor[*x*].Servo.EstMinDac**. When the value of **Motor[*x*].Servo.EstTimer** exceeds that of saved setup element **Motor[*x*].Servo.EstTime**, Power PMAC will use its calculated value of the plant gain (**Motor[*x*].Servo.EstGain**) in the adaptation of the servo loop control.

In any servo cycle that the conditions the proper conditions for plant gain estimation are not met, the value of **Motor[*x*].Servo.EstTimer** is set to 0.

Motor[x].Servo.GainFactor

Description: Present relative servo gain for adaptive control

Range: Non-negative floating-point

Units: none (ratio)

Motor[x].Servo.GainFactor contains the present relative servo-loop gain value resulting from the adaptive control algorithm's latest estimated plant gain calculation. It is only used if the adaptive servo control algorithm is selected. It is used to rescale the standard saved servo loop gains, with the purpose of keeping the resulting loop gain and servo loop performance as constant as possible. Note that this value is not saved; at power-on/reset, it is automatically set to 1.0.

Motor[x].Servo.GainFactor will not go below the value of saved setup element **Motor[x].Servo.MinGainFactor**, and it will not go above the value of saved setup element **Motor[x].Servo.MaxGainFactor**.

Motor[x].Servo.Integrator

Description: Integrated position error output

Range: Floating-point

Units: 16-bit servo command output bits

Motor[x].Servo.Integrator contains the present output value of the standard servo position-error integrator. It is added to the output of other servo terms to get the net servo output. Its magnitude is limited by saved setup element **Motor[x].Servo.MaxInt**.

Motor[x].Servo.Status

Description: Output hysteretic deadband state

Range: 0 .. 1

Units: Boolean

Motor[x].Servo.Status contains the present state of the servo loop's output hysteretic deadband. It is 0 if the deadband is "off" and the servo-loop output is active. It is 1 if the deadband is "on" and the servo-loop output is forced to 0. Power PMAC only computes this deadband state when motor desired velocity is exactly equal to 0.0.

Motor[x].Servo.ua*i*

Description: Desired-position polynomial input from *i* cycles ago

Range: Floating-point

Units: motor units

Motor[*x*].Servo.ua*i* contains the input value from ***i*** servo cycles before of the "A" servo polynomial that acts on the net desired position into the servo algorithm. It is multiplied by corresponding saved setup element servo gain **Motor[*x*].Servo.Ka*i***.

Saved setup element **Motor[*x*].Ctrl** must select a servo algorithm that uses the polynomial filters in order for Power PMAC to use these terms. There are 7 of these elements: **ua1** through **ua7**. If saved setup element **Motor[*x*].Servo.SwPoly7** is set to the default value of 0, only **ua1** and **ua2** are used. If it is set to 1, all 7 values are used.

Motor[*x*].Servo.ub*i*

Description: Actual-position polynomial input from *i* cycles ago

Range: Floating-point

Units: motor units

Motor[*x*].Servo.ub*i* contains the input value from ***i*** servo cycles before of the "B" servo polynomial that acts on the net actual position into the servo algorithm. It is multiplied by corresponding saved setup element servo gain **Motor[*x*].Servo.Kb*i***.

Saved setup element **Motor[*x*].Ctrl** must select a servo algorithm that uses the polynomial filters in order for Power PMAC to use these terms. There are 7 of these elements: **ub1** through **ub7**. If saved setup element **Motor[*x*].Servo.SwPoly7** is set to the default value of 0, only **ub1** and **ub2** are used. If it is set to 1, all 7 values are used.

Motor[*x*].Servo.uc*i*

Description: Servo-error polynomial input from *i* cycles ago

Range: Floating-point

Units: motor units

Motor[*x*].Servo.uc*i* contains the input value from ***i*** servo cycles before of the "C" and "D" servo polynomial numerator and denominator that act on the position (following) error in the servo algorithm. It is multiplied by corresponding saved setup element servo gain **Motor[*x*].Servo.Kc*i***.

Saved setup element **Motor[*x*].Ctrl** must select a servo algorithm that uses the polynomial filters in order for Power PMAC to use these terms. There are 7 of these elements: **uc1** through **uc7**. If saved setup element **Motor[*x*].Servo.SwPoly7** is set to the default value of 0, only **uc1** and **uc2** are used. If it is set to 1, all 7 values are used.

Motor[*x*].Servo.ud*i*

Description: Servo-error polynomial output from *i* cycles ago

Range: Floating-point

Units: motor units

Motor[*x*].Servo.ud*i* contains the output value from ***i*** servo cycles before of the "C" and "D" servo polynomial numerator and denominator that act on the net desired position into the servo algorithm. It is multiplied by corresponding saved setup element servo gain **Motor[*x*].Servo.Kd*i***.

Saved setup element **Motor[*x*].Ctrl** must select a servo algorithm that uses the polynomial filters in order for Power PMAC to use these terms. There are 7 of these elements: **ud1** through **ud7**. If saved setup element **Motor[*x*].Servo.SwPoly7** is set to the default value of 0, only **ud1** and **ud2** are used. If it is set to 1, all 7 values are used.

Motor[*x*].Servo.ue*i*

Description: Inner-loop velocity polynomial output from *i* cycles ago

Range: Floating-point

Units: motor units

Motor[*x*].Servo.ue*i* contains the output value from ***i*** servo cycles previous of the "E" servo polynomial that acts on the inner (velocity) loop position into the servo algorithm. It is multiplied by corresponding saved setup element servo gain **Motor[*x*].Servo.Ke*i***.

Saved setup element **Motor[*x*].Ctrl** must select a servo algorithm that uses the polynomial filters in order for Power PMAC to use these terms. There are 2 of these elements: **ue1** and **ue2**.

Motor[*x*].Servo.uf*i*

Description: Desired-velocity polynomial output from *i* cycles ago

Range: Floating-point

Units: motor units

Motor[*x*].Servo.uf*i* contains the output value from ***i*** servo cycles previous of the "F" servo polynomial that acts on the net desired velocity in the feedforward path of the servo algorithm. It is multiplied by corresponding saved setup element servo gain **Motor[*x*].Servo.Kf*i***.

Saved setup element **Motor[*x*].Ctrl** must select a servo algorithm that uses the polynomial filters in order for Power PMAC to use these terms. There are 2 of these elements: **uf1** and **uf2**.

Motor[*x*].Servo.Xint

Description: Cross-coupled integrated position error-difference output

Range: Floating-point

Units: 16-bit servo command output bits

Motor[*x*].Servo.Xint contains the present output value of the cross-coupled servo position-error-difference integrator. It is added to the output of other servo terms to get the net servo output. It is only used in the cross-coupled gantry control algorithm.

Motor[*x*].TraceData[*i*]. Trajectory Substructure Elements

Each motor has a substructure **Motor[*x*].TraceData[*i*]** that contains the key information about already executed equations of commanded motion if the trace buffer is activated by setting **Motor[*x*].TraceSize** greater than 0. These equations are derived from the commanded moves for the motor, whether directly with jog and home commands, or from programmed axes assigned to the motors. The index value ***i*** can range from 0 to **TraceSize** - 1. The trace buffer permits reverse execution of motion with negative time base values. The buffer is used in a rotary fashion.

Motor[*x*].TraceData[*i*].Accel

Description: Motor trace buffer trajectory equation acceleration coefficient

Range: Floating-point

Units: Motor position units per millisecond^3 / 2

Motor[*x*].TraceData[*i*].Accel contains the acceleration coefficient for the buffered cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to one-half of the acceleration at the starting time for the equation for this section or segment of motion.

Motor[*x*].TraceData[*i*].Jerk

Description: Motor trace buffer trajectory equation jerk coefficient

Range: Floating-point

Units: Motor position units per millisecond^2 / 6

Motor[*x*].TraceData[*i*].Jerk contains the jerk (rate of change of acceleration) coefficient for the buffered cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to one-sixth of the jerk at the starting time for the equation for this section or segment of motion.

Motor[*x*].TraceData[*i*].Nsync

Description: Motor trace buffer synchronizing line label

Range: 0 .. 2^{32}-1

Units: Enumeration

Motor[*x*].TraceData[*i*].Nsync contains the synchronizing line label value for the section or segment in the trace buffer for the motor. This is the same value as was in **Coord[*x*].Nsync** when this section or segment was originally executing.

Motor[*x*].TraceData[*i*].Pos

Description: Motor trace buffer trajectory equation position coefficient

Range: Floating-point

Units: Motor position units

Motor[*x*].TraceData[*i*].Pos contains the position coefficient for the buffered cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to the position at the starting time for the equation for this section or segment of motion.

Motor[*x*].TraceData[*i*].Time

Description: Motor trace buffer trajectory equation time span

Range: Floating-point

Units: Milliseconds

Motor[*x*].TraceData[*i*].Time contains the time span for the buffered cubic equation of trajectory commanded motion for the motor. This is the time from beginning to end of the section or segment when executing at 100% time base value.

Motor[*x*].TraceData[*i*].TPExec

Description: Motor trace buffer target-position pointer

Range: Power PMAC memory addresses

Units: Power PMAC memory addresses

Motor[*x*].TraceData[*i*].TPExec contains address of the target- position buffer entry (if defined) for this section or segment of the trace buffer for the motor. This is primarily for internal use, permitting the user to obtain programmed-move target-position and distance-to-go information corresponding to this section or segment.

Motor[*x*].TraceData[*i*].Vel

Description: Motor trace buffer trajectory equation velocity coefficient

Range: Floating-point

Units: Motor position units per millisecond

Motor[*x*].TraceData[*i*].Vel contains the velocity coefficient for the buffered cubic equation of trajectory commanded motion for the motor. This coefficient is equivalent to the velocity at the starting time for the equation for this section or segment of motion.

Motor[x].TraceExec. Trajectory Substructure Elements

Each motor has a substructure **Motor[*x*].TraceExec** that contains the key information about the equations of commanded motion currently executing from the trace buffer. It can be used if the trace buffer is activated by setting **Motor[*x*].TraceSize** greater than 0. These equations are derived from the commanded moves for the motor, whether directly with jog and home commands, or from programmed axes assigned to the motors. The trace buffer permits reverse execution of motion with negative time base values. The buffer is used in a rotary fashion.

Motor[*x*].TraceExec.Accel

Description: Motor executing trace buffer trajectory equation acceleration coefficient

Range: Floating-point

Units: Motor position units per millisecond^2 / 2

Motor[*x*].TraceExec.Accel contains the acceleration coefficient for the cubic equation of trajectory commanded motion for the section or segment currently executing from the trace buffer for the motor. This coefficient is equivalent to one-half of the acceleration at the starting time for the equation for this section or segment of motion.

Motor[*x*].TraceExec.Jerk

Description: Motor executing trace buffer trajectory equation jerk coefficient

Range: Floating-point

Units: Motor position units per millisecond^3 / 6

Motor[*x*].TraceExec.Jerk contains the jerk (rate of change of acceleration) coefficient for the cubic equation of trajectory commanded motion for the section or segment currently executing from the trace buffer for the motor. This coefficient is equivalent to one-sixth of the jerk at the starting time for the equation for this section or segment of motion.

Motor[*x*].TraceExec.Nsync

Description: Motor executing trace buffer synchronizing line label

Range: 0 .. 2^{32}-1

Units: Enumeration

Motor[*x*].TraceExec.Nsync contains the synchronizing line label value for the section or segment currently executing from the trace buffer for the motor. This is the same value as was in **Coord[*x*].Nsync** when this section or segment was originally executing.

Motor[x].TraceExec.Pos

Description: Motor executing trace buffer trajectory equation position coefficient

Range: Floating-point

Units: Motor position units

Motor[x].TraceExec.Pos contains the position coefficient for the cubic equation of trajectory commanded motion for the section or segment currently executing from the trace buffer for the motor. This coefficient is equivalent to the position at the starting time for the equation for this section or segment of motion.

Motor[x].TraceExec.Time

Description: Motor executing trace buffer trajectory equation time span

Range: Floating-point

Units: Milliseconds

Motor[x].TraceExec.Time contains the time span for the cubic equation of trajectory commanded motion for the section or segment currently executing from the trace buffer for the motor. This is the time from beginning to end of the section or segment when executing at 100% time base value.

Motor[x].TraceExec.TPExec

Description: Motor executing trace buffer target-position pointer

Range: Power PMAC memory addresses

Units: Power PMAC memory addresses

Motor[x].TraceExec.TPExec contains address of the target- position buffer entry (if defined) for the section or segment currently executing from the trace buffer for the motor. This is primarily for internal use, permitting the user to obtain programmed-move target-position and distance-to-go information corresponding to this section or segment.

Motor[*x*].TraceExec.Vel

Description: Motor executing trace buffer trajectory equation velocity coefficient

Range: Floating-point

Units: Motor position units per millisecond

Motor[*x*].TraceExec.Vel contains the velocity coefficient for the cubic equation of trajectory commanded motion for the section or segment currently executing from the trace buffer for the motor. This coefficient is equivalent to the velocity at the starting time for the equation for this section or segment of motion.

MuxIo. Status Data Structure Elements

The **MuxIo.** data structure comprises the multiplexed I/O ("thumbwheel port") settings for the Power PMAC. The elements in this structure provide access to multiplexed IO devices which use the serial thumbwheel-multiplexer protocol, such as ACC-34A and ACC-34AA.

MuxIo.PortA[*n*].Parity

Description: Received/calculated parity word for Port A input/output word

Range: $00 .. FF (0 .. 255)

Units: Enumeration

MuxIo.PortA[*n*].Parity holds the last received or calculated parity word based upon **MuxIo.PortA[*n*].Dir** setting. The index ***n*** is the address of multiplexed I/O device as selected by DIP switches on the device.

If **MuxIo.PortA[*n*].Dir** is set to the default value of 0, declaring the port as an input port, **MuxIo.PortA[*n*].Parity** holds the last received parity word.

If **MuxIo.PortA[*n*].Dir** is set to a value of 1, declaring the port as an output port, **MuxIo.PortA[*n*].Parity** holds the last calculated parity word, based upon **MuxIo.PortA[*n*].Data** image word.

If **MuxIo.PortA[*n*].AutoParityCheck** is set to default value of 0, disabling the parity checking function for the port, the input word is always stored in **MuxIo.PortA[*n*].Data** as received. Although the parity value for each received input word is stored in **MuxIo.PortA[*n*].Parity** and result of its comparison with calculated parity word is stored in **MuxIo.PortA[*n*].ParityStatus**, no automatic action, such as rejection of the data, is taken. This method is useful if the user decides to take some other action other than rejecting the data and keeping the previous value in **MuxIo.PortA[*n*].Data**.

If **MuxIo.PortA[*n*].AutoParityCheck** is set to a value of 1, enabling the parity checking function for the port, the input word is stored in **MuxIo.PortA[*n*].Data** only if the received parity value, which is stored in **MuxIo.PortA[*n*].Parity**, matches with calculated parity value for each received input word. The result of this comparison is stored in **MuxIo.PortA[*n*].ParityStatus**. In case of a mismatch, the received input word is rejected and previous value of **MuxIo.PortA[*n*].Data** is kept.

MuxIo.PortB[*n*].Parity

Description: Received/calculated parity word for Port B input/output word

Range: $00 .. FF (0 .. 255)

Units: Enumeration

MuxIo.PortB[*n*].Parity holds the last received or calculated parity word based upon **MuxIo.PortB[*n*].Dir** setting. The index ***n*** is the address of multiplexed I/O device as selected by DIP switches on the device.

If **MuxIo.PortB[*n*].Dir** is set to the default value of 0, declaring the port as an input port, **MuxIo.PortB[*n*].Parity** holds the last received parity word.

If **MuxIo.PortB[*n*].Dir** is set to a value of 1, declaring the port as an output port, **MuxIo.PortB[*n*].Parity** holds the last calculated parity word, based upon **MuxIo.PortB[*n*].Data** image word.

If **MuxIo.PortB[*n*].AutoParityCheck** is set to default value of 0, disabling the parity checking function for the port, the input word is always stored in **MuxIo.PortB[*n*].Data** as received. Although the parity value for each received input word is stored in **MuxIo.PortB[*n*].Parity** and result of its comparison with calculated parity word is stored in **MuxIo.PortB[*n*].ParityStatus**, no automatic action, such as rejection of the data, is taken. This method is useful if the user decides to take some other action other than rejecting the data and keeping the previous value in **MuxIo.PortB[*n*].Data**.

If **MuxIo.PortB[*n*].AutoParityCheck** is set to a value of 1, enabling the parity checking function for the port, the input word is stored in **MuxIo.PortB[*n*].Data** only if the received parity value, which is stored in **MuxIo.PortB[*n*].Parity**, matches with calculated parity value for each received input word. The result of this comparison is stored in **MuxIo.PortB[*n*].ParityStatus**. In case of a mismatch, the received input word is rejected and previous value of **MuxIo.PortB[*n*].Data** is kept.

MuxIo.PortA[*n*].ParityStatus

Description: Parity word comparison result for Port A

Range: 0 .. 1

Units: Boolean

MuxIo.PortA[*n*].ParityStatus indicates the comparison result between the calculated parity word and the transferred one for the most recent read or write access on the port. A value of 0 indicates a mismatch (false) and a value of 1 indicates a correct match (true).

If **MuxIo.PortA[*n*].Dir** is set to the default value of 0, declaring the port as an input port, the **MuxIo.PortA[*n*].Parity** holds the last recived parity word. This value is then compared with a calculated parity word based upon the received input word and result of this comparison is stored in **MuxIo.PortA[*n*].ParityStatus**. If **MuxIo.PortA[*n*].AutoParityCheck** is set to default value of 0, input word is stored in **MuxIo.PortA[*n*].Data** as received.

If **MuxIo.PortA[*n*].AutoParityCheck** is set to a value of 1, input word is stored in **MuxIo.PortA[*n*].Data** only if **MuxIo.PortA[*n*].ParityStatus** holds a value of 1 (true). In case of a mismatch, the received input word is rejected and previous value of **MuxIo.PortA[*n*].Data** is kept.

If **MuxIo.PortA[*n*].Dir** is set to 1, declaring the port as an output port, the **MuxIo.PortA[*n*].Parity** holds the last calculated parity word and transmitted ahead of the output image word to the multiplexed device. Upon transmission of the image word LSB, a confirmation bit is sent back from the multiplexed device indicating the result of comparison between transmitted **MuxIo.PortA[*n*].Parity** and locally calculated parity word by multiplexed I/O device. This bit is stored in **MuxIo.PortA[*n*].ParityStatus** for the user to be able to check, but no automatic action is taken based on its setting.

MuxIo.PortB[*n*].ParityStatus

Description: Parity word comparison result for Port B

Range: 0 .. 1

Units: Boolean

MuxIo.PortB[*n*].ParityStatus indicates the comparison result between the calculated parity word and the transferred one for the most recent read or write access on the port. A value of 0 indicates a mismatch (false) and a value of 1 indicates a correct match (true).

MuxIo.PortB[*n*].Dir is set to the default value of 0, declaring the port as an input port, the **MuxIo.PortB[*n*].Parity** holds the last recived parity word. This value is then compared with a calculated parity word based upon the received input word and result of this comparison is stored in **MuxIo.PortB[*n*].ParityStatus.** If **MuxIo.PortB[*n*].AutoParityCheck** is set to default value of 0, input word is stored in **MuxIo.PortB[*n*].Data** as received.

If **MuxIo.PortB[*n*].AutoParityCheck** is set to a value of 1, input word is stored in **MuxIo.PortB[*n*].Data** only if **MuxIo.PortB[*n*].ParityStatus** holds a value of 1 (true). In case of a mismatch, the received input word is rejected and previous value of **MuxIo.PortB[*n*].Data** is kept.

If **MuxIo.PortA[*n*].Dir** / **MuxIo.PortB[*n*].Dir** is set to 1, declaring the port as an output port, the **MuxIo.PortA[*n*].Parity** / **MuxIo.PortB[*n*].Parity** holds the last calculated parity word and transmitted ahead of the output image word to the multiplexed device. Upon transmission of the image word LSB, a confirmation bit is sent back from the multiplexed device indicating the result of comparison between transmitted **MuxIo.PortA[*n*].Parity** / **MuxIo.PortB[*n*].Parity** and locally calculated parity word by multiplexed I/O device. This bit is stored in **MuxIo.PortA[*n*].ParityStatus** / **MuxIo.PortB[*n*].ParityStatus** for the user to be able to check, but no automatic action is taken based on its setting.

Plc[*i*]. PLC Program Status Data Structure Elements

The **Plc[*i*].** data structure contains elements pertaining to Script program PLC*i*.

Plc[*i*].Active

Description: PLC program activation status

Range: 0 .. 1

Units: Boolean

Plc[*i*].Active contains the present activation status for the Script PLC*i* program. If the program is enabled, whether running or paused, **Plc[*i*].Active** is 1. If the program is disabled, **Plc[*i*].Active** is 0.

Plc[*i*].Running

Description: PLC program execution status

Range: 0 .. 1

Units: Boolean

Plc[*i*].Running contains the present execution status for the Script PLC*i* program. If the program is presently running – enabled and not paused, **Plc[*i*].Running** is 1. If the program is disabled or paused, **Plc[*i*].Running** is 0.

Plc[*i*].MaxTime

Description: Maximum calculation time for PLC program execution

Range: Non-negative floating-point

Units: µsec

Plc[*i*].MaxTime contains the longest time in microseconds for a scan of the Script PLC*i* program (**Plc[*i*].Time** for a scan) since this element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSf** (µsec / clock cycle).

Plc[*i*].MinTime

Description: Minimum calculation time for PLC program execution

Range: Non-negative floating-point

Units: µsec

Plc[*i*].MinTime contains the shortest time in microseconds for a scan of the Script PLC*i* program (**Plc[*i*].Time** for a scan) since this element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSf** (µsec / clock cycle).

Plc[*i*].Time

Description: Latest calculation time for PLC program execution

Range: Non-negative floating-point

Units: µsec

Plc[*i*].Time contains the time in microseconds from beginning to end of the most recent scan for the Script PLC*i* program. The time is from the start to the end of a scan; if interrupted by higher-priority tasks (phase, servo, real-time interrupt), it will include the time in those tasks as well. If the PLC program is disabled, it will have a value of zero.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSf** (µsec / clock cycle).

Plc[*i*].Ldata Local Data Status Elements

The **Plc[*i*].Ldata** substructure contains elements specifying generalized local data for the Script PLC program. This substructure is identical to the **Ldata** substructure for each coordinate system and each communications thread. Refer to the description of the **Ldata** structure above.

PowerBrick[*i*]. Status Data Structure Elements

The **PowerBrick[*i*]** data structure name is an alias in the Script environment for the underlying **Gate3[*i*]** data structure. The data structure elements for the Power Brick servo interface board are listed under the **Gate3[*i*]** data structure, above.

Sys. Global Status Data Structure Elements

The status elements documented in this section are "global", concerning operation of the full Power PMAC system.

Sys.AbortAll

Description: "Abort-all active" status bit

Range: 0 .. 1

Units: Boolean

The **Sys.AbortAll** status bit is set to 1 when Power PMAC is stopping or has stopped due to the "Abort All" input, even if the input is no longer set in the abort state.

Sys.AbortAll is set back to 0 when the input is not set and any motor or coordinate system is commanded to exit its final "abort all" state.

Sys.AbortAll is bit 9 of 32-bit element **Sys.Status**.

Sys.AbortAllCount

Description: Cumulative number of scans finding global abort input

Range: 0 .. 255

Units: Scans

Sys.AbortAllCount contains the present accumulated number of scans in which the "global abort" state was detected as set. Each scan in which the abort input is detected as set, **Sys.AbortAllCount** is incremented by one, until it exceeds the fault threshold specified by **Sys.AbortAllLimit**. Each scan in which the abort input is not detected as set, **Sys.AbortAllCount** is decremented by one, until it reaches zero.

The register for the global abort input bit is specified by **Sys.pAbortAll**. The bit within this register is specified by **Sys.AbortAllBit**.

Sys.AdaptiveCtrl

Description: Address of adaptive servo control algorithm

Range: Positive integer

Units: Power PMAC memory addresses

Sys.AdaptiveCtrl contains the address of the start of the adaptive servo control algorithm. If **Motor[*x*].Ctrl** is set to **Sys.AdaptiveCtrl**, the motor will execute this algorithm as its servo algorithm each cycle. It is rarely necessary to know the numerical value of this address.

Sys.BgDeltaTime

Description: Time elapsed between start of last two background task scans

Range: Positive floating-point

Units: Microseconds

Sys.BgDeltaTime contains the time elapsed in microseconds between the start of software tasks executing in the most recent two background task scans. Note that while the background scan is set to start on a timed basis as set by saved setup element **Sys.BgSleepTime** (1 millisecond by default), because it has lower priority than all interrupt-based tasks, the actual time between the start of 2 consecutive scans can vary significantly.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.BgForceInOr

Description: Background buffered input forcing words logical OR

Range: 0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Sys.BgForceInOr is a 32-bit register in which each bit contains the logical OR of the matching bits of **BufIo[*i*].ForceInOn** and **BufIo[*i*].ForceInOff** for all buffered input registers that are currently being scanned at the beginning of each background cycle.

Sys.BgForceInOr is intended to provide the user with a quick check on whether any of the background buffered inputs are still being forced on or off after a debugging section, as this forcing could interfere with proper operation of the machine. It has no internal use in the Power PMAC.

Each bit of **Sys.BgForceInOr** is logically ORed with the matching bits of **Sys.BgForceOutOr**, **Sys.FgForceInOr**, and **Sys.FgForceOutOr** to compute an overall status word **Sys.ForceOr** that is usually used as a first check as to whether any forcing bits are set.

For example, if **Sys.BgForceInOr** returns a value of $00004100, this means that bit 14 is set in at least one of the forcing words for background buffered inputs, as is bit 8. It does not indicate which of the forcing words has either of these bits set.

Sys.BgForceInOr is new in V2.2 firmware, released 3rd quarter 2016.

Sys.BgForceOutOr

Description: Background buffered output forcing words logical OR

Range: 0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Sys.BgForceOutOr is a 32-bit register in which each bit contains the logical OR of the matching bits of **BufIo[*i*].ForceOutOn** and **BufIo[*i*].ForceOutOff** for all buffered output registers that are currently being scanned at the end of each background cycle.

Sys.BgForceOutOr is intended to provide the user with a quick check on whether any of the background buffered outputs are still being forced on or off after a debugging section, as this forcing could interfere with proper operation of the machine. It has no internal use in the Power PMAC.

Each bit of **Sys.BgForceOutOr** is logically ORed with the matching bits of **Sys.BgForceInOr**, **Sys.FgForceInOr**, and **Sys.FgForceOutOr** to compute an overall status word **Sys.ForceOr** that is usually used as a first check as to whether any forcing bits are set.

For example, if **Sys.BgForceOutOr** returns a value of $00020800, this means that bit 17 is set in at least one of the forcing words for background buffered outputs, as is bit 11. It does not indicate which of the forcing words has either of these bits set.

Sys.BgForceOutOr is new in V2.2 firmware, released 3rd quarter 2016.

Sys.BgTime

Description: Latest calculation time for background task scan

Range: Positive floating-point

Units: Microseconds

Sys.BgTime contains the time in microseconds from beginning to end of the most recent background task scan. The time is from the start to the end of a scan; if interrupted by higher-priority tasks (phase, servo, real-time interrupt), it will include the time in those tasks as well.

For single-core processors, for which the core splits its time between background and interrupt tasks, the time actually spent in background tasks for this most recent background cycle can be calculated by the following sequence of equations:

$$BgTaskTime'' = Sys.BgTime - \text{int}\left(\frac{Sys.BgTime}{Sys.PhaseDeltaTime}\right) * Sys.PhaseTime$$

$$BgTaskTime' = BgTaskTime'' - \text{int}\left(\frac{BgTaskTime''}{Sys.ServoDeltaTime}\right) * ServoTaskTime$$

$$BgTaskTime = BgTaskTime' - \text{int}\left(\frac{BgTaskTime'}{Sys.RtIntDeltaTime}\right) * RtIntTaskTime$$

where:

$$ServoTaskTime = Sys.ServoTime - \left[\text{int}\left(\frac{Sys.ServoTime}{Sys.PhaseDeltaTime}\right) + 1\right] * Sys.PhaseTime$$

and:

$$RtIntTaskTime = Sys.RtIntTime - \left[\text{int}\left(\frac{Sys.RtIntTime}{Sys.PhaseDeltaTime}\right) + 1\right] * Sys.PhaseTime - \left[\text{int}\left(\frac{Sys.RtIntTime}{Sys.ServoDeltaTime}\right) + 1\right] * ServoTaskTime$$

For multi-core processors, interrupt tasks are performed on a separate core from background tasks, so the time reported in **Sys.BgTime** is the actual time spent in the most recent background task cycle.

If directly queried from an on-line command, the processing of communications may cause memory caching issues that increase the reported time. It is better to query the averaged time in status element **Sys.FltrBgTime**.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.BgWdTimer

Description: Background cycles executed since last RTI

Range: Non-negative integer

Units: Background software cycles

Sys.BgWdTimer contains the number of background software cycles that have executed since the most recent real-time interrupt (RTI) period. If this value exceeds that of saved setup element **Sys.BgWDTReset** (or exceeds a value of 10 if **Sys.BgWDTReset** is set to its default value of 0), a "soft" watchdog timer trip will occur, disabling motion programs and servo tasks, forcing

hardware outputs into their "reset" state, and setting status element **Sys.WDTFault** to 2. Such a fault is indicative of a failure of interrupt tasks to occur.

Sys.BufPos[*i*][*j*]

Description: Extended position buffer elements

Range: Floating-point

Units: Motor position units

Sys.BufPos[*i*][*j*] contains the buffered actual position of a servo cycle for a motor utilizing the extended position buffering function. The first index (***i***) is the buffer number, with a range from 0 to 7, and the second index (***j***) is the servo cycle index, with a range from 0 to 255.

A motor uses this function if saved setup element **Motor[*x*].pBufPos** (for outer-loop actual position) or **Motor[*x*].pBufPos2** (for inner-loop actual position) is set to the address of the beginning of one of these buffers (to **Sys.BufPos[*i*][0].a**). Then each servo cycle, it will copy its actual position value to **Sys.BufPos[*i*][*j*]**, where ***i*** is the buffer number, and ***j*** is equal to the value in the low 8 bits of automatically incrementing global status element **Sys.ServoCount** for the servo cycle.

In this way, the buffers are used in a rotary fashion, always containing the most recent 256 position values from the motor. These values can be used for custom application-specific features. Note that there is no protection against multiple motors writing to the same buffer; the higher-numbered motor would overwrite the value from the lower-numbered motor.

Sys.BufSizeErr

Description: "Insufficient buffer memory" status bit

Range: 0 .. 1

Units: Boolean

The **Sys.BufSizeErr** status bit is set to 1 at power-on/reset if the active memory (RAM) in Power PMAC does not have enough capacity to reserve the saved project's declared buffer sizes for programs, tables, lookahead, and user shared memory. If this occurs, these buffer sizes are set to the default sizes of 1MB each for tables and user shared memory, and 16MB each for programs and lookahead.

Sys.BufSizeErr is set to 0 on power-on/reset when there is sufficient memory for the declared buffer sizes.

Sys.BufSizeErr is bit 10 of 32-bit element **Sys.Status**.

Sys.CardDPRAutoDetect

Description: Mask of automatically detected dual-ported RAM cards

Range: $0 .. $FFFF

Units: Bit field

Sys.CardDPRAutoDetect contains the result of the Power PMAC's automatic search at power-on/reset for any accessory cards with dual-ported RAM processor interfaces present. Bit ***i*** of **Sys.CardDPRAutoDetect** is set to 1 if the card represented by the **Acc*xx*[*i*]** data structure is present. It is set to 0 if that card is not present.

Sys.CardIOAutoDetect

Description: Mask of automatically detected I/O cards

Range: $0 .. $FFFF

Units: Bit field

Sys.CardIOAutoDetect contains the result of the Power PMAC's automatic search at power-on/reset for any general-purpose I/O cards present. Bit ***i*** of **Sys.CardIOAutoDetect** is set to 1 if the card represented by the **Acc*xx*[*i*]** data structure is present. It is set to 0 if that card is not present.

Sys.ClockSF

Description: Processor clock frequency scale factor

Range: Positive floating-point

Units: µsec per clock cycle

Sys.ClockSF contains the number of microseconds per clock cycle of the Power PMAC processor. For example, it equals 0.00125 for a processor running at 800 MHz. It is used internally to calculate times for various tasks performed, as the fundamental measurements are in clock cycles.

Sys.ClockSource

Description: System clock source IC index

Range: -63 .. 63

Units: Enumeration

Sys.ClockSource contains the index of the Servo or MACRO IC, if any, that is providing the system phase and servo clocks as detected at power-on/reset by the processor. The values this element can take, and the IC the element represents, are:

- 0: No clock source found
- 4 .. 19: **Gate1[*i*]** ***i*** = **Sys.ClockSource**
- 32 .. 47: **Gate2[*i*]** ***i*** = **Sys.ClockSource** - 32
- 48 .. 63: **Gate3[*i*]** ***i*** = **Sys.ClockSource** - 48
- -1: Multiple clock sources found
- -4 .. -19: **Gate1[*i*]*** ***i*** = **-Sys.ClockSource**
- -32 .. -47: **Gate2[*i*]*** ***i*** = **-Sys.ClockSource** - 32
- -48 .. -63: **Gate3[*i*]*** ***i*** = **-Sys.ClockSource** - 48

* ID chip setting and **Gate*n*** clock source word do not agree.

Sys.ConfigLoadErr

Description: "Saved configuration load error" status bit

Range: 0 .. 1

Units: Boolean

The **Sys.ConfigLoadErr** status bit is set to 1 if the processor detects an error in the saved setup element configuration during the power-up/reset process. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time. It is bit 4 of 32-bit element **Sys.Status**.

Sys.Coords

Description: Range of active coordinate systems

Range: 0 .. 128

Units: Coordinate systems

Sys.Coords contains a number one greater than the highest-numbered coordinate system that has any motors assigned to it through axis definition statements or kinematics algorithms. Each real-time interrupt (RTI), Power PMAC checks each coordinate system from 0 to **Sys.Coords** - 1 to see what RTI tasks, if any, need to be done for that coordinate system. Periodically in background, Power PMAC checks all coordinate systems up to **Sys.MaxCoords** - 1 to find the highest-numbered coordinate system with motors assigned to it, and automatically sets **Sys.Coords** to a value one larger than this.

Sys.CpuFreq

Description: Processor operational clock frequency

Range: Positive integer

Units: Hertz

Sys.CpuFreq contains the frequency of the fundamental clock signal inside the Power PMAC processor, expressed in Hertz. It will report 800,000,000 for an 800 MHz processor, 1,000,000,000 for a 1 GHz processor, and 1,200,000,000 for a 1.2 GHz processor.

Sys.CpuTemp

Description: Processor operational temperature

Range: Floating-point

Units: Degrees Celsius

Sys.CpuTemp contains the present operational temperature of the Power PMAC processor, expressed in degrees Celsius with resolution of 0.1°, as measured by sensors inside the processor. The temperature should remain below 65°C for reliable operation.

If it reports a value of 0, there is no temperature sensor in the processor.

Sys.CPUType

Description: Processor model

Range: Positive integer

Units: Enumeration

Sys.CPUType contains the enumeration of the processor model used in the Power PMAC system. It corresponds to the Power PMAC response to the on-line **cpu** command query. The models presently supported are:

- **Sys.CPUType** = 1: CPU_460EX (Single-core Power PC)
- **Sys.CPUType** = 2: CPU_440EP (Prototype models only)
- **Sys.CPUType** = 3: CPU_x86 (Soft Power PMAC)
- **Sys.CPUType** = 4: CPU_APM86xxx (Dual-core Power PC, 465EX)
- **Sys.CPUType** = 5: CPU_ARM (Dual-core ARM1021)

Sys.Default

Description: “Factory default configuration set” status bit

Range: 0 .. 1

Units: Boolean

The **Sys.Default** status bit is set to 1 if the processor puts the system into the factory default configuration during the power-up/reset process, either due to a re-initialization command, a configuration change since the last **save** command, or an error during the process. It is 0 otherwise. It is bit 7 of 32-bit element **Sys.Status**.

Sys.EcatLicense

Description: Maximum number of motors licensed for EtherCAT operation

Range: 0 .. 256

Units: Motors

Sys.EcatLicense contains the maximum number of motors licensed for operation over the EtherCAT network in this Power PMAC. An individual Power PMAC must be purchased with a license for a specific maximum number of motors to be operated over EtherCAT in order to be able to command motors over an EtherCAT network. The value reported for this element reflects this number of motors.

Sys.EcatMasterReady

Description: EtherCAT master stack initialization complete

Range: 0 .. 1

Units: Boolean

Sys.EcatMasterReady contains the initialization status of the Acontis EtherCAT master stack software (**Sys.EcatType** = 1). If it is 0, the stack software is still initializing, and it is not ready to command devices over the EtherCAT network.

If it is 1, the initialization is complete, and the stack software ir ready to command devices over the network. **ECAT[*i*].Enable** should not be set to 1 to activate the network until this status bit reports a value of 1.

Sys.EcatMasterReady is not used if **Sys.EcatType** is 0, specifying use of the Etherlab master stack software.

Sys.FgForceInOr

Description: Foreground buffered input forcing words logical OR

Range: 0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Sys.FgForceInOr is a 32-bit register in which each bit contains the logical OR of the matching bits of **BufIo[*i*].ForceInOn** and **BufIo[*i*].ForceInOff** for all buffered input registers that are currently being scanned at the beginning of each foreground cycle.

Sys.FgForceInOr is intended to provide the user with a quick check on whether any of the foreground buffered inputs are still being forced on or off after a debugging section, as this forcing could interfere with proper operation of the machine. It has no internal use in the Power PMAC.

Each bit of **Sys.FgForceInOr** is logically ORed with the matching bits of **Sys.FgForceOutOr**, **Sys.BgForceInOr**, and **Sys.BgForceOutOr** to compute an overall status word **Sys.ForceOr** that is usually used as a first check as to whether any forcing bits are set.

For example, if **Sys.FgForceInOr** returns a value of $0000C000, this means that bit 15 is set in at least one of the forcing words for foreground buffered inputs, as is bit 4. It does not indicate which of the forcing words has either of these bits set.

Sys.FgForceInOr is new in V2.2 firmware, released 3rd quarter 2016.

Sys.FgForceOutOr

Description: Foreground buffered output forcing words logical OR

Range: 0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Sys.FgForceOutOr is a 32-bit register in which each bit contains the logical OR of the matching bits of **BufIo[*i*].ForceOutOn** and **BufIo[*i*].ForceOutOff** for all buffered output registers that are currently being scanned at the end of each foreground cycle.

Sys.FgForceOutOr is intended to provide the user with a quick check on whether any of the foreground buffered outputs are still being forced on or off after a debugging section, as this forcing could interfere with proper operation of the machine. It has no internal use in the Power PMAC.

Each bit of **Sys.FgForceOutOr** is logically ORed with the matching bits of **Sys.FgForceInOr**, **Sys.BgForceInOr**, and **Sys.BgForceOutOr** to compute an overall status word **Sys.ForceOr** that is usually used as a first check as to whether any forcing bits are set.

For example, if **Sys.FgForceOutOr** returns a value of $01000080, this means that bit 24 is set in at least one of the forcing words for foreground buffered outputs, as is bit 7. It does not indicate which of the forcing words has either of these bits set.

Sys.FgForceOutOr is new in V2.2 firmware, released 3rd quarter 2016.

Sys.FileConfigErr

Description: "File configuration error" status bit

Range: 0 .. 1

Units: Boolean

The **Sys.FileConfigErr** status bit is set to 1 if the processor detects an error in the configuration of the system files during the power-up/reset process. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time. It is bit 6 of 32-bit element **Sys.Status**.

Sys.FlashSizeErr

Description: "Insufficient flash size" status bit

Range: 0 .. 1

Units: Boolean

The **Sys.FlashSizeErr** status bit is set to 1 if the built-in NAND flash memory on Power PMAC does not have enough capacity to store the active user project in the most recent **save** command. The **save** command will be rejected with an error in this case.

Sys.FlashSizeErr is set to 0 on power-on/reset and when a successful **save** command is performed.

Sys.FlashSizeErr is bit 11 of 32-bit element **Sys.Status**.

Sys.FltrBgTime

Description: Averaged calculation time for background task scans

Range: Positive floating-point

Units: Microseconds

Sys.FltrBgTime contains the average time in microseconds the Power PMAC processor spent in calculations for tasks in recent background task scans. The time is from the start to the end of a scan; if interrupted by higher-priority tasks (phase, servo, real-time interrupt), it will include the time in those tasks as well.

The averaging is performed using an exponential low-pass filter that each cycle computes the result as 255/256 of the previous cycle's average plus 1/256 of the most recent cycle's value. The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles * 256. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle) and divide by 256.

The measured time for this cycle starts at the beginning of actual execution of the background tasks of a cycle (not at an interrupt) and ends at the end of the tasks of a cycle, including all of the background C PLC programs.

For a single-core CPU, where this core performs both background and interrupt tasks, this time value includes periods between the beginning and the end that is spent in interrupt tasks. These periods must be subtracted out to get the actual task time for a background scan. (For a multi-core CPU, interrupt tasks are performed simultaneously on a separate core, and so their time does not need to be subtracted out.)

In general, some interrupts will have phase tasks only, some will have phase and servo tasks, and some will have phase, servo, and RTI tasks. The average interrupt task time can be calculated as:

$$tt_{int} = tt_p + \frac{PhaseDeltaTime}{ServoDeltaTime} tt_s + \frac{PhaseDeltaTime}{RtIntDeltaTime} tt_r$$

In this equation, tt_p, tt_s, and tt_r are the average task times for the phase, servo, and real-time interrupts, respectively.

To compute the number of interrupt periods during an average background cycle, the following equation can be used:

$$\left(n_p\right)_b = \frac{FltrBgTime}{PhaseDeltaTime}$$

Then the average background task time can be calculated as:

$$tt_b = FltrBgTime - \left(n_p\right)_b * tt_{int}$$

Sys.FltrPhaseTime

Description: Averaged calculation time for phase tasks

Range: Positive floating-point

Units: Microseconds

Sys.FltrPhaseTime contains the average time in microseconds the Power PMAC processor spent in calculations for tasks in recent phase interrupts. The value for the most recent cycle is found in **Sys.PhaseTime**.

The averaging is performed using an exponential low-pass filter that each cycle computes the result as 255/256 of the previous cycle's average plus 1/256 of the most recent cycle's value. The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles * 256. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle) and divide by 256.

Sys.FltrRtIntTime

Description: Averaged calculation time for real-time interrupt tasks

Range: Positive floating-point

Units: Microseconds

Sys.FltrRtIntTime contains the average time in microseconds from the real-time interrupt until the Power PMAC processor has completed real-time interrupt task calculations for recent real-time interrupts. Since every real-time interrupt is also a servo interrupt and a phase interrupt and these tasks execute first, this time will include the time spent in phase and servo tasks for at least one cycle of each. If the real-time interrupt tasks were interrupted by any further phase or servo interrupts, this time includes the time spent in those higher-priority tasks for those cycles as well.

The averaging is performed using an exponential low-pass filter that each cycle computes the result as 255/256 of the previous cycle's average plus 1/256 of the most recent cycle's value. The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles * 256. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle) and divide by 256.

To compute the average time actually spent in RTI tasks in recent cycles (tt_r), we first calculate the number of phase cycles in the reported RTI time $(n_p)_r$ and the number of servo cycles in the reported RTI time $(n_s)_r$ as:

$$\left(n_p\right)_r = \text{int}\left(\frac{FltrRtIntTime}{PhaseDeltaTime}\right) + \min\left(\frac{FltrRtIntTime \;\%\; PhaseDeltaTime}{FltrPhaseTime}, 1.0\right)$$

$$\left(n_s\right)_r = \text{int}\left(\frac{FltrRtIntTime}{ServoDeltaTime}\right) + \min\left(\frac{\left(FltrRtIntTime - \left(n_p\right)_s PhaseTime\right)\%\; ServoDeltaTime}{FltrServoTime}, 1.0\right)$$

Then the time actually spent in real-time interrupt tasks for this most recent real-time interrupt cycle can be calculated by the equation:

$$tt_r = FltrRtIntTime - (n_p)_r * FltrPhaseTime - (n_s)_r * FltrServoTime$$

Note that the reported times for RTI tasks do *not* include the time spent in executing the RTI CPLC (if any).

Sys.FltrServoTime

Description: Averaged calculation time for servo tasks

Range: Positive floating-point

Units: Microseconds

Sys.FltrServoTime contains the average time in microseconds from the servo interrupt until the Power PMAC processor has completed servo task calculations in the most recent servo interrupt. Since every servo interrupt is also a phase interrupt and phase tasks execute first, this time will include the time spent in phase tasks for at least one cycle. If the servo tasks were interrupted by any further phase interrupts, this time includes the time spent in phase tasks for those cycles as well. The time for the most recent cycle is found in **Sys.ServoTime**.

The averaging is performed using an exponential low-pass filter that each cycle computes the result as 255/256 of the previous cycle's average plus 1/256 of the most recent cycle's value. The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles * 256. To convert this value to microseconds, multiply by **Sys.ClockSF**(µsec / clock cycle) and divide by 256.

To compute the average time actually spent in servo tasks in recent servo cycles (tt_s), we first calculate the number of phase cycles in the reported servo time $(n_p)_s$ as:

$$(n_p)_s = \text{int}\left(\frac{FltrServoTime}{PhaseDeltaTime}\right) + \min\left(\frac{FltrServoTime \,\%\, PhaseDeltaTime}{FltrPhaseTime}, 1.0\right)$$

Then the average time actually spent in servo tasks in recent servo cycles can be calculated by the equation:

$$tt_s = FltrServoTime - (n_p)_s * FltrPhaseTime$$

Sys.ForceOr

Description: Buffered input/output forcing words logical OR

Range: 0 .. $FFFFFFFF (0 .. $2^{32} - 1$)

Units: Bit field

Sys.ForceOr is a 32-bit register in which each bit contains the logical OR of the matching bits of **BufIo[*i*].ForceInOn**, **BufIo[*i*].ForceInOff**, **BufIo[*i*].ForceOutOn** and **BufIo[*i*].ForceOutOff** for all buffered input and output registers that are currently being scanned in each foreground or background cycle.

Sys.ForceOr is intended to provide the user with a quick check on whether any of the buffered inputs or outputs are still being forced on or off after a debugging section, as this forcing could interfere with proper operation of the machine. It has no internal use in the Power PMAC.

More detailed information as to which bits are being forced is present in **Sys.BgForceInOr**, for background inputs, **Sys.BgForceOutOr**, for background outputs, **Sys.FgForceInOr**, for foreground inputs, and **Sys.FgForceOutOr**, for foreground outputs.

For example, if **Sys.ForceOr** returns a value of $20000100, this means that bit 29 is set in at least one of the forcing words for buffered inputs or outputs, as is bit 8. It does not indicate which of the forcing words has either of these bits set.

Sys. ForceOr is new in V2.2 firmware, released 3rd quarter 2016.

Sys.GantryXCtrl

Description: Address of cross-coupled gantry servo control algorithm

Range: Positive integer

Units: Power PMAC memory addresses

Sys.GantryXCtrl contains the address of the start of the adaptive servo control algorithm. If **Motor[*x*].Ctrl** is set to **Sys.GantryXCtrl**, the motor will execute this algorithm as its servo algorithm each cycle. It is rarely necessary to know the numerical value of this address.

Sys.Gate1AddrErrDetect

Description: Mask of DSPGATE1 ICs with addressing errors

Range: $0 .. $FFFF0

Units: Bit field

Sys.Gate1AddrErrDetect contains the result of the Power PMAC's automatic test at power-on/reset for any PMAC2-style DSPGATE1 Servo ICs present. Bit ***i*** of **Sys.Gate1AddrErrDetect** is set to 1 if a test value written to the IC represented by the **Gate1[*i*]** data structure can also be read back at the address of any other DSPGATE1 IC. It is set to 0 if this error is not detected.

Sys.Gate1AutoDetect

Description: Mask of automatically detected DSPGATE1 ICs

Range: $0 .. $FFFF0

Units: Bit field

Sys.Gate1AutoDetect contains the result of the Power PMAC's automatic search at power-on/reset for any PMAC2-style DSPGATE1 Servo ICs present. Bit ***i*** of **Sys.Gate1AutoDetect** is set to 1 if the IC represented by the **Gate1[*i*]** data structure is present. It is set to 0 if that IC is not present.

Sys.Gate2AddrErrDetect

Description: Mask of DSPGATE2 ICs with addressing errors

Range: $0 .. $FFFF

Units: Bit field

Sys.Gate2AddrErrDetect contains the result of the Power PMAC's automatic test at power-on/reset for any PMAC2-style DSPGATE2 MACRO ICs present. Bit ***i*** of **Sys.Gate2AddrErrDetect** is set to 1 if a test value written to the IC represented by the **Gate2[*i*]** data structure can also be read back at the address of any other DSPGATE2 IC. It is set to 0 if this error is not detected.

Sys.Gate2AutoDetect

Description: Mask of automatically detected DSPGATE2 ICs

Range: $0 .. $FFFF

Units: Bit field

Sys.Gate2AutoDetect contains the result of the Power PMAC's automatic search at power-on/reset for any PMAC2-style DSPGATE2 MACRO ICs present. Bit ***i*** of **Sys.Gate2AutoDetect** is set to 1 if the IC represented by the **Gate2[*i*]** data structure is present. It is set to 0 if that IC is not present.

Sys.Gate3AddrErrDetect

Description: Mask of DSPGATE3 ICs with addressing errors

Range: $0 .. $FFFF

Units: Bit field

Sys.Gate3AddrErrDetect contains the result of the Power PMAC's automatic test at power-on/reset for any PMAC3-style DSPGATE3 ICs present. Bit ***i*** of **Sys.Gate1AddrErrDetect** is set to 1 if a test value written to the IC represented by the **Gate3[*i*]** data structure can also be read back at the address of any other DSPGATE3 IC. It is set to 0 if this error is not detected.

Sys.Gate3AutoDetect

Description: Mask of automatically detected DSPGATE3 ICs

Range: $0 .. $FFFF

Units: Bit field

Sys.Gate3AutoDetect contains the result of the Power PMAC's automatic search at power-on/reset for any PMAC3-style DSPGATE3 machine-interface ICs present. Bit ***i*** of **Sys.Gate3AutoDetect** is set to 1 if the IC represented by the **Gate3[*i*]** data structure is present. It is set to 0 if that IC is not present.

Sys.HWChangeErr

Description: "Hardware change detected" status bit

Range: 0 .. 1

Units: Boolean

The **Sys.HWChangeErr** status bit is set to 1 if the processor detects a change in the system hardware configuration from the last save operation during the power-up/reset process. In this case, the system is put in the factory default configuration. A change could occur due to added or removed hardware, hardware with a changed address setting, or malfunctioning hardware that could not be detected.

It is 0 if no such error is found at this time. It is bit 5 of 32-bit element **Sys.Status**.

Sys.IntBusy

Description: Interrupt levels presently active or suspended

Range: 0 .. 15

Units: Bit field

Sys.IntBusy displays which interrupt tasks are presently executing or have been suspended due to a higher-priority interrupt. It is intended for use from a "background" core in a multi-core Power PMAC CPU in algorithms that want to access information that could be changed by an interrupt routine in the "foreground" core. It is of little practical use in a single-core Power PMAC.

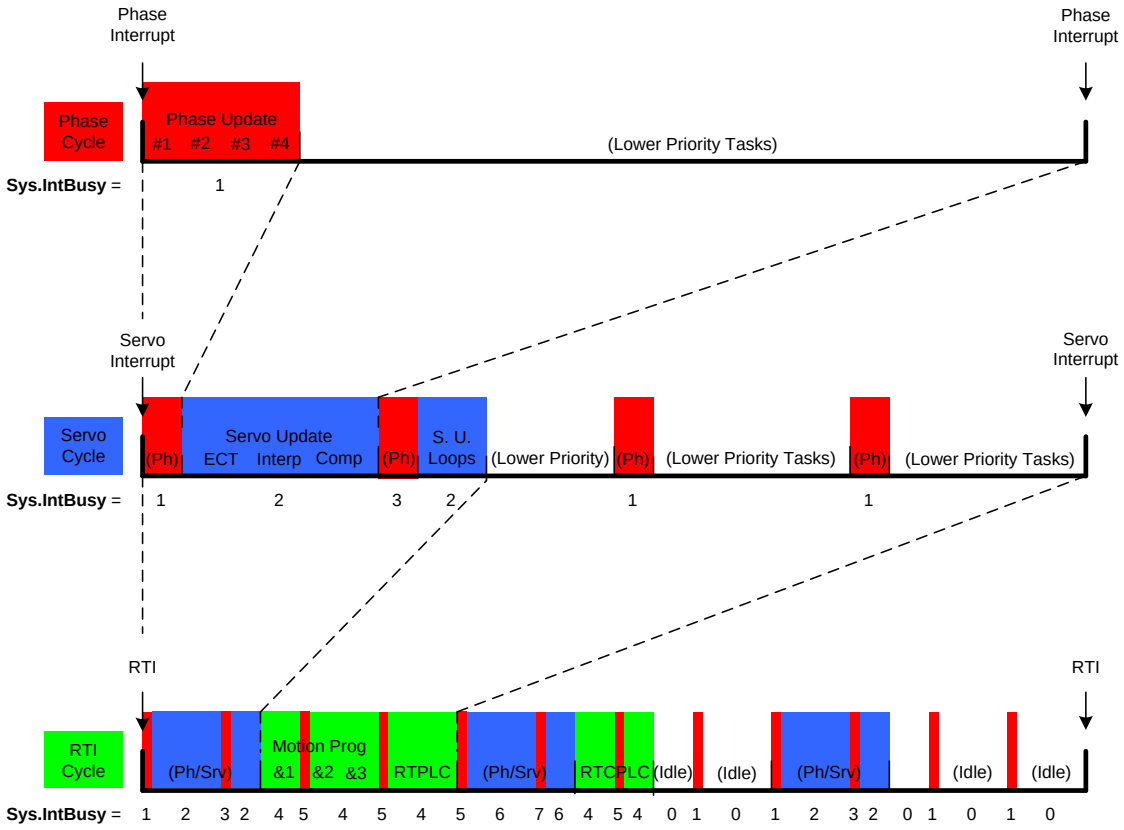
Sys.IntBusy is a 4-bit field, with each bit representing one of the interrupt priority levels. The bit is set to 1 if the routine at that level is active or suspended; it is 0 otherwise. The interrupt level each bit represents is shown here:

- Bit 0 (value 1): Phase interrupt
- Bit 1 (value 2): Servo interrupt
- Bit 2 (value 4): Real-time interrupt
- Bit 3 (value 8): Capture-compare interrupt (optional)

The first three of these interrupts are executed in all Power PMAC systems, and these bits are automatically set and cleared by Power PMAC firmware. Of these interrupts, phase has a higher priority than servo, which has a higher priority than real-time.

For example, if **Sys.IntBusy** has a value of 7, phase tasks are currently executing (+1), having interrupted and suspended incomplete servo tasks (+2), which had interrupted and suspended incomplete real-time interrupt tasks (+4).

The following diagram shows the values of **Sys.IntBusy** for a sample timeline of the use of a foreground core:



The capture-compare interrupt is available only on Power PMAC systems with DSPGATE3 Servo ICs, and require a user-written interrupt service routine. If the user wishes the status of this

routine to be reflected in the value of **Sys.IntBusy**, the routine must explicitly set the bit at the beginning (e.g. `pshm->IntBusy |= 8;`) and clear it at the end (e.g. `pshm->IntBusy &= 7;`).

Sys.IntBusy cannot be written to from the Script environment. It is new in V2.1 firmware, released 1st quarter 2016.

Sys.LegacyCtrl

Description: Address of Turbo-PMAC format servo control algorithm

Range: Positive integer

Units: Power PMAC memory addresses

Sys.LegacyCtrl contains the address of the start of the extended PID servo control algorithm with polynomial filters whose structure is like the Turbo PMAC servo algorithm. If **Motor[*x*].Ctrl** is set to **Sys.LegacyCtrl**, the motor will execute this algorithm as its servo algorithm each cycle. It is rarely necessary to know the numerical value of this address.

Sys.Lock

Description: Process locking control bit field

Range: $0 .. $FFFFFFFF

Units: Bit field

Sys.Lock is a 32-bit read-only element that shows the present status of the 32 single-bit user execution process-locking non-saved setup elements **Sys.Lock[*i*]**. The status of each **Sys.Lock[*i*]** (***i*** = 0 to 31) element is shown in bit ***i*** of **Sys.Lock**, with value 2^i.

Sys.Lock is mainly used for monitoring and debugging the use of the individual **Sys.Lock[*i*]** bits. No bit values are changed by a read access to the 32-bit element **Sys.Lock**, unlike Script read accesses to the individual **Sys.Lock[*i*]** bits.

Sys.MaxBgTime

Description: Maximum calculation time for background task scan

Range: Positive floating-point

Units: Microseconds

Sys.MaxBgTime contains the longest time in microseconds the Power PMAC processor has spent in calculations for tasks in any background task scan (**Sys.BgTime** for the scan) since this

element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.MaxPhaseDeltaTime

Description: Maximum time elapsed between start of two phase cycles

Range: Positive floating-point

Units: Microseconds

Sys.MaxPhaseDeltaTime contains the longest time elapsed in microseconds between the start of software tasks executing under two consecutive phase interrupts (**Sys.PhaseDeltaTime** for the latest interrupt). Note that while the phase interrupt signal is generated by very precise hardware circuits, there is variation in the latency in responding to the interrupt to the start of the phase-interrupt software, so there can be cycle-to-cycle variation in this value.

Sys.MaxPhaseDeltaTime is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation. The worst case variation in the latency in response to the hardware interrupt signal can be computed as:

$$MaxVariation = \frac{Sys.MaxPhaseDeltaTime - Sys.MinPhaseDeltaTime}{2}$$

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.MaxPhaseTime

Description: Maximum calculation time for phase tasks

Range: Positive floating-point

Units: Microseconds

Sys.MaxPhaseTime contains the longest time in microseconds the Power PMAC processor has spent in calculations for tasks in any phase interrupt (**Sys.PhaseTime** for the interrupt) since this element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.MaxRtIntTime

Description: Maximum calculation time for real-time-interrupt tasks

Range: Positive floating-point

Units: Microseconds

Sys.MaxRtIntTime contains the longest time in from the real-time interrupt until the Power PMAC processor has completed real-time interrupt task calculations in any real-time interrupt (**Sys.RtIntTime** for the interrupt) since this element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSf** (µsec / clock cycle).

Note that the reported times for RTI tasks do *not* include the time spent in executing the RTI CPLC (if any).

Sys.MaxServoTime

Description: Maximum calculation time for servo tasks

Range: Positive floating-point

Units: Microseconds

Sys.MaxServoTime contains the longest time in from the servo interrupt until the Power PMAC processor has completed servo task calculations in any servo interrupt (**Sys.ServoTime** for the interrupt) since this element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.MinBgTime

Description: Minimum calculation time for background task scan

Range: Positive floating-point

Units: Microseconds

Sys.MinBgTime contains the shortest time in microseconds the Power PMAC processor has spent in calculations for tasks in any background task scan (**Sys.BgTime** for the scan) since this

element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.MinPhaseDeltaTime

Description: Minimum time elapsed between start of two phase cycles

Range: Positive floating-point

Units: Microseconds

Sys.MinPhaseDeltaTime contains the shortest time elapsed in microseconds between the start of software tasks executing under two consecutive phase interrupts (**Sys.PhaseDeltaTime** for the latest interrupt). Note that while the phase interrupt signal is generated by very precise hardware circuits, there is variation in the latency in responding to the interrupt to the start of the phase-interrupt software, so there can be cycle-to-cycle variation in this value.

Sys.MinPhaseDeltaTime is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation. The worst case variation in the latency in response to the hardware interrupt signal can be computed as:

$$MaxVariation = \frac{Sys.MaxPhaseDeltaTime - Sys.MinPhaseDeltaTime}{2}$$

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.MinPhaseTime

Description: Minimum calculation time for phase tasks

Range: Positive floating-point

Units: Microseconds

Sys.MinPhaseTime contains the shortest time in microseconds the Power PMAC processor has spent in calculations for tasks in any phase interrupt (**Sys.PhaseTime** for the interrupt) since this element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.MinRtIntTime

Description: Minimum calculation time for real-time-interrupt tasks

Range: Positive floating-point

Units: Microseconds

Sys.MinRtIntTime contains the shortest time in microseconds from the real-time interrupt until the Power PMAC processor has completed real-time interrupt task calculations in any real-time interrupt (**Sys.RtIntTime** for the interrupt) since this element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Note that the reported times for RTI tasks do *not* include the time spent in executing the RTI CPLC (if any).

Sys.MinServoTime

Description: Minimum calculation time for servo tasks

Range: Positive floating-point

Units: Microseconds

Sys.MinServoTime contains the shortest time in microseconds from the servo interrupt until the Power PMAC processor has completed servo task calculations in any servo interrupt (**Sys.ServoTime** for the interrupt) since this element has been set to 0.0. It is automatically set to 0.0 at power-on/reset; a user can set it to 0.0 at any time to restart the period of evaluation.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.NoClocks

Description: “No system clocks found” status bit

Range: 0 .. 1

Units: Boolean

The **Sys.NoClocks** status bit is set to 1 if the processor does not find any phase or servo clock signals at the end of the power-up/reset initialization process and saved setup element **Sys.CpuTimerIntr** is set to its default value of 0, so the processor is not generating is own interrupts. It is 0 if it does find these clock signals at this time, or if the processor is generating its own interrupts. If this bit is set, no motors can be enabled. **Sys.NoClocks** is bit 8 of 32-bit element **Sys.Status**.

Sys.OffsetCardDPR[*i*]

Description: Base address offset of dual-ported RAM card

Range: $0, $E00000 .. $F18000

Units: Power PMAC I/O address offsets

Sys.OffsetCardDPR[*i*] contains the base address offset of the accessory card with a dual-ported RAM processor interface of index ***i***. This offset is from the start of memory mapped I/O whose value can be found in **Sys.piom**. If no card of this type and index is present, this element will return a value of $0. Status element **Sys.CardDPRAutoDetect** reports which cards are present. Users will seldom need to know these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.OffsetCardDPRCid[*i*]

Description: Base address offset of card identification for dual-ported RAM card

Range: $0, $D00000 .. $D18D00

Units: Power PMAC I/O address offsets

Sys.OffsetCardDPRCid[*i*] contains the base address offset of the card identification for the accessory card with a dual-ported RAM processor interface of index ***i***, whether or not this card is present. This offset is from the start of memory mapped I/O whose value can be found in **Sys.piom**. Status element **Sys.CardDPRAutoDetect** reports which cards are present. Users will seldom need to know these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.OffsetCardIO[*i*]

Description: Base address offset of general-purpose I/O card

Range: $0, $A00000 .. $D18100

Units: Power PMAC I/O address offsets

Sys.OffsetCardIO[*i*] contains the base address offset of the general-purpose I/O accessory card of index ***i***. This offset is from the start of memory mapped I/O whose value can be found in

Sys.piom. If no card of this type and index is present, this element will return a value of $0. Status element **Sys.CardIOAutoDetect** reports which cards are present. Users will seldom need to know these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.OffsetCardIOCid[*i*]

Description: Base address offset of card identification for general-purpose I/O card

Range: $0, $D00000 .. $D18D00

Units: Power PMAC I/O address offsets

Sys.OffsetCardIOCid[*i*] contains the base address offset of the card identification for the general-purpose I/O accessory card of index ***i***, whether or not this card is present. This offset is from the start of memory mapped I/O whose value can be found in **Sys.piom**. Status element **Sys.CardIOAutoDetect** reports which cards are present. Users will seldom need to know these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.OffsetGate1[*i*]

Description: Base address offset of DSPGATE1 Servo IC

Range: $0, $600000 .. $718100

Units: Power PMAC I/O address offsets

Sys.OffsetGate1[*i*] contains the base address offset of the PMAC2-style DSPGATE1 Servo IC of index ***i***. This offset is from the start of memory mapped I/O whose value can be found in **Sys.piom**. If no IC of this type and index is present, this element will return a value of $0. Status element **Sys.Gate1AutoDetect** reports which ICs are present. Users will seldom need to know these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.OffsetGate1Cid[*i*]

Description: Base address offset of card identification for DSPGATE1 IC card

Range: $0, $D00000 .. $D18D00

Units: Power PMAC I/O address offsets

Sys.OffsetGate1Cid[*i*] contains the base address offset of the card identification for the accessory card with a DSPGATE1 Servo IC of index ***i***, whether or not this card is present. This offset is from the start of memory mapped I/O whose value can be found in **Sys.piom**. Status element **Sys.Gate1AutoDetect** reports which cards are present. Users will seldom need to know

these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.OffsetGate2[*i*]

Description: Base address offset of DSPGATE2 MACRO IC

Range: $0, $800000 .. $819800

Units: Power PMAC I/O address offsets

Sys.OffsetGate2[*i*] contains the base address offset of the PMAC2-style DSPGATE2 MACRO IC of index ***i***. This offset is from the start of memory mapped I/O whose value can be found in **Sys.piom**. If no IC of this type and index is present, this element will return a value of $0. Status element **Sys.Gate2AutoDetect** reports which ICs are present. Users will seldom need to know these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.OffsetGate2Cid[*i*]

Description: Base address offset of card identification for DSPGATE2 IC card

Range: $0, $D00000 .. $D18D00

Units: Power PMAC I/O address offsets

Sys.OffsetGate2Cid[*i*] contains the base address offset of the card identification for the accessory card with a DSPGATE2 MACRO IC of index ***i***, whether or not this card is present. This offset is from the start of memory mapped I/O whose value can be found in **Sys.piom**. Status element **Sys.Gate1AutoDetect** reports which cards are present. Users will seldom need to know these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.OffsetGate3[*i*]

Description: Base address offset of DSPGATE3 machine-interface IC

Range: $0, $800000 .. $819800

Units: Power PMAC I/O address offsets

Sys.OffsetGate3[*i*] contains the base address offset of the PMAC3-style DSPGATE3 machine-interface IC of index ***i***. This offset is from the start of memory mapped I/O whose value can be found in **Sys.piom**. If no IC of this type and index is present, this element will return a value of $0. Status element **Sys.Gate3AutoDetect** reports which ICs are present. Users will seldom need to know these numerical address values, but they can be valuable for debugging purposes and for setting up C pointer variables.

Sys.PhaseCount

Description: Number of phase cycles since power-on/reset

Range: 0 .. 2^{32}-1

Units: Phase cycles

Sys.PhaseCount contains the number of phase cycles that have occurred since the most recent power-on or reset of the Power PMAC.

Sys.PhaseDeltaTime

Description: Time elapsed between start of last two phase cycles

Range: Positive floating-point

Units: Microseconds

Sys.PhaseDeltaTime contains the time elapsed in microseconds between the start of software tasks executing under the most recent two phase interrupts. Note that while the phase interrupt signal is generated by very precise hardware circuits, there is variation in the latency in responding to the interrupt to the start of the phase-interrupt software, so there can be cycle-to-cycle variation in this value.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.PhaseErrorCtr

Description: Counter for phase-interrupt overrun errors

Range: Non-negative integers

Units: Count

Sys.PhaseErrorCtr contains the number of times since power-on/reset the phase-interrupt tasks have not completed before the next phase interrupt occurs. A condition where this error occurs should be considered a serious problem, as it can seriously impact the quality of control, and possibly lead to a watchdog timer trip due to processor overload.

Sys.PhaseMotors

Description: Range of motors with active phase algorithms

Range: 0 .. 256

Units: Motors

Sys.PhaseMotors contains a number one greater than the highest-numbered motor for which **Motor[*x*].PhaseCtrl** is greater than 0, specifying an active phase-interrupt algorithm. Each phase interrupt, Power PMAC checks each motor from 0 to **Sys.PhaseMotors** - 1 to see what phase tasks, if any, need to be done for that motor. Periodically in background, Power PMAC checks all motors up to **Sys.MaxMotors** - 1 to find the highest-numbered motor for which **Motor[*x*].PhaseCtrl** is greater than 0, and automatically sets **Sys.PhaseMotors** to a value one larger than this.

Sys.PhaseTime

Description: Latest calculation time for phase tasks

Range: Positive floating-point

Units: Microseconds

Sys.PhaseTime contains the time in microseconds the Power PMAC processor spent in calculations for tasks in the most recent phase interrupt.

If directly queried from an on-line command, the processing of communications may cause memory caching issues that increase the reported time. It is better to query the averaged time in status element **Sys.FltrPhaseTime**.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.PidCtrl

Description: Address of basic PID servo control algorithm

Range: Positive integer

Units: Power PMAC memory addresses

Sys.PidCtrl contains the address of the start of the basic PID servo control algorithm. If **Motor[*x*].Ctrl** is set to **Sys.PidCtrl**, the motor will execute this algorithm as its servo algorithm each cycle. It is rarely necessary to know the numerical value of this address.

Sys.piom

Description: Address of start of memory-mapped I/O space

Range: Positive integer

Units: Power PMAC memory addresses

Sys.piom contains the byte address of the start of the memory-mapped I/O space in Power PMAC. All hardware register and element addresses are referenced to this base address as offsets. This can vary between versions of the Power PMAC firmware. It is rare that a user will need to know the numerical value of this element, but this element is often used to specify the address of an I/O register that does not have a data structure element name (e.g. **Motor[1].pBrakeOut** = **Sys.piom** + $A0000C).

Sys.PosCtrl

Description: Address of position-output servo control algorithm

Range: Positive integer

Units: Power PMAC memory addresses

Sys.PosCtrl contains the address of the start of the position-output servo control algorithm. If **Motor[*x*].Ctrl** is set to **Sys.PosCtrl**, the motor will execute this algorithm as its servo algorithm each cycle. When this algorithm is selected, Power PMAC is not actually closing any servo loops for the motor; it is simply outputting the net commanded position for external use. It is rarely necessary to know the numerical value of this address.

Sys.ProjectLoadErr

Description: "Project load error" status bit

Range: 0 .. 1

Units: Boolean

The **Sys.ProjectLoadErr** status bit is set to 1 if the processor detects an error in the loading of the user project files into active memory during the power-up/reset process. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time. It is bit 3 of 32-bit element **Sys.Status**.

Sys.pushm

Description: Address of start of user shared memory buffer

Range: Positive integer

Units: Power PMAC memory addresses

Sys.pushm contains the byte address of the start of the user shared memory buffer in Power PMAC. The **Cdata[*i*]**, **Ddata[*i*]**, **Fdata[*i*]**, **Idata[*i*]**, and **Udata[*i*]** arrays are referenced to this base address. It is rare that a user will need to know the numerical value of this element, but this element can be used to specify the address of a buffer register.

Sys.PwrOnFault

Description: "Power-up/reset load fault" status bit

Range: 0 .. 1

Units: Boolean

The **Sys.PwrOnFault** status bit is set to 1 if the processor detects an error during the power-up/reset process. This bit is the logical OR of **Sys.ProjectLoadErr**, **Sys.ConfigLoadErr**, **Sys.HWChangeError**, and **Sys.FileConfigErr**, so will be set if any of those errors is detected. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time. It is bit 2 of 32-bit element **Sys.Status**.

Sys.RtIntBusyCtr

Description: Counter for real-time-interrupt re-entry events

Range: Non-negative integers

Units: Count

Sys.RtIntBusyCtr contains the number of times since power-on/reset the real-time-interrupt tasks have not completed before the next real-time interrupt occurs. A situation where this condition occurs is not necessarily a serious problem, but may indicate possible overloading of the processor at this priority level. If such an occurrence causes the next real-time interrupt cycle's tasks to be missed completely, the value of **Sys.RtIntErrorCtr** will be incremented as well.

Sys.RtIntDeltaTime

Description: Time elapsed between start of last two real-time interrupt cycles

Range: Positive floating-point

Units: Microseconds

Sys.RtIntDeltaTime contains the time elapsed in microseconds between the start of software tasks executing under the most recent two real-time interrupts (RTIs). Note that while the interrupt signal driving the RTI is generated by very precise hardware circuits, there is variation

in the latency in responding to the interrupt to the start of the RTI software, so there can be cycle-to-cycle variation in this value.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.RtIntErrorCtr

Description: Counter for real-time-interrupt overrun errors

Range: Non-negative integers

Units: Count

Sys.RtIntErrorCtr contains the number of times since power-on/reset the real-time-interrupt tasks have missed a cycle due to overrun of the previous cycle's tasks. Unlike with phase and servo-interrupt errors of this type, this is not necessarily a serious problem if it happens occasionally, but foreground Script and C PLCs will "skip a beat" when this re-entry occurs.

Sys.RtIntTime

Description: Latest calculation time for real-time-interrupt tasks

Range: Positive floating-point

Units: Microseconds

Sys.RtIntTime contains the time in microseconds from the real-time interrupt until the Power PMAC processor has completed real-time interrupt task calculations for the most recent real-time interrupt. Since every real-time interrupt is also a servo interrupt and a phase interrupt and these tasks execute first, this time will include the time spent in phase and servo tasks for at least one cycle of each. If the real-time interrupt tasks were interrupted by any further phase or servo interrupts, this time includes the time spent in those higher-priority tasks for those cycles as well.

The time actually spent in real-time interrupt tasks for this most recent real-time interrupt cycle can be calculated by the equation:

$$RtIntTaskTime = Sys.RtIntTime - \left[\text{int}\left(\frac{Sys.RtIntTime}{Sys.PhaseDeltaTime} \right) + 1 \right] * Sys.PhaseTime$$
$$- \left[\text{int}\left(\frac{Sys.RtIntTime}{Sys.ServoDeltaTime} \right) + 1 \right] * ServoTaskTime$$

where:

$$ServoTaskTime = Sys.ServoTime - \left[\text{int}\left(\frac{Sys.ServoTime}{Sys.PhaseDeltaTime} \right) + 1 \right] * Sys.PhaseTime$$

If directly queried from an on-line command, the processing of communications may cause memory caching issues that increase the reported time. It is better to query the averaged time in status element **Sys.FltrRtIntTime**.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Note that the reported times for RTI tasks do *not* include the time spent in executing the RTI CPLC (if any).

Sys.RunTime

Description: Time from most recent firmware start to present

Range: Positive floating-point

Units: seconds

Sys.RunTime contains the time in seconds from the most recent start of the Power PMAC firmware application until the present. The Power PMAC firmware application starts in tens of seconds from initial power-up. It restarts after a reset from a **$$$** or **$$$***** command. The time from power-up to the most recent start of the Power PMAC firmware application is found in **Sys.StartTime**.

Sys.ServoBusyCtr

Description: Counter for servo-interrupt re-entry events

Range: Non-negative integers

Units: Count

Sys.ServoBusyCtr contains the number of times since power-on/reset the servo-interrupt tasks have not completed before the next servo interrupt occurs. A situation where this condition occurs should be considered a potentially serious problem due to its possible impact on control quality and processor loading. If such an occurrence causes the next servo cycle's tasks to be missed completely, the value of **Sys.ServoErrorCtr** will be incremented as well.

Sys.ServoCount

Description: Number of servo cycles since power-on/reset

Range: 0 .. 2^{32}-1

Units: Servo cycles

Sys.ServoCount contains the number of servo cycles that have occurred since the most recent power-on or reset of the Power PMAC.

Sys.ServoCtrl

Description: Address of standard extended PID servo control algorithm

Range: Positive integer

Units: Power PMAC memory addresses

Sys.ServoCtrl contains the address of the start of the standard extended PID servo control algorithm with polynomial filters. If **Motor[*x*].Ctrl** is set to **Sys.ServoCtrl**, the motor will execute this algorithm as its servo algorithm each cycle. This is the default setting for all motors. It is rarely necessary to know the numerical value of this address.

Sys.ServoDeltaTime

Description: Time elapsed between start of last two servo cycles

Range: Positive floating-point

Units: Microseconds

Sys.ServoDeltaTime contains the time elapsed in microseconds between the start of software tasks executing under the most recent two servo interrupts. Note that while the servo interrupt signal is generated by very precise hardware circuits, there is variation in the latency in responding to the interrupt to the start of the servo-interrupt software, so there can be cycle-to-cycle variation in this value.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.ServoErrorCtr

Description: Counter for servo-interrupt overrun errors

Range: Non-negative integers

Units: Count

Sys.ServoErrorCtr contains the number of times since power-on/reset the servo-interrupt tasks have missed a cycle due to overrun of the previous cycle's tasks. A condition where this error occurs should be considered a serious problem, as it seriously impacts the quality of control, and can possibly lead to a watchdog timer trip due to processor overload.

Sys.ServoMotors

Description: Range of motors with active servo algorithms

Range: 0 .. 256

Units: Motors

Sys.ServoMotors contains a number one greater than the highest-numbered motor for which **Motor[*x*].ServoCtrl** is greater than 0, specifying an active servo-interrupt algorithm. Each servo interrupt, Power PMAC checks each motor from 0 to **Sys.ServoMotors** - 1 to see what servo tasks, if any, need to be done for that motor. Periodically in background, Power PMAC checks all motors up to **Sys.MaxMotors** - 1 to find the highest-numbered motor for which **Motor[*x*].ServoCtrl** is greater than 0, and automatically sets **Sys.ServoMotors** to a value one larger than this.

Sys.ServoTime

Description: Latest calculation time for servo tasks

Range: Positive floating-point

Units: Microseconds

Sys.ServoTime contains the time in microseconds from the servo interrupt until the Power PMAC processor has completed servo task calculations in the most recent servo interrupt. Since every servo interrupt is also a phase interrupt and phase tasks execute first, this time will include the time spent in phase tasks for at least one cycle. If the servo tasks were interrupted by any further phase interrupts, this time includes the time spent in phase tasks for those cycles as well.

The time actually spent in servo tasks for this most recent servo cycle can be calculated by the equation:

$$ServoTaskTime = Sys.ServoTime - \left[\text{int}\left(\frac{Sys.ServoTime}{Sys.PhaseDeltaTime} \right) + 1 \right] * Sys.PhaseTime$$

If directly queried from an on-line command, the processing of communications may cause memory caching issues that increase the reported time. It is better to query the averaged time in status element **Sys.FltrServoTime**.

The raw internal units of this register, which would be obtained by data gathering or in a C program, are CPU clock cycles. To convert this value to microseconds, multiply by **Sys.ClockSF** (µsec / clock cycle).

Sys.SineTable[*i*]

Description: Sine lookup table entry

Range: -1.0 – 1.0

Units: None

Sys.SineTable[*i*] contains the entry with index ***i*** of the sine lookup table in Power PMAC. This table has 2048 entries, with an index range of 0 to 2047. Each entry is a single-precision floating-point value, whose value can be computed as:

$$Sys.SineTable[i] = \sin\left(\frac{360}{2048} i^{\circ}\right)$$

This sine table is primarily used by Power PMAC's built-in commutation routines. Saved setup elements **Motor[*x*].pSineTable** and **Motor[*x*].pVoltSineTable** by default are set to the address of the start of this table: **Sys.SineTable[0].a**.

Sys.StartTime

Description: Time from power-on to most recent firmware start

Range: Positive floating-point

Units: seconds

Sys.StartTime contains the time in seconds from the most recent power-up of the Power PMAC processor (when it started fundamental operations) until the most recent start of the Power PMAC firmware application. On the power-up, this time will typically be in tens of seconds. After a reset from a **$$$** or **$$$***** command, this will show the time in seconds from initial power-up until the Power PMAC application starts running again at the end of the reset cycle.

Sys.Status

Description: Global status word

Range: $0 .. $FFFFFFFF

Units: Bit field

Sys.Status is the 32-bit global status word containing many individual global status bits. The following table provides a list of these status bits in the word:

Bit #	Hex Val	Element Name: Sys.*{element}*	Description
12–31	-		(*Reserved*)
11	$800	**FlashSizeErr**	Insufficient flash memory to store user project
10	$400	**BufSizeErr**	Insufficient memory for declared size of buffers
9	$200	**AbortAll**	Stopping or stopped due to “abort all” input
8	$100	**NoClocks**	No system clocks found
7	$80	**Default**	Factory default configuration (by cmd or error)
6	$40	**FileConfigErr**	System file configuration error
5	$20	**HWChangeErr**	Hardware change detected since save
4	$10	**ConfigLoadErr**	Saved configuration load error
3	$8	**ProjectLoadErr**	Project load error
2	$4	**PwrOnFault**	Power-on/reset load fault (OR of bits 3–6)
1	$2	**WDTFault (bit 1)**	Software watchdog fault (interrupt failure)
0	$1	**WDTFault (bit 0)**	Software watchdog fault (background failure)

The value of this element is reported in response to the **?** query command. Its contents are reported as eight hexadecimal digits (four bits per digit). The on-line query command **backup Sys.Status** causes Power PMAC to report the values of all of the individual elements as English text.

Each element is described in more detail in its own individual specification.

Sys.Time

Description: Time from power-on to present

Range: Positive floating-point

Units: seconds

Sys.Time contains the time in seconds from the most recent power-up of the Power PMAC until the present. It is equal to the sum of **Sys.StartTime**, which contains the time in seconds from the most recent power-up of the Power PMAC processor until the most recent start of the Power PMAC firmware application, and **Sys.RunTime**, which contains the time in seconds from the most recent start of the Power PMAC firmware application until the present.

Sys.WDTFault

Description: "Watchdog timer fault" status element

Range: 0 .. 3

Units: Bit field

Sys.WDTFault is a 2-bit status element indicating the status of the software watchdog timer.

Bit 1 (value 2) "*RT Interrupt soft watchdog timer fault*" is set to 1 if the processor detects a soft watchdog fault because too many background cycles (> **Sys.BgWDTReset**) have elapsed since the last real-time interrupt. In this case, all user programs are disabled, and all hardware interfaces are put in their reset state. This bit is 0 is no such error has been found. It is bit 1 of 32-bit element **Sys.Status**.

Bit 0 (value 1) "*Background soft watchdog timer fault*" is set to 1 if the processor detects a soft watchdog fault because too many real-time interrupt cycles (> **Sys.WDTReset**) have elapsed since the last background cycle. In this case, all user programs are disabled, and all hardware interfaces are put in their reset state. This bit is 0 is no such error has been found. It is bit 0 of 32-bit element **Sys.Status**.

In normal operation, the value of this two-bit element is zero. If a hard watchdog timer trip occurs, the processor is shut down completely, and this element has no value.

Sys.WdTimer

Description: Foreground RTI-cycle watchdog countdown timer

Range: Non-negative integer

Units: Real-time interrupt cycles

Sys.WdTimer contains the present value of the real-time interrupt (RTI) watchdog countdown timer. Each real-time interrupt, this value is decremented by 1. Each background cycle, this value is set to the value of saved setup element **Sys.WDTReset** (or to 5000 if **Sys.WDTReset** is set to its default value of 0). If the value of **Sys.WdTimer** reaches 0, a "soft" watchdog timer trip will occur, disabling motion programs and servo loops, forcing hardware outputs into their "reset" state, and setting status element **Sys.WDTFault** to 1. Such a failure is indicative of the processor time being overloaded with interrupt tasks, leaving insufficient time for background tasks.

POWER PMAC ON-LINE COMMAND SPECIFICATION

This section documents the "on-line" commands in the Power PMAC Script language. On-line commands are executed immediately, and then discarded. Unlike with buffered program commands, you cannot list back a sequence of on-line commands that you have sent to Power PMAC.

#

Function: Report thread's currently addressed motor

Scope: Communications-thread specific

Syntax: **#**

The **#** command causes the Power PMAC to return the number of the motor currently addressed for the communications thread over which this command is sent. This is the motor that will act on subsequent motor-specific commands sent in this thread until a different motor is addressed with a ***#{constant}*** command.

This command queries the value of the thread's data structure element **Ldata.Motor**. It is equivalent to querying that element directly with the **Ldata.Motor** command.

Other communications threads may be addressing different motors at the same time.

At power-on/reset, Power PMAC defaults to addressing Motor 0 (**#0**).

#{constant}

Function: Select thread's addressed motor

Scope: Communications-thread specific

Syntax: ***#{constant}***

where:

- ***{constant}*** is an integer in the range 0 to **Sys.MaxMotors** - 1.

The ***#{constant}*** command makes the motor specified by ***{constant}*** the addressed motor for the communications thread over which this command is sent. This is the motor that will act on subsequent motor-specific commands sent in this thread until a different motor is addressed with another ***#{constant}*** command.

This command sets the value of the thread's data structure element **Ldata.Motor**. It is equivalent to setting that element directly with the **Ldata.Motor=*{expression}*** command.

Note that the ***#{list}*** command can be used to specify multiple motors to act on the motor-specific command immediately following the list; if the presently addressed motor is not in the list, it will not act

on that command. The **#*{list}*** command does not change which motor is addressed for subsequent commands.

Other communications threads may be addressing different motors at the same time.

At power-on/reset, Power PMAC defaults to addressing Motor 0 (**#0**).

Examples

```
#1j+                    // Address Motor 1, command it to jog positive
j-                      // Command addressed motor (#1) to jog negative
#2j+                    // Address Motor 2, command it to jog positive
#3..6hm                 // Command Motors 3 - 6 to start homing-search moves
j/                      // Command addressed motor (#2) to stop jogging
```

#{constant}->0

Function: Assign motor to coordinate system, null definition

Scope: Coordinate-system specific

Syntax: **#*{constant}*->0**

where:

- ***{constant}*** is an integer in the range 0 to to **Sys.MaxMotors** - 1.

The **#*{constant}*->0** command causes Power PMAC to assign the specified motor to the addressed coordinate system, but in a mode where the motor will not perform its own trajectory calculations. Mostly, this mode is used for inactive motors (those with **Motor[*x*].ServoCtrl** = 0) as a way of showing they are not tied to any axis. By default, all motors are assigned to Coordinate System 0 with this definition.

This setting is also appropriate for the case where an active motor has been put in "gantry following" mode by setting **Motor[*x*].ServoCtrl** to 8, and it uses the trajectory calculated by the motor specified by **Motor[*x*].CmdMotor**. In this case, it is still important for the motor to be in the same coordinate system with the "gantry leader" so all gantry motors can take the same action on a fault. Bit 0 of **Motor[*x*].FaultMode** should be set to 1 for all gantry motors so they are all killed on a fatal following error, amplifier fault, or integrated-current fault of any motor in the gantry.

If the motor has a "non-zero" definition in one coordinate system, an attempt to use this definition for another coordinate system will be rejected with an error. However, if the motor has this "zero" definition in one coordinate system, Power PMAC will accept this axis definition (or any other axis definition) in another coordinate system without error. So if you wish to remove a motor from a coordinate system where it has a real axis definition, first define it to "zero" in that coordinate system, then give it an axis definition (real or zero) in another coordinate system.

Note that the action of this command is slightly different from that in Turbo PMAC, where it removed the motor from the coordinate system completely.

#{constant}->{axis definition}

Function: Set specified motor's axis definition

Scope: Coordinate-system specific

Syntax: **#*{constant}*->*{axis definition}***

where:

- ***{constant}*** is an integer in the range 0 to **Sys.MaxMotors** - 1.
- ***{axis definition}*** consists of one or more sets of ***[{constant}]{axis}*** separated by the + or – arithmetic operators, in which:
  - the optional floating-point ***{constant}***is the axis "scale factor" representing the number of motor units per axis unit ("engineering" or "user" unit); if none is explicitly specified, Power PMAC assumes a value of 1.0;
  - ***{axis}*** is the character or double character specifying the axis the motor is to be assigned to (A, B, C, U, V, W, X, Y, Z, AA, BB, … HH, LL, MM, … ZZ);
  - a final optional floating-point ***{constant}*** following a + or – operator representing the offset between the motor zero position and the axis zero position in units of motor units; if none is explicitly specified, Power PMAC assumes a value of 0.0;

The **#*{constant}*->*{axis definition}*** command assigns the specified motor to a set of axes in the addressed coordinate system. It also defines the scaling and starting programming offset for the axis or axes. In the majority of cases, there is a one-to-one matching between Power PMAC motors and axes, so this axis definition then uses only one axis name for the motor.

A scale factor is typically used with the axis character, so that axis moves can be specified in standard units (e.g. millimeters, inches, degrees). *This number is what defines what the user units will be for the axis.* If no scale factor is specified, the user unit for the axis is equal to one motor "count".

Occasionally an offset parameter is used to allow the axis zero position for programming purposes to be different from the motor "home" zero position. (This is the starting offset; it can later be changed through offset commands.)

Each programmed move or segment, Power PMAC solves the equation specified by this axis-definition statement to compute the commanded motor position.

If the specified motor is presently assigned to an axis in a different coordinate system, Power PMAC will reject this command with an error. If the specified motor is presently assigned to an axis in the addressed coordinate system, the old definition will be overwritten by this new one.

Examples

```
&1                          // Address C.S.1, definitions will be to axes in this C.S.
#1->X                       // Motor 1 assigned to X-axis, user units of motor units
#4->2000A                   // Motor 4 assigned to A-axis, 2000 counts per user unit
#9->333.333ZZ-666.667       // Non-integer values OK
```

```
&2                              // Address C.S.2, definitions will be to axes in this C.S.
#2->Y                           // Multiple motors may be assigned to same axis
#3->Y                           // Both motors will move when Y-axis is commanded

&3                              // Address C.S.3, definitions will be to axes in this C.S.
#5->8660X-5000Y                 // These provide a 30° rotation of X and Y
#6->5000X+8660Y                 // With 10,000 counts per user unit

&4                              // Address C.S.4, definitions will be to axes in this C.S.
// These next two definitions create a 2D Cartesian correcting for
// a 1 arc-minute squareness error
#12->10000X                     // 10,000 counts per user unit
#13->-29.1X+10000Y              // The "-29.1X" term is the correction factor
```

#{constant}->I

Function: Define specified motor as inverse-kinematic axis

Scope: Coordinate-system specific

Syntax: **#*{constant}*->I**

where:

- ***{constant}*** is an integer in the range 0 to **Sys.MaxMotors** - 1.

The **#*{constant}*->I** command causes Power PMAC to assign the specified motor as an "inverse-kinematic" axis in the presently addressed coordinate system. A motor assigned in this way must get its commanded positions each programmed move from the user-written inverse-kinematic subroutine for the coordinate system, not simply from an axis-definition equation. At the end of each execution of the inverse-kinematic subroutine, Power PMAC expects to find the commanded position for Motor *xx* in local variable L*xx* for the coordinate system.

If the specified motor is presently assigned to an axis in a different coordinate system, Power PMAC will reject this command with an error. If the specified motor is presently assigned to an axis in the addressed coordinate system, the old definition will be overwritten by this new one.

#{constant}->S[{constant}]

Function: Define specified motor as spindle axis

Scope: Coordinate-system specific

Syntax: **#*{constant}*->S*[{constant}]***

where:

- the first ***{constant}*** is an integer in the range 0 to **Sys.MaxMotors** - 1.
- the optional second ***{constant}*** is 0 or 1.

The **#*{constant}*->S*[{constant}]*** command causes Power PMAC to assign the specified motor as a "spindle" axis in the presently addressed coordinate system. A motor defined this way can be commanded as a motor separately from the positioning axes in the coordinate system, but is still part of the coordinate system for fault reaction purposes. It is typically commanded with motor jogging or open-loop commands in this mode.

Note that it is not necessary to define a motor as a spindle axis to use it as a spindle. If the motor will only be used as a spindle in the application, there is no advantage in defining it as a spindle axis, and it will cause memory to be reserved, but never used, in a lookahead buffer. However, this definition is useful for a rotary axis that is sometimes used as a positioning axis and sometimes used as a velocity-mode spindle. Changing the definition between a rotary positioning axis and a spindle axis during a program does not require reconfiguration of coordinate system structures such as the lookahead buffer.

The optional constant after the letter **S** specifies the "time base" control for the axis when in spindle mode. If the axis is defined without a following constant (e.g. **#4->S**), the motor will use the time base of the coordinate system in which it is defined. In this mode, if the time base of the coordinate system is changed, the speed of the spindle will change. If the positioning axes are stopped with an **h** [**hold**] command, which forces the coordinate system time base to zero, the spindle will stop as well.

If the axis is defined with a **0** following the **S** (e.g. **#4->S0**), the motor will use the time base of Coordinate System 0, regardless of which coordinate system the axis is defined in. In this mode, if the time base of the coordinate system is changed, the speed of the spindle will *not* change. If the positioning axes are stopped with an **h** [**hold**] command, which forces the coordinate system time base to zero, the spindle will *not* stop as well. However, it is possible to modify the speed of the spindle interactively in this mode by changing the time-base value of Coordinate System 0.

If the axis is defined with a **1** following the **S** (e.g. **#4->S1**), the motor will use a fixed time base value of %100, regardless of the time base value of the coordinate system the axis is defined in. In this mode, if the time base of the coordinate system is changed, the speed of the spindle will *not* change. If the positioning axes are stopped with an **h** [**hold**] command, which forces the coordinate system time base to zero, the spindle will *not* stop as well. It is *not* possible to modify the speed of the spindle interactively in this mode by changing the time-base value of any coordinate system.

Note that no axis scale factor (constant before the letter) and no axis offset (added or subtracted after the letter) are permitted in this definition, as there is no conversion from axis positions to motor positions in this mode. A motor defined as a spindle axis will *not* respond directly to **S** commands in a motion program (although it may use the variable value set by an **S** command).

It is important that the move data in the coordinate system's lookahead buffer be fully purged before the axis definition for a motor in the coordinate system is changed to or from a spindle axis to prevent possible inconsistencies in the buffer. The **lhpurge** command is the best way to empty the contents of the buffer.

In older firmware versions, this can be done by declaring a move mode that does not use the lookahead buffer, such as **rapid** mode. This mode only needs to be in effect for an instant to purge the lookahead buffer in preparation for the new axis definition; the next program command can change the move mode back to another mode.

The present "spindle status" of the motor can be found in status element **Motor[*x*].SpindleMotor**. A value of 0 indicates it is not a spindle motor; a value of 1 that it is a spindle motor using the defined C.S.

time base; a value of 2 that it is a spindle motor using the time base of C.S.0; a value of 3 that it is a spindle motor using a fixed 100% time base.

The present state of the coordinate system's lookahead buffer with regard to spindle motors can be evaluated in the status elements **Coord[*x*].LHMotorSlots** and **Coord[*x*].NumMotors**. The **LHMotorSlots** element contains the number of motor "columns" in structure the table, equaling the number of motors defined in the coordinate system in any form, including as spindle axes. The **NumMotors** element contains the number of table motor columns that are presently being used – this excludes any spindle axes.

Examples

A motor is commonly changed between a positioning axis and a spindle axis from within a motion program or its subprogram (often in a G-code subroutine) using the **cmd""** construct. A robust procedure to make this change will look something like:

```
dwell 0;                              // Stop blending and lookahead
lhpurge;                              // Purge lookahead buffer
Ldata.CmdStatus = 1;                  // Will change when command executed
cmd"&1#4->S0";                        // Define Motor 4 as spindle in CS1
sendallcmds;                          // Wait for command buffer to empty
do dwell 0;                           // Loop quickly while waiting
while (Ldata.CmdStatus == 1);         // Until command fully executed
                                      // Could check for error here (if < 0)
```

The comparable procedure to change it back to a rotary positioning axis will look something like:

```
dwell 0;                              // Stop blending and lookahead
lhpurge;                              // Purge lookahead buffer
Ldata.CmdStatus = 1;                  // Will change when command executed
cmd"&1#4->100C";                      // Define Motor 4 as C-axis in CS1
sendallcmds;                          // Wait for command buffer to empty
do dwell 0;                           // Loop quickly while waiting
while (Ldata.CmdStatus == 1);         // Until command fully executed
                                      // Could check for error here (if < 0)
```

#{list}

Function: Specify motors for next command

Scope: Communications-thread specific

Syntax: ***#{list}***

where:

- ***{list}*** is a set of multiple integer constants, separated by commas, and/or ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, all in the range 0 to **Sys.MaxMotors** - 1.

The ***#{list}*** command specifies which motors will be affected by the motor-specific command immediately following the list. This construct permits a single motor command to operate on multiple motors.

If there is no motor-specific command immediately following the list on the same command line, this command will be rejected with an error.

The specification of multiple motors with the ***#{list}*** command only affects the immediately following command; it does not change the modally addressed motor for this communications thread.

Examples

```
#1j+                          // Modally address Motor 1, command to jog positive
#2,3,4j-                      // Command Motors 2, 3, and 4 to jog negative
j/                            // Stop jog on modally addressed motor (Motor 1)
#5..8,12,14..16hm             // Start homing-search move on all motors in list
#3..6                         // Specify list without command
error 20: ILLEGAL CMD         // Power PMAC rejects with error
```

#{list}->

Function: Report axis definition for specified motor(s)

Scope: Global

Syntax: ***#{list}->***

where:

- ***{list}*** is a set of one or more integer constants, separated by commas, and/or ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, all in the range 0 to **Sys.MaxMotors** - 1.

The ***#{list}->*** command causes Power PMAC to report the present axis definition of the specified motor or motors. If a motor has been defined in the presently addressed coordinate system, just the axis definition will be reported. If the motor has been defined in a different coordinate system, Power PMAC will report the number of that coordinate system followed by the definition in that coordinate system.

If there is no definition of this motor in any coordinate system, Power PMAC considers the motor as "belonging to" Coordinate System 0 (&0), but having an axis definition of "0" there.

Note that this command works a little differently than in the older PMAC and Turbo PMAC coordinate systems. In those controllers, the axis definition would only be reported if the motor were defined in the presently addressed coordinate system.

Examples

```
&1                      // Address Coordinate System 1
#1->                    // Request axis definition of Motor 1
#1->500X                // Power PMAC reports definition (in addressed C.S.)

#5->                    // Request axis definition of Motor 5
&2#5->500Y              // Power PMAC reports definition (in different C.S.)

#1..7->                 // Request axis definitions of Motors 1 - 7
#1->500X                // Power PMAC reports definition (in addressed C.S.)
#2->500Y                // Power PMAC reports definition (in addressed C.S.)
#3->500Z                // Power PMAC reports definition (in addressed C.S.)
```

```
&2#4->500X              // Power PMAC reports definition (in separate C.S.)
#5->500Y                // Power PMAC reports definition (in separate C.S.)
#6->500Z                // Power PMAC reports definition (in separate C.S.)
&0#7->0                 // Power PMAC reports definition (in separate C.S.)
```

#*

Function: Specify all motors for next command

Scope: Communications-thread specific

Syntax: **#***

The **#*** command specifies that all active motors will be affected by the motor-specific command immediately following the list. This construct permits a single motor command to operate on all motors.

If there is no motor-specific command immediately following this command on the same command line, this command will be rejected with an error.

The specification of all motors with the **#*** command only affects the immediately following command; it does not change the modally addressed motor for this communications thread.

Examples

```
#1j+                            // Modally address Motor 1, command to jog positive
#*j-                            // Command all motors to jog negative
j/                              // Stop jog on modally addressed motor (Motor 1)
```

$

Function: Establish phase reference for motor

Scope: Motor specific

Syntax: **$**

The **$** command causes Power PMAC to attempt to establish the phase reference and possibly close the servo loop for a PMAC-commutated (**Motor[*x*].PhaseCtrl** bit 0 [value 1] or bit 2 [value 4] = 1) synchronous (**Motor[*x*].DtOverRotorTc** = 0) motor. On other types of motors, where there is no need to establish a phase reference, the **$** command can be used to close the servo loop (a **j/** command is also suitable for these motors.)

This command must immediately be preceded by a motor number (e.g **#1**) or list of motor numbers (e.g. **#2,4** or **#5..8**). Otherwise it will be rejected with an error.

The phase reference can be established either with a phasing-search move, which excites the motor and uses the resulting action to figure out the rotor phase angle, or by reading an absolute position sensor to find the present rotor phase angle. To specify a "phasing read", **Motor[*x*].pAbsPhasePos** must be set to a non-zero value containing the address of the register to read for the absolute phase position, and **Motor[*x*].PhaseFindingTime** must be set to a value less than 5.

For a phasing read, the saved setup elements **Motor[*x*].pAbsPhasePosFormat**, **Motor[*x*].pAbsPhasePosSf**, and **Motor[*x*].AbsPhasePosOffset** must be set properly to process the data that is read correctly.

A phasing-search move is selected if **Motor[*x*].PhaseFindingTime** is 5 or greater. The move uses the saved setup values in **Motor[*x*].PhaseFindingDac** and **Motor[*x*].PhaseFindingTime**. These specify the method ("four guess" or "stepper") and the parameters for the phasing search.

Progress of the phasing-search move or absolute phase read can be monitored with the status element **Motor[*x*].PhaseFindingStep** (> 0 when the search or read is in progress, <= 0 when not). When the referencing is done, this element will be set to 0 if no problem was encountered in the search or read, or to a negative number if a problem was found. Note that Power PMAC cannot detect all possible problems with a phasing search or read, so further protections, such as good settings for **Motor[*x*].FatalFeLimit**, are essential.

The success of the search or read can be also be checked afterwards with the status element **Motor[*x*].PhaseFound**. This is set to 1 if Power PMAC judges that there was sufficient motion in the correct pattern to find the phase reference. It is set to 0 otherwise. The motor cannot be enabled unless this bit is set to 1.

WARNING

Closing the servo loop on a synchronous commutated motor without first establishing a valid phase reference can lead to a dangerous runaway condition.

It is possible to command a phasing reference, either a search or a read, from within a Power PMAC program by setting **Motor[*x*].PhaseFindingStep** to 1.

Bit 0 (value 1) of **Motor[*x*].PowerOnMode** determines the state of the motor after a successful phase referencing is finished. If it is set to the default value of 0, the motor will be left in the "killed" state (open loop, zero output, amplifier disabled). If it is set to 1, the motor will be left in the closed-loop enabled state. However, if Power PMAC judges that it found a problem in the search or read, the motor will be left in the killed state regardless of the setting of this bit.

$$$

Function: Full controller reset

Scope: Global

Syntax: **$$$**

The **$$$** command causes the control application in Power PMAC to do a full reset. The effect on the application is equivalent to cycling power off, then on. However, the computer hardware and operating system continue to operate as this reset occurs. (The on-line **reboot** command causes the computer to restart as well.)

The Power PMAC will force its interface ASICs into "reset mode" at the beginning of execution of this command, forcing all output values to zero. It will release them into "operating mode" as it starts to reload their saved configuration values near the end of the operation.

In executing this command, Power PMAC copies the last saved values of the controller configuration – setup data structure elements, tables, programs, etc. – from non-volatile flash memory to active memory (RAM). *This means that any configuration information in Power PMAC's active memory that was not saved to flash memory will be lost when this command is executed.*

Note that this command causes the copying of the last saved values of the saved setup elements into the active registers first, and then executes the commands in the saved project files. It is possible for on-line commands in these files that write to the setup elements to cause the overwriting of saved values of the setup elements themselves.

Because this command immediately causes the Power PMAC application to enter its power-up/reset cycle, it provides no acknowledging characters to this command.

$$$***

Function: Full controller reset and re-initialization

Scope: Global

Syntax: **$$$*****

The **$$$*** ** command causes Power PMAC to do a full reset, plus a re-initialization of active memory. All user programs and other buffers are erased in active memory. All setup data-structure elements are returned to their factory default values. Note that the last saved configuration in flash memory is not affected by this command, and this configuration can be restored to active memory with a subsequent **$$$** reset command, or by cycling power.

On the **$$$*** ** command, the Power PMAC CPU builds certain default settings based on the hardware it finds. It is strongly recommended to issue this command any time the hardware configuration of the Power PMAC system is changed.

The Power PMAC will force its interface ASICs into "reset mode" at the beginning of execution of this command, forcing all output values to zero. It will release them into "operating mode" as it starts to reload their default configuration values near the end of the operation.

Because this command immediately causes the Power PMAC application to enter its power-up/reset cycle, it provides no acknowledging characters to this command.

%

Function: Report time-base override value

Scope: Coordinate-system specific

Syntax: `%`

The `%` command causes Power PMAC to report the present time-base "override" value for the addressed coordinate system. A value of 100 indicates "real time"; i.e. move speeds and times in the coordinate system occur as specified. A value of 50 indicates that move speeds are only half as fast as specified, so moves will take twice the time they would in "real time". A negative value indicates that the coordinate system is executing moves in the reverse direction.

Power PMAC reports the override value in response to this command regardless of what is controlling the override for this coordinate system (even if the source is not `%{constant}` commands).

A coordinate system with no motors assigned to it through axis definition statements will always report a % value of 0.

%{constant}

Function: Set time-base override value

Scope: Coordinate-system specific

Syntax: `%{constant}`

where:

- `{constant}` is a floating-point value in the range -200.0 to +200.0 specifying the desired time-base override value as a percentage of real time (+100 represents real time)

The `%{constant}` command can be used to set the time-base "override" value for the presently addressed coordinate system. The command causes Power PMAC to set the value of the **Coord[*x*].DesTimeBase** data-structure element to be set to the true servo update time (in milliseconds, as defined by the value of the **Sys.ServoPeriod** setup data-structure element) multiplied by the percentage in `{constant}`.

If the coordinate system has been told to use this register for its "time base" value with the pointer variable **Coord[*x*].pDesTimeBase** containing the address of **Coord[*x*].DesTimeBase** (this is the default), then this value will be used to control the coordinate system's time base. The actual time base value (in the element **Coord[*x*].TimeBase**) will ramp toward this desired value at a rate set by the element **Coord[*x*].TimeBaseSlew**. Each servo update period (which is a physically fixed time), the Power PMAC increments the time value along the commanded trajectories for each motor in the coordinate system by the value in the **Coord[*x*].TimeBase** element. When the value of the used **Coord[*x*].TimeBase** element equals the value of the target **Coord[*x*].DesTimeBase** element, the slewing stops.

A value of +100 in the command specifies "real time"; i.e. move speeds and times in the coordinate system occur as specified. A value of 50 specifies that move speeds are only half as fast as specified, so moves will take twice the time they would in "real time". A value of 0 "freezes" time in the coordinate system (like a "hold" command). Values less than -200 or greater than +200 are rejected with an error.

Negative time-base values for reversing through trajectories are supported when motor "trace buffers" are defined with the **Motor[*x*].TraceSize** saved setup element.

From within a program, the comparable effect can be achieved by writing a value directly to the **Coord[*x*].DesTimeBase** element. For example the program command

```
Coord[1].DesTimeBase = Sys.ServoPeriod * MyPercent / 100
```

sets the percentage value for C.S. 1 to the value of declared user variable "MyPercent". (This type of command could also be used as an on-line command to set the percent override to a variable value.) When set in this way, the value is not limited to the +/-200% range.

&

Function: Report thread's currently addressed coordinate system

Scope: Communications-thread specific

Syntax: **&**

The **&** command causes the Power PMAC to return the number of the coordinate system currently addressed for the communications thread over which this command is sent. This is the coordinate system that will act on subsequent motor-specific commands sent in this thread until a different motor is addressed with a ***&{constant}*** command.

This command queries the value of the thread's data structure element **Ldata.Coord**. It is equivalent to querying that element directly with the **Ldata.Coord** command.

Other communications threads may be addressing different coordinate systems at the same time.

At power-on/reset, Power PMAC defaults to addressing Coordinate System 0 (**&0**).

&{constant}

Function: Select thread's addressed coordinate system

Scope: Communications-thread specific

Syntax: ***&{constant}***

where:

- ***{constant}*** is an integer in the range 0 to **Sys.MaxCoords** - 1.

The ***&{constant}*** command makes the coordinate system specified by ***{constant}*** the addressed coordinate system for the communications thread over which this command is sent. This is the coordinate system that will act on subsequent coordinate-system-specific commands sent in this thread until a different coordinate system is addressed with another ***&{constant}*** command.

This command sets the value of the thread's data structure element **Ldata.Coord**. It is equivalent to setting that element directly with the **Ldata.Coord=*{expression}*** command.

Note that the ***&{list}*** command can be used to specify multiple coordinate systems to act on the coordinate-system-specific command immediately following the list; if the presently addressed coordinate system is not in the list, it will not act on that command. The ***&{list}*** command does not change which coordinate system is addressed for subsequent commands.

Other communications threads may be addressing different coordinate systems at the same time.

Examples

```
&1b1r                   // Address C.S. 1, command it to run from the beginning of PROG 1
q                       // Command addressed C.S. (&1) to quit running program
&2%50                   // Address C.S. 2, command it to set override of 50%
&3..6a                  // Command C.S. 3 - 6 to abort programs and moves
%100                    // Command addressed C.S. (&2) to set override of 100%
```

&{list}

Function: Specify coordinate systems for next command

Scope: Communications-thread specific

Syntax: ***&{list}***

where:

- ***{list}*** is a set of multiple integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to to **Sys.MaxCoords** - 1.

The ***&{list}*** command specifies which coordinate systems will be affected by the coordinate-system-specific command immediately following the list. This construct permits a single coordinate-system command to operate on multiple coordinate systems.

If there is no coordinate-system-specific command immediately following the list on the same command line, this command will be rejected with an error.

The specification of multiple motors with the ***&{list}*** command only affects the immediately following command; it does not change the modally addressed coordinate system for this communications thread.

Examples

```
&1r                         // Modally address C.S. 1, command to run selected program
&2,3,4a                     // Command C.S. 2, 3, and 4 to abort programs and moves
%50                         // Set override of 50% on modally addressed C.S. (&1)
&5..8,12,14..16h            // Command hold on all C.S.'s in list
&3..6                       // Specify list without command
error 20: ILLEGAL CMD       // Power PMAC rejects with error
```

&*

Function: Specify all coordinate systems for next command

Scope: Communications-thread specific

Syntax: **&***

The **&*** command specifies that all active coordinate systems will be affected by the coordinate-system-specific command immediately following the list. This construct permits a single coordinate-system command to operate on all coordinate systems.

If there is no coordinate-system-specific command immediately following this command on the same command line, this command will be rejected with an error.

The specification of all coordinate systems with the **&*** command only affects the immediately following command; it does not change the modally addressed coordinate system for this communications thread.

Examples

```
&1b1r                   // Address C.S. 1, command it to run from the beginning of PROG 1
&*a                     // Command all active C.S. to abort programs and moves
b2r                     // Command C.S 1 to run from beginning of PROG 2
```

\

Function: Quick stop in lookahead

Scope: Coordinate-system specific

Syntax: ****

The **** command causes the Power PMAC to calculate and execute the quickest stop within the lookahead buffer for the specified coordinate system(s) that does not violate acceleration constraints for any motor within the coordinate system. Motion will continue to a controlled stop along the programmed path, but the stop will not necessarily be at a programmed point. If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

The **** quick-stop command is generally the best command to stop motion interactively within lookahead. Its function is much like that of a traditional feed-hold command, but unlike the regular **h** feed-hold command in Power PMAC, it is guaranteed to observe acceleration constraints.

Once stopped, several options are possible:

- Jog axes away with any of the jogging commands. The on-line jog commands can be used to jog any of the motors in the coordinate system away from the stopped point. However, before execution of the programmed path can be resumed, all motors must be returned to the original stopping point with the **j=** (jog return) command.
- Start reverse execution along the path with the **<** command.
- Resume forward execution with the **>**, **r**, or **s** command.

- End program execution with the **a** command.

If the **** command is given to a coordinate system that is not currently executing moves within the segmented lookahead buffer, Power PMAC will execute the **h** "feed-hold" command instead.

The equivalent buffered direct program command is **lh**.

<

Function: Start reverse execution in lookahead buffer

Scope: Coordinate-system specific

Syntax: **<**

The **<** command causes the Power PMAC to start reverse execution in the lookahead buffer for the specified coordinate system(s). If the program is currently executing in the forward direction, it will be brought to a quick stop (the equivalent of the **** command) first. If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

Deceleration from a forward move (if any) and acceleration in the reverse direction observe the **Motor[*x*].InvAmax** acceleration limits.

Execution proceeds backward through points buffered in the lookahead buffer, observing velocity and acceleration constraints just as in the forward direction. This execution continues until one of the following occurs:

- Reverse execution reaches the "beginning" of the lookahead buffer – the oldest stored point still remaining in the lookahead buffer – and it comes to a controlled stop at this point, observing acceleration limits in decelerating to a stop.
- The **** "quick-stop" command is given, which causes Power PMAC to come to the quickest possible stop in the lookahead buffer.
- The **>** "resume-forward", **r** "run", or **s** "step" command is given, which causes Turbo PMAC to resume normal forward execution of the program, adding to the lookahead buffer as necessary.
- An error condition occurs, or a non-recoverable stopping command is given.

If any motor has been jogged away from the "quick-stop" point, and not returned with a **j=** command, Power PMAC will reject the **<** "back-up" command, reporting an error.

If the **<** command is given to a coordinate system that is not currently executing moves within the segmented lookahead buffer, Power PMAC will execute the **h** "feed-hold" command instead.

The equivalent buffered direct program command is **lh<**.

>

Function: Resume forward execution in lookahead buffer

Scope: Coordinate-system specific

Syntax: **>**

The **>** command causes the Power PMAC to resume forward execution in the lookahead buffer for the addressed coordinate system. It is typically used to resume normal operation after a \ "quick-stop" command, or a **<** "back-up" command. If the program is currently executing in the backward direction, it will be brought to a quick stop (the equivalent of the **** command) first. If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

If previous forward execution had been in continuous mode (started with the **r** command), the **>** command will resume it in continuous mode. If previous forward execution had been in single-step mode (started with the **s** command), the **>** command will resume it in single-step mode. The **r** and **s** commands can also be used to resume forward execution, but they may change the continuous/single-step mode.

Deceleration from a backward move (if any) and acceleration in the forward direction observe the **Motor[*x*].InvAmax** acceleration limits.

If any motor has been jogged away from the "quick-stop" point, and not returned with a **j=** command, Power PMAC will reject the **>** "resume" command, reporting an error.

If the **** command is given to a coordinate system that is not currently executing moves within the segmented lookahead buffer, Power PMAC will execute the **r** "run" command instead.

The equivalent buffered direct program command is **lh>**.

?

Function: Report status word value(s)

Scope: Global, motor/coordinate-system specific

Syntax:

?	(global status)
#*{list}*?	(motor status)
&*{list}*?	(coordinate system status)

The **?** command causes Power PMAC to report the present value of the status word(s) for the "global" Power PMAC system, or for the specified motor(s) or coordinate system(s). If not immediately preceded by a motor list or coordinate system list, it will report the value of the global status word. If immediately preceded by a motor list, it will report the values of the status words for all motors in the list. If immediately preceded by a coordinate-system list, it will report the values of the status words for all coordinate systems in the list.

Note that specifying a list of multiple motors or multiple coordinate systems does not change the modally addressed motor or coordinate system for subsequent motor-specific or coordinate-system-specific commands.

The status value is reported as a 16-digit hexadecimal ASCII word for motors and coordinate systems, representing the 2 32-bit status words. The status value is reported as an 8-digit hexadecimal ASCII word for the global status, representing the single 32-bit status word.

Motor Status

For the motor status, the following table shows how the reported status value represents the individual status bits. The two columns on the left explain how the status bits can be interpreted from the hexadecimal ASCII response to this command; the next column gives a basic description of the information in the status bit; the following two columns explain how the status bits can be interpreted from the numerical values of the data structure elements **Motor[*x*].Status[*i*]**, and the last column shows the data structure element name for the bit.

The same information can be received in text form using the element names with the command **`backup Motor[x].Status`**. Each individual element name is reported back with its present value.

Char #	Hex Val	Description	Word #	Bit #	Element Name: Motor[*x*].
1	8	Trigger search move in progress	0	31	**TriggerMove**
1	4	Home search move in progress	0	30	**HomeInProgress**
1	2	Hardware negative limit set	0	29	**MinusLimit**
1	1	Hardware positive limit set	0	28	**PlusLimit**
2	8	Warning following error	0	27	**FeWarn**
2	4	Fatal following error	0	26	**FeFatal**
2	2	Stopped on hardware position limit	0	25	**LimitStop**
2	1	Amplifier fault error	0	24	**AmpFault**
3	8	Software negative limit set	0	23	**SoftMinusLimit**
3	4	Software positive limit set	0	22	**SoftPlusLimit**
3	2	Integrated current (I^2T) fault	0	21	**I2tFault**
3	1	Trigger not found	0	20	**TriggerNotFound**
4	8	Amplifier warning	0	19	**AmpWarn**
4	4	Sensor loss error	0	18	**EncLoss**
4	2	(*Reserved*)	0	17	
4	1	(*Reserved*)	0	16	
5	8	Home complete	0	15	**HomeComplete**
5	4	Desired velocity zero	0	14	**DesVelZero**
5	2	Closed-loop mode	0	13	**ClosedLoop**
5	1	Amplifier enabled	0	12	**AmpEna**
6	8	In position	0	11	**InPos**
6	4	(*Reserved*)	0	10	
6	2	Block request flag set	0	9	**BlockRequest**
6	1	Phase reference established	0	8	**PhaseFound**
7	8	Triggered move speed select	0	7	**TriggerSpeedSel**
7	4	Gantry homing complete	0	6	**GantryHomed**
7	2	Spindle motor definition bit 1	0	5	**SpindleMotor** (bit 1)
7	1	Spindle motor definition bit 0	0	4	**SpindleMotor** (bit 0)
8	8	(*Reserved*)	0	3	
8	4	(*Reserved*)	0	2	
8	2	(*Reserved*)	0	1	

8	1	(*Reserved*)	0	0	
9	8	Motor used in PMATCH calculations	1	31	**Csolve**
9	4	Stopped on software position limit	1	30	**SoftLimit**
9	2	Servo output limited	1	29	**DacLimit**
9	1	Backlash direction	1	28	**BlDir**
10	8	SW limit causing move modification	1	27	**SoftLimitDir**
10	4	(*Reserved*)	1	26	
10	2	(*Reserved*)	1	25	
10	1	(*Reserved*)	1	24	
11	8	(*Reserved*)	1	23	
11	4	(*Reserved*)	1	22	
11	2	(*Reserved*)	1	21	
11	1	(*Reserved*)	1	20	
12	8	(*Reserved*)	1	19	
12	4	(*Reserved*)	1	18	
12	2	(*Reserved*)	1	17	
12	1	(*Reserved*)	1	16	
13	8	(*Reserved*)	1	15	
13	4	(*Reserved*)	1	14	
13	2	(*Reserved*)	1	13	
13	1	(*Reserved*)	1	12	
14	8	(*Reserved*)	1	11	
14	4	(*Reserved*)	1	10	
14	2	(*Reserved*)	1	9	
14	1	(*Reserved*)	1	8	
15	8	(*Reserved*)	1	7	
15	4	(*Reserved*)	1	6	
15	2	(*Reserved*)	1	5	
15	1	(*Reserved*)	1	4	
16	8	(*Reserved*)	1	3	
16	4	(*Reserved*)	1	2	
16	2	(*Reserved*)	1	1	
16	1	(*Reserved*)	1	0	

First character returned:

Bit 31 ***Trigger search move in progress*** (**TriggerMove**): This bit is set to 1 at the beginning of a move-until-trigger (homing search, jog-until-trigger, program rapid-mode move-until-trigger). It is set to 0 when the pre-specified trigger condition is found and the post-trigger move is started, or when the move ends without a trigger being found.

Bit 30 ***Home search move in progress*** (**HomeInProgress**): This bit is set to 1 at the beginning of a homing search move. It is set to 0 when the pre-specified trigger condition is found and the post-trigger move is started, or when the move ends without a trigger being found.

Bit 29 ***Hardware negative limit set*** (**MinusLimit**): This bit is set to 1 when the motor is presently in its negative hardware limit. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously.

Bit 28 ***Hardware positive limit set*** (**PlusLimit**): This bit is set to 1 when the motor is presently in its positive hardware limit. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously.

Second character returned:

Bit 27 ***Warning following error*** (**FeWarn**): This bit is 1 if the magnitude of the following error for the motor exceeds its warning following error limit as set by **Motor[*x*].WarnFeLimit**. It is 0 if the magnitude of the following error is less than this limit, or if the motor has been disabled due to exceeding its fatal following error limit.

Bit 26 ***Fatal following error*** (**FeFatal**): This bit is 1 if the motor has been disabled because the magnitude of the following error for the motor has exceeded its fatal following error limit as set by **Motor[*x*].FatalFeLimit**. It is 0 at all other times, becoming 0 again when the motor is re-enabled.

Bit 25 ***Stopped on hardware position limit*** (**LimitStop**): This bit is 1 if the motor has stopped because it hit either its positive or negative hardware overtravel limit, even if it is presently not in that limit. It is 0 at all other times, including when into a limit, but moving out of it.

Bit 24 ***Amplifier fault error*** (**AmpFault**): This bit is 1 if the motor has been disabled because of an amplifier fault error, even if the amplifier fault signal condition is no longer present, or because of a calculated "I^2T" integrated current fault (in which case bit 21 is also set). It is 0 at all other times, becoming 0 again when the motor is re-enabled.

Third character returned:

Bit 23 ***Software negative limit set*** (**SoftMinusLimit**): This bit is set to 1 when the motor has reached or exceeded its negative software limit as set by **Motor[*x*].MinPos** (which must be less than **Motor[*x*].MaxPos** to be active). It is 0 otherwise, even if the motor is still stopped from having hit this limit previously.

Bit 22 ***Software positive limit set*** (**SoftPlusLimit**): This bit is set to 1 when the motor has reached or exceeded its positive software limit as set by **Motor[*x*].MaxPos** (which must be greater than **Motor[*x*].MinPos** to be active). It is 0 otherwise, even if the motor is still stopped from having hit this limit previously.

Bit 21 ***Integrated current (I^2T) fault*** (**I2tFault**): This bit is set to 1 when the motor has been disabled from exceeding its integrated current limit as set by **Motor[*x*].I2tTrip**. The amplifier fault bit (bit 24) will also be set in this case. It will be 0 at all other times, becoming 0 when the motor is re-enabled. (Note that if the amplifier faults due to its own integrated current fault calculations, this bit will not be set.)

Bit 20 ***Trigger not found*** (**TriggerNotFound**): This bit is set to 1 if a jog-until-trigger or program rapid-mode move-until-trigger ends without the pre-specified trigger condition being found. This is not an error condition, but subsequent actions will often depend on whether this bit is set or not. It is 0 at all other times, changing back to 0 when the next move is started.

Fourth character returned:

Bit 19 ***Amplifier warning*** (**AmpWarn**): This bit is set to 1 if **Motor[*x*].AmpFaultLevel** bit 1 (value 2) is set to 1, requiring two consecutive readings of the amplifier fault bit in its specified fault state to trigger an error, and there has been one reading of the amplifier fault bit in its fault state.

Bits 18 .. 16 ***{Reserved for future use}***

Fifth character returned:

Bit 15 ***Position reference established*** (**HomeComplete**): This bit is set to 1 if a position reference is properly established for the motor, either with a homing-search move, or an absolute position read. It is automatically set to 0 at power-up/reset, and at the beginning of a homing-search move. In a homing-search move, it is set to 1 when the pre-specified trigger condition is found, which is before the post-trigger portion of the move is complete.

Bit 14 ***Desired velocity zero*** (**DesVelZero**): This bit is set to 1 if the motor is in closed-loop control and the commanded velocity is zero (i.e. it is trying to hold position), or it is in open-loop mode (enabled or disabled) with the actual velocity exactly equal to zero. It is zero either if the motor is in closed-loop mode with non-zero commanded velocity, or if it is in open-loop mode with non-zero actual velocity.

Bit 13 ***Closed-loop mode*** (**ClosedLoop**): This bit is set to 1 if the motor is in closed-loop control. It is zero if the motor is in open-loop mode (enabled or disabled).

Bit 12 ***Amplifier Enabled*** (**AmpEna**): This bit is set to 1 if the motor is enabled (in closed-loop or open-loop control). Note that there does not need to be an active amplifier-enable output signal in this case. This bit is 0 if the motor is disabled.

Sixth character returned:

Bit 11 ***In position*** (**InPos**): This bit is set to 1 when all of the conditions for "in position" are satisified: the motor is closed-loop, the desired velocity is zero, the move timer is not active (no move, dwell, or delay being executed, the magnitude of the following error is less than or equal to **Motor[*x*].InPosBand**, and all of these conditions have been true for (**Motor[*x*].InPosTime** – 1) consecutive servo cycles. It is 0 otherwise.

Bit 10 ***{Reserved for future use}***

Bit 9 ***Block request flag set*** (**BlockRequest**): This bit is set to 1 if the motor has just entered a new move section, and is requesting that the equations for the next upcoming move section for the motion queue be calculated. It is 0 otherwise. It is primarily for internal use.

Bit 8 ***Phase reference established*** (**PhaseFound**): This bit is set to 0 on power-up/reset for a motor (**Motor[*x*].PhaseCtrl** bit 0 or bit 2 = 1) commutated by Power PMAC that is synchronous (**Motor[*x*].DtOverRotorTc** = 0.0). It is set to 1 if a phase reference is properly established for the motor, either with a phasing-search move, or an absolute position read.

Seventh character returned:

Bit 7 ***Trigger Speed Select*** (**TriggerSpeedSel**): This bit is set to 1 during a move-until-trigger if the move is done at the velocity specified by **Motor[*x*].MaxSpeed**. It is set to 0 if the move is done at the velocity specified by **Motor[*x*].JogSpeed**.

Bit 6 ***Gantry homing complete*** (**GantryHomed**): This bit is set to 1 if this motor is a follower in a leader/follower gantry system, the home triggers for both leader and follower motors have been found, and the skew between motors has been fully removed. It is 0 otherwise.

Bit 5 ***Spindle definition bit 1*** (**SpindleMotor** bit 1): This bit is set to 1 if this motor is defined as a spindle axis and uses either C.S. 0's time base (**S0**) or a fixed %100 time base (**S1**). It is set to 0 if the motor is not defined as a spindle axis, or if it is defined as a spindle axis using the defined coordinate system's time base (**S**).

Bit 4 ***Spindle definition bit 0*** (**SpindleMotor** bit 0): This bit is set to 1 if this motor is defined as a spindle axis and uses either the defined coordinate system's time base (**S**) or a fixed %100 time base (**S1**). It is set to 0 if the motor is not defined as a spindle axis, or if it is defined as a spindle axis using C.S. 0's time base (**S0**).

Eighth character returned:

Bits 3 .. 0 ***{Reserved for future use}***

Ninth character returned:

Bit 31 ***Motor used in PMATCH calculations*** (**Csolve**): This bit is set to 1 if this motor's position is used to calculate axis positions in a coordinate-system "pmatch" (position match) function (at motion-program start, **pmatch** command execution, axis position/velocity/following-error query). It is 0 when the motor has a "null" definition or a redundant axis definition, and so is not used in these calculations.

Bit 30 ***Stopped on software position limit*** (**SoftLimit**): This bit is 1 if the motor has stopped, or if the present move will be stopped, because it reached or will reach either its positive or negative software overtravel limit, even if it is presently not in that limit. It is 0 at all other times, including when into a limit, but moving out of it.

Bit 29 ***Servo output limited*** (**DacLimit**): This bit is 1 if the motor's servo output command value is presently saturated to the magnitude of **Motor[*x*].MaxDac**. It is 0 otherwise.

Bit 28 ***Backlash direction*** (**BlDir**): This bit is 1 if the motor's backlash function is enabled and the motor is executing or has most recently executed a position move in the negative direction. It is 0 otherwise.

Tenth character returned:

Bit 27 ***Encoder loss error*** (**EncLoss**): This bit is set to 1 if this motor has been disabled because of an encoder-loss error, even if the error condition is no longer present. It is 0 otherwise, becoming 0 again when the motor is re-enabled.

Bit 26 ***Soft limit direction*** (**SoftLimitDir**): This bit is set to 1 if the most recent move stopped, modified, or rejected due to a software position limit was affected because of the negative soft limit. It is 0 if such move was limited by the positive soft limit, or if no such limiting has occurred since power-on/reset.

Bits 25 .. 24 ***{Reserved for future use}***

Eleventh character returned:

Bits 23 .. 20 ***{Reserved for future use}***

Twelfth character returned:

Bits 19 .. 16 ***{Reserved for future use}***

Thirteenth character returned:

Bits 15 .. 12 ***{Reserved for future use}***

Fourteenth character returned:

Bits 11 .. 8 ***{Reserved for future use}***

Fifteenth character returned:

Bits 7 .. 4 ***{Reserved for future use}***

Sixteenth character returned:

Bits 3 .. 0 ***{Reserved for future use}***

Coordinate System Status

For the coordinate-system status, the following table shows how the reported status value represents the individual status bits. The two columns on the left explain how the status bits can be interpreted from the hexadecimal ASCII response to this command; the next column gives a basic description of the information in the status bit; the following two columns explain how the status bits can be interpreted from the numerical values of the data structure elements **Coord[*x*].Status[*i*]**, and the last column shows the data structure element name for the bit.

The same information can be received in text form using the element names with the command **`backup Coord[x].Status`**. Each individual element name is reported back with its present value.

Char #	Hex Val	Description	Word #	Bit #	Element Name: Coord[*x*].
1	8	Trigger search move in progress (any motor in C.S.)	0	31	**TriggerMove**
1	4	Home search move in progress (any motor in C.S.)	0	30	**HomeInProgress**
1	2	Hardware negative limit set (any motor in C.S.)	0	29	**MinusLimit**
1	1	Hardware positive limit set (any motor in C.S.)	0	28	**PlusLimit**
2	8	Warning following error (any motor in C.S.)	0	27	**FeWarn**
2	4	Fatal following error (any motor in C.S.)	0	26	**FeFatal**
2	2	Stopped on hardware limit (any motor in C.S.)	0	25	**LimitStop**
2	1	Amplifier fault (any motor in C.S.)	0	24	**AmpFault**
3	8	Software negative limit set (any motor in C.S.)	0	23	**SoftMinusLimit**
3	4	Software positive limit set (any motor in C.S.)	0	22	**SoftPlusLimit**
3	2	Integrated current (I^2T) fault (any motor in C.S.)	0	21	**I2tFault**
3	1	Trigger not found (any motor in C.S.)	0	20	**TriggerNotFound**
4	8	Amp warning (any motor in C.S.)	0	19	**AmpWarn**
4	4	Encoder loss error (any motor in C.S.)	0	18	**EncLoss**
4	2	(*Reserved*)	0	17	
4	1	Move timer enabled	0	16	**TimerEnabled**

5	8	Home complete (all motors in C.S.)	0	15	**HomeComplete**
5	4	Desired velocity zero (all motors in C.S.)	0	14	**DesVelZero**
5	2	Closed-loop mode (all motors in C.S.)	0	13	**ClosedLoop**
5	1	Amplifier enabled (all motors in C.S.)	0	12	**AmpEna**
6	8	In position (all motors in C.S.)	0	11	**InPos**
6	4	(Reserved)	0	10	
6	2	Block request flag set	0	9	**BlockRequest**
6	1	Timers enabled	0	8	**TimersEnabled**
7	8	XX/YY/ZZ-axis circle radius error / Error word bit 7	0	7	**RadiusError (bit 1)** **ErrorStatus (bit 7)**
7	4	X/Y/Z-axis circle radius error / Error word bit 6	0	6	**RadiusError (bit 0)** **ErrorStatus (bit 6)**
7	2	Stopped on software position limit (any motor in C.S.) / Error word bit 5	0	5	**SoftLimit** **ErrorStatus (bit 5)**
7	1	Run time error / Error word bit 4	0	4	**RunTimeError** **ErrorStatus (bit 4)**
8	8	PVT mode error / Error word bit 3	0	3	**PvtError** **ErrorStatus (bit 3)**
8	4	Linear-to-PVT mode error / Error word bit 2	0	2	**LinToPvtError** **ErrorStatus (bit 2)**
8	2	Buffer error / Error word bit 1	0	1	**ErrorStatus (bit 1)**
8	1	Synchronous assignment buffer error / Error word bit 0	0	0	**ErrorStatus (bit 0)**
9	8	Valid coordinate system axis definition for PMATCH	1	31	**Csolve**
9	4	Linear-to-PVT move buffered	1	30	**LinToPvtBuf**
9	2	Feed hold accel/decel	1	29	**FeedHold (bit 1)**
9	1	Feed hold time base source	1	28	**FeedHold (bit 0)**
10	8	Block active	1	27	**BlockActive**
10	4	Continuous motion request	1	26	**ContMotion**
10	2	Cutter comp mode bit 1	1	25	**CCMode (bit 1)**
10	1	Cutter comp mode bit 0	1	24	**CCMode (bit 0)**
11	8	Move mode bit 1 (=0 for blended and spline, 1 for rapid and pvt modes)	1	23	**MoveMode (bit 1)**
11	4	Move mode bit 0 (=0 for blended and pvt, 1 for rapid and spline modes)	1	22	**MoveMode (bit 0)**
11	2	Segmented PVT-mode move in progress	1	21	**SegMove (bit 1)**
11	1	Segmented linear-mode move in progress	1	20	**SegMove (bit 0)**
12	8	First segment move	1	19	**SegMoveAccel**
12	4	Last segment move	1	18	**SegMoveDecel**
12	2	Segmentation enabled	1	17	**SegEnabled**
12	1	Segment stop request	1	16	**SegStopReq**
13	8	Lookahead wrap	1	15	**LookAheadWrap** **LHStatus (bit 7)**
13	4	Lookahead lookback	1	14	**LookAheadLookBack** **LHStatus (bit 6)**
13	2	Lookahead direction	1	13	**LookAheadDir** **LHStatus (bit 5)**
13	1	Lookahead stop	1	12	**LookAheadStop** **LHStatus (bit 4)**
14	8	Lookahead change	1	11	**LookAheadChange** **LHStatus (bit 3)**
14	4	Lookahead recalculation	1	10	**LookAheadReCalc** **LHStatus (bit 2)**

14	2	Lookahead flush	1	9	**LookAheadFlush LHStatus (bit 1)**
14	1	Lookahead active	1	8	**LookAheadActive LHStatus (bit 0)**
15	8	Cutter comp added arc	1	7	**CCAddedArc**
15	4	Cutter comp turn-off request	1	6	**CCOffReq**
15	2	(*Reserved*)	1	5	
15	1	Cutter comp move type bit 1	1	4	**CCMoveType (bit 1)**
16	8	Cutter comp move type bit 0	1	3	**CCMoveType (bit 0)**
16	4	3D cutter comp active	1	2	**CC3Active**
16	2	Blending disabled for sharp corner	1	1	**SharpCornerStop**
16	1	Added dwell disable	1	0	**AddedDwellDis**

First character returned:

Bit 31 ***Trigger search move in progress*** (**TriggerMove**): This bit is set to 1 at the beginning of a move-until-trigger (homing search, jog-until-trigger, program rapid-mode move-until-trigger) for any motor in the coordinate system. It is set to 0 when the pre-specified trigger condition is found and the post-trigger move is started, or when the move ends without a trigger being found.

Bit 30 ***Home search move in progress*** (**HomeInProgress**): This bit is set to 1 at the beginning of a homing search move for any motor in the coordinate system. It is set to 0 when the pre-specified trigger condition is found and the post-trigger move is started, or when the move ends without a trigger being found.

Bit 29 ***Hardware negative limit set*** (**MinusLimit**): This bit is set to 1 when any motor in the coordinate system is presently in its negative hardware limit. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously.

Bit 28 ***Hardware positive limit set*** (**PlusLimit**): This bit is set to 1 when any motor in the coordinate system is presently in its positive hardware limit. It is 0 otherwise, even if the motor is still stopped from having hit this limit previously.

Second character returned:

Bit 27 ***Warning following error*** (**FeWarn**): This bit is 1 if the magnitude of the following error for any motor in the coordinate system exceeds its warning following error limit as set by **Motor[*x*].WarnFeLimit**. It is 0 if the magnitude of the following error is less than this limit, or if the motor has been disabled due to exceeding its fatal following error limit.

Bit 26 ***Fatal following error*** (**FeFatal**): This bit is 1 if any motor in the coordinate system has been disabled because the magnitude of the following error for the motor has exceeded its fatal following error limit as set by **Motor[*x*].FatalFeLimit**. It is 0 at all other times, becoming 0 again when the motor is re-enabled.

Bit 25 ***Stopped on hardware position limit*** (**LimitStop**): This bit is 1 if any motor in the coordinate system has stopped because it hit either its positive or negative hardware overtravel limit, even if it is presently not in that limit. It is 0 at all other times, including when into a limit, but moving out of it.

Bit 24 ***Amplifier fault error*** (**AmpFault**): This bit is 1 if any motor in the coordinate system has been disabled because of an amplifier fault error, even if the amplifier fault signal condition is no longer

present, or because of a calculated "I^2T" integrated current fault (in which case bit 21 is also set). It is 0 at all other times, becoming 0 again when the motor is re-enabled.

Third character returned:

Bit 23 ***Software negative limit set*** (**SoftMinusLimit**): This bit is set to 1 when any motor in the coordinate system has reached or exceeded its negative software limit as set by **Motor[*x*].MinPos** (which must be less than **Motor[*x*].MaxPos** to be active). It is 0 otherwise, even if the motor is still stopped from having hit this limit previously.

Bit 22 ***Software positive limit set*** (**SoftPlusLimit**): This bit is set to 1 when any motor in the coordinate system has reached or exceeded its positive software limit as set by **Motor[*x*].MaxPos** (which must be greater than **Motor[*x*].MinPos** to be active). It is 0 otherwise, even if the motor is still stopped from having hit this limit previously.

Bit 21 ***Integrated current (I^2T) fault*** (**I2tFault**): This bit is set to 1 when any motor in the coordinate system has been disabled from exceeding its integrated current limit as set by **Motor[*x*].I2tTrip**. The amplifier fault bit (bit 24) will also be set in this case. It will be 0 at all other times, becoming 0 when the motor is re-enabled. (Note that if the amplifier faults due to its own integrated current fault calculations, this bit will not be set.)

Bit 20 ***Trigger not found*** (**TriggerNotFound**): This bit is set to 1 if, for any motor in the coordinate system a jog-until-trigger or program rapid-mode move-until-trigger ends without the pre-specified trigger condition being found. This is not an error condition, but subsequent actions will often depend on whether this bit is set or not. It is 0 at all other times, changing back to 0 when the next move is started.

Fourth character returned:

Bit 19 ***Amplifier warning*** (**AmpWarn**): This bit is set to 1 if, for any motor in the coordinate system, **Motor[*x*].AmpFaultLevel** bit 1 (value 2) is set to 1, requiring two consecutive readings of the amplifier fault bit in its specified fault state to trigger an error, and there has been one reading of the amplifier fault bit in its fault state.

Bit 18 ***Encoder loss error*** (**EncLoss**): This bit is 1 if any motor in the coordinate system has been disabled because of an encoder loss error. It is 0 at all other times, becoming 0 again when the motor is re-enabled.

Bit 17 ***{Reserved for future use}***

Bit 16 ***Move timer enabled*** (**TimerEnabled**): This bit is set to 1 if any motor in the coordinate system is performing a move or move section of definite time. It is 0 otherwise.

Fifth character returned:

Bit 15 ***Position reference established*** (**HomeComplete**): This bit is set to 1 if a position reference is properly established for all motors in the coordinate system, either with a homing-search move, or an absolute position read. It is automatically set to 0 at power-up/reset, and at the beginning of a homing-search move for a motor.

Bit 14 ***Desired velocity zero*** (**DesVelZero**): This bit is set to 1 if all motors in the coordinate system are in closed-loop control and the commanded velocity is zero (i.e. are trying to hold position), or are in

open-loop mode (enabled or disabled) with the actual velocity exactly equal to zero. It is zero either if any motor in the coordinate system is in closed-loop mode with non-zero commanded velocity, or in open-loop mode with non-zero actual velocity.

Bit 13 ***Closed-loop mode*** (**ClosedLoop**): This bit is set to 1 if all motors in the coordinate system are in closed-loop control. It is zero if any motor in the coordinate system is in open-loop mode (enabled or disabled).

Bit 12 ***Amplifier Enabled*** (**AmpEna**): This bit is set to 1 if all motors in the coordinate system are enabled (in closed-loop or open-loop control). Note that there do not need to be active amplifier-enable output signals in this case. This bit is 0 if any motor in the coordinate system is disabled.

Sixth character returned:

Bit 11 ***In position*** (**InPos**): This bit is set to 1 when all of the conditions for "in position" are satisfied for every motor in the coordinate system: the motor is closed-loop, the desired velocity is zero, the move timer is not active (no move, dwell, or delay being executed, the magnitude of the following error is less than or equal to **Motor[*x*].InPosBand**, and all of these conditions have been true for (**Motor[*x*].InPosTime** – 1) consecutive servo cycles. It is 0 otherwise.

Bit 10 ***{Reserved for future use}***

Bit 9 ***Block request flag set*** (**BlockRequest**): This bit is set to 1 if any motor in the coordinate system has just entered a new move section, and is requesting that the equations for the next upcoming move section for the motion queue be calculated. It is 0 otherwise. It is primarily for internal use.

Bit 8 ***Move timers enabled*** (**TimersEnabled**): This bit is set to 1 if every motor in the coordinate system is performing a move or move section of definite time. It is 0 otherwise.

Seventh character returned:

Bit 7 ***XX/YY/ZZ-axis circle radius error / Error word bit 7*** (**RadiusError** bit 1 **/ ErrorStatus** bit 7): This bit is set to 1 if a circle move in the XX/YY/ZZ axis space has a radius error exceeding that set by **Coord[*x*].RadiusErrorLimit** (in which case none of the other 7 bits of **ErrorStatus** will be set). If another of these bits is also set, then it simply indicates that bit 7 (value 128) of the error code is 1. Refer to a list of the error codes, below. The bit is 0 if there is no error code with bit 7 set.

Bit 6 ***X/Y/Z-axis circle radius error / Error word bit 6*** (**RadiusError** bit 0 **/ ErrorStatus** bit 6): This bit is set to 1 if a circle move in the X/Y/Z axis space has a radius error exceeding that set by **Coord[*x*].RadiusErrorLimit** (in which case none of the other 7 bits of **ErrorStatus** will be set). If another of these bits is also set, then it simply indicates that bit 6 (value 64) of the error code is 1. Refer to a list of the error codes, below. The bit is 0 if there is no error code with bit 6 set.

Bit 5 ***Stopped on software position limit / Error word bit 5*** (**SoftLimit / ErrorStatus** bit 5): This bit is 1 if a motion program in the coordinate system has stopped because a motor in the coordinate system reached either its positive or negative software overtravel limit, even if it is presently not in that limit (in which case none of the other 7 bits of **ErrorStatus** will be set). If another of these bits is also set, then it simply indicates that bit 5 (value 32) of the error code is 1. Refer to a list of the error codes, below. It is 0 at all other times, including when into a limit, but moving out of it.

Bit 4 ***Run time error / Error word bit 4*** (**RunTimeError / ErrorStatus** bit 4): This bit is 1 if the motion program in the coordinate system has been aborted due to lack of calculation time (in which case none of the other 7 bits of **ErrorStatus** will be set). If another of these bits is also set, then it simply indicates that bit 4 (value 16) of the error code is 1. Refer to a list of the error codes, below. It is 0 at all other times.

Eighth character returned:

Bit 3 ***PVT mode error / Error word bit 3*** (**PvtError / ErrorStatus** bit 3): This bit is 1 if the motion program in the coordinate system has been aborted due to an error in executing a PVT-mode move (in which case none of the other 7 bits of **ErrorStatus** will be set). If another of these bits is also set, then it simply indicates that bit 3 (value 8) of the error code is 1. Refer to a list of the error codes, below. It is 0 at all other times.

Bit 2 ***Linear-to-PVT mode error / Error word bit 2*** (**LinToPvtError / ErrorStatus** bit 2): This bit is 1 if the motion program in the coordinate system has been aborted due to an error in executing a linear-to-PVT-mode move (in which case none of the other 7 bits of **ErrorStatus** will be set). If another of these bits is also set, then it simply indicates that bit 2 (value 4) of the error code is 1. Refer to a list of the error codes, below. It is 0 at all other times.

Bit 1 ***Buffer error / Error word bit 1*** (**BufferError / ErrorStatus** bit 1): This bit is 1 if the motion program in the coordinate system has been aborted due to an error in the program buffer (in which case none of the other 7 bits of **ErrorStatus** will be set). If another of these bits is also set, then it simply indicates that bit 1 (value 2) of the error code is 1. Refer to a list of the error codes, below. It is 0 at all other times.

Bit 0 ***Synchronous assignment buffer error / Error word bit 0*** (**SyncBufferError / ErrorStatus** bit 0): This bit is 1 if the motion program in the coordinate system has been aborted due to an error in the synchronous assignment buffer (in which case none of the other 7 bits of **ErrorStatus** will be set). If another of these bits is also set, then it simply indicates that bit 1 (value 2) of the error code is 1. Refer to a list of the error codes, below. It is 0 at all other times.

Ninth character returned:

Bit 31 ***Valid axis definitions for PMATCH calculations*** (**Csolve**): This bit is set to 1 if the axis definitions for this coordinate system are satisfactory to calculate axis positions in a coordinate-system "PMATCH" (position match) function (at motion-program start, **pmatch** command execution, axis position/velocity/following-error query). It is 0 otherwise.

Bit 30 ***Linear-to-PVT move buffered*** (**LinToPvtBuf**): This bit is 1 if a linear mode move that has been converted to a PVT move (because **Coord[*x*].SegLinToPvt** > 0) is presently buffered and awaiting execution. It is 0 otherwise. This bit is primarily for internal use.

Bit 29 ***Feed hold accel/decel*** (**FeedHold** bit 1): This bit is 1 if the coordinate system is presently decelerating to a stop due to a "feed hold" (h, hold) command, or a "quick stop" (****, **lh**) command outside of lookahead, or re-accelerating out of this condition. It is 0 otherwise.

Bit 28 ***Feed hold time base*** (**FeedHold** bit 0): This bit is 1 if the coordinate system is presently using the "feed hold time base" rather than its normally specified time base. This is the case during deceleration and full stop due to a "feed hold" (h, hold) command, or a "quick stop" (****, **lh**) command outside of lookahead. It is 0 otherwise, including during re-acceleration from a feed-hold condition.

Tenth character returned:

Bit 27 ***Block Active*** (**BlockActive**): This bit is set to 1 if the coordinate system is presently calculating between a `bstart` (block-start) and a **`bstop`** (block-stop) command. It is 0 otherwise. It is primarily for internal use.

Bit 26 ***Continuous motion request*** (**ContMotion**): This bit is set to 1 if the coordinate system is executing a sequence of programmed moves that it is blending together without stops in between. It is 0 if it is not currently executing such a sequence, for whatever reason.

Bit 25 ***Cutter compensation mode bit 1*** (**CCMode** bit 1): This bit is set to 1 if the cutter compensation mode is set to 2 or 3 (by **`ccmode2`** or **`ccmode3`**). It is 0 otherwise.

Bit 24 ***Cutter compensation mode bit 0*** (**CCMode** bit 0): This bit is set to 1 if the cutter compensation mode is set to 1 or 3 (by **`ccmode1`** or **`ccmode3`**). It is 0 otherwise.

Eleventh character returned:

Bit 23 ***Move mode bit 1*** (**MoveMode** bit 1): This bit is set to 1 if the coordinate system is executing a PVT or rapid move (**MoveMode** = 2 for PVT, = 3 for rapid), or has most recently executed a move of one of these modes. It is set to 0 if the coordinate system is executing a blended (linear or circle) or spline move (**MoveMode** = 0 for blended, = 1 for spline), or has most recently executed a move of one of these modes.

Bit 22 ***Move mode bit 0*** (**MoveMode** bit 0): This bit is set to 1 if the coordinate system is executing a spline or rapid move (**MoveMode** = 1 for spline, = 3 for rapid), or has most recently executed a move of one of these modes. It is set to 0 if the coordinate system is executing a blended (linear or circle) or PVT move (**MoveMode** = 0 for blended, = 2 for PVT), or has most recently executed a move of one of these modes.

Bit 21 ***Segmented PVT-mode move in progress*** (**SegMove** bit 1): This bit is set to 1 if the coordinate system is presently executing a PVT move in segmentation mode. It is 0 otherwise. This bit is primarily for internal use.

Bit 20 ***Segmented blended-mode move in progress*** (**SegMove** bit 0): This bit is set to 1 if the coordinate system is presently executing a linear or circle mode move in segmentation mode. It is 0 otherwise. This bit is primarily for internal use.

Twelfth character returned:

Bit 19 ***First move segment executing*** (**SegMoveAccel**): This bit is set to 1 if the coordinate system is presently executing the initial (added) segment from a stop in a segmented move or move sequence. It is 0 otherwise. This bit is primarily for internal use.

Bit 18 ***Last move segment executing*** (**SegMoveDecel**): This bit is set to 1 if the coordinate system is presently executing the final (added) segment to a stop in a segmented move or move sequence. It is 0 otherwise. This bit is primarily for internal use.

Bit 17 ***Segmented move executing*** (**SegEnabled**): This bit is set to 1 if the coordinate system is presently executing (or has most recently executed) a segmented move (linear, circle, or pvt move with **Coord[*x*].SegMoveTime** > 0). It is 0 otherwise.

Bit 16 ***Segmented move stop request*** (**SegStopReq**): This bit is set to 1 if the coordinate system has been requested to stop during a segmented move sequence. It is 0 otherwise. This bit is primarily for internal use.

Thirteenth character returned:

Bit 15 ***Lookahead buffer wraparound / Lookahead status bit 7*** (**LookAheadWrap** / **LHStatus** bit 7): This bit is set to 1 if the lookahead buffer has filled and "wrapped around" in the present continuous move sequence. This means that it is not possible to reverse through the entire sequence. It is 0 otherwise.

Bit 14 ***Lookahead lookback / Lookahead status bit 6*** (**LookAheadLookBack** / **LHStatus** bit 6): This bit is set to 1 if the lookahead algorithm is presently calculating "backwards" in the buffer to ensure that a controlled stop is possible at the end of the most recently added move. It is 0 otherwise.

Bit 13 ***Lookahead direction / Lookahead status bit 5*** (**LookAheadDir** / **LHStatus** bit 5): This bit is set to 1 if the lookahead buffer is presently executing in the reverse direction. It is 0 otherwise.

Bit 12 ***Lookahead stop / Lookahead status bit 4*** (**LookAheadStop** / **LHStatus** bit 4): This bit is set to 1 if the execution of the lookahead buffer is stopped or decelerating to a stop due to a "quick-stop" (**\, lh**) command. It is 0 otherwise.

Fourteenth character returned:

Bit 11 ***Lookahead execution change / Lookahead status bit 3*** (**LookAheadChange** / **LHStatus** bit 3): This bit is set to 1 if the execution mode of the lookahead buffer is presently changing between any two of the modes of forward, reverse, and stopped. It is 0 otherwise.

Bit 10 ***Lookahead buffer recalculation / Lookahead status bit 2*** (**LookAheadReCalc** / **LHStatus** bit 2): This bit is set to 1 if the lookahead calculations are working within the already existing buffer, as on reversal or "recovery" from reversal, rather than adding new segments to the buffer. It is 0 otherwise.

Bit 9 ***Lookahead buffer flush / Lookahead status bit 1*** (**LookAheadFlush** / **LHStatus** bit 1): This bit is set to 1 if the lookahead buffer has reached the end of a sequence and is no longer adding new segments to the buffer (but is still executing existing segments). It is 0 otherwise.

Bit 8 ***Lookahead active / Lookahead status bit 0*** (**LookAheadActive** / **LHStatus** bit 0): This bit is set to 1 if the coordinate system is presently executing within the lookahead buffer. It is 0 otherwise.

Fifteenth character returned:

Bit 7 ***Cutter comp added arc move*** (**CCAddedArc**): This bit is set to 1 if the presently calculated move is an automatically added arc move around an outside corner in 2D cutter radius compensation. It is 0 otherwise.

Bit 6 ***Cutter comp turn-off request*** (**CCOffReq**): This bit is set to 1 if the coordinate system has been told to turn off cutter compensation by a **CC0** command but has not yet computed the lead-out move that removes compensation. It is 0 otherwise.

Bit 5 ***{Reserved for future use}***

Bit 4 ***Cutter comp move type bit 1*** (**CCMoveType** bit 1): This bit is set to 1 if the coordinate system is executing a linear or circle move with cutter compensation active (**CCMoveType** = 2 for linear, = 3 for circle). It is 0 if the coordinate system is executing a dwell or zero-distance/out-of-plane move with cutter compensation active (**CCMoveType** = 0 for dwell, = 1 for zero-distance/out-of-plane), or if cutter compensation is not active. It is primarily for internal use.

Sixteenth character returned:

Bit 3 ***Cutter comp move type bit 0*** (**CCMoveType** bit 0): This bit is set to 1 if the coordinate system is executing a zero-distance/out-of-plane move or circle move with cutter compensation active (**CCMoveType** = 1 for zero-distance/out-of-plane, = 3 for circle). It is 0 if the coordinate system is executing a dwell or linear move with cutter compensation active (**CCMoveType** = 0 for dwell, = 2 for linear). It is primarily for internal use.

Bit 2 ***3D cutter comp active*** (**CC3Active**): This bit is set to 1 if three-dimensional cutter radius compensation is presently active. It is 0 otherwise.

Bit 1 ***Blending disabled for sharp corner*** (**SharpCornerStop**): This bit is set to 1 if blending has been disabled at a corner because the angle is sharper than that specified by **Coord[*x*].CornerBlendBp**. It is 0 otherwise.

Bit 0 ***Added dwell disabled*** (**AddedDwellDis**): This bit is set to 1 if blending has been disabled at a corner because the angle is sharper than that specified by **Coord[*x*].CornerBlendBp**, but there is no added dwell at the corner because the angle is not as sharp as that specified by **Coord[*x*].CornerDwellBp**. It is 0 otherwise.

Coord[x].ErrorStatus code values:

Value	Error Name	Description
0	NoError	Normal execution
1	SyncBufferError	Error in synchronous variable assignment buffer
2	BufferError	Error in motion program buffer
3	CCMoveTypeError	Illegal move mode or command while cutter compensation active (rapid, pvt, spline, lin-to-pvt, new normal, pmatch, pclr, pset, pload)
4	LinToPvtError	Error in automatic linear-to-PVT-mode conversion
5	CCLeadOutMoveError	Illegal cutter compensation lead-out move (circle mode or length less than cutter radius)
6	CCLeadInMoveError	Illegal cutter compensation lead-in move (circle mode or length less than cutter radius)
7	CCBufSizeError	Insufficient size for cutter compensation move buffer (not enough to find next in-plane move
8	PvtError	Illegally specified PVT-mode move
9	CCFeedRateError	Moves not specified by feedrate for cutter comp
10	CCDirChangeError	Full reversal while in cutter comp
11	CCNoSolutionError	No solution could be found for compensated move
12	CC3NdotTError	3D cutter compensation vector calculation error ("NdotT" value less than 1st entry in tool-tip table)
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
16	RunTimeError	Insufficient move calculation time
17	InPosTimeOutError	Exceeded specified time limit for in-position
18	LHNumMotorsError	Mismatch between # of motors used and # in lookahead buffer
19	TraceBufferError	Error in reversal through trace buffer
20	TimedUnderflowError	Too much timebase reversal without trace buffer
21	FeedRateError	Stopped from illegal feedrate (with **FProtect** = 1)
22–31	(*Reserved for future use*)	
32	SoftLimitError	Stopped from exceeding software position limit
33–63	(*Reserved for future use*)	
64	XYZRadiusError	Radius error in X/Y/Z-space circle move
65–127	(*Reserved for future use*)	
128	XXYYZZRadiusError	Radius error in XX/YY/ZZ-space circle move
129–255	(*Reserved for future use*)	

Global Status

For the global status, the following table shows how the reported status value represents the individual status bits. The two columns on the left explain how the status bits can be interpreted from the hexadecimal ASCII response to this command; the next column gives a basic description of the information in the status bit; the following two columns explain how the status bits can be interpreted from the numerical values of the data structure elements **Sys.Status**, and the last column shows the data structure element name for the bit.

The same information can be received in text form using the element names with the command **backup Sys.Status**. Each individual element name is reported back with its present value.

Char #	Hex Val	Description	Word #	Bit #	Element Name: Sys.
1	8	(*Reserved*)	0	31	
1	4	(*Reserved*)	0	30	
1	2	(*Reserved*)	0	29	
1	1	(*Reserved*)	0	28	
2	8	(*Reserved*)	0	27	
2	4	(*Reserved*)	0	26	
2	2	(*Reserved*)	0	25	
2	1	(*Reserved*)	0	24	
3	8	(*Reserved*)	0	23	
3	4	(*Reserved*)	0	22	
3	2	(*Reserved*)	0	21	
3	1	(*Reserved*)	0	20	
4	8	(*Reserved*)	0	19	
4	4	(*Reserved*)	0	18	
4	2	(*Reserved*)	0	17	
4	1	(*Reserved*)	0	16	
5	8	(*Reserved*)	0	15	
5	4	(*Reserved*)	0	14	
5	2	(*Reserved*)	0	13	
5	1	(*Reserved*)	0	12	
6	8	Insufficient flash memory size error	0	11	**FlashSizeErr**
6	4	Insufficient user buffer size error	0	10	**BufSizeErr**
6	2	"Abort all" condition	0	9	**AbortAll**
6	1	No system clocks found	0	8	**NoClocks**
7	8	Factory default configuration (by cmd or error)	0	7	**Default**
7	4	System file configuration error	0	6	**FileConfigErr**
7	2	Hardware change detected since save	0	5	**HWChangeErr**
7	1	Saved configuration load error	0	4	**ConfigLoadErr**
8	8	Project load error	0	3	**ProjectLoadErr**
8	4	Power-on/reset load fault (OR of bits 3–6)	0	2	**PwrOnFault**
8	2	Software watchdog fault bit 1	0	1	**WDTFault (bit 1)**
8	1	Software watchdog fault bit 0	0	0	**WDTFault (bit 0)**

First character returned:

Bits 31 .. 28 ***{Reserved for future use}***

Second character returned:

Bits 27 .. 24 ***{Reserved for future use}***

Third character returned:

Bits 23 .. 20 ***{Reserved for future use}***

Fourth character returned:

Bits 19 .. 16 ***{Reserved for future use}***

Fifth character returned:

Bits 15 .. 12 ***{Reserved for future use}***

Sixth character returned:

Bit 11 ***Insufficient flash size error*** **(FlashSizeErr)**: This bit is set to 1 on a save command if the built-in NAND flash memory does not have enough capacity to store the user project. It is 0 otherwise.

Bit 10 ***Insufficient user buffer size error*** (**BufSizeErr**): This bit is set to 1 on power-on/reset if the RAM capacity is not sufficient for the declared size of the project user buffers. It is 0 otherwise.

Bit 9 ***"Abort all" condition*** (**AbortAll**): This bit is set to 1 if the PMAC is stopping or stopped due to the setting of the specified "abort all" input. It is 0 otherwise.

Bit 8 ***No system clocks found*** (**NoClocks**): This bit is set to 1 if the processor does not find any phase or servo clocks at the end of the power-up/reset process. It is 0 if it does find these clock signals at this time.

Seventh character returned:

Bit 7 ***Factory default configuration set*** (**Default**): This bit is set to 1 if the processor puts the system into the factory default configuration during the power-up/reset process, either due to a re-initialization command or an error during the process. It is 0 otherwise.

Bit 6 ***System file configuration error*** (**FileConfigErr**): This bit is set to 1 if the processor detects an error in the configuration of the system files during the power-up/reset process. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time.

Bit 5 ***Hardware change detected*** (**HWChangeErr**): This bit is set to 1 if the processor detects a change in the system hardware configuration from the last save operation during the power-up/reset process. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time.

Bit 4 ***Saved configuration load error*** (**ConfigLoadErr**): This bit is set to 1 if the processor detects an error in the user-saved configuration during the power-up/reset process. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time.

Eighth character returned:

Bit 3 ***Project load error*** (**ProjectLoadErr**): This bit is set to 1 if the processor detects an error in the loading of the user project files into active memory during the power-up/reset process. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time.

Bit 2 ***Power-up/reset load fault*** (**PwrOnFault**): This bit is set to 1 if the processor detects an error during the power-up/reset process. This bit is the logical OR of bits 3 – 6, so will be set if any of those errors is detected. In this case, the system is put in the factory default configuration. It is 0 if no such error is found at this time.

Bit 1 ***Background soft watchdog timer fault*** (**WDTFault** bit 1): This bit is set to 1 if the processor detects a soft watchdog fault because too many background cycles have elapsed since the last real-time interrupt. In this case, all user programs are disabled, and all hardware interfaces are put in their reset state. This bit is 0 is no such error has been found.

Bit 0 ***Foreground soft watchdog timer fault*** (**WDTFault** bit 0): This bit is set to 1 if the processor detects a soft watchdog fault because too many real-time interrupt cycles have elapsed since the background cycle. In this case, all user programs are disabled, and all hardware interfaces are put in their reset state. This bit is 0 is no such error has been found.

Examples

```
?                          // Request global status word
000000A4                   // Factory default configuration due to hardware change

#1?                        // Request Motor 1 status words
0000380000000000           // "Closed loop", "amp enabled", "in-position" bits set

&1?                        // Request C.S. 1 status words
0000300000000000           // "Closed loop", "amp enabled" bits set
```

a

Function: Abort motion programs and moves

Scope: Coordinate-system specific

Syntax: **a**, **abort**

The **a** command causes Power PMAC to abort motion programs and moves for the specified coordinate system(s). If not immediately preceded by a coordinate system list, it will abort motion programs and moves for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will abort motion programs and moves for all coordinate systems in the list. If the coordinate system is executing a motion program or one of its subprograms, the program counter for the motion program is set to the beginning of the (top-level) motion program.

Note that specifying a list of multiple coordinate systems does not change the modally addressed coordinate system for subsequent coordinate-system-specific commands.

Motors that are aborted are decelerated to a controlled (closed-loop) stop at the rates or times set by data-structure elements **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**. Power PMAC breaks into the commanded move profile for all motors in the specified coordinate system(s) immediately upon receipt of this command and computes the individual deceleration profiles for each of these motors. Note that these deceleration profiles are independent, so there is not necessarily any coordination in the time or the path for the stopping.

Coord[*x*].AbortTimeBase allows the user to specify the minimum time base value for which the motor deceleration profiles will be executed at the present time base value. If the time base value is lower than this when the command is issued, the profiles will be executed at 100% time base.

The abort command is useful for emergency stops when a controlled stop is permitted (some applications will abort for the fastest possible stop, and then cut off power to the drives with a time-delay relay to satisfy regulatory requirements).

When motion program operation is already suspended by some other means (e.g. a “quit” or “hold”) command, the abort command is useful to fully stop program execution so another program can be started or new programs downloaded to the Power PMAC.

An abort command to stop a motion program should not be given if there is a desire to then resume execution of the motion program from the stopped point. In general, the abort command will not stop motion at a programmed point, and not even along a programmed path.

Note that the **a** command does not enable killed motors, as it did in PMAC and Turbo PMAC. The coordinate-system-specific **enable** command or the motor-specific **j/** command should be used instead. However, the **a** command does close the loop on open-loop enabled motors. If these motors have a non-zero velocity at the time of the command, they are decelerated to a stop at the rates or times set by data-structure elements **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**.

The short form of this command (**a**) is useful for typing in terminal mode. The long form (**abort**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**abort**) must be used. The short form (**a**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

The equivalent buffered direct program command is **abort**.

Examples

```
a                                   // Abort motors and programs in addressed C.S.
&2a                                 // Address C.S.2 and abort motors and programs there
&1,3,5a                             // Abort motors and programs in C.S. 1, 3, and 5
&*a                                 // Abort motors and programs in all C.S.
```

abort

Function: Abort motion programs and moves

Scope: Coordinate-system specific

Syntax: **a**, **abort**

The **abort** command is the "long-form" version of the **a** command, causing Power PMAC to abort motion programs and moves for the specified coordinate system(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

adisable

Function: Abort followed by delayed disable

Scope: Coordinate-system specific

Syntax: **adisable**

The **adisable** command causes Power PMAC to abort motion programs and moves for the specified coordinate system(s) automatically followed by a "delayed disable" of all the motors in the coordinate system(s). If not immediately preceded by a coordinate-system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list. If the coordinate system is executing a motion program or one of its subprograms, the program counter for the motion program is set to the beginning of the (top-level) motion program.

Note that specifying a list of multiple coordinate systems does not change the modally addressed coordinate system for subsequent coordinate-system-specific commands.

The initial action on an **adisable** command is equivalent to the action of an **abort** command, with all motors in the coordinate system(s) decelerated to a controlled (closed-loop) stop at the rates or times set by **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**, starting immediately.

However, as each motor reaches a commanded stop (**Motor[*x*].DesVelZero** = 1), it is then automatically disabled as if it were given a **dkill** command. If a brake output is specified for the motor (**Motor[*x*].pBrakeOut** > 0), the brake is immediately engaged at this instant, and the motor is killed (open-loop, zero command output, amplifier disabled) after a delay of **Motor[*x*].BrakeOnDelay** milliseconds.

The action taken on an **adisable** command is equivalent to that taken on the "global abort" input if **Coord[*x*].AbortAllMode** is set to 2.

The equivalent buffered direct program command is **adisable**.

Note

While the result of the **adisable** command is virtually the same as a "Category 1" safe stop under the IEC-61800-5-2 machine safety standard, that standard requires the actual removal of power from the motor or drive after a controlled stop. Use of this command under the standard would also require a time-delay relay to remove either bus power or gate-driver power after the controlled stop.

b[{constant}]

Function: Point program counter to specified motion program

Scope: Coordinate-system specific

Syntax: **b*[{constant}]*,begin*[{constant}]***

where:

- ***{constant}*** is a non-negative value whose integer part specifies the motion program number, and whose fractional part specifies the line jump label number within that program

The **b*[{constant}]*** causes the specified coordinate system(s) to set their program counter(s) to the beginning of the present (if no number given) or point in the specified motion program (if a number is given). If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list. The next **r** or **s** command will start motion program execution at this point.

Note that specifying a list of multiple coordinate systems does not change the modally addressed coordinate system for subsequent coordinate-system-specific commands.

The integer part of ***{constant}*** specifies the number of the motion program that the program counter will point to. "Fixed" motion programs (**prog**) can have numbers from 1 through 2^{31}-1. The rotary motion program buffer for the coordinate system is considered **prog 0** for the purposes of this command.

The fractional part of ***{constant}*** (if any) specifies the line-jump label number within the program that the program counter will point to. This fractional component is multiplied by 1,000,000 (and rounded down if necessary) to obtain the line-jump label number. So the line-jump label numbers that can be used range from **N0:** to **N999999:**.

If there is no explicitly declared **N0:** line-jump label, there is an implicit **N0**: at the beginning of a program. So without an explicitly declared **N0:**, a **b** command with only an integer value points the program counter to the beginning of the specified program number (this is the most common use of the command).

Execution of the **b** command clears the **Coord[*x*].ProgActive** flag, which permits the motion program buffer to be opened for overwriting or deletion.

The short form of this command (**b**) is useful for typing in terminal mode. The long form (**begin**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**begin**) must be used. The short form (**b**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

The equivalent buffered direct program command is **begin**.

Examples

```
b1                      // Set program counter of addressed C.S. to top of prog 1
&2b5r                   // Set C.S.2 program counter to top of prog 5 and run
&1b9.2                  // Set C.S.1 program counter to N200000: of prog 9
&1b9.200000             // Set C.S.1 program counter to N200000: of prog 9
&1b9.200001             // Set C.S.1 program counter to N200001: of prog 9
&1b9.000002             // Set C.S.1 program counter to N2: of prog 9
&3,4b0r                 // Set C.S.3 & 4 program counters to top of own rotary program
                        // buffers and run
```

backup

Function: Report values of saved setup or status elements

Scope: Global

Syntax: **backup *{structure}[.]***

where:

- ***{structure}*** is the name of a data structure whose saved setup elements are to be reported, including the following list:
  - **Motor[*x*].** (Motor data structure)
  - **Motor[*x*].Servo.** (Motor servo data structure)
  - **Motor[*x*].Status** (Motor status bits)

- **Coord[*x*].** (Coordinate system data structure)
- **Coord[*x*].Status** (Coordinate system status bits)
- **Gate*n*** (ASIC data structures)
- **EncTable[*n*].** (Encoder conversion table entry data structure)
- **CompTables.** (All compensation tables)
- **Sys.** (Global elements)
- **Sys.Status** (Global status bits)
- **Mdefs** (M-variable definitions)

The **backup** command causes Power PMAC to report the present active values of saved-setup or status-reporting data structure elements for the specified data structures or all data structures. It does not cause these values to be copied to non-volatile flash memory (although this command is used internally by the **save** command as part of the process for copying the values to flash memory).

The backup command without any structure name following causes a report of the values of all saved data structure elements. If a structure is named, the values for the saved setup elements for that structure will be reported. For an indexed structure (**Motor[*x*].**, **Coord[*x*].**, **EncTable[*n*].**) the index value must be specified with a constant, and the closing square bracket must be followed with a period character. If a second period character is used, values will be reported for all structures of the type starting with the specified index and continuing to the maximum permitted index.

When the structure name is that of a status data structure (global, motor, or coordinate system), the values reported are not that of saved setup elements, but of read-only status elements. (These commands are used by the IDE status windows.)

The element values are always reported starting with the element name and the equals sign (e.g. `Coord[1].FeedTime=1000`), so this text string can subsequently be used to restore the value to the Power PMAC. This is true regardless of the **echo** setting, which affects the reporting format for other query commands.

Examples

```
backup Motor[5].              // Report values for Motor 5 only
backup Motor[2]..             // Report values for Motors 2 and higher
backup Motor[13].Servo.       // Report values for Motor 13 servo algorithm
backup Coord[3].              // Report values for C.S. 3
backup Coord[0]..             // Report values for all C.S.
backup Gates                  // Report values for all ASICs
backup EncTable[8].           // Report values for ECT entry 8
backup EncTable[0]..          // Report values for all ECT entries
backup CompTables             // Report values for all compensation tables
backup Sys.                   // Report values of global elements
backup Motor[3].Status        // Report values of Motor 3 status elements
backup Coord[2].Status        // Report values of C.S. 2 status elements
```

begin*[{constant}]*

Function: Point program counter to specified motion program

Scope: Coordinate-system specific

Syntax: **b*[{constant}]***, **begin*[{constant}]***

where:

- ***{constant}*** is a non-negative value whose integer part specifies the motion program number, and whose fractional part specifies the line jump label number within that program

The **begin*[{constant}]*** is the "long-form" version of the **b*[{constant}]*** command, causing the specified coordinate system(s) to set their program counter(s) to the beginning of the present (if no number given) or point in the specified motion program (if a number is given). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

bpclear

Function: Erase breakpoint in specified program

Scope: Global

Syntax: **bpclear *{program name},{line offset}[.{cmd offset}]***

where:

- ***{program name}*** is the label for the program to take the breakpoint, consisting of either **prog** or **plc** and the program number

- ***{line offset}*** is a non-negative integer specifying the number of lines from the top of the program where the breakpoint will be (e.g. the 3rd program line has an offset of 2)

- ***{cmd offset}*** is a non-negative integer specifying the number of commands from the beginning of the program line where the breakpoint will be (e.g. the 4th command has an offset of 3)

The **bpclear** command causes Power PMAC to eliminate a break point in the specified script program at the specified point. In order for the command to do anything, the break point specified in the command by the program name, line offset, and command offset must match an existing break point. If it does not, no action will be taken. When clearing an existing break point, the data structure **Bp[*i*]** for that break point is cleared. If there are still break points with higher index values, the index values for those break points are decremented so there are no gaps in the indices for the remaining break points.

The program is specified by its name (e.g. **prog 7** or **plc 3**). The point in the program for the break is specified by the "line offset" and the "command offset". The line offset specifies the number of program lines from the top that the break will be. This value will be one less than the "line number". The optional "command offset" specifies the number of separate commands from the start of the line where the break will be placed. This can be useful when many separate commands are placed on a single program command. This value will be one less than the "command number".

bpclearall

Function: Erase all breakpoints

Scope: Global

Syntax: **bpclearall**

The **bpclearall** command causes Power PMAC to eliminate all break points that have been established. All **Bp[*i*]** data structures are cleared, so all element values in these structures report as "nan" (not a number).

bpset

Function: Define breakpoint in specified program

Scope: Global

Syntax: **bpset *{program name},{line offset}[.{cmd offset}]*: *{repetitions}***

where:

- ***{program name}*** is the label for the program to take the breakpoint, consisting of either **prog** or **plc** and the program number

- ***{line offset}*** is a non-negative integer specifying the number of lines from the top of the program where the breakpoint will be (e.g. the 3rd program line has an offset of 2)

- ***{cmd offset}*** is a non-negative integer specifying the number of commands from the beginning of the program line where the breakpoint will be (e.g. the 4th command has an offset of 3)

- ***{repetitions}*** is an integer specifying the number of times the command following the breakpoint will execute before the program will be paused at the breakpoint (e.g. 3 repetitions means the program will pause the 4th time the breakpoint is encountered)

The **bpset** command causes Power PMAC to establish a break point in the specified script program at the specified point after the specified number of repetitions. The command sets up a **Bp[*i*]** data structure to support the break point. The first break point defined uses **Bp[0]**; each subsequent command uses the next available index value.

The program is specified by its name (e.g. **prog 7** or **plc 3**).

The point in the program for the break is specified by the "line offset" and the "command offset". The line offset specifies the number of program lines from the top that the break will be. This value will be one less than the "line number". That is, to create a break on the *n*th line of the program, the line offset will be *n*-1 (e.g. to create a break on the 17th program line, set the line offset to 16).

The optional "command offset" specifies the number of separate commands from the start of the line where the break will be placed. This can be useful when many separate commands are placed on a single program command. This value will be one less than the "command number". That is, to create a break before the *n*th command of the program line, the command offset will be *n*-1 (e.g. to create a break before the 3rd command, set the command offset to 2). If no command offset is specified, a value of 0 will be used, so the break will occur before the first command in the line.

The "repetitions" value specifies the number of times the specified command at (i.e. immediately following) the break point will execute before program execution is paused at the break point. If it is set to 0, program execution will pause the first time the break point is encountered. If it is set to 99, the break point will be passed 99 times, and program execution will be paused the 100th time the break point is encountered.

When the **bpset** command is issued, the data structure element **Bp[*i*].Count** is set to the number of repetitions specified in the command. Then, each time the break point is encountered during program execution, if this element value is greater than 0, it is decremented by 1. If the break point is encountered while this element value is 0, program execution is paused at the break point. If the "repetitions" value is set to -1, **Bp[*i*].Count** will never hit 0 as it decrements, so the break point can be used as a cycle counter without the possibility of stopping the program.

Note that the original value of **Bp[*i*].Count** is not retained. If program execution is simply restarted, it will be paused again the next time the break point is encountered, no matter how many repetitions were initially specified. If a greater number of repetitions is desired before pausing again, the **bpset** command could be reissued for the same break point with a non-zero number of repetitions, or a value directly assigned to the **Bp[*i*].Count** element.

brickacver

Function: Report Brick AC amplifier firmware version

Scope: Global

Syntax: **brickacver**

The **brickacver** command causes Power PMAC processor to query the Brick AC amplifier firmware and report the version of the amplifier firmware that is present. The version is reported in the form ***{version}.{release}***. This value can only be reported if non-saved setup element **BrickAC.Monitor** is set to 1

The amplifer firmware version is distinct from the Power PMAC CPU firmware version, which can be queried with the **vers** command.

Note

bricklvver

Function: Report Brick LV amplifier firmware version

Scope: Global

Syntax: **bricklvver**

The **bricklvver** command causes Power PMAC processor to query the Brick LV amplifier firmware and report the version of the amplifier firmware that is present. The version is reported in the form ***{version}.{release}***. This value can only be reported if non-saved setup element **BrickLV.Monitor** is set to 1

The amplifer firmware version is distinct from the Power PMAC CPU firmware version, which can be queried with the **vers** command.

Note

buffer

Function: Report programs in Power PMAC buffer

Scope: Global

Syntax: **buffer**

The buffer command causes Power PMAC to report the program buffers that are in program memory. For each program buffer, it reports the name, the offset (in bytes) from the start of program buffer memory, and the size of the buffer in bytes.

Example

```
buffer
Prog1 offset 38655, size 436
Prog5 offset 254, size 151
Prog6 offset 405, size 2179
Prog10 offset 38580, size 75
Prog15 offset 41246, size 413
Prog25 offset 2950, size 39
Prog26 offset 2989, size 50
SubProg1000 offset 37205, size 1375
SubProg100000 offset 39091, size 150
SubProg100001 offset 39241, size 283
SubProg100002 offset 39524, size 166
SubProg100003 offset 39690, size 127
SubProg100004 offset 39817, size 172
SubProg100005 offset 39989, size 615
SubProg100006 offset 40604, size 642
Plc1 offset 2584, size 158
Plc2 offset 42367, size 396
Plc3 offset 42763, size 152
Plc5 offset 41659, size 384
Plc6 offset 42043, size 324
&1Inverse offset 37136, size 69
&1Forward offset 37023, size 113
```

bufioforceclear

Function: Clear all forcing words for buffered I/O

Scope: Global

Syntax: **bufioforceclear**

The **bufioforceclear** command causes Power PMAC to clear all of the "forcing" words **BufIo[*i*].ForceInOff**, **BufIo[*i*].ForceInOn**, **BufIo[*i*].ForceOutOff**, and **BufIo[*i*].ForceOutOn** for the buffered I/O functionality. It will do so for all index values ***i*** from 0 through 63, regardless of whether the buffering of I/O for that index is presently active or not.

This command is useful at the end of a debugging or maintenance session that employed forcing of inputs or outputs to ensure that no buffered inputs or outputs are left in a forced state that could prevent proper normal operation of the machine. It is possible to see whether any forcing bits in active buffered input or output registers are set or not in status element **Sys.ForceOr** and related elements.

All forcing words for the Power PMAC buffered I/O function are automatically set to 0 on a power-up/reset.

The **bufioforceclear** command is new in V2.2 firmware, released 3rd quarter 2016.

clear all buffers

Function: Erase contents of all rotary motion program buffers

Scope: Global

Syntax: **clear all buffers**

The **clear all buffers** command causes Power PMAC to erase the contents of all rotary motion program buffers, plus deleting all motion-program, PLC-program and kinematic-subroutine buffers.

All rotary motion program buffers must be closed when this command is given, and execution of all motion programs (rotary and fixed) must be fully stopped (as with an **a** abort command, or terminating at the end of the fixed motion program buffer), not just suspended (as with an **h** hold, **q** quit, or **s** step command, or execution having reached the end of the loaded rotary buffer).

Note that the structures of the rotary motion program buffers themselves remain, and new contents can be entered into them.

clear gather

Function: Erase contents of servo data gathering storage buffer

Scope: Global

Syntax: **clear gather**

The **clear gather** command causes Power PMAC to erase the contents of the servo-interrupt data gathering storage buffer, whether the standard buffer or a buffer set up in the user shared memory buffer with **Gather.UserBufStart**. It is generally *not* necessary to do this to prepare the buffer for the next data gathering event.

However, if the next data gathering event will use the storage buffer in "wrap-around" mode (**Gather.Enable** = 3) and it is possible that the gathering will stop before wrap-around, the **clear gather** command should be used before starting this, so that the uploading routine would not retrieve partial data from a previous gathering event.

The **clear gather** command can also be executed directly as a buffered program command in motion programs, PLC programs, or subprograms called from either type of program.

The **clear gather** command is new in V2.1 firmware, released 1st quarter 2016.

clear phase gather

Function: Erase contents of phase data gathering storage buffer

Scope: Global

Syntax: **clear phase gather**

The **clear phase gather** command causes Power PMAC to erase the contents of the phase-interrupt data gathering storage buffer, whether the standard buffer or a buffer set up in the user shared

memory buffer with **Gather.PhaseUserBufStart**. It is generally *not* necessary to do this to prepare the buffer for the next data gathering event.

However, if the next data gathering event will use the storage buffer in "wrap-around" mode (**Gather.PhaseEnable** = 3) and it is possible that the gathering will stop before wrap-around, the **clear phase gather** command should be used before starting this, so that the uploading routine would not retrieve partial data from a previous gathering event.

The **clear phase gather** command can also be executed directly as a buffered program command in motion programs, PLC programs, or subprograms called from either type of program.

The **clear phase gather** command is new in V2.1 firmware, released 1st quarter 2016.

clear plc *{constant}*

Function: Erase contents of specified PLC program buffer

Scope: Global

Syntax: **clear plc** ***{constant}***

where:

- ***{constant}*** is an integer in the range 0 to 31 specifying the number of the PLC program to be erased

The **clear plc** ***{constant}*** command causes Power PMAC to erase the contents of the specified script PLC program buffer. Its action is equivalent to that of the combination of commands **open plc** ***{constant}*** **close**.

clear prog *{constant}*

Function: Erase contents of specified motion program buffer

Scope: Global

Syntax: **clear prog** ***{constant}***

where:

- ***{constant}*** is a non-negative 32-bit integer specifying the number of the motion program to be erased

The **clear prog** ***{constant}*** command causes Power PMAC to erase the contents of the specified script motion program buffer. Its action is equivalent to that of the combination of commands **open prog** ***{constant}*** **close**.

clear rotary

Function: Erase contents of rotary motion program buffer

Scope: Coordinate-system specific

Syntax: **clear rotary**

The **clear rotary** command causes Power PMAC to erase the contents of the rotary motion program buffer for the addressed coordinate system.

The rotary buffer for the coordinate system must be closed when this command is given, and execution of the program must be fully stopped (as with an **a** abort command), not just suspended (as with an **h** hold, **q** quit, or **s** step command, or execution having reached the end of the buffer).

Note that the structure of the rotary buffer itself remains, and new contents can be entered into it. The program pointer is set to the beginning of the buffer, so execution can immediately resume on an **r** (run) or **s** (step) command.

close

Function: Close currently open program buffer

Scope: Global

Syntax: **close**

The **close** command causes Power PMAC to close the currently open program buffer of any type (**prog**, **plc**, **subprog**, **forward**, **inverse**) if that program buffer was opened in the same communications thread.

If the **close** command is issued immediately after the **open** command that prepares the buffer for entry and clears it (e.g. **open prog 73 close**), the program buffer itself is eliminated – it is not retained as an empty buffer.

If there is no program buffer open when this command is issued, the command will be accepted without error, but no action will be taken.

close all buffers

Function: Close currently open program buffer on any thread

Scope: Global

Syntax: **close all buffers**

The **close all buffers** command causes Power PMAC to close the currently open program buffer of any type (**prog**, **plc**, **subprog**, **forward**, **inverse**), no matter what communications thread opened that

buffer. This command should be used with great care, because it can interfere with the action of another thread. It is intended for recovery if another thread has crashed.

If there is no program buffer open, the command will be accepted without error, but no action will be taken.

cpu

Function: Report CPU type

Scope: Global

Syntax: **cpu**

The **cpu** command causes Power PMAC to report the type and number of the CPU used.

Examples

```
cpu                                  // Query CPU type
Power PC, 460EX                      // Power PMAC response
cpu                                  // Query CPU type
Power PC, APM86xxx                   // Power PMAC response
```

cpx

Function: Compile and execute following program command(s) as motion program

Scope: Coordinate-system specific

Syntax: **cpx** ***{program command(s)}***

The **cpx** command causes Power PMAC to compile (parse) the program command sequence that immediately follows it on the same line and execute it as a motion program. This provides a simple method for executing some basic buffered program commands without going through the full open/download/close/run cycle.

The buffered program commands that can be executed with the **cpx** command include:

- Single line **while** loops
- Single-line **do** .. **while** loops
- Single-line **if** conditional branches
- Variable modification assignments (e.g. **P1+=1**)
- Axis moves (any move mode)

Commands that are legal either as buffered program commands or on-line commands (e.g. ***{variable}={expression}***) do not require the **cpx** command preceding it.

For coordinate-system specific actions such as axis moves or modification of coordinate-system variables, the presently addressed coordinate system for the communications thread is affected.

Because this executes the commands as a motion program, a **pmatch** command is automatically and implicitly executed before any of the embedded commands, automatically calculating the starting axis positions.

Examples

```
cpx while (P100<40) P(P100)=0 P100++          // Execute WHILE loop
cpx if (Q10<0) Q300=72                        // Execute conditional branch
cpx Motor[4].ProgJogPos+=5000                 // Modify data structure element value
cpx linear X10 Y20 Z30 F5                     // Execute linear-mode move
```

CX

Function: Compile and execute following program command(s) as PLC

Scope: Coordinate-system specific

Syntax: ***cx {program command(s)}***

The **cx** command causes Power PMAC to compile (parse) the program command sequence that immediately follows it on the same line and execute it as a PLC program. This provides a simple method for executing some basic buffered program commands without going through the full open/download/close/run cycle.

The buffered program commands that can be executed with the **cx** command include:

- Single line **while** loops
- Single-line **do** .. **while** loops
- Single-line **if** conditional branches
- Variable modification assignments (e.g. **P1+=1**)
- Axis moves (forces the coordinate system into **rapid** mode)

Commands that are legal either as buffered program commands or on-line commands (e.g. ***{variable}={expression}***) do not require the **cx** command preceding it.

For coordinate-system specific actions such as axis moves or modification of coordinate-system variables, the presently addressed coordinate system for the communications thread is affected.

Because this executes the commands as a PLC program, there is no automatic implicit execution of a **pmatch** command to ensure that the axis starting positions for a move command are properly aligned with the present motor positions. If this is required, the command must be included explicitly on the line.

Examples

```
cx while (P100<40) P(P100)=0 P100++   // Execute WHILE loop
cx if (Q10<0) Q300=72                 // Execute conditional branch
cx Motor[4].ProgJogPos+=5000          // Modify data structure element value
cx pmatch X10 Y20 Z30                 // Execute RAPID-mode move
```

{data structure element}

Function: Report value of data structure element

Scope: Global

Syntax: ***{data structure element}***

where:

- ***{data structure element}*** is the name of the particular member of a pre-defined data structure

The ***{data structure element}*** command causes Power PMAC to report the present value of the specified data structure element. In the Power PMAC script language, the names of the elements are not case-sensitive. (Note that if you access data structure elements from a C program, the names are case sensitive.) With the exception of elements that are part of the local data structure for this communications thread, the full name of the element, including which structure it belongs to, must be specified.

Bit 0 (value 1) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the data structure will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                                   // Include variable names in query responses
Sys.ServoPeriod                         // Query value of system's servo update period
Sys.ServoPeriod=0.250                   // Power PMAC response of value (in msec)
Motor[3].JogSpeed                       // Query value of Motor 3's commanded jog speed
Motor[3].JogSpeed=38                    // Power PMAC response of value (in motor units/ms)
echo7                                   // Do not include variable names in query responses
Coord[2].MaxFeedrate                    // Query value of C.S. 3's maximum feedrate
25                                      // Power PMAC response of the value (in user units)
Plc[5].Ldata.Motor                      // Query value of PLC 5's addressed motor
7                                       // Power PMAC response of the value (Motor 7)
Ldata.Motor                             // Query value of thread's own addressed motor
4                                       // Power PMAC response of the value (Motor 4)
```

{data structure element}.a

Function: Report address of data structure element

Scope: Global

Syntax: ***{data structure element}*.a**

where:

- ***{data structure element}*** is the name of the particular member of a pre-defined data structure

The ***{data structure element}*.a** command causes Power PMAC to report the absolute address of the specified data structure element. This command is primarily for diagnostic purposes, as most uses

of data structure element addresses do not require the user to know the numerical value of the address. Absolute addresses of elements may not stay the same from one firmware release to another.

***{data structure element}*.a** can also be used as a variable value in mathematical expressions.

In the Power PMAC script language, the names of the elements are not case-sensitive. (Note that if you access data structure elements from a C program, the names are case sensitive.) With the exception of elements that are part of the data structure for this communications thread, the full name of the element, including which structure it belongs to, must be specified.

If the address of the element reports back as 0, the element is a "function" element, not a "register" element. That is, accessing the element value invokes the execution mathematical/logical function, rather than simply reading the value in a register. Function elements cannot be used in synchronous assignments, and they are not directly accessible from C programs.

Bit 0 (value 1) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the data structure will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                                   // Include variable names in query responses
Gate1[4].Chan[0].Dac[0].a               // Query value of DAC output address
Gate1[4].Chan[0].Dac[0].a=$f968008      // Power PMAC response of address

echo7                                   // Do not include variable names in query responses
Gate1[4].Chan[0].Dac[0].a               // Query value of DAC output address
$f968008                                // Power PMAC response of address
```

{data structure element}={expression}

Function: Set value of data structure element

Scope: Global

Syntax: ***{data structure element}={expression}***

where:

- ***{data structure element}*** is the name of the particular member of a pre-defined data structure
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified data structure element

The ***{data structure element}={expression}*** command causes Power PMAC to set the value of the specified data structure element to the value of the expression on the right side of the equals sign. In the Power PMAC script language, the names of the elements are not case-sensitive. (Note that if you access data structure elements from a C program, the names are case sensitive.) With the exception of elements that are part of the data structure for this communications thread, the full name of the element, including which structure it belongs to, must be specified.

Examples

```
Sys.ServoPeriod=0.250          // Set val of system's servo update period to 0.250 (ms)
Motor[3].JogSpeed=38           // Set value of Motor 3's commanded jog speed
Coord[2].MaxFeedrate=25        // Set value of C.S. 2's maximum feedrate
Plc[5].Ldata.Motor=7           // Set value of PLC 5's addressed motor
Ldata.Motor=4                  // Set value of thread's own addressed motor
```

d

Function: Report desired position value(s)

Scope: Motor or coordinate-system specific

Syntax: **d**

The **d** command causes Power PMAC to report the present value of the desired position(s) for the specified motor(s) or coordinate system(s). If not immediately preceded by a motor list or coordinate system list, it will report the value of the desired position for the presently addressed motor. If immediately preceded by a motor list, it will report the values of the desired positions for all motors in the list. If immediately preceded by a coordinate-system list, it will report the values of the desired positions for all active axes for all coordinate systems in the list.

Note that specifying a list of multiple motors or multiple coordinate systems does not change the modally addressed motor or coordinate system for subsequent motor-specific or coordinate-system-specific commands.

When reporting motor positions, the positions are given in the base motor units. The reference ("zero") motor position for this reporting is dependent on two control settings that determine how the values in "offset-mode" command position registers are used. Offset-mode command position registers permit the superposition of several position command sources.

Motor[*x*].CompDesPos, which is typically the output of a cam table or a cam-style compensation table, is always an offset-mode register. **Motor[*x*].ActiveMasterPos**, which is typically the result of position following (electronic gearing), is an offset-mode register if bit 1 (value 2) of **Motor[*x*].MasterCtrl** is set to 1.

If **Motor[*x*].PosReportMode** is set to its default value of 0, or in an older firmware version that does not have this new element, the values in these offset-mode registers are subtracted from the net desired position. Working from the source registers for **Motor[*x*]**, the reported position is calculated as:

+ **DesPos** // Net command position relative to power-on/reset location
- **CompDesPos** // Compensation table desired position command
- **ActiveMasterPos * MasterCtrl**OffsetBit // Master position when in offset mode
- **HomePos** // Motor zero position relative to power-on/reset location

If **Motor[*x*].PosReportMode** is set to 1, the values in these offset-mode registers are *not* subtracted from the net desired position. In this case, the reported position is calculated as:

+ **DesPos** // Net command position relative to power-on/reset location
- **HomePos** // Motor zero position relative to power-on/reset location

In both cases, the net desired position value **DesPos** is the sum of the trajectory desired position **Desired.Pos**, the table-based desired position **CompDesPos**, and the following desired position **ActiveMasterPos**.

When reporting axis positions, the positions are given in the scaled user (engineering) units, relative to the axis' programming origin (which is not necessarily the same as the zero position of the underlying motor). The reported value for each axis is preceded by the axis name (letter or double letter). Power PMAC leaves the values for each axis whose position is computed due to this command in local variable for the communications thread L(256+**Sys.MaxMotors**+*n*), where *n* is the "axis index" value 0 to 31 (0 for A, 1 for B, etc.).

Because this command is processed in background, if the values for multiple motors or axes are requested, it is possible that the reported values will not all be from the same servo cycle.

If a report is requested of a coordinate system with no motors assigned to axes in that C.S., Power PMAC will return the string "`No Motors`". If the set of axis definitions or the forward kinematics subroutine for the C.S. does not permit the proper calculation of axis data, Power PMAC will return the string "`No Solution`".

Examples

```
d                              // Query desired position of presently addressed motor
1001                           // Power PMAC response in counts

#1..4d                         // Query desired positions of Motors 1 - 4
982 -3462 27 86643             // Power PMAC response in counts

&1d                            // Query desired positions of axes in C.S. 1
X0.982 Y-3.462 Z0.027 C86.643 // Power PMAC response in user units
```

D{data}

Function: Report value of D-variable

Scope: Communications-thread specific

Syntax: **D*{data}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **D*{data}*** command causes Power PMAC to report the present value of the specified D-variable local to this communications thread. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 54); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Note that the D-variables for the communications thread do not correspond to the D-variables for any coordinate system, so this command cannot be used to query the value of D-variables for a coordinate

system. For that purpose, the values of the data structure elements **Coord[*x*].Ldata.D[*i*]** should be queried.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                                   // Include variable names in query responses
D1                                      // Query value of variable D1
D1=17.5                                 // Power PMAC response of the value of D1

D(P1)                                   // Query value of D-variable numbered by P1
D17=3                                   // Power PMAC response of the value of D17

D(P1+P17)                               // Query value of D-variable numbered by P1+P17
D20=-5.35                               // Power PMAC response of the value of D20

echo7                                   // Do not include variable names in query responses
D1                                      // Query value of variable D1
17.5                                    // Power PMAC response of the value of D1
```

D*{data}*=*{expression}*

Function: Assign value to D-variable

Scope: Communications-thread specific

Syntax: **D*{data}*=*{expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **D*{data}*=*{expression}*** command causes Power PMAC to set the specified D-variable local to this communications thread to the value of the expression on the right side of the equals sign. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 54); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, the command will be entered into that buffer for later execution.

Each communications thread has its own set of D-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of D-variables are independent of each other. Note that the D-variables for the communications thread do not correspond to

the D-variables for any coordinate system, so this command cannot be used to set the value of D-variables for a coordinate system.

Examples

```
D1=17.5                         // Set variable D1 to 17.5
D(P1)=3                         // Set D-variable numbered by P1 to 3
D(P1+P17)=5*sqrt(P100)          // Set D-var numbered by (P1+P17) to expression value
```

D{variable list}

Function: Report value(s) of D-variable(s) in list

Scope: Communications-thread specific

Syntax: **D*{variable list}*[:*{constant}*]**

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}*[,*{constant}*[,*{constant}*]]** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- the optional ***{constant}*** is a positive integer specifying the number of reported values per response line. If no value is specified here, only one value per response line is reported. This can only be used if the list is specified by a consecutive range of numbers.

The **D*{variable list}*** command causes Power PMAC to report the present value(s) of the D-variable(s) in the list local to this communications thread. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Each communications thread has its own set of D-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of D-variables are independent of each other. Note that the D-variables for the communications thread do not correspond to

the D-variables for any coordinate system, so this command cannot be used to query the value of D-variables for a coordinate system. For that purpose, the values of the data structure elements **Coord[*x*].Ldata.D[*i*]** should be queried.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
D1                          // Query value of variable D1 (set quantity 1, spacing 1)
D1=17.5                     // Power PMAC response of the value of D1

D10..12                     // Query value of D100, D101, D102
D10=-4                      // Power PMAC response of the value of D10
D11=3.14159                 // Power PMAC response of the value of D11
D12=0                       // Power PMAC response of the value of D12

D30,3                       // Query value of 3 D-vars, starting at D30 (spacing 1)
D30=-5.35                   // Power PMAC response of the value of D30
D31=7.12                    // Power PMAC response of the value of D31
D32=376452                  // Power PMAC response of the value of D32

D30,3,3                     // Query value of 3 D-vars, starting at D30, spacing 3
D30=-5.35                   // Power PMAC response of the value of D30
D33=99                      // Power PMAC response of the value of D33
D36=-0.002                  // Power PMAC response of the value of D36

D1..10:5                    // Query value of D1 - D10, 5 values per response line
D1=7,3,5,2,11               // Power PMAC response of the value of D1 - D5
D6=8,33,4,9,-2              // Power PMAC response of the value of D6 - D10
```

D*{variable list}={expression}*

Function: Set value(s) of D-variable(s) in list

Scope: Communications-thread specific

Syntax: **D*{variable list}={expression}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).

- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **D*{variable list}={expression}*** command causes Power PMAC to set the value(s) of the D-variable(s) in the list local to this communications thread to the value of the expression on the right side of the equals sign. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Each communications thread has its own set of D-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of D-variables are independent of each other. Note that the D-variables for the communications thread do not correspond to the D-variables for any coordinate system, so this command cannot be used to set the value of D-variables for a coordinate system.

Examples

```
D1=17.5                     // Set value of D1 (set quantity 1, spacing 1) to 17.5
D10..12=P1+5                // Set value of D10, D11, D12 to (P1+5)
D30,3=-7                    // Set value of 3 D-vars, starting at D30 (spacing 1) to -7
D30,3,3=sqrt(P25)           // Set value of 3 D-vars, starting at D30, spacing 3
                            // (D30, D33, D36) to the square root of P25
```

date

Function: Report firmware issuance date

Scope: Global

Syntax: **date**

The **date** command causes Power PMAC to report the date of issuance of the firmware it is using. The date is reported as:

- The first three letters of the English name of the month (e.g. `Apr`)
- The 2-digit day of the month
- The 4-digit year number (C.E.)

Example

```
date                            // Query date of issuance of firmware
Jan 30 2008                     // Power PMAC response
```

ddisable

Function: Delayed disable of all motors in coordinate system

Scope: Coordinate-system specific

Syntax: **ddisable**

The **ddisable** command causes Power PMAC to perform a delayed disable ("kill") of the control of all motors in the specified coordinate system(s). It is basically equivalent to the motor-specific **dkill** (delayed kill) command for all motor(s) in the coordinate system(s), but unlike the motor command, it will act on motors in a coordinate system that is executing a motion program, aborting the motion program in the process.

If not immediately preceded by a coordinate-system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list. The **&*ddisable** command disabled the motors in all coordinate systems on the Power PMAC. Note that the motor **dkill** command does a delayed kill for the specified motors.

"Killing" a motor causes the servo loop to be opened, the servo output value forced to zero (although output bias terms are still applied), and the amplifier-enable signal forced to the disable state.

If a motor's automatic brake-control function is enabled by setting **Motor[*x*].pBrakeOut** to a non-zero value (the address of the brake-output register), then the brake output will be commanded to engage immediately on the **ddisable** command. If, as well, the motor is in a closed-loop zero-velocity state, the actual killing of the motor will be delayed by **Motor[*x*].BrakeOnDelay** milliseconds, giving the brake system time to engage fully.

This delayed-disable command is intended for planned disabling of motors with brakes, so that the brakes have time to engage fully. Emergency disabling of motors should be done with the similar immediate **disable** command.

Closed-loop enabled control of a motor can be resumed with a **j/** command. Closed-loop enabled control for all the motors in a coordinate system can be resumed with an **enable** command. Open-loop enabled control of a motor can be resumed with the **out*{constant}*** command.

Note the difference in syntax between the on-line delayed-disable command (**&*x*ddisable**) and the buffered-program delayed-disable command (**ddisable*x***).

define lookahead

Function: Define a lookahead buffer

Scope: Coordinate-system specific

Syntax: **define lookahead *{constant}***

where:

- ***{constant}*** is a positive 32-bit integer representing the number of move segments that can be stored in the buffer for each motor in the coordinate system. It must be greater than or equal to 1024.

The **define lookahead** command causes Power PMAC to create a lookahead buffer for the addressed coordinate system, allocating sufficient memory to store information for the specified number of move segments. The lookahead buffer permits Power PMAC to check ahead robustly in the programmed trajectory for violations of position, velocity, and acceleration limits for all motors in the coordinate system, slowing the trajectory from programmed speeds as needed to ensure that no limits are violated.

At the time this command is issued, Power PMAC looks to see how many motors have been assigned to axes in this coordinate system, either through axis-definition statements or through kinematic subroutines, to determine how much memory must be allocated. Therefore, the coordinate system must be defined before the lookahead buffer is created for the coordinate system.

If there is already a defined lookahead buffer for this coordinate system when this command is issued, the command will be rejected with an error. If you wish to redefine the lookahead buffer for a coordinate system, either to change the number of segments it can store, or for a changed number of motors in the coordinate system, you must first eliminate the existing buffer with a **delete lookahead** or **delete all lookahead** command.

The constant value in the command specifies the number of move "segments" that can be stored in the lookahead buffer for each motor in the coordinate system. Each segment takes **Coord[*x*].SegMoveTime** milliseconds at the programmed speeds – the lookahead buffer may extend these segment times to prevent violations of velocity and acceleration and velocity limits. The status element **Coord[*x*[.LHSize** is set to this value. The buffer must contain at least 1024 segments.

The saved setup element **Coord[*x*].LHDistance** specifies how many segments the coordinate system will actually look ahead in operation. The buffer must be sized large enough to store all of the lookahead segments calculated, which means this constant must be at least as large as **Coord[*x*].LHDistance**. If backup (retrace) capability is desired, the buffer must be sized large enough to cover the desired lookahead distance plus the desired backup distance.

For robust acceleration limiting, the lookahead algorithm must be checking ahead at least as far as the worst-case stopping distance. Expressed as a time, this is the maximum velocity divided by the maximum acceleration. The maximum velocity can be from **Motor[*x*].MaxSpeed** or derived from **Coord[*x*].MaxFeedrate** (often lower). Taking the highest maximum velocity-to-acceleration ratio for all the motors in the coordinate system, the equation for the minimum number of segments to look ahead is:

$$AheadSegments = \frac{4}{3} * \frac{V_{\max}}{A_{\max}} * \frac{1}{2 * SegMoveTime}$$

The equation for the number of segments that must be stored to cover a desired backup distance is:

$$BackSegments = \frac{BackupDist\ (motor\_units)}{V_{\max}(motor\_units / msec) * SegMoveTime\ (msec / seg)}$$

The buffer must be sized for a number of segments at least as large as the sum of these two values. Each segment in the buffer requires (40 + 16*N) bytes in the lookahead buffer space, where N is the number of motors assigned to axes (positioning or spindle) in the coordinate system. For example, if there are 5 motors assigned to axes in the coordinate system, each segment requires 120 bytes, and a 2000-segment buffer requires 240,000 bytes of memory. The default lookahead buffer space is 16 MB (16,777,216 bytes); this can be changed using the IDE's project manager. This buffer space is used for the lookahead segment buffers, cutter compensation move block buffers, and target position buffers for all coordinate systems.

define rotary

Function: Define a rotary motion program buffer

Scope: Coordinate-system specific

Syntax: **define rotary** ***{constant} [,[{constant}]][,{constant}]***

where:

- the first ***{constant}*** is a positive 32-bit integer equal to 2048 or greater representing the number of bytes of memory to be reserved for this buffer
- the optional second ***{constant}*** is a positive 32-bit integer equal to 1024 or greater representing the number of bytes of memory to be reserved for the preliminary processing of a single program line. If no number is specified here, a value of 1024 is used. This value must not be greater than one-half of the buffer size specified by the first constant.
- the optional third ***{constant}*** is a positive integer that specifies the size of the local-variable "stack offset" used when this program calls a subroutine or subprogram; if no number is specified here, a value of 256 is used.

The **define rotary** command causes Power PMAC to create a rotary motion program buffer for the addressed coordinate system, allocating the specified amount of memory. Rotary buffers permit the downloading of motion program command lines during the execution of the program.

A single axis-move command with a constant destination or distance (e.g. **X10**) requires 9 bytes of memory. The buffer size specified in the first constant should be large enough to permit the downloading to get ahead of the calculated point in the program far enough so that the calculation will not "catch" up before the next downloading event from the host computer. It must be at least twice as large as the preliminary line-buffer size specified in the second constant, and must be at least 2048 in all cases.

The preliminary line-buffer size specified in the (optional) second constant determines the size of the temporary storage buffer that the processed form of each command line is stored in before being placed in the rotary buffer itself. If no value is specified here, 1024 bytes are reserved for this purpose. It is a very rare application that will require more than 1024 bytes to process a single command line.

The stack offset size specified in the (optional) third constant determines the maximum number of local variables that can be used by the rotary program itself that will not be reused by the local variables of any subprogram called by the rotary program. If no value is specified here, a stack offset of 256 is used.

The rotary program buffer uses memory in the overall (script) program memory space in RAM. This space is 16 megabytes by default, but the size can be changed using the IDE's project manager.

delete all lookahead

Function: Eliminate all lookahead buffers

Scope: Global

Syntax: **delete all lookahead**

The **delete all lookahead** command causes Power PMAC to eliminate the defined lookahead buffers for all coordinate systems, freeing the memory for other use.

delete all rotary

Function: Eliminate all rotary motion program buffers

Scope: Global

Syntax: **delete all rotary**

The **delete all rotary** command causes Power PMAC to eliminate the defined rotary motion program buffers for all coordinate systems, freeing the memory for other use.

All rotary motion program buffers must be closed when this command is given, and execution of all rotary motion programs must be fully stopped (as with an **a** abort command), not just suspended (as with an **h** hold, **q** quit, or **s** step command, or execution having reached the end of the buffer).

delete lookahead

Function: Eliminate lookahead buffer for specified coordinate system

Scope: Coordinate-system specific

Syntax: **delete lookahead**

The **delete lookahead** command causes Power PMAC to eliminate the defined lookahead buffer for the specified coordinate system(s), freeing its memory for other use.

If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

delete rotary

Function: Eliminate rotary motion program buffer for specified coordinate system

Scope: Coordinate-system specific

Syntax: **delete rotary**

The **delete rotary** command causes Power PMAC to eliminate the defined rotary motion program buffer for the specified coordinate system(s), freeing its memory for other use.

If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

The specified rotary motion program buffer must be closed when this command is given, and execution of the rotary motion program must be fully stopped (as with an **a** abort command), not just suspended (as with an **h** hold, **q** quit, or **s** step command, or execution having reached the end of the buffer).

disable

Function: Disable all motors in coordinate system

Scope: Coordinate-system specific

Syntax: **disable**

The **disable** command causes Power PMAC to "disable" control of all motors that have been defined in the specified coordinate system(s). It is basically equivalent to the **k** (kill) motor-specific command for all motors in the coordinate system(s), but unlike the motor command, it will act on motors in a coordinate system that is executing a motion program, aborting the motion program in the process.

If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

Disabling control for a motor consists of setting the amplifier-enable state to "false" (disabled) opening the position/velocity servo loop, and forcing a zero servo-output value. It can go to this state directly from either a closed-loop enabled state or an open-loop enabled state.

This immediate-disable command is intended for emergency disabling of motors. Planned disabling of motors with automatic brake control should be done with the similar delayed **ddisable** command, so that the brakes have time to engage fully. For motors without automatic brake control, it does not matter which command is used.

The **disable** command has no effect on a motor that is already in a disabled state.

disable bgcplc

Function: Disable specified background C PLC program(s)

Scope: Global

Syntax: **disable bgcplc** ***{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the background C PLC programs to be disabled.

The **disable bgcplc** command causes Power PMAC to stop the execution of the specified background C PLC program(s) by inhibiting the start of subsequent scans. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. with a **sleep** command).

If a C PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other existing C PLC programs specified in the command.

Examples

```
disable bgcplc 0              // Disable execution of CPLC 0
disable bgcplc 2,4,6          // Disable execution of CPLCs 2, 4, and 6
disable bgcplc 7..10          // Disable execution of CPLCs 7, 8, 9, and 10
disable bgcplc 11,13..16,20   // Disable execution of CPLCs 11, 13, 14, 15, 16, and 20
```

disable plc

Function: Disable specified Script PLC program(s)

Scope: Global

Syntax: **disable plc *{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the Script PLC programs to be disabled.

The **disable plc** command causes Power PMAC to stop the execution of the specified Script PLC program(s) by inhibiting the start of subsequent scans. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. the program was stuck inside a **while** loop).

The similar **pause plc** command can be used to stop PLC program execution in a way that permits re-starting at other than the beginning of the program if execution was halted there.

If a PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other existing PLC programs specified in the command.

Examples

```
disable plc 1                 // Disable execution of PLC 1
disable plc 2,4,6             // Disable execution of PLCs 2, 4, and 6
disable plc 7..10             // Disable execution of PLCs 7, 8, 9, and 10
disable plc 11,13..16,20      // Disable execution of PLCs 11, 13, 14, 15, 16, and 20
```

disable rticplc

Function: Disable foreground C PLC program

Scope: Global

Syntax: **disable rticplc**

The **disable rticplc** command causes Power PMAC to stop the execution of the foreground C PLC program that executes under the real-time interrupt by inhibiting the start of subsequent scans. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. with a **sleep** command).

If there is no foreground C PLC program present in the Power PMAC, no error will be reported.

dkill

Function: Delayed motor kill

Scope: Motor specific

Syntax: **dkill**

The **dkill** command causes Power PMAC to perform a delayed "kill" the servo outputs of the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list. The **#*dkill** command kills all motors on the Power PMAC. Note that the coordinate-system **ddisable** command does a delayed kill for all motors in a coordinate system.

"Killing" a motor causes the servo loop to be opened, the servo output value forced to zero (although output bias terms are still applied), and the amplifier-enable signal forced to the disable state. Note that Power PMAC automatically kills a motor on an amplifier fault or a fatal following error fault.

If the motor's automatic brake-control function is enabled by setting **Motor[*x*].pBrakeOut** to a non-zero value (the address of the brake-output register), then the brake output will be commanded to engage immediately on the **dkill** command. If, as well, the motor is in a closed-loop zero-velocity state, the actual killing of the motor will be delayed by **Motor[*x*].BrakeOnDelay** milliseconds, giving the brake system time to engage fully.

This delayed-kill command is intended for planned killing of motors with brakes, so that the brakes have time to engage fully. Emergency killing of motors should be done with the similar immediate **k** command.

Power PMAC will reject a motor-specific **dkill** command with an error if the motor is in a coordinate system that is currently running a motion program; program execution must be stopped before these commands can be given. However (starting in V1.4 released July 2011), the **#*dkill** command will act on all motors, even those in coordinate systems executing motion programs, aborting any motion programs in the process.

Closed-loop enabled control of a motor can be resumed with a **j/** command. Closed-loop enabled control for all the motors in a coordinate system can be resumed with an **enable** command. Open-loop enabled control of a motor can be resumed with the **out*{constant}*** command.

Note the difference in syntax between the on-line delayed-disable command (**#*x*dkill**) and the buffered-program delayed-kill command (**dkill*x***).

ecat alias

Function: Set an alias address for an EtherCAT slave device

Syntax: **ecat alias *{constant} {constant} [{constant}]***

where:

- the first ***{constant}*** specifies the absolute position value of the slave device that is to be given an alias address
- the second ***{constant}*** specifies the alias address number to be given to this slave device
- the last ***{constant}*** is optional and specifies the EtherCAT network index (0 – 7), which is also the EtherCAT Master Number, on which this slave device is located. If no value is specified, the device is assumed to be on **ECAT[0]**.

The **ecat alias** command causes the Power PMAC to assign an alias address to a slave device on an EtherCAT network. Setting of alias addresses for slave devices on your network is strongly recommended if you may be changing the network wiring, as it makes the software addressing of the devices independent of their physical location on the network, permitting the user to continue to communicate with a drive that has been already configured even if the network wiring is reorganized.

The device must support software setting of the alias address for this command to have any effect. Some EtherCAT slave devices support only hardware setting of the alias address through on-board switches; this command will have no effect on those devices.

Power PMAC cannot execute this command if **ECAT[*i*].Enable** is 1, enabling cyclic communications for the network.

After setting the alias value, Power PMAC automatically executes an **ecat slaves** command for the network, displaying the resulting configuration of the slaves on the network.

Example

```
ecat alias 7 16
0 VID=$00000002 PC=$044D2C52 0:0 PREOP + EK1101 EtherCAT-Koppler (2A E-Bus)
1 VID=$00000002 PC=$10103052 0:1 PREOP + EL4112-0010 2K. Ana. Ausgang +/-1t
2 VID=$00000002 PC=$10243052 0:2 PREOP + EL4132 2Ch. Ana. Ausgang +/-10V, t
3 VID=$00000002 PC=$101A3052 0:3 PREOP + EL4122 2K. Ana. Ausgang 4-20mA, 1t
4 VID=$00000002 PC=$07E83052 0:4 PREOP + EL2024 4K. Dig. Ausgang 24V, 2A
5 VID=$00000002 PC=$0C503052 0:5 PREOP + EL3152 2K. Ana. Eingang 4-20mA
6 VID=$00000002 PC=$05F12C52 0:6 PREOP + EK1521 1-Port EtherCAT-Abzweig (F)
7 VID=$00000002 PC=$05DD2C52 16:0 PREOP + EK1501 EtherCAT-Koppler (2A E-Bus)
8 VID=$00000002 PC=$138A3052 16:1 PREOP + EL5002 2K. SSI Encoder
```

```
9 VID=$00000002 PC=$04562C52 16:2 PREOP + EK1110 EtherCAT-Verlungerung
```

ecat assign

Function: Assign slave device(s) to **ECAT[*i*].Slave[*j*]** structure(s)

Syntax: **`ecat assign [{constant} [{constant}]]`**

The first optional ***{constant}*** specifies the absolute position value of the slave device that is to be assigned to a structure. If no value is specified, all slaves on the specified network are reported for **ECAT[0]**. If a value of -1 is specified, all slaves are assigned for all 8 possible EtherCAT networks.

The second optional ***{constant}*** specifies the EtherCAT network index (0 – 7) (i.e. the EtherCAT Master Number) for which the slave devices are to be assigned. If no value is specified, slaves are assigned for **ECAT[0]**.

The **`ecat assign`** command causes the Power PMAC to assign the specified slave device(s) to the next available **ECAT[*i*].Slave[*j*]** structure(s) for their network(s). If a non-negative constant value follows the command, it specifies the absolute position of the single device on the network that is to be assigned to the structure. If a second constant is specified, this specifies the index (0 – 7) of the network for which this assignment is done; otherwise the index is assumed to be 0.

If no constant values follow the command, all the slave devices on EtherCAT network 0 are assigned to **ECAT[0].Slave[*j*]** structures, reassigning any that are already assigned. If a value of -1 follows the command, all the slave devices on all EtherCAT networks are assigned to **ECAT[*i*].Slave[*j*]** structures, reassigning any that are already assigned.

For each device assigned to an **ECAT[*i*].Slave[*j*]** structure, the following element values are automatically set using information obtained from the device, as through the **`ecat slaves`** command:

- **ECAT[*i*].Slave[*j*].VendorID**
- **ECAT[*i*].Slave[*j*].ProductCode**
- **ECAT[*i*].Slave[*j*].Enable**
- **ECAT[*i*].Slave[*j*].Position**
- **ECAT[*i*].Slave[*j*].Alias**

When a single device is assigned, the index value ***j*** of the **Slave[*j*]** substructure is equivalent to the value of **ECAT[*i*].SlaveCount** when the command is issued. After the structure element values are set, the value of **ECAT[*i*].SlaveCount** is automatically incremented by 1.

When all of the slave devices on a network are assigned, the index value ***j*** of the **Slave[*j*]** sub-structures used range from 0 to the number of slave devices minus one, and the value of **Ecat[*i*].SlaveCount** is automatically set to the number of slave devices.

Typically, the **`ecat assign`** command without parameters is used in the initial configuration of a network so that all devices are assigned to slave substructures. The **`ecat assign`** command with parameter(s) is usually used when a network is already configured, but a new device must be added to the network. This command cannot be executed if EtherCAT cyclic operations are occurring (i.e. **ECAT[*i*].Enable** = 1).

ecat config

Function: Automatically configure **Ecat[*i*].Slave[*j*]** structures and sub-structures

Syntax: **`ecat config`**

The **`ecat config`** command causes the Power PMAC to assign all the slave devices to **Ecat[*i*].Slave[*j*]** structures for their networks, reassigning any that are already assigned

For each device assigned to an **Ecat[*i*].Slave[*j*]** structure, the following element values are automatically set using information obtained from the device, just like when using the **`ecat slaves`** command:

- **ECAT[*i*].Slave[*j*].VendorID**
- **ECAT[*i*].Slave[*j*].ProductCode**
- **ECAT[*i*].Slave[*j*].Enable**
- **ECAT[*i*].Slave[*j*].Position**
- **ECAT[*i*].Slave[*j*].Alias**

In addition, this command causes Power PMAC to assign the following substructures:

- **ECAT[*i*].Slave[*j*].PDO[*k*].**
- **ECAT[*i*].Slave[*j*].PDOMapping[*k*].**
- **ECAT[*i*].Slave[*j*].SyncManager[*k*].**

The index value ***j*** of the **Slave[*j*]** substructures used range from 0 to the number of slave devices minus one, and the value of **ECAT[*i*].SlaveCount** is automatically set to the number of slave devices.

ecat slaves

Function: Displays the EtherCAT network's slave devices

Syntax: **ecat slaves** ***[{constant}]***

where:

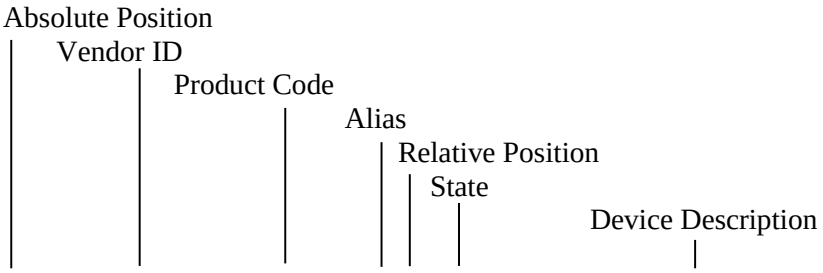
- the optional ***{constant}*** specifies the EtherCAT network index (0 – 7), which also corresponds to the EtherCAT Master Number, for which slave devices are to be reported. If no value is specified, slaves are reported for **ECAT[0]**. If a value of -1 is specified, slaves are reported for all 8 possible EtherCAT networks.

The **ecat slaves** command causes the Power PMAC to report information on the slave devices that are presently attached to the specified EtherCAT network(s). The information is reported in text form, with one line per slave. Each line contains the following information:

- **Absolute position**: The physical location order of the slave on the network is reported as an integer.
- **Vendor ID**: The ID number for the manufacturer of the slave device is reported in the form `VID=$xxxxxxxx`.
- **Product code**: The code number for the product used as the slave device is reported in the form `PC=$xxxxxxxx`.
- **Alias**: The alias number for the slave device is reported as an integer value.
- **Relative position**: The relative position value of the slave device is reported as an integer following a colon character.
- **State**: The operational state of the slave device is reported as a short text word.
- **Device description**: The description of the product used as the slave device is reported in text form.

Example

```
ecat slaves 0
```




```
0 VID=$00000002 PC=$044D2C52 0:0 PREOP + EK1101 EtherCAT-Koppler (2A E-Bus)
1 VID=$00000002 PC=$10103052 0:1 PREOP + EL4112-0010 2K. Ana. Ausgang +/-1t
2 VID=$00000002 PC=$10243052 0:2 PREOP + EL4132 2Ch. Ana. Ausgang +/-10V, t
3 VID=$00000002 PC=$101A3052 0:3 PREOP + EL4122 2K. Ana. Ausgang 4-20mA, 1t
4 VID=$00000002 PC=$07E83052 0:4 PREOP + EL2024 4K. Dig. Ausgang 24V, 2A
5 VID=$00000002 PC=$0C503052 0:5 PREOP + EL3152 2K. Ana. Eingang 4-20mA
6 VID=$00000002 PC=$05F12C52 0:6 PREOP + EK1521 1-Port EtherCAT-Abzweig (F)
7 VID=$00000002 PC=$05DD2C52 1:0 PREOP + EK1501 EtherCAT-Koppler (2A E-Bus)
8 VID=$00000002 PC=$138A3052 1:1 PREOP + EL5002 2K. SSI Encoder
9 VID=$00000002 PC=$04562C52 1:2 PREOP + EK1110 EtherCAT-Verlungerung
```

echo

Function: Report command echo mode

Scope: Communications-thread specific

Syntax: **echo**

The **echo** command causes Power PMAC to return the present value of the "echo" control parameter for the communications thread. This value determines how the response to certain query commands is formatted, mainly whether the name of the queried variable is included in the response, or just its value. To see what the returned value means, refer to the description of the **echo*{constant}*** command, below.

echo*{constant}*

Function: Set command echo mode

Scope: Communications-thread specific

Syntax: **echo*{constant}***

where:

- ***{constant}*** is an integer in the range of 0 to 127, whose 7 separate bits (0 – 6) control what is "echoed back" in the response to certain query commands

The **echo*{constant}*** command causes Power PMAC to set the value of the "echo" control parameter for the communications thread to the specified constant value. This value determines how the response to certain query commands is formatted, mainly whether the name of the queried variable is included in the response, or just its value. Note that each communications thread has its own independent echo control parameter.

The echo control parameter consists of five independent control bits.

Bit 0, with a value of 1, controls whether data-structure element names are included in the response when the value of an element is queried. If the bit is set to 0, the name is included (e.g. the response to the query **Motor[1].JogSpeed** would be something like `Motor[1].JogSpeed=200`). If the bit is set to 1, the name is not included (e.g. the response to the query **Motor[1].JogSpeed** would be something like `200`).

Bit 1, with a value of 2, controls whether numbered variable names are included in the response when the value of the variable is queried. If the bit is set to 0, the name is included (e.g. the response to the query **P1** would be something like `P1=10`). If the bit is set to 1, the name is not included (e.g. the response to the query **P1** would be something like `10`).

Bit 2, with a value of 4, controls whether I-variable and M-variable names are included in the response when the *definition* of the I-variable or M-variable is queried. If the bit is set to 0, the name is included (e.g. the response to the query **M1->** would be something like `M1->Sys.ServoCount`). If the bit is set to 1, the name is not included (e.g. the response to the query **M1** would be something like `Sys.ServoCount`).

Bit 3, with a value of 8, controls whether data-structure elements whose values are addresses or "bit-wise" logical values report back these values in hexadecimal (base 16) or decimal (base 10). If the bit is set to 0, they are reported back in hexadecimal (e.g. the response to the query **Gate1[4].Chan[0].Dac[0].a** would be something like `$d5700008`). If the bit is set to 1, they are reported back in decimal (e.g. the response to the query **Gate1[4].Chan[0].Dac[0].a** would be something like `3580887048`).

Bit 4, with a value of 16, controls whether the reported text descriptions of status bits provided in response to a **backup *xxx*.status** command are enumerations (e.g. `False` or `True`) or integer values (e.g. `0` or `1`). If the bit is set to 0, they are integer values. If the bit is set to 1, they are enumerations. (The Status windows in the IDE automatically set this bit to 1 to receive enumerations.)

Bit 5, with a value of 32, controls whether inactive motors (**Motor[*x*].ServoCtrl** = 0) and coordinate systems (no motors assigned) are fully backed up with the **backup** and **save** commands. If the bit is set to 0, these motors and coordinate systems are not fully backed up, saving storage space and time. If the bit is 1, they are fully backed up, facilitating better comparisons between revisions.

Bit 6, with a value of 64, controls whether multiple partial-word saved setup elements that are part of a single full-word element are backed up as several partial-word elements or one full-word element. If the bit is set to 0, the full-word element is backed up, which is faster and more compact. If the bit is 1, the partial-word elements are backed up, which is easier to understand from reading the backup file. This control bit is new in V2.0 firmware (released 1st quarter 2015); the default setting of 0 for the bit provides operation compatible with older revisions.

The power-on default value of the echo control parameter is 0, so all names are echoed back in the query responses. (The Watch window in the IDE automatically sets this to 7.) The present value of the echo control parameter can be found by using the **echo** query command.

enable

Function: Enable all motors in coordinate system

Scope: Coordinate-system specific

Syntax: **enable**

The **enable** command causes Power PMAC to "enable" closed-loop servo control of all presently disabled ("killed") motors that have been defined in the specified coordinate system(s). It does not affect already-enabled motors in the coordinate system, whether open-loop or closed-loop.

If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

Enabling servo control for a motor consists of setting the amplifier-enable state to "true" (enabled) and closing the position/velocity servo loop. From either a "killed" state or an open-loop enabled state, the motor will end up in an enabled, closed-loop, zero-commanded-velocity state.

enable bgcplc

Function: Enable specified background C PLC program(s)

Scope: Global

Syntax: **enable bgcplc** ***{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the background C PLC programs to be disabled.

The **enable bgcplc** command causes Power PMAC to permit the execution of the specified background C PLC program(s) at their normal priority and timing. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. with a **sleep** command).

If a C PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other existing C PLC programs specified in the command.

Examples

```
enable bgcplc 0                 // Enable execution of CPLC 0
enable bgcplc 2,4,6             // Enable execution of CPLCs 2, 4, and 6
enable bgcplc 7..10             // Enable execution of CPLCs 7, 8, 9, and 10
enable bgcplc 11,13..16,20      // Enable execution of CPLCs 11, 13, 14, 15, 16, and 20
```

enable plc

Function: Enable specified Script PLC program(s)

Scope: Global

Syntax: **enable plc** ***{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the Script PLC programs to be enabled.

The **enable plc** command causes Power PMAC to permit the execution of the specified Script PLC program(s) at their normal priority and timing. Execution will start at the beginning of the program, even if execution was halted in the middle of the program (e.g. with a **pause plc** command), or (if already enabled) the most recent scan of execution did not stop at the end of the program.

The similar **resume plc** command can be used to re-start PLC program execution at other than the beginning of the program if execution was halted there.

If a PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other existing PLC programs specified in the command.

Examples

```
enable plc 1                    // Enable execution of PLC 1
enable plc 2,4,6                // Enable execution of PLCs 2, 4, and 6
enable plc 7..10                // Enable execution of PLCs 7, 8, 9, and 10
enable plc 11,13..16,20         // Enable execution of PLCs 11, 13, 14, 15, 16, and 20
```

enable rticplc

Function: Enable foreground C PLC program

Scope: Global

Syntax: **enable rticplc**

The **enable rticplc** command causes Power PMAC to permit the execution of the foreground C PLC program that executes under the real-time interrupt at their normal priority and timing. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. with a **sleep** command).

If there is no foreground C PLC program present in the Power PMAC, no error will be reported.

f

Function: Report following error value(s)

Scope: Motor or Coordinate-system specific

Syntax: **f**

The **f** command causes Power PMAC to report the present value of the (position) "following" error(s) for the specified motor(s) or coordinate system(s). This value is the instantaneous difference between the desired and actual positions. If not immediately preceded by a motor list or coordinate system list, it will report the value of the following error for the presently addressed motor. If immediately preceded by a motor list, it will report the values of the following errors for all motors in the list. If immediately preceded by a coordinate-system list, it will report the values of the following errors for all active axes for all coordinate systems in the list.

Note that specifying a list of multiple motors or multiple coordinate systems does not change the modally addressed motor or coordinate system for subsequent motor-specific or coordinate-system-specific commands.

When reporting motor following errors, the errors are given in the defined motor units. When reporting axis following errors, the errors are given in the scaled axis units. The reported value for each axis is preceded by the axis name (letter or double letter).

Because this command is processed in background, the user should be aware of a couple of possible issues. If the values for multiple motors or axes are requested, it is possible that the reported values will not all be from the same servo cycle. In the case of axis values, because multiple reads of the source registers are required for each axis value, there is a slight possibility that a given value will be incorrect if a new servo update starts in the middle of the process.

If a report is requested of a coordinate system with no motors assigned to axes in that C.S., Power PMAC will return the string "`No Motors`". If the set of axis definitions or the forward kinematics subroutine for the C.S. does not permit the proper calculation of axis data, Power PMAC will return the string "`No Solution`".

Examples

```
f                              // Query following error of presently addressed motor
-23                            // Power PMAC response in motor units

#1..4f                         // Query following errors of Motors 1 - 4
33 -22 4 173                   // Power PMAC response in motor units

&1f                            // Query following errors of axes in C.S. 1
X0.033 Y-0.022 Z0.004 C0.173   // Power PMAC response in user axis units
```

fload

Function: Reload result of last "fast save" command from non-volatile memory

Scope: Global

Syntax: **fload**

The **fload** command causes the Power PMAC to execute the commands in the file "pp_custom_save.cfg" " in the non-volatile flash-memory folder "/opt/ppmac/usrflash/Project/Configuration". This file was created by the most recent **fsave** command, and it typically contains commands to set the values of certain user-selected variables. Refer to the description of the **fsave** command to see how this file is created.

The main use of the **fload** command is to resstore the last saved "machine state" information to facilitate restarting after an unexpected shutdown.

The **fload** command is new in V2.0 firmware, introduced 1st quarter 2015.

This functionality can be implemented in the C programming environment using the FastLoad() function.

free

Function: Report buffer free memory

Scope: Global

Syntax: **free**

The **free** command causes Power PMAC to report the number of bytes of memory still available (unused) in several user memory buffers in RAM. The related **size** command returns the total number of bytes of memory, used or unused, for each of these memory buffers.

Example

```
free                                          // Query free buffer memory
Program Buffer = 16740722                     // Power PMAC response
User Buffer = 1048576
Table Buffer = 1002305
Lookahead Buffer = 16777216
```

fsave

Function: Perform customized "fast save" to non-volatile memory

Scope: Global

Syntax: **fsave**

The **fsave** command permits the user to perform a customized "fast save" of the values of specified parameters to non-volatile memory. The command causes Power PMAC to execute the query commands contained in the user-created template file "pp_custom_save.tpl" in the active-memory folder "/var/ftp/ppmac/usrflash/Project/Configuration". (This file should be created in the "Configuration" folder of the IDE project manager and it is then automatically copied to Power PMAC as part of the project download action.)

Power PMAC's responses to this command are automatically placed in the file "pp_custom_save.cfg" in the non-volatile flash-memory folder "/opt/ppmac/usrflash/Project/Configuration". They are also placed in a file of the same name in the active memory folder "/var/ftp/ppmac/usrflash/Project/Configuration", which can be uploaded to the host computer.

Power PMAC's responses to queries of variable values or definitions are automatically "long form" (e.g. P1=5 instead of just 5), regardless of the present echo mode setting. This permits the responses to be used as on-line commands to set variable values or definitions on a subsequent **fload** command.

The main use of the **fsave** command is to store the present "machine state" information to facilitate restarting after an unexpected shutdown.

The **fsave** command is new in V2.0 firmware, introduced 1st quarter 2015.

This functionality can be implemented in the C programming environment using the FastSave() function.

Example

If the file "pp_custom_save.tpl" contains the following Power PMAC commands:

```
P100..102
Sys.Idata[500],4
Tdata[1].Diag[6],3
```

The automatically created file "pp_custom_save.cfg" would look something like:

```
P100=17
```

```
P101=-36
P102=1002
Sys.Idata[500]=0,25,-376,0
Tdata[1].Diag[6]=2,2,1
```

g

Function: Report axis distances-to-go of presently executing move

Scope: Coordinate-system specific

Syntax: **g**

The **g** command causes Power PMAC to report the "distances to go" for axes in the presently executing move. If not immediately preceded by a coordinate system list, it will report the distances to go for the presently addressed coordinate system. If immediately preceded by a coordinate system list, it will report the distances to go for all coordinate systems in the list.

"Distance to go" for an axis is computed as the axis' target position for the presently executing move minus the immediate commanded position for the axis. For the X, Y, and Z axes, if cutter radius compensation is enabled, the target position value offset by the cutter compensation is used in this calculation. The present desired position for the axis is calculated from the corresponding motor desired position(s), processed through the coordinate system definition (either by inverting axis definition statements or using the forward kinematic subroutine), and any matrix transformations in force. The difference is reported as a signed quantity.

Target position buffering must be enabled by setting saved setup element **Coord[*x*].TPSize** greater than 0, and to a value sufficient to store positions for all of the moves between move calculation time and move execution time. The distances to go will be reported for all defined axes in the coordinate system.

The distances to go will be reported in text form, with the axis letter name followed by the numerical value. The distances will be reported as programmed, with any axis transformations in force at the time the move was calculated. If saved setup element **Coord[*x*].Ndisplay** is set to 1, the present value of status element **Coord[*x*].Nsync**, which is automatically set to the value of the most recent synchronizing line label at the beginning of execution of the move in the program following the label, is reported at the beginning of the response, following the letter "N". This makes it easy to identify the particular move when these labels are used.

If **Coord[*x*].TPSize** is set to 0 when this command is issued, Power PMAC will return the error message `No targets defined`. If **Coord[*x*].TPSize** is greater than 0, but not large enough to buffer all moves between calculation and execution, erroneous values may be reported.

The on-line **g** command performs the same calculations as the buffered program **dtogread** command. In addition to returning a text string, it puts the axis values into local D-variables as the buffered program command does.

Example

```
Coord[1].Ndisplay = 0                // Do not report N label
&1g                                  // Request distance to go
A5.72 C0 X-0.007 Y3.227 Z3.753       // Power PMAC responds w/o N label
```

```
Coord[1].Ndisplay = 1                  // Report N label
&1g                                    // Request distance to go
N430 A5.72 C0 X-0.007 Y3.227 Z3.753    // Power PMAC responds w/ N label
```

h

Function: Perform a feed hold

Scope: Coordinate-system specific

Syntax: **h**, **hold**

The **h** command causes Power PMAC to suspend motion program execution for the specified coordinate system(s) by bringing the coordinate-system time base to zero, decelerating along its path starting immediately. If not immediately preceded by a coordinate system list, it will hold execution for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will hold execution for all coordinate systems in the list.

The motion program execution is suspended while in feed hold mode, but technically it is still executing. If it is subsequently desired that program execution will not be resumed, program execution should be fully aborted with the **a** command.

The **h** command is very similar in effect to a **%0** command, except that deceleration and subsequent re-acceleration are controlled by **Coord[*x*].FeedHoldSlew**, not by **Coord[*x*].TimeBaseSlew**. Also, execution can be resumed with an **R** or **S** command, instead of a **%100** command. In addition **h** works under external time base, whereas a **%0** command does not.

In general, motion will not stop at a programmed point on an **h** command. Full-speed execution along the path will commence again on an **r** or **s** command. The ramp up to the full time-base value (whether internally or externally set) will take place at a rate set by **Coord[*x*].FeedHoldSlew**. Once the full time-base value is reached, **Coord[*x*].TimeBaseSlew** determines the rate of any time-base changes.

Note that specifying a list of multiple coordinate systems does not change the modally addressed coordinate system for subsequent coordinate-system-specific commands.

The short form of this command (**h**) is useful for typing in terminal mode. The long form (**hold**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**hold**) must be used. The short form (**h**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

The equivalent buffered direct program command is **hold**.

Examples

```
h                                      // Hold program execution in addressed C.S.
&2h                                    // Address C.S.2 and hold program execution there
&1,3,5h                                // Hold program execution in C.S. 1, 3, and 5
&*h                                    // Hold program execution in all C.S.
```

hm

Function: Start homing-search move

Scope: Motor specific

Syntax: **hm**, **home**

The **hm** command causes Power PMAC to start homing-search move(s) for the specified motor(s). If not immediately preceded by a motor list, it will start a homing-search move for the presently addressed motor. If immediately preceded by a motor list, it will start homing-search moves for all motors in the list.

Note that specifying a list of multiple motors does not change the modally addressed motor for subsequent motor-specific commands.

The characteristics of the homing-search move profile for a given motor are controlled by data structure setup elements for that motor: **Motor[*x*].HomeVel**, **Motor[*x*].JogTa**, **Motor[*x*].JogTs**, and **Motor[*x*].HomeOffset**.

The on-line home command simply starts the homing search routine. Power PMAC provides no automatic indication that the search has completed or whether the move completed successfully. The status of the motor must be monitored to determine when the move has completed and the success of the search.

By contrast, when a homing-search move is commanded by a buffered command within a motion program – e.g. **home1,2** – the motion program will keep track of the completion status itself as part of its sequencing algorithms. When the homing-search move is commanded by a buffered command within a PLC program, the status of the motor must be monitored to determine when the move has completed and the success of the search, as with the on-line command.

The short form of this command (**hm**) is useful for typing in terminal mode. The long form (**home**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**home**) must be used. The short form (**hm**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line homing-search move command (**#*x*hm**) and the buffered-program homing-search move command (**home*x***).

Examples

```
hm                          // Start homing-search move on the addressed motor
#1hm                        // Address Motor 1, start homing-search move on it
#2,4,6hm                    // Command Motors 2, 4, and 6 to start homing move
#*hm                        // Command all active motors to start homing move
```

hmz

Function: Perform zero-move homing

Scope: Motor specific

Syntax: **hmz**, **homez**

The **hmz** command causes Power PMAC to perform a zero-move homing for the specified motor(s). If not immediately preceded by a motor list, it will perform a zero-move a homing for the presently addressed motor. If immediately preceded by a motor list, it will perform a zero-move homing for all motors in the list.

Note that specifying a list of multiple motors does not change the modally addressed motor for subsequent motor-specific commands.

If **Motor[*x*].pAbsPos** is set to the default value of 0, trigger, the **hmz** command causes Power PMAC simply to re-define the present commanded position as the home (motor-zero) position, without movement.

However, if **Motor[*x*].pAbsPos** is set to a non-zero value, Power PMAC will read the register whose address is specified by the value for the present absolute position of the motor. The position data read at this address is processed according to the settings of **Motor[*x*].AbsPosFormat**, **Motor[*x*].AbsPosSf**, and **Motor[*x*].HomeOffset**.

The short form of this command (**hmz**) is useful for typing in terminal mode. The long form (**homez**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**homez**) must be used. The short form (**hmz**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line zero-move homing command (**#*x*hmz**) and the buffered-program zero-move homing command (**homez*x***).

Examples

```
hmz                    // Perform zero-move homing on the addressed motor
#1hmz                  // Address Motor 1, perform zero-move homing on it
#2,4,6hmz              // Command Motors 2, 4, and 6 to do zero-move homing
#*hmz                  // Command all active motors to do zero-move homing
```

hold

Function: Perform a feed hold

Scope: Coordinate-system specific

Syntax: **h**, **hold**

The **hold** command is the "long-form" version of the **h** command, causing Power PMAC to suspend motion program execution for the specified coordinate system(s) by bringing the coordinate-system time base to zero, decelerating along its path starting immediately. If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

home

Function: Start homing-search move

Scope: Motor specific

Syntax: **hm**, **home**

The **home** command is the "long-form" version of the **hm** command, causing Power PMAC to start homing-search move(s) for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

homez

Function: Perform zero-move homing

Scope: Motor specific

Syntax: **hmz**, **homez**

The **hmz** command is the "long-form" version of the **hmz** command, causing Power PMAC to perform a zero-move homing for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

I*{data}*

Function: Report value of I-variable

Scope: Global

Syntax: **I*{data}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **I*{data}*** command causes Power PMAC to report the present value of the specified I-variable. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 65,535 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                                   // Include variable names in query responses
I1                                      // Query value of variable I1
I1=4                                    // Power PMAC response of the value of I1

I(P1)                                   // Query value of I-variable numbered by P1
I122=3                                  // Power PMAC response of the value of I122

I(200+P17)                              // Query value of I-variable numbered by 200+P17
I220=-5.35                              // Power PMAC response of the value of I220

echo2                                   // Do not include variable names in query responses
I1                                      // Query value of variable I1
4                                       // Power PMAC response of the value of I1
```

I{data}={expression}

Function: Assign value to I-variable

Scope: Global

Syntax: **I*{data}*=*{expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **I*{data}*=*{expression}*** command causes Power PMAC to set the specified I-variable to the value of the expression on the right side of the equals sign. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 65,535 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, the command will be entered into that buffer for later execution.

Examples

```
I1=4                          // Set variable I1 to 4
I(P1)=3                       // Set I-variable numbered by P1 to 3
I(P1+P17)=5*sqrt(P100)        // Set I-var numbered by (P1+P17) to expression value
```

I{data}->

Function: Report definition of I-variable

Scope: Global

Syntax: **I*{data}*->**

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **I*{data}*->** command causes Power PMAC to report the definition of the specified I-variable. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 65,535 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression

must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Each I-variable is defined to a Power PMAC data structure element. Unlike M-variables, I-variable definitions cannot be changed by the user.

Bit 2 (value 4) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                          // Include variable names in query responses
I122->                         // Query definition of variable I122
I122->Motor[1].JogSpeed        // Power PMAC response of the definition of I122

I(P1)->                        // Query definition of I-variable numbered by P1 (=222)
I222->Motor[2].JogSpeed        // Power PMAC response of the definition of I222

echo4                          // Do not include variable names in query responses
I122->                         // Query definition of variable I122
Motor[1].JogSpeed              // Power PMAC response of the definition of I122
```

I*{variable list}*

Function: Report value(s) of I-variable(s) in list

Scope: Global

Syntax: **I*{variable list}*[:*{constant}*]**

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
    - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
    - ***{constant}*[,*{constant}*[,*{constant}*]]** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- the optional ***{constant}*** is a positive integer specifying the number of reported values per response line. If no value is specified here, only one value per response line is reported. This can only be used if the list is specified by a consecutive range of numbers.

The **I*{variable list}*** command causes Power PMAC to report the present value(s) of the I-variable(s) in the list. The list can either be a set of consecutively numbered variables, specified by the

numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
I100                        // Query value of var I100 (set quantity 1, spacing 1)
I100=1                      // Power PMAC response of the value of I100

I120..122                   // Query value of I120, I121, I122
I120=-4                     // Power PMAC response of the value of I120
I121=3.5                    // Power PMAC response of the value of I121
I122=25                     // Power PMAC response of the value of I122

I120,3                      // Query value of 3 I-vars, starting at I120 (spacing 1)
I120=-4                     // Power PMAC response of the value of I120
I121=3.5                    // Power PMAC response of the value of I121
I122=25                     // Power PMAC response of the value of I122

I122,3,100                  // Query value of 3 I-vars, starting at I120, spacing 100
I122=25                     // Power PMAC response of the value of I122
I222=25                     // Power PMAC response of the value of I222
I322=25                     // Power PMAC response of the value of I322

I3100..3109:5               // Query value of I3100 - I3109, 5 values per line
I3100=0,0,0,0,0             // Power PMAC response of the value of I3100 - I3104
I3105=0,0,0,0,0             // Power PMAC response of the value of I3105 - I3109
```

I*{variable list}={expression}*

Function: Set value(s) of I-variable(s) in list

Scope: Global

Syntax: **I*{variable list}={expression}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
    - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).

  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).

- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **I*{variable list}={expression}*** command causes Power PMAC to set the value(s) of the I-variable(s) in the list to the value of the expression on the right side of the equals sign. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Examples

```
I100=1                  // Set value of I100 (set quantity 1, spacing 1) to 1
I122..123=P1+5          // Set value of I122, I123 to (P1+5)
II22,2=17.5             // Set val of 2 I-vars, starting at I122 (spacing 1) to 17.5
I122,3,100=P25+7        // Set value of 3 I-vars, starting at I122, spacing 100
                        // (I122, I222, I322) to (P25+7)
```

I*{variable list}->*

Function: Report definition(s) of I-variable(s) in list

Scope: Global

Syntax: **I*{variable list}->***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:

  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).

  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).

The **I{*variable list*}->** command causes Power PMAC to report the definition(s) of the I-variable(s) in the list. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, the command will be rejected with an error.

Bit 2 (value 4) of the echo mode parameter for the communications thread, set by the **echo{*constant*}** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
I122->                          // Query definition of variable I122
I122->Motor[1].JogSpeed         // Power PMAC response of the definition of I122

I120..122->                     // Query definitions of I120, I121, I122
I120->Motor[1].JogTa            // Power PMAC response of the definition of I120
I121->Motor[1].JogTs            // Power PMAC response of the definition of I121
I122->Motor[1].JogSpeed         // Power PMAC response of the definition of I122

I120,3->                        // Query defs of 3 I-vars starting with I120 (spacing 1)
I120->Motor[1].JogTa            // Power PMAC response of the definition of I120
I121->Motor[1].JogTs            // Power PMAC response of the definition of I121
I122->Motor[1].JogSpeed         // Power PMAC response of the definition of I122

I122,3,100->                    // Query defs of 3 I-vars starting with I120, spacing 100
I122->Motor[1].JogSpeed         // Power PMAC response of the definition of I122
I222->Motor[2].JogSpeed         // Power PMAC response of the definition of I222
I322->Motor[3].JogSpeed         // Power PMAC response of the definition of I322
```

j+

Function: Indefinite jog positive move

Scope: Motor specific

Syntax: **j+**, **jog+**

The **j+** command causes Power PMAC to start an indefinite jog move in the positive direction for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

If the software positive limit is enabled for the motor (**Motor[*x*].MaxPos** > **Motor[*x*].MinPos**), this command is treated as a command to jog to the positive limit, so it will automatically stop at the software positive limit (not just begin to stop as it passes the limit) even if no other command is given.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j+**) is useful for typing in terminal mode. The long form (**jog+**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog+**) must be used. The short form (**j+**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j+**) and the buffered-program jog command (**jog+*x***).

j-

Function: Indefinite jog negative move

Scope: Motor specific

Syntax: **j-**, **jog-**

The **j-** command causes Power PMAC to start an indefinite jog move in the negative direction for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

If the software negative limit is enabled for the motor (**Motor[*x*].MinPos** < **Motor[*x*].MaxPos**), this command is treated as a command to jog to the negative limit, so it will automatically stop at the software negative limit (not just begin to stop as it passes the limit) even if no other command is given.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j-**) is useful for typing in terminal mode. The long form (**jog-**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog-**) must be used. The short form (**j-**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j-**) and the buffered-program jog command (**jog-*x***).

j/

Function: Jog stop, or close loop

Scope: Motor specific

Syntax: **j/**, **jog/**

The **j/** command causes Power PMAC to come to a controlled closed-loop stop on the specified motor(s), whether the motor is in a closed-loop or open-loop state at the time of the command. If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The commanded stop is governed by the values of motor data-structure setup elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs** in force at the time of the command. If the command is issued during a closed-loop move (jogging or homing-search), it uses the motor commanded velocity as the starting value for its deceleration ramp. If the command is issued when the motor is in an open-loop state, it uses the motor actual velocity as the starting value.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j/**) is useful for typing in terminal mode. The long form (**jog/**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog/**) must be used. The short form (**j/**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j/**) and the buffered-program jog command (**jog/*x***).

j=

Function: Jog move to pre-jog position

Scope: Motor or coordinate-system specific

Syntax: **j=**, **jog=**

The **j=** command causes Power PMAC to start a jog move to the most recent programmed position for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list. If immediately preceded by a coordinate system list, it will do so for all motors for each coordinate system in the list.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

The **j=** command is useful when a motion program has been temporarily suspended and one or more of the motors jogged away from the suspended position. This command can be used to return the motors to the suspended position so the program can be resumed.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j=**) is useful for typing in terminal mode. The long form (**jog=**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog=**) must be used. The short form (**j=**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j=**) and the buffered-program jog command (**jogret*x***).

j={constant}

Function: Jog move to specified position

Scope: Motor specific

Syntax: **j=*{constant}***, **jog=*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the destination position in motor units

The **j=*{constant}*** command causes Power PMAC to start a jog move to the specified position (in motor units, relative to the motor zero position) for the specified motor(s). If not immediately preceded by

a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j=*{constant}***) is useful for typing in terminal mode. The long form (**jog=*{constant}***) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog=*{constant}***) must be used. The short form (**j=*{constant}***) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j=*{constant}***) and the buffered-program jog command (**jog*x*=*{data}***).

Examples

```
j=0                          // Jog addressed motor to zero position
#1j=10000                    // Jog motor 1 to position of 10,000 units
#2j=-347.28                  // Jog motor 2 to position of -347.28 units
#3,4,5j=76921                // Jog motors 3 - 5 to position of 76,921 units
```

j==*{constant}*

Function: Jog move to specified position, making that position the “pre-jog” position

Scope: Motor specific

Syntax: **j==*{constant}***, **jog==*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the destination position in motor units

The **j==*{constant}*** command causes Power PMAC to start a jog move to the specified position (in motor units, relative to the motor zero position) for the specified motor(s). This destination position is also made the “pre-jog” position, so subsequent **J=** commands will return the motor to that position. If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j==*{constant}***) is useful for typing in terminal mode. The long form (**jog==*{constant}***) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog==*{constant}***) must be used. The short form (**j==*{constant}***) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j==*{constant}***) and the buffered-program jog command (**jogret*x*=*{data}***).

Examples

```
j==0                          // Jog addressed motor to zero position
#1j==10000                    // Jog motor 1 to position of 10,000 units
#2j==-347.28                  // Jog motor 2 to position of -347.28 units
#3,4,5j==76921                // Jog motors 3 - 5 to position of 76,921 units
```

j=*

Function: Jog move to specified variable position

Scope: Motor specific

Syntax: **j=***, **jog=***

The **j=*** command causes Power PMAC to start a jog move to the position set by the motor data structure element **Motor[*x*].ProgJogPos** (in motor units, relative to the motor zero position) for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j=***) is useful for typing in terminal mode. The long form (**jog=***) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog=***) must be used. The short form (**j=***) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j=***) and the buffered-program jog command (**jog*x*=*{data}***).

Examples

```
Motor[1].ProgJogPos=10000          // Set jog position value to 10,000 units
#1j=*                              // Jog motor 1 to preset position value
Motor[2].ProgJogPos=-347.28        // Set jog position value to -347.28 units
#2j=*                              // Jog motor 2 to preset position value
Motor[3].ProgJogPos=P1             // Set jog position value to value of P1
Motor[4].ProgJogPos=P1             // Set jog position value to value of P1
Motor[5].ProgJogPos=P1             // Set jog position value to value of P1
#3,4,5j=*                          // Jog motors 3 – 5 to position of P1 units
```

j:*{constant}*

Function: Jog move of specified distance relative to command position

Scope: Motor specific

Syntax: **j:*{constant}***, **jog:*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the signed destination distance in motor units

The **j:*{constant}*** command causes Power PMAC to start a jog move of the specified distance (in motor units, relative to the present motor *command* position) for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

Compare this command to the similar **j^*{constant}*** command, which is relative to the present *actual* position. In general, the **j:*{constant}*** command is more useful, because its destination is not dependent on the position following error at the time of the command.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j:*{constant}***) is useful for typing in terminal mode. The long form (**jog:*{constant}***) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog:*{constant}***) must be used. The short form (**j:*{constant}***) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j:*{constant}***) and the buffered-program jog command (**jog*x*:*{data}***).

Examples

```
j:100                       // Jog addressed motor a distance of +100 units
#1j:5000                    // Jog motor 1 a distance of +5,000 units
#2j:-279.43                 // Jog motor 2 a distance of -279.43 units
#3,4,5j:49232               // Jog motors 3 – 5 a distance of 49,232 units
```

j:*

Function: Jog move of specified variable distance relative to command position

Scope: Motor specific

Syntax: **j:*** , **jog:***

The **j:*** command causes Power PMAC to start a jog move of the (signed) distance set by the motor data structure element **Motor[*x*].ProgJogPos** (in motor units, relative to the present motor *command* position) for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

Compare this command to the similar **j^*** command, which is relative to the present *actual* position. In general, the **j:*** command is more useful, because its destination is not dependent on the position following error at the time of the command.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j:***) is useful for typing in terminal mode. The long form (**jog:***) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog:***) must be used. The short form (**j:***) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j:***) and the buffered-program jog command (**jog*x*:*{data}***).

Examples

```
Motor[1].ProgJogPos=5000          // Set jog distance value to 5,000 units
#1j:*                             // Jog motor 1 the preset distance
Motor[2].ProgJogPos=-822.67       // Set jog position value to -822.67 units
#2j:*                             // Jog motor 2 the preset distance
Motor[3].ProgJogPos=P1+500        // Set jog distance value to (P1+500)
```

```
Motor[4].ProgJogPos=P1+500          // Set jog distance value to (P1+500)
Motor[5].ProgJogPos=P1+500          // Set jog distance value to (P1+500)
#3,4,5j:*                           // Jog motors 3 - 5 the preset distance
```

j^{constant}

Function: Jog move of specified distance relative to actual position

Scope: Motor specific

Syntax: **j^*{constant}***, **jog^*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the signed destination distance in motor units

The **j^*{constant}*** command causes Power PMAC to start a jog move of the specified distance (in motor units, relative to the present motor *actual* position) for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

Compare this command to the similar **j:*{constant}*** command, which is relative to the present *command* position. In general, the **j:*{constant}*** command is more useful, because its destination is not dependent on the position following error at the time of the command.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j^*{constant}***) is useful for typing in terminal mode. The long form (**jog^*{constant}***) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog^*{constant}***) must be used. The short form (**j^*{constant}***) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j:*{constant}***) and the buffered-program jog command (**jog*x*:*{data}***).

Examples

```
j^100                               // Jog addressed motor a distance of +100 units
#1j^5000                            // Jog motor 1 a distance of +5,000 units
#2j^-279.43                         // Jog motor 2 a distance of -279.43 units
#3,4,5j^49232                       // Jog motors 3 - 5 a distance of 49,232 units
```

j^*

Function: Jog move of specified variable distance relative to actual position

Scope: Motor specific

Syntax: **j^***, **jog^***

The **j^*** command causes Power PMAC to start a jog move of the (signed) distance set by the motor data structure element **Motor[*x*].ProgJogPos** (in motor units, relative to the present motor *actual* position) for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

Compare this command to the similar **j:*** command, which is relative to the present *command* position. In general, the **j:*** command is more useful, because its destination is not dependent on the position following error at the time of the command.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of this command (**j^***) is useful for typing in terminal mode. The long form (**jog^***) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog^***) must be used. The short form (**j^***) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line jog command (**#*x*j^***) and the buffered-program jog command (**jog*x*^*{data}***).

Examples

```
Motor[1].ProgJogPos=5000            // Set jog distance value to 5,000 units
#1j^*                               // Jog motor 1 the preset distance
Motor[2].ProgJogPos=-822.67         // Set jog position value to -822.67 units
#2j^*                               // Jog motor 2 the preset distance
Motor[3].ProgJogPos=P1+500          // Set jog distance value to (P1+500)
Motor[4].ProgJogPos=P1+500          // Set jog distance value to (P1+500)
Motor[5].ProgJogPos=P1+500          // Set jog distance value to (P1+500)
#3,4,5j^*                           // Jog motors 3 - 5 the preset distance
```

{jog command}^{constant}

Function: Jog-until-trigger move

Scope: Motor specific

Syntax:
j=^*{constant}*, jog=^*{constant}*
j=*{constant}*^*{constant}*, jog=*{constant}*^*{constant}*
j:*{constant}*^*{constant}*, jog:*{constant}*^*{constant}*
j^*{constant}*^*{constant}*, jog^*{constant}*^*{constant}*
j=*^*{constant}*, jog=*^*{constant}*
j:*^*{constant}*, jog:*^*{constant}*
j^*^*{constant}*, jog^*^*{constant}*

where:

- ***{constant}*** at the end of the command is a floating-point value specifying the (signed) distance from the trigger position to the commanded destination of the post-trigger move, in motor units.

A "jog-until-trigger" command causes Power PMAC to start a definite jog move of the type specified for the specified motor(s). If a pre-defined trigger condition occurs during this move, Power PMAC will automatically break into the move trajectory and replace the remaining portion with a jog move to a destination whose distance from the trigger position is determined by the final ***{constant}*** in the command. If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

The trigger condition is set by the motor setup data-structure element **Motor[*x*].CaptureMode**.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of these commands (**j**…) is useful for typing in terminal mode. The long form (**jog**…) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog**…) must be used. The short form (**j**…) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between these on-line jog commands (e.g. **#*x*j=*{constant}*^***) and the buffered-program jog command (e.g. **jog*x*=*{data}*^*{data}***).

Examples

```
j=^0                        // Jog addressed motor to pre-jog position if no trigger,
                            // return to trigger position if found
#1j=25000^-1000             // Jog Motor 1 to 25,000 units if no trigger,
                            // return to trigger position minus 1,000 units if found
#2j:30^0.2                  // Jog Motor 2 a distance of 30 units if no trigger,
                            // return to trigger position plus 0.2 units if found
#3j^-75^1.1                 // Jog Motor 3 a distance of -75 units if no trigger,
                            // return to trigger position plus 1.1 units if found
```

```
Motor[4].ProgJogPos=-5000      // Set (no-trigger) jog position to -5,000 units
#4j=*^25                       // Jog Motor 4 to preset position if no trigger,
                               // return to trigger position plus 25 units if found
Motor[5].ProgJogPos=32.1       // Set (no-trigger) jog distance to +32.1 units
#5j=*^-0.01                    // Jog Motor 5 the preset distance if no trigger,
                               // return to trigger position minus 0.01 units if found
Motor[6].ProgJogPos=-7.9       // Set (no-trigger) jog distance to -7.9 units
#6j=*^225                      // Jog Motor 6 the preset distance if no trigger,
                               // return to trigger position plus 225 units if found
```

*{jog command}^**

Function: Variable jog-until-trigger move

Scope: Motor specific

Syntax: **j=^***, **jog=*^***
j=*{constant}^**, **jog=*{constant}^****
j:*{constant}^**, **jog:*{constant}^****
j^*{constant}^**, **jog^*{constant}^****
j=*^*, **jog=*^***
j:*^* , **jog:*^***
j^*^*, **jog^*^***

A variable "jog-until-trigger" command causes Power PMAC to start a definite jog move of the type specified for the specified motor(s). If a pre-defined trigger condition occurs during this move, Power PMAC will automatically break into the move trajectory and replace the remaining portion with a jog move to a destination whose distance from the trigger position is determined by the value of **Motor[*x*].JogOffset** at the time of the trigger. If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The commanded jog move is governed by the values of motor data-structure setup elements **Motor[*x*].JogSpeed**, **Motor[*x*].JogTa**, and **Motor[*x*].JogTs** in force at the time of the command. The command can be issued successfully even if the motor is presently executing another jogging or homing-search move, or even if the motor is open-loop enabled or killed.

The trigger condition is set by the motor setup data-structure element **Motor[*x*].CaptureMode**.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

The short form of these commands (**j**…) is useful for typing in terminal mode. The long form (**jog**…) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**jog**…) must be used. The short form (**j**…) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between these on-line jog commands (e.g. **#*x*j=*{constant}^****) and the buffered-program jog command (e.g. **jog*x*=*{data}^{data}***).

Examples

```
Motor[1].JogOffset=2*P1        // Set post-trigger offset to 2*P1
#1j=25000^*                    // Jog Motor 1 to 25,000 units if no trigger,
                               // return to trigger position plus offset if found
#2j:30^*                       // Jog Motor 2 a distance of 30 units if no trigger,
                               // return to trigger position plus offset if found
#3j^-75^*                      // Jog Motor 3 a distance of -75 units if no trigger,
                               // return to trigger position plus offset if found
Motor[4].ProgJogPos=-5000      // Set (no-trigger) jog position to -5,000 units
Motor[4].JogOffset=P5-4        // Set post-trigger offset to P5-4
#4j=*^*                        // Jog Motor 4 to preset position if no trigger,
                               // return to trigger position plus offset if found
Motor[5].ProgJogPos=32.1       // Set (no-trigger) jog distance to +32.1 units
Motor[5].JogOffset=5*P9        // Set post-trigger offset to 5*P9
#5j=*^*                        // Jog Motor 5 the preset distance if no trigger,
                               // return to trigger position plus offset if found
Motor[6].ProgJogPos=-7.9       // Set (no-trigger) jog distance to -7.9 units
Motor[6].JogOffset=P8          // Set post-trigger offset to P8
#6j=*^225                      // Jog Motor 6 the preset distance if no trigger,
                               // return to trigger position plus 225 units if found
```

jog

Function: Indefinite jog move transitioning from open loop

Scope: Motor specific

Syntax: **jog**

The **jog** command causes Power PMAC to start an indefinite jog move, transitioning from an open-loop move (from an **out*{constant}*** command), using the present actual velocity of the open loop move as its commanded velocity, for the specified motor(s). If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

The **jog** command has no effect on the motor if the motor is already executing a jog move.

Power PMAC will reject this command with an error if the motor is defined to a positioning axis in a coordinate system that is running a motion program at the time of the command.

jog+

Function: Indefinite jog positive move

Scope: Motor specific

Syntax: **j+**, **jog+**

The **jog+** command is the "long-form" version of the **j+** command, causing Power PMAC to start an indefinite jog move in the positive direction for the specified motor(s). If **Sys.NoShortCmds** is set to its

default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog-

Function: Indefinite jog negative move

Scope: Motor specific

Syntax: **j-**, **jog-**

The **j-** command is the "long-form" version of the **j-** command, causing Power PMAC to start an indefinite jog move in the negative direction for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog/

Function: Jog stop, or close loop

Scope: Motor specific

Syntax: **j/**, **jog/**

The **j/** command is the "long-form" version of the **j/** command, causing Power PMAC to come to a controlled closed-loop stop on the specified motor(s), whether the motor is in a closed-loop or open-loop state at the time of the command. If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog=

Function: Jog move to pre-jog position

Scope: Motor or coordinate-system specific

Syntax: **j=**, **jog=**

The **j=** command is the "long-form" version of the **j=** command, causing Power PMAC to start a jog move to the most recent programmed position for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog=*{constant}*

Function: Jog move to specified position

Scope: Motor specific

Syntax: **j=*{constant}***, **jog=*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the destination position in motor units

The **jog=*{constant}*** command is the "long-form" version of the **j=*{constant}*** command, causing Power PMAC to start a jog move to the specified position (in motor units, relative to the motor zero position) for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog==*{constant}*

Function: Jog move to specified position, making that position the "pre-jog" position

Scope: Motor specific

Syntax: **j==*{constant}***, **jog==*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the destination position in motor units

The **jog==*{constant}*** command is the "long-form" version of the **j==*{constant}*** command, causing Power PMAC to start a jog move to the specified position (in motor units, relative to the motor zero position) for the specified motor(s). This destination position is also made the "pre-jog" position, so subsequent **J=** commands will return the motor to that position. If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog=*

Function: Jog move to specified variable position

Scope: Motor specific

Syntax: **j=***, **jog=***

The **jog=*** command is the "long-form" version of the **j=*** command, causing Power PMAC to start a jog move to the position set by the motor data structure element **Motor[*x*].ProgJogPos** (in motor units, relative to the motor zero position) for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog:*{constant}*

Function: Jog move of specified distance relative to command position

Scope: Motor specific

Syntax: **j:*{constant}***, **jog:*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the signed destination distance in motor units

The **jog:*{constant}*** command is the "long-form" version of the **j:*{constant}*** command, causing Power PMAC to start a jog move of the specified distance (in motor units, relative to the present motor *command* position) for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog:*

Function: Jog move of specified variable distance relative to command position

Scope: Motor specific

Syntax: **j:***, **jog:***

The **jog:*** command is the "long-form" version of the **j:*** command, causing Power PMAC to start a jog move of the (signed) distance set by the motor data structure element **Motor[*x*].ProgJogPos** (in motor units, relative to the present motor *command* position) for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog^*{constant}*

Function: Jog move of specified distance relative to actual position

Scope: Motor specific

Syntax: **j^*{constant}***, **jog^*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the signed destination distance in motor units

The **jog^*{constant}*** command is the "long-form" version of the **j^*{constant}*** command, causing Power PMAC to start a jog move of the specified distance (in motor units, relative to the present motor *actual* position) for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

jog^*

Function: Jog move of specified variable distance relative to actual position

Scope: Motor specific

Syntax: **j^***, **jog^***

The **jog^*** command is the "long-form" version of the **j^*** command, causing Power PMAC to start a jog move of the (signed) distance set by the motor data structure element **Motor[*x*].ProgJogPos** (in motor units, relative to the present motor *actual* position) for the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

k

Function: Kill motor output

Scope: Motor specific

Syntax: **k**, **kill**

The **k** command causes Power PMAC to "kill" the servo outputs of the specified motor(s). If not immediately preceded by a motor list, it will kill the outputs of the presently addressed motor. If immediately preceded by a motor list, it will kill the outputs of all motors in the list. Note that the coordinate-system **disable** command kills all motors in a coordinate system.

"Killing" a motor causes the servo loop to be opened, the servo output value forced to zero (although output bias terms are still applied), and the amplifier-enable signal forced to the disable state. Note that Power PMAC automatically kills a motor on an amplifier fault or a fatal following error fault.

This immediate-kill command is intended for emergency killing of motors. Planned killing of motors with automatic brake control should be done with the similar delayed **dkill** command, so that the brakes have time to engage fully. For motors without automatic brake control, it does not matter which command is used.

Power PMAC will reject a motor-specific **k** command with an error if the motor is in a coordinate system that is currently running a motion program; program execution must be stopped before these commands can be given. However, the **#*k** command will act on all motors, even those in coordinate systems executing motion programs, aborting any motion programs in the process.

Closed-loop enabled control of a motor can be resumed with a **j/** command. Closed-loop enabled control for all the motors in a coordinate system can be resumed with an **enable** command. Open-loop enabled control of a motor can be resumed with the **out*{constant}*** command.

The short form of this command (**k**) is useful for typing in terminal mode. The long form (**kill**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**kill**) must be used. The short form (**k**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

Note the difference in syntax between the on-line kill command (**#*x*k**) and the buffered-program kill command (**kill*x***).

kill

Function: Kill motor output

Scope: Motor specific

Syntax: **k**, **kill**

The **kill** command is the "long-form" version of the **k** command, causing Power PMAC to "kill" the servo outputs of the specified motor(s). If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

L*{data}*

Function: Report value of L-variable

Scope: Communications-thread specific

Syntax: **L*{data}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **L*{data}*** command causes Power PMAC to report the present value of the specified L-variable local to this communications thread. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Each communications thread has its own set of L-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of L-variables are independent of each other.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                                   // Include variable names in query responses
L1                                      // Query value of variable L1
```

```
L1=17.5                                 // Power PMAC response of the value of L1

L(P1)                                   // Query value of L-variable numbered by P1
L17=3                                   // Power PMAC response of the value of L17

L(P1+P17)                               // Query value of L-variable numbered by P1+P17
L20=-5.35                               // Power PMAC response of the value of L20

echo2                                   // Do not include variable names in query responses
L1                                      // Query value of variable L1
17.5                                    // Power PMAC response of the value of L1
```

L{data}={expression}

Function: Assign value to L-variable

Scope: Communications-thread specific

Syntax: **L*{data}={expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **L*{data}={expression}*** command causes Power PMAC to set the specified L-variable local to this communications thread to the value of the expression on the right side of the equals sign. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, the command will be entered into that buffer for later execution.

Each communications thread has its own set of L-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of L-variables are independent of each other.

Examples

```
L1=17.5                         // Set variable L1 to 17.5
L(P1)=3                         // Set L-variable numbered by P1 to 3
L(P1+P17)=5*sqrt(P100)          // Set L-var numbered by (P1+P17) to expression value
```

L{variable list}

Function: Report value(s) of L-variable(s) in list

Scope: Communications-thread specific

Syntax: **L*{variable list}*[:*{constant}*]**

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}*[,*{constant}*[,*{constant}*]]** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- the optional ***{constant}*** is a positive integer specifying the number of reported values per response line. If no value is specified here, only one value per response line is reported. This can only be used if the list is specified by a consecutive range of numbers.

The **L*{variable list}*** command causes Power PMAC to report the present value(s) of the L-variable(s) in the list local to this communications thread. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Each communications thread has its own set of L-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of L-variables are independent of each other.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
L1                          // Query value of variable L1 (set quantity 1, spacing 1)
L1=17.5                     // Power PMAC response of the value of L1

L100..102                   // Query value of L100, L101, L102
L100=-4                     // Power PMAC response of the value of L100
```

```
L101=3.14159                  // Power PMAC response of the value of L101
L102=0                        // Power PMAC response of the value of L102

L50,3                         // Query value of 3 L-vars, starting at L50 (spacing 1)
L50=-5.35                     // Power PMAC response of the value of L50
L51=7.12                      // Power PMAC response of the value of L51
L52=376452                    // Power PMAC response of the value of L52

L50,3,3                       // Query value of 3 L-vars, starting at L50, spacing 3
L50=-5.35                     // Power PMAC response of the value of L50
L53=99                        // Power PMAC response of the value of L53
L56=-0.002                    // Power PMAC response of the value of L56

L1..10:5                      // Query value of L1 - L10, 5 values per response line
L1=7,3,5,2,11                 // Power PMAC response of the value of L1 - L5
L6=8,33,4,9,-2                // Power PMAC response of the value of L6 - L10
```

L{variable list}={expression}

Function: Set value(s) of L-variable(s) in list

Scope: Communications-thread specific

Syntax: **L*{variable list}*=*{expression}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **L*{variable list}*=*{expression}*** command causes Power PMAC to set the value(s) of the L-variable(s) in the list local to this communications thread to the value of the expression on the right side of the equals sign. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Each communications thread has its own set of L-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of L-variables are independent of each other.

Examples

```
L1=17.5                     // Set value of L1 (set quantity 1, spacing 1) to 17.5
L100..102=P1+5              // Set value of L100, L101, L102 to (P1+5)
L50,3=-7                    // Set value of 3 L-vars, starting at L50 (spacing 1) to -7
L50,3,3=sqrt(P25)           // Set value of 3 L-vars, starting at L50, spacing 3
                            // (L50, L53, L56) to the square root of P25
```

list apc

Function: List program at program counter when aborted

Scope: Coordinate-system specific

Syntax: **`list apc [,[{constant}]]`**

where:

- the optional ***{constant}*** specifies the number of lines to be reported from the location of the program counter when the program was aborted

The **`list apc`** command causes Power PMAC to report the contents of the motion program or subprogram pointed to by the addressed coordinate system when it was aborted, starting at the point of the coordinate system's program counter at that time. The contents are reported in ASCII text form. No user-declared variable names or text substitutions are used.

The aborting could be from a user command, or automatic from a fault of some kind. When the abort occurs, the program counter for the coordinate system is automatically reset to the beginning of the (top-level) program. This command permits the user to see where the program was calculating when the abort occurred for diagnostic and debugging purposes.

If no value is specified at the end of the command and no comma is used (i.e. **`list apc`**), the entire remaining program is reported, and no line numbers are shown preceding each line of the program.

If no value is specified at the end of the command, but a comma is used (i.e. **`list apc,`**), the entire remaining program is reported, and line numbers are shown preceding each line of the program.

If a numerical value is specified at the end of the command after a comma (e.g. **`list apc,5`**), the number of lines specified by the value is reported, starting at the location of the program counter. The line number is shown preceding each reported line of the program. In addition, the program name is shown at the top, preceded by the names of any higher-level programs calling the currently executing program.

Note that the "line numbers" specified in these commands have no relationship to any "line labels" ("jump" or "sync") used in the routines. The first text line of the routine is always line number 0, the second line is line number 1, and so on.

Examples

```
list apc                    // List remaining program starting at program counter
Y20                         // Power PMAC response
dwell 50
X0 Y0

list apc,                   // List remaining program starting at program counter
prog 25                     // Power PMAC response
43:Y20
44:dwell 50
45:X0 Y0
46:end

list apc,2                  // List 2 program lines starting at program counter
prog 25                     // Power PMAC response
43:Y20
44:dwell 50
```

list *{kinematic buffer}*

Function: List contents of specified kinematic buffer

Scope: Coordinate-system specific

Syntax: **list *{kinematic buffer}[,[{constant},[{constant}]]]***

where:

- ***{kinematic buffer}*** specifies the type of the Power PMAC kinematic routine for the coordinate system. It can take the following forms:
  - **forward** for the forward-kinematic routine
  - **inverse** for the inverse-kinematic routine
- the first optional ***{constant}*** specifies the number of lines to be reported from the top (if no second constant is specified), or the number of the first line to be reported (if a second constant is reported)
- the second optional ***{constant}*** specifies the number of lines to be reported starting with the line specified by the first ***{constant}***

The **list *{kinematic buffer}*** command causes Power PMAC to report the contents of the specified kinematic-routine buffer for the addressed coordinate system. The contents are reported in ASCII text form. No user-declared variable names or text substitutions are used.

If no values are specified after the routine type and no comma is used (e.g. **list forward**), the entire routine is reported, and no line numbers are shown preceding each line of the routine.

If no values are specified after the routine type, but a comma is used (e.g. **list forward,**), the entire routine is reported, and line numbers are shown preceding each line of the routine.

If a single value is specified after the routine type (e.g. **list inverse,5**), this number of lines is reported, starting from the beginning of the routine, and the line number is shown preceding each reported line of the routine.

If two values are specified after the routine type (e.g. **list forward,8,20**), the first value specifies the starting line number to be reported, and the second value specifies the number of lines to be reported. The line number is shown preceding each reported line of the routine.

Note that the "line numbers" specified in these commands have no relationship to any "line labels" ("jump" or "sync") used in the routines. The first text line of the routine is always line number 0, the second line is line number 1, and so on.

list *{program buffer}*

Function: List contents of specified program buffer

Scope: Global

Syntax: **list *{program buffer}*[,*{constant}*,[*{constant}*]]]**

where:

- ***{program buffer}*** specifies the name of the Power PMAC program. It can take the following forms:
  - **prog *{constant}*** for motion programs, where ***{constant}*** is the number of the motion program
  - **plc *{constant}*** for PLC programs, where ***{constant}*** is the number of the PLC program
  - **subprog *{constant}*** for motion programs, where ***{constant}*** is the number of the subprogram
- the first optional ***{constant}*** specifies the number of lines to be reported from the top (if no second constant is specified), or the number of the first line to be reported (if a second constant is reported). Alternately, the letters **apc** can be used to specify the point in the program where execution was aborted.
- the second optional ***{constant}*** specifies the number of lines to be reported starting with the line specified by the first ***{constant}***

The **list *{program buffer}*** command causes Power PMAC to report the contents of the specified program buffer. The contents are reported in ASCII text form. No user-declared variable names or text substitutions are used.

If no values are specified after the program type and number and no comma is used (e.g. **`list prog 2000`**), the entire program is reported, and no line numbers are shown preceding each line of the program.

If no values are specified after the program type and number, but a comma is used (e.g. **`list prog 2000,`**), the entire program is reported, and line numbers are shown preceding each line of the program.

If a single value is specified after the program type and number (e.g. **`list plc 10,5`**), this number of lines is reported, starting from the beginning of the program, and the line number is shown preceding each reported line of the program.

If two values are specified after the program type and number (e.g. **`list subprog 1000,8,20`**), the first value specifies the starting line number to be reported, and the second value specifies the number of lines to be reported. The line number is shown preceding each reported line of the program.

Note that the "line numbers" specified in these commands have no relationship to any "line labels" ("jump" or "sync") used in programs. The first text line of the program is always line number 0, the second line is line number 1, and so on.

list pc

Function: List program at program counter

Scope: Coordinate-system specific

Syntax: **list pc** ***[,[{constant}]]***

where:

- the optional ***{constant}*** specifies the number of lines to be reported from the location of the program counter

The **list pc** command causes Power PMAC to report the contents of the motion program or subprogram presently pointed to by the addressed coordinate system, starting at the point of the coordinate system's program counter. The contents are reported in ASCII text form. No user-declared variable names or text substitutions are used. The program counter points to the next line in the motion program or subprogram to be calculated. This may be several moves ahead of the programmed move currently executing as a result of motion program calculations.

If no value is specified at the end of the command (i.e. **list pc**), the entire remaining program is reported, and no line numbers are shown preceding each line of the program.

If no value is specified at the end of the command, but a comma is used (i.e. **list pc,**), the entire remaining program is reported, and line numbers are shown preceding each line of the program.

If a numerical value is specified at the end of the command after a comma (e.g. **list pc,5**), the number of lines specified by the value is reported, starting at the location of the program counter. The line number is shown preceding each reported line of the program. In addition, the program name is shown at the top, preceded by the names of any higher-level programs calling the currently executing program.

Note that the "line numbers" specified in these commands have no relationship to any "line labels" ("jump" or "sync") used in the routines. The first text line of the routine is always line number 0, the second line is line number 1, and so on.

Examples

```
list pc                         // List remaining program starting at program counter
Y20                             // Power PMAC response
dwell 50
X0 Y0

list pc,                        // List remaining program starting at program counter
prog 25                         // Power PMAC response
43:Y20
44:dwell 50
45:X0 Y0
46:end

list pc,2                       // List 2 program lines starting at program counter
prog 25                         // Power PMAC response
43:Y20
44:dwell 50
```

list rotary

Function: List contents of rotary motion program buffer

Scope: Coordinate-system specific

Syntax: **list rotary** ***[,[{constant},[{constant}]]]***

where:

- the first optional ***{constant}*** specifies the number of lines to be reported from the most recently calculated line (if no second constant is specified), or the number of the first line after the most recently calculated line to be reported (if a second constant is reported). Alternately, the letters **apc** can be used to specify the point in the program where execution was aborted.

- the second optional ***{constant}*** specifies the number of lines to be reported starting with the line specified by the first ***{constant}***

The **list rotary** command causes Power PMAC to report the contents of the rotary motion program buffer for the addressed coordinate system. Only those program command lines that have not yet been calculated by the Power PMAC can be reported. Once a program command has been calculated (even if a resulting move has not yet been executed), it cannot ever be reported back with the **list rotary** command.

If no value is specified at the end of the command (i.e. **list rotary**), the entire remaining program is reported, and no line numbers are shown preceding each line of the program.

If no value is specified at the end of the command, but a comma is used (i.e. **list rotary,**), the entire remaining program is reported, and line numbers are shown preceding each line of the program.

If a numerical value is specified at the end of the command after a comma (e.g. **list rotary,5**), the number of lines specified by the value is reported, starting at the location of the program counter. The line number is shown preceding each reported line of the program. In addition, the program name is shown at the top, preceded by the names of any higher-level programs calling the currently executing program.

The rotary buffer for the coordinate system must be closed when this command is given.

M*{data}*

Function: Report value of M-variable

Scope: Global

Syntax: **M*{data}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **M{data}** command causes Power PMAC to report the present value of the specified M-variable. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 16,383 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo{constant}** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                                   // Include variable names in query responses
M1                                      // Query value of variable M1
M1=17.5                                 // Power PMAC response of the value of M1

M(P1)                                   // Query value of I-variable numbered by M1
M122=3                                  // Power PMAC response of the value of M122

M(200+P17)                              // Query value of M-variable numbered by P1+P17
M220=-5.35                              // Power PMAC response of the value of M220

echo7                                   // Do not include variable names in query responses
M1                                      // Query value of variable M1
17.5                                    // Power PMAC response of the value of M1
```

M{data}={expression}

Function: Assign value to M-variable

Scope: Global

Syntax: **M{data}={expression}**

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **M{data}={expression}** command causes Power PMAC to set the specified M-variable to the value of the expression on the right side of the equals sign. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 16,383 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, the command will be entered into that buffer for later execution.

Examples

```
M1=17.5                         // Set variable M1 to 17.5
M(P1)=3                         // Set M-variable numbered by P1 to 3
M(P1+P17)=5*sqrt(P100)          // Set M-var numbered by (P1+P17) to expression value
```

M{data}->

Function: Report definition of M-variable

Scope: Global

Syntax: **M*{data}*->**

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **M*{data}*->** command causes Power PMAC to report the definition of the specified M-variable. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 16,383 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Unlike I-variables, M-variable definitions can be changed by the user.

Bit 2 (value 4) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                           // Include variable names in query responses
M162->                          // Query definition of variable M162
M162->Motor[1].ActPos           // Power PMAC response of the definition of M162

M(P1)->                         // Query definition of M-variable numbered by P1 (=0)
M0->u.io:$A00000.8.1            // Power PMAC response of the definition of M0

echo7                           // Do not include variable names in query responses
M162->                          // Query definition of variable M162
Motor[1].ActPos                 // Power PMAC response of the definition of M162
```

M{data}->{address definition}

Function: Set definition of M-variable with address

Scope: Global

Syntax: **M*{data}*->*{address definition}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{address definition}*** consists of
  ***{format}.{mem bank}*:*{addr offset}*[.*{bit offset}*[.*{width}*]]**
  where:
  - ***{format}*** is a single letter representing the variable type (how the value at this address is to be interpreted). The choices are:
    - **s** specifies a signed integer of up to 32 bits that saturates (not rolls over) when a command attempts to assign a value greater than its range to it
    - **i** specifies a signed integer of up to 32 bits that rolls over (not saturates) when a command attempts to assign a value greater than its range to it
    - **u** specifies an unsigned integer of up to 32 bits that rolls over (not saturates) when a command attempts to assign a value greater than its range to it
    - **f** specifies a single-precision (32-bit) floating-point variable
    - **d** specifies a double-precision (64-bit) floating-point variable
  - ***{mem bank}*** specifies which "memory bank" is to be accessed with this M-variable. The choices are:
    - **user** specifies the reserved user buffer in Power PMAC RAM
    - **io** specifies the memory-mapped I/O in Power PMAC's address space
  - ***{adr offset}*** is a non-negative integer that specifies the byte-address offset from the base address of the selected memory bank.
  - the optional ***{bit offset}*** is an integer in the range 0 – 31 representing the starting bit number of the variable in the register at the specified address. This specification is valid only for integer variable forms. If no value is specified, a value of 0 is used
  - the optional ***{width}*** is an integer in the range 1 – 32 representing the number of bits to be used for the variable in the register at the specified address. This specification is valid only for integer variable forms. If no value is specified for either bit offset or width, a value of 32 is used for width. If a value is specified for bit offset, but not for width, a value of 1 is used for width.

The **M{data}->{address definition}** command causes Power PMAC to set the definition of the specified M-variable according to the address definition. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 16,383 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error. . Only the value of the expression at the time this command is sent to Power PMAC matters; if the expression value subsequently changes, the definition does not port to a different M-variable.

The first item in the address definition is the letter representing the variable format. The format determines how the information in the register is to be interpreted and treated. There are 3 possible integer formats: **s** (signed saturating integer), **i** (signed overflowing integer), and **u** (unsigned overflowing integer). Integer formats can be 1 – 32 bits in width. There are 2 possible floating-point formats: **f** (32-bit single-precision) and **d** (64-bit double-precision).

The second item in the address definition is the "memory bank" in which the variable is defined. Here there are two choices: **user** specifies the Power PMAC's large reserved user buffer in RAM (user-configurable size, 1 MB by default), and **io** specifies the addresses in Power PMAC's memory-mapped I/O for axis-interface registers, general-purpose analog and digital I/O, and even dual-ported RAM registers.

The third item in the address definition is the numerical "address offset" of the register of interest from the starting address of the memory bank chosen. Note that by specifying the offset here, it is not necessary to know the absolute physical memory address. Power PMAC will compute the absolute address by adding this offset to the base address for that memory bank. For reference, the base address of the user memory buffer can be found at **Sys.pushm**, and the base address of the memory-mapped I/O can be found at **Sys.piom**.

Address offsets for all integer formats and for single-precision floating-point format must be on "long integer" boundaries, and so evenly divisible by 4. Address offsets for double-precision floating-point format must be on "long long integer" boundaries and so evenly divisible by 8.

For the integer formats, it is possible to use only part of the 32-bit register defined by the address offset. If a bit offset is explicitly declared, this specifies the starting (lowest) bit number from the 32-bit register that will be used. This must take a value of 0 – 31. If no value is declared here, the starting bit number is 0 (and all 32 bits are used).

If a bit offset has been explicitly declared, a bit width can also be declared, specifying the number of bits to be used in the definition, starting at the bit declared as the bit offset. If no bit width is declared after an offset has been declared, a width of 1 is used. The sum of the bit offset and the bit width must not exceed 32.

Examples

```
M0->u.io:$A00000.8.1          // Set def of M0 in I/O space, 1 bit starting at bit 8
M80->s.user:0                 // Set def of M80 in user buffer, 32-bit signed int
M(P1)->d.user:$100            // Set def of M(P1) in user buffer, 64-bit float
M5000->u.io:$A10000.8.8       // Set def of M5000 in I/O space, 8-bit unsigned int
```

M{data}->{data structure element}

Function: Set definition of M-variable with data structure element

Scope: Global

Syntax: **M{data}->{data structure element}**

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{data structure element}*** is the name of a pre-defined Power PMAC data structure element

The **M{data}->{data structure element}** command causes Power PMAC to set the definition of the specified M-variable to point to the specified data structure element. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 16,383 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error. Only the value of the expression at the time this command is sent to Power PMAC matters; if the expression value subsequently changes, the definition does not port to a different M-variable.

This type of M-variable definition permits a "shorthand" name for a data structure element, or a customized name when a variable name is declared or defined for the M-variable.

In this type of definition, the M-variable automatically takes on the format of the specified data structure element.

Examples

```
M0->Sys.ServoCount          // Set definition of M0 to global servo cycle counter
M162->Motor[1].ActPos       // Set definition of M162 to #1 actual position register
M5187->Coord[1].InPos       // Set definition of M5187 to &1 in-position bit
M(P10)->Sys.Fdata[0]        // Set definition of M(P10) to user buffer location 0
                            // used as single-precision float
M15000->Sys.Ddata[L0]       // Set definition of M15000 to user buffer indexable
                            // location, used as double-precision float
```

M{data}->{self-referenced definition}

Function: Set definition of M-variable to self-referenced definition

Scope: Global

Syntax: **M{data}->{self-referenced definition}**

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

- ***{self-referenced definition}*** consists of
  ****[{format}[.{width}]]***
  where:

    - ***{format}*** is a single letter representing the variable type (how the value at this address is to be interpreted). The choices are:

        - **s** specifies a signed integer of up to 32 bits that saturates (not rolls over) when a command attempts to assign a value greater than its range to it

        - **i** specifies a signed integer of up to 32 bits that rolls over (not saturates) when a command attempts to assign a value greater than its range to it

        - **u** specifies an unsigned integer of up to 32 bits that rolls over (not saturates) when a command attempts to assign a value greater than its range to it

        - **f** specifies a single-precision (32-bit) floating-point variable

        - **d** specifies a double-precision (64-bit) floating-point variable (actually the last bit of the mantissa is lost, so technically it is only 63 bits). If no format is specified, Power PMAC will use the **d** format.

    - the optional ***{width}*** is an integer in the range 1 – 32 representing the number of bits to be used for the variable in the register at the specified address. This specification is valid only for integer variable forms. If no value is specified, a value of 32 is used for width.

The **M*{data}*->*{self-referenced definition}*** command causes Power PMAC to set the definition of the specified M-variable to use part of the definition register for the value of the variable. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 16,383 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error. . Only the value of the expression at the time this command is sent to Power PMAC matters; if the expression value subsequently changes, the definition does not port to a different M-variable.

A self-referenced M-variable is a general-purpose global user variable of a specific format. It can be useful to add to the total number of global variables, or to create a variable with a format different from that of the global P-variables, which are all double-precision floating-point variables. It can also be useful to "clear" a variable's definition, so that assigning a value to this variable cannot write to anything harmful.

It is not possible to execute synchronous assignments (e.g. **M100==1**) on a self-referenced M-variable. Attempting such an assignment in a Power PMAC program will result in a "no-op"; no error will be

Note reported.

Examples

```
M70->*u                          // Set M70 as self-defined unsigned 32-bit integer
M80->*s.16                       // Set M80 as self-defined 16-bit signed integer
M8000->*f                        // Set M8000 as self-defined 32-bit float
M(P1)->*d                        // Set M(P1) as self-defined 64-bit float
M9000->*                         // Set M8000 as self-defined 64-bit float
```

M*{variable list}*

Function: Report value(s) of M-variable(s) in list

Scope: Global

Syntax: **M*{variable list}*[:*{constant}*]**

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}*[,*{constant}*[,*{constant}*]]** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- the optional ***{constant}*** is a positive integer specifying the number of reported values per response line. If no value is specified here, only one value per response line is reported. This can only be used if the list is specified by a consecutive range of numbers.

The **M*{variable list}*** command causes Power PMAC to report the present value(s) of the M-variable(s) in the list. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Examples

```
M100                        // Query value of var M100 (set quantity 1, spacing 1)
M100=1                      // Power PMAC response of the value of M100

M120..122                   // Query value of M120, M121, M122
M120=-4                     // Power PMAC response of the value of M120
M121=3.5                    // Power PMAC response of the value of M121
M122=25                     // Power PMAC response of the value of M122

M120,3                      // Query value of 3 M-vars, starting at M120 (spacing 1)
M120=-4                     // Power PMAC response of the value of M120
M121=3.5                    // Power PMAC response of the value of M121
M122=25                     // Power PMAC response of the value of M122

M120,3,100                  // Query value of 3 M-vars, starting at M120, spacing 100
M122=25                     // Power PMAC response of the value of M122
M222=25                     // Power PMAC response of the value of M222
M322=25                     // Power PMAC response of the value of M322

M3100..3109:5               // Query value of M3100 - M3109, 5 values per line
M3100=0,0,0,0,0             // Power PMAC response of values of M3100 - M3104
M3105=0,0,0,0,0             // Power PMAC response of values of M3105 - M3109
```

M*{variable list}*=*{expression}*

Function: Set value(s) of M-variable(s) in list

Scope: Global

Syntax: **M*{variable list}*=*{expression}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The ***M{variable list}={expression}*** command causes Power PMAC to set the value(s) of the I-variable(s) in the list to the value of the expression on the right side of the equals sign. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Examples

```
M100=1                  // Set value of M100 (set quantity 1, spacing 1) to 1
M122..123=P1+5          // Set value of M122, M123 to (P1+5)
MI22,2=17.5             // Set val of 2 M-vars, starting at M122 (spacing 1) to 17.5
M122,3,100=P25+7        // Set value of 3 M-vars, starting at M122, spacing 100
                        // (M122, M222, MI322) to (P25+7)
```

M{variable list}->

Function: Report definition(s) of M-variable(s) in list

Scope: Global

Syntax: ***M{variable list}->***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
    - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
    - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).

The ***M{variable list}->*** command causes Power PMAC to report the definition(s) of the M-variable(s) in the list. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

Bit 2 (value 4) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

If a program buffer is open when this command is sent to Power PMAC, the command will be rejected with an error.

Examples

```
M0->                          // Query definition of M0
M0->u.io:$A00000.8.1          // Power PMAC response of the definition of M0

M0..2->                       // Query definitions of M0, M1, M2
M0->u.io:$A00000.8.1          // Power PMAC response of the definition of M0
M1->u.io:$A00000.9.1          // Power PMAC response of the definition of M1
M2->u.io:$A00000.10.1         // Power PMAC response of the definition of M2

M0,3->                        // Query defs of 3 M-vars starting with M0 (spacing 1)
M0->u.io:$A00000.8.1          // Power PMAC response of the definition of M0
M1->u.io:$A00000.9.1          // Power PMAC response of the definition of M1
M2->u.io:$A00000.10.1         // Power PMAC response of the definition of M2

M0,3,100->                    // Query defs of 3 M-vars starting with M0, spacing 100
M0->u.io:$A00000.8.1          // Power PMAC response of the definition of M0
M100->u.io:$A08000.8.1        // Power PMAC response of the definition of M100
M200->u.io:$A10000.8.1        // Power PMAC response of the definition of M200
```

M*{variable list}*->*{address definition}*

Function: Set definition of M-variable(s) in list with address

Scope: Global

Syntax: **M*{variable list}*->*{address definition}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- ***{address definition}*** consists of ***{format}.{mem bank}:{addr offset}[.{bit offset}[.{width}]]*** where:

- ***{format}*** is a single letter representing the variable type (how the value at this address is to be interpreted). The choices are:
  - **s** specifies a signed integer of up to 32 bits that saturates (not rolls over) when a command attempts to assign a value greater than its range to it
  - **i** specifies a signed integer of up to 32 bits that rolls over (not saturates) when a command attempts to assign a value greater than its range to it
  - **u** specifies an unsigned integer of up to 32 bits that rolls over (not saturates) when a command attempts to assign a value greater than its range to it
  - **f** specifies a single-precision (32-bit) floating-point variable
  - **d** specifies a double-precision (64-bit) floating-point variable
- ***{mem bank}*** specifies which "memory bank" is to be accessed with this M-variable. The choices are:
  - **user** specifies the reserved user buffer in Power PMAC RAM
  - **io** specifies the memory-mapped I/O in Power PMAC's address space
- ***{adr offset}*** is a non-negative integer that specifies the byte-address offset from the base address of the selected memory bank.
- the optional ***{bit offset}*** is an integer in the range 0 – 31 representing the starting bit number of the variable in the register at the specified address. This specification is valid only for integer variable forms. If no value is specified, a value of 0 is used
- the optional ***{width}*** is an integer in the range 1 – 32 representing the number of bits to be used for the variable in the register at the specified address. This specification is valid only for integer variable forms. If no value is specified for either bit offset or width, a value of 32 is used for width. If a value is specified for bit offset, but not for width, a value of 1 is used for width.

The **M*{variable list}*->*{address definition}*** command causes Power PMAC to set the definition(s) of the M-variable(s) in the list according to the address definition. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

The first item in the address definition is the letter representing the variable format. The format determines how the information in the register is to be interpreted and treated. There are 3 possible integer formats: **s** (signed saturating integer), **i** (signed overflowing integer), and **u** (unsigned overflowing integer). Integer formats can be 1 – 32 bits in width. There are 2 possible floating-point formats: **f** (32-bit single-precision) and **d** (64-bit double-precision).

The second item in the address definition is the "memory bank" in which the variable is defined. Here there are two choices: **`user`** specifies the Power PMAC's large reserved user buffer in RAM (user-configurable size, 1 MB by default), and **`io`** specifies the addresses in Power PMAC's memory-mapped I/O for axis-interface registers, general-purpose analog and digital I/O, and even dual-ported RAM registers.

The third item in the address definition is the numerical "address offset" of the register of interest from the starting address of the memory bank chosen. Note that by specifying the offset here, it is not necessary to know the absolute physical memory address. Power PMAC will compute the absolute address by adding this offset to the base address for that memory bank. For reference, the base address of the user memory buffer can be found at **Sys.pushm**, and the base address of the memory-mapped I/O can be found at **Sys.piom**.

Address offsets for all integer formats and for single-precision floating-point format must be on "long integer" boundaries, and so evenly divisible by 4. Address offsets for double-precision floating-point format must be on "long long integer" boundaries and so evenly divisible by 8.

For the integer formats, it is possible to use only part of the 32-bit register defined by the address offset. If a bit offset is explicitly declared, this specifies the starting (lowest) bit number from the 32-bit register that will be used. This must take a value of 0 – 31. If no value is declared here, the starting bit number is 0 (and all 32 bits are used).

If a bit offset has been explicitly declared, a bit width can also be declared, specifying the number of bits to be used in the definition, starting at the bit declared as the bit offset. If no bit width is declared after an offset has been declared, a width of 1 is used. The sum of the bit offset and the bit width must not exceed 32.

Even though this command permits multiple variables to take the same definition in a single command, this capability is seldom useful.

Examples

```
M0->u.io:$A00000.8.1          // Set def of M0 in I/O space, 1 bit starting at bit 8
M80->s.user:0                 // Set def of M80 in user buffer, 32-bit signed int
M5000->u.io:$A10000.8.8       // Set def of M5000 in I/O space, 8-bit unsigned int
```

M{variable list}->{data structure element}

Function: Set definition(s) of M-variable(s) in list with data structure element

Scope: Global

Syntax: **M*{variable list}->{data structure element}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- ***{data structure element}*** is the name of a pre-defined Power PMAC data structure element

The **M*{variable list}->{data structure element}*** command causes Power PMAC to set the definition of the M-variable(s) in the list to point to the specified data structure element. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

This type of M-variable definition permits a "shorthand" name for a data structure element, or a customized name when a variable name is declared or defined for the M-variable.

In this type of definition, the M-variable automatically takes on the format of the specified data structure element.

Even though this command permits multiple variables to take the same definition in a single command, this capability is seldom useful.

Examples

```
M0->Sys.ServoCount            // Set definition of M0 to global servo cycle counter
M162->Motor[1].ActPos         // Set definition of M162 to #1 actual position register
```

```
M5187->Coord[1].InPos          // Set definition of M5187 to &1 in-position bit
M15000->Sys.Ddata[L0]          // Set definition of M15000 to user buffer indexable
                               // location, used as double-precision float
```

M{variable list}->{self-referenced definition}

Function: Set definition of M-variable(s) in list to self-referenced definition

Scope: Global

Syntax: **M*{variable list}->{self-referenced definition}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- ***{self-referenced definition}*** consists of
  ****[{format}[.{width}]]***
  where:
  - ***{format}*** is a single letter representing the variable type (how the value at this address is to be interpreted). The choices are:
    - **s** specifies a signed integer of up to 32 bits that saturates (not rolls over) when a command attempts to assign a value greater than its range to it
    - **i** specifies a signed integer of up to 32 bits that rolls over (not saturates) when a command attempts to assign a value greater than its range to it
    - **u** specifies an unsigned integer of up to 32 bits that rolls over (not saturates) when a command attempts to assign a value greater than its range to it
    - **f** specifies a single-precision (32-bit) floating-point variable
    - **d** specifies a double-precision (64-bit) floating-point variable (actually the last bit of the mantissa is lost, so technically it is only 63 bits). If no format is specified, Power PMAC will use the **d** format.

- the optional ***{width}*** is an integer in the range 1 – 32 representing the number of bits to be used for the variable in the register at the specified address. This specification is valid only for integer variable forms. If no value is specified, a value of 32 is used for width.

The **M*{variable list}->{self-referenced definition}*** command causes Power PMAC to set the definition of the M-variable(s) in the list to use part of the definition register for the value of the variable. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

A self-referenced M-variable is a general-purpose global user variable of a specific format. It can be useful to add to the total number of global variables, or to create a variable with a format different from that of the global P-variables, which are all double-precision floating-point variables. It can also be useful to "clear" a variable's definition, so that assigning a value to this variable cannot write to anything harmful.

Because each variable's definition uses its own register in the self-referenced definition, it can be very useful to define multiple variables in a single command.

Examples

```
M70->*u                     // Set M70 as self-defined unsigned 32-bit integer
M80..89->*s.16              // Set M80 – M89 as self-defined 16-bit signed integers
M8000->*f                   // Set M8000 as self-defined 32-bit float
M10000..10999->*d           // Set M10000 – M10999 as self-defined 64-bit floats
M0..16383->*                // Clear all M-variable definitions
```

open forward

Function: Open forward-kinematic program buffer for entry

Scope: Communications-thread, coordinate-system specific

Syntax: **open forward*[,[{constant}]][,{constant}]***

where:

- the optional first ***{constant}*** is a positive integer that specifies the size of the local variable "stack offset" when this program calls a subroutine or subprogram; if no number is specified here, a value of 256 is used

- the optional second ***{constant}*** is a positive integer that specifies the maximum number of line jump labels that may be used in this program; if no number is specified here, a value of 1024 is used

The **open forward** command causes Power PMAC to open the forward-kinematic program buffer of the addressed coordinate system for entry of program commands on this communications thread. Subsequent program commands sent on this thread will be entered into the program buffer. When entry of

the program is finished, the **close** command should be used to prevent further command lines from being entered into the buffer, and to permit the program to be executed.

The forward-kinematic program for a coordinate system contains calculations for converting motor ("joint") coordinates to axis ("tool-tip") coordinates. This program provides more flexibility in this conversion than the scaling and offsets of axis-definition statements. If a forward-kinematic program exists in the Power PMAC for a coordinate system, then it will automatically be executed anytime a motor-to-axis conversion is required for that coordinate system (starting a motion program, a **pmatch** or **pset** command, a request for axis positions or velocities).

If there are no arguments after the program number, Power PMAC will use the default local-variable stack offset (256) in subroutine calls, and reserve the default table size for line jump labels (1024). If a number is declared after a first comma, the number specifies the size of the stack offset used when this program calls a subroutine (**R0** of the subroutine is equivalent to **L(StackOffset)** of this program). A larger number permits more arguments to be passed in a given subroutine call, but fewer levels of subroutines before the overall stack of 8192 variables is exhausted.

Note that if the subroutine is downloaded through the IDE project manager, the IDE will automatically declare the minimum stack offset that is sufficient to handle the number of local variables used.

If a number is declared after a second comma, the number specifies the maximum number of line jump labels that may be used in the program. A larger number requires more memory storage, and slightly more time in executing a jump.

Note that if the subroutine is downloaded through the IDE project manager, the IDE will automatically declare the minimum line-label buffer that is sufficient to handle the number of line jump labels used.

Opening a program buffer with this command automatically erases the contents of any existing program buffer of this number (there is no need for a "clear" command as in PMAC or Turbo PMAC). If a command line is sent on this thread while the buffer is open that is not valid for entry into this buffer (either due to illegal syntax, or syntax that is only valid as an on-line command), the command will be rejected with an error, the buffer contents will be erased, and the buffer will automatically be closed.

No other program buffers of any kind may be open on any thread when this command is sent, or Power PMAC will reject this command with an error. No motion programs may be running (even with execution suspended in a manner that permits re-starting at that point, as with hold, quit, or single-step commands), and no PLC programs may be enabled when this command is sent, or Power PMAC will reject this command with an error. (An **a** abort command can be given to fully exit an executing or suspended motion program.)

Examples

```
open forward                    // Open forward-kinematic program buffer for addressed
                        // C.S. default stack offset, default number of labels
&2open forward,64               // Open F.K program buffer for C.S. 2,
                                // stack offset of 64, default number of labels
&4open forward,,2048            // Open F.K. program buffer for C.S. 4,
                                // default stack offset, up to 2048 labels
&12open forward,512,256         // Open F.K. program buffer for C.S. 12,
                                // stack offset of 512, up to 256 labels
```

open inverse

Function: Open inverse-kinematic program buffer for entry

Scope: Communications-thread, coordinate-system specific

Syntax: **open inverse*[,[{constant}]][,{constant}]***

where:

- the optional first ***{constant}*** is a positive integer that specifies the size of the local variable "stack offset" when this program calls a subroutine or subprogram; if no number is specified here, a value of 256 is used

- the optional second ***{constant}*** is a positive integer that specifies the maximum number of line jump labels that may be used in this program; if no number is specified here, a value of 1024 is used

The **open inverse** command causes Power PMAC to open the inverse-kinematic program buffer of the addressed coordinate system for entry of program commands on this communications thread. Subsequent program commands sent on this thread will be entered into the program buffer. When entry of the program is finished, the **close** command should be used to prevent further command lines from being entered into the buffer, and to permit the program to be executed.

The inverse-kinematic program for a coordinate system contains calculations for converting axis ("tool-tip") coordinates to motor ("joint") coordinates. This program provides more flexibility in this conversion than the scaling and offsets of axis-definition statements. If any axis in the coordinate system is defined as an inverse-kinematic axis (**#*x*->I**), then the inverse-kinematic program for that coordinate system will automatically be executed anytime an axis-to-motor conversion is required for that coordinate system (once per programmed move for non-segmented moves, once per segment for segmented moves).

If there are no arguments after the program number, Power PMAC will use the default local-variable stack offset (256) in subroutine calls, and reserve the default table size for line jump labels (1024). If a number is declared after a first comma, the number specifies the size of the stack offset used when this program calls a subroutine (**R0** of the subroutine is equivalent to **L(StackOffset)** of this program). A larger number permits more arguments to be passed in a given subroutine call, but fewer levels of subroutines before the overall stack of 8192 variables is exhausted.

Note that if the subroutine is downloaded through the IDE project manager, the IDE will automatically declare the minimum stack offset that is sufficient to handle the number of local variables used.

If a number is declared after a second comma, the number specifies the maximum number of line jump labels that may be used in the program. A larger number requires more memory storage, and slightly more time in executing a jump.

Note that if the subroutine is downloaded through the IDE project manager, the IDE will automatically declare the minimum line-label buffer that is sufficient to handle the number of line jump labels used.

Opening a program buffer with this command automatically erases the contents of any existing program buffer of this number (there is no need for a "clear" command as in PMAC or Turbo PMAC). If a command line is sent on this thread while the buffer is open that is not valid for entry into this buffer

(either due to illegal syntax, or syntax that is only valid as an on-line command), the command will be rejected with an error, the buffer contents will be erased, and the buffer will automatically be closed.

No other program buffers of any kind may be open on any thread when this command is sent, or Power PMAC will reject this command with an error. No motion programs may be running (even with execution suspended in a manner that permits re-starting at that point, as with hold, quit, or single-step commands), and no PLC programs may be enabled when this command is sent, or Power PMAC will reject this command with an error. (An **a** abort command can be given to fully exit an executing or suspended motion program.)

Examples

```
open inverse                    // Open inverse-kinematic program buffer for addressed
                                // C.S. default stack offset, default number of labels
&2open inverse,64               // Open I.K program buffer for C.S. 2,
                                // stack offset of 64, default number of labels
&4open inverse,,2048            // Open I.K. program buffer for C.S. 4,
                                // default stack offset, up to 2048 labels
&12open inverse,512,256         // Open I.K. program buffer for C.S. 12,
                                // stack offset of 512, up to 256 labels
```

open plc

Function: Open PLC program buffer for entry

Scope: Communications-thread specific

Syntax: **open plc*{constant}*[,[*{constant}*]][,*{constant}*]**

where:

- the first ***{constant}*** is a positive integer in the range 0 to 31 specifying the number of the PLC program whose buffer is being opened
- the optional second ***{constant}*** is a positive integer that specifies the size of the local variable "stack offset" when this program calls a subroutine or subprogram; if no number is specified here, a value of 256 is used
- the optional third ***{constant}*** is a positive integer that specifies the maximum number of line jump labels that may be used in this program; if no number is specified here, a value of 1024 is used

The **open plc** command causes Power PMAC to open the specified PLC program buffer for entry of program commands on this communications thread. Subsequent program commands sent on this thread will be entered into the program buffer. When entry of the program is finished, the **close** command should be used to prevent further command lines from being entered into the buffer, and to permit the program to be executed.

If the program is downloaded through the IDE project manager, a text name can be used for the program (e.g. **open plc SafetyInterlocks**), and the IDE will automatically substitute a program number

for the text name. The text name can be used in all other references to the program (e.g. **enable plc SafetyInterlocks**).

If there are no arguments after the program number, Power PMAC will use the default local-variable stack offset (256) in subroutine calls, and reserve the default table size for line jump labels (1024). If a number is declared after a first comma, the number specifies the size of the stack offset used when this program calls a subroutine (**R0** of the subroutine is equivalent to **L(StackOffset)** of this program). A larger number permits more arguments to be passed in a given subroutine call, but fewer levels of subroutines before the overall stack of 8192 variables is exhausted.

Note that if the program is downloaded through the IDE project manager, the IDE will automatically declare the minimum stack offset that is sufficient to handle the number of local variables used.

If a number is declared after a second comma, the number specifies the maximum number of line jump labels that may be used in the program. A larger number requires more memory storage, and slightly more time in executing a jump.

Note that if the program is downloaded through the IDE project manager, the IDE will automatically declare the minimum line-label buffer that is sufficient to handle the number of line jump labels used.

Opening a program buffer with this command automatically erases the contents of any existing program buffer of this number (there is no need for a "clear" command as in PMAC or Turbo PMAC). If a command line is sent on this thread while the buffer is open that is not valid for entry into this buffer (either due to illegal syntax, or syntax that is only valid as an on-line command), the command will be rejected with an error, the buffer contents will be erased, and the buffer will automatically be closed.

The specified PLC program must be disabled (not just paused or in single-step mode) when this command is sent, or Power PMAC will reject this command with an error.

No other program buffers of any kind may be open on any thread when this command is sent, or Power PMAC will reject this command with an error.

Examples

```
open plc 0                    // Open PLC program buffer 0, default stack and labels
open plc 1,64                 // Open PLC program buffer 1,
                              // stack offset of 64, default number of labels
open plc 7,,2048              // Open PLC program buffer 7,
                              // default stack offset, up to 2048 labels
open plc 23,512,256           // Open PLC program buffer 23,
                              // stack offset of 512, up to 256 labels
```

open prog

Function: Open motion program buffer for entry

Scope: Communications-thread specific

Syntax: **open prog*{constant}*[,[*{constant}*]][,*{constant}*]**

where:

- the first ***{constant}*** is a non-negative 32-bit integer specifying the number of the motion program whose buffer is being opened

- the optional second ***{constant}*** is a positive integer that specifies the size of the local variable "stack offset" when this program calls a subroutine or subprogram; if no number is specified here, a value of 256 is used

- the optional third ***{constant}*** is a positive integer that specifies the maximum number of line jump labels that may be used in this program; if no number is specified here, a value of 1024 is used

The **open prog** command causes Power PMAC to open the specified motion program buffer for entry of program commands on this communications thread. Subsequent program commands sent on this thread will be entered into the program buffer. When entry of the program is finished, the **close** command should be used to prevent further command lines from being entered into the buffer, and to permit the program to be executed.

If the program is downloaded through the IDE project manager, a text name can be used for the program (e.g. **open prog CalibrationMoves**), and the IDE will automatically substitute a program number for the text name. The text name can be used in all other references to the program (e.g. **&1 start CalibrationMoves**).

The **open prog 0** command is equivalent to the open rot command, opening the rotary motion program buffer for the addressed coordinate system.

If there are no arguments after the program number, Power PMAC will use the default local-variable stack offset (256) in subroutine calls, and reserve the default table size for line jump labels (1024). If a number is declared after a first comma, the number specifies the size of the stack offset used when this program calls a subroutine (**R0** of the subroutine is equivalent to **L(StackOffset)** of this program). A larger number permits more arguments to be passed in a given subroutine call, but fewer levels of subroutines before the overall stack of 8192 variables is exhausted.

Note that if the program is downloaded through the IDE project manager, the IDE will automatically declare the minimum stack offset that is sufficient to handle the number of local variables used.

If a number is declared after a second comma, the number specifies the maximum number of line jump labels that may be used in the program. A larger number requires more memory storage, and slightly more time in executing a jump.

Note that if the program is downloaded through the IDE project manager, the IDE will automatically declare the minimum line-label buffer that is sufficient to handle the number of line jump labels used.

Opening a program buffer with this command automatically erases the contents of any existing program buffer of this number (there is no need for a "clear" command as in PMAC or Turbo PMAC). If a command line is sent on this thread while the buffer is open that is not valid for entry into this buffer (either due to illegal syntax, or syntax that is only valid as an on-line command), the command will be rejected with an error, the buffer contents will be erased, and the buffer will automatically be closed.

If this motion program already exists in Power PMAC memory, it must not be running (even with execution suspended in a manner that permits re-starting at that point, as with hold, quit, or single-step

commands) when this command is sent, or Power PMAC will reject this command with an error. (An **a** abort command can be given to fully exit an executing or suspended motion program.)

No other program buffers of any kind may be open on any thread when this command is sent, or Power PMAC will reject this command with an error.

Examples

```
open prog 1                          // Open motion program buffer 1,
                                     // default stack and labels
open prog 100,64                     // Open motion program buffer 100,
                                     // stack offset of 64, default number of labels
open prog 2001,,2048                 // Open motion program buffer 2001,
                                     // default stack offset, up to 2048 labels
open prog 5,512,256                  // Open motion program buffer 5,
                                     // stack offset of 512, up to 256 labels
```

open rotary

Function: Open a rotary motion program buffer for entry

Scope: Coordinate-system specific

Syntax: **open rotary**

The **open rotary** command causes Power PMAC to open the rotary motion program buffer for the addressed coordinate system for entry. Subsequent program commands valid for rotary motion programs sent on this port are entered into the rotary motion program buffer.

No other program buffers may be open when this command is sent. The rotary buffer must already have been defined for this coordinate system.

The rotary motion program buffer is considered motion program 0 for the coordinate system, so the command **open prog 0** has the same result.

Once the rotary buffer has been defined and opened, program commands that are sent with this coordinate system addressed are automatically entered into the rotary buffer.

The open rotary buffer will be closed for further entry when a **close** command is issued while this coordinate system is addressed.

open subprog

Function: Open subprogram buffer for entry

Scope: Communications-thread specific

Syntax: **open subprog***{constant}[,[{constant}]][,{constant}]*

where:

- the first ***{constant}*** is a positive 32-bit integer specifying the number of the subprogram whose buffer is being opened

- the optional second ***{constant}*** is a positive integer that specifies the size of the local variable "stack offset" when this program calls a subroutine or subprogram; if no number is specified here, a value of 256 is used

- the optional third ***{constant}*** is a positive integer that specifies the maximum number of line jump labels that may be used in this program; if no number is specified here, a value of 1024 is used

The **open subprog** command causes Power PMAC to open the specified subprogram buffer for entry of program commands on this communications thread. Subsequent program commands sent on this thread will be entered into the program buffer. When entry of the subprogram is finished, the **close** command should be used to prevent further command lines from being entered into the buffer, and to permit the program to be executed.

If the program is downloaded through the IDE project manager, a text name can be used for the program (e.g. **open subprog DrillCycle**), and the IDE will automatically substitute a program number for the text name. The text name can be used in all other references to the program (e.g. **call DrillCycle**).

If there are no arguments after the subprogram number, Power PMAC will use the default local-variable stack offset (256) in subroutine calls, and reserve the default table size for line jump labels (1024). If a number is declared after a first comma, the number specifies the size of the stack offset used when this program calls a subroutine (R0 of the subroutine is equivalent to **L(StackOffset)** of this program). A larger number permits more arguments to be passed in a given subroutine call, but fewer levels of subroutines before the overall stack of 8192 variables is exhausted.

Note that if the program is downloaded through the IDE project manager, the IDE will automatically declare the minimum stack offset that is sufficient to handle the number of local variables used.

If a number is declared after a second comma, the number specifies the maximum number of line jump labels that may be used in the program. A larger number requires more memory storage, and slightly more time in executing a jump.

Note that if the program is downloaded through the IDE project manager, the IDE will automatically declare the minimum line-label buffer that is sufficient to handle the number of line jump labels used.

Opening a subprogram buffer with this command automatically erases the contents of any existing program buffer of this number (there is no need for a "clear" command as in PMAC or Turbo PMAC). If a command line is sent on this thread while the buffer is open that is not valid for entry into this buffer (either due to illegal syntax, or syntax that is only valid as an on-line command), the command will be rejected with an error, the buffer contents will be erased, and the buffer will automatically be closed.

No other program buffers of any kind may be open on any thread when this command is sent, or Power PMAC will reject this command with an error. No motion programs may be running (even with execution suspended in a manner that permits re-starting at that point, as with hold, quit, or single-step commands) when this command is sent, or Power PMAC will reject this command with an error. (An **a** abort

command can be given to fully exit an executing or suspended motion program.) No PLC programs may be enabled when this command is sent, or Power PMAC will reject this command with an error.

No other program buffers of any kind may be open on any thread when this command is sent, or Power PMAC will reject this command with an error.

Examples

```
open subprog 1                      // Open motion program buffer 1,
                                    // default stack and labels
open subprog 100,64                 // Open motion program buffer 100,
                                    // stack offset of 64, default number of labels
open subprog 2001,,2048             // Open motion program buffer 2001,
                                    // default stack offset, up to 2048 labels
open subprog 5,512,256              // Open motion program buffer 5,
                                    // stack offset of 512, up to 256 labels
```

out*{constant}*

Function: Open-loop output

Scope: Motor specific

Syntax: **out*{constant}***

where:

- ***{constant}*** is a floating-point value specifying the output magnitude as a percentage of maximum for the motor servo loop.

The **out*{constant}*** command causes Power PMAC to put the specified motor(s) in open-loop enabled mode and force a servo-loop output of the specified magnitude, expressed as a percentage of the maximum output parameter for the motor: **Motor[*x*].MaxDac**. If not immediately preceded by a motor list, it will do so for the presently addressed motor. If immediately preceded by a motor list, it will do so for all motors in the list.

This command is commonly used for set-up and diagnostic purposes (for instance, a positive **out** command must cause position to count in the positive directions, or closed-loop control cannot be established), but it can also be used in actual applications.

If the motor is *not* commutated by Power PMAC, this command will create a constant value in the single output register for the motor (e.g. a DAC or pulse-frequency register). If the motor is commutated by Power PMAC, the commutation algorithm is still active, and the specified value is the quadrature (torque) command input to the commutation algorithm.

The **out*{constant}*** command in Power PMAC is the equivalent of the **o*{constant}*** command in the older PMAC and Turbo PMAC controllers.

If the value is specified outside the range +/-100.0, the output will saturate at +/-100% of the output limit value. No error will be reported.

p

Function: Report actual position value(s)

Scope: Motor or Coordinate-system specific

Syntax: **p**

The **p** command causes Power PMAC to report the present value of the actual position(s) for the specified motor(s) or coordinate system(s). If not immediately preceded by a motor list or coordinate system list, it will report the value of the actual position for the presently addressed motor. If immediately preceded by a motor list, it will report the values of the actual positions for all motors in the list. If immediately preceded by a coordinate-system list, it will report the values of the actual positions for all active axes for all coordinate systems in the list.

Note that specifying a list of multiple motors or multiple coordinate systems does not change the modally addressed motor or coordinate system for subsequent motor-specific or coordinate-system-specific commands.

When reporting motor positions, the positions are given in the base motor units. The reference ("zero") motor position for this reporting is dependent on two control settings that determine how the values in "offset-mode" command position registers are used. Offset-mode command position registers permit the superposition of several position command sources.

Motor[*x*].CompDesPos, which is typically the output of a cam table or a cam-style compensation table, is always an offset-mode register. **Motor[*x*].ActiveMasterPos**, which is typically the result of position following (electronic gearing), is an offset-mode register if bit 1 (value 2) of **Motor[*x*].MasterCtrl** is set to 1.

If **Motor[*x*].PosReportMode** is set to its default value of 0, or in an older firmware version that does not have this new element, the values in these offset-mode registers are subtracted from the net desired position. Working from the source registers for **Motor[*x*]**, the reported position is calculated as:

+ **Pos** // Raw feedback position relative to power-on/reset location
+ **CompPos** // Compensation table measurement correction
+ **PresBlSize** // Backlash correction
- **CompDesPos** // Compensation table command offset
- **ActiveMasterPos * MasterCtrl**OffsetBit // Master position when in offset mode
- **HomePos** // Motor zero position relative to power-on/reset location

If **Motor[*x*].PosReportMode** is set to 1, the values in these offset-mode registers are *not* subtracted from the net desired position. In this case, the reported position is calculated as:

+ **Pos** // Raw feedback position relative to power-on/reset location
+ **CompPos** // Compensation table measurement correction
+ **PresBlSize** // Backlash correction
- **HomePos** // Motor zero position relative to power-on/reset location

When reporting axis positions, the positions are given in the scaled user axis units, relative to the axis' programming origin (which is not necessarily the same as the zero position of the underlying motor). The reported value for each axis is preceded by the axis name (letter or double letter). Power PMAC leaves the values for each axis whose position is computed due to this command in local variable for the

communications thread L(256+**Sys.MaxMotors**+*n*), where *n* is the "axis index" value 0 to 31 (0 for A, 1 for B, etc.).

Because this command is processed in background, if the values for multiple motors or axes are requested, it is possible that the reported values will not all be from the same servo cycle.

If a report is requested of a coordinate system with no motors assigned to axes in that C.S., Power PMAC will return the string "`No Motors`". If the set of axis definitions or the forward kinematics subroutine for the C.S. does not permit the proper calculation of axis data, Power PMAC will return the string "`No Solution`".

Examples

```
p                              // Query actual position of presently addressed motor
1001                           // Power PMAC response in counts

#1..4p                         // Query actual positions of Motors 1 - 4
982 -3462 27 86643             // Power PMAC response in counts

&1p                            // Query actual positions of axes in C.S. 1
X0.982 Y-3.462 Z0.027 C86.643 // Power PMAC response in user units
```

P{data}

Function: Report value of P-variable

Scope: Global

Syntax: **P*{data}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **P*{data}*** command causes Power PMAC to report the present value of the specified P-variable. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 65,535 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                                  // Include variable names in query responses
P1                                     // Query value of variable P1
P1=17.5                                // Power PMAC response of the value of P1

P(P1)                                  // Query value of P-variable numbered by P1
P17=3                                  // Power PMAC response of the value of P17

P(P1+P17)                              // Query value of P-variable numbered by P1+P17
P20=-5.35                              // Power PMAC response of the value of P20

echo7                                  // Do not include variable names in query responses
P1                                     // Query value of variable P1
17.5                                   // Power PMAC response of the value of P1
```

P{data}={expression}

Function: Assign value to P-variable

Scope: Global

Syntax: ***P{data}={expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The ***P{data}={expression}*** command causes Power PMAC to set the specified P-variable to the value of the expression on the right side of the equals sign. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 65,535 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, the command will be entered into that buffer for later execution.

Examples

```
P1=17.5                          // Set variable P1 to 17.5
P(P1)=3                          // Set P-variable numbered by P1 to 3
P(P1+P17)=5*sqrt(P100)           // Set P-var numbered by (P1+P17) to expression value
```

P{variable list}

Function: Report value(s) of P-variable(s) in list

Scope: Global

Syntax: ***P{variable list}[:{constant}]***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).

  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).

- the optional ***{constant}*** is a positive integer specifying the number of reported values per response line. If no value is specified here, only one value per response line is reported. This can only be used if the list is specified by a consecutive range of numbers.

The **P*{variable list}*** command causes Power PMAC to report the present value(s) of the P-variable(s) in the list. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
P1                          // Query value of variable P1 (set quantity 1, spacing 1)
P1=17.5                     // Power PMAC response of the value of P1

P100..102                   // Query value of P100, P101, P102
P100=-4                     // Power PMAC response of the value of P100
P101=3.14159                // Power PMAC response of the value of P101
P102=0                      // Power PMAC response of the value of P102

P50,3                       // Query value of 3 P-vars, starting at P50 (spacing 1)
P50=-5.35                   // Power PMAC response of the value of P50
P51=7.12                    // Power PMAC response of the value of P51
P52=376452                  // Power PMAC response of the value of P52

P50,3,3                     // Query value of 3 P-vars, starting at P50, spacing 3
P50=-5.35                   // Power PMAC response of the value of P50
P53=99                      // Power PMAC response of the value of P53
P56=-0.002                  // Power PMAC response of the value of P56

P1..10:5                    // Query value of P1 - P10, 5 values per response line
P1=7,3,5,2,11               // Power PMAC response of the value of P1 - P5
P6=8,33,4,9,-2              // Power PMAC response of the value of P6 - P10
```

P{variable list}={expression}

Function: Set value(s) of P-variable(s) in list

Scope: Global

Syntax: ***P{variable list}={expression}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The ***P{variable list}={expression}*** command causes Power PMAC to set the value(s) of the P-variable(s) in the list to the value of the expression on the right side of the equals sign. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Examples

```
P1=17.5                     // Set value of P1 (set quantity 1, spacing 1) to 17.5
P100..102=P1+5              // Set value of P100, P101, P102 to (P1+5)
P50,3=-7                    // Set value of 3 P-vars, starting at P50 (spacing 1) to -7
P50,3,3=sqrt(P25)           // Set value of 3 P-vars, starting at P50, spacing 3
                            // (P50, P53, P56) to the square root of P25
```

pause

Function: Quit motion program execution

Scope: Coordinate-system specific

Syntax: **q**, **pause**

The **pause** command is the "long-form" version of the **q** command, causing Power PMAC to cease motion program execution for the specified coordinate system(s) at the end of the last calculated move. If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

pause plc

Function: Pause execution of specified PLC program(s)

Scope: Global

Syntax: **pause plc** ***{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the PLC programs whose execution is to be paused.

The on-line **pause plc** command causes Power PMAC to stop the execution of the specified PLC program(s) by inhibiting the start of subsequent scans. Execution can be re-started at the point where it was halted (with the **resume plc** command, even if this command halted execution in the middle of the program (e.g. the program was stuck inside a **while** loop).

The similar **disable plc** command can be used to stop PLC program execution in a way that only permits re-starting at the beginning of the program, regardless of where execution was halted.

If a PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other exiting PLC programs specified in the command.

Examples

```
pause plc 1                  // Pause execution of PLC 1
pause plc 2,4,6              // Pause execution of PLCs 2, 4, and 6
pause plc 7..10              // Pause execution of PLCs 7, 8, 9, and 10
pause plc 11,13..16,20       // Pause execution of PLCs 11, 13, 14, 15, 16, and 20
```

pmatch

Function: Re-match axis positions to motor positions

Scope: Coordinate-system specific

Syntax: **pmatch**

The **pmatch** command causes Power PMAC to recalculate the axis starting positions for the specified coordinate system(s) to match the present motor commanded positions. It does this by inverting the axis-definition statement equations and solving for axis position, or if there is a forward-kinematic subroutine for the coordinate system, by executing that subroutine.

If not immediately preceded by a coordinate system list, it will do this matching for the presently addressed coordinate system. If immediately preceded by a coordinate system list, it will do this matching for all coordinate systems in the list.

This matching function is automatically executed (without an explicit **pmatch** command) by Power PMAC each time execution of a motion program is started with a command, to make sure the first move is calculated correctly.

However, the Power PMAC does not automatically execute this function before an axis move commanded from a program where the top-level program is a PLC program. If there is a chance that any of the motors in the coordinate system have moved since the last commanded axis move, or the axis/motor relationship has changed since the last commanded axis move, the **pmatch** command must be given first to cause Power PMAC to compute the correct axis starting positions.

The **pmatch** command can also be executed directly as a buffered program command in motion programs, PLC programs, or subprograms called from either type of program. (There is no need to use the **cmd"pmatch"** technique, as was required in older PMACs.)

If an axis move is commanded when the axis and motor positions do not agree according to the present axis/motor relationship, Power PMAC will use the wrong axis starting point in the computation of the move, and there will be a sudden commanded step to this starting point at the beginning of the move.

q

Function: Quit motion program execution

Scope: Coordinate-system specific

Syntax: **q**, **pause**

The **q** command causes Power PMAC to cease motion program execution for the specified coordinate system(s) at the end of the last calculated move. If not immediately preceded by a coordinate system list, it will cease program execution for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will cease program execution for all coordinate systems in the list.

If the coordinate system is looking ahead in the motion program because it is in a sequence of blended moves, one or more moves after the presently executing move may have already been calculated and placed in the motion queue. These moves will be executed before the stop from a **q** command.

The motion program execution is suspended while stopped on this command, but technically it is still in the program, and execution can be resumed at this point. If it is subsequently desired that program execution will not be resumed, program execution should be fully aborted with the **a** command.

The short form of this command (**q**) is useful for typing in terminal mode. The long form (**pause**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**pause**) must be used. The short form (**q**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

The equivalent buffered direct program command is **pause**.

Examples

```
q                                   // Quit program execution in addressed C.S.
&2q                                 // Address C.S.2 and quit program  execution there
&1,3,5q                             // Quit program execution in C.S. 1, 3, and 5
&*q                                 // Quit program execution in all C.S.
```

Q{data}

Function: Report value of Q-variable

Scope: Coordinate-system specific

Syntax: **Q*{data}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **Q*{data}*** command causes Power PMAC to report the present value of the specified Q-variable for the addressed coordinate system (&n). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
&1                                  // Modally address C.S.1, will access C.S.1's Q-vars
echo0                               // Include variable names in query responses
Q1                                  // Query value of variable Q1
Q1=17.5                             // Power PMAC response of the value of Q1

Q(P1)                               // Query value of Q-variable numbered by P1
Q17=3                               // Power PMAC response of the value of Q17
```

```
Q(P1+P17)                          // Query value of Q-variable numbered by P1+P17
Q20=-5.35                          // Power PMAC response of the value of Q20

echo7                              // Do not include variable names in query responses
Q1                                 // Query value of variable Q1
17.5                               // Power PMAC response of the value of Q1
```

Q{data}={expression}

Function: Assign value to Q-variable

Scope: Coordinate-system specific

Syntax: ***Q{data}={expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The ***Q{data}={expression}*** command causes Power PMAC to set the specified Q-variable in the addressed coordinate system (&n) to the value of the expression on the right side of the equals sign. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, the command will be entered into that buffer for later execution.

Examples

```
&3                          // Modally address C.S.3, will access C.S.3's Q-vars
Q1=17.5                     // Set variable Q1 to 17.5
Q(P1)=3                     // Set Q-variable numbered by P1 to 3
Q(P1+P17)=5*sqrt(P100)      // Set Q-var numbered by (P1+P17) to expression value
```

Q{variable list}

Function: Report value(s) of Q-variable(s) in list

Scope: Coordinate-system specific

Syntax: ***Q{variable list}[:{constant}]***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- the optional ***{constant}*** is a positive integer specifying the number of reported values per response line. If no value is specified here, only one value per response line is reported. This can only be used if the list is specified by a consecutive range of numbers.

The **Q*{variable list}*** command causes Power PMAC to report the present value(s) of the Q-variable(s) for the addressed coordinate system (&n) in the list. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
&2                          // Modally address C.S.2, will access C.S.2's Q-vars
```

```
Q1                          // Query value of variable Q1 (set quantity 1, spacing 1)
Q1=17.5                     // Power PMAC response of the value of Q1

Q100..102                   // Query value of Q100, P101, P102
Q100=-4                     // Power PMAC response of the value of Q100
Q101=3.14159                // Power PMAC response of the value of Q101
Q102=0                      // Power PMAC response of the value of Q102

Q50,3                       // Query value of 3 Q-vars, starting at Q50 (spacing 1)
Q50=-5.35                   // Power PMAC response of the value of Q50
Q51=7.12                    // Power PMAC response of the value of Q51
Q52=376452                  // Power PMAC response of the value of Q52

Q50,3,3                     // Query value of 3 Q-vars, starting at Q50, spacing 3
Q50=-5.35                   // Power PMAC response of the value of Q50
Q53=99                      // Power PMAC response of the value of Q53
Q56=-0.002                  // Power PMAC response of the value of Q56

Q1..10:5                    // Query value of Q1 - Q10, 5 values per response line
Q1=7,3,5,2,11               // Power PMAC response of the value of Q1 - Q5
Q6=8,33,4,9,-2              // Power PMAC response of the value of Q6 - Q10
```

Q{variable list}={expression}

Function: Set value(s) of Q-variable(s) in list

Scope: Coordinate-system specific

Syntax: ***Q{variable list}={expression}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The ***Q{variable list}={expression}*** command causes Power PMAC to set the value(s) of the Q-variable(s) for the addressed coordinate system (&n) in the list to the value of the expression on the right side of the equals sign. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the

set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Examples

```
&4                          // Modally address C.S.4, will access C.S.4's Q-vars
Q1=17.5                     // Set value of Q1 (set quantity 1, spacing 1) to 17.5
Q100..102=P1+5              // Set value of Q100, Q101, Q102 to (P1+5)
Q50,3=-7                    // Set value of 3 Q-vars, starting at Q50 (spacing 1) to -7
Q50,3,3=sqrt(P25)           // Set value of 3 Q-vars, starting at Q50, spacing 3
                            // (Q50, Q53, Q56) to the square root of P25
```

r

Function: Run motion program

Scope: Coordinate-system specific

Syntax: **r**, **run**

The **r** command causes Power PMAC to start continuous motion program execution for the specified coordinate system(s) at the present point of the coordinate-system program counter. If not immediately preceded by a coordinate system list, it will start program execution for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will start program execution for all coordinate systems in the list.

Initial setting of the program counter is usually done with the **b*{constant}*** command. If program execution has been suspended, as with the **s**, **q**, or **h** command, execution will resume at the point where it was stopped.

The coordinate system must be in a proper condition in order for Power PMAC to accept this command for the coordinate system. Otherwise, the Power PMAC will reject this command with an error. The following conditions must hold for the coordinate system for this command to be accepted:

- The selected motion program (and any label selected) must be present and valid.
- All motors assigned to position axes in the coordinate system must be activated (**Motor[*x*].ServoCtrl** > 0).
- All motors assigned to position axes in the coordinate system must be enabled and closed-loop.
- No motor assigned to a position axis in the coordinate system may have both overtravel limits set.
- If axis-definition statements are used, Power PMAC must be able to compute all starting axis positions from present motor positions (**Coord[*x*].Csolve** = 1). If kinematic definitions are used (**#x->I**), valid kinematic subroutines for the coordinate system must be present.

- If **Coord[*x*].HomeRequired** = 1, all motors assigned to position axes in the coordinate system must have established a position reference through a successful homing search move or absolute position read.

- If re-starting from a suspended state that is not at a programmed point, all motors must be at the same commanded position as they were when initially stopped in the suspended state.

While it is possible to execute a motion program with no motors assigned to position axes in the coordinate system, it will execute in a very fast "dry run" mode.

The short form of this command (**r**) is useful for typing in terminal mode. The long form (**run**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**run**) must be used. The short form (**r**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

The equivalent buffered direct program command is **run**.

Examples

```
r                                   // Start program execution in addressed C.S.
b5r                                 // Point to beginning of Prog 5 and start program
                                    // execution in addressed C.S.
&2r                                 // Address C.S.2 and start program execution there
&1,3,5r                             // Start program execution in C.S. 1, 3, and 5
&*r                                 // Start program execution in all C.S.
```

R*{data}*

Function: Report value of R-variable

Scope: Communications-thread specific

Syntax: **R*{data}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable

The **R*{data}*** command causes Power PMAC to report the present value of the specified R-variable local to this communications thread. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

Each communications thread has its own set of R-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of R-variables are independent of each other.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
echo0                                   // Include variable names in query responses
R1                                      // Query value of variable R1
R1=17.5                                 // Power PMAC response of the value of R1

R(P1)                                   // Query value of R-variable numbered by P1
R17=3                                   // Power PMAC response of the value of R17

R(P1+P17)                               // Query value of R-variable numbered by P1+P17
R20=-5.35                               // Power PMAC response of the value of R20

echo7                                   // Do not include variable names in query responses
R1                                      // Query value of variable R1
17.5                                    // Power PMAC response of the value of R1
```

R*{data}={expression}*

Function: Assign value to R-variable

Scope: Communications-thread specific

Syntax: **R*{data}={expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the variable
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **R*{data}={expression}*** command causes Power PMAC to set the specified R-variable local to this communications thread to the value of the expression on the right side of the equals sign. If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, the command will be entered into that buffer for later execution.

Each communications thread has its own set of R-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of R-variables are independent of each other.

Examples

```
R1=17.5                          // Set variable R1 to 17.5
R(P1)=3                          // Set R-variable numbered by P1 to 3
R(P1+P17)=5*sqrt(P100)           // Set R-var numbered by (P1+P17) to expression value
```

R{variable list}

Function: Report value(s) of R-variable(s) in list

Scope: Communications-thread specific

Syntax: **R*{variable list}*[:*{constant}*]**

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}*[,*{constant}*[,*{constant}*]]** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- the optional ***{constant}*** is a positive integer specifying the number of reported values per response line. If no value is specified here, only one value per response line is reported. This can only be used if the list is specified by a consecutive range of numbers.

The **R*{variable list}*** command causes Power PMAC to report the present value(s) of the R-variable(s) in the list local to this communications thread. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Each communications thread has its own set of R-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of R-variables are independent of each other.

Bit 1 (value 2) of the echo mode parameter for the communications thread, set by the **echo*{constant}*** command, determines whether the name of the variable will be included in the response or not. If the bit is 0, the name is included; if the bit is 1, the name is not included.

Examples

```
R1                          // Query value of variable R1 (set quantity 1, spacing 1)
R1=17.5                     // Power PMAC response of the value of R1

R100..102                   // Query value of R100, R101, R102
R100=-4                     // Power PMAC response of the value of R100
R101=3.14159                // Power PMAC response of the value of R101
R102=0                      // Power PMAC response of the value of R102

R50,3                       // Query value of 3 R-vars, starting at R50 (spacing 1)
R50=-5.35                   // Power PMAC response of the value of R50
R51=7.12                    // Power PMAC response of the value of R51
R52=376452                  // Power PMAC response of the value of R52

R50,3,3                     // Query value of 3 R-vars, starting at R50, spacing 3
R50=-5.35                   // Power PMAC response of the value of R50
R53=99                      // Power PMAC response of the value of R53
R56=-0.002                  // Power PMAC response of the value of R56

R1..10:5                    // Query value of R1 - R10, 5 values per response line
R1=7,3,5,2,11               // Power PMAC response of the value of R1 - R5
R6=8,33,4,9,-2              // Power PMAC response of the value of R6 - R10
```

R*{variable list}={expression}*

Function: Set value(s) of R-variable(s) in list

Scope: Communications-thread specific

Syntax: **R*{variable list}={expression}***

where:

- ***{variable list}*** specifies a set of one or more variables. The list can take one of two forms:
  - ***{constant}..{constant}*** specifies a range of consecutively numbered variables, starting with the variable numbered by the first integer value, and ending with the variable numbered by the second integer value (which must be greater than the first).
  - ***{constant}[,{constant}[,{constant}]]*** specifies a set of variables, starting with the variable numbered by the first integer value, where the optional second integer value specifies the quantity of variables in the set (the quantity is 1 if not specified explicitly), and the optional third integer value specifies the spacing in number between members of the set (the spacing is 1 if not specified explicitly).
- ***{expression}*** is a mathematical expression containing the value that is to be assigned to the specified variable

The **R*{variable list}={expression}*** command causes Power PMAC to set the value(s) of the R-variable(s) in the list local to this communications thread to the value of the expression on the right side of the equals sign. The list can either be a set of consecutively numbered variables, specified by the numbers of the starting and ending variables, or it can be a set of evenly spaced variables, specified by the starting variable number, the quantity of variables in the set, and the spacing between members of the set. All variable numbers specified in the list must reference valid variables, or the entire command will be rejected with an error.

If a program buffer is open when this command is sent to Power PMAC, and there is more than one variable in the list, the command will be rejected with an error. If there is only a single variable in the list, the command will be entered into that buffer for later execution.

Each communications thread has its own set of R-variables. In addition, each coordinate system has its own set for use by motion programs, and each PLC has its own set. All of these sets of R-variables are independent of each other.

Examples

```
R1=17.5                     // Set value of R1 (set quantity 1, spacing 1) to 17.5
R100..102=P1+5              // Set value of R100, R101, R102 to (P1+5)
R50,3=-7                    // Set value of 3 R-vars, starting at R50 (spacing 1) to -7
R50,3,3=sqrt(P25)           // Set value of 3 R-vars, starting at R50, spacing 3
                            // (R50, R53, R56) to the square root of P25
```

reboot

Function: Full controller and computer restart

Scope: Global

Syntax: **reboot**

The **reboot** command causes the Power PMAC computer to do a full restart. The effect on the computer is equivalent to cycling power off, then on. The action includes reloading the operating system from flash memory into active RAM. The action on the control application in Power PMAC is equivalent to that of the **$$$** reset command.

The Power PMAC will force its interface ASICs into "reset mode" at the beginning of execution of this command, forcing all output values to zero. It will release them into "operating mode" as it starts to reload their saved configuration values near the end of the operation.

In executing this command, Power PMAC copies the last saved values of the controller configuration – setup data structure elements, tables, programs, etc. – from non-volatile flash memory to active memory (RAM). *This means that any configuration information in Power PMAC's active memory that was not saved to flash memory will be lost when this command is executed.*

Note that this command causes the copying of the last saved values of the saved setup elements into the active registers first, and then executes the commands in the saved project files. It is possible for on-line commands in these files that write to the setup elements to cause the overwriting of saved values of the setup elements themselves.

Because this command immediately causes the Power PMAC application to enter its power-up/reset cycle, it provides no acknowledging characters to this command.

resetverbose

Function: Take Power PMAC out of "verbose" response mode

Scope: Global

Syntax: **resetverbose**

The **resetverbose** command takes Power PMAC out of the "verbose" response mode and returns it to the standard response mode. Verbose response mode is a debugging mode in which Power PMAC reports individual steps of complex operations such as the **save** command. It is seldom used in actual applications.

Power PMAC is put into verbose reponse mode using the **setverbose** on-line command.

resume plc

Function: Resume execution of specified PLC program(s)

Scope: Global

Syntax: **resume plc** ***{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the PLC programs whose execution is to be resumed.

The **resume plc** command causes Power PMAC to permit the execution of the specified PLC program(s) at their normal priority and timing. Execution will start at the point where execution was halted, even if it was halted in the middle of the program (e.g. with a **pause plc** command).

The similar **enable plc** command can be used to re-start PLC program execution at the beginning of the program, even if execution was halted elsewhere.

If a PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other exiting PLC programs specified in the command.

Examples

```
resume plc 1                  // Resume execution of PLC 1
resume plc 2,4,6              // Resume execution of PLCs 2, 4, and 6
resume plc 7..10              // Resume execution of PLCs 7, 8, 9, and 10
resume plc 11,13..16,20       // Resume execution of PLCs 11, 13, 14, 15, 16, and 20
```

rotfree

Function: Report size of largest block of available memory in rotary buffer

Scope: Coordinate-system specific

Syntax: **rotfree**

The **rotfree** command causes Power PMAC to report the number of bytes in the largest block of available memory in the rotary motion program buffer for the specified coordinate system(s). If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

Available memory in the buffer consists either of memory that has not yet been written to with downloaded program lines or memory with program lines that have already been executed by the Power PMAC.

If the storage pointer for the buffer is closer to the end of the buffer than the execution pointer, there will usually be two blocks of available memory –from the beginning of the buffer to the execution pointer, and from the storage pointer to the end of the buffer. The response to the **rotfree** command will report the size of the larger of these two blocks. (The related **rotfreeall** command, new in V2.1 firmware, will report the size of both blocks.)

However, if the storage pointer is closer to the start of the buffer than the execution pointer, because it has wrapped around one more time than the execution pointer, there will only be one block of available memory – from the storage pointer to the execution pointer.

A single program line must be stored in a continuous block of memory, so the response to the **rotfree** command reports the size of the longest program line that can be downloaded to the buffer.

Each letter/number combination (e.g. **X10**) in the program occupies 9 bytes of memory. The end-of-line indicator occupies 1 byte. So a basic 3-axis command line (e.g. **X10 Y20 Z30**) would occupy 9*3 + 1, or 28 bytes of memory.

Many users will prefer to use the "**N**" synchronous program line labels in their rotary programs and monitor **Coord[*x*].Ncalc** and/or **Coord[*x*].Nsync** to decide when more program lines need to be downloaded.

rotfreeall

Function: Report size of blocks of available memory in rotary buffer

Scope: Coordinate-system specific

Syntax: **rotfreeall**

The **rotfreeall** command causes Power PMAC to report the number of bytes in the block or blocks of available memory in the rotary motion program buffer for the specified coordinate system(s). If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

Available memory in the buffer consists either of memory that has not yet been written to with downloaded program lines or memory with program lines that have already been executed by the Power PMAC.

If the storage pointer for the buffer is closer to the end of the buffer than the execution pointer, there will usually be two blocks of available memory – from the beginning of the buffer to the execution pointer, and from the storage pointer to the end of the buffer The response to the **rotfreeall** command will report the size of each of these blocks, separated by the + sign. (The related **rotfree** command will report only the size of the larger block.)

However, if the storage pointer is closer to the start of the buffer than the execution pointer, because it has wrapped around one more time than the execution pointer, there will only be one block of available memory – from the storage pointer to the execution pointer. The response to the **rotfreeall** command will report this single value.

A single program line must be stored in a continuous block of memory, so the response to the **rotfreeall** command provides both the size of the longest program line that can be downloaded to the buffer and the total available memory in the buffer.

The **rotfreeall** command is new in V2.1 firmware, released 1st quarter 2016.

run

Function: Run motion program

Scope: Coordinate-system specific

Syntax: **r**, **run**

The **run** command is the "long-form" version of the **r** command, causing Power PMAC to start continuous motion program execution for the specified coordinate system(s) at the present point of the coordinate-system program counter. If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

s

Function: Execute single step of motion program

Scope: Coordinate-system specific

Syntax: **s**, **step**

The **s** command causes Power PMAC to start single-step motion program execution for the specified coordinate system(s) at the present point of the coordinate-system program counter. If not immediately preceded by a coordinate system list, it will start program execution for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will start program execution for all coordinate systems in the list. The exact behavior of the coordinate system in response to this command is dependent on the setting of saved setup element **Coord[*x*].StepMode**.

If the coordinate system is already executing a motion program when this command is sent, the command puts the program in single-step mode. If **Coord[*x*].StepMode** is set to 0, one more move will be calculated, and execution will stop at the end of this next move. In this case, its action is similar to the **q** command, although it will execute an additional move compared to **q**. If **Coord[*x*].StepMode** is set to a value greater than 0, no more moves will be calculated, and execution will stop at the end of the most recently calculated move.

If program execution has already stopped in single-step mode, a step command will cause program execution to advance until one move is executed if **Coord[*x*].StepMode** is set to 0. It will cause one program line to be executed if **Coord[*x*].StepMode** is set to a value greater than 0. (The different non-zero values of **Coord[*x*].StepMode** lead to slightly different behavior when 2D tool-radius compensation is enabled.)

Regardless of the setting of **Coord[*x*].StepMode**, if single-step execution encounters a **bstart** command in the motion program, it will continue until (and only until) the next **bstop** command in the motion program, regardless of how many moves or program lines are included.

Initial setting of the program counter is usually done with the **b*{constant}*** command. If program execution has been suspended, as with the **s**, **q**, or **h** command, execution will resume at the point where it was stopped.

If program execution encounters a **bstart** (block start) command, single-step execution will continue until a **bstop** (block stop) command is encountered, no matter how many program lines or move commands are in between, or what the setting of **Coord[*x*].StepMode** is.

The motion program execution is suspended while stopped after this command, but technically it is still in the program, and execution can be resumed at this point. If it is subsequently desired that program execution will not be resumed, program execution should be fully aborted with an **a** (abort) command.

The coordinate system must be in a proper condition in order for Power PMAC to accept this command for the coordinate system. Otherwise, the Power PMAC will reject this command with an error. The following conditions must hold for the coordinate system for this command to be accepted:

- The selected motion program (and any label selected) must be present and valid.
- All motors assigned to position axes in the coordinate system must be activated (**Motor[*x*].ServoCtrl** > 0).
- All motors assigned to position axes in the coordinate system must be enabled and closed-loop.
- No motor assigned to a position axis in the coordinate system may have both overtravel limits set.
- If axis-definition statements are used, Power PMAC must be able to compute all starting axis positions from present motor positions (**Coord[*x*].Csolve** = 1). If kinematic definitions are used (**#x->I**), valid kinematic subroutines for the coordinate system must be present.
- If **Coord[*x*].HomeRequired** = 1, all motors assigned to position axes in the coordinate system must have established a position reference through a successful homing search move or absolute position read.
- If re-starting from a suspended state that is not at a programmed point, all motors must be at the same commanded position as they were when initially stopped in the suspended state.

While it is possible to execute a motion program with no motors assigned to position axes in the coordinate system, it will execute in a very fast "dry run" mode.

The short form of this command (**s**) is useful for typing in terminal mode. The long form (**step**) is desirable for clarity and self-documentation when the command is issued from a program.

If saved setup element **Sys.NoShortCmds** is set to 1, the long form of this command (**step**) must be used. The short form (**s**) will be rejected as an illegal command (Error 20). This setting minimizes the chance of issuing this command mistakenly.

The equivalent buffered direct program command is **step**.

Examples

```
s                           // Start single-step execution in addressed C.S.
```

```
b5s                          // Point to beginning of Prog 5 and start single-step
                             // execution in addressed C.S.
&2s                          // Address C.S.2 and start single-step execution there
&1,3,5s                      // Start single-step execution in C.S. 1, 3, and 5
&*s                          // Start single-step execution in all C.S.
```

save

Function: Copy setup configuration to non-volatile memory

Scope: Global

Syntax: **save**

The **save** command causes Power PMAC to copy the setup configuration and project files from active memory (RAM and ASIC memory-mapped registers) to non-volatile memory (flash), so this information can be retained through power-down or reset. On the next power-up or reset, this information is copied back from non-volatile memory to active memory.

For programs to be stored in flash memory in a **save** command, they must have been downloaded to active memory in Power PMAC as part of a file (the IDE project download does this automatically), as it is the file structure in RAM that is copied to flash memory.

Starting in V2.0 firmware (released 1st quarter 2015), if there is not sufficient space in flash memory to store the configuration and project, the command will be rejected with an error (`Out of disk space. Save may have failed. Space available = ... Space required = ...`) and the global status bit **Sys.FlashSizeErr** will be set to 1.

setverbose

Function: Put Power PMAC into "verbose" response mode

Scope: Global

Syntax: **setverbose**

The **setverbose** command puts Power PMAC into the "verbose" response mode. Verbose response mode is a debugging mode in which Power PMAC reports individual steps of complex operations such as the **save** command. It is seldom used in actual applications.

Power PMAC is taken out of verbose reponse mode and returned to the standard response mode using the **resetverbose** on-line command.

Example

```
setverbose                                    // Put into verbose response mode
save                                          // Store project to flash memory
Changing echoMode to zero during Save operation
Successful: SaveConfiguration using /var/ftp/usrflash/Project/Configuration/pp_save.cfg

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

SaveToFlash: Do NOT Power off until Finished!!!

SaveToFlash: cp

SaveToFlash: sync()

SaveToFlash: mount

SaveToFlash: Finish SAVING to Flash.

Save Complete

Restoring echoMode to original 0 value

resetverbose                                  // Take out of verbose response mode
save                                          // Store project to flash memory
Changing echoMode to zero during Save operation
Save Complete

Restoring echoMode to original 0 value
```

size

Function: Report total buffer memory size

Scope: Global

Syntax: **size**

The **size** command causes Power PMAC to report the total number of bytes of memory present, used or unused, in several user memory buffers in RAM. The related **free** command returns the number of still

unused bytes of memory for each of these memory buffers. The buffer sizes are set by directives in the `pp_proj.ini` file of the IDE project manager.

Example

```
size                                        // Query total buffer memory
Program Buffer = 16777216                   // Power PMAC response
User Buffer = 1048576
Table Buffer = 1048576
Lookahead Buffer = 16777216
```

start[{constant}]

Function: Point program counter to specified motion program and run

Scope: Coordinate-system specific

Syntax: **start*[{constant}]***

where:

- ***{constant}*** is a non-negative integer specifying the motion program number

The **start*[{constant}]*** command causes the specified coordinate system(s) to set their program counter(s) to the beginning of the present (if no number given) or specified motion program and begin continuous execution of that program. If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

The **start*[{constant}]*** command is equivalent to the command sequence **b*[{constant}]*r** (begin and run).

The coordinate system must be in a proper condition in order for Power PMAC to accept this command for the coordinate system. Otherwise, the Power PMAC will reject this command with an error. The following conditions must hold for the coordinate system for this command to be accepted:

- The selected motion program (and any label selected) must be present and valid.
- All motors assigned to position axes in the coordinate system must be activated (**Motor[*x*].ServoCtrl** > 0).
- All motors assigned to position axes in the coordinate system must be enabled and closed-loop.
- No motor assigned to a position axis in the coordinate system may have both overtravel limits set.
- If axis-definition statements are used, Power PMAC must be able to compute all starting axis positions from present motor positions (**Coord[*x*].Csolve** = 1). If kinematic definitions are used (**#x->I**), valid kinematic subroutines for the coordinate system must be present.

- If **Coord[*x*].HomeRequired** = 1, all motors assigned to position axes in the coordinate system must have established a position reference through a successful homing search move or absolute position read.

- If re-starting from a suspended state that is not at a programmed point, all motors must be at the same commanded position as they were when initially stopped in the suspended state.

While it is possible to execute a motion program with no motors assigned to position axes in the coordinate system, it will execute in a very fast "dry run" mode.

step

Function: Execute single step of motion program

Scope: Coordinate-system specific

Syntax: **s**, **step**

The **step** command is the "long-form" version of the **s** command, causing Power PMAC to start single-step motion program execution for the specified coordinate system(s) at the present point of the coordinate-system program counter. If **Sys.NoShortCmds** is set to its default value of 0, either form of the command can be used. If **Sys.NoShortCmds** is set to 1, only this long-form version can be used.

For details on this command, refer to the description of the short-from version of the command.

step plc

Function: Execute single step of specified PLC program(s)

Scope: Global

Syntax: **step plc** ***{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the PLC programs whose execution is to be single-stepped.

The **step plc** command causes Power PMAC to permit the "single-step" execution of the specified PLC program(s). If execution of a specified PLC had been halted, execution will start at the point where execution was halted, even if it was halted in the middle of the program (e.g. with a **pause plc** command). If a specified PLC program was in continuous execution mode at the time of the command, the continuous execution will be stopped.

In single-step mode, only the commands on a single program line will be executed in response to a command. Otherwise, execution will be paused. Single-step mode is typically used for debugging purposes.

The similar **resume plc** command can be used to re-start continuous PLC program execution at the point where execution was halted.

If a PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other exiting PLC programs specified in the command.

Examples

```
step plc 1                    // Execute single line of PLC 1
step plc 2,4,6                // Execute single line of PLCs 2, 4, and 6
step plc 7..10                // Execute single line of PLCs 7, 8, 9, and 10
step plc 11,13..16,20         // Execute single line of PLCs 11, 13, 14, 15, 16, and 20
```

stop

Function: Stop motion program execution, reset program counter to top

Scope: Coordinate-system specific

Syntax: **stop**

The **stop** command causes Power PMAC to halt motion program execution for the specified coordinate system(s), permitting any moves already calculated by the program to finish. It also causes the coordinate-system program counter to be reset to the beginning of the (top-level) motion program. If not immediately preceded by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

string*{constant}*

Function: Report contents of specified text string

Syntax: **string*{constant}***

where:

- ***{constant}*** is non-negative integer specifying the byte offset from the start of user shared memory of the first character in the string to be reported.

The **string*{constant}*** command causes the Power PMAC to report the text contents of the present value of the specified string variable. The variable starts at the specified offset from the beginning of the user shared memory buffer, and continues until a "null" terminating character (byte value of 0) is encountered. The value is reported as ASCII text.

Note that unlike most other query commands, the Power PMAC response does *not* automatically add a carriage-return character to the end of the text response.

Example

```
string250                          // Query string starting at 250
```

```
Overpressure Error                    // Power PMAC response
```

t

Function: Report axis target positions of presently executing move

Scope: Coordinate-system specific

Syntax: **t**

The **t** command causes Power PMAC to report the axis target positions for the presently executing move. If not immediately preceded by a coordinate system list, it will report the target positions for the presently addressed coordinate system. If immediately preceded by a coordinate system list, it will report the target positions for all coordinate systems in the list.

Target position buffering must be enabled by setting saved setup element **Coord[*x*].TPSize** greater than 0, and to a value sufficient to store positions for all of the moves between move calculation time and move execution time. The positions will be reported only for those axes specified by saved setup element **Coord[*x*].TPCoords**.

The target positions will be reported in text form, with the axis letter name followed by the numerical value. The positions will be reported as programmed, with any axis transformations in force at the time the move was calculated. If saved setup element **Coord[*x*].Ndisplay** is set to 1, the present value of status element **Coord[*x*].Nsync**, which is automatically set to the value of the most recent synchronizing line label at the beginning of execution of the move in the program following the label, is reported at the beginning of the response, following the letter "N". This makes it easy to identify the particular move when these labels are used.

If cutter radius compensation is active for the move, and there is a non-zero offset from compensation for the axis, the position for the X, Y, or Z axis will be reported with two values – the programmed value for the axis immediately following the axis letter name, then a colon, and finally the automatically computed axis offset for cutter compensation. If the compensation adds an arc move around an outside corner at the blend to the next move, the offsets at the end of the arc will be reported if bit 0 (value 1) of **Coord[*x*].CCCtrl** is set to the default value of 0; the offsets at the beginning of the arc will be reported if this bit is set to 1.

If **Coord[*x*].TPSize** is set to 0 when this command is issued, Power PMAC will return the error message `No targets defined`. If **Coord[*x*].TPSize** is greater than 0, but not large enough to buffer all moves between calculation and execution, erroneous values may be reported.

The on-line **t** command performs fundamentally the same calculations as the buffered program **tread** command. In addition to returning a text string, it puts the axis values into local D-variables as the buffered program command does.

Examples

```
Coord[1].Ndisplay = 0                 // Do not report N label
Coord[1].TPCoords = $1C5              // Show A,C,X,Y,Z data
&1t
A20 C-12.25 X33.7223 Y-5.78 Z0

Coord[1].TPCoords = $1C0              // Show X,Y,Z data
```

```
&1t
X33.7223 Y-5.78 Z0

Coord[1].Ndisplay = 1                  // Report N label
&1t
N1260 X45.1:3.72 Y-8:-1.34 Z0
```

type

Function: Report Power PMAC system type

Scope: Global

Syntax: **type**

The **type** command causes Power PMAC to report the kind of Power PMAC system it is.

Example

```
type                            // Query version number of firmware
PWR PMAC UMAC                   // Power PMAC response
```

undefine

Function: Clear coordinate system definitions

Scope: Coordinate-system specific

Syntax: **undefine**

The **undefine** command causes Power PMAC to clear all of the axis definitions in the specified coordinate system(s). If not immediately preceded by a coordinate system list, it will do so for the presently addressed motor. If immediately preceded by a coordinate-system list, it will do so for all coordinate systems in the list.

This command does not affect the axis definitions in any of the other coordinate systems. It can be useful to prepare for a new set of axis definitions.

To clear all of the axis definitions in every coordinate system, use the **undefine all** command.

undefine all

Function: Clear all coordinate system definitions

Scope: Global

Syntax: **undefine all**

The **undefine all** command causes Power PMAC to clear all of the axis definitions in all coordinate systems.. It can be useful to prepare for a new set of axis definitions.

To clear all of the axis definitions in specified coordinate systems, use the **undefine** command.

v

Function: Report actual velocity value(s)

Scope: Motor or Coordinate-system specific

Syntax: **v**

The **v** command causes Power PMAC to report the present value of the actual velocity (velocities) for the specified motor(s) or coordinate system(s). If not immediately preceded by a motor list or coordinate system list, it will report the value of the actual velocity for the presently addressed motor. If immediately preceded by a motor list, it will report the values of the actual velocities for all motors in the list. If immediately preceded by a coordinate-system list, it will report the values of the actual velocities for all active axes for all coordinate systems in the list.

Note that specifying a list of multiple motors or multiple coordinate systems does not change the modally addressed motor or coordinate system for subsequent motor-specific or coordinate-system-specific commands.

When reporting motor velocities, the velocities are given in the defined motor units per servo cycle. When reporting axis velocities, the velocities are given in the scaled axis position units, per coordinate-system time unit (as specified by the data structure element **Coord[*x*].FeedTime** – usually seconds or minutes). The reported value for each axis is preceded by the axis name (letter or double letter). In both cases, a running-average filtered velocity value is reported.

Because this command is processed in background, the user should be aware of a couple of possible issues. If the values for multiple motors or axes are requested, it is possible that the reported values will not all be from the same servo cycle. In the case of axis values, because multiple reads of the source registers are required for each axis value, there is a slight possibility that a given value will be incorrect if a new servo update starts in the middle of the process.

If a report is requested of a coordinate system with no motors assigned to axes in that C.S., Power PMAC will return the string "`No Motors`". If the set of axis definitions or the forward kinematics subroutine for the C.S. does not permit the proper calculation of axis data, Power PMAC will return the string "`No Solution`".

Examples

```
v                           // Query actual velocity of presently addressed motor
1.21                        // Power PMAC response in counts per msec

#1..4v                      // Query actual velocities of Motors 1 - 4
3.62 -7.83 0 601            // Power PMAC response in counts per msec

&1v                         // Query actual velocities of axes in C.S. 1
X36.2 Y-78.3 Z0 C6010       // Power PMAC response in user units
```

vers

Function: Report firmware version

Scope: Global

Syntax: **vers**

The **vers** command causes Power PMAC to report the version number of the firmware it is using.

Example

```
vers                                    // Query version number of firmware
1.4.0.62                                // Power PMAC response
```

POWER PMAC PROGRAM COMMAND SPECIFICATION

This section documents buffered program commands for Power PMAC. Sending a buffered program command to Power PMAC simply enters that command into the open buffer; the command is not actually executed until the program is run and the sequencing of the program reaches this particular command.

abort

Function: Abort motion programs and moves

Syntax: **abort*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **abort** command causes the addressed or listed coordinate system(s) to abort motion program and moves.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Motors that are aborted are decelerated to a controlled (closed-loop stop) at the rates or times set by data-structure elements **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**. Power PMAC breaks into the commanded move profile for all motors in the specified coordinate system(s) immediately upon receipt of this command and computes the individual deceleration profiles for each of these motors. Note that these deceleration profiles are independent, so there is not necessarily any coordination in the time or the path for the stopping.

Coord[*x*].AbortTimeBase allows the user to specify the minimum time base value at which the motor deceleration profiles will be executed. If the time base value is lower than this when the command is issued, the profiles will be executed at 100% time base.

The abort command is useful for emergency stops when a controlled stop is permitted (some applications will abort for the fastest possible stop, then cut off power to the drives with a time-delay relay to satisfy regulatory requirements).

When motion program operation is already suspended by some other means (e.g. a "quit" or "hold") command, the abort command is useful to fully stop program execution so another program can be started or new programs downloaded to the Power PMAC.

An abort command to stop a motion program should not be given if there is a desire to then resume execution of the motion program from the stopped point. In general, the abort command will not stop motion at a programmed point, and not even along a programmed path. The abort command automatically causes the program counter to reset to the beginning of the (top-level) motion program.

If motors in the coordinate system are in the open-loop enabled state, this command will close the loop, with a controlled deceleration to stop if necessary. However, this command will not enabled motors that are disabled (killed).

Note that the equivalent on-line command uses the shorter form **a**, and any coordinate-system list precedes it (e.g. **&1,2a**).

Examples

```
abort;                          // Abort motors in presently addressed C.S
abort1;                         // Abort motors in C.S.1
abort3,5..7;                    // Abort motors in C.S. 3, 5, 6, & 7
```

abs

Function: Absolute move mode specification

Syntax: **abs[({*axis list*})]**

where:

- ***{axis list}*** is a set of axis names (single letter or double letter) separated by commas, and/or ranges of consecutive axes denoted by two periods between starting and ending axis names. All axis names used must be in alphabetical order, single-letter names first, followed by double-letter names.

The **abs** command without arguments causes all subsequent axis destinations in motion commands for all axes in the coordinate system commanded by the program to be treated as absolute positions. This is known as absolute mode, and it is the power-on default condition.

An **abs** command with arguments causes the specified axes in the coordinate system running the program to be in absolute mode, and all others stay the way they were before. No spaces are permitted anywhere within this command.

Execution of the **abs** command causes some or all bits of the 32-bit data-structure element **Coord[*x*].IncAxes** (incremental-mode axes) to be cleared.

Examples

```
abs;                              // Put all axes in C.S. in absolute mode
abs(X,Y);                         // Put X & Y axes in C.S. in abs mode, leave others
abs(V);                           // Put V axis in C.S. in abs mode, leave others
abs(XX,YY,ZZ,CC);                 // Put listed axes in C.S. in abs mode, leave others
abs(A..Z);                        // Put all axes in spec'ed range in abs mode
abs(A,B,UU..ZZ);                  // Put spec'd axes in abs mode
```

abs(*{vector list}*)

Function: Absolute circle center vector specification

Syntax: **abs(*{vector list}*)**

where:

- ***{vector list}*** is a set of vector component names (I, J, K, II, JJ, KK) separated by commas, and/or ranges of consecutive vector components denoted by two periods between starting and ending vector component names. All vector component names used must be in alphabetical order, single-letter names first, followed by double-letter names.

The **abs(*{vector list}*)** command puts the specified circle-center vector components into absolute mode. In absolute mode, the center-vector component specifies the signed distance to the circle center from the programming origin for the matching axis (X for I, Y for J, and Z for K), instead of from the move starting point, as in incremental mode. At power-on/reset, all vector components are in incremental mode.

Execution of the **abs(*{vector list}*)** command causes some bits of the coordinate system status data structure element **Coord[*x*].cdata** to be set.

Note that the **abs** command without any list causes all of the *axes* in the coordinate system to be put in absolute move mode, without affecting the circle-center vectors.

Examples

```
abs(I,J,K);                   // Put I/J/K vector set in abs mode
abs(I..K);                    // Put I/J/K vector set in abs mode
abs(I..KK);                   // Put both vector sets in abs mode
```

adisable

Function: Abort followed by delayed disable

Scope: Coordinate-system specific

Syntax: **adisable*[{list}]***

The **adisable** command causes Power PMAC to abort motion programs and moves for the addressed or listed coordinate system(s) automatically followed by a "delayed disable" of all the motors in the coordinate system(s).

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **`Ldata.Coord=`*`{expression}`*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

The initial action on an **`adisable`** command is equivalent to the action of an **`abort`** command, with all motors in the coordinate system(s) decelerated to a controlled (closed-loop) stop at the rates or times set by **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**, starting immediately.

However, as each motor reaches a commanded stop (**Motor[*x*].DesVelZero** = 1), it is then automatically disabled as if it were given a **`dkill`** command. If a brake output is specified for the motor (**Motor[*x*].pBrakeOut** > 0), the brake is immediately engaged at this instant, and the motor is killed (open-loop, zero command output, amplifier disabled) after a delay of **Motor[*x*].BrakeOnDelay** milliseconds.

The action taken on an **`adisable`** command is equivalent to that taken on the "global abort" input if **Coord[*x*].AbortAllMode** is set to 2.

The **`adisable`** command is new in firmware version 1.6, released 1st quarter 2014.

Note

While the result of the **`adisable`** command is virtually the same as a "Category 1" safe stop under the IEC-61800-5-2 machine safety standard, that standard requires the actual removal of power from the motor or drive after a controlled stop. Use of this command under the standard would also require a time-delay relay to remove either bus power or gate-driver power after the controlled stop.

Examples

```
adisable;                    // Abort and disable motors in presently addressed C.S
adisable1;                   // Abort and disable motors in C.S.1
adisable3,5..7;              // Abort and disable motors in C.S. 3, 5, 6, & 7
```

{axis}{data}[{axis}{data}…]

Function: Position-only move command

Syntax: ***{axis}{data}[{axis}{data}…]***

where:

- ***{axis}*** is the character or double character specifying the axis to be moved (A, B, C, U, V, W, X, Y, Z, AA, BB, … HH, LL, MM, … ZZ)
- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the end position or distance
- ***[{axis}{data}…]*** is the optional specification of simultaneous movement for more axes

This is the basic Power PMAC move specification command. It consists of one or more groupings of an axis label and its associated value. The value for an axis is scaled (with units determined by the axis definition); it represents a position if the axis is in absolute (**abs**) mode, or a distance from the most recent commanded position if the axis is in incremental (**inc**) mode. The order in which the axes are specified does not matter.

This command tells the axes where to move; it does not tell them how to move there. Other program commands and parameters define how; these must be set up ahead of time.

The type of motion a given move command generates is dependent on the mode of motion and the state of the system at the beginning of the move.

If one or more of the ***{axis}{data}*** structures follow some sort of subroutine or subprogram call on the same program line, the value of ***{data}*** can be passed to the subroutine or subprogram as an argument if that routine executes a **read** command specifying the axis name. In this case, the structure is "used" by the **read** command and will not execute as a move command.

Examples

```
X1000                              // Command X axis alone
X(P1+P2)                           // Command X axis alone
Y(Q100+500) Z35 C(P100)            // Command Y, Z, & C axes together
X1000 Y1000 XX2000 YY2000          // Command X, Y, XX, & YY axes together
A(P1) B(P2) C(P3)                  // Command A, B, & C axes together
X(Q1*sin(Q2/Q3)) U500              // Command X & U axes together
```

{axis}{data}:{data}[{axis}{data}:{data}…]

Function: Position-and-velocity move command

Syntax: ***{axis}{data}:{data}[{axis}{data}:{data}…]***

where:

- ***{axis}*** is the character or double character specifying the axis to be moved (A, B, C, U, V, W, X, Y, Z, AA, BB, … HH, LL, MM, … ZZ)

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the end position or distance for the axis

- **:*{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the signed end velocity for the axis

- ***[{axis}{data}:{data}…]*** is the optional specification of simultaneous movement for more axes

In the case of PVT (position, velocity, time) motion mode, both the ending position and velocity are specified for each segment of each axis. The command consists of one or more groupings of axis labels with two data items separated by a colon character.

The first data item for each axis is the scaled ending position or distance (depending on whether the axis is in absolute (**abs**) or incremental (**inc**) mode; position scaling is determined by the axis definition statement), and the second data item (after the colon) is the ending velocity.

The velocity units are the scaled position units as established by the axis definition statements divided by the time units as set by the **Coord[*x*].FeedTime** saved setup element. The velocity here is a signed quantity, not just a magnitude. See the examples in the PVT mode description of the Writing a Motion Program section.

The time for the segment is the argument for the most recently executed **pvt** command.

In PVT mode, if no velocity is given for the segment, Power PMAC assumes an ending velocity of zero for the segment.

Examples

```
X1000:50                        // X pos/dist of +1000, end vel of +50
Y500:-32 Z737.2:68.93           // Y pos/dist of +500, end vel of -32
                                // Z pos/dist of +737.2, end vel of +68.93
A(P1+P2):(P3) B(sin(Q1)):0      // A pos/dist of P1+P2, end vel of P3
                                // B pos/dist of sin(Q1), end vel of 0
```

{axis}{data}^{data}[{axis}{data}^{data}…]

Function: Move-until-trigger command

Syntax: ***{axis}{data}^{data}[{axis}{data}^{data}…]***

where:

- ***{axis}*** is the character or double character specifying the axis to be moved (A, B, C, U, V, W, X, Y, Z, AA, BB, … HH, LL, MM, … ZZ)

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the end position or distance for the axis in the absence of a trigger

- **^*{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the distance from the trigger position to stop if a trigger is found

- ***[{axis}{data}^{data}…]*** is the optional specification of simultaneous movement for more axes

In the **rapid** move mode, this move specification permits a move-until-trigger function. The first part of the move description for an axis – before the ^ sign – specifies where to move in the absence of a trigger. It is a position if the axis is in absolute mode; it is a distance if the axis is in incremental mode. In both cases the units are the scaled axis user units. If no trigger is found before this destination is reached, the move is a standard **rapid** move.

The second part of the move description for an axis – after the ^ sign – specifies the distance from the trigger position to end the post-trigger move if a trigger is found. The distance is expressed in the scaled axis user units.

Each motor assigned to an axis specified in the command executes a separate move-until-trigger. All the assigned motors will start together, but each can have its own trigger condition. If a common trigger is required, the trigger signal must be wired into all motor interfaces. Each motor can finish at a separate time; the next line in the program will not start to execute until all motors have finished their moves. No blending into the next move is possible.

The trigger condition is set by the saved setup data-structure element **Motor[*x*].CaptureMode**.

On the same line, some axes may be specified for normal untriggered **rapid** moves that will execute starting simultaneously.

Note

These triggered moves cannot be performed on motors that are assigned to axes through kinematic subroutines (where the motor's axis definitions is of the form **#*x*->I**).

Examples

```
X1000^0                              // X pos/dist of +1000, post-trig dist of 0
X10^-0.01 Y5.43^0.05                 // X pos/dist of +10, post-trig dist of -0.01
                                     // Y pos/dist of +5.43, post-trig dist of +0.05
A(P1)^(P2) B10^200 C(P3)^0 X10       // A pos/dist of P1, post-trig dist of P2
                                     // B pos/dist of +10, post-trig dist of +200
                                     // C pos/dist of P3, post-trig dist of 0
                                     // X pos/dist of +10, no trigger
```

{axis}{data}[{axis}{data}…]{vector}{data}[{vector}{data}…]

Function: Circular-arc move command

Syntax:

{axis}{data}[{axis}{data}…]{vector}{data}[{vector}{data}…]

where:

- ***{axis}*** is the character or double character specifying the axis to be moved (A, B, C, U, V, W, X, Y, Z, AA, BB, … HH, LL, MM, … ZZ)

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the end position or distance

- ***[{axis}{data}…]*** is the optional specification of simultaneous movement for more axes

- ***{vector}*** is a character (I, J, or K) specifying a vector component parallel to the X, Y, or Z axis, respectively, a double character (II, JJ, or KK) specifying a vector component parallel to the XX, YY, or ZZ axis respectively, or the character R specifying the magnitude of the radius vector (for the X, Y, and Z axis set only).

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the vector component or radius magnitude

- ***[{vector}{data}…]*** is the optional specification of additional vector components

For a circular mode move, both the move endpoint and the vector to the arc center are specified. The endpoint is specified just as in a **linear** mode move, either by position (referenced to the coordinate system origin), or distance (referenced to the starting position).

Circular interpolation can be performed on any plane within the X/Y/Z Cartesian axis set for a coordinate system, or on any plane within the XX/YY/ZZ Cartesian axis set. The plane or planes are specified with the **normal** command. Only these axes can be circularly interpolated.

Any axis not in the plane of circular interpolation will be linearly interpolated. For example, if the XY-plane has been specified as the plane of circular interpolation, a Z-axis move will be linearly interpolated, producing a helical path in XYZ-space. A rotary axis move will also be linearly interpolated, which can be used to keep the tool angle tangent or normal to the circular path.

The center of the arc for a circular move must also be specified in the move command. This is usually done by defining the vector to the center of the arc, using I, J, and K components for the X/Y/Z Cartesian axis set, or II, JJ, and KK components for the XX/YY/ZZ Cartesian axis set. Any vector component not specified is given a value of 0.0 by Power PMAC.

The starting point of the vector can either be the starting point of the move (default) or the programming origin of the coordinate system. "Incremental vector" mode uses the starting point of the move; "absolute vector" mode uses the programming origin. The **inc(*{vector list}*)** program command puts the specified vector components in incremental mode; the **abs(*{vector list}*)** program command puts the specified vector components in absolute mode. Note that the vector incremental/absolute mode is independent of the axis incremental absolute mode.

Alternatively, just the magnitude of the vector to the center can be specified with **R*{data}*** on the command line (XYZ Cartesian axis set only). If this is the case, Power PMAC will calculate the location of the center itself. If the value specified by ***{data}*** is positive, Power PMAC will compute the short arc path to the destination (<= 180°); if it is negative, Power PMAC will compute the long arc path (>= 180°). It is not possible to specify a full circle in one command with the R vector specifier.

The direction of the arc to the destination point – clockwise or counterclockwise – is controlled by whether the coordinate system is in **circle1** (clockwise) or **circle2** (counterclockwise) mode for the X/Y/Z axis set, and in **circle3** (clockwise) or **circle4** (counterclockwise) mode for the XX/YY/ZZ axis set. The sense of clockwise in the plane is determined by the direction of the **normal** vector to the plane.

If the destination point is a different distance from the center point than is the starting point, the radius is changed smoothly through the course of the move, creating an exponential spiral path. This is useful in compensating for any round-off errors in the specifications. Saved setup element **Coord[*x*].RadiusErrorLimit**, if set to a positive value, sets a threshold for the magnitude of this difference above which an error will be generated (status element **Coord[*x*].RadiusError** set), and the move will not be executed, with the motion program stopped.

If the vector from the starting point to the center point does not lie in the circular interpolation plane, the projection of that vector into the plane is used. If the destination point does not lie in the same circular interpolation plane as the starting point, a helical move is done to the destination point.

If the destination point (or its projection into the circular interpolation plane containing the starting point) is the same as the starting point, a full 360° arc is made in the specified direction (provided that IJK vector specification is used). In this case, only the vector needs to be specified in the move command, because for any axis whose destination is not specified, the destination point is automatically taken to be the same as the starting point.

If no vector, and no radius magnitude, is specified in the move command, a linear move will be done to the destination point, even if the program is in circle mode.

Note

Power PMAC performs arc moves by segmenting the arc and performing the best cubic fit on each segment. The coordinate-system data-structure setup element **Coord[*x*].SegMoveTime** determines the time for each segment. This element *must* be set greater than zero to put the coordinate system into this segmentation mode in order for arc moves to be done. If **Coord[*x*].SegMoveTime** is set to zero, circular arc moves will be done in linear fashion

Examples

```
normal K-1                      // Specifies XY-plane for circles
X5000 Y3000 I1000 J1000         // Specifies arc move in XY-plane
X5 Y-1 R2                       // Specifies arc move in XY-plane
X-20 Y10 Z3 J5                  // Specifies arc move in XY-plane with linear Z move
X10 Y20 C5 I5 J5                // Specifies arc move in XY-plane with rotary axis move
J10                             // Specifies a full circle of 10 unit radius

normal J-1                              // Specifies ZX-plane for circles
X(P101) Z(P102) I(P201) K(P202)         // Specifies arc move in ZX-plane
X10 I5

normal I-1                              // Specifies YZ-plane for circles
Y5 Z3 R2                                // Specifies arc move in YZ-plane
```

begin

Function: Point program counter to specified motion program

Syntax: **begin*[{list}]*:*{data}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

- ***{data}*** is a floating-point constant (without parentheses) or expression (in parentheses) specifying the motion program number (integer part) and optionally numeric line jump label (fractional part multiplied by 1,000,000)

The **begin** command causes the addressed or listed coordinate system(s) to set their program counter(s) to the beginning or specified numeric line jump label of the specified motion program.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

The integer part of ***{data}*** (i.e. rounded down) specifies the number of the motion program. If there is no fractional part, the program counter will point to the beginning of this motion program. If there is a fractional part, the fraction is multiplied by 1,000,000 to determine the numeric line jump label to which the program counter will point. The rotary motion program buffer for the coordinate system is considered motion program **0** for the purposes of this command.

Note that the equivalent on-line command uses the shorter form **b**, any coordinate-system list precedes it, and the program number is not preceded by a colon (e.g. **&1,2b75**).

Examples

```
begin:2;                        // Point presently addressed C.S. to top of prog 2
begin1:75;                      // Point C.S.1 to top of prog 75
begin3:75.001;                  // Point C.S.3 to jump label N1000: of prog 75
begin8:(P1+P2/1000000);         // Point C.S.8 to jump label specified by P2 in motion
                                // program specified by P1
```

break

Function: End of **case** command branch, exit from **while** loop

Syntax: **break**

The **break** command signifies the end of the set of commands to be executed by a specific **case** branch within a **switch** multi-branch structure. It also permits exiting from the middle of a **while** or **do..while** loop.

When the program flow hits a **break** command, execution jumps to the next command after the right "curly bracket" (**}**) that finishes the entire **switch** structure. If there is no **break** command at the end of a **case** branch, program execution continues into the next case.

If a **break** command is encountered in the execution of commands inside a **while** or **do..while** loop, program execution will jump to the first command following the end of the loop, regardless of the present state of the **while** condition.

bstart

Function: Mark start of stepping block

Syntax: **bstart**

The **bstart** ("block start") command marks the beginning of a set of program commands to be executed with a single "step" command in a motion program. Execution on a step command will proceed until the next **bstop** command is encountered. Without the **bstart**..**bstop** construct, execution on a step command would continue only until the next move, dwell, or delay command if **Coord[*x*].StepMode** is 0, or to the end of the program line if **Coord[*x*].StepMode** is greater than 0. With this construct, there can be zero, one, or more move, dwell, or delay commands executed on a step command.

This command does not affect program execution in continuous mode (on a "run" command) if **Coord[*x*].BlendDisable** is set to 0, modally enabling blending. However, if **Coord[*x*].BlendDisable** is set to 1, modally disabling blending, moves between the **bstart** and **bstop** commands will be blended together, either on a "run" or a "step" command.

This command is a "no op" in a PLC program. If a PLC program is stepped, it simply acts as an instruction to be executed (with no result) on a single-step command. It does not affect how future instructions are executed.

This structure is particularly useful for executing a single sequence of PVT-mode moves, because the individual segments do not end at zero velocity, making normal stepping very difficult.

Example

```
bstart;                          // Begin section of continuous execution
```

```
inc pvt200;                    // Move mode setting
x10:100;                       // First specified move (accel)
x20:100;                       // Second specified move (slew)
x10:0;                         // Third specified move (decel)
bstop;                         // End section of continuous execution
```

bstop

Function: Mark end of stepping block

Syntax: **bstop**

The **bstop** ("block stop") command marks the end of a set of program commands to be executed with a single "step" command in a motion program. Execution on a step command that encounters a **bstart** command will proceed until the **bstop** command is encountered. Without the **bstart**..**bstop** construct, execution on a step command would continue only until the next move, dwell, or delay command if **Coord[*x*].StepMode** is 0, or to the end of the program line if **Coord[*x*].StepMode** is greater than 0. With this construct, there can be zero, one, or more move, dwell, or delay commands executed on a step command.

This command is a "no op" in a PLC program. If a PLC program is stepped, it simply acts as an instruction to be executed (with no result) on a single-step command. It does not affect how future instructions are executed.

C{data}{assignment operator}{expression}

Function: Assign value to specified C-variable(s)

Syntax: **C*{data}*[,*{expression}*][,*{expression}*]**
** *{assignment operator}{expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the [first] variable to which a value is to be assigned

- the first optional ***{expression}*** specifies the number of variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is only assigned to a single variable

- the second optional ***{expression}*** specifies the "spacing" between the numbers of the multiple variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is assigned to consecutively numbered variables

- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable

- the final ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The **C*{data}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified C-variable(s). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 63 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be treated as a "no-operation".

Each coordinate system has its own set of C-variables for use by kinematic subroutine programs, primarily for passing axis position and velocity values to and from these routines. Generally, C-variables will be used only in these kinematic subroutine programs.

The specified C-variable can be named directly in its basic letter/number format (e.g. **C6**), or when using the IDE software, a name can be given to the variable with a `#define` text substitution. When writing software in the kinematic subroutine folders of the IDE's project manager, the IDE automatically defines the substitutions **KinPosAxis*α*** and **KinVelAxis*α*** for all axes. In these latter cases, the substitution to the basic letter/number variable name is made automatically during the download.

Most commonly, the evaluated expression after the operator is simply placed in the variable directly, using the **=** assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and **--** can be used without any following expression.

It is possible to use the same value in the assignment to multiple variables of evenly spaced numbering. If a second value is used after the initial variable name, this specifies the number of variables that will be assigned a value (if no second value is used, this defaults to 1 – the initial variable specified). If a third value is also used, this specifies the numerical spacing between these multiple variables (if no third value is used, this defaults to 1, so consecutively numbered variables are used.

Since C-variables are local variables, they cannot use the delayed synchronous assignment operators available to global variables, because their "context" could be lost by the time of the actual assignment.

Examples

```
C1=17.5;                        // Set variable C1 to 17.5
C(P1)+=3;                       // Add 3 to C-variable numbered by P1
C(P1+P17)=5*sqrt(P100);         // Set C-var numbered by (P1+P17) to expression value
C0,64=0.0;                      // Initialize all 64 C-variables
C1,4,2=0.0;                     // Initialize C1, C3, C5, and C7
```

call*{data}*

Function: Jump to subprogram, with return

Syntax: **call*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or expression (in parentheses), with the integer part representing the subprogram number to be called, and the fractional part representing the line jump label of the destination line (the line jump label number is equal to the fractional part multiplied by 1,000,000; every program has an implicit **N0** at the top)

The **call** command causes program execution to jump to the subprogram (**subprog**) specified by the integer part of the value in ***{data}*** and the line jump label (**N*{constant}*** followed by a colon) specified by the fractional value in ***{data}***, with a jump back to the commands immediately following the **call** upon encountering the next **return** command. Power PMAC multiplies the fractional value by 1,000,000 to obtain the number of the jump line label. Any residual fraction after this multiplication is truncated. Therefore, one can think of the jump line label number as the first six digits to the right of the decimal point in the associated value, filled out with zeros if necessary.

Note

In the older PMAC and Turbo PMAC controllers, the fractional value is multiplied by 100,000, not 1,000,000, so that programs ported from those older controllers would have to be modified in this respect.

If the value in ***{data}*** is an integer, the jump will be to the beginning of the subprogram of the specified number if this subprogram has no **N0:** jump label. If the subprogram does have an explicit **N0:** jump label, the jump will be to that label, whether or not it is at the beginning of the subprogram.

Motion programs, PLC programs, and subprograms can use the **call** command, but only subprograms can be called. When the top-level program is a motion program, called subprograms execute according to motion program rules; when the top-level program is a PLC program, called subprograms execute according to PLC program rules. A given subprogram can be called from both motion and PLC programs.

Subroutine and subprogram calls of all types (**call**, **callsub**, **gosub**, **G**, **M**, **T**, **D**) may be nested a total of 255 deep (256 including the top level). Indefinite recursion should not be used.

If the subprogram specified by the **call** command does not exist, program execution stops with an error at the **call** command. However, if the subprogram exists but the specified jump line label does not exist, the jump will be to the top (beginning) of the subprogram.

The **call** command permits argument passing to the subprogram using the local variable stack. Local variable **L0** in the subroutine is equivalent to local variable **R0** (and to local variable **L*{StackOffset}***) in the calling routine. The default value for ***StackOffset*** is 256; if a different number is desired, it must be declared explicitly in the **open prog**, **open plc**, or **open subprog** command that initiates the downloading of the program. The IDE project manager automatically assigns the optimal stack offset value during its download.

If letters and data (e.g. **X1000 DD50**) follow the **call*{data}*** syntax, these values can be arguments to be passed to the subprogram, permitting programs in the RS-274 letter/number syntax to use subroutines with arguments. If arguments are to be passed, the first line executed in the subroutine should be a **read** command. This command will take the values associated with the specified letters and place them in the appropriate D-variable matching the letter.

For example, the value following A is placed in local variable D1, the value following B is placed in D2, and so on, to the value following Z being placed in D26. The value following the double letter AA is placed in D27, the value following BB in D28, and so on, to the value following ZZ being placed in D52. The subprogram can then use these variable values. There is only one set of D-variables per coordinate system (for when the top-level program is a motion program) and per PLC program, so if there are nested subroutines/subprograms using **read** commands, the same set of D-variables is used each time. Refer to the **read** command specification for more details.

Examples

```
call 250;                    // Call to subprog 250
call (Q100);                 // Call to subprogram numbered by value of Q100
call 1500 D10;               // Call to subprog 1500, ready to read D value
```

callsub*{data}*

Function: Jump within program, pushing variable stack, with return

Syntax: **callsub*{data}***

where:

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the line jump label of the destination line

The **callsub** command causes the program execution (motion, PLC, or subprogram) to jump to the line jump label (**N*{constant}*** followed by a colon) specified in ***{data}***, with a jump back to the commands immediately following the **callsub** upon encountering the next **return** command. If the value in an expression for ***{data}*** does not evaluate exactly to an integer value, it is rounded down to the next integer before use. A constant value must be a non-negative integer in the range 0 through 999,999, or the command will be rejected at download time with a syntax error.

If ***{data}*** is a constant, the path to the label will have been linked before program run time, so the jump is very quick. If ***{data}*** is a variable expression, it must be evaluated at run time, and the appropriate label then searched for. This takes slightly more time.

Subroutine and subprogram calls of all types (**call**, **callsub**, **gosub**, **G**, **M**, **T**, **D**) may be nested a total of 255 deep (256 including the top level). Indefinite recursion should not be used.

When entering a program, if a program line with the jump label number specified by a **callsub*{constant}*** command is not present when entry is ended with the close command, an error is generated and the program buffer is cleared.

In program execution, if the jump line label in the program specified by a **callsub** command with a calculated expression value does not exist, program execution stops with an error at the **callsub** command.

Unlike the similar **gosub** command, the **callsub** command permits argument passing to the subroutine using the local variable stack. Local variable **L0** in the subroutine is equivalent to local variable **R0** (and to local variable **L*{StackOffset}***) in the calling routine. The default value for ***StackOffset*** is 256; if a different number is desired, it must be declared explicitly in the **open prog**, **open plc**, or **open subprog** command that initiates the downloading of the program. The IDE project manager automatically assigns the optimal stack offset value during its download.

If letters and data (e.g. **X1000 DD50**) follow the **callsub*{data}*** syntax, these values can be arguments to be passed to the subprogram, permitting programs in the RS-274 letter/number syntax to use subroutines with arguments. If arguments are to be passed, the first line executed in the subroutine should be a **read** command. This command will take the values associated with the specified letters and place them in the appropriate D-variable matching the letter.

For example, the value following A is placed in local variable D1, the value following B is placed in D2, and so on, to the value following Z being placed in D26. The value following the double letter AA is placed in D27, the value following BB in D28, and so on, to the value following ZZ being placed in D52. The subprogram can then use these variable values. There is only one set of D-variables per coordinate system (for when the top-level program is a motion program) and per PLC program, so if there are nested subroutines/subprograms using **read** commands, the same set of D-variables is used each time. Refer to the **read** command specification for more details.

Examples

```
callsub 400;                    // Call to jump label N400: of this program
callsub (Q123);                 // Call to jump label numbered by value of Q123
callsub 5000 E5;                // Call to jump label N5000:, ready to pass E value
```

case*{constant}*:

Function: **switch** command branch

Syntax: **case*{constant}*:**

where:

- ***{constant}*** is an integer

The **case** command starts a specific branch within a **switch** multiple-branch structure. When the mathematical expression in the **switch** command (rounded down to the next integer if necessary) is equal to the value of the constant of the **case** command, the program commands following the colon will be executed, whether on this program line or subsequent program lines, until either a **break** command is reached, or the right "curly bracket" (**}**) that finishes the entire **switch** structure is reached. No starting or ending curly brackets are needed to delimit the

commands to execute on a particular case. Use of such brackets inside an individual case is not permitted.

Note that if there is no **break** command before the next **case** command, program execution will continue into the commands following the next **case** command.

Example

```
switch (P1) {                          // Branch based on value of P1
  case 1: X10 F5; break;               // Action if P1 = 1
  case 2: X17 F7; break;               // Action if P1 = 2
  case 3: X-3 F2; break;               // Action if P1 = 3
  default: dwell 100; break;           // Action if P1 != 1, 2, or 3
}
```

ccall*{constant}*

Function: Conditionally call subprogram

Syntax: **ccall*{constant}***

where:

- ***{constant}*** is an integer in the range 0 to 31 representing the index of the conditional subprogram call

The **ccall** command permits an optional and flexible subprogram call in the execution of a motion program. **ccall*{constant}*** executes the subprogram call specified by the most recent **cdef*{constant}*** command for the same index number.

If no **cdef*{constant}*** command has been executed for this index number, or there has been a **cundef*{constant}*** command issued for the same index number since the last **cdef*{constant}*** command for this index number, then **ccall*{constant}*** will be a "no-op", causing no action.

The **ccall** command can be used in CNC-style "G-code" programs to implement "implied" subroutine calls for functions such as canned cycles. PMAC implements G-codes as subprogram calls, but with functions like canned cycles, the G-code is only explicitly used the first time. The **cdef** command can be used in the canned-cycle subroutine to cause the part program to call the same subroutine on subsequent lines. The **cundef** command can be used to stop this action. A **ccall** command can simply be added to every line of the part program to implement this.

cclr*{constant}*

Function: Clear conditional execution flag

Syntax: **cclr*{constant}***

where:

- ***{constant}*** is an integer in the range 0 to 31 representing the conditional flag number

The **cclr** command causes the specified conditional-execution flag for the coordinate system to be cleared (set to 0). There are 32 flags for each coordinate system, numbered 0 through 31, stored as individual bits in the 32-bit data structure element **Coord[*x*].Cflags**. Flag *n* is stored as bit *n*, with a value of 2^n, in this element.

When flag *n* is cleared, the program commands in the same program line following the command **cexec*n*** will not execute, and the program commands in same program line following the command **cskip*n*** will execute. A cleared flag can be set with the **cset** command.

The **cclr** command is a buffered program command. To execute it effectively as an on-line command, use the **cx** "compile and execute" structure: e.g. **&*x* cx cclr*n***.

ccmode0

Function: Turn off cutter radius compensation

Syntax: **ccmode0**

The **ccmode0** command turns off 2D or 3D cutter radius compensation, reducing it gradually through the next **linear** mode move. This is equivalent to the **G40** command of the machine-tool standard RS-274 language.

Note

In the older PMAC and Turbo PMAC controllers, the equivalent command was **cc0**. In Power PMAC, **cc0** is an axis move command.

ccmode1

Function: Turn on 2D cutter radius compensation left

Syntax: **ccmode1**

The **ccmode1** command turns on 2D cutter radius compensation offset to the left (when looking in the direction of tool movement), introducing it gradually through the next **linear** mode move. This is equivalent to the **G41** command of the machine-tool standard RS-274 language.

In order for the 2D compensation to become active, the coordinate system must be in segmentation mode (**Coord[*x*].SegMoveTime** > 0), the moves must be programmed in feedrate (**F**) mode instead of time (**tm**) mode, and a cutter compensation buffer must be defined for the coordinate system (**Coord[*x*].CCSize** >= 2). 2D cutter radius compensation is valid for **linear** and **circle** mode moves.

Note

In the older PMAC and Turbo PMAC controllers, the equivalent command was **cc1**. In Power PMAC, **cc1** is an axis move command.

ccmode2

Function: Turn on 2D cutter radius compensation right

Syntax: **ccmode2**

The **ccmode2** command turns on 2D cutter radius compensation offset to the right (when looking in the direction of tool movement), introducing it gradually through the next **linear** mode move. This is equivalent to the **G42** command of the machine-tool standard RS-274 language.

In order for the 2D compensation to become active, the coordinate system must be in segmentation mode (**Coord[*x*].SegMoveTime** > 0), the moves must be programmed in feedrate (**F**) mode instead of time (**tm**) mode, and a cutter compensation buffer must be defined for the coordinate system (**Coord[*x*].CCSize** >= 2). 2D cutter radius compensation is valid for **linear** and **circle** mode moves.

Note

In the older PMAC and Turbo PMAC controllers, the equivalent command was **cc2**. In Power PMAC, **cc2** is an axis move command.

ccmode3

Function: Turn on 3D cutter radius compensation

Syntax: **ccmode3**

The **ccmode3** command turns on 3D cutter radius compensation, introducing it gradually through the next **linear** mode move. Because the offsets in 3D compensation are explicitly specified in the part program, there is no need to declare a left or right mode as there is in 2D compensation.

When the ccmode3 command is executed, both the tool-orientation vector and the surface-normal vector are automatically set to the null vector. The tool-orientation vector can subsequently be specified by the **txyz** program command, and the surface-normal vector can subsequently be specified by the **nxyz** program command. Both vectors must be non-zero for any compensation will actually occur. In addition a tool-tip geometry must have been specified in the **Coord[*x*].CC3Data** structure elements.

In order for the 3D compensation to become active, the coordinate system must be in segmentation mode (**Coord[*x*].SegMoveTime** > 0), the moves must be programmed in feedrate (**F**) mode instead of time (**tm**) mode, and a cutter compensation buffer must be defined for the coordinate system (**Coord[*x*].CCSize** >= 2). 3D cutter radius compensation is valid for **linear** and **circle** mode moves, but generally only used for short **linear** mode moves.

Note

In the older PMAC and Turbo PMAC controllers, the equivalent command was **cc3**. In Power PMAC, **cc3** is an axis move command.

ccr*{data}*

Function: Specify cutter radius magnitude for 2D compensation

Syntax: **ccr*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the radius of the cylindrical tool

The **ccr*{data}*** command specifies the magnitude of the radius of the tool for 2D cutter radius compensation, expressed in the units of the X, Y, and Z axes. This function is often part of the D tool data codes used in the machine-tool standard RS-274 language.

In operation, the tool-center path will be offset by this distance from the programmed path perpendicular to the path in the defined plane of compensation.

This value is not used in 3D cutter radius compensation, which uses the **Coord[*x*].CC3Data** structure to define a potentially complex tool-tip end geometry.

cdef{constant} {subprogram call}

Function: Define conditional subprogram call

Syntax: **cdef*{constant} {subprogram call}***

where:

- ***{constant}*** is an integer in the range 0 to 31 representing the index of the conditional subprogram call
- ***{subroutine call}*** is a legal Power PMAC subprogram call command from the set:
  - **call*{constant}***
  - **G*{constant}***
  - **M*{constant}***
  - **T*{constant}***
  - **D*{constant}***

The **cdef*{constant}{subprogram call}*** command permits the user to specify a subprogram call command to be executed by subsequent **ccall*{constant}*** commands for the same index number executed by motion programs in the same coordinate system.

The subprogram calls specified in the **cdef** command must use a constant number (e.g. **cdef20 call500**), even though when directly used could use a mathematical expression (e.g. **call(p20+1)**).

The **cdef** command can be used in CNC-style "G-code" programs to implement "implied" subroutine calls for functions such as canned cycles. PMAC implements G-codes as subroutine calls, but with functions like canned cycles, the G-code is only explicitly used the first time. The **cdef** command can be used in the canned-cycle subroutine to cause the part program to call the subroutine on subsequent lines. A **ccall** command can simply be added to every line of the part program to implement this.

Power PMAC does not check at download time to see if the specified subprogram or label is part of the download. In execution, a call to a non-existent subprogram is a "no-op".

The effect of a **cdef** command is undone by a **cundef** command for the same index number.

Example

```
open subprog 1000              // G-code implementation subroutines
…
N80000: cundef1                // Canned cycle cancel routine
return
N81000: cdef1 G81              // Canned cycle 1 implementation routine
read(X,Y,P)                    // Read parameters for cycle
…                              // Only execute cycle if parameters passed
return
N82000: cdef1 G82              // Canned cycle 2 implementation routine
read(X,Y,L)                    // Read parameters for cycle
```

```
…                               // Only execute cycle if parameters passed
return
N83000: cdef1 G83               // Canned cycle 3 implementation code
read(X,Y,R)                     // Read parameters for cycle
…                               // Only execute cycle if parameters passed
return
…

// In part program, "ccall1" is added to every line

open prog 1                     // Part program
…
ccall1 G81 X25 Y40 P10          // G81 canned cycle explicitly called
ccall1 X35 Y60 P10              // G81 canned cycle called by "ccall1"
ccall1 X45 Y20 P5               // G81 canned cycle called by "ccall1"
ccall1 G80                      // G80 canned cycle cancel explicitly called
                                // Note no parameters to G81
ccall1 G00 X75 Y100             // No call, just rapid move
ccall1 G82 X80 Y90 L10          // G82 canned cycle explicitly called
ccall1 X85 Y110 L20             // G82 canned cycle called by "ccall1"
ccall1 X95 Y120 L10             // G82 canned cycle called by "ccall1"
ccall1 G80                      // G80 canned cycle cancel explicitly called
                                // Note no parameters to G82
…
```

cexec{constant}

Function: Conditionally execute remainder of program line

Syntax: **cexec*{constant}***

where:

- ***{constant}*** is an integer in the range 0 to 31 representing the conditional flag number

The **cexec** command determines whether the rest of the program line will be executed or not, depending on the present value of the specified conditional-execution flag for the coordinate system. There are 32 flags for each coordinate system, numbered 0 through 31, stored as individual bits in the 32-bit data structure element **Coord[*x*].Cflags**. Flag *n* is stored as bit *n*, with a value of 2^n, in this element.

If the value of Boolean flag *n* is 0 (cleared), the program commands in the same program line following the command **cexec*n*** will not execute. If the value of Boolean flag *n* is 1 (set), the program commands in the same program line following the command **cexec*n*** will execute. There can be more than one **cexec*n*** and/or **cskip*n*** command in a single program line.

A cleared flag can be set with the **cset*n*** command. A set flag can be cleared with the **cclr*n*** command.

circle

Function: Set circular interpolation move mode

Syntax: **circle*{constant}***

where:

- ***{constant}*** is an integer from 1 to 4 specifying the circular mode.

The **circle** command puts the motion program in the circular-interpolation move mode for one of the Cartesian axis sets. In this mode, other axes will move in linear-interpolation mode Subsequent move commands in the motion program will be processed according to the rules of this mode.

The four variants of the **circle** command, and the resulting modes, are:

- **circle1** : X/Y/Z axis set in clockwise circular interpolation mode
- **circle2** : X/Y/Z axis set in counter-clockwise circular interpolation mode
- **circle3** : XX/YY/ZZ axis set in clockwise circular interpolation mode
- **circle4** : XX/YY/ZZ axis set in counter-clockwise circular interpolation mode

Putting one set of Cartesian axes into a circle mode does not affect the other set of Cartesian axes if that set is already in a linear or circle mode. However, if the other set of Cartesian axes is in a different mode (**pvt**, **rapid**, **spline**), it will be put in linear mode. A command for another move mode (**linear**, **pvt**, **rapid**, **spline**) will take both sets of Cartesian axes out of circle mode.

The plane of circular interpolation in each Cartesian axis set is determined by the **normal** command. If any axis in the particular Cartesian set is a feedrate axis, all axes in that set used for circular interpolation will be treated as feedrate axes for a circular move, regardless of whether they are explicitly declared as such.

The coordinate system must be in move-segmentation mode, with saved data-structure setup element **Coord[*x*].SegMoveTime** (Isx13) set greater than 0, in order to perform circular interpolation. If this element is set to 0 to disable segmentation, the moves will be linearly interpolated.

clear gather

Function: Erase contents of servo data gathering storage buffer

Syntax: **clear gather**

The **clear gather** command causes Power PMAC to erase the contents of the servo-interrupt data gathering storage buffer, whether the standard buffer or a buffer set up in the user shared memory buffer with **Gather.UserBufStart**. It is generally *not* necessary to do this to prepare the buffer for the next data gathering event.

However, if the next data gathering event will use the storage buffer in "wrap-around" mode (**Gather.Enable** = 3) and it is possible that the gathering will stop before wrap-around, the **clear gather** command should be used before starting this, so that the uploading routine would not retrieve partial data from a previous gathering event.

The **clear gather** command can also be executed directly as an on-line command outside of any program.

The **clear gather** command is new in V2.1 firmware, released 1st quarter 2016.

clear phase gather

Function: Erase contents of phase data gathering storage buffer

Syntax: **clear phase gather**

The **clear phase gather** command causes Power PMAC to erase the contents of the phase-interrupt data gathering storage buffer, whether the standard buffer or a buffer set up in the user shared memory buffer with **Gather.PhaseUserBufStart**. It is generally *not* necessary to do this to prepare the buffer for the next data gathering event.

However, if the next data gathering event will use the storage buffer in "wrap-around" mode (**Gather.PhaseEnable** = 3) and it is possible that the gathering will stop before wrap-around, the **clear phase gather** command should be used before starting this, so that the uploading routine would not retrieve partial data from a previous gathering event.

The **clear phase gather** command can also be executed directly as an on-line command outside of any program.

The **clear phase gather** command is new in V2.1 firmware, released 1st quarter 2016.

cmd

Function: Issue command to Power PMAC command parser

Syntax: **cmd***[,]"{string}"[,{expression}…]*

where:

- ***{string}*** is a formatted ASCII text string consisting of literal alphanumeric characters, escape sequences, and formatted variable sequences, as explained below
- ***{expression}*** is a mathematical expression containing the value that is to be sent for the matching formatted variable sequence in the string. There must be one expression for each formatted variable sequence.

The **cmd** program command causes the Power PMAC to issue the formatted text string as a command to the Power PMAC, just as if it were sent from an external computer through a

"gpascii" thread. The Power PMAC has a dedicated communications thread always enabled to process these commands, independent of other communications threads.

When the command is processed as part of program execution, the string is created and buffered for processing in the communications thread. Once the string is buffered, program execution continues; it does not wait until the string is processed by the thread.

The string to be sent is a combination of standard ASCII characters, "escape sequences" that are started with a "backslash" (\) character and permit the output of control and special characters, and "format sequences" that are started with a "percent" (%) character and specify how the numerical value of an expression is to be formatted as a string. For each format sequence, there must be a matching expression following the string.

The following escape sequences can be used (all letters must be lower case):

- `\a` Bell (ASCII 7)
- `\b` Backspace (ASCII 8)
- `\t` Horizontal tab (ASCII 9)
- `\n` New line (ASCII 10)
- `\v` Vertical tab (ASCII 11)
- `\f` Form feed (ASCII 12)
- `\r` Carriage return (ASCII 13)
- `\\` Backslash character
- `\?` Question-mark character
- `\'` Single-quote character
- `\"` Double-quote character
- `\ooo` Octal specification of ASCII character code (of range `000` to `377`)
- `\xhh` Hexadecimal specification of ASCII character code (of range `x00` to `xff`)

The following format sequences can be used (all letters must be used in upper or lower case as shown):

- `%d` Signed integer, decimal format
- `%u` Unsigned integer, decimal format
- `%x` Unsigned integer, hexadecimal format, use lower case
- `%n`x Unsigned integer, hexadecimal format, using *n* digits (*n* = 1 to 8), use lower case
- `%X` Unsigned integer, hexadecimal format, use upper case
- `%n`X Unsigned integer, hexadecimal format, using *n* digits (*n* = 1 to 8), use upper case
- `%f` Floating-point value, up to 6 digits total
- `%n`f Floating-point value, up to *n* digits total (*n* = 1 to 31)
- `%n.m`f Floating-point value, up to *n* digits total (*n* = 1 to 31), up to *m* fractional digits (*m* < *n*)
- `%s` Text string, arbitrary length (null terminated)
- `%n`s Text string, up to *n* characters (shorter if null terminator encountered)
- `%c` Single character of specified byte value
- `%n`c *n* repetitions of single character of specified byte value
- `%%` "Percent" character

Leading zeros are not created for decimal-format integers. Leading zeros are created for hexadecimal-format integers (only) when the number of digits is specified. Leading and trailing zeros are not created for floating-point values, whether or not the number of digits is specified. If the number of digits specified for a floating-point is not sufficient to represent the value without an exponent, it will automatically be represented with an exponent (e.g. `2.345e8`). The decimal point is only used for floating-point values if there is a non-zero fractional component. In floating-point values, any decimal point, minus sign, or exponent used does not count as a digit.

For each formatted variable sequence in the string, there must be an ***{expression}*** that specifies the numerical value that is to be converted to text. Each expression can be as simple as a constant or a variable, but can include mathematical operators and expressions as well. For a string variable, the expression value (rounded down to the next integer if necessary) represents the starting index in user shared memory of the string variable to be used.

If the command is a motor-specific or coordinate-system-specific command that is not immediately preceded by a motor or coordinate-system list, it will act on the motor specified by **Ldata.Motor** or **Ldata.Coord**, respectively, for the program. It is strongly recommended that the command explicitly include the motor or coordinate-system list.

Note that no responses to the command can be handled. If there is a response, it will be lost, but it will not cause an error.

In the local data structure for the coordinate system (if in a motion program) or the PLC program, the element **Ldata.CmdStatus** is set to 0 on the successful execution of the Power PMAC script command. It is set to a negative number if there is an error in executing the command. Users who wish to monitor the execution of the command should set this element to a positive number before issuing the command.

Also in the local data structure, the element **Ldata.CmdCount** increments on the execution of the command, successful or unsuccessful. Note that it does nothing if there is an error in executing the command. The user should use the **`sendallcmds`** command to ensure the command buffer is flushed before checking the count. The user can write to **Ldata.CmdCount**; some users will set it to 0 before issuing a set of commands, then monitor it to see that the expected number of commands have been executed

If there is an error in processing a command, then **Ldata.CmdStatus** will have a negative value until the next successful command is executed (or the user overwrites it with a non-negative value).

No local variables can be used inside the text commands, as the command processor is not part of any local process.

Examples

```
cmd"&1#4->1000C";                    // Assign Motor 4 to C-axis in C.S. 1

cmd"&1#4->%fC",24400*EngMode+1000;   // Assign # 4 as English or metric

Ldata.CmdStatus=1;                   // Set so can look for successful execution
cmd"M1->u.io:$A00000.%u,1",P10;      // Assign M-variable to bit specified by P10
sendallcmds;                         // Force commands from buffer
while (Ldata.CmdStatus > 0) {}       // Wait for command to execute
if (Ldata.CmdStatus < 0) {           // Error in execution?
  {error trapping commands}          // Handle the error
};

Ldata.CmdCount=0;                    // Set so can look for command execution
cmd"&1#4->0";                        // Clear axis definition in C.S. 1
cmd"&2#4->1000C";                    // Make axis definition in C.S. 2
sendallcmds;                         // Force commands from buffer
while (Ldata.CmdCount < 2) {}        // Wait for command to execute
```

continue

Function: Jump to loop condition evaluation

Syntax: **continue**

The **continue** command causes program execution inside a **while** or **do..while** loop to jump immediately to the evaluation of the loop condition in the **while** command, bypassing any intervening program commands.

If this command is sent to Power PMAC outside of a **while** or **do..while** loop, it will be rejected by Power PMAC with an error.

Example

```
P1=0; P2=0;                  // Initialize loop counter and conditional value
while (P1<1000) {            // Loop conditional
  P1++;                      // Increment loop counter
  if (P2>0) continue;        // Jump back to loop conditional
  dwell1000;                 // Pause for 1 second
}                            // End of loop
```

cout:*{data}*

Function: Command open-loop output of specified percentage

Syntax: **cout*[{list}]*:*{data}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

- ***{data}*** is a floating-point constant (without parentheses) or expression (in parentheses) specifying the percentage of maximum servo output

The **cout:*{data}*** command causes Power PMAC to put the addressed or specified motor(s) in open-loop enabled mode and force a servo-loop output of the specified magnitude, expressed as a (signed) percentage of the maximum output parameter for the motor: **Motor[*x*].MaxDac**. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the specified value, whether using a constant or an expression, has a magnitude of greater than 100.0, the value used will have a magnitude of 100.0, with the sign of the specified value. No status bit is set and no error is generated. Thus it is not possible to specify an open-loop magnitude greater than that of **Motor[*x*].MaxDac**, even if **Motor[*x*].MaxDac** is less than its top possible value.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

In general, this command should only be used in PLC programs, as standard operation of motion programs requires that all motors assigned to axes in the executing coordinate system be closed-loop.

Note that the on-line command for open-loop outputs uses the shorter form **out** (without a colon) and any motor list precedes it (e.g. **#1,2out25**).

Examples

```
cout:10;                    // 10% output from presently addressed motor
cout1:(Q1);                 // Q1% output from Motor 1, regardless of addressed
cout1,2,3:-15;              // -15% output from Motors 1, 2, and 3
cout1..3,5..7:0;            // 0% output from Motors 1, 2, 3, 5, 6, and 7
```

cset*{constant}*

Function: Set conditional execution flag

Syntax: **cset*{constant}***

where:

- ***{constant}*** is an integer in the range 0 to 31 representing the conditional flag number

The **cset** command causes the specified conditional-execution flag for the coordinate system to be set (set to 1). There are 32 flags for each coordinate system, numbered 0 through 31, stored as individual bits in the 32-bit data structure element **Coord[*x*].Cflags**. Flag *n* is stored as bit *n*, with a value of 2^n, in this element.

When flag *n* is set, the program commands in the same program line following the command **cexec*n*** will execute, and the program commands in same program line following the command **cskip*n*** will not execute. A set flag can be cleared with the **cclr** command.

The **cset** command is a buffered program command. To execute it effectively as an on-line command, use the **cx** "compile and execute" structure: e.g. **&*x* cx cset*n***.

cskip*{constant}*

Function: Conditionally skip remainder of program line

Syntax: **cskip*{constant}***

where:

- ***{constant}*** is an integer in the range 0 to 31 representing the conditional flag number

The **cskip** command determines whether the rest of the program line will be executed or not, depending on the present value of the specified conditional-execution flag for the coordinate system. There are 32 flags for each coordinate system, numbered 0 through 31, stored as individual bits in the 32-bit data structure element **Coord[*x*].Cflags**. Flag *n* is stored as bit *n*, with a value of 2^n, in this element.

If the value of Boolean flag *n* is 1 (set), the program commands in the same program line following the command **cskip*n*** will not execute. If the value of Boolean flag *n* is 0 (cleared), the program commands in the same program line following the command **cskip*n*** will execute. There can be more than one **cskip*n*** and/or **cexec*n*** command in a single program line.

A cleared flag can be set with the **cset*n*** command. A set flag can be cleared with the **cclr*n*** command.

cundef*{constant}*

Function: Undefine conditional subprogram call

Syntax: **cundef*{constant}***

where:

- ***{constant}*** is an integer in the range 0 to 31 representing the index of the conditional subroutine call

The **cundef*{constant}*** command permits the user to undo a subprogram call that was previously defined by a **cdef*{constant}*** command of the same index value for the coordinate system. After a **cundef*{constant}*** command is executed, subsequent **ccall*{constant}*** commands of the same index number will be "no-ops", not causing any subroutine call.

{data structure element}{assignment operator}{expression}

Function: Assign value to specified data structure element

Syntax: ***{data structure element}{assignment operator}{expression}***

where:

- ***{data structure element}*** is the name of the particular member of a pre-defined data structure
- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable
- ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The ***{data structure element}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified data structure element.

Most commonly, this evaluated value is simply placed in the element directly, using the **=** standard assignment operator. However, this value can be arithmetically or logically combined with the existing value of the element with standard assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and −− can be used without any following expression.

If it is desired to delay the actual assignment of the value to the variable until the start of the actual execution of the next move in the program, a "synchronous assignment operator" can be used. These are useful in motion programs for setting outputs during a blended or spline sequence of multiple moves, because the motion program must be calculating one or more moves ahead to compute how the moves are blended or splined together. With a standard assignment operator, the value would appear to get set too early. The synchronous assignment, by delaying the actual assignment until the execution of the next move starts, makes the output occur at the intuitively expected time.

Synchronous assignments are also useful in PLC programs that command motor or axis moves directly to ensure that the commanded move has started before checking to see if it has finished. For motor moves, note that the motor must be assigned to a coordinate system, because the synchronous assignment queues that hold the delayed assignment belong to coordinate systems.

Most commonly, the expression value evaluated at the time the command is found is simply placed in the variable directly at the start of execution of the next move, using the **==** synchronous assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with arithmetic synchronous assignment operators **+==**, **-==**, ***==**, **/==**, and **%==**, or logical synchronous variable assignment operators **&==**, **|==**, and **^==**. In addition, the synchronous increment and decrement operators **++=** and **--=** can be used without any following expression.

Note that the arithmetic synchronous assignment operators can only be used with floating-point elements, and the logical synchronous assignment operators can only be used with fixed-point (integer) elements. No synchronous assignment operators can be used with "local" (**Ldata**) elements.

Examples

```
Motor[3].JogSpeed=38;          // Set value of Motor 3's commanded jog speed
Coord[2].MaxFeedrate+=25;      // Add 25 to C.S. 2's maximum feedrate
Motor[5].Servo.Ki==7;          // Set #5's integral gain to 0 at next move
Ldata.Motor=4;                 // Set value of program's own addressed motor
```

D*{data}*

Function: Tool data code (D-Code)

Syntax: **D*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) in the range 0.000 to 999.999 representing the code number and therefore the line jump label of the subroutine to implement that code

Power PMAC interprets a "D-code" as a **call*{Coord[x].Dprog}*.(*{data}**1000)** command, where ***{data}*** is the D-code "number". That is, this command causes a jump to the subprogram whose number is specified by the value of **Coord[*x*].Dprog**, at the line jump label whose number is 1000 times the code number. For example, with **Coord[*x*].Dprog** at its default value of 1003, **D03** is a call to line jump label **N3000:** of **subprog 1003**. Program execution will jump back to the calling program on the next **return** command.

This structure permits the implementation of customizable D-code routines for RS-274-compatible code, as from CAD/CAM programs and for CNC-style applications. Arguments can be passed to these subroutines by following the D-code in the calling program with one or more sets of ***{letter}{data}***. The values following each letter can be obtained by the subroutine using the **read** command. The default value of **Coord[*x*].Dprog** is 1003 for all coordinate systems, so by default all coordinate systems will share the same set of D-codes. However, if different values are assigned for different coordinate systems, separate code implementations can be written.

D-codes are typically used as "tool-data codes" in RS-274 programs to specify tool length, radius, and the like. While the core set of D-codes in the standards use integer numbers in the range 0 to

99, this scheme permits fractional code numbers, a range up to 999.999, and the use of mathematical expressions for code numbers.

Subroutine and subprogram calls of all types (**call**, **callsub**, **gosub**, **G**, **M**, **T**, **D**) may be nested a total of 255 deep (256 including the top level). Indefinite recursion should not be used.

If the jump line label in the program specified by the D-code command does not exist, the jump is to the top (beginning) of the specified D-code subprogram. There is no error. This permits the user application to decide how to handle "non-existent" D-codes.

Examples

```
D03                                  // Call to label N3000: of D-code subprogram
D23.7                                // Call to label N23700: of D-code subprogram
D135                                 // Call to label N135000: of D-code subprogram
```

D{data}{assignment operator}{expression}

Function: Assign value to specified D-variable(s)

Syntax: **D*{data}[,{expression}[,{expression}]***
{assignment operator}{expression}

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the [first] variable to which a value is to be assigned
- the first optional ***{expression}*** specifies the number of variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is only assigned to a single variable
- the second optional ***{expression}*** specifies the "spacing" between the numbers of the multiple variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is assigned to consecutively numbered variables
- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable
- the final ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The **D*{data}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified D-variable(s). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 53 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression,

the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be treated as a "no-operation".

Each coordinate system has its own set of D-variables for use by **read** commands, primarily for passing arguments from "G-code" style programs. Generally, D-variables will be used only for this purpose.

The specified D-variable can be named directly in its basic letter/number format (e.g. **D6**), or when using the IDE software, a name can be given to the variable with a `#define` text substitution. In the second case, the substitution to the basic letter/number variable name is made automatically during the download.

It is possible to use the same value in the assignment to multiple variables of evenly spaced numbering. If a second value is used after the initial variable name, this specifies the number of variables that will be assigned a value (if no second value is used, this defaults to 1 – the initial variable specified). If a third value is also used, this specifies the numerical spacing between these multiple variables (if no third value is used, this defaults to 1, so consecutively numbered variables are used.

Most commonly, the evaluated expression after the operator is simply placed in the variable directly, using the **=** assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and **--** can be used without any following expression.

Since D-variables are local variables, they cannot use the delayed synchronous assignment operators available to global variables, because their "context" could be lost by the time of the actual assignment.

Examples

```
D1=17.5;                        // Set variable D1 to 17.5
D(P1)+=3;                       // Add 3 to D-variable numbered by P1
D(P1+P17)=5*sqrt(P100);         // Set D-var numbered by (P1+P17) to expression value
D0,53=0.0;                      // Set all 53 D-variables to 0
D1,4,2=0.0;                     // Set D1, D3, D5, and D7 to 0
```

ddisable

Function: Disable all motors in coordinate system

Syntax: **ddisable*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **ddisable** command causes Power PMAC to perform a "delayed disable" of control of all motors that have been defined in the addressed or specified coordinate system(s). It is basically equivalent to the **dkill** motor-specific command for all motors in the coordinate system(s), but unlike the motor command, it will act on motors in a coordinate system that is executing a motion program, aborting the motion program in the process.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Disabling control for a motor consists of setting the amplifier-enable state to "false" (disabled) opening the position/velocity servo loop, and forcing a zero servo-output value. From either a closed-loop or open-loop enabled state, the motor will end up in a "killed" state.

If the motor's automatic brake-control function is enabled by setting **Motor[*x*].pBrakeOut** to a non-zero value (the address of the brake-output register), then the brake output will be commanded to engage immediately on the **ddisable** command. If, as well, the motor is in a closed-loop zero-velocity state, the actual killing of the motor will be delayed by **Motor[*x*].BrakeOnDelay** milliseconds, giving the brake system time to engage fully.

This delayed-disable command is intended for planned disabling of motors with brakes, so that the brakes have time to engage fully. Emergency disabling of motors should be done with the similar immediate **disable** command.

The **ddisable** command has no effect on a motor that is already in a disabled state.

Examples

```
ddisable;              // Delayed disable of all motors in presently addressed C.S.
ddisable4;             // Delayed disable of all motors in C.S. 4
ddisable5..8;          // Delayed disable of all motors in C.S. 5, 6, 7, & 8
```

default:

Function: Start of **switch** command branch for no match

Syntax: **default:**

The **default** command signifies the start of the branch of a **switch** multi-branch structure that is executed when the **switch** expression value (rounded down to the next integer if necessary) does not match any of the **case** command constants. The **default** command is not required to come after all of the **case** commands in a **switch** structure, although it is most commonly put there. A **switch** structure does not require a **default** command.

When there is no match, the program commands following the colon of the **default** command will be executed, whether on this program line or subsequent program lines, until either a **break** command is reached, or the right "curly bracket" (**}**) that finishes the entire **switch** structure is reached. No starting or ending curly brackets are needed to delimit the commands to execute on the default case. Use of such brackets inside an individual case is not permitted.

Note that if there is no **break** command before the next **case** command, program execution will continue into the commands following the next **case** command.

Example

```
switch (P1) {                          // Branch based on value of P1
  case 1: X10 F5; break;               // Action if P1 = 1
  case 2: X17 F7; break;               // Action if P1 = 2
  case 3: X-3 F2; break;               // Action if P1 = 3
  default: dwell 100; break;           // Action if P1 != 1, 2, or 3
}
```

delay*{data}*

Function: Delay all axes for specified time

Syntax: **delay*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the delay time in milliseconds

The **delay** command, when executed, commands all axes in the coordinate system running the motion program to maintain their present commanded positions for the specified time. This time includes half of any deceleration time of an incoming move and half of any acceleration time of an outgoing move.

The commanded delay time is affected by override or time-base values – at a 50% override, the actual delay is twice that specified. Therefore, the use of a **delay** command while in external time-base mode will not cause a loss of synchronicity to the master input.

A **delay** command does not stop any "lookahead" calculations – subsequent lines in the motion program can be evaluated before the actual execution of the delay time is finished.

A **delay** command is equivalent to a zero-distance move of the time specified in milliseconds. As with a move command, if the specified delay time is less than the acceleration time currently in force, the delay will be for the acceleration time, not the specified delay time.

The similar **dwell** command has several subtle differences: the dwell time does not include any of the incoming deceleration time or of the outgoing acceleration time; it does not vary with override and time-base values; and it does inhibit lookahead calculations.

Examples

```
delay250;                    // Zero-distance move of 250 msec (at %100)
delay(P1+P2);                // Zero-distance move of P1+P2 msec (at %100)
```

disable

Function: Disable all motors in coordinate system

Syntax: **disable*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **disable** command causes Power PMAC to "disable" control of all motors that have been defined in the addressed or specified coordinate system(s). It is basically equivalent to the **kill** motor-specific command for all motors in the coordinate system(s), but unlike the motor command, it will act on motors in a coordinate system that is executing a motion program, aborting the motion program in the process.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Disabling control for a motor consists of setting the amplifier-enable state to "false" (disabled) opening the position/velocity servo loop, and forcing a zero servo-output value. From either a closed-loop or open-loop enabled state, the motor will end up in a "killed" state.

This immediate-disable command is intended for emergency disabling of motors. Planned disabling of motors with automatic brake control should be done with the similar delayed **ddisable** command, so that the brakes have time to engage fully. For motors without automatic brake control, it does not matter which command is used.

The **disable** command has no effect on a motor that is already in a disabled state.

Examples

```
disable;                          // Disable all motors in presently addressed C.S.
disable4;                         // Disable all motors in C.S. 4
disable5..8;                      // Disable all motors in C.S. 5, 6, 7, & 8
```

disable bgcplc

Function: Disable specified background C PLC program(s)

Syntax: **disable bgcplc** ***{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the background C PLC programs to be disabled.

The **disable bgcplc** command causes Power PMAC to stop the execution of the specified background C PLC program(s) by inhibiting the start of subsequent scans. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. with a **sleep** command).

If a C PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other existing C PLC programs specified in the command.

Examples

```
disable bgcplc 0              // Disable execution of CPLC 0
disable bgcplc 2,4,6          // Disable execution of CPLCs 2, 4, and 6
disable bgcplc 7..10          // Disable execution of CPLCs 7, 8, 9, and 10
disable bgcplc 11,13..16,20   // Disable execution of CPLCs 11, 13, 14, 15, 16, and 20
```

disable plc

Function: Disable operation of specified PLC program(s)

Syntax: **disable plc *{list}***

where:

- ***{list}*** is a set of one or more constants or ranges of constants from 0 to 31 specifying the PLC programs to be disabled

The **disable plc** command causes Power PMAC to disable (stop executing) the specified PLC program or programs. Subsequent scans of the specified programs are inhibited until the program is re-enabled. Execution can subsequently be resumed at the top of the program with the **enable plc** command.

Note that the enable/disable status of a PLC is checked only at the start of a scan. If a PLC program disables itself during a scan with this command, the present scan will still complete.

PLC programs are specified by number (0 to 31) and may be used singularly in this command, or with multiple individual numbers separated by commas, or in a range of consecutively numbered programs.

Examples

```
disable plc 0;              // Disable execution of PLC 0
disable plc 1,2,5;          // Disable execution of PLCs 1, 2, & 5
disable plc 1..16;          // Disable execution of PLCs 1 thru 16
disable plc 3..6,8;         // Disable execution of PLCs 3 thru 6 & 9
```

disable rticplc

Function: Disable foreground C PLC program

Syntax: **disable rticplc**

The **disable rticplc** command causes Power PMAC to stop the execution of the foreground C PLC program that executes under the real-time interrupt by inhibiting the start of subsequent scans. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. with a **sleep** command).

If there is no foreground C PLC program present in the Power PMAC, no error will be reported.

dkill

Function: Issue delayed motor kill command

Syntax: **dkill*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

The **dkill** command causes a "delayed kill" the addressed or specified motor(s). If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their actions simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

"Killing" a motor causes the servo loop to be opened, the servo output value forced to zero (although output bias terms are still applied), and the amplifier-enable signal forced to the disable state. Note that Power PMAC automatically kills a motor on an amplifier fault or a fatal following error fault.

The program **dkill** motor command will not affect a motor that is in a coordinate system that is currently running a motion program; program execution must be stopped first. However, the similar program **ddisable** coordinate system command will act on all motors in the coordinate system, even if the coordinate system is currently executing a motion program.

If the motor's automatic brake-control function is enabled by setting **Motor[*x*].pBrakeOut** to a non-zero value (the address of the brake-output register), then the brake output will be commanded to engage immediately on the **dkill** command. If, as well, the motor is in a closed-loop zero-velocity state, the actual killing of the motor will be delayed by **Motor[*x*].BrakeOnDelay** milliseconds, giving the brake system time to engage fully.

This delayed-kill command is intended for planned killing of motors with brakes, so that the brakes have time to engage fully. Emergency killing of motors should be done with the similar immediate **kill** command.

In general, this command should only be used in PLC programs, as standard operation of motion programs requires that all motors assigned to axes in the executing coordinate system be enabled and closed-loop.

Note that the on-line command for killing motors uses the same form **dkill**, and any motor list precedes it (e.g. **#1,2dkill**).

Examples

```
dkill;                          // Acts on presently addressed motor
dkill1;                         // Acts on Motor 1, regardless of addressed
dkill1,2,3;                     // Acts on Motors 1, 2, and 3
```

```
dkill1..3,5..7;                // Acts on Motor 1, 2, 3, 5, 6, and 7
```

do

Function: Start conditional loop

Syntax: **do**

Full Structure Syntax:

```
do {command}[{command}…] while({condition})

do {
   {command}
   [{command}…]
}
while({condition})
```

The **do** command marks the start of commands to be executed in a conditional loop that is ended by a **while** command. The condition of the **while** command is evaluated at the end of each loop, so these actions are always executed at least once.

If commands follow immediately on the same program line, only these commands on the same line are executed in the loop. In this case, a **while** command must follow these commands, on the same line or at the beginning of the next program line.

If the left "curly bracket" (**{**) is the next character in the program after the **do** command, whether on the same program line or the next program line, this bracket denotes the start of the commands to be executed in the loop. This command execution will continue until a right curly bracket (**}**) is encountered. A **while** command must follow the closing bracket, on the same line or at the beginning of the next program line.

Examples

```
do dwell 50 while (M1 == 0);          // Dwell until input M1 goes true

do {                                  // Start do loop
  X100 F20;                           // Move out
  dwell 50;                           // Dwell in "out" position
  X0 F10;                             // Move back
  dwell 50;                           // Dwell in "back" position
}                                     // Dwell in "out" position
while (P100 < 10);                    // Condition for continuing loop
```

dread

Function: Report axis present desired positions

Syntax: **dread**

The **dread** command causes Power PMAC to calculate the present axis desired positions in the coordinate system addressed by the program in **Ldata.Coord**. The desired positions will be calculated for all active axes in the coordinate system.

The present desired position for the axis is calculated from the corresponding motor desired position(s), processed through the coordinate system definition (either by inverting axis definition statements or using the forward kinematic subroutine), and any matrix transformations in force.

The axis desired position values will be copied into local D-variables for the program, where they can be used in subsequent calculations. The positions will be returned in the present axis units, with any axis matrix transformations in force. The D-variable D*i* used for each axis can be found in the following table:

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' desired positions 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.

The buffered program **dread** command performs the same calculations as the on-line **d** command.

dtogread

Function: Report axis distances-to-go of presently executing move

Syntax: **dtogread**

The **dtogread** command causes Power PMAC to calculate the axis distances to go for the presently executing move in the coordinate system addressed by the program in **Ldata.Coord**. Target position buffering must be enabled by setting saved setup element **Coord[*x*].TPSize** greater than 0, and to a value sufficient to store positions for all of the moves between move calculation time and move execution time. The distances to go will be calculated for all defined axes in the coordinate system.

"Distance to go" for an axis is computed as the axis' target position for the presently executing move minus the immediate commanded position for the axis. For the X, Y, and Z axes, if cutter radius compensation is enabled, the target position value offset by the cutter compensation is used in this calculation. The present desired position for the axis is calculated from the corresponding motor desired position(s), processed through the coordinate system definition (either by inverting axis definition statements or using the forward kinematic subroutine), and any matrix transformations in force. The difference is reported as a signed quantity.

The distances to go will be copied into local D-variables for the program, where they can be used in subsequent calculations. The distances will be returned as programmed, with any axis matrix

transformations in force at the time the move was calculated. The D-variable D*i* used for each axis can be found in the following table:

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' distances to go have been reported. Each bit *i* of D32 corresponds to the number of the variable D*i* for which a variable has been reported. The value of D32 should be equivalent to the value of **Coord[*x*].TPCoords**. If the returned value of D32 is 0, no distances have been reported.

The buffered program **dtogread** command performs the same calculations as the on-line **g** command.

dwell*{data}*

Function: Dwell all axes for specified time

Syntax: **dwell*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the dwell time in milliseconds

The **dwell** command, when executed, commands all axes in the coordinate system running the motion program to maintain their present commanded positions for the specified time. This time does not include any of the deceleration time of an incoming move or acceleration time of an outgoing move.

The commanded dwell time is not affected by any override or time-base values – it always operates in "real time" as defined by the global saved data-structure setup element **Sys.ServoPeriod**. Therefore, the use of any **dwell** statement, even a **dwell0**, while in external time-base mode, will cause a loss of synchronicity to the master input.

A **dwell** command stops all "lookahead" calculations – subsequent lines in the motion program are not evaluated until after the actual execution of the dwell time is finished.

The similar **delay** command has several subtle differences: the delay time includes half of the incoming deceleration time and half of the outgoing acceleration time; it varies with override and time-base values; and it does not inhibit any lookahead calculations.

Examples

```
dwell 250;                  // Dwell for 250 msec (at all % values)
dwell (P1+P2);              // Dwell for P1+P2 msec (at all % values)
dwell 0;                    // Disable blending - momentary stop
```

else

Function: Start false condition branch

Syntax: **else**

Full Structure Syntax:

```
else {command}[{command}…]

else {
   {command}
   [{command}…]
}
```

The **else** command marks the start of commands to be executed if the preceding **if** condition is evaluated as false. An **else** command requires a matching preceding **if** command, but an **if** command does not require a following **else** command.

If commands follow immediately on the same program line, only these commands on the same line are executed in the event of a false **if** condition.

If there are no commands following on the same program line, the next program line must start with the left "curly bracket" (**{**) to denote the start of the commands to be executed on a false condition. This command execution will continue until a right curly bracket (**}**) is encountered. It is permissible to put the starting left curly bracket on the same line with the **else** command.

Examples

```
if (M1 == 0) P20++;                    // Increment P20 if input M1 false
else P20--;                            // Otherwise decrement P20

if (P500 > 0) P500--;                  // Decrement P500 if greater than 0
else {                                 // Start of branch for P500 <= 0
  X50 tm200;                           // Execute X-axis move
  dwell 75;                            // Dwell for 75 msec
  P500 = 50;                           // Set P500 to 50
}                                      // End of branch for P500 <= 0
```

enable

Function: Enable all motors in coordinate system

Syntax: **enable***{list}*

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers,

in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **enable** command causes Power PMAC to "enable" closed-loop servo control of all disabled motors that have been defined in the addressed or specified coordinate system(s).

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Enabling servo control for a motor consists of setting the amplifier-enable state to "true" (enabled) and closing the position/velocity servo loop. From either a "killed" state or an open-loop enabled state, the motor will end up in an enabled, closed-loop, zero-commanded-velocity state.

Examples

```
enable;                    // Enable all motors in presently addressed C.S.
enable4;                   // Enable all motors in C.S. 4
enable5..8;                // Enable all motors in C.S. 5, 6, 7, & 8
```

enable bgcplc

Function: Enable specified background C PLC program(s)

Syntax: **disable bgcplc *{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the background C PLC programs to be disabled.

The **enable bgcplc** command causes Power PMAC to permit the execution of the specified background C PLC program(s) at their normal priority and timing. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. with a **sleep** command).

If a C PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other existing C PLC programs specified in the command.

Examples

```
enable bgcplc 0                 // Enable execution of CPLC 0
enable bgcplc 2,4,6             // Enable execution of CPLCs 2, 4, and 6
enable bgcplc 7..10             // Enable execution of CPLCs 7, 8, 9, and 10
enable bgcplc 11,13..16,20      // Enable execution of CPLCs 11, 13, 14, 15, 16, and 20
```

enable plc

Function: Enable operation of specified PLC program(s)

Syntax: **enable plc** ***{list}***

where:

- ***{list}*** is a set of one or more constants or ranges of constants from 0 to 31 specifying the PLC programs to be enabled

The **enable plc** command causes Power PMAC to enable (start continuous execution of) the specified PLC program or programs in their normal order of execution. Each specified PLC program will start execution at the top of the program. PLC programs are specified by number (0 to 31) and may be used singularly in this command, or with multiple individual numbers separated by commas, or in a range of consecutively numbered programs.

Examples

```
enable plc 0;                   // Enable execution of PLC 0
enable plc 1,2,5;               // Enable execution of PLCs 1, 2, & 5
enable plc 1..16;               // Enable execution of PLCs 1 thru 16
enable plc 3..6,8;              // Enable execution of PLCs 3 thru 6 & 8
```

enable rticplc

Function: Enable foreground C PLC program

Scope: Global

Syntax: **enable rticplc**

The **enable rticplc** command causes Power PMAC to permit the execution of the foreground C PLC program that executes under the real-time interrupt at their normal priority and timing. Execution can only be re-started at the beginning of the program, even if this command halted execution in the middle of the program (e.g. with a **sleep** command).

If there is no foreground C PLC program present in the Power PMAC, no error will be reported.

F*{data}*

Function: Vector feedrate specification

Syntax: **F*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the vector feedrate in user length units per user time units

The **F** command specifies the vector feedrate (velocity magnitude) to be used by subsequent linear and circle mode moves, with floating-point resolution. It puts the coordinate system in feedrate mode for these moves, as opposed to move-time mode. The specified value must be positive. It overrides any previous **tm** or **F** value, and is overridden by any subsequent **tm** or **F** statement.

Execution of an **F** command in a motion program causes Power PMAC to set **Coord[*x*].Tm** to the negative of the specified value. (The minus sign indicates that the coordinate system is in feedrate mode and not move-time mode.) An **F** command is a "no op" in a PLC program; if you wish to set a coordinate system feedrate value directly from a PLC program, you should write directly to **Coord[*x*].Tm**.

The units of velocity specified in an **F** command are scaled position units (as set by the axis definitions) per scaled time unit (defined by the coordinate-system data structure setup element **Coord[*x*].FeedTime** – Isx90).

The velocity specified here is the vector velocity of all of the vector-feedrate axes of the coordinate system. That is, the move time is calculated as the vector distance of the feedrate axes (square root of the sum of the squares of the individual axes) divided by the feedrate value specified here. Any non-feedrate axes commanded to move on the same move-command line will move at the speed necessary to finish in this same amount of time.

Axes are designated as vector-feedrate axes with the **frax** command. If no **frax** command is used, the default feedrate axes are the X, Y, and Z axes. Any axis involved in circular interpolation is automatically a feedrate axis, regardless of whether it was specified in the latest **frax** command. In multi-axis systems, feedrate specification of moves is really only useful for systems with Cartesian geometries, for which these moves give a constant velocity in the plane or in 3D space, regardless of movement direction.

Note

If non-saved setup element **Coord[*x*].InvTimeMode** is set to a non-zero value, the coordinate system is in "inverse time" mode, and the value specified by an **F** command specifies the inverse of the time for the move, not the vector speed. Refer to the description of **Coord[*x*].InvTimeMode** for details.

If the feedrate programmed with the **F** command exceeds the limit specified by the coordinate-system data-structure setup element **Coord[*x*].MaxFeedrate** (Isx98), the limiting value will be used instead.

If the move time calculated from vector distance and feedrate is less than the acceleration time, the move time will be increased to match the acceleration time, resulting in axis velocities lower than those determined by dividing axis distances by the move time.

If the velocity requested for any motor in the coordinate system determined by its move distance divided by the move time exceeds the magnitude limit set by the data-structure element **Motor[*x*].MaxSpeed** (when active), the move time for that move will be extended just enough so that no motor limit is violated.

If vector-feedrate axes and non-feedrate axes are commanded together on the same program line, and the time for any non-feedrate axis, computed as the axis distance divided by the coordinate-system data-structure setup element **Coord[*x*].AltFeedrate** (Isx86), is greater than the move time calculated for the vector feedrate axes, then this longer time will be used for the move, resulting in a lower effective vector feedrate.

If only non-feedrate axes are commanded to move in a feedrate-specified move, Power PMAC will compute the move time as the longest distance commanded for any axis divided by the coordinate-system data-structure setup element **Coord[*x*].AltFeedrate**. If this parameter is set to zero, it will compute the move time as the longest distance divided by the programmed feedrate.

Examples

```
F100                    // Feedrate of 100 axis units per C.S. time unit
F31.25                  // Feedrate of 31.25 axis units per C.S. time unit
F(Q10)                  // Feedrate of Q10 axis units per C.S. time unit
F(sin(P8*P9))           // Feedrate of sin(P8*P9) axis units per C.S. time unit
```

frax

Function: Vector feedrate axis specification

Syntax: ***frax[({axis list}]***

where:

- ***{axis list}*** is a set of axis names (single letter or double letter) separated by commas, and/or ranges of consecutive axes denoted by two periods between starting and ending axis names. All axis names used must be in alphabetical order, single-letter names first, followed by double-letter names.

The **frax** command specifies which axes are to be involved in the vector-feedrate (velocity) calculations for upcoming feedrate-specified (**F**) moves. Power PMAC calculates the time for these moves as the vector distance (square root of the sum of the squares of the axis distances) of all the feedrate axes divided by the feedrate. Any non-feedrate axes commanded on the same line will complete in the same amount of time, moving at whatever speed is necessary to cover the distance in that time.

Vector feedrate has obvious geometrical meaning only in a Cartesian system, for which it results in constant tool speed regardless of direction, but it is possible to specify for non-Cartesian systems, and for more than three axes.

The default vector feedrate axes for each coordinate system are X, Y, and Z, and few users will need to change this.

Note

In a feedrate-specified move, if the move time for any non-feedrate axis, computed as axis distance divided by the data-structure element **Coord[*x*].AltFeedRate**, is greater than the move time for the feedrate axes, computed as the vector distance divided by the feedrate, Power PMAC will use the move time for the non-feedrate axis instead.

The **frax** command without arguments causes only the X, Y, and Z axes in the coordinate system to be feedrate axes in subsequent move commands. The **frax** command with arguments causes the specified axes to be feedrate axes, and all axes not specified to be non-feedrate axes, in subsequent move commands. No spaces are permitted anywhere within this command.

The **frax** command causes bits in the 32-bit **Coord[*x*].FRAxes** data structure element for the feedrate axes to be set to 1, and bits for the non-feedrate axes to be set to 0.

Examples

```
frax(X,Y)                    // X & Y as only vector feedrate axes
inc                          // Incremental move mode
X30 Y40 Z10 F100             // 3-axis move command at vector feedrate of 100
```

Vector distance is sqrt(30^2 + 40^2) = 50 mm. At a speed of 100 mm/sec, move time (unblended) is 0.5 sec. X-axis speed is 30/0.5 = 60 mm/sec; Y-axis speed is 40/0.5 = 80 mm/sec; Z-axis speed is 10/0.5 = 20 mm/sec.

```
Z20                          // Feedrate of 100 axis units per C.S. time unit
```

Vector distance is sqrt(0^2 +0^2) = 0 mm. With **AltFeedRate** = 50 (mm/sec), Z-axis speed is 50 mm/sec, move time (unblended) is 0.4 sec.

```
frax(X,Y,Z)                  // X, Y, & Z as vector feedrate axes
inc                          // Incremental move mode
X-30 Y-40 Z120 F65           // 3-axis move command at vector feedrate of 65
```

Vector distance is sqrt(-30^2 + -40^2 +120^2) = 130 mm. Move time is 130/65 = 2.0 sec. X-axis speed is 30/2.0 = 15 mm/sec; Y-axis speed is 40/2.0 = 20 mm/sec; Z-axis speed is 120/2.0 = 60 mm/sec.

frax2

Function: Secondary feedrate-axis specification

Syntax: **frax2***[({axis list})]*

where:

- ***{axis list}*** is a set of axis names (single letter or double letter) separated by commas, and/or ranges of consecutive axes denoted by two periods between starting and ending axis names. All axis names used must be in alphabetical order, single-letter names first, followed by double-letter names.

The **frax2** command specifies which axes, if any, are to be involved in the secondary vector-feedrate calculations for upcoming feedrate-specified (**F**) moves. The axes involved in the primary vector-feedrate calculations are specified by the **frax** command, with X, Y, and Z being the primary vector-feedrate axes by default at power-on/reset.

For a feedrate-specified move, Power PMAC will first compute the potential time for the move as the vector-distance (square root of the sum of the squared of the axis distances) of the primary feedrate axes divided by the commanded feedrate. Then, if there are any secondary vector feedrate axes as specified by the most recent **frax2** command, it will compute the potential time for these axes in the same way.

Power PMAC will choose the larger of these two times as its move time. (If there are any axes in the coordinate system that are not in the primary or secondary feedrate axis lists, the potential time for these axes is computed as their distances divided by **Coord[*x*].AltFeedRate**, and the larger of this time and the time from the feedrate axes is used.)

At power-on/reset, there are no axes in the **frax2** list. The **frax2** command is primarily intended for systems with dual Cartesian programming spaces that must be kept coordinated without exceeding vector feedrate limits in either space. The most common use of this command is **frax2(XX,YY,ZZ)**.

The **frax2** command with arguments causes bits in the 32-bit **Coord[*x*].FR2Axes** data structure element for the specified axes to be set to 1, and for the non-specified axes to be set to 0. No spaces are permitted anywhere within this command.

The **frax2** command without arguments only the XX, YY, and ZZ axes in the coordinate system to be secondary feedrate axes in subsequent move commands.

If an axis is specified as a primary feedrate axis, it will not be used as a secondary feedrate axis even if it is specified as such.

All axes can be removed from the secondary vector-feedrate axis list using the **nofrax2** command.

The **frax2** command is new in V2.1 firmware, released 1st quarter 2016.

Example

```
frax(X,Y,Z)                    // X, Y, & Z as (primary) vector feedrate axes
frax2(XX,YY,ZZ)                // XX, YY, & ZZ as secondary vector feedrate axes
inc linear                     // Incrmental mode, linear interpolation
X3 Y4 XX5 YY12 F10             // 4-axis move command at vector feedrate of 10
```

Primary vector distance is sqrt($3^2 + 4^2$) = 5. Secondary vector distance is sqrt($5^2 + 12^2$) = 13. Move time is 13 / 10 = 1.3.

fread

Function: Report axis present following errors

Syntax: **fread**

The **fread** command causes Power PMAC to calculate the present axis position (following) errors in the coordinate system addressed by the program in **Ldata.Coord**. The following errors will be calculated for all active axes in the coordinate system.

The present following error for the axis is calculated from the corresponding motor following error(s), processed through the coordinate system definition (either by inverting axis definition statements or using the forward kinematic subroutine), and any matrix transformations in force. If a forward kinematic subroutine is used, it must be able to perform a "double pass" execution, so desired and actual positions can be used to calculate errors.

The axis following error values will be copied into local D-variables for the program, where they can be used in subsequent calculations. The velocities will be returned in the present axis units, with any axis matrix transformations in force. The D-variable D*i* used for each axis can be found in the following table:

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' following errors 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.

When executed from a background PLC program, there is a slight possibility of an error in one of the reported axis values due to an interrupt occurring between the first and last read operations required to compute a value and updating one of the source registers.

The buffered program **fread** command performs the same calculations as the on-line **f** command.

G{data}

Function: Preparatory code (G-code)

Syntax: **G*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) in the range 0.000 to 999.999 representing the code number and therefore the line jump label of the subroutine to implement that code

Power PMAC interprets a "G-code" as a **call*{Coord[x].Gprog}*.(*{data}**1000)** command, where ***{data}*** is the G-code "number". That is, this command causes a jump to the subprogram whose number is specified by the value of **Coord[*x*].Gprog**, at the line jump label whose number is 1000 times the code number. For example, with **Coord[*x*].Gprog** at its default value of 1000, **G03** is a call to line jump label **N3000:** of **subprog 1000**. Program execution will jump back to the calling program on the next **return** command.

This structure permits the implementation of customizable G-code routines for RS-274-compatible code, as from CAD/CAM programs and for CNC-style applications. Arguments can be passed to these subroutines by following the G-code in the calling program with one or more sets of ***{letter}{data}***. The values following each letter can be obtained by the subroutine using the **read** command. The default value of **Coord[*x*].Gprog** is 1000 for all coordinate systems, so by default all coordinate systems will share the same set of G-codes. However, if different values are assigned for different coordinate systems, separate code implementations can be written.

G-codes are typically used as "preparatory codes" in RS-274 programs to set modes of operation and the like. While the core set of G-codes in the standards use integer numbers in the range 0 to 99, this scheme permits fractional code numbers, a range up to 999.999, and the use of mathematical expressions for code numbers.

Subroutine and subprogram calls of all types (**call**, **callsub**, **gosub**, **G**, **M**, **T**, **D**) may be nested a total of 255 deep (256 including the top level). Indefinite recursion should not be used.

If the jump line label in the program specified by the G-code command does not exist, the jump is to the top (beginning) of the specified G-code subprogram. There is no error. This permits the user application to decide how to handle "non-existent" G-codes.

Examples

```
G01                                  // Call to label N1000: of G-code subprogram
G54.1                                // Call to label N54100: of G-code subprogram
G122                                 // Call to label N122000: of G-code subprogram
```

gosub*{data}*

Function: Jump within program, with return

Syntax: **gosub*{data}***

where:

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the line jump label of the destination line

The **gosub** command causes the motion program execution to jump to the line jump label (**N*{constant}*** followed by a colon) specified in ***{data}***, with a jump back to the commands immediately following the **gosub** upon encountering the next **return** command. If the value in an expression for ***{data}*** does not evaluate exactly to an integer value, it is rounded down to the next integer before use. A constant value must be a non-negative integer in the range 0 through 999,999, or the command will be rejected at download time with a syntax error.

If ***{data}*** is a constant, the path to the label will have been linked before program run time, so the jump is very quick. If ***{data}*** is a variable expression, it must be evaluated at run time, and the appropriate label then searched for. This takes slightly more time.

Subroutine and subprogram calls of all types (**call**, **callsub**, **gosub**, **G**, **M**, **T**, **D**) may be nested a total of 255 deep (256 including the top level). Indefinite recursion should not be used.

When entering a program, if a program line with the jump label number specified by a **gosub*{constant}*** command is not present when entry is ended with the close command, an error is generated and the program buffer is cleared.

In program execution, if the jump line label in the program specified by a **gosub** command with a calculated expression value does not exist, program execution stops with an error at the **gosub** command.

No passing of arguments to and from the subroutine with the local variable stack is permitted with the **gosub** command. For that capability the **callsub** or **call** command should be used instead.

If letters and data (e.g. **X1000 DD50**) follow the **gosub*{data}*** syntax, these values can be arguments to be passed to the subprogram, permitting programs in the RS-274 letter/number syntax to use subroutines with arguments. If arguments are to be passed, the first line executed in the subroutine should be a **read** command. This command will take the values associated with the specified letters and place them in the appropriate D-variable matching the letter.

For example, the value following A is placed in local variable D1, the value following B is placed in D2, and so on, to the value following Z being placed in D26. The value following the double letter AA is placed in D27, the value following BB in D28, and so on, to the value following ZZ being placed in D52. The subprogram can then use these variable values. There is only one set of D-variables per coordinate system (for when the top-level program is a motion program) and per PLC program, so if there are nested subroutines/subprograms using **read** commands, the same set of D-variables is used each time. Refer to the **read** command specification for more details.

Examples

```
gosub 400;                    // Call to jump label N400: of this program
gosub (Q123);                 // Call to jump label numbered by value of Q123
gosub 5000 E5;                // Call to jump label N5000:, ready to pass E value
```

goto*{data}*

Function: Jump within program, without return

Syntax: **goto*{data}***

where:

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the line jump label of the destination line

The **goto** command causes the motion or PLC program execution to jump to the line jump label (**N*{constant}*** followed by a colon) specified in ***{data}***, with no jump back. If the value in an expression for ***{data}*** does not evaluate exactly to an integer value, it is rounded down to the next integer before use. A constant value must be a non-negative integer in the range 0 through 999,999, or the command will be rejected at download time with a syntax error.

If ***{data}*** is a constant, the path to the label will have been linked before program run time, so the jump is very quick. If ***{data}*** is a variable expression, it must be evaluated at run time, and the appropriate label then searched for. This takes slightly more time.

If the jump line label in the program specified by the **goto** command does not exist, program execution stops with an error at the **goto** command.

Note

Modern philosophies of the proper structuring of computer code strongly discourage the use of **goto**, because of its tendency to make code undecipherable.

Examples

```
gosub 400;                    // Jump to jump label N400: of this program
gosub (Q123);                 // Jump to jump label numbered by value of Q123
```

hold

Function: Perform a feed hold

Syntax: **hold*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **hold** command causes the addressed or listed coordinate system(s) to suspend motion program execution by bringing the coordinate-system time base to zero, decelerating along its path starting immediately.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

The motion program execution is suspended while in feed hold mode, but technically it is still executing. If it is subsequently desired that program execution will not be resumed, program execution should be fully aborted with an abort command.

The **hold** command is very similar in effect to a **%0** command, except that deceleration and subsequent re-acceleration are controlled by **Coord[*x*].FeedHoldSlew** (Isx95), not by **Coord[*x*].TimeBaseSlew** (Isx94). Also, execution can be resumed with a run or step command, instead of a **%100** command. In addition **hold** works under external time base, whereas a **%0** command does not.

In general, motion will not stop at a programmed point on a **hold** command. Full-speed execution along the path will commence again on a run or step command. The ramp up to the full time-base value (whether internally or externally set) will take place at a rate set by **Coord[*x*].FeedHoldSlew**. Once the full time-base value is reached, **Coord[*x*].TimeBaseSlew** determines the rate of any time-base changes.

Note that the equivalent on-line command uses the shorter form **h**, and any coordinate-system list precedes it (e.g. **&1,2h**).

Examples

```
hold;                                   // Do feed hold in presently addressed C.S
hold1;                                  // Do feed hold in C.S.1
hold3,5..7;                             // Do feed hold in C.S. 3, 5, 6, & 7
```

home

Function: Perform homing-search move

Syntax: **home*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

The **home** command causes the addressed or specified motor(s) to execute a homing-search move. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the entire homing-search move or set of moves, including the post-trigger offset(s), must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the homing-search move is started.

Homing-search speed and direction are determined by the setup data-structure element **Motor[*x*].HomeVel**. Homing-search acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for homing-search moves uses the shorter form **hm**, and any motor list precedes it (e.g. **#1,2hm**).

Examples

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

homez

Function: Program zero-move homing

Syntax: **homez*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

The **homez** command causes the addressed or specified motor(s) to execute a "zero-move" homing operation. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If a motor is activated in "gantry follower" mode (**Motor[*x*].ServoCtrl** = 8), the **homez** command causes it to move slowly to the home-trigger position (with offset) it found while following the "gantry leader" motor in a homing search move. This permits the command to eliminate any power-up skew between gantry motors.

If **Motor[*x*].pAbsPos** is set to the default value of 0, trigger, the **homez** command causes Power PMAC simply to re-define the present commanded position as the home (motor-zero) position, without movement. If there is a position "following" error at the time of this command, the resulting *actual* position will be the negative of the following error.

However, if **Motor[*x*].pAbsPos** is set to a non-zero value, Power PMAC will read the register whose address is specified by the value for the present absolute position of the motor. The position data read at this address is processed according to the settings of **Motor[*x*].AbsPosFormat**, **Motor[*x*].AbsPosSf**, and **Motor[*x*].HomeOffset**.

Note that the on-line command for homing-search moves uses the shorter form **hmz**, and any motor list precedes it (e.g. **#1,2hmz**).

Examples

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

I{data}{assignment operator}{expression}

Function: Assign value to specified I-variable(s)

Syntax: **I*{data}*[,*{expression}*[,*{expression}*]]**
** *{assignment operator}{expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the [first] variable to which a value is to be assigned

- the first optional ***{expression}*** specifies the number of variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is only assigned to a single variable

- the second optional ***{expression}*** specifies the "spacing" between the numbers of the multiple variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is assigned to consecutively numbered variables

- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable

- the final ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The **I*{data}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified I-variable(s). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 16,383 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be treated as a "no-operation".

I-variables provide shorthand references to saved setup data structure elements whose functions match those on the older Turbo PMAC. This permits experienced Turbo PMAC users who have memorized those I-variable numbers to use them for Power PMAC as well. There is absolutely no requirement to refer to any saved data structure element by the I-variable name.

The specified I-variable can be named directly in its basic letter/number format (e.g. **I122**), by the name of its underlying data structure element (e.g. **Motor[1].JogSpeed**) or when using

the IDE software, a name can be given to the variable with a `#define` text substitution (e.g. **`Mtr1Vj`**). In the final case, the substitution to the basic letter/number variable name is made automatically during the download.

It is possible to use the same value in the assignment to multiple variables of evenly spaced numbering. If a second value is used after the initial variable name, this specifies the number of variables that will be assigned a value (if no second value is used, this defaults to 1 – the initial variable specified). If a third value is also used, this specifies the numerical spacing between these multiple variables (if no third value is used, this defaults to 1, so consecutively numbered variables are used.

Most commonly, the evaluated expression after the operator is simply placed in the variable directly, using the **=** standard assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with standard assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and **--** can be used without any following expression.

If it is desired to delay the actual assignment of the value to the variable until the start of the actual execution of the next move in the program, a "synchronous assignment operator" can be used. These are useful in motion programs for setting outputs during a blended or spline sequence of multiple moves, because the motion program must be calculating one or more moves ahead to compute how the moves are blended or splined together. With a standard assignment operator, the value would appear to get set too early. The synchronous assignment, by delaying the actual assignment until the execution of the next move starts, makes the output occur at the intuitively expected time.

Synchronous assignments are also useful in PLC programs that command motor or axis moves directly to ensure that the commanded move has started before checking to see if it has finished. For motor moves, note that the motor must be assigned to a coordinate system, because the synchronous assignment queues that hold the delayed assignment belong to coordinate systems.

Most commonly, the expression value evaluated at the time the command is found is simply placed in the variable directly at the start of execution of the next move, using the **==** synchronous assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with arithmetic synchronous assignment operators **+==**, **-==**, ***==**, **/==**, and **%==**, or logical synchronous variable assignment operators **&==**, **|==**, and **^==**. In addition, the synchronous increment and decrement operators **++=** and **--=** can be used without any following expression.

Note that the arithmetic synchronous assignment operators can only be used with floating-point I-variables, and the logical synchronous assignment operators can only be used with fixed-point (integer) I-variables.

Examples

```
I122=17.5;                    // Set variable I122 to 17.5
I(L10)+=3;                    // Add 3 to I-variable numbered by L10
I(P10+P17)=5*sqrt(P100);      // Set I-var numbered by (P10+P17) to expression value
I290==1;                      // Set variable I290 to 1 at start of next move execution
I388++=;                      // Increment I388 at start of next move execution
I122,3,100=50;                // Set I122, I222, I322 to 50
```

if ({condition})

Function: Conditional branch

Syntax: **if(*{condition}*)**

where:

- ***{condition}*** is a simple or compound condition, or a mathematical expression without a conditional comparator

Full Structure Syntax:

```
if({condition}){command}[{command}…]

if({condition}) {
   {command}
   [{command}…]
}
```

The **if** command permits conditional branching in motion and PLC programs. When the condition inside parentheses evaluates as true, or the expression evaluates as any non-zero value, the program command(s) immediately following this command will be executed.

If commands follow immediately on the same program line, only these commands on the same line are executed in the event of a true condition.

If the left "curly bracket" (**{**) is the next character in the program after the right parenthesis that closes the condition, whether on the same program line or the next program line, this bracket denotes the start of the commands to be executed on a true condition. This command execution will continue until a right curly bracket (**}**) is encountered.

When the condition evaluates as false, or the expression evaluates as exactly 0.0, the commands that would be executed on a true condition are skipped. If an **else** command immediately follows these, the commands following the **else** will be executed instead. Otherwise, no actions are taken on a false condition.

It is possible to nest **if** conditions within other conditional structures (**if**, **while**, **switch**) and to nest other conditional structures within **if** conditions. The nesting of conditional structures can go up to 32 levels deep in total.

Examples

```
if (M10 == 0) P20++;          // Increment P20 if input M1 false

if (P500 > 0) {               // Start branch if P500 if greater than 0
  X50 tm200;                  // Execute X-axis move
  dwell 75;                   // Dwell for 75 msec
}                             // End of branch for P500 > 0
else P500 = 100;              // Otherwise set P500 to 100
```

inc

Function: Incremental move mode specification

Syntax: **inc[({*axis list*})]**

where:

- ***{axis list}*** is a set of axis names (single letter or double letter) separated by commas, and/or ranges of consecutive axes denoted by two periods between starting and ending axis names. All axis names used must be in alphabetical order, single-letter names first, followed by double-letter names.

The **inc** command without arguments causes all subsequent axis destinations in motion commands for all axes in the coordinate system commanded by the program to be treated as incremental distances from the last command point. This is known as incremental mode, as opposed to the default absolute mode.

An **inc** command with arguments causes the specified axes in the coordinate system running the program to be in incremental mode, and all others stay the way they were before. No spaces are permitted anywhere within this command.

Execution of the **inc** command causes some or all bits of the 32-bit data-structure element **Coord[*x*].IncAxes** (incremental-mode axes) to be set.

Examples

```
inc;                              // Put all axes in C.S. in incremental mode
inc(X,Y);                         // Put X & Y axes in C.S. in inc mode, leave others
inc(V);                           // Put V axis in C.S. in inc mode, leave others
inc(XX,YY,ZZ,CC);                 // Put listed axes in C.S. in inc mode, leave others
inc(A..Z);                        // Put all axes in spec'ed range in inc mode
inc(A,B,UU..ZZ);                  // Put spec'd axes in inc mode
```

inc({vector list})

Function: Incremental circle center vector specification

Syntax: **inc({*vector list*})**

where:

- ***{vector list}*** is a set of vector component names (I, J, K, II, JJ, KK) separated by commas, and/or ranges of consecutive vector components denoted by two periods between starting and ending vector component names. All vector component names used must be in alphabetical order, single-letter names first, followed by double-letter names.

The **inc({*vector list*})** command puts the specified circle-center vector components into incremental mode. In incremental mode, the center-vector component specifies the signed distance to the circle center from the move starting point for the matching axis (X for I, Y for J,

and Z for K), instead of from the programming origin, as in absolute mode. At power-on/reset, all vector components are in incremental mode.

Execution of the **inc(*{vector list}*)** command causes some bits of the coordinate system status data structure element **Coord[*x*].cdata** to be cleared.

Note that the **inc** command without any list causes all of the *axes* in the coordinate system to be put in incremental move mode, without affecting the circle-center vectors.

Examples

```
inc(I,J,K);                          // Put I/J/K vector set in inc mode
inc(I..K);                           // Put I/J/K vector set in inc mode
inc(I..KK);                          // Put both vector sets in inc mode
```

jog+

Function: Perform indefinite jog-positive move

Syntax: **jog+*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

The **jog+** command causes the addressed or specified motor(s) to start an indefinite jog move in the positive direction. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the initial commanded acceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the jogging move is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual

velocity of the motor at the instant of the command as the starting velocity for its acceleration profile.

Jogging speed is determined by the setup data-structure element **Motor[*x*].JogSpeed**. Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for comparable jogging moves uses the shorter form **j+**, and any motor list precedes it (e.g. **#1,2j+**).

Examples

```
jog+;                              // Acts on presently addressed motor
jog+1;                             // Acts on Motor 1, regardless of addressed
jog+1,2,3;                         // Acts on Motors 1, 2, & 3
jog+1..3,5..7;                     // Acts on Motors 1 thru 3, 5 thru 7
```

jog-

Function: Perform indefinite jog-negative move

Syntax: **jog-*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

The **jog-** command causes the addressed or specified motor(s) to start an indefinite jog move in the negative direction. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the initial commanded acceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the jogging move is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual velocity of the motor at the instant of the command as the starting velocity for its acceleration profile.

Jogging speed is determined by the setup data-structure element **Motor[*x*].JogSpeed**. Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for comparable jogging moves uses the shorter form **j-**, and any motor list precedes it (e.g. **#1,2j-**).

Examples

```
jog-;                              // Acts on presently addressed motor
jog-1;                             // Acts on Motor 1, regardless of addressed
jog-1,2,3;                         // Acts on Motors 1, 2, & 3
jog-1..3,5..7;                     // Acts on Motors 1 thru 3, 5 thru 7
```

jog/

Function: Perform jog-stop move

Syntax: **jog/*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

The **jog/** command causes the addressed or specified motor(s) to come to a closed-loop zero-velocity state, stopping a jog-type move if necessary. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the commanded deceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level

program is a PLC program, the next command in the program is executed as soon as the profile from the command is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual velocity of the motor at the instant of the command as the starting velocity for its deceleration profile. If the actual velocity is exactly zero, no profile is created.

Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for comparable jogging moves uses the shorter form **j/**, and any motor list precedes it (e.g. **#1,2j/**).

Examples

```
jog/;                              // Acts on presently addressed motor
jog/1;                             // Acts on Motor 1, regardless of addressed
jog/1,2,3;                         // Acts on Motors 1, 2, & 3
jog/1..3,5..7;                     // Acts on Motors 1 thru 3, 5 thru 7
```

jog=*{data}*

Function: Perform jog to specified position

Syntax: **jog*[{list}]*=*{data}***

jog*[{list}]*=*

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.
- ***{data}*** is a floating-point constant (without parentheses) or expression (in parentheses) specifying the destination position in motor units

The **jog=*{data}*** command causes the addressed or specified motor(s) to start a definite jog move to the motor position specified by ***{data}***. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the **jog=*** syntax is used instead, the destination position is specified by the value of element **Motor[*x*].ProgJogPos**.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either

type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the initial commanded acceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the jogging move is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual velocity of the motor at the instant of the command as the starting velocity for its acceleration profile.

Jogging speed is determined by the setup data-structure element **Motor[*x*].JogSpeed**. Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for comparable jogging moves uses the shorter form **j=*{constant}***, and any motor list precedes it (e.g. **#1,2j=1000**). On-line jog commands must use constants for positions or distances; they cannot use variables or expressions.

Examples

```
jog=5000;                          // Acts on presently addressed motor
jog1=(Q1);                         // Acts on Motor 1, regardless of addressed
jog1,2,3=-2468;                    // Acts on Motors 1, 2, & 3
jog1..3,5..7=0;                    // Acts on Motors 1 thru 3, 5 thru 7
jog25..27=*;                       // Acts on Motors 25 thru 27
```

jog:*{data}*

Function: Perform jog of specified distance relative to command position

Syntax: **jog*[{list}]*:*{data}***

jog*[{list}]*:*

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

- ***{data}*** is a floating-point constant (without parentheses) or expression (in parentheses) specifying the signed destination distance in motor units

The **jog:*{data}*** command causes the addressed or specified motor(s) to start a definite jog move of the signed distance specified by ***{data}*** from the present commanded position. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the **jog:*** syntax is used instead, the destination distance is specified by the value of element **Motor[*x*].ProgJogPos**.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the initial commanded acceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the jogging move is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual position and velocity of the motor at the instant of the command as the starting position and velocity for its acceleration profile.

Jogging speed is determined by the setup data-structure element **Motor[*x*].JogSpeed**. Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for comparable jogging moves uses the shorter form **j:*{constant}***, and any motor list precedes it (e.g. **#1,2j:-1000**). On-line jog commands must use constants for positions or distances; they cannot use variables or expressions.

Examples

```
jog:5000;                          // Acts on presently addressed motor
jog1:(Q1);                         // Acts on Motor 1, regardless of addressed
jog1,2,3:-2468;                    // Acts on Motors 1, 2, & 3
jog1..3,5..7:0;                    // Acts on Motors 1 thru 3, 5 thru 7
jog25..27:*;                       // Acts on Motors 25 thru 27
```

jog^*{data}*

Function: Perform jog of specified distance relative to actual position

Syntax: **jog*[{list}]^{data}***

jog*[{list}]*^*

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

- ***{data}*** is a floating-point constant (without parentheses) or expression (in parentheses) specifying the signed destination distance in motor units

The **jog^*{data}*** command causes the addressed or specified motor(s) to start a definite jog move of the signed distance specified by ***{data}*** from the present *actual* position. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the **jog^*** syntax is used instead, the destination distance is specified by the value of element **Motor[*x*].ProgJogPos**.

Note that because the distance is specified to the present actual, not desired, position, its action is dependent on the following error at the instant the command is executed. For this reason, its action is not entirely predictable. Many users will prefer the similar **jog:*{data}*** command, which specifies the distance from the present desired position. However, the **jog^0** command can be useful in certain applications to "swallow up" the existing following error.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the initial commanded acceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the jogging move is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual velocity of the motor at the instant of the command as the starting velocity for its acceleration profile.

Jogging speed is determined by the setup data-structure element **Motor[*x*].JogSpeed**. Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for comparable jogging moves uses the shorter form **j^*{constant}***, and any motor list precedes it (e.g. **#1,2j^-1000**). On-line jog commands must use constants for positions or distances; they cannot use variables or expressions.

Examples

```
jog^5000;                          // Acts on presently addressed motor
jog1^(Q1);                         // Acts on Motor 1, regardless of addressed
jog1,2,3^-2468;                    // Acts on Motors 1, 2, & 3
jog1..3,5..7^0;                    // Acts on Motors 1 thru 3, 5 thru 7
jog25..27^*                        // Acts on Motors 25 thru 27
```

jogret

Function: Perform jog to pre-jog position

Syntax: **jogret*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

The **jogret** command causes the addressed or specified motor(s) to start a definite jog move to the last "pre-jog" position. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the initial commanded acceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the jogging move is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual velocity of the motor at the instant of the command as the starting velocity for its acceleration profile.

Jogging speed is determined by the setup data-structure element **Motor[*x*].JogSpeed**. Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for comparable jogging moves uses the shorter form **j=**, and any motor list precedes it (e.g. **#1,2j=**).

Examples

```
jogret;                          // Acts on presently addressed motor
jogret1;                         // Acts on Motor 1, regardless of addressed
jogret1,2,3;                     // Acts on Motors 1, 2, & 3
jogret1..3,5..7;                 // Acts on Motors 1 thru 3, 5 thru 7
```

jogret=*{data}*

Function: Perform jog to specified position, making that position the pre-jog position

Syntax: **jogret*[{list}]={data}***

jogret*[{list}]=**

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

- ***{data}*** is a floating-point constant (without parentheses) or expression (in parentheses) specifying the destination position in motor units

The **jogret=*{data}*** command causes the addressed or specified motor(s) to start a definite jog move to the motor position specified by ***{data}***, making that position the "pre-jog" position for subsequent **jogret** commands. If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the **jogret=*** syntax is used instead, the destination position is specified by the value of element **Motor[*x*].ProgJogPos**.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motor program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the initial commanded acceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the jogging move is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual velocity of the motor at the instant of the command as the starting velocity for its acceleration profile.

Jogging speed is determined by the setup data-structure element **Motor[*x*].JogSpeed**. Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

Note that the on-line command for comparable jogging moves uses the shorter form **j==*{constant}***, and any motor list precedes it (e.g. **#1,2j==1000**). On-line jog commands must use constants for positions or distances; they cannot use variables or expressions.

Examples

```
jogret=5000;                          // Acts on presently addressed motor
jogret1=(Q1);                         // Acts on Motor 1, regardless of addressed
jogret1,2,3=-2468;                    // Acts on Motors 1, 2, & 3
jogret1..3,5..7=0;                    // Acts on Motors 1 thru 3, 5 thru 7
jogret25..27=*;                       // Acts on Motors 25 thru 27
```

{jog command}^{data}

Function: Perform Jog-until-trigger move

Scope: Motor Specific

Syntax:
```
jog[{list}]={data}^{data}
jog[{list}]:{data}^{data}
jog[{list}]^{data}^{data}
jog[{list}]=*^{data}
jog[{list}]:*^{data}
jog[{list}]^*^{data}
```

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

- the ***{data}*** at the end of the command is a floating-point constant (without parentheses) or expression (in parentheses) specifying the (signed) distance from the trigger position to the commanded destination of the post-trigger move, in motor units.

The "jog-until-trigger" class of commands causes the addressed or specified motor(s) to start a definite jog move of the type specified. If a pre-defined trigger condition occurs during this move, Power PMAC will automatically break into the move trajectory and replace the remaining portion with a jog move to a destination whose distance from the trigger position is determined by the final ***{data}*** in the command. If the command is not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the **jog=*^*{data}**, **jog:*^*{data}**, or **jog^*^*{data}** syntax is used instead, the pre-trigger destination position or distance is specified by the value of element **Motor[*x*].ProgJogPos**.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded, starting their moves simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

If the top-level program is a motion program, the initial commanded acceleration profile for all motors must complete before the next command in the motion program is executed. If the top-level program is a PLC program, the next command in the program is executed as soon as the jogging move is started.

If the top-level program is a PLC program, the motor(s) could be in an open-loop state (enabled or killed) at the time of this command. If this is the case, Power PMAC will use the actual velocity of the motor at the instant of the command as the starting velocity for its acceleration profile.

Jogging speed is determined by the setup data-structure element **Motor[*x*].JogSpeed**. Jogging acceleration profiles are determined by the setup data-structure elements **Motor[*x*].JogTa** and **Motor[*x*].JogTs**.

The trigger condition is set by the motor setup data-structure element **Motor[*x*].CaptureMode**.

Note that the on-line commands for comparable jogging moves use the shorter form **j**… instead of **jog**…, and any motor list precedes it (e.g. **#1,2j=10000^-50**). On-line jog commands must use constants for positions or distances; they cannot use variables or expressions.

Examples

```
jog=10000^-50;          // Acts on presently addressed motor
jog1:-50000^(P10);      // Acts on Motor 1, regardless of addressed
jog1..3=*^0;            // Motors 1, 2, & 3 jog to ProgJogPos, but return to trigger pos
```

kill

Function: Issue motor kill command

Syntax: **kill*[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of motors supported by the particular version of Power PMAC firmware.

The **kill** command causes the addressed or specified motor(s) to be "killed". If not followed by a list of motors, it acts on the single motor that is presently addressed by the program.

If the top-level program is a PLC program, the addressed motor is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Motor**. If the top-level program is a motion program, it is determined by an element in the data structure for the coordinate system that is executing the program: **Coord[*x*].Ldata.Motor**. A motion program may only command motors that are defined to an axis in the coordinate system that is running the program. Note that in either type of program, a motor can be addressed with the command **Ldata.Motor=*{expression}*** – the parent structure name is not needed.

If the command specifies a set of motors through a motor list, all motors in the list will be commanded simultaneously. Specifying a motor in a list does not affect the program's modally addressed motor.

"Killing" a motor causes the servo loop to be opened, the servo output value forced to zero (although output bias terms are still applied), and the amplifier-enable signal forced to the disable state. Note that Power PMAC automatically kills a motor on an amplifier fault or a fatal following error fault.

The program **kill** motor command will not affect a motor that is in a coordinate system that is currently running a motion program; program execution must be stopped first. However, the similar program **disable** coordinate system command will act on all motors in the coordinate system, even if the coordinate system is currently executing a motion program.

This immediate-kill command is intended for emergency killing of motors. Planned killing of motors with automatic brake control should be done with the similar delayed **dkill** command, so that the brakes have time to engage fully. For motors without automatic brake control, it does not matter which command is used.

In general, this command should only be used in PLC programs, as standard operation of motion programs requires that all motors assigned to axes in the executing coordinate system be enabled and closed-loop.

Note that the on-line command for killing motors uses the shorter form **k**, and any motor list precedes it (e.g. **#1,2k**).

Examples

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

L{data}{assignment operator}{expression}

Function: Assign value to specified L-variable(s)

Syntax: **L*{data}*[,*{expression}*[,*{expression}*]]**
** *{assignment operator}{expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the [first] variable to which a value is to be assigned
- the first optional ***{expression}*** specifies the number of variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is only assigned to a single variable
- the second optional ***{expression}*** specifies the "spacing" between the numbers of the multiple variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is assigned to consecutively numbered variables
- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable
- the final ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The **L*{data}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified L-variable(s). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be treated as a "no-operation".

L-variables are local stack variables used within a program and to pass values back to higher-level routines. Each coordinate system has its own set of L-variables for use by top-level motion programs and their subprograms. In addition, each PLC program has its own set, and each communications thread has its own set. All of these sets of L-variables are independent of each

other. The specified L-variable can be named directly in its basic letter/number format (e.g. **L10**), or when using the IDE software, either by a declared `local` variable name (auto-assigned to the L-variable), or by a name given to a specific L-variable with a `#define` text substitution. In these latter cases, the substitution to the basic letter/number variable name is made automatically during the download.

It is possible to use the same value in the assignment to multiple variables of evenly spaced numbering. If a second value is used after the initial variable name, this specifies the number of variables that will be assigned a value (if no second value is used, this defaults to 1 – the initial variable specified). If a third value is also used, this specifies the numerical spacing between these multiple variables (if no third value is used, this defaults to 1, so consecutively numbered variables are used.

Most commonly, the evaluated expression after the operator is simply placed in the variable directly, using the **=** assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and **--** can be used without any following expression.

Since L-variables are local variables, they cannot use the delayed synchronous assignment operators available to global variables, because their "context" could be lost by the time of the actual assignment.

Examples

```
L1=17.5;                        // Set variable L1 to 17.5
L(P1)+=3;                       // Add 3 to L-variable numbered by P1
L(P1+P17)=5*sqrt(P100);         // Set L-var numbered by (P1+P17) to expression value
L100,16=0;                      // Set L100 - L115 to 0
L20,5,10++;                     // Increment L20, L30, L40, L50, L60
```

lh\

Function: Command quick stop in lookahead

Syntax: **lh** ***[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **lh** command causes the Power PMAC to calculate and execute the quickest stop within the lookahead buffer for the addressed or listed coordinate system(s) that does not violate acceleration constraints for any motor within the coordinate system. Motion will continue to a controlled stop along the programmed path, but the stop will not necessarily be at a programmed point. If not immediately followed by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

The **lh** quick-stop command is generally the best command to stop motion interactively within lookahead. Its function is much like that of a traditional feed-hold command, but unlike the regular **hold** feed-hold command in Power PMAC, it is guaranteed to observe acceleration constraints set by **Motor[*x*].InvAmax** for all motors in the coordinate system.

Once stopped, several options are possible:

- Jog axes away with any of the jogging commands. The on-line jog commands can be used to jog any of the motors in the coordinate system away from the stopped point. However, before execution of the programmed path can be resumed, all motors must be returned to the original stopping point with the **j=** command.
- Start reverse execution along the path with the **<** command.
- Resume forward execution with the **>**, **run**, or **step** command.
- End program execution with the **abort** command.

If the **lh** command is given to a coordinate system that is not currently executing moves within the lookahead buffer, Power PMAC will execute the **hold** "feed-hold" command instead.

Note that the equivalent on-line command uses the shorter form ****, and any coordinate-system list precedes it (e.g. **&1,2**).

Examples

```
lh\;                                    // Acts on presently addressed motor
lh\1;                                   // Acts on C.S. 1, regardless of addressed
```

lh<

Function: Command reversal in lookahead

Syntax: **lh<** ***[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **lh<** command causes the Power PMAC to start reverse execution in the lookahead buffer for the addressed or listed coordinate system(s). If the program is currently executing in the forward direction, it will be brought to a quick stop (the equivalent of the **lh** command) first. If not immediately followed by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Deceleration from a forward move (if any) and acceleration in the reverse direction observe the **Motor[*x*].InvAmax** acceleration limits.

Execution proceeds backward through points buffered in the lookahead buffer, observing velocity and acceleration constraints just as in the forward direction. This execution continues until one of the following occurs:

- Reverse execution reaches the "beginning" of the lookahead buffer – the oldest stored point still remaining in the lookahead buffer – and it comes to a controlled stop at this point, observing acceleration limits in decelerating to a stop.
- The **lh** "quick-stop" command is given, which causes Power PMAC to come to the quickest possible stop in the lookahead buffer.
- The **lh>** "resume-forward", **run**, or **step** command is given, which causes Power PMAC to resume normal forward execution of the program, adding to the lookahead buffer as necessary.
- An error condition occurs, or a non-recoverable stopping command is given.

If any motor has been jogged away from the "quick-stop" point, and not returned with a **j=** command, Power PMAC will reject the **lh<** "back-up" command, reporting an error.

If the coordinate system is not currently in the middle of a lookahead sequence, Power PMAC will treat this command as a **hold** "feed-hold" command.

Note that the equivalent on-line command uses the shorter form **<**, and any coordinate-system list precedes it (e.g. **&1,2<**).

Examples

```
lh<;                                    // Acts on presently addressed motor
lh<1;                                   // Acts on C.S. 1, regardless of addressed
```

lh>

Function: Resume forward execution in lookahead

Syntax: **lh>** ***[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **lh>** command causes the Power PMAC to resume forward execution in the lookahead buffer for the addressed coordinate system. It is typically used to resume normal operation after a **lh** "quick-stop" command, or a **lh<** "back-up" command. If the program is currently executing in the backward direction, it will be brought to a quick stop (the equivalent of the **lh** command) first. If not immediately followed by a coordinate system list, it will do so for the presently addressed coordinate system. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

If previous forward execution had been in continuous mode (started with the **run** command), the **lh>** command will resume it in continuous mode. If previous forward execution had been in single-step mode (started with the **step** command), the **lh>** command will resume it in single-step mode. The **run** and **step** commands can also be used to resume forward execution, but they may change the continuous/single-step mode.

Deceleration from a backward move (if any) and acceleration in the forward direction observe the **Motor[*x*].InvAmax** acceleration limits.

If any motor has been jogged away from the "quick-stop" point, and not returned with a **jog=** command, Power PMAC will reject the **lh>** "resume" command, reporting an error.

If the coordinate system is not currently in the middle of a lookahead sequence, Power PMAC will treat this command as a **run** command.

Note that the equivalent on-line command uses the shorter form **>**, and any coordinate-system list precedes it (e.g. **&1,2>**).

Examples

```
lh>;                                    // Acts on presently addressed motor
lh>1;                                   // Acts on C.S. 1, regardless of addressed
```

lhpurge

Function: Empty contents of lookahead buffer

Syntax: **lhpurge**

The **lhpurge** command, when executed from a motion program or one of its subprograms, empties the lookahead buffer for the coordinate system that is running the motion program. It has no action when executed from a PLC program or one of its subprograms. It should be executed before changing the definition of a motor in the coordinate system between a position axis (which actively uses the lookahead buffer) and a spindle axis (which does not actively use the lookahead buffer, but has a slot reserved in the buffer). Use of the **lhpurge** command will prevent possible inconsistencies in the lookahead buffer after an axis definition change.

Once the **lhpurge** command is executed, it is not possible to reverse through already executed moves, as those moves have been erased from the buffer. Execution of the **lhpurge** command will force the commanded motion to a momentary stop, even if no **dwell** command is used before or after. Execution clears the **Coord[*x*].LookAheadActive** status bit for the coordinate system.

Examples

A motor is commonly changed between a positioning axis and a spindle axis from within a motion program or its subprogram (often in a G-code subroutine) using the **cmd""** construct. A robust procedure to make this change will look something like:

```
dwell 0;                              // Stop blending and lookahead
lhpurge;                              // Purge lookahead buffer
Ldata.CmdStatus = 1;                  // Will change when command executed
cmd"&1#4->S0";                        // Define Motor 4 as spindle in CS1
sendallcmds;                          // Wait for command buffer to empty
do dwell 0;                           // Loop quickly while waiting
while (Ldata.CmdStatus == 1);         // Until command fully executed
                                      // Could check for error here (if < 0)
```

The comparable procedure to change it back to a rotary positioning axis will look something like:

```
dwell 0;                              // Stop blending and lookahead
lhpurge;                              // Purge lookahead buffer
Ldata.CmdStatus = 1;                  // Will change when command executed
cmd"&1#4->100C";                      // Define Motor 4 as C-axis in CS1
sendallcmds;                          // Wait for command buffer to empty
do dwell 0;                           // Loop quickly while waiting
while (Ldata.CmdStatus == 1);         // Until command fully executed
                                      // Could check for error here (if < 0)
```

linear

Function: Set linear-interpolation move mode

Syntax: **linear**

The **linear** command puts the motion program in the linear-interpolation move mode (this is the default mode on power-up/reset). Subsequent move commands in the motion program will be processed according to the rules of this mode.

The **linear** command takes the program out of any of the other move modes (**circle**, **pvt**, **rapid**, **spline**). A command for any of these other move modes takes the program out of **linear** mode.

M{data}

Function: Machine output code (M-code)

Syntax: **M*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) in the range 0.000 to 999.999 representing the code number and therefore the line jump label of the subroutine to implement that code

Power PMAC interprets an "M-code" as a **call*{Coord[x].Mprog}*.(*{data}**1000)** command, where ***{data}*** is the M-code "number". That is, this command causes a jump to the subprogram whose number is specified by the value of **Coord[*x*].Mprog**, at the line jump label whose number is 1000 times the code number. For example, with **Coord[*x*].Mprog** at its default value of 1001, **M03** is a call to line jump label **N3000:** of **subprog 1001**. Program execution will jump back to the calling program on the next **return** command.

This structure permits the implementation of customizable M-code routines for RS-274-compatible code, as from CAD/CAM programs and for CNC-style applications. Arguments can be passed to these subroutines by following the M-code in the calling program with one or more sets of ***{letter}{data}***. The values following each letter can be obtained by the subroutine using the **read** command. The default value of **Coord[*x*].Mprog** is 1001 for all coordinate systems, so by default all coordinate systems will share the same set of M-codes. However, if different values are assigned for different coordinate systems, separate code implementations can be written.

M-codes are typically used as "machine output codes" in RS-274 programs to control machine functions such as spindle operation, coolant, conveyors, and the like. While the core set of M-codes in the standards use integer numbers in the range 0 to 99, this scheme permits fractional code numbers, a range up to 999.999, and the use of mathematical expressions for code numbers.

Subroutine and subprogram calls of all types (**call**, **callsub**, **gosub**, **G**, **M**, **T**, **D**) may be nested a total of 255 deep (256 including the top level). Indefinite recursion should not be used.

If the jump line label in the program specified by the M-code command does not exist, the jump is to the top (beginning) of the specified M-code subprogram. There is no error. This permits the user application to decide how to handle "non-existent" M-codes.

Examples

```
M03                                  // Call to label N3000: of M-code subprogram
M28.3                                // Call to label N28300: of M-code subprogram
M197                                 // Call to label N197000: of M-code subprogram
```

M{data}{assignment operator}{expression}

Function: Assign value to specified M-variable(s)

Syntax: **M*{data}*[,*{expression}*[,*{expression}*]]**
** *{assignment operator}{expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the [first] variable to which a value is to be assigned

- the first optional ***{expression}*** specifies the number of variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is only assigned to a single variable

- the second optional ***{expression}*** specifies the "spacing" between the numbers of the multiple variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is assigned to consecutively numbered variables

- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable

- the final ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The **M*{data}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified M-variable(s). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 16,383 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be treated as a "no-operation".

M-variables are global "pointer" variables that can be used to access Power PMAC hardware and software registers of interest. They must be defined to a register before they can be used.

It is possible to use the same value in the assignment to multiple variables of evenly spaced numbering. If a second value is used after the initial variable name, this specifies the number of variables that will be assigned a value (if no second value is used, this defaults to 1 – the initial variable specified). If a third value is also used, this specifies the numerical spacing between these multiple variables (if no third value is used, this defaults to 1, so consecutively numbered variables are used.

Most commonly, the evaluated expression after the operator is simply placed in the variable directly, using the **=** standard assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with standard assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and **--** can be used without any following expression.

If it is desired to delay the actual assignment of the value to the variable until the start of the actual execution of the next move in the program, a "synchronous assignment operator" can be used. These are useful in motion programs for setting outputs during a blended or spline sequence of multiple moves, because the motion program must be calculating one or more moves ahead to compute how the moves are blended or splined together. With a standard assignment operator, the value would appear to get set too early. The synchronous assignment, by delaying the actual assignment until the execution of the next move starts, makes the output occur at the intuitively expected time.

Synchronous assignments are also useful in PLC programs that command motor or axis moves directly to ensure that the commanded move has started before checking to see if it has finished. For motor moves, note that the motor must be assigned to a coordinate system, because the synchronous assignment queues that hold the delayed assignment belong to coordinate systems.

Most commonly, the expression value evaluated at the time the command is found is simply placed in the variable directly at the start of execution of the next move, using the **==** synchronous assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with arithmetic synchronous assignment operators **+==**, **-==**, ***==**, **/==**, and **%==**, or logical synchronous variable assignment operators **&==**, **|==**, and **^==**. In addition, the synchronous increment and decrement operators **++=** and **--=** can be used without any following expression.

Note that the arithmetic synchronous assignment operators can only be used with floating-point M-variables, and the logical synchronous assignment operators can only be used with fixed-point (integer) M-variables. No synchronous assignment operators of any kind can be used with self-referenced M-variables, local variables, or "function" data structure elements.

Examples

```
M1=17.5;                        // Set variable M1 to 17.5
M(L10)+=3;                      // Add 3 to M-variable numbered by L10
M(P10+P17)=5*sqrt(P100);        // Set M-var numbered by (P10+P17) to expression value
M25==1;                         // Set variable M25 to 1 at start of next move execution
M50++=;                         // Increment M50 at start of next move execution
M100,24=0;                      // Set M100 - M123 to 0
M20,5,10^=1;                    // Logically invert M20, M30, M40, M50, M60
```

N*{constant}*:

Function: Program line jump label

Syntax: **N*{constant}*:**

where:

- ***{constant}*** is an unsigned 32-bit integer (range of 0 to 4,294,967,295)

This is a numeric "jump" line label that permits the flow of execution of a motion program to jump to that line with a **goto**, **gosub**, **callsub**, **call**, **G**, **M**, **T**, or **D** program command or a **B** on-line command.

Note that a colon must immediately follow the constant (no spaces) for this to be treated as a jump label. Without the colon, this is treated as "synchronizing status" label. It is valid to have both types of labels on a single line if both functionalities are desired.

A line only needs a jump label if the user wishes to be able to jump to that line. Line labels do not have to be in any sort of numerical order. The label must be at the beginning of a line.

By default, a program has address pointers reserved for 1024 line jump labels. If more line jump labels are desired, the number of address pointers to reserve must explicitly be declared in the **open prog** or **open plc** command that initiates the downloading of the program. It is also possible to reserve a number smaller than the default to save memory or quicken the search on a variable jump.

Note

There is by default an implied **N0:** at the beginning of every motion program. However, if you put an explicit **N0:** any other place in the program, the explicit **N0:** will be used for jumps instead of the beginning.

N*{data}*

Function: Program synchronizing status label

Syntax: **N*{data}***

where:

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the label value

This is a numeric "synchronizing status" label that cause Power PMAC to automatically set two pre-defined data-structure elements to the value of ***{data}***, one at the time program calculation reaches the label, the other at the time that execution of the next move actually begins. This permits easy monitoring of program and move-execution status by the user.

Note that no colon follows a "synchronizing status" label. If a label is followed by a colon character, it is treated as a "jump" label instead. It is valid to have both types of labels on a single line if both functionalities are desired.

At the time that program calculation encounters the synchronizing status label, it evaluates the value in ***{data}***. If necessary, it truncates this value to an unsigned 32-bit integer. It immediately sets the data-structure element **Coord[*x*].Ncalc** to this value.

It also places an instruction for the synchronous assignment of data-structure element **Coord[*x*].Nsync** to this value in the synchronous-assignment queue. At the time when the actual execution of the next move in the motion program begins, this instruction will be pulled from the queue and executed. If the program is looking ahead in a blended move sequence, the synchronous assignment can occur substantially after the initial assignment.

User monitoring of these two data elements makes it very easy to see where the program calculation and move execution points are at any instant. While primarily intended for monitoring motion program execution, it does also work for PLC programs (where it affects the elements for the coordinate system specified by **Ldata.Coord** for the program).

nofrax

Function: Clear vector feedrate axis specification

Syntax: **nofrax**

The **nofrax** command specifies that none of the axes in the coordinate system are to be considered vector feedrate axes for upcoming feedrate-specified (**F**) moves. In such a move, the Power PMAC will use the data structure element **Coord[*x*].AltFeedRate** (Isx86) for the axis with the greatest distance to calculate the move time.

The **nofrax** command works by clearing all of the bits in the **Coord[*x*].FRAxes** 32-bit word (one bit per axis). These bits will stay cleared until one or more is set by a subsequent **frax** command.

nofrax2

Function: Clear secondary feedrate-axis specification

Syntax: **nofrax2**

The **nofrax2** command specifies that none of the axes in the coordinate system are to be considered secondary vector feedrate axes for upcoming feedrate-specified (**F**) moves.

The **nofrax2** command works by clearing all of the bits in the **Coord[*x*].FR2Axes** 32-bit word (one bit per axis). These bits will stay cleared until one or more is set by a subsequent **frax2** command.

At power-on/reset, none of the axes in a coordinate system are considered secondary feedrate axes (**Coord[*x*].FR2Axes** = $0), so there is no need to issue a **nofrax2** command to empty the list initially.

The **nofrax2** command is new in V2.1 firmware, released 1st quarter 2016.

nop*{expression}*

Function: Expression/function execution without return value

Syntax: **nop*{expression}***

where:

- ***{expression}*** is a valid mathematical expression in Power PMAC syntax

The **nop** (no-operation) command permits the evaluation of a mathematical expression, particularly involving a vector, matrix, or string function, without the need to put the return value of the function into a variable. This is particularly useful for those functions whose primary result is not the returned value.

The nop statement is new in V2.0 firmware, released 1st quarter 2015.

Examples

```
// Function execution with returned values
P54 = minv(60,50,2);           // Invert matrix with determinant into P54
P100 = strcpy(256,384);        // Copy string with end index into P100
// Function execution with no returned values
nop(minv(60,50,2));            // Invert matrix with no returned determinant
nop(strcpy(256,384));          // Copy string with no returned end index
```

normal*{vector}{data}[{vector}{data}…]*

Function: Normal plane specification

Syntax: **normal***{vector}{data}[{vector}{data}…]*

where:

- ***{vector}*** is a character (I, J, or K) specifying a vector component parallel to the X, Y, or Z axis, respectively, or a double character (II, JJ, or KK) specifying a vector component parallel to the XX, YY, or ZZ axis respectively
- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the vector component

The **normal** statement specifies the orientation in the 3D-space of the X/Y/Z Cartesian axis set or of the XX/YY/ZZ Cartesian axis set of the plane for circular interpolation, 2D tool (cutter) radius compensation, and corner angle computations. It does this by defining the normal (perpendicular) vector to that plane.

The vector components that can be specified for the X/Y/Z axis set are I (X-direction), J (Y-direction), and K (Z-direction). The vector components that can be specified for the XX/YY/ZZ axis set are II (XX-direction), JJ (YY-direction), and KK (ZZ-direction). The ratio of the component values determines the orientation of the normal vector, and therefore of the plane. The length of this vector does not matter – it does not have to be a unit vector (Power PMAC automatically normalizes the vector to unit magnitude when it stores it.)

The direction sense of the vector does matter, because it defines the clockwise sense of an arc move, and the sense of cutter-compensation offset. Power PMAC uses a right-hand rule; that is, in a right-handed coordinate system (**I x J = K**), if your right thumb points in the direction of the normal vector specified here, your right fingers will curl in the direction of a clockwise arc in the circular plane, and in the direction of offset-right from direction of movement in the compensation plane. In general, the negative normal vector produces the clockwise/counterclockwise sense expected.

The power-on defaults are a K-1.0 normal vector for the X/Y/Z axis set, specifying the X/Y-plane, and a KK-1.0 normal vector for the XX/YY/ZZ axis set, specifying the XX/YY-plane.

Examples

The standard settings to produce circles in the principal planes will therefore be:

```
normal K-1;                    // XY plane -- equivalent to G17 NC code
normal J-1;                    // ZX plane -- equivalent to G18 NC code
normal I-1;                    // YZ plane -- equivalent to G19 NC code

normal KK-1;                   // XX/YY plane
normal JJ-1;                   // ZZ/XX plane
normal II-1;                   // YY/ZZ plane
```

By using more than one vector component, a plane skewed from the principal planes can be defined:

```
normal I0.866 J0.500;                // Tilted 30 deg from principal plane
normal JJ25 KK-25;                   // Tilted 45 deg from principal plane
normal J(-sin(Q1)) K(-cos(Q1));      // Tilted Q1 deg from principal plane
```

nxyz

Function: Specify surface-normal vector for 3D cutter radius compensation

Syntax: **nxyz*{vector}{data}[{vector}{data}…]***

where:

- ***{vector}*** is a character (I, J, or K) specifying a vector component parallel to the X, Y, or Z-axis, respectively
- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the vector component

The **nxyz** command specifies the surface-normal vector used for 3D cutter radius compensation by setting the components of this vector along the X, Y, and Z axes. The absolute magnitude of these components, set by the values following the I, J, and K characters, respectively, do not matter; the relative magnitudes of these components determine the orientation of the vector. The vector must be in the direction from the part surface to the tool.

The surface-normal vector must be defined in the base machine coordinates. If the XYZ space of the coordinate system has been rotated with a transformation matrix, the vector coordinates will not be the same as the programming coordinates.

The surface-normal vector defined with this command will be used to determine the compensation offset of the end of the move commanded on the same program line, and any subsequent moves until a new surface-normal vector is specified. However, it is standard practice that a surface-normal vector be specified separately for every commanded move in 3D compensation.

If a component of the surface-normal vector is not explicitly declared in an **nxyz** command, that component will automatically be set to 0.0.

P{data}{assignment operator}{expression}

Function: Assign value to specified P-variable(s)

Syntax: **P*{data}*[,*{expression}*[,*{expression}*]]**
** *{assignment operator}{expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the [first] variable to which a value is to be assigned

- the first optional ***{expression}*** specifies the number of variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is only assigned to a single variable

- the second optional ***{expression}*** specifies the "spacing" between the numbers of the multiple variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is assigned to consecutively numbered variables

- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable

- the final ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The **P*{data}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified P-variable(s). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 65,535 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be treated as a "no-operation".

P-variables are global user variables that can be given whatever function the user chooses. The specified P-variable can be named directly in its basic letter/number format (e.g. **P125**), or when using the IDE software, either by a declared `global` variable name (auto-assigned to the P-variable), or by a name given to a specific P-variable with a `#define` text substitution. In these latter cases, the substitution to the basic letter/number variable name is made automatically during the download.

Most commonly the evaluated expression after the operator is simply placed in the variable directly, using the **=** standard assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with standard assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and **--** can be used without any following expression.

If it is desired to delay the actual assignment of the value to the variable until the start of the actual execution of the next move in the program, a "synchronous assignment operator" can be

used. These are useful in motion programs for setting outputs during a blended or spline sequence of multiple moves, because the motion program must be calculating one or more moves ahead to compute how the moves are blended or splined together. With a standard assignment operator, the value would appear to get set too early. The synchronous assignment, by delaying the actual assignment until the execution of the next move starts, makes the output occur at the intuitively expected time.

Synchronous assignments are also useful in PLC programs that command motor or axis moves directly to ensure that the commanded move has started before checking to see if it has finished. For motor moves, note that the motor must be assigned to a coordinate system, because the synchronous assignment queues that hold the delayed assignment belong to coordinate systems.

Most commonly, the expression value evaluated at the time the command is found is simply placed in the variable directly at the start of execution of the next move, using the **==** synchronous assignment operator. However, this value can be arithmetically combined with the existing value of the variable with arithmetic synchronous assignment operators **+==**, **-==**, ***==**, **/==**, and **%==**. In addition, the synchronous increment and decrement operators **++=** and **--=** can be used without any following expression. However, since P-variables are floating-point, the logical synchronous variable assignment operators **&==**, **|==**, and **^==** cannot be used.

It is possible to use the same value in the assignment to multiple variables of evenly spaced numbering. If a second value is used after the initial variable name, this specifies the number of variables that will be assigned a value (if no second value is used, this defaults to 1 – the initial variable specified). If a third value is also used, this specifies the numerical spacing between these multiple variables (if no third value is used, this defaults to 1, so consecutively numbered variables are used.

Examples

```
P1=17.5;                        // Set variable P1 to 17.5
P(L10)+=3;                      // Add 3 to P-variable numbered by L10
P(P10+P17)=5*sqrt(P100);        // Set P-var numbered by (P10+P17) to expression value
P25==1;                         // Set variable P25 to 1 at start of next move execution
P50++=;                         // Increment P50 at start of next move execution
P100,16=0;                      // Set P100 - P115 to 0
P20,5,10++;                     // Increment P20, P30, P40, P50, P60
```

pause

Function: Suspend operation of motion program in specified coordinate system(s)

Syntax: **pause *[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **pause** command causes Power PMAC to suspend operation of motion programs in the specified coordinate system(s) after execution of the latest calculated move. Execution can subsequently be resumed at this point with the program **resume** or **run** command.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

The motion program execution is suspended while in paused mode, but technically it is still executing. If it is subsequently desired that program execution will not be resumed, program execution should be fully aborted with an abort command.

Note that the equivalent on-line command uses the shorter form **q**, and any coordinate-system list precedes it (e.g. **&1,2q**).

Examples

```
pause;                          // Pause motion program in addressed C.S.
pause 2;                        // Pause motion program in C.S.2
pause 1..16;                    // Pause motion program in C.S.1 thru 16
pause 3..6,8;                   // Pause motion program in C.S.3, 4, 5, 6, 8
```

pause plc

Function: Suspend operation of specified PLC program(s)

Syntax: **pause plc *{list}***

where:

- ***{list}*** is a set of one or more constants or ranges of constants from 0 to 31 specifying the PLC programs to be disabled

The **pause plc** command causes Power PMAC to suspend operation of the specified PLC program or programs at their present point of execution. Execution can subsequently be resumed at this point with the **resume plc** command.

PLC programs are specified by number (0 to 31) and may be used singularly in this command, or with multiple individual numbers separated by commas, or in a range of consecutively numbered programs.

Examples

```
pause plc 0;                    // Suspend execution of PLC 0
pause plc 1,2,5;                // Suspend execution of PLCs 1, 2, & 5
pause plc 1..16;                // Suspend execution of PLCs 1 thru 16
pause plc 3..6,8;               // Suspend execution of PLCs 3 thru 6 & 8
```

pclear

Function: Set all axis/motor positions to zero

Syntax: **pclear**

The **pclear** command causes Power PMAC to redefine the present positions for all axes in the coordinate system to zero. It also redefines the positions of all motors defined to these axes so that the axis-definition relationships are still valid. No move is made on any axis as a result of this command – the value of the present commanded position for all axes in the coordinate system.

The **pclear** command is equivalent to a **pset*{axis}*0*{axis}*0**… command, or to a **homez** command for all motors (without absolute power-up position) in the coordinate system.

If the top-level program executing this command is a motion program, the axes and motors affected are automatically those for the coordinate system executing the program. If the top-level program executing this command is a PLC program, the axes and motors affected are those for the coordinate system addressed by the PLC's data structure element **Plc[*i*].Ldata.Coord**. In a PLC program, the coordinate system can be addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

Note that unlike in PMAC and Turbo PMAC, the **pset** and similar commands (such as **pclear**) in the Power PMAC does change the motor zero position, and therefore the frame of reference for the software position limits and the compensation tables. To change the axis programming frame of reference without changing the motor zero, the axis transformation matrices should be used instead.

pload

Function: Restore saved **pset** offsets

Syntax: **pload**

The **pload** command causes Power PMAC to restore the last saved motor offsets from **pset** commands (or from before any **pset** commands are issued) for the affected coordinate system. These offsets were saved with the most recent **pstore** command. It has the effect of a **pset** command, changing the present axis and motor positions for the coordinate system, without causing any motion.

If the top-level program executing this command is a motion program, the axes and motors affected are automatically those for the coordinate system executing the program. If the top-level program executing this command is a PLC program, the axes and motors affected are those for

the coordinate system addressed by the PLC's data structure element **Plc[*i*].Ldata.Coord**. In a PLC program, the coordinate system can be addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

Note that unlike in PMAC and Turbo PMAC, the **pset** command in the Power PMAC does change the motor zero position, and therefore the frame of reference for the software position limits and the compensation tables. To change the axis programming frame of reference without changing the motor zero, the axis transformation matrices should be used instead.

pmatch

Function: Re-match axis positions to motor positions

Syntax: **pmatch**

The **pmatch** command causes Power PMAC to recalculate the axis starting positions for the coordinate system to match the present motor commanded positions. It does this by inverting the axis-definition statement equations and solving for axis position, or if there is a forward-kinematic subroutine for the coordinate system, by executing that subroutine.

This function is automatically executed (without an explicit **pmatch** command) by Power PMAC each time execution of a motion program is started with a command, to make sure the first move is calculated correctly.

However, the Power PMAC does not automatically execute this function before an axis move commanded from a program where the top-level program is a PLC program. If there is a chance that any of the motors in the coordinate system have moved since the last commanded axis move, or the axis/motor relationship has changed since the last commanded axis move, the **pmatch** command must be given first to cause Power PMAC to compute the correct axis starting positions.

If the top-level program executing this command is a motion program, the axes and motors affected are automatically those for the coordinate system executing the program. If the top-level program executing this command is a PLC program, the axes and motors affected are those for the coordinate system addressed by the PLC's data structure element **Plc[*i*].Ldata.Coord**. In a PLC program, the coordinate system can be addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If an axis move is commanded when the axis and motor positions do not agree according to the present axis/motor relationship, Power PMAC will use the wrong axis starting point in the computation of the move, and there will be a sudden commanded step to this starting point at the beginning of the move.

pread

Function: Report axis present actual positions

Syntax: **pread**

The **pread** command causes Power PMAC to calculate the present axis actual positions in the coordinate system addressed by the program in **Ldata.Coord**. The actual positions will be calculated for all active axes in the coordinate system.

The present actual position for the axis is calculated from the corresponding motor actual position(s), processed through the coordinate system definition (either by inverting axis definition statements or using the forward kinematic subroutine), and any matrix transformations in force.

The axis actual position values will be copied into local D-variables for the program, where they can be used in subsequent calculations. The positions will be returned in the present axis units, with any axis matrix transformations in force.

The D-variable **D*i*** used for each axis can be found in the following table:

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' actual positions 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.

The buffered program **pread** command performs the same calculations as the on-line **p** command.

pset*{axis}{data}[{axis}{data}…]*

Function: Redefine positions of axes and their motors

Syntax: **pset*{axis}{data}[{axis}{data}…]***

where:

- ***{axis}*** is the character or double character specifying the axis whose motor position is to be redefined (A, B, C, U, V, W, X, Y, Z, AA, BB, … HH, LL, MM, … ZZ)
- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the end position or distance
- ***[{axis}{data}…]*** is the optional specification of position redefinition for more axes

The **pset** command permits the user to redefine the present position(s) of the specified axes and the motor(s) defined to these axes. No move is made on any axis as a result of this command – the value of the present commanded position for any axis listed in the command is set to the specified value, and the motor positions for any motors defined to the axis are adjusted to keep the axis-definition relationship still valid.

If the top-level program executing this command is a motion program, the axes and motors affected are automatically those for the coordinate system executing the program. If the top-level program executing this command is a PLC program, the axes and motors affected are those for the coordinate system addressed by the PLC's data structure element **Plc[*i*].Ldata.Coord**. In a PLC program, the coordinate system can be addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

Note that unlike in PMAC and Turbo PMAC, the **pset** command in the Power PMAC does change the motor zero position, and therefore the frame of reference for the software position limits and the compensation tables. To change the axis programming frame of reference without changing the motor zero, the axis transformation matrices should be used instead.

Internally, this command changes the value of the data structure element **Motor[*x*].HomePos**, for each motor defined to an axis named in the command. This register holds the difference between the motor zero position and the underlying position sensor (e.g. encoder counter) zero position.

Examples

```
pset X10 Y20;                  // Set present X position to 10 units, Y to 20
pset A(P1) Y(P2);              // Set present A position to value of P1, B to P2
```

pstore

Function: Save present **pset** offsets

Syntax: **pstore**

The **pstore** command causes Power PMAC to save the present motor offsets from **pset** commands (or from before any **pset** commands are issued) for the affected coordinate system. These offsets can then be restored with a **pload** command. The **pstore** command does not change the active offsets or cause any motion. Only one set of offsets can be stored in this way for a coordinate system.

If the top-level program executing this command is a motion program, the axes and motors affected are automatically those for the coordinate system executing the program. If the top-level program executing this command is a PLC program, the axes and motors affected are those for the coordinate system addressed by the PLC's data structure element **Plc[*i*].Ldata.Coord**. In a PLC program, the coordinate system can be addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

Note that unlike in PMAC and Turbo PMAC, the **pset** command in the Power PMAC does change the motor zero position, and therefore the frame of reference for the software position limits and the compensation tables. To change the axis programming frame of reference without changing the motor zero, the axis transformation matrices should be used instead.

pvt*{data}*

Function: Set position-velocity-time move mode

Syntax: **pvt*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the move time in milliseconds

The **pvt** command puts the motion program in the position-velocity-time move mode. The value in ***{data}*** is evaluated and used as the time, in milliseconds, for each programmed move. If the motion program is already in PVT mode, the command can be used to change the time for the moves (this can be done without stopping). Subsequent move commands in the motion program will be processed according to the rules of this mode.

Note that the PVT move times cannot be set or changed with **ta** or **tm** commands in Power PMAC, as they could be in Turbo PMAC.

The **pvt** command takes the program out of any of the other move modes (**circle**, **linear**, **rapid**, **spline**). A command for any of these other move modes takes the program out of **pvt** mode.

Examples

```
pvt200;                    // Set PVT mode with move time of 200 msec
pvt(L10);                  // Set PVT mode with move time of (L10) msec
```

Q{data}{assignment operator}{expression}

Function: Assign value to specified Q-variable(s)

Syntax: **Q*{data}*[,*{expression}*[,*{expression}*]**
** *{assignment operator}{expression}***

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the [first] variable to which a value is to be assigned

- the first optional ***{expression}*** specifies the number of variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is only assigned to a single variable

- the second optional ***{expression}*** specifies the "spacing" between the numbers of the multiple variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is assigned to consecutively numbered variables

- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable

- the final ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The **Q*{data}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified Q-variable(s). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be treated as a "no-operation".

Q-variables are global (usable by multiple programs) but coordinate-system-specific (independent between different coordinate systems) user variables that can be given whatever function the user chooses. The specified Q-variable can be named directly in its basic letter/number format (e.g. **Q125**), or when using the IDE software, either by a declared `csglobal` variable name (auto-assigned to the Q-variable), or by a name given to a specific Q-variable with a `#define` text substitution. In these latter cases, the substitution to the basic letter/number variable name is made automatically during the download.

Most commonly, the evaluated expression after the operator, this value can be arithmetically or logically combined with the existing value of the variable with standard assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and **--** can be used without any following expression.

If it is desired to delay the actual assignment of the value to the variable until the start of the actual execution of the next move in the program, a "synchronous assignment operator" can be used. These are useful in motion programs for setting outputs during a blended or spline sequence of multiple moves, because the motion program must be calculating one or more moves ahead to compute how the moves are blended or splined together. With a standard assignment operator, the value would appear to get set too early. The synchronous assignment, by delaying the actual assignment until the execution of the next move starts, makes the output occur at the intuitively expected time.

Synchronous assignments are also useful in PLC programs that command motor or axis moves directly to ensure that the commanded move has started before checking to see if it has finished. For motor moves, note that the motor must be assigned to a coordinate system, because the synchronous assignment queues that hold the delayed assignment belong to coordinate systems.

Most commonly, the expression value evaluated at the time the command is found is simply placed in the variable directly at the start of execution of the next move, using the **==** synchronous assignment operator. However, this value can be arithmetically combined with the existing value of the variable with arithmetic synchronous assignment operators **+==**, **-==**, ***==**, **/==**, and **%==**. In addition, the synchronous increment and decrement operators **++=** and **--=** can be used without any following expression. However, since Q-variables are floating-point, the logical synchronous variable assignment operators **&==**, **|==**, and **^==** cannot be used.

It is possible to use the same value in the assignment to multiple variables of evenly spaced numbering. If a second value is used after the initial variable name, this specifies the number of variables that will be assigned a value (if no second value is used, this defaults to 1 – the initial

variable specified). If a third value is also used, this specifies the numerical spacing between these multiple variables (if no third value is used, this defaults to 1, so consecutively numbered variables are used.

Examples

```
Q1=17.5;                        // Set variable Q1 to 17.5
Q(L10)+=3;                      // Add 3 to Q-variable numbered by L10
Q(P10+P17)=5*sqrt(P100);        // Set Q-var numbered by (P10+P17) to expression value
Q25==1;                         // Set variable Q25 to 1 at start of next move execution
Q50++=;                         // Increment Q50 at start of next move execution
Q100,16=0;                      // Set Q100 - Q115 to 0
Q20,5,10++;                     // Increment Q20, Q30, Q40, Q50, Q60
```

R{data}{assignment operator}{expression}

Function: Assign value to specified R-variable(s)

Syntax: **R*{data}[,{expression}[,{expression}]***
{assignment operator}{expression}

where:

- ***{data}*** is a non-negative integer constant (without parentheses) or a mathematical expression in parentheses representing the number of the [first] variable to which a value is to be assigned
- the first optional ***{expression}*** specifies the number of variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is only assigned to a single variable
- the second optional ***{expression}*** specifies the "spacing" between the numbers of the multiple variables to which a (single) value is to be assigned. If this is not explicitly specified, 1 is used, so a value is assigned to consecutively numbered variables
- ***{assignment operator}*** is the mathematical operator that controls how the value from the expression is assigned to the variable
- the final ***{expression}*** is the mathematical expression – combination of constants, variables, logical and arithmetic operators, and functions – whose value is evaluated to contribute to the assignment

The **R*{data}{assignment operator}{expression}*** command causes the value evaluated from the expression to be used to place a value in the specified R-variable(s). If the variable number is specified with a constant, the constant must be an integer value representing a valid variable number (0 – 8,191 in the standard Power PMAC firmware); otherwise the command will be rejected with an error. If the variable number is specified with a mathematical expression, the expression must evaluate to a valid variable number (rounded down to the next integer if necessary); otherwise the command will be treated as a "no-operation".

R-variables are local stack variables used within a program and to pass values to lower-level routines. Each coordinate system has its own set of R-variables for use by top-level motion programs and their subprograms. In addition, each PLC program has its own set, and each communications thread has its own set. All of these sets of R-variables are independent of each other.

Most commonly, the evaluated expression after the operator is simply placed in the variable directly, using the **=** assignment operator. However, this value can be arithmetically or logically combined with the existing value of the variable with assignment operators **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **>>=**, and **<<=**. In addition, the increment and decrement operators **++** and **--** can be used without any following expression.

Since R-variables are local variables, they cannot use the delayed synchronous assignment operators available to global variables, because their "context" could be lost by the time of the actual assignment.

It is possible to use the same value in the assignment to multiple variables of evenly spaced numbering. If a second value is used after the initial variable name, this specifies the number of variables that will be assigned a value (if no second value is used, this defaults to 1 – the initial variable specified). If a third value is also used, this specifies the numerical spacing between these multiple variables (if no third value is used, this defaults to 1, so consecutively numbered variables are used.

Examples

```
R1=17.5;                    // Set variable R1 to 17.5
R(P1)+=3;                   // Add 3 to R-variable numbered by P1
R(P1+P17)=5*sqrt(P100);     // Set R-var numbered by (P1+P17) to expression value
R10,16=0;                   // Set R10 - R25 to 0
R20,5,10++;                 // Increment R20, R30, R40, R50, R60
```

rapid

Function: Set rapid-traverse move mode

Syntax: **rapid**

The **rapid** command puts the motion program in the rapid-traverse move mode, which is intended to provide minimum-time point-to-point moves. Subsequent move commands in the motion program will be processed according to the rules of this mode.

The **rapid** command takes the program out of any of the other move modes (**circle**, **pvt**, **linear**, **spline**). A command for any of these other move modes takes the program out of **rapid** mode.

read

Function: Read arguments for subroutine

Syntax: **read(*{letter list}*)**

where:

- ***{letter}*** is a single or double letter character specifying the argument to be sent (A, B, … , Z, AA, BB, … , ZZ)

- ***{letter list}*** is a set of letters (single or double) separated by commas, and/or ranges of consecutive axes denoted by two periods between starting and ending letters. All letters used must be in alphabetical order, single-letter first, followed by double-letter.

The **read** command allows a subroutine or subprogram to take arguments from the calling routine that are provided in letter/number form. It is particularly useful for creating parameterized subroutines that are callable from standard RS-274 "G-code" programs common in CAD/CAM and CNC applications. No spaces are permitted anywhere within the command.

The **read** command looks at the remainder of the line in the calling program that contains the calling command (**gosub**, **callsub**, **call**, **G**, **M**, **T**, **D**), takes the values following the specified letters, and puts them into particular D-variables for the coordinate system (if the top-level program is a motion program) or PLC (if the top-level program is a PLC program).

For single letters, the value for the *n*th letter of the alphabet is put in **D*n***. For double letters, the value for the *n*th letter of the alphabet is put in D(26+*n*). There is only one set of D-variables per coordinate system (for when the top-level program is a motion program) and per PLC program, so if there are nested subroutines/subprograms using **read** commands, the same set of D-variables is used each time.

The **read** command scans the calling line until it sees a letter that is not in the list of letters to be read, or something other than a letter/value combination. (Note that if it encounters a letter not to be read, it stops, even if some letters in the list have not yet been read.) Each letter value successfully read into a D-variable causes a bit to be set in **D0**, noting that it was read (bit *n*-1 for the *n*th letter of the alphabet for single letters, bit *n*+25 for double letters). For any letter not successfully read in the most recent **read** command, the corresponding bit of **D0** is set to zero.

Starting in V2.0 firmware, released 1st quarter 2015, the same bits are set in **D54** as in **D0**. **D0** is also used in kinematics routines, and has a slight chance of overwriting the value of **D0** set here. **D54** has no other use than in the **read** command.

If a letter value is read from the calling line, the normal function of the letter (e.g. an axis move) is overridden, so that letter serves merely to pass a parameter to the subroutine. If there are remaining letter values on the calling line that are not read, those will be executed according to their normal function after the return from the subroutine.

The D-variable and flag bit of D0 associated with each letter 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

Examples

```
read(a,b,c);                    // Read values associated with A, B, and C
read(a..z);                     // Read values associated with all single letters
read(d..h,q,aa..gg);            // Read valuese associated with D thru H, Q, AA thru GG
```

resume

Function: Restart suspended motion program in specified coordinate system(s)

Syntax: **resume** ***[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **resume** command causes Power PMAC to restart operation of suspended motion programs in the specified coordinate system(s) at the point where execution was suspended. It is typically used when execution has been suspended with the **pause** command.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program:

Plc[*i*].Ldata.Coord. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Examples

```
resume;                     // Resume motion program in addressed C.S.
resume 2;                   // Resume motion program in C.S.2
resume 1..16;               // Resume motion program in C.S.1 thru 16
resume 3..6,8;              // Resume motion program in C.S.3, 4, 5, 6, 8
```

resume plc

Function: Resume execution of specified PLC program(s)

Syntax: **resume plc *{list}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to 31, specifying the numbers of the PLC programs whose execution is to be resumed.

The **resume plc** command causes Power PMAC to permit the execution of the specified PLC program(s) at their normal priority and timing. Execution will start at the point where execution was halted, even if it was halted in the middle of the program (e.g. with a **pause plc** command).

The similar **enable plc** command can be used to re-start PLC program execution at the beginning of the program, even if execution was halted elsewhere.

If a PLC program specified in the command is not present in the Power PMAC, no error will be reported, and the command will still operate on any other exiting PLC programs specified in the command.

Examples

```
resume plc 1;               // Resume execution of PLC 1
resume plc 2,4,6;           // Resume execution of PLCs 2, 4, and 6
resume plc 7..10;           // Resume execution of PLCs 7, 8, 9, and 10
resume plc 11,13..16,20;    // Resume execution of PLCs 11, 13, 14, 15, 16, and 20
```

return

Function: Return from subroutine jump/End main program

Syntax: **return**

The **return** command causes program execution to jump back to the routine that called the execution of this routine. If the **return** command is encountered at the "top" level of execution, the program execution pointer is returned to the beginning of the program. In the case of a motion program (prog), program execution is stopped. In the case of a PLC program, program execution will continue on the next scan unless it has been disabled before the next scan is to start.

If the **return** command is encountered during execution of a subroutine or subprogram, program execution jumps back to the command immediately following the calling command and continues there.

There is an implicit **return** command at the end of every program buffer, so when program execution reaches the end of the buffer, a **return** command is executed even if not explicitly entered into the program buffer.

run

Function: Run motion program in specified coordinate system(s)

Syntax: **run** ***[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **run** command causes Power PMAC to start continuous execution of the already selected motion program(s) in the specified coordinate system(s) at the present point of the program counter.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Typically, the motion program for the coordinate system was selected initially with the **begin** command. If motion-program execution was halted with a command such as **step**, **pause**, or **hold**, the **run** command will cause execution to start where it had been stopped.

The coordinate system must be in a proper condition in order for Power PMAC to accept this command for the coordinate system. Otherwise, the Power PMAC will reject this command with an error. The following conditions must hold for the coordinate system for this command to be accepted:

- The selected motion program (and any label selected) must be present and valid.
- All motors assigned to position axes in the coordinate system must be activated (**Motor[*x*].ServoCtrl** > 0).
- All motors assigned to position axes in the coordinate system must be enabled and closed-loop.
- No motor assigned to a position axis in the coordinate system may have both overtravel limits set.
- Power PMAC must be able to compute all starting axis positions from present motor positions (**Coord[*x*].Csolve** = 1).
- If **Coord[*x*].HomeRequired** = 1, all motors assigned to position axes in the coordinate system must have established a position reference through a successful homing search move or absolute position read.
- If re-starting from a suspended state that is not at a programmed point, all motors must be at the same commanded position as they were when initially stopped in the suspended state.

Note that the equivalent on-line command uses the shorter form **r**, and any coordinate-system list precedes it (e.g. **&1,2r**).

Examples

```
run;                        // Start motion program in addressed C.S.
run 2;                      // Start motion program in C.S.2
run 1..16;                  // Start motion program in C.S.1 thru 16
run 3..6,8;                 // Start motion program in C.S.3, 4, 5, 6, 8
```

S{data}

Function: Spindle data command (S-Code)

Syntax: **S*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the value to be passed to the storage variable for later use

The **S*{data}*** command causes Power PMAC to load the value in ***{data}*** into the local variable D53 for the coordinate system executing the program. It takes no other action. It is intended to pass spindle-speed information in RS-274-style machine tool programs. The algorithms that actually control the spindle would then use D53 in their routines; for instance to set jog speed for a closed-loop spindle, or voltage output for an open-loop spindle.

There is no other automatic use of D53, so the value can safely be left in this variable until needed as long as the user application does not use it for any other purpose. However, it may be desirable to copy the value into a global variable for subsequent use. The value in D53 can be accessed from outside the motion program and its subroutines using the data structure element **Coord[*x*].Ldata.D[53]**.

Note

This command is distinct from **S*{data}*** information passed as part of a subroutine call through a **read(S)** command; in that form, the value is placed in local variable D19 for the coordinate system.

send

Function: Send formatted text string to specified port

Syntax: **send*{constant}*[,]"*{string}*"[,*{expression}*…]**

where:

- ***{constant}*** is an integer in the range of 0 to 7 (inclusive) specifying the software communications port over which the string will be sent

- ***{string}*** is a formatted ASCII text string consisting of literal alphanumeric characters, escape sequences, and formatted variable sequences, as explained below

- ***{expression}*** is a mathematical expression containing the value that is to be sent for the matching formatted variable sequence in the string. There must be one expression for each formatted variable sequence.

The **send** command causes the Power PMAC to transmit a formatted text string to the specified software communications port. The constant value immediately following the **send** – 0 - 7, indicates the software port number over which the string will be sent. The host computer should have started the Power PMAC "send getsends" utility with a `send getsends -n` (where *n* is the matching port number) telnet command so these strings can be transmitted to the host. (In the IDE, the "Unsolicited Messages" window sets this up automatically and displays sent strings, and so is an excellent tool for debugging applications.)

The capability to use software port numbers 5, 6, and 7 was added in V2.0 firmware, released 1st quarter 2015.

When the command is processed as part of program execution, the string is created and buffered for transmission over the specified port. Once the string is buffered, program execution continues; it does not wait until the string is transmitted over the port. However, if a previously created string for the same port is still in the buffer, not yet having been transmitted, program execution waits until the new string can be placed in the buffer.

The string to be sent is a combination of standard ASCII characters, "escape sequences" that are started with a "backslash" (\) character and permit the output of control and special characters, and "format sequences" that are started with a "percent" (%) character and specify how the numerical value of an expression is to be formatted as a string. For each format sequence, there must be a matching expression following the string.

The following escape sequences can be used (all letters must be lower case):

- `\a` Bell (ASCII 7)
- `\b` Backspace (ASCII 8)
- `\t` Horizontal tab (ASCII 9)
- `\n` New line (ASCII 10)
- `\v` Vertical tab (ASCII 11)
- `\f` Form feed (ASCII 12)
- `\r` Carriage return (ASCII 13)
- `\\` Backslash character
- `\?` Question-mark character
- `\'` Single-quote character
- `\"` Double-quote character
- `\ooo` Octal specification of ASCII character code (of range `000` to `377`)
- `\xhh` Hexadecimal specification of ASCII character code (of range `x00` to `xff`)

The following format sequences can be used (all letters must be used in upper or lower case as shown):

- `%d` Signed integer, decimal format
- `%u` Unsigned integer, decimal format
- `%x` Unsigned integer, hexadecimal format, use lower case
- `%n`x Unsigned integer, hexadecimal format, using *n* digits (*n* = 1 to 8), use lower case
- `%X` Unsigned integer, hexadecimal format, use upper case
- `%n`X Unsigned integer, hexadecimal format, using *n* digits (*n* = 1 to 8), use upper case
- `%f` Floating-point value, up to 6 digits total
- `%n`f Floating-point value, up to *n* digits total (*n* = 1 to 31)
- `%n.m`f Floating-point value, up to *n* digits total (*n* = 1 to 31), up to *m* fractional digits (*m* < *n*)
- `%s` Text string, arbitrary length (null terminated)
- `%n`s Text string, up to *n* characters (shorter if null terminator encountered)
- `%c` Single character of specified byte value

- `%nc` *n* repetitions of single character of specified byte value
- `%%` "Percent" character

Leading zeros are not created for decimal-format integers. Leading zeros are created for hexadecimal-format integers (only) when the number of digits is specified. Leading and trailing zeros are not created for floating-point values, whether or not the number of digits is specified. If the number of digits specified for a floating-point is not sufficient to represent the value without an exponent, it will automatically be represented with an exponent (e.g. `2.345e8`). The decimal point is only used for floating-point values if there is a non-zero fractional component. In floating-point values, any decimal point, minus sign, or exponent used does not count as a digit.

For each formatted variable sequence in the string, there must be an ***{expression}*** that specifies the numerical value that is to be converted to text. Each expression can be as simple as a constant or a variable, but can include mathematical operators and expressions as well. For a string variable, the expression value (rounded down to the next integer if necessary) represents the starting index in user shared memory of the string variable to be used.

If bit *n* of saved setup element **Sys.SendFileMode** is set to 1, then the characters "-n:" are automatically appended to the beginning of any string sent to port *n*, and the "null" character (ASCII 0) is automatically appended to the end.

In a telnet communications session, the Power PMAC "send getsends" utility is started with the general syntax `send getsends -n [-n2…] [-b] [-fname]` , where *n*, *n2*, etc. are the numbers of the software communications ports to activate for potential unsolicited communications, the optional "`-b`" implements binary mode (without binary mode, the Power PMAC cannot send the "null" character without terminating the string, and Power PMAC automatically appends the port number to the beginning of the string), and the optional *fname* enables the writing of the string into a file by specifying the name of that file.

If bit *n* of saved setup element **Sys.SendFileMode** is set to 1, then each string sent to port *n* is pre-appended with the "***-n:***" characters and post-appended with the "null" character.

Examples

```
send0,"Hello, World!";                  // Pure literal string to Port 0
send1,"Hello, World!\n";                // String with "new line" control character at end
send2,"Cycle counter=%u\n",P100;        // String with unsigned integer value
send3,"Xpos: %9fmm Ypos: %9fmm\n",Q107*25.4,Q108*25.4;
                                        // String with 2 floating-point expressions
send4,"App error #%u, Code %s\n",P20,1024;
                                        // Text with integer and string varaibles
```

sendall

Function: Send all buffered "send" strings

Syntax: **`sendall`**

The **`sendall`** command causes the Power PMAC to force the output of all pending text strings from previous **`send`** commands on all communications ports. Program execution is halted until

all pending strings are sent. This command is useful for ensuring that all previously created strings have been flushed from the buffers.

sendallcmds

Function: Send all buffered "cmd" strings

Syntax: **sendallcmds**

The **sendallcmds** command causes the Power PMAC to force the execution of all pending text strings from previous **cmd** commands from all programs. Program execution is halted until all pending strings are executed. This command is useful for ensuring that all previously created strings have been flushed from the buffers.

sendallsystemcmds

Function: Send all buffered "system" strings

Syntax: **sendallsystemcmds**

The **sendallsystemcmds** command causes the Power PMAC to force the execution of all pending text strings from previous **system** commands from all programs. Program execution is halted until all pending strings are executed. This command is useful for ensuring that all previously created strings have been flushed from the buffers.

spline*{data}*

Function: Set cubic B-spline move mode and time(s)

Syntax: **spline*{data0}*[spline*{data1}*[spline*{data2}*]]**

where:

- ***{datan}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the move section time in milliseconds

The **spline** command puts the motion program in the cubic B-spline move mode and sets the spline move time(s), expressed in milliseconds. If the motion program is already in the cubic B-spline mode, the command can be used to change the time for the moves (this can be done without stopping). Subsequent move commands in the motion program will be processed according to the rules of this mode.

In the cubic B-spline mode, each programmed move consists of three sections. In the middle of a spline mode sequence, the sections for each programmed move overlap sections from the previous and following programmed moves, with the resulting profile derived from the superposition of these three moves.

The time for the first section is held in status data structure element **Coord[*x*].T0Spline**. The time for the second section is held in **Coord[*x*].T1Spline**. The time for the third section is held in **Coord[*x*].T2Spline**.

For each programmed move in **spline** mode, only the first section is fully calculated (superimposed on later sections of earlier moves, if present). The second and third sections are only tentatively calculated, as the times for these sections can be changed with subsequent **spline** commands and programmed move commands.

The exact action of a **spline** command depends on whether the program is already executing a spline-mode sequence or not, and on the setting of saved setup element **Coord[*x*].SplineTimeRotate**.

At the start of a sequence, if only a single time is specified (e.g. **spline50**), all three sections use this specified time. If two times are specified (e.g. **spline50 spline75**), the first time is used for the first section (**T0Spline**) and the second time is used for the second and third sections (**T1Spline** and **T2Spline**). If three times are specified (e.g. **spline50 spline75 spline100**), each time is used for the comparable section of the move.

If a sequence is already executing with **Coord[*x*].SplineTimeRotate** set to its default value of 0, if only a single time is specified, only the time for the third section (**T2Spline**) is set to the specified value. The times for the first two sections (**T0Spline** and **T1Spline**) are left at their previous values.

Setting **Coord[*x*].SplineTimeRotate** to 1 puts Power PMAC in a mode compatible with the older PMACs in terms of changing spline section times. In this case, a **spline*{data}*** command only changes the time for the third section (**T2Spline**) of the next move. (If multiple times are specified on the same line, each one overwrites the previous one, so only the last time specified is used.)

However, in the second move after this command, this new time is also used as the time for the second section (**T1Spline**), and in the third move, it is also used for the first section (**T0Spline**). In this way, the spline time is changed over three moves, and no section time is recalculated.

Note that the spline section times cannot be set or changed with **ta** or **tm** commands in Power PMAC, as they could be in Turbo PMAC.

The **spline** command takes the program out of any of the other move modes (**circle**, **linear**, **rapid**, **pvt**). A command for any of these other move modes takes the program out of **spline** mode.

Examples

```
spline 200;                          // Set spline mode with all 3 section times of 200 msec
spline (L10);                        // Set spline mode with all 3 section times of (L10) msec
spline 200 spline 300;               // Set spline mode with 1st section time of 200 msec,
                                     // 2nd and 3rd section times of 300 msec
spline 200 spline 300 spline 400;    // Set spline mode with 1st section time
                                     // of 200 msec, 2nd section time of 300 msec;
                                     // 3rd section time of 400 ms
```

start

Function: Point program counter to specified motion program and run

Syntax: **start*[{list}]*:*{data}***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

- ***{data}*** is a floating-point constant (without parentheses) or expression (in parentheses) specifying the motion program number (integer part) and optionally numeric line jump label (fractional part multiplied by 1,000,000)

The **start** command causes the addressed or listed coordinate system(s) to set their program counter(s) to the beginning or specified numeric line jump label of the specified motion program, and to commence continuous execution of the program at this point.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

The integer part of ***{data}*** (i.e. rounded down) specifies the number of the motion program. If there is no fractional part, the program counter will point to the beginning of this motion program. If there is a fractional part, the fraction is multiplied by 1,000,000 to determine the numeric line jump label to which the program counter will point.

The coordinate system must be in a proper condition in order for Power PMAC to accept this command for the coordinate system. Otherwise, the Power PMAC will reject this command with an error. The following conditions must hold for the coordinate system for this command to be accepted:

- The selected motion program (and any label selected) must be present and valid.

- All motors assigned to position axes in the coordinate system must be activated (**Motor[*x*].ServoCtrl** > 0).

- All motors assigned to position axes in the coordinate system must be enabled and closed-loop.
- No motor assigned to a position axis in the coordinate system may have both overtravel limits set.
- Power PMAC must be able to compute all starting axis positions from present motor positions (**Coord[*x*].Csolve** = 1).
- If **Coord[*x*].HomeRequired** = 1, all motors assigned to position axes in the coordinate system must have established a position reference through a successful homing search move or absolute position read.
- If re-starting from a suspended state that is not at a programmed point, all motors must be at the same commanded position as they were when initially stopped in the suspended state.

Example

```
start:2;                        // Point addressed C.S. to top of prog 2 and run
start1:75;                      // Point C.S.1 to top of prog 75 and run
start3:75.001;                  // Point C.S.3 to jump label N1000: of prog 75 and run
start8:(P1+P2/1000000);         // Point C.S.8 to jump label specified by P2 in motion
                                // program specified by P1, and run
```

step

Function: Single-step motion program in specified coordinate system(s)

Syntax: **step** ***[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **step** command causes Power PMAC to start single-step execution of the already selected motion program(s) in the specified coordinate system(s) at the present point of the program counter.

The exact behavior of the coordinate system in response to this command is dependent on the setting of saved setup element **Coord[*x*].StepMode**.

If the coordinate system is already executing a motion program when this command is sent, the command puts the program in single-step mode. If **Coord[*x*].StepMode** is set to 0, one more move will be calculated, and execution will stop at the end of this next move. In this case, its action is similar to the **q** command, although it will execute an additional move compared to **q**. If **Coord[*x*].StepMode** is set to a value greater than 0, no more moves will be calculated, and execution will stop at the end of the most recently calculated move.

If program execution has already stopped in single-step mode, a step command will cause program execution to advance until one move is executed if **Coord[*x*].StepMode** is set to 0. It will cause one program line to be executed if **Coord[*x*].StepMode** is set to a value greater than 0. (The different non-zero values of **Coord[*x*].StepMode** lead to slightly different behavior when 2D tool-radius compensation is enabled.)

Regardless of the setting of **Coord[*x*].StepMode**, if single-step execution encounters a **bstart** command in the motion program, it will continue until (and only until) the next **bstop** command in the motion program, regardless of how many moves or program lines are included.

If the command is not immediately followed by a coordinate-system list, it will act on the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Typically, the motion program for the coordinate system was selected initially with the **begin** command. If motion-program execution was halted with a command such as **step**, **pause**, or **hold**, the **step** command will cause execution to start where it had been stopped.

The motion program execution is suspended when a step has finished, but technically it is still executing. If it is subsequently desired that program execution will not be resumed, program execution should be fully aborted with an abort or stop command.

The coordinate system must be in a proper condition in order for Power PMAC to accept this command for the coordinate system. Otherwise, the Power PMAC will reject this command with an error. The following conditions must hold for the coordinate system for this command to be accepted:

- The selected motion program (and any label selected) must be present and valid.
- All motors assigned to position axes in the coordinate system must be activated (**Motor[*x*].ServoCtrl** > 0).
- All motors assigned to position axes in the coordinate system must be enabled and closed-loop.
- No motor assigned to a position axis in the coordinate system may have both overtravel limits set.
- Power PMAC must be able to compute all starting axis positions from present motor positions (**Coord[*x*].Csolve** = 1).

- If **Coord[*x*].HomeRequired** = 1, all motors assigned to position axes in the coordinate system must have established a position reference through a successful homing search move or absolute position read.

- If re-starting from a suspended state that is not at a programmed point, all motors must be at the same commanded position as they were when initially stopped in the suspended state.

Note that the equivalent on-line command uses the shorter form **s**, and any coordinate-system list precedes it (e.g. **&1,2s**).

Examples

```
step;                       // Single-step motion program in addressed C.S.
step 2;                     // Single-step motion program in C.S.2
step 1..16;                 // Single-step motion program in C.S.1 thru 16
step 3..6,8;                // Single-step motion program in C.S.3, 4, 5, 6, 8
```

step plc

Function: Do single step of specified PLC program(s)

Syntax: **step plc *{list}***

where:

- ***{list}*** is a set of one or more constants or ranges of constants from 0 to 31 specifying the PLC programs to be enabled

The **step plc** command causes Power PMAC to execute a single program step of the specified PLC program or programs in their normal order of execution. Each specified PLC program will start execution at the point where execution was last suspended. PLC programs are specified by number (0 to 31) and may be used singularly in this command, or with multiple individual numbers separated by commas, or in a range of consecutively numbered programs.

Examples

```
step plc 0;                 // Single-step execution of PLC 0
step plc 1,2,5;             // Single-step execution of PLCs 1, 2, & 5
step plc 1..16;             // Single-step execution of PLCs 1 thru 16
step plc 3..6,8;            // Single-step execution of PLCs 3 thru 6 and 8
```

stop

Function: Halt operation of motion program in specified coordinate system(s)

Syntax: **stop *[{list}]***

where:

- ***{list}*** is a set of integer constants, separated by commas, and ranges of consecutive integer constants denoted by two periods between starting and ending constant numbers, in the range 0 to one less than the number of coordinate systems supported by the particular version of Power PMAC firmware.

The **stop** command causes Power PMAC to halt operation of motion programs in the specified coordinate system(s) after execution of the latest calculated move. The program counter is then automatically set back to the beginning of this motion program, so execution cannot subsequently be resumed at the stopped point.

If not immediately followed by a coordinate-system list, it will do so for the coordinate system that is presently addressed by the program. If immediately followed by a coordinate-system list, it will do so for all coordinate systems in the list. Specifying a coordinate system in a list does not affect the program's modally addressed coordinate system.

If the top-level program issuing this command is a PLC program, the addressed coordinate system is determined by an element in the data structure for the PLC program: **Plc[*i*].Ldata.Coord**. In a PLC program, a coordinate system can be modally addressed with the command **Ldata.Coord=*{expression}*** – the parent structure name is not needed.

If the top-level program issuing this command is a motion program, the modally addressed coordinate system (**Coord[*x*].Ldata.Coord**) cannot be changed from the one executing the program, so if a motion program is to issue this command to another coordinate system, it must do so by listing that coordinate system.

Examples

```
stop;                          // Halt motion program in addressed C.S.
stop 2;                        // Halt motion program in C.S.2
stop 1..16;                    // Halt motion program in C.S.1 thru 16
stop 3..6,8;                   // Halt motion program in C.S.3, 4, 5, 6, 8
```

switch (*{expression}*)

Function: Conditional multiple-case branch

Syntax: **switch(*{expression}*)*[{constant}]***

where:

- ***{expression}*** is a mathematical expression controlling which branch will be executed

- ***{constant}*** is an optional positive integer constant representing the maximum number of cases that can be used under this switch command. If no value is specified here, a value of 256 is used.

Full Structure Syntax:

```
switch({expression}) [{constant}]
{
```

```
    case{constant}:{command}[{command}…][break}
    [case{constant}:{command}[{command}…][break}]
    [default:{command}[{command}…]
}
```

The **switch** command permits multiple-case branching in programs. The expression inside the parentheses is evaluated when the command is executed. If it does not evaluate exactly to an integer value, the resulting value is rounded down to the next integer.

The optional constant value after the expression specifies the maximum number of cases that be used under this command. Smaller maximum-number values mean quicker execution (regardless of the actual number of cases used). If no value is specified, Power PMAC uses a value of 256. The IDE project manager automatically specifies the smallest number that supports the number of cases used.

The next character in the program after this command, whether following on the same program line or at the start of the next program line, must be a left "curly bracket" (**{**) to indicate the start of the possible branches. This starting bracket is followed by one or more **case** commands. Each **case** command is followed by an integer constant and a colon; the colon is followed by the set of commands to be executed when the expression value in the **switch** command matches the constant value in that **case** command. Commands continue executing until a **break** command is encountered, or the ending right curly bracket is reached.

There can also be a **default** command as the last of set of branches following the starting bracket. This is followed by a colon and the set of commands to be executed if the expression value in the **switch** command does not match any of the constant values in the set of **case** commands. If there is no match for the expression value and no **default** command, no actions will be taken due to this **switch** command.

The set of possible cases that can be selected by a **switch** command, including a possible **default** case, must be followed by a right curly bracket (**}**) at the start of the next program line.

Examples

```
switch (P1) {                          // Branch based on value of P1
  case 1: X10 F5; break;               // Action if P1 = 1
  case 2: X17 F7; break;               // Action if P1 = 2
  case 3: X-3 F2; break;               // Action if P1 = 3
  default: dwell 100;                  // Action if P1 != 1, 2, or 3
}

switch (P0) {                          // Branch based on value of P0
  case 4: P4++;                        // Action if P0 = 4
  case 3: P3++;                        // Action if P0 = 3 or 4
  case 2: P2++;                        // Action if P0 = 2, 3, or 4
  case 1: P1++;                        // Action if P0 = 1, 2, 3, or 4
}
```

system

Function: Issue command to operating system command parser

Syntax: **system***[,]"{string}"[,{expression}…]*

where:

- ***{string}*** is a formatted ASCII text string consisting of literal alphanumeric characters, escape sequences, and formatted variable sequences, as explained below
- ***{expression}*** is a mathematical expression containing the value that is to be sent for the matching formatted variable sequence in the string. There must be one expression for each formatted variable sequence.

The **system** program command causes the Power PMAC to issue the formatted text string as a command to the operating system of the Power PMAC, just as if it were sent from an external computer in a Telnet session.

When the command is processed as part of program execution, the string is created and buffered for processing by the operating system. Once the string is buffered, program execution continues; it does not wait until the string is processed by the thread.

The string to be sent is a combination of standard ASCII characters, “escape sequences” that are started with a “backslash” (\) character and permit the output of control and special characters, and “format sequences” that are started with a “percent” (%) character and specify how the numerical value of an expression is to be formatted as a string. For each format sequence, there must be a matching expression following the string.

The following escape sequences can be used (all letters must be lower case):

- \a Bell (ASCII 7)
- \b Backspace (ASCII 8)
- \t Horizontal tab (ASCII 9)
- \n New line (ASCII 10)
- \v Vertical tab (ASCII 11)

- \f Form feed (ASCII 12)
- \r Carriage return (ASCII 13)
- \\ Backslash character
- \? Question-mark character
- \' Single-quote character
- \" Double-quote character
- *ooo* Octal specification of ASCII character code (of range 000 to 377)
- \x*hh* Hexadecimal specification of ASCII character code (of range x00 to xff)

The following format sequences can be used (all letters must be used in upper or lower case as shown):

- %d Signed integer, decimal format
- %u Unsigned integer, decimal format
- %x Unsigned integer, hexadecimal format, use lower case
- %*n*x Unsigned integer, hexadecimal format, using *n* digits (*n* = 1 to 8), use lower case
- %X Unsigned integer, hexadecimal format, use upper case
- %*n*X Unsigned integer, hexadecimal format, using *n* digits (*n* = 1 to 8), use upper case
- %f Floating-point value, up to 6 digits total
- %*n*f Floating-point value, up to *n* digits total (*n* = 1 to 31)
- %*n*.*m*f Floating-point value, up to *n* digits total (*n* = 1 to 31), up to *m* fractional digits (*m* < *n*)
- %s Text string, arbitrary length (null terminated)
- %*n*s Text string, up to *n* characters (shorter if null terminator encountered)
- %c Single character of specified byte value
- %*n*c *n* repetitions of single character of specified byte value
- %% "Percent" character

Leading zeros are not created for decimal-format integers. Leading zeros are created for hexadecimal-format integers (only) when the number of digits is specified. Leading and trailing zeros are not created for floating-point values, whether or not the number of digits is specified. If the number of digits specified for a floating-point is not sufficient to represent the value without an exponent, it will automatically be represented with an exponent (e.g. 2.345e8). The decimal point is only used for floating-point values if there is a non-zero fractional component. In floating-point values, any decimal point, minus sign, or exponent used does not count as a digit.

For each formatted variable sequence in the string, there must be an ***{expression}*** that specifies the numerical value that is to be converted to text. Each expression can be as simple as a constant or a variable, but can include mathematical operators and expressions as well. For a string variable, the expression value (rounded down to the next integer if necessary) represents the starting index in user shared memory of the string variable to be used.

Note that no responses to the command can be handled. If there is a response, it will be lost, but it will not cause an error.

In the local data structure for the coordinate system (if in a motion program) or the PLC program, the element **Ldata.SystemCmdStatus** is set to 0 on the successful execution of the Power PMAC script command. It is set to a negative number if there is an error in executing the command.

Users who wish to monitor the execution of the command should set this element to a positive number before issuing the command.

Also in the local data structure, the element **Ldata.SystemCmdCount** increments on the execution of the command, successful or unsuccessful. Note that it does nothing if there is an error in executing the command. The user should use the **sendallsystemcmds** command to ensure the command buffer is flushed before checking the count. The user can write to **Ldata.SystemCmdCount**; some users will set it to 0 before issuing a set of commands, then monitor it to see that the expected number of commands have been executed

If there is an error in processing a command, then **Ldata.SystemCmdStatus** will have a negative value until the next successful command is executed (or the user overwrites it with a non-negative value).

Examples

```
system"/opt/ppmac/tune/parabolicmove 1 2000 500 0 0 0";
// Execute provided "parabolic move" tuning C program

Ldata.SystemCmdStatus=1;              // Set to start
system"/var/ftp/usrflash/Project/C\ Language/Background\ Programs/cplc1.out";
// Execute user C application program
while (Ldata.SystemCmdStatus > 0) {}; // Wait until done
```

T{data}

Function: Tool Select Code (T-Code)

Syntax: **T*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) in the range 0.000 to 999.999 representing the code number and therefore the line jump label of the subroutine to implement that code

Power PMAC interprets a "T-code" as a **call*{Coord[x].Tprog}*.(*{data}**1000)** command, where ***{data}*** is the T-code "number". That is, this command causes a jump to the subprogram whose number is specified by the value of **Coord[*x*].Tprog**, at the line jump label whose number is 1000 times the code number. For example, with **Coord[*x*].Tprog** at its default value of 1002, **T03** is a call to line jump label **N3000:** of **subprog 1002**. Program execution will jump back to the calling program on the next **return** command.

This structure permits the implementation of customizable T-code routines for RS-274-compatible code, as from CAD/CAM programs and for CNC-style applications. Arguments can be passed to these subroutines by following the T-code in the calling program with one or more sets of ***{letter}{data}***. The values following each letter can be obtained by the subroutine using the **read** command. The default value of **Coord[*x*].Tprog** is 1002 for all coordinate systems, so by default all coordinate systems will share the same set of T-codes. However, if different values are assigned for different coordinate systems, separate code implementations can be written.

T-codes are typically used as "tool-select codes" in RS-274 programs to control automatic tool changers and the like. While the core set of T-codes in the standards use integer numbers in the range 0 to 99, this scheme permits fractional code numbers, a range up to 999.999, and the use of mathematical expressions for code numbers.

Subroutine and subprogram calls of all types (**call**, **callsub**, **gosub**, **G**, **M**, **T**, **D**) may be nested a total of 255 deep (256 including the top level). Indefinite recursion should not be used.

If the jump line label in the program specified by the T-code command does not exist, the jump is to the top (beginning) of the specified T-code subprogram. There is no error. This permits the user application to decide how to handle "non-existent" T-codes.

ta{data}

Function: Acceleration/deceleration time specification

Syntax: **ta*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the acceleration/deceleration time in milliseconds

The **ta** command specifies the commanded acceleration and deceleration time for linear and circle mode moves, in units of milliseconds with floating-point resolution. The specified value must be zero or positive. Zero acceleration time specification will not be rejected and will not create run-time errors, but will create step changes in the command velocity profile. (This can have legitimate uses in "external time base" applications, where the true acceleration control comes from the ramping of the time-base value.)

It is possible to set a separate final deceleration time for linear and circle moves with the **td** (deceleration time) command. However, in order to do this, the **td** command must come after the **ta** command, because the **ta** command sets both acceleration and deceleration times.

When the **ta** command is executed in a motion program, the data-structure elements **Coord[*x*].Ta** and **Coord[*x*].Td** for the coordinate system running the program are set to the value evaluated in ***{data}***. If no **ta** command is used in the motion program, the values in the data structure elements **Coord[*x*].Ta** and **Coord[*x*].Td** are used directly.

A **ta** command is a "no op" in a PLC program; if you wish to set a coordinate system acceleration time value directly from a PLC program, you should write directly to **Coord[*x*].Ta** and possibly **Coord[*x*].Td**.

If the rate of acceleration for any motor in the coordinate system exceeds the limit set by the data-structure element **Motor[*x*].InvAmax**, the acceleration time for that move will be extended just enough so that no motor limit is violated. This automatic limiting is operational if the coordinate system in not in segmentation mode (**Coord[*x*].SegMoveTime** = 0), or if the special lookahead function is enabled while in segmentation mode (**Coord[*x*].SegMoveTime** > 0).

If the specified S-curve acceleration time (**Coord[*x*].Ts**) is greater than half the acceleration or deceleration time, the overall acceleration or deceleration time used will be twice the specified S-curve time.

If the specified move time, either set directly with the **tm** command, or computed at the vector distance divided by the vector feedrate (**F**), is less than the acceleration time, the move time will be increased to match the acceleration time. This means that the acceleration time acts as the minimum permissible move time.

Examples

```
ta100;                          // Set accel & decel times of 100 msec
ta(P20);                        // Set accel & decel times of P20 msec
ta(45.3+sqrt(Q10));             // Set accel & decel times of 45.3+sqrt(Q10) msec
```

td*{data}*

Function: Deceleration time specification

Syntax: **td*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the deceleration time in milliseconds

The **td** command specifies the commanded deceleration time for linear and circle mode moves, in units of milliseconds with floating-point resolution. The specified value must be zero or positive. Zero deceleration time specification will not be rejected and will not create run-time errors, but will create step changes in the command velocity profile. (This can have legitimate uses in "external time base" applications, where the true deceleration control comes from the ramping of the time-base value.)

The **td** command permits the user to specify a final deceleration time for linear and circle mode moves or move sequences that is different from the initial acceleration and blending times. Note that a **ta** command sets both acceleration and deceleration times, so if a separate deceleration time is desired, the **td** command must follow the **ta** command.

When the **td** command is executed in a motion program, the data structure element **Coord[*x*].Td** for the coordinate system running the program is set to the value evaluated in ***{data}***. If no **td** command is used in the motion program, the value in the data structure elements **Coord[*x*].Td** is used directly.

A **td** command is a "no op" in a PLC program; if you wish to set a coordinate system deceleration time value directly from a PLC program, you should write directly to **Coord[*x*].Td**.

If the rate of acceleration for any motor in the coordinate system exceeds the magnitude limit set by the data structure element **Motor[*x*].InvDmax**, the deceleration time for that move will be extended just enough so that no motor limit is violated. This automatic limiting is operational if the coordinate system in not in segmentation mode (**Coord[*x*].SegMoveTime** = 0), or if the

special lookahead function is enabled while in segmentation mode (**Coord[*x*].SegMoveTime** > 0).

If the specified S-curve acceleration time (**Coord[*x*].Ts**) is greater than half the deceleration time, the overall deceleration time used will be twice the specified S-curve time.

If the specified move time, either set directly with the **tm** command, or computed at the vector distance divided by the vector feedrate (**F**), is less than the acceleration time, the move time will be increased to match the acceleration time. This means that the acceleration time acts as the minimum permissible move time.

Examples

```
td100;                          // Set decel time of 100 msec
td(P20);                        // Set decel time of P20 msec
td(45.3+sqrt(Q10));             // Set decel time of 45.3+sqrt(Q10) msec
```

tm*{data}*

Function: Move time specification

Syntax: **tm*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the move time in milliseconds

The **tm** command specifies the time to be taken by subsequent linear and circle mode moves, in units of milliseconds with floating-point resolution (before acceleration and deceleration times are added). The specified value must be zero or positive. It overrides any previous **tm** or **F** value, and is overridden by any subsequent **tm** or **F** command.

Execution of a **tm** command in a motion program causes Power PMAC to set **Coord[*x*].Tm** to the specified value. (The positive sign indicates that the coordinate system is in feedrate mode and not move-time mode.) A **tm** command is a "no op" in a PLC program; if you wish to set a coordinate system feedrate value directly from a PLC program, you should write directly to **Coord[*x*].Tm**.

If the specified move time is less than the acceleration time, the move time will be increased to match the acceleration time, resulting in axis velocities lower than those determined by dividing axis distances by the move time.

If the velocity requested for any motor in the coordinate system determined by its move distance divided by the move time exceeds the magnitude limit set by the data-structure element **Motor[*x*].MaxSpeed**, the move time for that move will be extended just enough so that no motor limit is violated. This automatic limiting is operational if the coordinate system in not in segmentation mode (**Coord[*x*].SegMoveTime** = 0), or if the special lookahead function is enabled while in segmentation mode (**Coord[*x*].SegMoveTime** > 0).

Examples

```
tm100;                          // Set move time of 100 msec
tm(P20);                        // Set move time of P20 msec
tm(45.3+sqrt(Q10));             // Set move time of 45.3+sqrt(Q20) msec
```

tread

Function: Report axis target positions of presently executing move

Syntax: **tread**

The **tread** command causes Power PMAC to calculate the axis target positions for the presently executing move in the coordinate system addressed by the program in **Ldata.Coord**. Target position buffering must be enabled by setting saved setup element **Coord[*x*].TPSize** greater than 0, and to a value sufficient to store positions for all of the moves between move calculation time and move execution time. The positions will be calculated only for those axes specified by saved setup element **Coord[*x*].TPCoords**.

The target positions will be copied into local D-variables for the program, where they can be used in subsequent calculations. The positions will be returned as programmed, with any axis matrix transformations in force at the time the move was calculated. The D-variable D*i* used for each axis can be found in the following table:

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' target positions have been reported. Each bit *i* of D32 corresponds to the number of the variable D*i* for which a variable has been reported. The value of D32 should be equivalent to the value of **Coord[*x*].TPCoords**. If the returned value of D32 is 0, no positions have been reported.

If cutter radius compensation is enabled for the move, the positions returned for the X, Y, and Z axes in D6, D7, and D8, respectively, include the offsets from compensation for those axes at the end of the move. If 2D compensation has added an arc move around an outside corner to blend into the next move, the reported compensated target positions for these axes are at the end of the arc if bit 0 (value 1) of **Coord[*x*].CCCtrl** is set to the default value of 0; they are at the beginning of the move if this bit is set to 1.

The buffered program **tread** command performs the same calculations as the on-line **t** command.

ts*{data}*

Function: S-curve time specification

Syntax: **ts*{data}***

where:

- ***{data}*** is a floating-point constant (no parentheses) or an expression (in parentheses) representing the S-curve acceleration/deceleration time in milliseconds

The **ts** command specifies the time at the beginning and end of both the commanded acceleration and deceleration for linear and circle mode moves that is spent in an "S-curve" profile, in units of milliseconds with floating-point resolution. The specified value must be zero or positive. During the S-curve time, the rate of acceleration changes linearly from zero to the maximum acceleration magnitude (at the beginning of the acceleration or deceleration), or linearly from the maximum acceleration magnitude to zero (at the end).

While it is possible to set a separate overall final deceleration time for linear and circle moves with the **td** (deceleration time) command, the specified S-curve time for both is the same.

When the **ts** command is executed in a motion program, the data structure element **Coord[*x*].Ts** for the coordinate system running the program are set to the value evaluated in ***{data}***. If no **ts** command is used in the motion program, the value in the data structure element **Coord[*x*].Ts** is used directly.

A **ts** command is a "no op" in a PLC program; if you wish to set a coordinate system S-curve time value directly from a PLC program, you should write directly to **Coord[*x*].Ts**.

If the magnitude of the rate of acceleration change (called "jerk") for any motor in the coordinate system exceeds the limit set by the data-structure element **Motor[*x*].InvJmax**, the S-curve time for that move acceleration will be extended just enough so that no motor limit is violated. This automatic limiting is only operational for linear mode moves when the coordinate system in not in segmentation mode (**Coord[*x*].SegMoveTime** = 0).

If the specified S-curve acceleration time (**Coord[*x*].Ts**) is greater than half the acceleration or deceleration time, the overall acceleration or deceleration time used will be twice the specified S-curve time.

Examples

```
ts100;                          // Set S-curve time of 100 msec
ts(P20);                        // Set S-curve time of P20 msec
ts(45.3+sqrt(Q10));             // Set S-curve time of 45.3+sqrt(Q10) msec
```

tsel*{data}*

Function: Select active transformation matrix for coordinate system

Syntax: **tsel*{data}***

where:

- ***{data}*** specifies the number of the transformation matrix selected

The **tsel** command selects the specified axis transformation matrix as the active transformation matrix for the addressed coordinate system. (For a motion program, this is the coordinate system that is running the program.) The valid numbers for axis transformation matrices are 0 to 255. If the value specified by ***{data}*** does not evaluate to an integer, the value is rounded *down* to the next integer.

The number of the presently selected matrix for the coordinate system can be seen in the data structure element **Coord[*x*].Tsel**.

A value of -1 deselects all transformation matrices, saving calculation time in the coordinate system. This is the power-up/reset default for all coordinate systems.

txyz

Function: Specify tool-orientation vector for 3D cutter radius compensation

Syntax: **txyz*{vector}{data}[{vector}{data}…]***

where:

- ***{vector}*** is a character (I, J, or K) specifying a vector component parallel to the X, Y, or Z-axis, respectively

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the vector component

The **txyz** command specifies the tool-orientation vector used for 3D cutter radius compensation by setting the components of this vector along the X, Y, and Z axes. The absolute magnitude of these components, set by the values following the I, J, and K characters, respectively, do not matter; the relative magnitudes of these components determine the orientation of the vector. The direction sense of the tool-orientation vector (base-to-tip or tip-to-base) does not matter.

The tool-orientation vector defined with this command will be used to determine the compensation offset of the end of the move commanded on the same program line, and any subsequent moves until a new surface-normal vector is specified. However, it is standard practice that a surface-normal vector be specified separately for every commanded move in 3D compensation.

If a component of the tool-orientation vector is not explicitly declared in a **txyz** command, that component will automatically be set to 0.0.

The **txyz** command does not cause the tool to be oriented in this direction; that must be done with axis move command components. The **txyz** command simply tells the 3D compensation algorithm what the orientation of the tool is in a standard format.

txyzscale*{data}*

Function: Set transformation matrix feedrate and compensation radius rescaling

Syntax: **txyzscale*{data}***

where:

- ***{data}*** is a constant (no parentheses) or an expression (in parentheses) representing the rescaling value

The **txyzscale*{data}*** command directly specifies the value to be written to the **Coord[*x*].TxyzScale** status element, which controls the rescaling of the vector feedrate and cutter compensation radius values for the X, Y, and Z axes in the coordinate system. This permits the feedrate and compensation radius values to be held constant when the selected transformation matrix has rescaled the dimensions of the X, Y, and Z axes.

If saved setup element **Coord[*x*].AutoTxyzScale** is set to 1, the **TxyzScale** value is computed automatically from the values of the transformation matrix when a **tsel*{data}*** command is executed. In this case, there is usually no need to use the **txyzscale*{data}*** command.

Manual setting of this rescaling value with the **txyzscale*{data}*** command can provide more flexibility than the automatic setting.

vread

Function: Report axis present actual velocities

Syntax: **vread**

The **vread** command causes Power PMAC to calculate the present axis actual (filtered) velocities in the coordinate system addressed by the program in **Ldata.Coord**. The actual velocities will be calculated for all active axes in the coordinate system.

The present actual velocity for the axis is calculated from the corresponding motor position history(ies), processed through the coordinate system definition (either by inverting axis definition statements or using the forward kinematic subroutine), and any matrix transformations in force. If a forward kinematic subroutine is used, it must be able to perform a “double pass” execution, so present and past positions can be used to calculate velocities.

The axis actual velocity values will be copied into local D-variables for the program, where they can be used in subsequent calculations. The velocities will be returned in the present axis units, with any axis matrix transformations in force. The D-variable D*i* used for each axis can be found in the following table:

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' actual velocities 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.

When executed from a background PLC program, there is a slight possibility of an error in one of the reported axis values due to an interrupt occurring between the first and last read operations required to compute a value and updating one of the source registers.

The buffered program **vread** command performs the same calculations as the on-line **v** command.

while(*{condition}*)

Function: Conditional loop

Syntax: **while(*{condition}*)**

where:

- ***{condition}*** is a simple or compound condition, or a mathematical expression without a conditional comparator

Full Structure Syntax:

```
while({condition}){command}[{command}…]

while({condition}) {
   {command}
   [{command}…]
}
```

The **while** command permits conditional looping in motion and PLC programs. When the condition inside parentheses evaluates as true, or the expression evaluates as any non-zero value, the program command(s) inside the loop will be executed.

The **while** command can be used before the looping action, in which case the condition is evaluated before the loop is executed even once. If the condition evaluates as false, none of the commands in the loop are executed even a single time. It can also be used at the end of a loop started by the **do** command. In this case, the commands in the loop are always executed at least once.

When the **while** command is used at the beginning of a loop, if commands follow immediately on the same program line, only these commands on the same line are executed in the event of a true condition.

If the left "curly bracket" (**{**) is the next character in the program after the right parenthesis that closes the condition, whether on the same program line or the next program line, this bracket denotes the start of the commands to be executed on a true condition. This command execution will continue until a right curly bracket (**}**) is encountered.

When the **while** command is used at the end of a loop that is started with a **do** command, any subsequent commands on the same program line are not part of the loop, and will not be executed until the condition evaluates as false.

It is possible to nest **while** loops within other conditional structures (**if**, **while**, **switch**) and to nest other conditional structures within **while** loops. The nesting of conditional structures can go up to 32 levels deep in total.

Examples

```
while (M1 == 0) dwell 10;      // Loop in 10 msec intervals as long as M1 is 0

while (P1 > 0) {}              // Idle as long as P1 is greater than 0

while (Q100) {                 // Loop to end bracket as long as Q100 is not 0
  X10;
  X0;
}                              // End of loop

P50 = 0;                       // Initialize loop counter
do {                           // Start of loop
  YY200;
  dwell 50;
  YY10;
  P50++
}                              // End of loop
while (P50 < 100);             // Conditional statement for looping 100 times
```

POWER PMAC PROJECT ENHANCED SCRIPT LANGUAGE COMMANDS

The project manager in the Power PMAC's Integrated Development Environment (IDE) software for the PC permits significant enhancements to the power and flexibility of the underlying Power PMAC Script language. This section describes these enhancements in detail. Note that these must be used through the "download" function of the IDE's project manager, which will process these into the underlying Power PMAC Script. If these enhanced commands are sent directly to the Power PMAC, they will be rejected with an error.

Script Pre-Processing Directives

The pre-processing directives in this section permit the user to specify control flags and use them to conditionally control downloading commands; also to create text substitutions for constants, variables, and expressions.

#define *{identifier}*

Function: Create a pre-processor flag

Scope: Global to Power PMAC project

Syntax: **#define *{identifier}***

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the pre-processor flag name

This directive allows the user to create a flag for conditional pre-processor directives in the IDE project manager, particularly the **#ifdef** and the **#ifndef** conditional directives. The effect of this directive can be undone by the **#undef** directive.

An identifier defined in this way, without any value assigned to it, is automatically give a value of 0. In this case, a **#ifdef *{identifier}*** condition will evaluate as true, but a **#if *{identifier}*** condition will evaluate as false.

Example

```
#define TestMode
#ifdef TestMode
  &1%25
#endif
```

#define *{identifier} {token string}*

Function: Create a text substitution macro

Scope: Global to Power PMAC project

Syntax: **#define *{identifier} {token string}***

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the text to be replaced in the download

- ***{token string}*** is a text string that is the replacement for the identifier. It can be terminated by a semicolon, carriage return, or comment start ('//' or '/*')

This directive allows the user to specify a text substitution in the downloaded project using the IDE project manager. Whenever the project manager comes across the text specified in ***{identifier}***, it substitutes the text in ***{token string}***. The identifier is case-sensitive. This substitution permits the use of names meaningful to the application in commands and programs.

Examples

```
#define True 1;
#define _Length2 Q100+10
#define SixFactorial (6*5*4*3*2)
#define Cycle_Counter P10
#define Clear_Cycle_Counter P10=0
```

#define {identifier} {token string} \

Function: Create a multi-line text substitution macro

Scope: Global to Power PMAC project

Syntax: **#define *{identifier} {token string}* **
{token string} [\]…

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the text to be replaced in the download

- ***{token string}*** is a text string that is the replacement for the identifier. It can be terminated by a semicolon, carriage return, or comment start ('//' or '/*')

This directive allows the user to specify a multi-line text substitution in the downloaded project using the IDE project manager. Whenever the project manager comes across the text specified in ***{identifier}***, it substitutes the text in the set of ***{token string}*** representations. The identifier is case-sensitive.

If there is a "****" backslash character after an individual token string, the text substitution will continue with the token string on the next line. (No line separator is included in the substituted text.) The substitution ends with the token string that is not followed by a backslash character.

Multi-line text substitution macros can only be invoked from within a downloaded IDE project. Unlike single-line macros, they cannot be invoked from command lines such as the IDE terminal window or other `gpascii -2` communications threads.

Examples

```
#define ClearCounters P100=0\
P101=0\
P102=0\
P103=0
```

#define *{identifier}* (*{argument list}*) *{token string}* *[\]*

Function: Create a function text substitution macro

Scope: Global to Power PMAC project

Syntax: **#define *{identifier}* (*{argument list}*) *{token string}***

#define *{identifier}*(*{argument list}*) *{token string}* \ *{token string}* *[\]*…

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the function name to be replaced in the download
- ***{argument list}*** is one or more text strings, separated by commas if more than one, representing the arguments to be used in the function
- ***{token string}*** is a text string that is the replacement for the identifier. It can be terminated by a semicolon, carriage return, or comment start ('//' or '/*')

This directive allows the user to specify a text substitution for a function with arguments in the downloaded project using the IDE project manager. Whenever the project manager comes across the text specified in ***{identifier}***, it substitutes the text in the set of ***{token string}*** representations with the specified arguments. The identifier is case-sensitive.

If there is a "****" backslash character after an individual token string, the text substitution will continue with the token string on the next line. (No line separator is included in the substituted text.) The substitution ends with the token string that is not followed by a backslash character.

Function text substitution macros, single-line or multi-line, can only be invoked from within a downloaded IDE project. Unlike single-line macros, they cannot be invoked from command lines such as the IDE terminal window or other `gpascii -2` communications threads.

Examples

```
#define SindSquared(Theta) sind(Theta)*sind(Theta)

#define SindAngleSum(A,B) sind(A)*cosd(B) + cosd(A)*sind(B)

#define _Pi 3.14159265
#define sinc(x) sin(_Pi*x)/(_Pi*x)

#define Factorial(Pnum,X) P(Pnum)=X;\
while (X > 1) {\
  X--;\
  P(Pnum)=P(Pnum)*X;\
}
```

#elif

Function: Start of compound false conditional directive branch

Scope: Global to Power PMAC project

Syntax: **#elif *{identifier}* [{comparator} {constant}]**

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the constant or parameter to be evaluated
- ***{comparator}*** is one of the following conditional comparators:==, !=, <, >, <=, >=
- ***{constant}*** is an integer constant in the signed 32-bit range

This directive allows the user to create a nested conditional branch in the false branch of a preceding condition in the execution of project download commands. It can be used with the following conditional directives: **#if**, **#ifdef**, **#ifndef**, and **#elif**.

If the condition on the line evaluates as true, the commands immediately following will execute, down to the next **#endif**, **#else**, or **#elif** directive.

When the directive is of the simple form **#elif *{identifier}***, the condition will evaluate as true only if the ***{identifier}*** has been defined and assigned a non-zero constant value.

When the directive is of the form **#elif *{identifier} {comparator} {constant}***, the condition will evaluate as true if the value of the entity of ***{identifier}*** has the relationship to the ***{constant}*** value prescribed by ***{comparator}***.

Example

```
#if SafetyLevel == 1
  Motor[5].FatalFeLimit = 2000
#elif SafetyLevel == 2
  Motor[5].FatalFeLimit = 1000
#else
```

```
  Motor[5].FatalFeLimit = 500
#endif
```

#else

Function: Start of false conditional directive branch

Scope: Global to Power PMAC project

Syntax: **#else**

This directive allows the user to create a conditional branch that will execute when the condition for a previous conditional directive evaluates as false. It can be used with the following conditional directives: **#if**, **#ifdef**, **#ifndef**, and **#elif**.

If the preceding conditional directive evaluates as false, execution skips to the command following this **#else** directive and continues to the next **#endif** directive. There can be no other conditional directives between **#else** and **#endif**.

Example

```
#define TestMode
#ifdef TestMode
  &1%25
#else
  &1%100
#endif
```

#endif

Function: End of false conditional directive branch

Scope: Global to Power PMAC project

Syntax: **#endif**

This directive allows the user to terminate a conditional branch that was started with one of the following conditional directives: **#if**, **#ifdef**, **#ifndef**, **#else**, and **#elif**.

Examples

```
#if SafetyLevel >= 3
  Motor[5].FatalFeLimit = 100
#endif

#define TestMode
#ifdef TestMode
  &1%25
#else
  &1%100
#endif
```

#if

Function: Start of conditional directive branch based on value

Scope: Global to Power PMAC project

Syntax: **#if *{identifier}* [*{comparator}* *{constant}*]**

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the constant or parameter to be evaluated
- ***{comparator}*** is one of the following conditional comparators: ==, !=, <, >, <=, >=
- ***{constant}*** is an integer constant in the signed 32-bit range

This directive allows the user to create a conditional branch in the execution of project download commands. If the condition on the line evaluates as true, the commands immediately following will execute, down to the next **#endif**, **#else**, or **#elif** directive.

When the directive is of the simple form **#if *{identifier}***, the condition will evaluate as true only if the ***{identifier}*** has been defined and assigned a non-zero constant value.

When the directive is of the form **#if *{identifier}* *{comparator}* *{constant}***, the condition will evaluate as true if the value of the entity of ***{identifier}*** has the relationship to the ***{constant}*** value prescribed by ***{comparator}***.

Examples

```
#if SafetyLevel >= 3
  Motor[5].FatalFeLimit = 100
#endif

#if Expanded_Machine
  Sys.MaxMotors = 9
#else
  Sys.MaxMotors = 5
#endif
```

#ifdef

Function: Start of conditional directive branch based on existence

Scope: Global to Power PMAC project

Syntax: **#ifdef *{identifier}***

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the constant or parameter to be evaluated

This directive allows the user to create a conditional branch in the execution of project download commands. If the condition on the line evaluates as true, the commands immediately following will execute, down to the next **#endif**, **#else**, or **#elif** directive.

The condition will evaluate as true if the entity of ***{identifier}*** has been defined with a **#define** directive, with or without a value assigned to it.

Example

```
#define _AbsMode
#ifdef _AbsMode
  Motor[1].pAbsPos = Gate3[0].Chan[0].SerialEncDataA.a
  …
#endif
```

#ifndef

Function: Start of conditional directive branch based on non-existence

Scope: Global to Power PMAC project

Syntax: **#ifndef *{identifier}***

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the constant or parameter to be evaluated

This directive allows the user to create a conditional branch in the execution of project download commands. If the condition on the line evaluates as true, the commands immediately following will execute, down to the next **#endif**, **#else**, or **#elif** directive.

The condition will evaluate as true if the entity of ***{identifier}*** has not been defined with a **#define** directive, or if such a definition has been undone with a **#undef** directive.

Example

```
#ifndef _AbsMode
  Motor[1].pAbsPos = 0
  …
#endif
```

#undef *{identifier}*

Function: Undo a pre-processor flag definition or macro substitution definition

Scope: Global to Power PMAC project

Syntax: **#undef *{identifier} [{identifier}…]***

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the pre-processor flag name or macro substitution name

This directive allows the user to undo the definition of a pre-processor flag definition of a macro substitution definition. Once a macro substitution definition is undone, the pre-processor will no longer perform the text substitution during the project loading.

Examples

```
#define _DebugMode
…
#undef _DebugMode

#define Indicator2Bank M100
…
#undef Indicator2Bank
```

Script Variable Declarations

The declarations in this section provide the easiest way of giving meaningful names to user variables of various types.

csglobal

Function: Auto-assign user name to coordinate-system specific global ("Q") variable

Scope: Declarations global to project; initialization values coordinate-system specific

Syntax:
```
csglobal {identifier} [= {expression}]
         [, {identifier} [= {expression}]]…;

csglobal {identifier} ({array_size})
         [, {identifier} ({array_size})]…;
```

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the user variable name
- ***{expression}*** is a coherent combination of constants, variables, functions and operators that evaluates to a numerical value. It can be terminated by a semicolon, carriage return, or comment start ('//' or '/*').
- ***{array_size}*** is a positive integer constant or a previously defined identifier for a positive integer constant

The **csglobal** declaration causes the user-defined identifier name to be automatically assigned to a set of Power PMAC coordinate-system global ("Q") variables – with the same Q-variable number for each coordinate system – by the IDE project manager. The identifier is case-sensitive.

There are 8,192 Q-variables (**Q0** - **Q8191**) for each coordinate system in Power PMAC. The project manager starts these assignments at the Q-variable whose number is specified by the Q-variable starting point defined in the IDE Project/Properties window. This is **Q1024** by default. Typically, lower-numbered variables are either used directly, or given user names with a `#define` directive (particularly when porting applications from the older Turbo PMAC).

It is possible to declare a user array of Q-variables with this declaration if the user name is immediately followed by the array size (either a positive integer constant or a previously defined identifier for a positive integer constant) in parentheses. In this case, the project manager will allocated the specified number of consecutively numbered Q-variables to the array.

For the declaration of a single variable (but not an array), the user can specify a value that will automatically be assigned to the variable on project download, power-up, or reset. The value is specified by a mathematical expression (which can be as simple as a single numerical constant). Note, however, that this value is only assigned to the Q-variable of the addressed coordinate system.

A **csglobal** declaration must be outside of any Power PMAC script program or subprogram. Users are encouraged to put all of these declarations in a single file such as the standard `global definitions.pmh` project file.

Examples

```
csglobal CycleCount;
&1
csglobal MaxErrors = 5;
csglobal Xscale, Yscale, Zscale;

#define True 1;
#define False 0;
&2
csglobal SysInitComplete = False;
csglobal OperationMode = True;

csglobal XyzOffset(3);
#define ListLength 500;
csglobal BatchRMS(ListLength);
```

global

Function: Auto-assign user name to global ("P") variable

Scope: Global to project

Syntax:

```
global {identifier} [= {expression}]
        [, {identifier} [= {expression}]]…;

global {identifier} ({array_size})
        [, {identifier} ({array_size})]…;
```

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the user variable name
- ***{expression}*** is a coherent combination of constants, variables, functions and operators that evaluates to a numerical value. It can be terminated by a semicolon, carriage return, or comment start ('//' or '/*').
- ***{array_size}*** is a positive integer constant or a previously defined identifier for a positive integer constant

The **global** declaration causes the user-defined identifier name to be automatically assigned to a Power PMAC global ("P") variable by the IDE project manager. The identifier is case-sensitive.

There are 65,536 P-variables (**P0** - **P65535**) in Power PMAC. The project manager starts these assignments at the P-variable whose number is specified by the P-variable starting point defined in the IDE Project/Properties window. This is **P8192** by default. Typically, lower-numbered

variables are either used directly, or given user names with a `#define` directive (particularly when porting applications from the older Turbo PMAC).

It is possible to declare a user array of P-variables with this declaration if the user name is immediately followed by the array size (either a positive integer constant or a previously defined identifier for a positive integer constant) in parentheses. In this case, the project manager will allocated the specified number of consecutively numbered P-variables to the array.

For the declaration of a single variable (but not an array), the user can specify a value that will automatically be assigned to the variable on project download, power-up, or reset. The value is specified by a mathematical expression (which can be as simple as a single numerical constant). This provides an easy method for initializing the variable value for an application.

A **`global`** declaration must be outside of any Power PMAC script program or subprogram. Users are encouraged to put all of these declarations in a single file such as the standard `global definitions.pmh` project file.

Examples

```
global CycleCount;
global MaxErrors = 5;
global Xscale, Yscale, Zscale;

#define True 1;
#define False 0;
global SysInitComplete = False;
global OperationMode = True;

global XyzOffset(3);
#define ListLength 500;
global BatchRMS(ListLength);
```

local

Function: Auto-assign user name to local (“L”) variable

Scope: Local to program or subprogram

Syntax:

```
local {identifier} [= {expression}]
        [, {identifier} [= {expression}]]…;

local {identifier} ({array_size})
        [, {identifier} ({array_size})]…;
```

where:

- ***{identifier}*** is a text string beginning with a letter or the ‘_’ underscore character and continuing with alphanumeric characters and ‘_’ (no spaces), representing the user variable name

- ***{expression}*** is a coherent combination of constants, variables, functions and operators that evaluates to a numerical value. It can be terminated by a semicolon, carriage return, or comment start (‘//’ or ‘/*’).

- ***{array_size}*** is a positive integer constant or a previously defined identifier for a positive integer constant

The **local** declaration causes the user-defined identifier name to be automatically assigned to a Power PMAC local (“L”) variable by the IDE project manager. The identifier is case-sensitive.

There is a stack of 8,192 local variables for each execution thread (coordinate system, PLC program, communications thread) in Power PMAC. The default stack offset for a program or subprogram in calling subroutines or subprograms is 256, but this can be changed in the program declaration. The IDE project manager automatically sets this stack offset to the minimum value that accommodates the passed and declared local variables found.

The project manager starts these assignments at the next L-variable after those used (if any) to passed arguments to and from the subprogram. For example, if there were three arguments passed to the subprogram and one return argument, those arguments would be assigned to **L0** through **L3** for the subprogram, and local variables declared inside the subprogram would start with **L4**.

It is possible to declare a user array of L-variables with this declaration if the user name is immediately followed by the array size (either a positive integer constant or a previously defined identifier for a positive integer constant) in parentheses. In this case, the project manager will allocated the specified number of consecutively numbered L-variables to the array.

For the declaration of a single variable (but not an array), the user can specify a value that will automatically be assigned to the variable each time the program or subprogram is entered at the top. The value is specified by a mathematical expression (which can be as simple as a single numerical constant). This provides an easy method for initializing the variable value when executing the routine.

A **local** declaration must be inside the Power PMAC script program or subprogram where it is used.

Examples

```
open prog 100
local LoopCount;
local MaxErrors = 5;
local Xscale, Yscale, Zscale;

#define True 1;
#define False 0;
open subprog StepAndRepeat
local StepComplete = False;
local OperationMode = True;

#define NumLoops 500;
open plc PickAndPlace
local XyzOffset(3);
local LengthError(NumLoops);
```

ptr

Function: Auto-assign user name to pointer (“M”) variable

Scope: Global to project

Syntax:
```
ptr {identifier}->{definition} [= {expression}]
   [,{identifier}->{definition} [= {expression}]]…;

ptr {identifier} ({array_size})->{definition}
   [, {identifier} ({array_size})->{definition}]…;
```

where:

- ***{identifier}*** is a text string beginning with a letter or the ‘_’ underscore character and continuing with alphanumeric characters and ‘_’ (no spaces), representing the user variable name
- ***{definition}*** is a valid Power PMAC M-variable definition of any type
- ***{expression}*** is a coherent combination of constants, variables, functions and operators that evaluates to a numerical value. It can be terminated by a semicolon, carriage return, or comment start (‘//’ or ‘/*’).
- ***{array_size}*** is a positive integer constant or a previously defined identifier for a positive integer constant

The **ptr** declaration causes the user-defined identifier name to be automatically assigned to a Power PMAC pointer (“M”) variable by the IDE project manager. The identifier is case-sensitive.

There are 16,384 M-variables (**M0** – **M16383**) in Power PMAC. The project manager starts these assignments at the M-variable whose number is specified by the M-variable starting point defined in the IDE Project/Properties window. This is **M8192** by default. Typically, lower-numbered variables are either used directly, or given user names with a `#define` directive (particularly when porting applications from the older Turbo PMAC).

It is possible to declare a user array of M-variables with this declaration if the user name is immediately followed by the array size (either a positive integer constant or a previously defined identifier for a positive integer constant) in parentheses. In this case, the project manager will allocated the specified number of consecutively numbered M-variables to the array. The definitions must be of the “address” type, and will be made to consecutively numbered registers.

For the declaration of a single variable (but not an array), the user can specify a value that will automatically be assigned to the variable on project download, power-up, or reset. The value is specified by a mathematical expression (which can be as simple as a single numerical constant). This provides an easy method for initializing the variable value for an application. (Of course, if the variable points to a read-only register, or a register that is written to by automatic Power PMAC algorithms, this assignment my not be effective.)

A **ptr** declaration must be outside of any Power PMAC script program or subprogram. Users are encouraged to put all of these declarations in a single file such as the standard `global definitions.pmh` project file.

Examples

```
ptr ContactSwitch->Acc68E[0].DataReg[1].2;
ptr LaserOn->Acc68E[1].DataReg[3].7 = 0;
ptr FirstIoCardReg(8)->u.io:$A00000.8.8+
```

Program and Subroutine Declarations

The Power PMAC program and subroutine declarations in this section provide a method for giving these routines meaningful names in the application.

open forward

Function: Open a forward-kinematic routine buffer

Scope: Coordinate-system specific

Syntax: **open forward [({*constant*})]**

where:

- the optional ***{constant}*** is an integer in the range 0 .. 127 representing the number of the coordinate system this routine is to be used for

The **open forward** command opens the buffer for the forward-kinematic subroutine for a coordinate system. If the coordinate system number is specified in parentheses at the end of the command, it will open the buffer for that coordinate system. If there is no number in parentheses at the end of the command, it will open the buffer for the modally addressed coordinate system.

The pre-processor automatically provides the following `#define` substitutions so the kinematic routine can use meaningful names:

```
#define KinPosMotorx          Lx              // Motor # x = 0 .. 255
#define KinPosMotor(x)        L(x)            // Motor # x = 0 .. 255
#define KinVelMotorx          Rx              // Motor # x = 0 .. 255
#define KinVelMotor(x)        R(x)            // Motor # x = 0 .. 255
#define KinPosAxisα           Cy              // Axis index y = 0 .. 31
#define KinPosAxis(y)         C(y)            // Axis index y = 0 .. 31
#define KinVelAxisα           C{y+32}         // Axis index y = 0 .. 31
#define KinVelAxis(y)         C(y+32)         // Axis index y = 0 .. 31
#define KinEnaAxisα           pow(2,y)        // Axis index y = 0 .. 31
#define KinAxisIndexα         y               // Axis index y = 0 .. 31
#define KinVelEna             D0              // Velocity calcs needed
#define KinAxisUsed           D0              // Bits for axes used
```

The correspondence between axis letter(s) ***α*** and corresponding variable number ***y*** is shown in the following table:

Axis *α*	***y***	**Axis *α***	***y***	**Axis *α***	***y***	**Axis *α***	***y***	**Axis *α***	***y***	**Axis *α***	***y***	**Axis *α***	***y***	**Axis *α***	***y***
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

The pre-processor will automatically specify the size of the local-variable stack offset and the number of line jump labels based on the content of the routine.

Example

This simple example provides the forward-kinematic routine for C.S. 2 with a setup equivalent to `#3->1000X` and `#4->1000Y`.

```
open forward (2)
if (KinVelEna) callsub 100;                  // For double-pass calculations
KinAxisUsed = KinEnaAxisX + KinEnaAxisY;     // X and Y computed
N100:
KinPosAxisX = KinPosMotor3 / 1000;
KinPosAxisY = KinPosMotor4 / 1000;
close
```

open inverse

Function: Open an inverse-kinematic routine buffer

Scope: Coordinate-system specific

Syntax: **`open inverse [({constant})]`**

where:

- the optional ***{constant}*** is an integer in the range 0 .. 127 representing the number of the coordinate system this routine is to be used for

The **`open inverse`** command opens the buffer for the inverse-kinematic subroutine for a coordinate system. If the coordinate system number is specified in parentheses at the end of the command, it will open the buffer for that coordinate system. If there is no number in parentheses at the end of the command, it will open the buffer for the modally addressed coordinate system.

The pre-processor automatically provides the `#define` substitutions shown above in the **`open forward`** description so the kinematic routine can use meaningful names.

The pre-processor will automatically specify the size of the local-variable stack offset and the number of line jump labels based on the content of the routine.

Example

This simple example provides the inverse-kinematic routine for C.S. 2 with a setup equivalent to `#3->1000X` and `#4->1000Y`.

```
open inverse (2)
KinPosMotor3 = 1000 * KinPosAxisX;
KinPosMotor4 = 1000 * KinPosAxisY;
if (KinVelEna) {                      // Velocity calcs needed (non-seg PVT)?
  KinVelMotor3 = 1000 * KinVelAxisX;
  KinVelMotor4 = 1000 * KinVelAxisY;
}
close
```

open plc *{identifier}*

Function: Open PLC program buffer for entry

Scope: Global

Syntax: **open plc *{identifier}***

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the user program name. It could also be an integer constant in the range 0 .. 31 or an identifier already assigned to one of these constants.

The **open plc** command opens the program buffer for one of the 32 Script PLC programs in Power PMAC. If the identifier is a constant in the range 0 .. 31 or a text string already assigned to one of these constant values, the buffer to be opened is directly specified by the value. (If this buffer has already been used by an automatically assigned identifier, a download error will be noted.)

If the identifier is a text string that has not previously been assigned to a constant value, the lowest-numbered PLC program buffer in the range 1 .. 31 not already used will be selected (unless the identifier is "RtPlc", in which case the PLC 0 program buffer will be selected). The identifier is case-sensitive.

The pre-processor will automatically specify the size of the local-variable stack offset and the number of line jump labels based on the content of the program.

The IDE Editor provides code snippet support for **plc**. From the intellisense list at **plc**, press the TAB key. If there is more than one parameter input, the TAB key moves you between them, and the ENTER key will exit the snippet so that you can enter other commands. Hovering the mouse over a parameter will give you information about that input; hovering over the **plc** command will give you a description of the command and the syntax of the arguments.

Example

```
open plc Coolant_Control      // Auto-assigned to next available buffer

enable plc Coolant_Control    // Command to start execution
```

open prog *{identifier}*

Function: Open motion program buffer for entry

Scope: Global

Syntax: **open prog *{identifier}***

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the user program name. It could also be an integer constant in the range 0 .. 2^{32}-1 or an identifier already assigned to one of these constants.

The **open prog** command opens the program buffer for one of the 1024 Script motion programs in Power PMAC. If the identifier is a constant in the range 0 .. 2^{32}-1 or a text string already assigned to one of these constant values, the buffer to be opened is directly specified by the value. (If this buffer has already been used by an automatically assigned identifier, a download error will be noted.)

If the identifier is a text string that has not previously been assigned to a constant value, the lowest-numbered motion program buffer in the range 100,000 and up not already used will be selected. The identifier is case-sensitive.

The pre-processor will automatically specify the size of the local-variable stack offset and the number of line jump labels based on the content of the program.

The IDE Editor provides code snippet support for **prog**. From the intellisense list at **prog**, press the TAB key. If there is more than one parameter input, the TAB key moves you between them, and the ENTER key will exit the snippet so that you can enter other commands. Hovering the mouse over a parameter will give you information about that input; hovering over the **prog** command will give you a description of the command and the syntax of the arguments.

Example

```
open prog CalibrationSequence          // Auto-assigned to next available buffer

&1 start CalibrationSequence           // Command to start execution
```

open subprog *{identifier}*

Function: Open subprogram buffer for entry

Scope: Global

Syntax: **open subprog *{identifier} [({call_list})]***

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the user subprogram name. It could also be an integer constant in the range 0 .. 2^{32}-1 or an identifier already assigned to one of these constants.

- the optional ***{call_list}*** is a set of identifiers for arguments to be passed to and from the subprogram

The **open subprog** command opens the program buffer for one of the 1024 Script subprograms in Power PMAC. If the identifier is a constant in the range 0 .. 2^{32}-1 or a text string

already assigned to one of these constant values, the buffer to be opened is directly specified by the value. (If this buffer has already been used by an automatically assigned identifier, a download error will be noted.) The subprogram is executed using a **`call`** command from a higher-level program

If the identifier is a text string that has not previously been assigned to a constant value, the lowest-numbered subprogram buffer in the range 100,000 and up not already used will be selected. The identifier is case-sensitive.

The optional "call list" allows for arguments to be passed to and from the subprogram on the local variable stack. If no stack arguments are to be passed, the list can be eliminated, or the **`(void)`** empty list can be used. Up to 32 identifiers can be used in the call list.

If arguments are to be passed to and/or from the subprogram, the call list will specify those arguments. Arguments passed to the subprogram *are not* preceded by the '&' character; arguments passed back to the calling program *are* preceded by the '&' character.

It is possible for an item in the call list to be an array, specified with the syntax ***`{identifier}({array_size})`***, with the array size being an integer constant in the range 1 to 256 or an identifier already assigned to an integer in this range. A maximum of 256 total individual values can be passed in the entire list.

The pre-processor will automatically specify the size of the local-variable stack offset and the number of line jump labels based on the content of the program.

The IDE Editor provides code snippet support for **`subprog`**. From the intellisense list at **`subprog`**, press the TAB key. If there is more than one parameter input, the TAB key moves you between them, and the ENTER key will exit the snippet so that you can enter other commands. Hovering the mouse over a parameter will give you information about that input; hovering over the **`subprog`** command will give you a description of the command and the syntax of the arguments.

Examples

```
open subprog Pythag (Adjacent, Opposite, &Hypotenuse)
Hypotenuse = sqrt(Adjacent*Adjacent + Opposite*Opposite);
return;
close

open subprog SquareArray (InputArray(20), &ReturnArray(20))
local Index = 0;
do {
  ReturnArray(Index) = InputArray(Index) * InputArray(Index);
  Index++;
} while (Index < 20);
return;
close
```

sub: *{identifier}*

Function: Declare a subroutine within a program or subprogram

Scope: Local to the Script program

Syntax: **sub:** ***{identifier}*** **[(*{sublabel_call_list}*)]**

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the user subroutine name.
- the optional ***{sublabel_call_list}*** is a set of identifiers for arguments to be passed to and from the subroutine

The **sub** declaration starts a subroutine within a Script program or subprogram (**prog**, **plc**, **subprog**, **forward**, **inverse**). The identifier will automatically be assigned to the lowest numeric jump label of 10,000 or above not already assigned. The identifier is case-sensitive. The subroutine is executed using a **callsub** command from within the same program.

The optional "call list" allows for arguments to be passed to and from the subprogram on the local variable stack. If no stack arguments are to be passed, the list can be eliminated, or the **(void)** empty list can be used. Up to 32 identifiers can be used in the call list.

If arguments are to be passed to and/or from the subprogram, the call list will specify those arguments. Arguments passed to the subprogram *are not* preceded by the '&' character; arguments passed back to the calling program *are* preceded by the '&' character.

It is possible for an item in the call list to be an array, specified with the syntax ***{identifier}*****(*{array_size}*)**, with the array size being an integer constant in the range 1 to 256 or an identifier already assigned to an integer in this range. A maximum of 256 total individual values can be passed in the entire list.

The IDE Editor provides code snippet support for **sub**. From the intellisense list at **sub**, press the TAB key. If there is more than one parameter input, the TAB key moves you between them, and the ENTER key will exit the snippet so that you can enter other commands. Hovering the mouse over a parameter will give you information about that input; hovering over the **sub** command will give you a description of the command and the syntax of the arguments.

Examples

```
open plc MyPlc                                  // Top-level PLC program
local MyEndTime;                                // Local to top-level section
local MyList(32);
…
callsub sub.GetEndTime(1000,&MyEndTime);
callsub sub.DelayOneSec();
callsub ShiftList(MyList(32),&MyList(32));
return;                                         // End of top-level section

sub: GetEndTime(TimeInc,&NewTime);              // Subroutine within program
  local StartTime;                              // Local to subroutine
```

```
  StartTime = Sys.Time;
  NewTime = StartTime + TimeInc;              // Result in returned variable
return;                                       // End of subroutine
sub: DelayOneSec(void)                        // Subroutine within program
  local DelayTime = Sys.Time + 1;             // Declare and set variable
  while (Sys.Time < DelayTime) {}
return;                                       // End of subroutine
sub: ShiftList(InList(32),&OutList(32));      // Subroutine within program
  local Index = 0;
  do {
    OutList(Index) = InList(Index+1);
    Index++;
  while (Index < 32);
  OutList(32) = 0;
return;                                       // End of subroutine
close                                         // End of program buffer
```

Subprogram and Subroutine Calling Commands

The pre-processor substantially improves the utility of the Script program commands that call subprograms and subroutines in Power PMAC so that arguments can be passed to and returned from these routines easily and intuitively.

call *{identifier}*

Function: Jump to subprogram, with return

Syntax: **call *{identifier}* [({call_list})]**

call *{identifier}*.*{sublabel}* [({sublabel_call_list})]

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the user subprogram name. It could also be an integer constant in the range 0 .. 2^{32}-1 or an identifier already assigned to one of these constants.
- the optional ***{call_list}*** is a set of identifiers for arguments to be passed to and from the subprogram
- ***{sublabel}*** is a text string representing the declared name of a subroutine within the subprogram
- the optional ***{sublabel_call_list}*** is a set of identifiers for arguments to be passed to and from the subroutine within the subprogram (which may be different from the call list to the start of the subprogram)

The **call** command causes program execution to jump to the subprogram (**subprog**) specified by the identifier. The identifier can just be the underlying number of the subprogram, but it can also be the user name of a subprogram declared with the pre-processor. The identifier is case-sensitive.

The optional "call list" allows for arguments to be passed to and from the subprogram on the local variable stack. It must match the call list declared for the subprogram. If no stack arguments are to be passed, the list must be eliminated, or the **()** empty parentheses can be used.

Arguments passed to the subprogram *are not* preceded by the '&' character. Note that in the **call** command (unlike the subprogram declaration), an argument passed to a subprogram can be a constant or an expression instead of a variable. Arguments returned from the program *are* preceded by the '&' character. A returned argument must be just a variable in both the call command and the subprogram definition.

It is possible for an item in the call list to be an array, specified with the syntax ***{identifier}*****({*start_index*})**, with the start index being a non-negative integer value. Note that the length of the array passed is specified in the subprogram definition; the index in the calling command specifies the starting element in the array to be used.

If there is no "sublabel" specified in the call command, the jump will be to the beginning of the specified subprogram. If a sublabel is specified, the jump will be to the start of matching subroutine within the subprogram.

The IDE Editor provides code snippet support for **call**. From the intellisense list at **call**, press the TAB key. If there is more than one parameter input, the TAB key moves you between them, and the ENTER key will exit the snippet so that you can enter other commands. Hovering the mouse over a parameter will give you information about that input; hovering over the **call** command will give you a description of the command and the syntax of the arguments.

Examples

```
// Call commands that use example subprogs at "open subprog"
call Pythag (Xvel, Yvel, &VecVel);
call SquareArray (RawData(10), &ProcData(10));

// Call commands that use subprogs shown below
call GetTime(&LoggedStartTime);
call GetTime.GetEndTime(60,&ProcEndTime);
call GetTime.DelayTimeSec(WaitTime);
call GetTime.DelayOneSec();

// Subprogram with subroutines
open subprog GetTime(&ReturnTime)     // Start of subprogram
  ReturnTime = Sys.Time;
return;                               // Jump back to calling routine
sub: GetEndTime(TimeInc,&NewTime)     // Start of subroutine within
  NewTime = Sys.Time + TimeInc;
return;                               // Jump back to calling routine
sub: DelayTimeSec(DelayTime)          // Start of subroutine within
  local Timer = Sys.Time + DelayTime
  while (Sys.Time < Timer) {}
return;                               // Jump back to calling routine
sub: DelayOneSec(void)                // Start of subroutine within
  local DelayTime = Sys.Time + 1;
  while (Sys.Time < DelayTime) {}
return;                               // Jump back to calling routine
close
```

callsub sub.*{identifier}*

Function: Jump to subroutine label within program, with return

Syntax: **callsub sub.*{identifier}* [(*{sublabel_call_list}*)]**

where:

- ***{identifier}*** is a text string beginning with a letter or the '_' underscore character and continuing with alphanumeric characters and '_' (no spaces), representing the user subroutine label.
- the optional ***{sublabel_call_list}*** is a set of identifiers for arguments to be passed to and from the subroutine

The **callsub** command causes program execution to jump to the subroutine label within the same program that is specified by the identifier. The identifier must be the user name of a subroutine declared with the pre-processor. The identifier is case-sensitive.

The optional "sublabel call list" allows for arguments to be passed to and from the subroutine on the local variable stack. It must match the sublabel call list declared for the subroutine. If no stack arguments are to be passed, the list must be eliminated, or the **()** empty parentheses can be used.

Arguments passed to the subroutine *are not* preceded by the '&' character. Note that in the **callsub** command (unlike the subprogram declaration), an argument passed to a subprogram can be a constant or expression instead of a variable. Arguments returned from the program *are* preceded by the '&' character. A returned argument must be just a variable in both the call command and the subprogram definition.

It is possible for an item in the call list to be an array, specified with the syntax ***{identifier}({start_index})***, with the start index being a non-negative integer value. Note that the length of the array passed is specified in the subroutine definition; the index in the calling command specifies the starting element in the array to be used.

The IDE Editor provides code snippet support for **callsub**. From the intellisense list at **callsub**, press the TAB key. If there is more than one parameter input, the TAB key moves you between them, and the ENTER key will exit the snippet so that you can enter other commands. Hovering the mouse over a parameter will give you information about that input; hovering over the **callsub** command will give you a description of the command and the syntax of the arguments.

Examples

```
open prog Myprog
local LoggedStartTime, ProcEndTime, WaitTime;

callsub sub.GetPresentTime(&LoggedStartTime);
callsub sub.GetEndTime(60,&ProcEndTime);
callsub sub.DelayTimeSec(WaitTime);
callsub sub.DelayOneSec();

return;                                 // End of main program

// Subroutines within same program
sub: GetPresentTime(&ReturnTime)
  ReturnTime = Sys.Time;
return;
sub: GetEndTime(TimeInc,&NewTime)
  NewTime = Sys.Time + TimeInc;
return;
sub: DelayTimeSec(DelayTime)
  local Timer = Sys.Time + DelayTime;
  while (Sys.Time < Timer) {}
return;
sub: DelayOneSec(void)
  local DelayTime = Sys.Time + 1;
  while (Sys.Time < DelayTime) {}
return;
close                                   // End of program buffer
```

POWER PMAC SCRIPT MATHEMATICAL FEATURE SPECIFICATION

The Power PMAC script language has a powerful and flexible set of operators and functions that make it easy for the user to implement a wide range of sophisticated calculations in an intuitive fashion. These mathematical capabilities are available in motion programs, PLC programs and on-line commands.

Operators

Power PMAC's script language provides several classes of operators: arithmetic, logical, standard assignment, synchronous assignment, conditional comparator, and conditional combinatorial operators.

Arithmetic Operators

The Power PMAC script language provides a full set of arithmetic operators, usable in motion programs, PLC programs, and on-line commands.

+

Function: Addition

The **+** operator implements the addition of the numerical values preceding and following it.

Multiplication, division, modulo (remainder), and bit-by-bit "and" operations have higher priority than addition, subtraction, bit-by-bit "or", and bit-by-bit "exclusive-or" operations. Operations of the same priority are implemented from left to right.

Note

The + sign may *not* be used as a "unary" operator to emphasize that the positive value of the following variable or constant is to be used (e.g. **`Var2=+Var1`** is not legal). This syntax will be rejected with an error.

-

Function: Subtraction or negation

The **-** operator implements the subtraction of the numerical value following it from the numerical value preceding it. If there is no numerical value immediately preceding it, it acts as a unary operator causing the negation of the numerical value following it (e.g. **`Var2=-Var1`**).

Multiplication, division, modulo (remainder), and bit-by-bit "and" operations have higher priority than addition, subtraction, bit-by-bit "or", and bit-by-bit "exclusive-or" operations. Operations of the same priority are implemented from left to right.

*

Function: Multiplication

The * operator implements the multiplication of the numerical values preceding and following it.

Multiplication, division, modulo (remainder), and bit-by-bit "and" operations have higher priority than addition, subtraction, bit-by-bit "or", and bit-by-bit "exclusive-or" operations. Operations of the same priority are implemented from left to right.

/

Function: Division

The / operator implements the division of the numerical value preceding it by the numerical value following it. The division operation is always a floating-point calculation (even if integer values are used). The quotient is computed as a floating-point value and used as such in subsequent calculations in the same expression; if it is then stored to an integer register, it is rounded at the time of storage.

Multiplication, division, modulo (remainder), and bit-by-bit "and" operations have higher priority than addition, subtraction, bit-by-bit "or", and bit-by-bit "exclusive-or" operations. Operations of the same priority are implemented from left to right.

If the divisor is equal to 0, the result will be the special floating-point representation of +/-infinity (inf). No error will be reported, and the program will not stop. Subsequent mathematical operations on this quotient will not yield predictable results. It is the programmer's responsibility to check for possible division-by-zero errors.

Examples

Operation	Result
`10 * 2 / 3`	6.666666667
`10 * ( 2 / 3)`	6.666666667
`10 / 0`	inf
`-10 / 0`	-inf

%

Function: Modulo (remainder)

The % operator causes the calculation of the remainder due to the division of the numerical value preceding it by the numerical value following it. The division operation is always a floating-point calculation (even if integer values are used). The quotient is computed as a floating-point value, then truncated to the next smallest (i.e. toward 0) integer so the remainder can be computed.

If the dividend "*n*" is a positive value, the modulo result is in the range $\{0 \leq \text{Result} < n\}$. If the dividend "*n*" is a negative value, the modulo result is in the range $\{-n < \text{Result} \leq 0\}$.

Note that the operation of the modulo operator with negative dividends and divisors is different in Power PMAC from the older PMAC and Turbo PMAC. Users may want to utilize the **rem** (remainder) function to emulate the operation of the older modulo operator when used with a negative divisor.

Multiplication, division, modulo (remainder), and bit-by-bit "and" operations have higher priority than addition, subtraction, bit-by-bit "or", and bit-by-bit "exclusive-or" operations. Operations of the same priority are implemented from left to right.

If the divisor is equal to 0, the division will saturate and the modulo result will be the special floating-point representation "not a number" (nan). No error will be reported, and the program will not stop. Subsequent mathematical operations on this quotient will not yield predictable results. It is the programmer's responsibility to check for possible division-by-zero errors.

Examples

Operation	Result
`11 % 4`	3
`-11 % 4`	-3
`11 % -4`	3
`-11 % -4`	-3
`3 % 2.5`	0.5
`-3 % 2.5`	-0.5
`3 % -2.5`	0.5
`-3 % -2.5`	-0.5
`3 % 0`	nan

Bit-by Bit Logical Operators

The Power PMAC script language provides the three standard bit-by-bit logical operators: and, or, and exclusive or. They can be used in motion programs, PLC programs, and on-line commands.

&

Function: Bit-by-bit "and"

The **&** operator implements the bit-by-bit logical "and" of the numerical value preceding it and the numerical value following it. A given bit of the result is equal to 1 if and only if the matching bits of both operands are equal to 1. The operation is done both on integer bits and fractional bits (if any).

Multiplication, division, modulo (remainder), and bit-by-bit "and" operations have higher priority than addition, subtraction, bit-by-bit "or", and bit-by-bit "exclusive-or" operations. Operations of the same priority are implemented from left to right.

This bit-by-bit "and" operator that logically combines the bits of numerical values is not to be confused with the && operator, which logically combines conditions.

Examples

Operation	Result
`3 & 1`	1
`3 & 2`	2
`3 & 3`	3
`3 & 4`	0
`$AA & $F`	$A (10)
`3 & -3`	1
`-3 & -2`	-4
`0.875 & 1.75`	0.75
`0.875 & -1.75`	0.25

|

Function: Bit-by-bit "or"

The **|** operator implements the bit-by-bit logical "or" of the numerical value preceding it and the numerical value following it. A given bit of the result is equal to 1 if the matching bit of either operand is equal to 1. The operation is done both on integer bits and fractional bits (if any).

Multiplication, division, modulo (remainder), and bit-by-bit "and" operations have higher priority than addition, subtraction, bit-by-bit "or", and bit-by-bit "exclusive-or" operations. Operations of the same priority are implemented from left to right.

This bit-by-bit "or" operator that logically combines the bits of numerical values is not to be confused with the || operator, which logically combines conditions.

Examples

Operation	Result
4 | 3	7
3 | 2	3
3 | 3	3
$F0 | $4	$F4 (244)
3 | -3	-1
-3 | -2	-1
0.5 | 0.375	0.875
0.875 | -1.75	-0.375

^

Function: Bit-by-bit "exclusive or"

The **^** sign implements the bit-by-bit logical "exclusive or" (XOR) of the numerical value preceding it and the numerical value following it. A given bit of the result is equal to 1 if and only if the matching bits of the two operands are different from each other. The operation is done both on integer bits and fractional bits (if any).

Multiplication, division, modulo (remainder), and bit-by-bit "and" operations have higher priority than addition, subtraction, bit-by-bit "or", and bit-by-bit "exclusive-or" operations. Operations of the same priority are implemented from left to right.

Examples

Operation	Result
2 ^ 1	3
2 ^ 2	0
5 ^ 7	2
$AA ^ $55	$FF
3 ^ -3	-2
0.5 ^ 0.875	0.375

<<

Function: Shift left

The **<<** operator implements the "shift left" operation, shifting the value preceding the operator by the number of bits specified by the value following the operator. It is effectively a multiplication by 2^n, where *n* is the value following the operator.

Both the value preceding the operator and the value following the operator can be floating-point values; they do not have to be integers for a valid operation.

This operator has equivalent algebraic precedence to multiplication and division.

>>

Function: Shift right

The **>>** operator implements the "shift right" operation, shifting the value preceding the operator by the number of bits specified by the value following the operator. It is effectively a division by 2^n, where *n* is the value following the operator.

Both the value preceding the operator and the value following the operator can be floating-point values; they do not have to be integers for a valid operation.

This operator has equivalent algebraic precedence to multiplication and division.

Standard Assignment Operators

The Power PMAC script language 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. These can be used in motion programs and PLC programs. Only the "=" basic assignment operator can be used in on-line 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.

=

Function: Basic assignment

Syntax: ***{variable} = {expression}***

The **=** assignment operator causes the mathematical expression following the operator to be evaluated and the resulting value assigned to the variable specified before the operator.

+=

Function: Assignment with addition

Syntax: ***{variable} += {expression}***

The **+=** assignment operator causes the mathematical expression following the operator to be evaluated, this value added to the present value of the variable specified before the operator, with the resulting value assigned to this variable.

-=

Function: Assignment with subtraction

Syntax: ***{variable} -= {expression}***

The **-=** assignment operator causes the mathematical expression following the operator to be evaluated, this value subtracted from the present value of the variable specified before the operator, with the resulting value assigned to this variable.

*=

Function: Assignment with multiplication

Syntax: ***{variable} *= {expression}***

The ***=** assignment operator causes the mathematical expression following the operator to be evaluated, this value multiplied with the present value of the variable specified before the operator, with the resulting value assigned to this variable.

/=

Function: Assignment with division

Syntax: ***{variable} /= {expression}***

The **/=** assignment operator causes the mathematical expression following the operator to be evaluated, this value divided into the present value of the variable specified before the operator, with the resulting value assigned to this variable.

If the divisor is equal to 0, the result will be the special floating-point representation of +/-infinity (inf). No error will be reported, and the program will not stop. Subsequent mathematical operations on this quotient will not yield predictable results. It is the programmer's responsibility to check for possible division-by-zero errors.

%=

Function: Assignment with modulo operation

Syntax: ***{variable} %= {expression}***

The **%=** assignment operator causes the mathematical expression following the operator to be evaluated, this value divided into the present value of the variable specified before the operator, the remainder of this division calculated, with the resulting value assigned to this variable.

If the divisor is equal to 0, the division will saturate and the modulo result will be the special floating-point representation "not a number" (nan). No error will be reported, and the program will not stop. Subsequent mathematical operations on this quotient will not yield predictable results. It is the programmer's responsibility to check for possible division-by-zero errors.

&=

Function: Assignment with bit-by-bit "and"

Syntax: ***{variable} &= {expression}***

The **&=** assignment operator causes the mathematical expression following the operator to be evaluated, this value logically combined with a bit-by-bit "and" operation to the present value of the variable specified before the operator, with the resulting value assigned to this variable.

|=

Function: Assignment with bit-by-bit "or"

Syntax: ***{variable} |= {expression}***

The **|=** assignment operator causes the mathematical expression following the operator to be evaluated, this value logically combined with a bit-by-bit "or" operation to the present value of the variable specified before the operator, with the resulting value assigned to this variable.

^=

Function: Assignment with bit-by-bit "exclusive or"

Syntax: ***{variable} ^= {expression}***

The **^=** assignment operator causes the mathematical expression following the operator to be evaluated, this value logically combined with a bit-by-bit "exclusive-or" (XOR) operation to the present value of the variable specified before the operator, with the resulting value assigned to this variable.

>>=

Function: Assignment with shift-right operation

Syntax: ***{variable} >>= {expression}***

The **>>=** assignment operator causes the mathematical expression following the operator to be evaluated, this value used to specify the number of bits to shift right the present value of the variable specified before the operator, with the resulting value assigned to this variable.

<<=

Function: Assignment with shift-left operation

Syntax: ***{variable} <<= {expression}***

The **<<=** assignment operator causes the mathematical expression following the operator to be evaluated, this value used to specify the number of bits to shift left the present value of the variable specified before the operator, with the resulting value assigned to this variable.

++

Function: Increment assignment

Syntax: ***{variable}++***

The **++** assignment operator causes the value of the variable preceding it to be incremented by 1.0 (toward +infinity). It may not be used as part of a more complex expression. The variable does not need to contain an integer value for proper operation of the increment operation.

--

Function: Decrement assignment

Syntax: ***{variable}--***

The **--** assignment operator causes the value of the variable preceding it to be decremented by 1.0 (toward –infinity). It may not be used as part of a more complex expression. The variable does not need to contain an integer value for proper operation of the decrement operation.

Synchronous Assignment Operators

Synchronous assignment operators are intended for use during blended move sequences in motion programs, where the statements in the motion program must be evaluated well before the corresponding commanded moves are actually executed. Use of a standard assignment operator would cause the value to be assigned to the variable out of sequence with the actual move execution. While this is appropriate for variables that will be used in the calculation of the move equations, if the assignment is used for a task such as setting a physical output value, the output would be set out of sequence with the associated moves.

Synchronous assignment operators eliminate this problem by delaying the actual assignment of the evaluated expression until the beginning of the actual execution of the next commanded move. The expression following the synchronous assignment operator is evaluated as it is encountered during the program calculation flow, but the resulting operation is placed in a pending queue and not actually implemented until the start of the actual execution of the next commanded move in the program.

Synchronous assignment operators are also useful in PLC programs that command axis or motor moves so subsequent logic can be sure that the commanded moves have actually started before looking to see that they have finished.

Synchronous assignment operators cannot be used to assign values to local variables, or "self-defined" M-variables. They cannot be used in on-line commands.

==

Function: Standard synchronous assignment

Syntax: ***{variable} == {expression}***

The **==** synchronous assignment operator causes the mathematical expression following the operator to be evaluated, with the resulting value placed in an "output queue", for assignment to the variable specified before the operator at the beginning of actual execution of the next move in the motion program.

Note that the **==** symbol inside a condition is the equality comparator.

+==

Function: Synchronous assignment with addition

Syntax: ***{variable} +== {expression}***

The **+==** assignment operator causes the mathematical expression following the operator to be evaluated, with the resulting value placed in an "output queue", for addition to the present value of the variable specified before the operator at the beginning of actual execution of the next move in the motion program, with the resulting value assigned to this variable.

-==

Function: Synchronous assignment with subtraction

Syntax: ***{variable} -== {expression}***

The **-==** assignment operator causes the mathematical expression following the operator to be evaluated, with the resulting value placed in an "output queue", for subtraction from the present value of the variable specified before the operator at the beginning of actual execution of the next move in the motion program, with the resulting value assigned to this variable.

*==

Function: Synchronous assignment with multiplication

Syntax: ***{variable} *== {expression}***

The ***==** assignment operator causes the mathematical expression following the operator to be evaluated, with the resulting value placed in an "output queue", for multiplication with the present value of the variable specified before the operator at the beginning of actual execution of the next move in the motion program, with the resulting value assigned to this variable. The variable must be of floating-point format, not integer.

/==

Function: Synchronous assignment with division

Syntax: ***{variable} /== {expression}***

The **/==** assignment operator causes the mathematical expression following the operator to be evaluated, with the resulting value placed in an "output queue", for division into the present value of the variable specified before the operator at the beginning of actual execution of the next move in the motion program, with the resulting value assigned to this variable. The variable must be of floating-point format, not integer.

If the divisor is equal to 0, the result will be the special floating-point representation of +/-infinity (inf). No error will be reported, and the program will not stop. Subsequent mathematical operations on this quotient will not yield predictable results. It is the programmer's responsibility to check for possible division-by-zero errors.

++=

Function: Synchronous increment assignment

Syntax: ***{variable}++=***

The **++=** assignment operator causes the value of the variable preceding it to be incremented by 1.0, with the resulting value placed in an "output queue" for assignment back to the variable at the beginning of the actual execution of the next programmed move. It may not be used as part of a more complex expression. The variable does not need to contain an integer value for proper operation of the increment operation.

--=

Function: Synchronous decrement assignment

Syntax: ***{variable}--=***

The **--=** assignment operator causes the value of the variable preceding it to be decremented by 1.0, with the resulting value placed in an "output queue" for assignment back to the variable at the beginning of the actual execution of the next programmed move. It may not be used as part of a more complex expression. The variable does not need to contain an integer value for proper operation of the decrement operation.

&==

Function: Synchronous assignment with bit-by-bit "and"

Syntax: ***{variable} &== {expression}***

The **&==** assignment operator causes the mathematical expression following the operator to be evaluated, with the resulting value placed in an "output queue", for logical combination using the bit-by-bit "and" operation with the present value of the variable specified before the operator at the beginning of actual execution of the next move in the motion program, with the resulting value assigned to this variable. The variable must be of integer format, not floating-point.

|==

Function: Synchronous assignment with bit-by-bit "or"

Syntax: ***{variable} |== {expression}***

The **|==** assignment operator causes the mathematical expression following the operator to be evaluated, with the resulting value placed in an "output queue", for logical combination using the bit-by-bit "or" operation with the present value of the variable specified before the operator at the beginning of actual execution of the next move in the motion program, with the resulting value assigned to this variable. The variable must be of integer format, not floating-point.

^==

Function: Synchronous assignment with bit-by-bit "exclusive-or"

Syntax: ***{variable} ^== {expression}***

The **^==** assignment operator causes the mathematical expression following the operator to be evaluated, with the resulting value placed in an "output queue", for logical combination using the bit-by-bit "exclusive-or" (XOR) operation with the present value of the variable specified before the operator at the beginning of actual execution of the next move in the motion program, with the resulting value assigned to this variable. The variable must be of integer format, not floating-point.

Conditional Comparators

The Power PMAC script language has a full set of conditional comparators to provide powerful and flexible logical capabilities. The result of a comparison is a Boolean true/false value that may be logically combined with other Boolean true false values inside a compound condition, but it may not be used as part of a general mathematical expression.

==

Function: Equality comparator

Syntax: ***{exp1} == {exp2}***

The **==** conditional comparator causes the mathematical expressions on both sides of it to be evaluated and compared. It returns a Boolean "true" if the two values are exactly equal; it returns a Boolean "false" if they are not exactly equal.

Note that the **==** symbol in a variable assignment command is the synchronous assignment operator.

!=

Function: Inequality comparator

Syntax: ***{exp1} != {exp2}***

The **!=** conditional comparator causes the mathematical expressions on both sides of it to be evaluated and compared. It returns a Boolean "true" if the two values are not exactly equal; it returns a Boolean "false" if they are exactly equal.

The **<>** syntax can also be used identically, although the program will list back with the **!=** syntax.

<

Function: Less-than comparator

Syntax: ***{exp1} < {exp2}***

The **<** conditional comparator causes the mathematical expressions on both sides of it to be evaluated and compared. It returns a Boolean "true" if the first value is less than the second value (i.e. closer to negative infinity); it returns a Boolean "false" if the two values are exactly equal or if the second value is less than the first value.

>

Function: Greater-than comparator

Syntax: ***{exp1} > {exp2}***

The **>** conditional comparator causes the mathematical expressions on both sides of it to be evaluated and compared. It returns a Boolean "true" if the first value is greater than the second value (i.e. closer to positive infinity); it returns a Boolean "false" if the two values are exactly equal or if the second value is greater than the first value.

<=

Function: Less-than-or-equal-to comparator

Syntax: ***{exp1} <= {exp2}***

The **<=** conditional comparator causes the mathematical expressions on both sides of it to be evaluated and compared. It returns a Boolean "true" if the first value is less than the second value (i.e. closer to negative infinity) or if the two values are exactly equal; it returns a Boolean "false" if the second value is less than the first value.

The **!>** syntax can also be used identically, although the program will list back with the **<=** syntax.

>=

Function: Greater-than-or-equal-to comparator

Syntax: ***{exp1} >= {exp2}***

The **>=** conditional comparator causes the mathematical expressions on both sides of it to be evaluated and compared. It returns a Boolean "true" if the first value is greater than the second value (i.e. closer to positive infinity) or if the two values are exactly equal; it returns a Boolean "false" if the second value is greater than the first value.

The **!<** syntax can also be used identically, although the program will list back with the **>=** syntax.

~

Function: Approximate equality comparator

Syntax: ***{exp1} ~ {exp2}***

The **~** conditional comparator causes the mathematical expressions on both sides of it to be evaluated and compared. It returns a Boolean "true" if the magnitude of the difference between

the two values is less than 0.5; it returns a Boolean "false" if the magnitude of the difference is 0.5 or greater.

Note that the **~** symbol outside of a condition is the bit-by-bit invert function.

!~

Function: Approximate inequality comparator

Syntax: ***{exp1 } !~ {exp2}***

The **!~** conditional comparator causes the mathematical expressions on both sides of it to be evaluated and compared. It returns a Boolean "true" if the magnitude of the difference between the two values is 0.5 or greater; it returns a Boolean "false" if the magnitude of the difference is less than 0.5.

<>

Function: Inequality comparator (alternate syntax)

Syntax: ***{exp1} <> {exp2}***

{See **!=**}

The **<>** conditional comparator is the alternate syntax for the **!=** inequality comparator. Although **<>** can be used in place of **!=**, it is stored and reports back as **!=**.

!>

Function: Less-than-or-equal-to comparator (alternate syntax)

Syntax: ***{exp1} !> {exp2}***

{See **<=**}

The **!>** conditional comparator is the alternate syntax for the **<=** less-than-or-equal-to comparator. Although **!>** can be used in place of **<=**, it is stored and reports back as **<=**.

!<

Function: Greater-than-or-equal-to comparator (alternate syntax)

Syntax: ***{exp1} !< {exp2}***

{See **>=**}

The **!<** conditional comparator is the alternate syntax for the **>=** greater-than-or-equal-to comparator. Although **!<** can be used in place of **>=**, it is stored and reports back as **>=**.

Conditional Combinatorial Operators

Power PMAC provides conditional combinatorial operators that permit the Boolean results of simple conditions to be combined into compound conditions.

&&

Function: Logical "and"

Syntax: ***{condition1} && {condition2}***

The **&&** conditional combinatorial operator causes the Boolean results of the logical conditions on both sides of it to be combined in a logical "and" operation. The resulting value is a Boolean "true" value if both conditions are "true"; it is a "Boolean "false" value if one or both of the conditions is "false". The conditions can be either explicit conditions with logical comparators or simply mathematical expressions that are true if they evaluate to any non-zero value.

||

Function: Logical "or"

Syntax: ***{condition1} || {condition2}***

The **||** conditional combinatorial operator causes the Boolean results of the logical conditions on both sides of it to be combined in a logical "or" operation. The resulting value is a Boolean "true" value if either condition is "true"; it is a "Boolean "false" value if both of the conditions are "false". The conditions can be either explicit conditions with logical comparators or simply mathematical expressions that are true if they evaluate to any non-zero value.

!

Function: Logical negation

Syntax: **!(*{condition}*)**

The **!** logical negation operator causes the Boolean results of the condition in the following parentheses to be negated. If the condition is true, or the expression has any non-zero value, the resulting value is "false". If the condition is false, or the expression has a value of exactly 0, the resulting value is "true". The condition must be an explicit condition with a logical comparator; it cannot be simply a mathematical expression.

Functions

Power PMAC implements a large set of mathematical functions as part of its script language's floating-point math library. These functions can be used in motion programs, PLC programs, and on-line commands.

Scalar Mathematical Functions

The following group of functions operates on and returns scalar (as opposed to vector or matrix) quantities.

~

Function: Bit-by-bit invert

Syntax: **~({*expression*})**

Domain: All real numbers

Domain units: User-determined

Range: All real numbers

Range units: User-determined

The **~** function implements the bit-by-bit invert operation, changing each bit of the value immediately following. It is primarily intended for use on unsigned integer values, but will operate on other types as well.

Note that the **~** symbol inside a condition is the approximate-equality comparator.

Examples

```
M1->*u.1                    // Self-referenced 1-bit var
M1=0                        // To start
M1=~(M1)                    // Invert bit
M1                          // Query value
1                           // Power PMAC response
M1=~(M1)                    // Invert bit
M1                          // Query value
0                           // Power PMAC response
M1->*u.4                    // Self-referenced 4-bit var
M1=0                        // To start
M1=~(M1)                    // Invert bits
M1                          // Query value
15                          // Power PMAC response
M1=~(M1)                    // Invert bits
M1                          // Query value
0                           // Power PMAC response
```

abs

Function: Absolute value

Syntax: **abs(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: Non-negative real numbers

Range units: User-determined

abs implements the absolute-value, or magnitude, function, of the mathematical expression contained inside the following parentheses.

Examples

```
Var2=abs(Var1)                              // Computes mag of Var1
if (Var3 <> 0) SgnVar3=abs(Var3)/Var3       // Sign of non-zero num
else SgnVar3=0                              // Sign value is 0
```

acos

Function: Trigonometric arc-cosine

Syntax: **acos(*{expression}*)**

Domain: -1.0 to +1.0

Domain units: none

Range: 0 to Pi

Range units: Radians

acos implements the standard trigonometric inverse cosine, or arc-cosine, function, of the mathematical expression contained inside the following parentheses, returning the angle in radians.

For an arc-cosine function with the result in units of degrees, use **acosd** instead.

If the argument inside the parentheses is outside of the legal domain of –1.0 to +1.0, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Angle=acos(CosValue)          // Finds angle whose cosine is CosValue
C(acos(Length/10))            // Move C axis to specified angle
```

acosd

Function: Trigonometric arc-cosine in degrees

Syntax: **acosd ({*expression*})**

Domain: -1.0 to +1.0

Domain units: none

Range: 0 to 180

Range units: Degrees

acosd implements the trigonometric inverse cosine, or arc-cosine, function, of the mathematical expression contained inside the following parentheses, returning the angle in degrees.

For the standard arc-cosine function with the result in units of radians, use **acos** instead.

If the argument inside the parentheses is outside of the legal domain of –1.0 to +1.0, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Angle=acosd(CosValue)          // Finds angle whose cosine is CosValue
C(acosd(Length/10))            // Move C axis to specified angle
```

acosh

Function: Inverse hyperbolic cosine

Syntax: **acosh({*expression*})**

Domain: Positive real numbers > 1.0

Domain units: none

Range: Positive real numbers

Range units: Radians

acosh implements the inverse hyperbolic cosine function of the mathematical expression contained inside the following parentheses.

If the argument inside the parentheses is outside of the legal domain of +1.0 to +infinity, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Example

```
Var2=acosh(Var1)
```

asin

Function: Trigonometric arc-sine

Syntax: **asin(*{expression}*)**

Domain: -1.0 to +1.0

Domain units: none

Range: -Pi/2 to Pi/2

Range units: Radians

asin implements the standard trigonometric inverse sine, or arc-sine, function, of the mathematical expression contained inside the following parentheses, returning the angle in radians.

For an arc-sine function with the result in units of degrees, use **asind** instead.

If the argument inside the parentheses is outside of the legal domain of –1.0 to +1.0, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Angle=asin(SinValue)          // Finds angle whose sine is SinValue
C(asin(Rise/10))              // Move C axis to specified angle
```

asind

Function: Trigonometric arc-sine in degrees

Syntax: **asind(*{expression}*)**

Domain: -1.0 to +1.0

Domain units: none

Range: 0 to 180

Range units: Degrees

asind implements the trigonometric inverse sine, or arc-sine, function, of the mathematical expression contained inside the following parentheses, returning the angle in degrees.

For the standard arc-sine function with the result in units of radians, use **asin** instead.

If the argument inside the parentheses is outside of the legal domain of –1.0 to +1.0, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Angle=asind(SinValue)          // Finds angle whose sine is SinValue
C(asind(Rise/10))              // Move C axis to specified angle
```

asinh

Function: Inverse hyperbolic sine

Syntax: **asinh(*{expression}*)**

Domain: All real numbers

Domain units: none

Range: All real numbers

Range units: Radians

asinh implements the inverse hyperbolic sine function of the mathematical expression contained inside the following parentheses.

Example

```
Var2=asinh(Var1)
```

atan

Function: Trigonometric arc-tangent

Syntax: **atan(*{expression}*)**

Domain: All real numbers

Domain units: none

Range: -Pi/2 to +Pi/2

Range units: Radians

atan implements the standard trigonometric inverse tangent, or arc-tangent, function, of the mathematical expression contained inside the following parentheses, returning the angle in

radians. This standard arc-tangent function returns values only in the +/-Pi/2-radian range; if a full +/-Pi-radian range is desired, the **atan2** function should be used instead.

For an arc-tangent function with the result in units of degrees, use **atand** instead.

Examples

```
Angle=atan(TanValue)                    // Finds angle whose tangent is TanValue
C(atan(Rise/Run))                       // Move C axis to specified angle
```

atan2

Function: Two-argument trigonometric arc-tangent

Syntax: **atan2(*{expression1}*,*{expression2}*)**

Domain: All real numbers in both arguments (but not 0 in both simultaneously)

Domain units: none

Range: -Pi to +Pi

Range units: Radians

atan2 implements the expanded (two-argument) inverse tangent, or arc-tangent, function, of the two mathematical expressions contained inside the following parentheses, returning the angle in radians.

This expanded arc-tangent function returns values in the full +/-Pi-radian range; if only the +/-Pi/2-radian range is desired, the standard **atan** function should be used instead. The **atan2** function makes use of the signs of both arguments, as well as the ratio of their magnitudes, to extend the range to a full 360 degrees. The first value in the parentheses following **atan2** is the "sine" argument; the second is the "cosine" argument. Note that only the signs and relative magnitudes of the two arguments are important; neither must actually be equal to the sine or cosine of the resulting angle.

For a two-argument arc-tangent function with the result in units of degrees, use **atan2d** instead.

If both arguments for the **atan2** function are equal to exactly 0.0, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Angle=atan2(SinAngle,CosAngle)
if (abs(XVel)>0 || abs(YVel)>0) VelAngle=atan2(YVel,XVel)
```

atan2d

Function: Two-argument trigonometric arc-tangent in degrees

Syntax: **atan2d(*{expression1}*,*{expression2}*)**

Domain: All real numbers in both arguments (but not 0 in both simultaneously)

Domain units: none

Range: -180 to +180

Range units: Degrees

atan2d implements the expanded (two-argument) inverse tangent, or arc-tangent, function, of the two mathematical expressions contained inside the following parentheses, returning the angle in degrees.

This expanded arc-tangent function returns values in the full +/-180-degree range; if only the +/-90-degree range is desired, the **atand** function should be used instead. The **atan2d** function makes use of the signs of both arguments, as well as the ratio of their magnitudes, to extend the range to a full 360 degrees. The first value in the parentheses following **atan2d** is the "sine" argument; the second is the "cosine" argument. Note that only the signs and relative magnitudes of the two arguments are important; neither must actually be equal to the sine or cosine of the resulting angle.

For a two-argument arc-tangent function with the result in units of radians, use **atan2** instead.

If both arguments for the **atan2d** function are equal to exactly 0.0, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Angle=atan2d(SinAngle,CosAngle)
if (abs(XVel)>0 || abs(YVel)>0) VelAngle=atan2d(YVel,XVel)
```

atand

Function: Trigonometric arc-tangent in degrees

Syntax: **atand(*{expression}*)**

Domain: All real numbers

Domain units: none

Range: -90 to +90

Range units: Degrees

atand implements the trigonometric inverse tangent, or arc-tangent, function, of the mathematical expression contained inside the following parentheses, returning the angle in degrees.

This single-argument arc-tangent function returns values only in the +/-90-degree range; if a full +/-180-degree range is desired, the **atand2** function should be used instead.

For a two-argument arc-tangent function with the result in units of radians, use **atan2** instead.

Examples

```
Angle=atand(TanValue)          // Finds angle whose tangent is TanValue
C(atand(Rise/Run))             // Move C axis to specified angle
```

atanh

Function: Inverse hyperbolic tangent

Syntax: **atanh(*{expression}*)**

Domain: -1.0 to +1.0

Domain units: none

Range: All real numbers

Range units: Radians

atanh implements the inverse hyperbolic tangent function of the mathematical expression contained inside the following parentheses.

If the argument inside the parentheses is outside of the legal domain of -1.0 to +1.0, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Example

```
Var2=atanh(Var1)
```

cbrt

Function: Cube root

Syntax: **sqrt(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: All real numbers

Range units: User-determined

cbrt implements the real cube-root function of the mathematical expression contained inside the following parentheses.

Example

```
Var1=cbrt(Var2)
```

ceil

Function: Round to higher integer

Syntax: **ceil(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: All integers

Range units: User-determined

ceil implements the rounding-to-higher-integer function of the mathematical expression contained inside the following parentheses. The rounding is always done in the positive direction.

Note that while the result is an integer number, it is still represented as a floating-point value.

Examples

```
Var1=2.5
Var2=ceil(Var1)              // Take ceil of positive value
Var2                         // Query resulting value
3                            // Next higher integer value
Var3=ceil(-Var1)             // Take ceil of negative value
Var3                         // Query resulting value
-2                           // Next higher integer value
```

cos

Function: Trigonometric cosine

Syntax: **cos(*{expression}*)**

Domain: All real numbers

Domain units: Radians

Range: -1.0 to +1.0

Range units: none

cos implements the standard trigonometric cosine function of the mathematical expression (in units of radians) contained inside the following parentheses.

For a cosine function with the argument in units of degrees, use **cosd** instead.

Examples

```
XDist=VecDist*cos(Angle)
X(Radius*cos(Azimuth))
```

cosd

Function: Trigonometric cosine using degrees

Syntax: **cosd(*{expression}*)**

Domain: All real numbers

Domain units: Degrees

Range: -1.0 to +1.0

Range units: none

cosd implements the trigonometric cosine function of the mathematical expression (in units of degrees) contained inside the following parentheses.

For the standard cosine function with the argument in units of radians, use **cos** instead.

Examples

```
XDist=VecDist*cosd(Angle)
X(Radius*cosd(Azimuth))
```

cosh

Function: Hyperbolic cosine

Syntax: **cosh(*{expression}*)**

Domain: All real numbers

Domain units: Radians

Range: Positive real numbers >= 1.0

Range units: none

cosh implements the hyperbolic cosine function of the mathematical expression contained inside the following parentheses.

Example

```
Var2=cosh(Var1)
```

exp

Function: Exponentiation base e (e^x)

Syntax: **exp(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: Positive real numbers

Range units: User-determined

exp implements the standard exponentiation function of the mathematical expression contained inside the following parentheses, raising "e" to the power of this expression.

It is possible to use $e^{x \ln(y)}$ to implement the y^x function, but **pow(y,x)** can be used as well.

Examples

```
Var2=exp(Var1)                 // Raises e to the power of Var1
Var3=exp(Var2*log(Var1))       // Raises Var1 to the power of Var2
```

exp2

Function: Exponentiation base 2 (2^x)

Syntax: **exp2(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: Positive real numbers

Range units: User-determined

exp2 implements the exponentiation function for base 2 of the mathematical expression contained inside the following parentheses, raising "2" to the power of this expression.

Examples

```
Var2=exp2(Var1)                    // Raises 2 to the power of Var1
```

floor

Function: Round to lower integer

Syntax: **floor(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: All integers

Range units: User-determined

floor implements the truncation to integer function of the mathematical expression contained inside the following parentheses. The truncation is always done in the negative direction.

Note that while the result is an integer number, it is still represented as a floating-point value. Unlike the similar **int** function, the **floor** function can return values outside the range of 32-bit signed integers.

Examples

```
Var1=2.5
Var2=floor(Var1)              // Take floor of positive value
Var2                          // Query resulting value
2                             // Next lower integer value
Var3=floor(-Var1)             // Take floor of negative value
Var3                          // Query resulting value
-3                            // Next lower integer value
```

int

Function: Round to lower integer

Syntax: **int(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: All integers

Range units: User-determined

int implements the truncation to integer function of the mathematical expression contained inside the following parentheses. The truncation is always done toward zero.

Unlike the similar **floor** function, **int** produces a 32-bit signed integer value. If the magnitude of the expression value is greater than what can be represented by a 32-bit signed integer, the result is truncated to the maximum positive or negative 32-bit integer value ($+2^{31}-1$ or -2^{31}). The resulting 32-bit integer value may then be converted to floating-point format in subsequent operations or assignments.

Examples

```
Var1=2.5
Var2=int(Var1)                  // Take int of positive value
Var2                            // Query resulting value
2                               // Next lower integer value
Var3=int(-Var1)                 // Take int of negative value
Var3                            // Query resulting value
-2                              // Next lower integer value
```

isnan

Function: “Not-a-number” check

Syntax: **isnan(*{expression}*)**

Domain: All real numbers, “not a number” representation

Domain units: User-determined

Range: 0, 1

Range units: None

isnan permits the user to check whether the expression evaluates to a valid numerical value or to the “not-a-number” (nan) representation for invalid values. The result is 0 for a valid numerical value, and 1 for “not-a-number”.

The "not-a-number" representation can come from a function domain error, as in taking the square root of a negative number, or from the value of a data structure element for hardware that is not physically present in the system.

Examples

```
InvalidSoln=isnan(acosd(Var2))
ICError=isnan(Gate1[Index].PwmPeriod)
```

ln

Function: Natural logarithm (alternate syntax)

{See **log**}

ln is the alternate syntax the natural logarithm (logarithm base "e") function. While **ln** can be used in place of **log**, it is stored and reports back as **log**.

log

Function: Natural logarithm

Syntax: **log(*{expression}*)**, or **ln(*{expression}*)**

Domain: Positive real numbers

Domain units: User-determined

Range: All real numbers

Range units: User-determined

log implements the natural logarithm (logarithm base "e") function of the mathematical expression contained inside the following parentheses. The **ln** syntax can also be used identically, although the program will list back with the **log** syntax.

To implement the logarithm using another base, you can divide the natural logarithm of the value by the natural logarithm of the base ($\log_y x = \ln x / \ln y$). Power PMAC provides the dedicated functions **log2** and **log10** for bases 2 and 10, respectively.

If the argument inside the parentheses is exactly 0, the result will be the special floating-point representation of "minus infinity" (-inf). If the argument inside the parentheses is a negative value, the result will be the special floating-point representation of "not a number" (nan). In either case, no error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Var2=log(Var1)                  // Takes the natural log of Var1
```

```
Var2=ln(Var1)                  // Takes the natural log of Var1
Var4=log(Var3)/log(8)          // Takes the log base 8 of Var3
```

log10

Function: Logarithm base 10

Syntax: **log10(*{expression}*)**

Domain: Positive real numbers

Domain units: User-determined

Range: All real numbers

Range units: User-determined

log10 implements the logarithm base-10 function of the mathematical expression contained inside the following parentheses.

If the argument inside the parentheses is exactly 0, the result will be the special floating-point representation of "minus infinity" (-inf). If the argument inside the parentheses is a negative value, the result will be the special floating-point representation of "not a number" (nan). In either case, no error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Example

```
Var2=log10(Var1)                        // Takes the log base 10 of Var1
```

log2

Function: Logarithm base 2

Syntax: **log2(*{expression}*)**

Domain: Positive real numbers

Domain units: User-determined

Range: All real numbers

Range units: User-determined

log2 implements the logarithm base-2 function of the mathematical expression contained inside the following parentheses.

If the argument inside the parentheses is exactly 0, the result will be the special floating-point representation of "minus infinity" (-inf). If the argument inside the parentheses is a negative value, the result will be the special floating-point representation of "not a number" (nan). In either case, no error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Var2=log2(Var1)                    // Takes the log base 2 of Var1
```

madd

Function: Multiply and add

Syntax: **madd(*{expression},{expression},{expression}*)**

Domains: All real numbers

Domain units: User-determined

Range: All real numbers

Range units: User-determined

madd implements a combined multiplication and addition operation, with the intermediate value held at extended precision. The values of the first two expressions are multiplied, and this (extended precision) product is added to the value of the third expression. The resulting sum is returned.

This function can be useful for evaluating the precision of certain mathematical operations.

Examples

```
P1=madd(sqrt(3),sqrt(3),-3)
P1
P1=-3.47625514608046876e-16
P2=sqrt(3)*sqrt(3)-3
P2
P2=-4.44089209850062616e-16
```

pow

Function: General exponentiation

Syntax: **pow(*{expression},{expression}*)**

Domains: All real numbers (but see combination limitations below)

Domain units: User-determined

Range: All real numbers

Range units: User-determined

pow implements the general exponentiation function, raising the value of the first expression to the power of the second expression.

If the first expression has a positive value, the second expression can be any real number, and the function will return a valid numerical value.

If the first expression has a zero value, the second expression must have a non-negative value in order for the function to return a valid numerical value; if the second expression has a negative value in this case, the result will be the special floating-point representation of "infinity" (inf). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

If the first expression has a negative value, the second expression must have an exact integer value in order for the function to return a valid numerical value; if the second expression has a non-integer value in this case, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Var3=pow(Var1,Var2)
```

qnrt

Function: Quintic (5^{th}) root

Syntax: **qnrt*({expression})***

Domain: All real numbers

Domain units: User-determined

Range: All real numbers

Range units: User-determined

qnrt implements the real quintic (5^{th}) root function of the mathematical expression contained inside the following parentheses.

Examples

```
Var2=qnrt(Var1)
```

qrrt

Function: Quartic (4^{th}) root

Syntax: **qrrt(*{expression}*)**

Domain: Non-negative real numbers

Domain units: User-determined

Range: Non-negative real numbers

Range units: User-determined

qrrt implements the positive real quartic (4^{th}) root function of the mathematical expression contained inside the following parentheses.

If the argument inside the parentheses is outside of the legal domain of non-negative numbers, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Var2=qrrt(Var1)
```

randx

Function: 64-bit random number generator

Syntax: **randx(*{expression}*)**

Domain: All real numbers

Domain units: none

Range: 0.0 to argument-value

Range units: none

randx implements a random-number generator, with the argument specifying the range of the returned value. The function returns a 64-bit floating-point value with a range of 0.0 to the value of the argument (positive or negative).

Foreground and background priority levels have separate random-number generators with their own seed values. Foreground tasks include motion programs and real-time PLCs; background tasks include background PLCs and on-line commands.

Each call of the **randx** function uses all four seed values (0, 1, 2, and 3) for the priority level, and updates them in preparation for the next call to the generator. All of the seed values are set to

specific values on power-up/reset, and many users will be satisfied to start with these default values. It is also possible to set seed values at any time with the **seed** function.

Example

```
Var1=randx(2.0)-1.0          // Result is between -1.0 and +1.0
```

rem

Function: Remainder

Syntax: **rem(*{expression},{expression}*)**

Domain: All real numbers (non-zero for 2nd argument)

Domain units: User-determined

Range: Non-negative real numbers

Range units: User-determined

rem implements the remainder function, returning the smallest-magnitude remainder when the value in the first expression is divided by the value of the second expression. The returned value will be in the range:

-Divisor/2 < Returned Value <= Divisor/2

Note that this range is different from the resulting range of the % modulo operator.

If the second expression has a value of exactly 0, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Var1=5
Var2=rem(Var1,2)             // Find remainder modulo 2
Var2                         // Query resulting value
1                            // Reports remainder value
Var1=5.01
Var3=rem(Var1,2)             // Find remainder modulo 2
Var3                         // Query resulting value
-0.99                        // Reports rounded value
```

rint

Function: Round to nearest integer

Syntax: **rint(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: All integers

Range units: User-determined

rint implements the round-to-nearest-integer function of the mathematical expression contained inside the following parentheses. If the fractional value of the expression is exactly 0.5, the result is always rounded up (towards positive infinity)

Note that while the result is an integer number, it is still represented as a floating-point value.

Examples

```
Var1=2.4
Var2=rint(Var1)                 // Round to nearest integer
Var2                            // Query resulting value
2                               // Reports rounded value
Var3=rint(Var1+0.2)             // Round expression to nearest integer
Var3                            // Query resulting value
3                               // Reports rounded value
Var4=rint(Var1+0.1)             // Round expression to nearest integer
Var4                            // Query resulting value
3                               // Reports rounded (up) value
```

rnd

Function: 32-bit random number generator

Syntax: **rnd(*{expression}*)**

Domain: 0 to 1

Domain units: none

Range: 0.0 to 0.99999…

Range units: none

rnd implements a random-number generator, with the "ones bit" of the argument being the generator number 0 or 1. There are two independent random-number generators (0 and 1) for foreground tasks such as motion programs and real-time PLCs, and two other independent random-number generators (also 0 and 1) for background tasks such as background PLCs and on-line commands. The argument can be any real number, but only the ones bit is evaluated to determine which generator is used. Usually, the function will be used either as **rnd(0)** or **rnd(1)**.

Regardless of the value of the argument, the function returns a value that is greater than or equal to 0.0 and less than 1.0. The returned number is a 32-bit floating-point value.

Each access to a given random-number generator uses two "seed" values. Generator 0 uses seed values 0 and 1; generator 1 uses seed values 2 and 3. In the process of computing the returned random value, both of these seed values are automatically updated to new values in preparation for the next access of this generator. All of the seed values are set to specific values on power-up/reset, and many users will be satisfied to start with these default values. It is also possible to set seed values at any time with the **seed** function.

Example

```
Var1=0; Var2=0;                    // Initialize values
while(Var1<1000) {                 // Loop 1000 times
  Var2+=rnd(1);                    // Add random value to sum
  Var1++;                          // Increment cycle counter
}                                  // Sum is ~on normal distribution
Var2=(Var2/500)-1;                 // Normalize and center on zero
```

seed

Function: Random number seed generator

Syntax: **seed(*{expression},{expression}*)**

Domain 1: Signed 32-bit integers

Domain 1 units: none

Domain 2: 0 .. 3

Domain 2 units: seed number

Range: Unsigned 32-bit integers

Range units: none

seed implements a generator of "seed" values for random-number generation. Two seed values are used for each random number generator. While it is possible to rely on the default starting seed values, this function permits you to set your own starting seed values.

The second argument specifies which seed value is to be generated. It can take a value of 0, 1, 2, or 3, specifying which of the four seed values for the priority level will be generated. (Both foreground tasks – motion programs and real-time PLCs – and background tasks – background PLCs and on-line commands – have independent sets of four seeds each.) At each priority level, **rnd(0)** uses seeds 0 and 1; **rnd(1)** uses seeds 2 and 3. The 64-bit random-number generation function **randx** uses all four seeds for the priority level.

The first argument specifies how the seed value is to be generated. If it is greater than zero, the seed value is set equal to the argument. If it is less than zero, the seed value is set from the value of the processor's internal timer register at the moment the function is invoked. The first case permits repeatable random-number sequences; the second case creates non-repeatable sequences.

If the first argument is equal to zero, the seed value is not changed, and the function simply returns the present value of the seed.

In all cases, the resulting value of the seed is both stored in an internal data structure and used as the returned value of the function. In most cases, the returned value is not of direct use, so is typically assigned to a dummy variable.

sin

Function: Trigonometric sine

Syntax: **sin(*{expression}*)**

Domain: All real numbers

Domain units: Radians

Range: -1.0 to +1.0

Range units: none

sin implements the standard trigonometric sine function of the mathematical expression (in units of radians) contained inside the following parentheses.

For a sine function with the argument in units of degrees, use **sind** instead.

Examples

```
YDist=VecDist*sin(Angle)
Y(Radius*sin(Azimuth))
```

sincos

Function: Combined trigonometric sine and cosine function

Syntax: **sincos(*{expression1}*,*{expression2}*)**

Domain1: All real numbers

Domain1 units: Radians

Domain2: Non-negative integers

Domain2 units: Local variable numbers

Range: -1.0 to +1.0

Range units: none

sincos implements a special combined trigonometric sine/cosine function of the first mathematical expression (in units of radians) contained inside the following parentheses. The result of the function is the sine of the first expression.

The second expression specifies the number of the local ("L") variable for the program or task executing the function where the calculated cosine value is placed. Local variable numbers are non-negative integers, and the second expression is rounded down, if necessary, to the nearest integer value to select which variable is used.

If both sine and cosine of an angle must be calculated, the combined **sincos** function is faster than separate uses of **sin** and **cos**.

For a combined sine/cosine function with the argument in units of degrees, use **sincosd** instead.

Examples

```
L1=VecDist*sincos(Angle,2)            // Sine into L1, cosine into L2
X(VecDist*L2)Y(L1)
```

sincosd

Function: Combined trigonometric sine and cosine function in degrees

Syntax: **sincosd(*{expression1},{expression2}*)**

Domain1: All real numbers

Domain1 units: Degrees

Domain2: Non-negative integers

Domain2 units: Local variable numbers

Range: -1.0 to +1.0

Range units: none

sincosd implements a special combined trigonometric sine/cosine function of the first mathematical expression (in units of degrees) contained inside the following parentheses. The result of the function is the sine of the first expression.

The second expression specifies the number of the local ("L") variable for the program or task executing the function where the calculated cosine value is placed. Local variable numbers are non-negative integers, and the second expression is rounded down, if necessary, to the nearest integer value to select which variable is used.

If both sine and cosine of an angle must be calculated, the combined **sincosd** function is faster than separate uses of **sind** and **cosd**.

For a combined sine/cosine function with the argument in units of radians, use **sincos** instead.

Examples

```
L1=VecDist*sincosd(Angle,2)          // Sine into L1, cosine into L2
X(VecDist*L2)Y(L1)
```

sind

Function: Trigonometric sine using degrees

Syntax: **sind(*{expression}*)**

Domain: All real numbers

Domain units: Degrees

Range: -1.0 to +1.0

Range units: none

sind implements the trigonometric sine function of the mathematical expression (in units of degrees) contained inside the following parentheses.

For the standard sine function with the argument in units of radians, use **sin** instead.

Examples

```
YDist=VecDist*sind(Angle)
Y(Radius*sind(Azimuth))
```

sinh

Function: Hyperbolic sine

Syntax: **sinh(*{expression}*)**

Domain: All real numbers

Domain units: Radians

Range: All real numbers

Range units: none

sinh implements the hyperbolic sine function of the mathematical expression contained inside the following parentheses.

Example

```
Var2=sinh(Var1)
```

sgn

Function: Algebraic sign

Syntax: **sgn(*{expression}*)**

Domain: All real numbers

Domain units: User-determined

Range: -1, 0, +1

Range units: None

sgn implements the algebraic sign ("signum") function of the mathematical expression contained inside the following parentheses. The result will be -1 if the expression has a negative value, +1 if the expression has a positive value, and 0 if the expression is exactly zero in value.

Examples

```
Direction=sgn(CmdVel)
```

sqrt

Function: Square root

Syntax: **sqrt(*{expression}*)**

Domain: Non-negative real numbers

Domain units: User-determined

Range: Non-negative real numbers

Range units: User-determined

sqrt implements the positive square-root function of the mathematical expression contained inside the following parentheses.

If the argument inside the parentheses is outside of the legal domain of non-negative numbers, the result will be the special floating-point representation of "not a number" (nan). No error will be reported, and the program will not stop. Subsequent operations on this value will yield unpredictable results. It is the programmer's responsibility to check for possible domain errors.

Examples

```
Var2=sqrt(Var1)
VecVel=sqrt(XVel*XVel+YVel*YVel)
```

tan

Function: Trigonometric tangent

Syntax: **tan(*{expression}*)**

Domain: All real numbers except +/-(2N-1)*Pi/2

Domain units: Radians

Range: All real numbers

Range units: none

tan implements the standard trigonometric tangent function of the mathematical expression (in units of radians) contained inside the following parentheses.

If the argument inside the parentheses approaches +/-(2N-1)*Pi/2 radians (+/-Pi/2, 3Pi/2, 5Pi/2, etc.), the **tan** function will "blow up" and a very large value will be returned. Subsequent calculations based on such a result will be of questionable validity. It is the programmer's responsibility to check for these conditions.

For a tangent function with the argument in units of degrees, use **tand** instead.

Examples

```
YDist=XDist*tan(Angle)
```

tand

Function: Trigonometric tangent using degrees

Syntax: **tand(*{expression}*)**

Domain: All real numbers except +/-(2N-1)*90 degrees

Domain units: Degrees

Range: All real numbers

Range units: none

tand implements the trigonometric tangent function of the mathematical expression (in units of degrees) contained inside the following parentheses.

If the argument inside the parentheses approaches +/-(2N-1)*90 degrees (+/-90, 270, 450, etc.), the **tand** function will "blow up" and a very large value will be returned. Subsequent calculations based on such a result will be of questionable validity. It is the programmer's responsibility to check for these conditions.

For the standard tangent function with the argument in units of radians, use **tan** instead.

Examples

```
YDist=XDist*tand(Angle)
```

tanh

Function: Hyperbolic tangent

Syntax: **tanh(*{expression}*)**

Domain: All real numbers

Domain units: Radians

Range: -1.0 to +1.0

Range units: none

tanh implements the hyperbolic tangent function of the mathematical expression contained inside the following parentheses.

Example

```
Var1=tanh(Var2)
```

Vector Mathematical Functions

The Power PMAC Script language provides a set of vector functions that permit powerful and flexible one-dimensional (1D) vector mathematical operations without the need to work with individual elements of the vector. A vector in Power PMAC uses a consecutively numbered set of global ("P") variables, local ("L") variables, or user buffer array double-precision floating-point (**Sys.Ddata[*i*]**) variables, depending on the setting of local non-saved **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The arguments of vector functions that specify variable numbers, element spacing, or number of elements are always truncated to an integer value (rounded towards negative infinity) before use in the function. If specified directly as an integer, or completed using only integer values, there should be no worries about rounding errors. However, if any floating-point values or functions were used in calculating these values, it is suggested that the application force proper rounding (e.g. **int(Var+0.5)**) to ensure that the desired integer value is used. Otherwise, slight inaccuracies in the calculation could cause an unintended value to be used (e.g. 2.9999999999 would specify 2).

When used within the IDE's project manager, references can be made to declared variable names without knowing the underlying variable numbers by using the declared name preceded by an & (ampersand) character. For example:

```
global MyVector1(6), MyVector2(6);
…
vcopy(&MyVector1(0),&MyVector2(0),6);
```

Note

Some of these vector functions return a value that is not the main result of the function. In firmware versions before V2.0, these returned values had to be assigned to a variable. However, starting in V2.0, released 1st quarter 2015, if used in the expression for **nop** program statement, no such assignment needs to be made.

sum

Function: Sum of vector elements

Syntax: **sum(*{exp1},{exp2},{exp3}*)**

The **sum** function calculates the (scalar) sum of a specified number of elements of the vector specified by the values of the expressions inside the following parentheses, and returns the resulting sum. The vector is formed by a set of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the vector.

The second expression ***exp2*** specifies the number of elements in that vector to be added.

The third expression ***exp3*** specifies the increment (spacing) between the elements to be added. A value of 1 in this expression specifies the use of consecutive elements in the vector.

The **sum** function can be useful for calculating the "trace" of a square matrix (the sum of the diagonal elements). In this use, the second expression should be equal to the number of rows and columns in the second matrix, and the third expression should be one greater than the second expression.

Example

```
P20=10 P21=5  P22=0          // Set first row of matrix
P23=5  P24=12 P25=0          // Set second row of matrix
P26=1  P27=2  P28=15         // Set third row of matrix
P30=sum(20,3,4)              // Add P20, P24, P28
P30
P30=37
```

sumprod

Function: Sum of products of vector elements

Syntax: **sumprod(*{exp1},{exp2},{exp3},{exp4},{exp5}*)**

The **sumprod** function calculates the (scalar) sum of a specified number of elements of the vectors specified by the values of the expressions inside the following parentheses, and returns the resulting sum. This function is useful for computing the dot product of vectors, including rows and columns of 2D matrices.

The vectors are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the first vector to be used in the operation.

The second expression ***exp2*** specifies the number of the starting variable of the second vector to be used in the operation.

The third expression ***exp3*** specifies the number of elements to be used in both source vectors.

The fourth expression ***exp4*** specifies the increment (spacing) between the elements to be used in the first vector.

The fifth expression ***exp5*** specifies the increment (spacing) between the elements to be used in the second vector. A value of 1 in these expressions specifies the use of consecutive elements in the vector.

Examples

```
P50=10 P51=5  P52=2          // Set contents of first vector
```

```
P60=20 P61=3  P62=1            // Set contents of second vector
P70=sumprod(50,60,3,1,1)       // Compute dot product of vectors
P70                            // Query result
P70=217

P20=1  P21=2  P22=3            // Set first row of matrix
P23=4  P24=5  P25=6            // Set second row of matrix
P26=7  P27=8  P28=9            // Set third row of matrix
P40=2  P41=3  P42=4            // Set contents of second vector
P10=sumprod(20,40,3,3,1)       // Multiply first column of matrix by vector
P10                            // Query result
P10=42
```

vcopy

Function: Vector copy

Syntax: **vcopy(*{exp1},{exp2},{exp3}*)**

The **vcopy** function copies the values of all of the elements in one vector (the "source" vector) to the elements of another vector (the "result" vector). The vectors are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the result vector.

The second expression ***exp2*** specifies the number of the starting variable of the source vector.

The third expression ***exp3*** specifies the number of elements both the source vector and the result vector.

The returned value of the function is a Boolean flag specifying the success of the operation. It is set to 1 on a successful operation, and to 0 on an unsuccessful operation. (Lack of success is usually due to a specifier out of range, such as a zero or negative number of elements.) Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

The **vcopy** function can also be used to copy 2D matrices. For the purpose of copying, they can be treated as 1D vectors. For example, 3 x 4 matrices can be treated as 12-element vectors.

Example

```
P80=10 P81=5  P82=2            // Set contents of first vector
P90=vcopy(90,80,3)             // Copy into second vector
P90,3                          // Query contents of second vector
P90=10
P91=5
P92=2
```

vmadd

Function: Vector multiply and add

Syntax: **vmadd(*{exp1},{exp2},{exp3},{exp4},{exp5}*)**

The **vmadd** function calculates the sum of the two vectors (with the first vector scaled by a constant before the addition) specified by the values of the expressions inside the following parentheses, and saves the resulting sum vector. The vectors are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the result (sum) vector.

The second expression ***exp2*** specifies the scale factor by which the elements of the first vector to be added will be multiplied before the vectors are added.

The third expression ***exp3*** specifies the number of the starting variable of the first vector to be added.

The fourth expression ***exp4*** specifies the number of the starting variable of the second vector to be added.

The fifth expression ***exp5*** specifies the number of elements in both source vectors and the result vector.

Setting the scale factor to +1 permits straight addition of the two source vectors. Setting the scale factor to -1 causes the first source vector to be subtracted from the second source vector.

The returned value of the function is a Boolean flag specifying the success of the operation. It is set to 1 on a successful operation, and to 0 on an unsuccessful operation. (Lack of success is usually due to a specifier out of range, such as a zero or negative number of elements.) Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

The **vmadd** function can also be used to add 2D matrices together. For the purpose of addition, they can be treated as 1D vectors. For example, 3 x 4 matrices can be treated as 12-element vectors.

Example

```
P30=5  P31=10 P32=3            // Set contents of first vector
P50=3  P51=6  P52=8            // Set contents of second vector
P0=vmadd(70,1,50,30,3)         // Standard addition of vectors
P70,3                          // Query result
P70=8
P71=16
P72=11
P0=vmadd(70,-1,50,30,3)        // Subtraction of vectors
P70,3                          // Query result
P70=2
P71=4
P72=-5
```

vscale

Function: Vector scale

Syntax: **vscale(*{exp1},{exp2},{exp3},{exp4]*)**

The **vscale** function takes the values of all of the elements in one vector (the "source" vector), multiplies them by a common scale factor, and writes the product values to the elements of another vector (the "result" vector). The vectors are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the result vector.

The second expression ***exp2*** specifies the scale factor as a floating-point number.

The third expression ***exp3*** specifies the number of the starting variable of the source vector.

The fourth expression ***exp4*** specifies the number of elements both the source vector and the result vector.

The returned value of the function is a Boolean flag specifying the success of the operation. It is set to 1 on a successful operation, and to 0 on an unsuccessful operation. (Lack of success is usually due to a specifier out of range, such as a zero or negative number of elements.) Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

The **vscale** function can also be used to scale 2D matrices. For the purpose of scaling, they can be treated as 1D vectors. For example, 3 x 4 matrices can be treated as 12-element vectors.

Example

```
P80=10 P81=5  P82=2              // Set contents of vector
P0=vscale(90,4,80,3)             // Scale vector by factor of 4
P90,3                            // Query contents of result vector
P90=40
P91=20
P92=8
```

Matrix Mathematical Functions

The Power PMAC Script language provides a set of matrix functions that permit powerful and flexible two-dimensional (2D) matrix mathematical operations without the need to work with individual elements of the matrix. A matrix in Power PMAC uses a consecutively numbered set of global ("P") variables, local ("L") variables, or user buffer array double-precision floating-point (**Sys.Ddata[*i*]**) variables, depending on the setting of local non-saved **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

Matrix elements on the same row use consecutively numbered variables; matrix elements on the same column are separated in variable number (index) by the number of columns.

Specifiers for row numbers and column numbers start at 0, so a matrix specified with 3 rows and 4 columns has Rows 0, 1, and 2, and Columns 0, 1, 2, and 3.

The returned value of a matrix function is always a single scalar value. If the main result of the function is a scalar, such as a determinant, this value is the returned value of the function. However, if the main result of the function is a matrix, the returned value will be an auxiliary value, such as a Boolean flag indicating successful completion or not, or the determinant of a matrix involved in the calculations.

Note

In firmware versions before V2.0, these returned values had to be assigned to a variable. However, starting in V2.0, released 1st quarter 2015, if used in the expression for **nop** program statement, no such assignment needs to be made.

Note that since the 2D matrices in Power PMAC are fundamentally arrays of consecutively numbered variables, they can also be treated as vectors for certain operations. The vector functions **vadd**, **vscale**, and **vcopy**, for adding, scaling, and copying, respectively, can be very useful for 2D matrices as well.

The arguments of matrix functions that specify variable numbers, row or column numbers, or number of rows or columns are always truncated to an integer value (rounded towards negative infinity) before use in the function. If specified directly as an integer, or completed using only integer values, there should be no worries about rounding errors. However, if any floating-point values or functions were used in calculating these values, it is suggested that the application force proper rounding (e.g. **int(Var+0.5)**) to ensure that the desired integer value is used. Otherwise, slight inaccuracies in the calculation could cause an unintended value to be used (e.g. 2.9999999999 would specify 2).

mdet

Function: Matrix determinant

Syntax: **mdet(*{exp1},{exp2}*)**

The **mdet** function calculates and returns the determinant of the square matrix specified by the values of the expressions inside the following parentheses. The matrix is formed by a set of

consecutively numbered global ("P") variables. The matrix is formed by a set of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the matrix.

The second expression ***exp2*** specifies the number of rows and columns in the square matrix.

Example

```
P50=2  P51=0                    // Set first row of matrix
P52=0  P53=2                    // Set second row of matrix
P54=mdet(50,2)                  // Compute determinant of matrix
P54                             // Query value
4
```

minv

Function: Matrix inverse

Syntax: **minv(*exp1*,*exp2*,*exp3*)**

The **minv** function calculates and saves the inverse of the square matrix specified by the values of the expressions inside the following parentheses. Both the source and the result matrices are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the result (inverted) matrix.

The second expression ***exp2*** specifies the number of the starting variable of the source matrix.

The third expression ***exp3*** specifies the number of rows and columns in the square source and result matrices.

The returned value of the function is the determinant of the source matrix. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation. If it is exactly 0, the inversion is invalid, and if it is very close to 0 (the definition of "very close" is application-dependent), the inversion may be questionable.

Example

```
P50=2  P51=0                    // Set first row of matrix
P52=0  P53=2                    // Set second row of matrix
P54=minv(60,50,2)               // Compute inverse of matrix
P54                             // Query determinant of inverse matrix
P54=4
P60,4                           // Query contents of inverse matrix
P60=0.5
P61=0
P62=0
P63=0.5
```

mminor

Function: Matrix minor determinant

Syntax: **mminor(*{exp1},{exp2},{exp3},{exp4}*)**

The **mdet** function calculates and returns the minor determinant of the square matrix specified by the values of the expressions inside the following parentheses. The matrix is formed by a set of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the overall matrix.

The second expression ***exp2*** specifies the number of rows and columns in the square matrix.

The third expression ***exp3*** specifies the row number of the overall matrix that is eliminated from the minor matrix.

The fourth expression ***exp4*** specifies the column number of the overall matrix that is eliminated from the minor matrix. Note that row and column numbers start with 0, not 1.

Example

```
P60=2 P61=0 P62=0                   // Set first row of matrix
P63=0 P64=3 P65=0                   // Set second row of matrix
P66=0 P67=0 P68=4                   // Set first row of matrix
P70=mminor(60,3,0,0)                // Take determinant of lower right minor
P70                                 // Query result
12
P70=mminor(60,3,1,1)                // Take determinant of outer corner minor
P70                                 // Query result
8
P70=mminor(60,3,2,2)                // Take determinant of upper left minor
P70                                 // Query result
6
```

mmadd

Function: Matrix multiply and add

Syntax: **mmadd(*{exp1},{exp2},{exp3},{exp4},{exp5},{exp6}*)**

The **mmadd** function calculates the product of two matrices specified by the values of the expressions inside the following parentheses, adds the resulting product matrix to a third matrix, saving the sum in that matrix. Both the source and the result matrices are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the result (product and sum) matrix. The values in this matrix before the function is executed will be added to the values of the product of the other two matrices.

The second expression ***exp2*** specifies the number of the starting variable of the first matrix to be multiplied.

The third expression ***exp3*** specifies the number of the starting variable of the second matrix to be multiplied. Remember that matrix multiplication is not commutative, so the order of these matrices matters.

The fourth expression ***exp4*** specifies the number of rows in the first source matrix to be multiplied, and so the number of rows in the result (product) matrix.

The fifth expression ***exp5*** specifies the number of columns in the first matrix to be multiplied, and so the number of rows in the second matrix to be multiplied.

The sixth expression ***exp6*** specifies the number of columns in the second matrix to be multiplied, and so the number of columns in the result (product) matrix.

The returned value of the function is a Boolean flag specifying the success of the operation. It is set to 1 on a successful operation, and to 0 on an unsuccessful operation. (Lack of success is usually due to a specifier out of range, such as a zero or negative number of rows or columns.) Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Example

```
P10=1  P11=2  P12=3                    // Set first row of first matrix
P13=4  P14=5  P15=6                    // Set second row of first matrix

P20=7  P21=8                           // Set first row of second matrix
P22=9  P23=10                          // Set second row of second matrix
P24=11 P25=12                          // Set third row of second matrix

P30=58  P31=64                         // Set first row of result matrix
P32=139 P33=154                        // Set second row of result matrix
P40=mmadd(30,10,20,2,3,2)              // Multiply two matrices, add to result
P30,4                                  // Query result matrix
P30=116
P31=128
P32=278
P33=308
```

mmul

Function: Matrix multiply

Syntax: **mmul(*{exp1},{exp2},{exp3},{exp4},{exp5},{exp6}*)**

The **mmul** function calculates the product of the two matrices specified by the values of the expressions inside the following parentheses, and saves the resulting product matrix. Both the source and the result matrices are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the result (product) matrix.

The second expression ***exp2*** specifies the number of the starting variable of the first matrix to be multiplied.

The third expression ***exp3*** specifies the number of the starting variable of the second matrix to be multiplied. Remember that matrix multiplication is not commutative, so the order of these matrices matters.

The fourth expression ***exp4*** specifies the number of rows in the first source matrix to be multiplied, and so the number of rows in the result (product) matrix.

The fifth expression ***exp5*** specifies the number of columns in the first matrix to be multiplied, and so the number of rows in the second matrix to be multiplied.

The sixth expression ***exp6*** specifies the number of columns in the second matrix to be multiplied, and so the number of columns in the result (product) matrix.

The returned value of the function is a Boolean flag specifying the success of the operation. It is set to 1 on a successful operation, and to 0 on an unsuccessful operation. (Lack of success is usually due to a specifier out of range, such as a zero or negative number of rows or columns.) Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Examples

```
P10=1  P11=2  P12=3                    // Set first row of first matrix
P13=4  P14=5  P15=6                    // Set second row of first matrix
P20=7  P21=8                           // Set first row of second matrix
P22=9  P23=10                          // Set second row of second matrix
P24=11 P25=12                          // Set third row of second matrix
P40=mmul(30,10,20,2,3,2)               // Multiply first matrix by second
P30,4                                  // Query result matrix
P30=58
P31=64
P32=139
P33=154
P40=mmul(30,20,10,3,2,3)               // Multiply second matrix by first
P30,9                                  // Query result matrix
```

```
P30=39
P31=54
P32=69
P33=49
P34=68
P35=87
P36=59
P37=82
P38=105
```

msolve

Function: Matrix solve

Syntax: **msolve(*{exp1},{exp2},{exp3},{exp4}*)**

The **msolve** function calculates the solution(s) of the set(s) of equations expressed by two matrices and saves the result by overwriting one of the matrices. The matrices are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the "constant" matrix (really set of column vectors) that serves as an input to the function, and which becomes the result matrix (set of column vectors) that contains the solution(s) to the set of equations.

The second expression ***exp2*** specifies the number of the starting variable of the (square) matrix that contains the variable coefficients.

The third expression ***exp3*** specifies the number of rows and columns in the square variable-coefficient matrix.

The fourth expression ***exp4*** specifies the number of columns in the constant/result matrix, and therefore the number of sets of equations to solve. This matrix has the same number of rows as the variable-coefficient matrix. For a single set of equations, this fourth expression should evaluate to 1, so the constant/result matrix is really a column vector.

The **msolve** function is useful to solve one or more complete sets of equations, each set of the form

$$\vec{y} = A\vec{x}$$

where *A* is an *n*-by-*n* square matrix, with $\vec{y}$ and $\vec{x}$ being *n*-row column vectors. The $\vec{y}$ vector (or set of vectors) is the first matrix specified for this function, and the *A* coefficient matrix is the second matrix specified. These are the "inputs" to the function. The **msolve** function computes the $\vec{x}$ vector(s) that provide the solution to the set of equations, and overwrites the input $\vec{y}$ vector with this solution.

If your equations are in the form

$$\vec{y} = A\vec{x} + \vec{b}$$

you will first need to subtract the $\vec{b}$ vector from the $\vec{y}$ vector using the **vmadd** function and use the resulting $(\vec{y} - \vec{b})$ vector as the first input to the **msolve** function.

The returned value of the function is the determinant of the square coefficient matrix. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the solution. If it is exactly 0, the solution is invalid, and if it is very close to 0 (the definition of "very close" is application-dependent), the solution may be questionable.

Examples

```
P60=1000 P61=0                          // Set first row of A matrix
P62=0    P63=1000                       // Set second row of A matrix
P70=2                                   // Set first element of B vector
P71=3                                   // Set second element of B vector
P1=msolve(70,60,2,1)                    // Solve set of equations
P70,2                                   // Query solution vector
P70=0.002
P71=0.003
P1                                      // Query matrix determinant
P1=1000000

P40=0.866  P41=-0.500 P42=0.000         // Set first row of A matrix
P43=0.500  P44=0.866  P45=0.000         // Set second row of A matrix
P46=0.000  P47=0.000  P48=1.000         // Set third row of A matrix

P80=10.0   P81=7.071                    // Set first elements of B vectors
P82=0.0    P83=7.071                    // Set second elements of B vectors
P84=5.0    P85=1.000                    // Set third elements of B vectors

P100=msolve(80,40,3,2)                  // Solve both sets of equations
P80,6                                   // Query result vectors
P80=8.6603810567664965
P81=9.65941101408461833
P82=-5.0002200096804259
P83=2.58809987639456152
P84=5
P85=1
P100                                    // Query matrix determinant
P100=0.999955999999999956
```

mtrans

Function: Matrix transpose

Syntax: **mtrans(*{exp1},{exp2},{exp3},{exp4}*)**

The **mtrans** function calculates and saves the transpose of the matrix specified by the values of the expressions inside the following parentheses. Both the source and the result matrices are formed by sets of consecutively numbered global ("P") variables, local ("L") variables, or user

buffer array double (**Sys.Ddata[*i*]**) variables, depending on the setting of **Ldata.Control** for the task (= 0 for P, 1 for L, or 2 for Ddata).

The first expression ***exp1*** specifies the number of the starting variable of the result (transposed) matrix.

The second expression ***exp2*** specifies the number of the starting variable of the source matrix.

The third expression ***exp3*** specifies the number of rows in the result matrix, and so the number of columns in the source matrix.

The fourth expression ***exp4*** specifies the number of columns in the result matrix, and so the number of rows in the source matrix.

The returned value of the function is a Boolean flag specifying the success of the transpose. It is set to 1 on a successful transpose operation, and to 0 on an unsuccessful transpose operation. (Lack of success is usually due to a specifier out of range, such as a zero or negative number of rows or columns.) Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Example

```
P40=1 P41=3 P42=5                    // Set first row of matrix
P43=7 P44=9 P45=11                   // Set second row of matrix
P46=mtrans(50,40,3,2)                // Take transpose of matrix
P46                                  // Query success flag
P46=1
P50,6                                // Query result matrix
P50=1
P51=7
P52=3
P53=9
P54=5
P55=11
```

Transformation Matrix Mathematical Functions

The Power PMAC Script language provides a set of functions for working with the axis transformation data structures. These permit various types of transformations – rotations, offsets, scalings, etc – to be implemented and combined quickly and easily.

Note

These transformation-matrix functions return a value that is not the main result of the function. In firmware versions before V2.0, these returned values had to be assigned to a variable. However, starting in V2.0, released 1st quarter 2015, if used in the expression for **nop** program statement, no such assignment needs to be made.

tinit

Function: Transformation matrix initialize

Syntax: **tinit(*exp1*)**

The **tinit** function initializes the axis-transformation matrix data structure **Tdata[*i*]** whose index is specified by the value of ***exp1***. The initialization sets all of the diagonal elements **Tdata[*i*].Diag[*m*]** to 1.0, all of the offset elements **Tdata[*i*].Bias[*m*]** to 0.0, and all of the off-diagonal elements **Tdata[*i*].UVW[*m*]**, **Tdata[*i*].XYZ[*m*]**, **Tdata[*i*].UUVVWW[*m*]**, and **Tdata[*i*].XXYYZZ[*m*]** to 0.0. This makes the axis transformation matrix an "identity matrix", providing no real transformation. It is useful to get the matrix in a known starting state.

The returned value of the function can be treated as a Boolean flag specifying the success of the initialization. If a valid transformation matrix index number is selected, Power PMAC computes the determinant of this matrix, which should be equal to 1.0. If an invalid transformation matrix index number is selected, Power PMAC sets the return value to 0.0 to indicate the operation was not successful. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

tprop

Function: Transformation matrix propagate (combine)

Syntax: **tprop(*exp1*,*exp2*,*exp3*)**

The **tprop** ("transform propagate") function combines the transformations of two axis-transformation matrices and places the result in a third axis-transformation matrix.

The result transformation matrix is the **Tdata[*i*]** data structure whose index is specified by ***exp1***. This result transformation matrix can be the same matrix as one of the two matrices to be combined.

The first transformation matrix to be combined is the **Tdata[*i*]** data structure whose index is specified by ***exp2***. (Remember that the combination of transformation matrices is not commutative.)

The second transformation matrix to be combined is the **Tdata[*i*]** data structure whose index is specified by ***exp3***.

The square "rotation" matrix of the first axis transformation is multiplied by the square "rotation" matrix of the second axis transformation, and the resulting "rotation" matrix is written to the result axis transformation. Then the square "rotation" matrix of the first axis transformation is multiplied by the column "offset" vector of bias terms of the second axis transformation. The resulting column vector is then added to the first transformations "offset" vector and the resulting sum vector is written to the "offset" vector of the result axis transformation.

The combination can be thought of as the straight multiplication of two extended "homogeneous" matrices, where the homogenous matrices append the "offset" vector to the square rotation matrix as an additional column on the right, and an additional row is added to the bottom with all 0 values under the original rotation matrix and a 1 value under the "offset" vector column.

The returned value of the function is the determinant of the resulting axis-transformation "rotation" matrix. A value of 0.0 indicates an invalid resulting matrix, either because an out of range index number, or a "degenerate" resulting transformation. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

String Functions

The Power PMAC Script language provides a full set of string manipulation functions. These permit the creation, manipulation, and processing of text strings. In Power PMAC, strings are character (byte) arrays (**Sys.Cdata[*i*]**) in the user shared memory buffer, referenced by their starting (byte) address offset (index) from the beginning of the user shared memory buffer.

Note

Some of these string functions return a value that is not the main result of the function. In firmware versions before V2.0, these returned values had to be assigned to a variable. However, starting in V2.0, released 1st quarter 2015, if used in the expression for **nop** program statement, no such assignment needs to be made.

sprintf

Function: Create formatted text string

Syntax: **sprintf(*{expression}*,"*{string}*"[,*{expression}*…])**

The **sprintf** function converts the numerical value of a mathematical expression to a formatted string.

The first ***{expression}*** specifies the starting index in user shared memory of the "target" string variable into which the formatted text representation of the value will be placed.

The ***{string}*** inside double quotes is a formatted ASCII text string consisting of literal alphanumeric characters, escape sequences, and formatted variable sequences.

The following escape sequences can be used (all letters must be lower case):

- `\a` Bell (ASCII 7)
- `\b` Backspace (ASCII 8)
- `\t` Horizontal tab (ASCII 9)
- `\n` New line (ASCII 10)
- `\v` Vertical tab (ASCII 11)
- `\f` Form feed (ASCII 12)
- `\r` Carriage return (ASCII 13)
- `\\` Backslash character
- `\?` Question-mark character
- `\'` Single-quote character
- `\"` Double-quote character
- `\ooo` Octal specification of ASCII character code (of range `000` to `377`)
- `\xhh` Hexadecimal specification of ASCII character code (of range `x00` to `xff`)

(Note that if you explicitly insert a null terminator (`'\0'` or `'\x0'`) character into the string, subsequent uses of the string variable will treat this as the end of the string, not the null terminator character that Power PMAC automatically appends after the user-specified text.)

The following format sequences can be used (all letters must be used in upper or lower case as shown):

- `%d` Signed integer, decimal format
- `%u` Unsigned integer, decimal format
- `%x` Unsigned integer, hexadecimal format, use lower case
- `%`*n*`x` Unsigned integer, hexadecimal format, using *n* digits (*n* = 1 to 8), use lower case
- `%X` Unsigned integer, hexadecimal format, use upper case
- `%`*n*`X` Unsigned integer, hexadecimal format, using *n* digits (*n* = 1 to 8), use upper case
- `%f` Floating-point value, up to 6 digits total
- `%`*n*`f` Floating-point value, up to *n* digits total (*n* = 1 to 31)
- `%`*n*`.`*m*`f` Floating-point value, up to *n* digits total (*n* = 1 to 31), up to *m* fractional digits (*m* < *n*)
- `%s` Text string, arbitrary length (null terminated)
- `%`*n*`s` Text string, up to *n* characters (shorter if null terminator encountered)
- `%c` Single character of specified byte value
- `%`*n*`c` *n* repetitions of single character of specified byte value
- `%%` "Percent" character

Leading zeros are not created for decimal-format integers. Leading zeros are created for hexadecimal-format integers (only) when the number of digits is specified. Leading and trailing zeros are not created for floating-point values, whether or not the number of digits is specified. If the number of digits specified for a floating-point is not sufficient to represent the value without an exponent, it will automatically be represented with an exponent (e.g. `2.345e8`). The decimal point is only used for floating-point values if there is a non-zero fractional component. In floating-point values, any decimal point, minus sign, or exponent used does not count as a digit.

For each formatted variable sequence in the string, there must be an ***{expression}*** that specifies the numerical value that is to be converted to text. Each expression can be as simple as a constant or a variable, but can include mathematical operators and expressions as well. For a string variable, the expression value (rounded down to the next integer if necessary) represents the starting index in user shared memory of the string variable to be used.

Power PMAC will automatically append a "null" terminating character (ASCII code 0) to the end of the string it creates.

The returned value of the function after successful execution is the number of characters in the string variable, not including the "null" terminating character. If the function is unsuccessful, a value of -1 is returned. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Examples

```
EndIndex = sprintf(256, "%d", CycleNum)
EndIndex = sprintf(256, "Cycle # %d executing\n", CycleNum)
EndIndex = sprintf(MaskStringIndex, "%6x", OutputMask)
nop(sprintf(XposStringIndex, "X: %9.4f",Motor[1].DesPos))
```

strcat

Function: Add to end of string variable

Syntax: **strcat(*{expression}*,*{expression}*)**

strcat(*{expression}*,"*{string}*")

The **strcat** function appends a text string of limited length onto the end of a string variable. This string can be in another string variable, or it can be specified directly.

The first ***{expression}*** specifies the starting index in user shared memory of the "target" string variable onto which the text string will be appended.

If there is a second ***{expression}*** following the comma, it specifies the starting index in user shared memory of the "source" string variable from which the text string will be appended. The appending will start from this index and continue until the "null" string terminator is found. Note that if there is not a "null" terminating character where expected, a very long sequence of bytes can be appended to the target string, overwriting other elements in the user shared memory buffer. The related **strncat** function limits the maximum number of characters that can be appended, and so is much safer in this regard.

If there is a literal text string in double quotes following the comma, this string will be appended onto the end of target variable, and a "null" character will automatically be appended after the last specified character. The text string can include standard text characters and the "escape sequences" listed under the **sprintf** function, but not the formatted variable sequences permitted in **sprintf**.

The returned value of the function after successful execution is the index in user shared memory of the null character that now ends the string in the target variable. If the function is unsuccessful, a value of -1 is returned. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Examples

```
EndIndex = strcat(256, 384)
EndIndex = strcat(256, 384+16*MsgNum)
nop(strcat(800, "due to oscillation"))
```

strchr

Function: Search for first instance of character in string variable

Syntax: **strchr(*{expression}*,*{expression}*)**

strchr(*{expression}*,'*{character}*')

The **strchr** function reports the presence and first location of a character in a string variable. The character can be specified by its ASCII-code value, or directly.

The first ***{expression}*** specifies the starting index in user shared memory of the string variable to be searched. The string will be searched from this index until the next "null" character

If there is a second ***{expression}*** following the comma, it specifies the ASCII value of the character to be searched for. Only the low byte of this value is used, so if the expression evaluates outside of the range 0 – 255, it will be "rolled over" into that range.

If there is a literal character in single quotes following the comma, this specifies the character that will be searched for in the string variable. Note that if the function specified this way in a buffered program is listed back, this will be reported as the ASCII code value for the character.

The returned value of the function after successful execution is the index in user shared memory of the first instance of the specified character. If the function is unsuccessful, not finding the specified character or otherwise, a value of -1 is returned.

Examples

```
CharIndex = strchr(256, 65)
CharIndex = strchr(256, 64+LtrNum)
nop(strchr(256, 'A'))
```

strcmp

Function: Compare two string variables

Syntax: **strcmp(*{expression}*,*{expression}*)**

strcmp(*{expression}*,"*{string}*")

The **strcmp** function compares the contents of a string variable to a text string. This second string can be another string variable, or it can be specified directly.

The first ***{expression}*** specifies the starting index in user shared memory of the first string variable to be compared. The string will be compared from this index until the next "null" character.

If there is a second ***{expression}*** following the comma, it specifies the starting index in user shared memory of the second string variable to be compared. The string will be compared from this index until the next "null" character.

If there is a literal text string in double quotes following the comma, this string will be compared to the value of the first string variable. The text string can include standard text characters and the "escape sequences" listed under the **sprintf** function, but not the formatted variable sequences permitted in **sprintf**.

If the two strings are exactly the same, the returned value of the function is 0. If they are not exactly the same, the returned value will be the "difference" of the two strings, the result of the "subtraction" of the second strings from the first, treated as integer numbers.

Examples

```
StrDiff = strcmp(256, 384)
StrDiff = strcmp(256, 512+32*CmdNum)
if (!(strcmp(HostCmdIndex, "Insert Part") { ...
```

strcpy

Function: Copy string variable

Syntax: **strcpy(*{expression}*,*{expression}*)**

strcpy(*{expression}*,"*{string}*")

The **strcpy** function copies a text string into a string variable. This string can be in another string variable, or it can be specified directly.

The first ***{expression}*** specifies the starting index in user shared memory of the "target" string variable into which the text string will be copied.

If there is a second ***{expression}*** following the comma, it specifies the starting index in user shared memory of the "source" string variable from which the text string will be copied. The copying will start from this index and continue until the "null" string terminator is found. Note that if there is not a "null" terminating character where expected, a very long sequence of bytes can be copied to the target string, overwriting other elements in the user shared memory buffer. The related **strncpy** function limits the maximum number of characters that can be copied, and so is much safer in this regard.

If there is a literal text string in double quotes following the comma, this string will be copied into the target variable, and a "null" character will automatically be appended after the last specified character. The text string can include standard text characters and the "escape sequences" listed under the **sprintf** function, but not the formatted variable sequences permitted in **sprintf**.

The returned value of the function after successful execution is the index in user shared memory of the null character that ends the string in the target variable. If the function is unsuccessful, a value of -1 is returned. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Examples

```
EndIndex = strcpy(256, 384)
EndIndex = strcpy(256, 384+16*MsgNum)
nop(strcpy(800, "Process timed out"))
```

strcspn

Function: Find longest substring without specified characters in string variable

Syntax: **strcspn(*{expression}*,*{expression}*)**

strcspn(*{expression}*,"*{string}*")

The **strcspn** function finds the length of the longest section of a string, beginning at its start, that consists only of characters *not* found in another string.

The first ***{expression}*** specifies the starting index in user shared memory of the string variable in which the length of the section is to be determined.

If there is a second ***{expression}*** following the comma, it specifies the starting index in user shared memory of the string variable that contains the characters whose absence is to be checked for in the first string variable.

If there is a literal text string in double quotes following the comma, this contains the characters whose absence is to be checked for in the first string variable. The text string can include standard text characters and the "escape sequences" listed under the **sprintf** function, but not the formatted variable sequences permitted in **sprintf**.

The returned value is the length of the substring in the first string variable, beginning at its start, that consists only of characters *not* found in the second string (variable or literal).

Examples

```
SubLen = strcspn(256, 384)
SubLen = strcspn(256, 512+32*CmdNum)
SubLen = strcspn(RecipeIndex, "XYZ")
```

strlen

Function: Find the number of characters in string variable

Syntax: **strlen(*{expression}*)**

The **strlen** function finds the length of a string variable.

The ***{expression}*** specifies the starting index in user shared memory of the string variable whose length is to be determined.

The returned value is the number of characters in the string variable, not including the "null" terminating character.

Examples

```
Len = strlen(256)
Len = strlen(RecipeIndex)
```

strncat

Function: Add limited text to end of string variable

Syntax: **strncat(*{expression}*,*{expression}*,*{expression}*)**

The **strncat** function appends a text string variable onto the end of another string variable.

The first ***{expression}*** specifies the starting index in user shared memory of the "target" string variable onto which the text string will be appended.

The second ***{expression}***, following the first comma, specifies the starting index in user shared memory of the "source" string variable from which the text string will be appended. The appending will start from this index and continue until the "null" string terminator is found, or until characters of the number specified in the final ***{expression}*** are appended, whichever comes first. If the function must limit the number of characters, it will automatically add a "null" string terminator to the target variable.

The returned value of the function after successful execution is the index in user shared memory of the null character that now ends the string in the target variable. If the function is unsuccessful, a value of -1 is returned. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Examples

```
EndIndex = strncat(256, 384, 8)
nop(strncat(256, 384+16*MsgNum, 12))
```

strncmp

Function: Compare limited lengths of two string variables

Syntax: **strncmp(*{expression}*,*{expression}*,*{expression}*)**

The **strncmp** function compares the contents of limited lengths of two string variables.

The first ***{expression}*** specifies the starting index in user shared memory of the first string variable to be compared. The string will be compared from this index until the next "null" character, or until the number of characters specified in the final ***{expression}*** are compared, whichever comes first.

The second ***{expression}***, following the first comma, specifies the starting index in user shared memory of the second string variable to be compared. The string will be compared from this index until the next "null" character, or until characters of the number specified in the final ***{expression}*** are compared, whichever comes first.

If the two (limited) strings are exactly the same, the returned value of the function is 0. If they are not exactly the same, the returned value will be the "difference" of the two strings, the result of the "subtraction" of the second string from the first, treated as integer numbers.

Examples

```
StrDiff = strncmp(256, 384, 8)
StrDiff = strncmp(256, 512+32*CmdNum, CmdCoreLen)
```

strncpy

Function: Copy limited string variable

Syntax: **strncpy(*{expression}*,*{expression}*,*{expression}*)**

The **strncpy** function copies a text string variable of limited length into another string variable.

The first ***{expression}*** specifies the starting index in user shared memory of the "target" string variable into which the text string will be copied.

The second ***{expression}***, following the first comma, specifies the starting index in user shared memory of the "source" string variable from which the text string will be copied. The copying will start from this index and continue until the "null" string terminator is found, or until characters of the number specified in the final ***{expression}*** are copied, whichever comes first. If the function must limit the number of characters, it will automatically add a "null" string terminator to the target variable.

The returned value of the function after successful execution is the index in user shared memory of the null character that ends the string in the target variable. If the function is unsuccessful, a value of -1 is returned. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Examples

```
EndIndex = strncpy(256, 384, 8)
nop(strncpy(256, 384+16*MsgNum, 12))
```

strpbrk

Function: Find first incidence of any of specified characters in string variable

Syntax: **strpbrk(*{expression}*,*{expression}*)**

strpbrk(*{expression}*,"*{string}*")

The **strpbrk** function finds the first incidence in a string variable of any of the characters contained in another string.

The first ***{expression}*** specifies the starting index in user shared memory of the string variable in which the incidence of a character is to be determined.

If there is a second ***{expression}*** following the comma, it specifies the starting index in user shared memory of the string variable that contains the characters whose presence is to be checked for in the first string variable.

If there is a literal text string in double quotes following the comma, this contains the characters whose presence is to be checked for in the first string variable. The text string can include standard text characters and the "escape sequences" listed under the **sprintf** function, but not the formatted variable sequences permitted in **sprintf**.

The returned value is the index in user shared memory of the first incidence in the first string variable of any of the characters in the second string (variable or literal).

Examples

```
CharIndex = strpbrk(256, 384)
CharIndex = strpbrk(256, 512+32*CmdNum)
CharIndex = strpbrk(RecipeIndex, "XYZ")
```

strrchr

Function: Search for last instance character in string variable

Syntax: **strrchr(*{expression}*,*{expression}*)**

strrchr(*{expression}*,'*{character}*')

The **strchr** function reports the presence and last location of a character in a string variable. The character can be specified by its ASCII-code value, or directly.

The first ***{expression}*** specifies the starting index in user shared memory of the string variable to be searched. The string will be searched from this index until the next "null" character

If there is a second ***{expression}*** following the comma, it specifies the ASCII value of the character to be searched for. Only the low byte of this value is used, so if the expression evaluates outside of the range 0 – 255, it will be "rolled over" into that range.

If there is a literal character in single quotes following the comma, this specifies the character that will be searched for in the string variable. Note that if the function specified this way in a buffered program is listed back, this will be reported as the ASCII code value for the character.

The returned value of the function after successful execution is the index in user shared memory of the last instance of the specified character. If the function is unsuccessful, not finding the specified character or otherwise, a value of -1 is returned.

Examples

```
CharIndex = strrchr(256, 65)
CharIndex = strrchr(256, 64+LtrNum)
nop(strrchr(256, 'A'))
```

strspn

Function: Find longest substring of specified characters in string variable

Syntax: **strspn(*{expression}*,*{expression}*)**

strspn(*{expression}*,"*{string}*")

The **strspn** function finds the length of the longest section of a string, beginning at its start, that consists only of characters found in another string.

The first ***{expression}*** specifies the starting index in user shared memory of the string variable in which the length of the section is to be determined.

If there is a second ***{expression}*** following the comma, it specifies the starting index in user shared memory of the string variable that contains the characters whose presence is to be checked for in the first string variable.

If there is a literal text string in double quotes following the comma, this contains the characters whose presence is to be checked for in the first string variable. The text string can include standard text characters and the "escape sequences" listed under the **sprintf** function, but not the formatted variable sequences permitted in **sprintf**.

The returned value is the length of the substring in the first string variable, beginning at its start, that consists only of characters found in the second string (variable or literal).

Examples

```
SubLen = strspn(256, 384)
SubLen = strspn(256, 512+32*CmdNum)
SubLen = strspn(RecipeIndex, "XYZ")
```

strstr

Function: Find first incidence of another string in string variable

Syntax: **strstr(*{expression}*,*{expression}*)**

strstr(*{expression}*,"*{string}*")

The **strstr** function finds the first incidence in a string variable of another string.

The first ***{expression}*** specifies the starting index in user shared memory of the string variable in which the incidence of another string is to be determined.

If there is a second ***{expression}*** following the comma, it specifies the starting index in user shared memory of the string variable whose contents are to be checked for in the first string variable.

If there is a literal text string in double quotes following the comma, this is the string whose presence is to be checked for in the first string variable. The text string can include standard text characters and the "escape sequences" listed under the **sprintf** function, but not the formatted variable sequences permitted in **sprintf**.

The returned value is the index in user shared memory of the first incidence in the first string variable of any of the second string (variable or literal).

Examples

```
CharIndex = strstr(256, 384)
CharIndex = strstr(256, 512+32*CmdNum)
CharIndex = strstr(RecipeIndex, "XYZ")
```

strtod

Function: Convert string text to numeric variable value

Syntax: **strtod(*{expression}*,*{expression}*)**

The **strtod** function converts a text string representing a decimal numeric value into that numeric value, returning that value.

The first ***{expression}*** specifies the starting index in user shared memory of the "source" string variable from which the numeric value will be derived. Power PMAC will use all the characters from this index until a character that cannot be interpreted as part of a numeric value, or a "null" terminating character is found. The text for the numeric value can take any of the forms that are valid for specifying numeric constant values in commands to PMAC. This includes scientific notation (e.g. `1.234e8`) and hexadecimal format (e.g. `$3FFF`).

The second ***{expression}***, following the first comma, specifies the number of the local (L) variable for the program or command thread into which the index in user shared memory of the null character that ends the string variable is found. If the function is not successful in interpreting any numeric value from the specified string, this variable will have a value of -1.

The returned value is the interpreted numeric value derived from the specified string. It is calculated as a double-precision floating-point value, so it can be used directly in any subsequent program calculations without further conversion, including storage to any of the general-purpose user variables, local or global. If it is stored to a variable or element of different format, Power PMAC will automatically perform the necessary type conversion. If the function is not successful in interpreting any numeric value from the specified string, the returned value will be 0 (but the success should be judged based on the value in the specified local variable).

Examples

```
MyVar = strtod(256, 0)
InchLength = 25.4 * strtod(500, EndPtr)
```

strtolower

Function: Copy limited string variable, converting upper case to lower case

Syntax: **strtolower(*{expression}*,*{expression}*,*{expression}*)**

The **strtolower** function copies a string variable of specified length into another string variable, converting any upper-case letters (A .. Z) to the equivalent lower-case letters (a .. z).

The first ***{expression}*** specifies the starting index in user shared memory of the "target" string variable into which the text string will be copied.

The second ***{expression}***, following the first comma, specifies the starting index in user shared memory of the "source" string variable from which the text string will be copied. The copying will start from this index and continue until the "null" string terminator is found, or until characters of the number specified in the final ***{expression}*** are copied, whichever comes first. If the function must limit the number of characters, it will automatically add a "null" string terminator to the target variable.

Note that it is possible that the "source" and "target" string variables be the same, in which case the function has the effect of converting the string variable.

The returned value of the function after successful execution is the index in user shared memory of the null character that ends the string in the target variable. If the function is unsuccessful, a value of -1 is returned.

Examples

```
EndIndex = strtolower(256, 384, 8)
EndIndex = strtolower(256, 256, 8)
nop(strtolower(256, 384+16*MsgNum, 12))
```

strtoupper

Function: Copy limited string variable, converting lower case to upper case

Syntax: **strtoupper(*{expression}*,*{expression}*,*{expression}*)**

The **strtoupper** function copies a string variable of specified length into another string variable, converting any lower-case letters (a .. z) to the equivalent upper-case letters (A .. Z).

The first ***{expression}*** specifies the starting index in user shared memory of the "target" string variable into which the text string will be copied.

The second ***{expression}***, following the first comma, specifies the starting index in user shared memory of the "source" string variable from which the text string will be copied. The copying will start from this index and continue until the "null" string terminator is found, or until characters of the number specified in the final ***{expression}*** are copied, whichever comes

first. If the function must limit the number of characters, it will automatically add a "null" string terminator to the target variable.

Note that it is possible that the "source" and "target" string variables be the same, in which case the function has the effect of converting the string variable.

The returned value of the function after successful execution is the index in user shared memory of the null character that ends the string in the target variable. If the function is unsuccessful, a value of -1 is returned.

Examples

```
EndIndex = strtoupper(256, 384, 8)
EndIndex = strtoupper(256, 256, 8)
nop(strtoupper(256, 384+16*MsgNum, 12))
```

Character Buffer Functions

Power PMAC has two functions for writing into character arrays in the user shared memory buffer. These access the character bytes directly, without concern as to how they might be arranged into string variables.

Note

These character buffer functions return a value that is not the main result of the function. In firmware versions before V2.0, these returned values had to be assigned to a variable. However, starting in V2.0, released 1st quarter 2015, if used in the expression for **nop** program statement, no such assignment needs to be made.

memcpy

Function: Copy character buffer memory contents

Syntax: **memcpy(*{expression}*,*{expression}*,*{expression}*)**

The **memcpy** function copies one or more consecutive byte locations in the user shared memory buffer from another set of locations.

The first ***{expression}*** specifies the (starting) index in user shared memory of the register(s) to be copied to. This index is the byte address offset from the start of the user buffer.

The second ***{expression}***, following the first comma, specifies the (starting) index in user shared memory of the register(s) to be copied from. This index is the byte address offset from the start of the user buffer.

The final ***{expression}*** specifies the number of bytes of memory to be copied.

The returned value of the function after successful execution is the index in user shared memory of the first register to have a value copied into it. If the function is unsuccessful, a value of -1 is returned. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Examples

```
CharIndex = memcpy(256, 512, 8)
CharIndex = memcpy(4096, 6144, 256)
nop(memcpy(3000, DestIndex, Length+1))
```

memset

Function: Set character buffer memory to specified character value

Syntax: **memset(*{expression}*,*{expression}*,*{expression}*)**

memset(*{expression}*,'*{character}*',*{expression}*)

The **memset** function forces one or more consecutive byte locations in the user shared memory buffer to a character value.

The first ***{expression}*** specifies the (starting) index in user shared memory of the register(s) to be forced. This index is the byte address offset from the start of the user buffer.

If there is a second ***{expression}*** following the first comma, it specifies the ASCII value of the character to be copied into memory. If this value is in the range 0 – 255, it represents a single character (byte), and the copying will be done a byte at a time. If the value is outside this range it represents a 32-bit word, and the copying will be done in 4-byte pieces.

If there is a literal character in single quotes following the first comma, this specifies the character (byte) that will be copied into memory, and the copying will be done a byte at a time. Note that if the function specified this way in a buffered program is listed back, this will be reported as the ASCII code value for the character.

The final ***{expression}*** specifies the number of bytes of string buffer memory to be forced with values. Note that this is expressed in bytes whether the copying is done in single-byte or 4-byte pieces.

The returned value of the function after successful execution is the index in user shared memory of the first register to have a value copied into it. If the function is unsuccessful, a value of -1 is returned. Even if not checked, it must be either be "put" somewhere, typically assigned to a dummy variable, or the function must be embedded within a program **nop** ("no-operation") statement. It can be useful for assessing the validity of the operation.

Examples

```
CharIndex = memset(256, 65, 1)
CharIndex = memset(256, 64+LtrNum, 8)
nop(memset(256, 'A', 1))
```

EtherCAT Network Functions

The following function can be used to access data on the EtherCAT network.

ecatCompleteSDO

Function: Complete Access for EtherCAT Service Dictionary Object (SDO)

Syntax: **ecatCompleteSDO**
**(*{expression},{expression},{expression},{expression},*
***{expression}*)**

The **ecatcompletesdo** function permits the user to access (write to or read from) the specified EtherCAT Service Dictionary Object (SDO) from within a program.

This function can only be used if **Sys.EcatType** is set to 0 (Etherlab stack).

The function takes 6 arguments.

- The first argument is the master index value "***i***" (i.e. the EtherCAT network index) of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The second argument is the slave index value "***j***" of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The third argument is the index number of the CompleteSDO.
- The fourth argument is the index in user byte data from which the data is transferred over EtherCAT from.
- The fifth argument is the data size of the requested object in bytes. It is the user's responsibility to know the size of the dictionary object they are requesting.
  Bits 16 (Value $10000, or 65536) controls read/write mode, "0" in this bit means write mode, "1" means read mode.

The return value of the function in the case of a valid access is 0. In the case of an invalid access is "not-a-number" (nan). Even if the returned value is not really used in the application, it still must be stored somewhere; for example, by assigning it to a "dummy" variable as shown in the example code below.

Example

```
global sdoresult;
sys.cdata[78] = $1
sys.cdata[79] = $0
sys.cdata[80] = $4
sys.cdata[81] = $17
sdoresult = ecatCompleteSDO(0,0,$1C12,78,4);         // Write to CAN object 4 bytes
sdoresult = ecatCompleteSDO(0,0,$1C12,82,$10004);    // Read from CAN object 4 bytes
```

ecatRegReadWrite

Function: Access an EtherCAT Regster

Syntax: **ecatregreadwrite**
**(*{expression},{expression},{expression},{expression},*
{expression},{expression})**

The **ecatregreadwrite** function permits the user to access (write to or read from) the specified EtherCAT Register from within a program. The function takes 6 arguments.

- The first argument is the master index value "***i***" (i.e. the EtherCAT network index) of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The second argument is the slave index value "***j***" of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The third argument is the "direction" of the access. It should be 0 to write to EtherCAT register, or 1 to read from EtherCAT register.
- The fourth argument is EtherCAT register.
- The fifth argument is the data value for EtherCAT register. If the function is used to write to EtherCAT register (fourth argument = 0), this is the value that will be written, If the function is used to read from EtherCAT register (fourth argument = 1), this value is not used (but a value must be present here, even if it is unused, in order for the function to execute).
- The sixth argument is the data size of the requested object in bytes. It is the users responsibility to know the size of the dictionary object they are requesting.

The return value of the function in the case of a valid "read" access is the data value of the specified EtherCAT Register. The return value in the case of an invalid "read" access is "not-a-number" (nan).

The return value of the function in the case of a valid "write" access is 0. The return value in the case of an invalid "write" access is "not-a number" (nan). Even if the returned value is not really used in the application, it still must be stored somewhere; for example, by assigning it to a "dummy" variable as shown in the example code below.

Example

```
global RegRWResult;
RegRWResult = ecatregreadwrite(0,0,0,$120,18,2);     // Write to EtherCAT Register 1 byte
RegRWResult = ecatregreadwrite(0,0,1,$134,0,2);      // Read from EtherCAT Register 4
bytes
```

ecatsdo

Function: Access an EtherCAT Service Dictionary Object (SDO)

Syntax: **ecatsdo ({*expression*},{*expression*},{*expression*},{*expression*},{*expression*},{*expression*})**

The **ecatsdo** function permits the user to access (write to or read from) the specified EtherCAT Service Dictionary Object (SDO) from within a program.

This function can only be used if **Sys.EcatType** is set to 0 (Etherlab stack). If **Sys.EcatType** is 1 (Acontis stack), use the similar **ecattypedsdo** function.

This function takes 6 arguments.

- The first argument is the "direction" of the access. It should be 0 to write to the SDO, or 1 to read from the SDO.
- The second argument is the slave index value "***j***" of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The third argument is the index number of the SDO.
- The fourth argument is the subindex number of the SDO.
- The fifth argument is the data value for the SDO. If the function is used to write to the SDO (first argument = 0), this is the value that will be written, If the function is used to read from the SDO (first argument = 1), this value is not used (but a value must be present here, even if it is unused, in order for the function to execute).
- The sixth argument is the master index value "***i***" (i.e. the EtherCAT network index) of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.

The return value of the function in the case of a valid "read" access is the data value of the specified SDO. The return value in the case of an invalid "read" access is "not-a-number" (nan).

The return value of the function in the case of a valid "write" access is 0. The return value in the case of an invalid "write" access is "not-a number" (nan). Even if the returned value is not really used in the application, it still must be stored somewhere; for example, by assigning it to a "dummy" variable as shown in the example code below.

Example

```
global sdoresult;
sdoresult = ecatsdo(0,0,$6060,0,10,0);        // Write to CAN object
sdoresult = ecatsdo(1,0,$6061,0,0,0);         // Read from CAN object
```

ecatSetSlaveStateMachine

Function: Set slave state

Syntax: **ecatsetslavestatemachine**
(*{expression},{expression},{expression},{expression}*)

The **ecatsetslavestatemachine** function permits the user to set slave state from within a program. The function takes 4 arguments.

- The first argument is the master index value "***i***" (i.e. the EtherCAT network index) of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The second argument is the slave index value "***j***" of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The third argument is EtherCAT slave mode:
  1: INIT
  2: PreOP
  4: SafeOP
  8: OP
- The fourth argument is timeout [msec] value. Suggested value is 2000msec.

Example

```
global SlaveStateResult;
SlaveStateResult = ecatsetslavestatemachine(0,4,2,2000);    // Set ECAT[0].Slave[4] to
PreOP mode
```

ecattypedsdo

Function: Access an EtherCAT Service Dictionary Object (SDO)

Syntax: **ecattypedsdo**
(*{expression}*,*{expression}*,*{expression}*,*{expression}*,
***{expression}*,*{expression}*,*{expression}*)**

The **ecattypedsdo** function permits the user to access (write to or read from) the specified EtherCAT Service Dictionary Object (SDO) from within a program.

This function takes 6 arguments.

- The first argument is the master index value "***i***" (i.e. the EtherCAT network index) of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The second argument is the slave index value "***j***" of the device, corresponding to the structure **ECAT[*i*].Slave[*j*]** assigned to the device.
- The third argument is the "direction" of the access. It should be 0 to write to the SDO, or 1 to read from the SDO.
- The fourth argument is the index number of the SDO.
- The fifth argument is the subindex number of the SDO.
- The sixth argument is the data value for the SDO. If the function is used to write to the SDO (first argument = 0), this is the value that will be written, If the function is used to read from the SDO (first argument = 1), this value is not used (but a value must be present here, even if it is unused, in order for the function to execute).
- The seventh argument is the data size of the requested object in bytes. It is the users responsibility to know the size of the dictionary object they are requesting.

The return value of the function in the case of a valid "read" access is the data value of the specified SDO. The return value in the case of an invalid "read" access is "not-a-number" (nan).

The return value of the function in the case of a valid "write" access is 0. The return value in the case of an invalid "write" access is "not-a number" (nan). Even if the returned value is not really used in the application, it still must be stored somewhere; for example, by assigning it to a "dummy" variable as shown in the example code below.

Example

```
global sdoresult;
sdoresult = ecattypedsdo(0,0,0,$6060,0,10,1);        // Write to CAN object 1 byte
sdoresult = ecattypedsdo(0,0,1,$6064,0,0,4);         // Read from CAN object 4 bytes
```

POWER PMAC I/O ADDRESS OFFSETS

This section provides a reference for the I/O address offsets of interface ICs used by the Power PMAC to communicate with real-world I/O. Largely, these are for reference only, as most users do not need to know the numerical addresses of the registers in these ICs; instead they will access the registers through pre-defined data structure elements.

The address offsets given in this section are referenced to the I/O base address. This I/O base address can differ between different versions of Power PMAC hardware and software. The absolute value of the I/O base address can be found in the **Sys.piom** data structure element.

Since most of the products with these interface ICs can be used with the older Turbo PMAC CPU as well, and many users wish to upgrade from the Turbo PMAC CPU to the Power PMAC CPU, Turbo PMAC addresses and settings are provided for reference where appropriate.

Overview by Chip Select Line

The Power PMAC CPU addresses different interface ICs through the use of "chip-select" lines and addressing lines. Different classes of interface ICs use different chip-select lines. The following table lists the chip-select lines that can be used in a Power PMAC system, the function of each line, and the base address offset (from **Sys.piom**) for that chip select.

Chip Select Line	Function	Sys.BusCtrl[*n*] Index	Power PMAC I/O Base Address Offset	Turbo PMAC (Absolute) Base Address
CS00-*	On-board I/O	0	$000000	$078800
CS02-*	On-board I/O	1	$100000	$078900
CS04-*	On-board I/O	2	$200000	$078A00
CS06-*	On-board I/O	3	$300000	$078B00
CS0-*	On-board Servo ICs	4	$400000	$078000
CS1-*	On-board Servo ICs	5	$500000	$078100
CS2-	Expansion Servo ICs	6	$600000	$078200
CS3-	Expansion Servo ICs	7	$700000	$078300
CS4-	MACRO ICs	8	$800000	$078400
CS5-	DSPGATE3 ICs	9	$900000	--
CS10-	Expansion I/O	10	$A00000	$078C00
CS12-	Expansion I/O	11	$B00000	$078D00
CS14-	Expansion I/O	12	$C00000	$078E00
CS16-	Expansion I/O	13	$D00000	$078F00
DPRCS- (MemCS0-)	Shared memory	14	$E00000	$060000
VMECS- (MemCS1-)	VME interface or Shared memory	15	$F00000	$070000

* Not presently implemented in any Power PMAC hardware

UMAC Addresses for PMAC2-Style Servo Cards

Power PMAC UMAC systems can accept several types of accessory boards that use the PMAC2-style "DSPGATE1" Servo IC. These include the ACC-24E2, ACC-24E2A, ACC-24E2S, and ACC-51E. Each board in this class must have a unique DIP-switch setting in a single UMAC rack; otherwise addressing conflicts will occur. The following table shows the possible switch settings, the resulting address offsets, and the **Gate1[*i*]** index values "***i***".

SW1-1 (CS2/3)	SW1-2 (Address bit 8)	SW1-3 (Address bit 15)	SW1-4 (Address bit 16)	Power PMAC "Gate1" Index #	Power PMAC I/O Base Address Offset	Turbo PMAC "Servo IC" #	Turbo PMAC Base Address
ON (0)	ON(0)	ON(0)	ON(0)	4	$600000	2	$078200
ON (0)	OFF(1)	ON(0)	ON(0)	5	$600100	2*	$078220
OFF(1)	ON(0)	ON(0)	ON(0)	6	$700000	3	$078300
OFF(1)	OFF(1)	ON(0)	ON(0)	7	$700100	3*	$078320
ON (0)	ON(0)	OFF(1)	ON(0)	8	$608000	4	$079200
ON (0)	OFF(1)	OFF(1)	ON(0)	9	$608100	4*	$079220
OFF(1)	ON(0)	OFF(1)	ON(0)	10	$708000	5	$079300
OFF(1)	OFF(1)	OFF(1)	ON(0)	11	$708100	5*	$079320
ON (0)	ON(0)	ON(0)	OFF(1)	12	$610000	6	$07A200
ON (0)	OFF(1)	ON(0)	OFF(1)	13	$610100	6*	$07A220
OFF(1)	ON(0)	ON(0)	OFF(1)	14	$710000	7	$07A300
OFF(1)	OFF(1)	ON(0)	OFF(1)	15	$710100	7*	$07A320
ON (0)	ON(0)	OFF(1)	OFF(1)	16	$618000	8	$07B200
ON (0)	OFF(1)	OFF(1)	OFF(1)	17	$618100	8*	$07B220
OFF(1)	ON(0)	OFF(1)	OFF(1)	18	$718000	9	$07B300
OFF(1)	OFF(1)	OFF(1)	OFF(1)	19	$718100	9*	$07B320

The conversion of a Turbo PMAC address ("*TurboAdr*") to a Power PMAC address offset ("*PowerAdr*") can be made using the following equation:

*PowerAdr = $600000 + (TurboAdr & $100) * $1000 + (TurboAdr & $7CF8) * 8 + (TurboAdr & 7) * 4 + "X" * $20*

where "X" is 1 for a Turbo PMAC X address, and 0 for a Turbo PMAC Y address.

UMAC Addresses for PMAC2-Style MACRO Cards

Power PMAC UMAC systems can accept one or more of the PMAC2-style ACC-5E MACRO interface accessory boards. Each ACC-5E board must have a unique DIP-switch setting in a single UMAC rack; otherwise addressing conflicts will occur. In addition, if an ACC-5E card has a second IC (automatically at the next higher address and index than the first), the user must ensure that the first IC on any other ACC-5E does not have an address conflict with this IC. The following table shows the possible switch settings, the resulting address offsets, and the **Gate2[*i*]**

index values "***i***"*.

SW1-1 (Address bit 15)	SW1-2 (Address bit 16)	SW1-3 (Address bit 11)	SW1-4 (Address bit 12)	Power PMAC "Gate2" Index #*	Power PMAC I/O Base Address Offset	Turbo PMAC MACRO IC #	Turbo PMAC Base Address
ON (0)	ON(0)	ON(0)	ON(0)	0	$800000	0	$078400
OFF(1)	ON(0)	ON(0)	ON(0)	1	$808000	1	$079400
ON (0)	OFF(1)	ON(0)	ON(0)	2	$810000	2	$07A400
OFF(1)	OFF(1)	ON(0)	ON(0)	3	$818000	3	$07B400
ON (0)	ON(0)	OFF(1)	ON(0)	4	$800800	4	$078500
OFF(1)	ON(0)	OFF(1)	ON(0)	5	$808800	5	$079500
ON (0)	OFF(1)	OFF(1)	ON(0)	6	$810800	6	$07A500
OFF(1)	OFF(1)	OFF(1)	ON(0)	7	$818800	7	$07B500
ON (0)	ON(0)	ON(0)	OFF(1)	8	$801000	8	$078600
OFF(1)	ON(0)	ON(0)	OFF(1)	9	$809000	9	$079600
ON (0)	OFF(1)	ON(0)	OFF(1)	10	$811000	10	$07A600
OFF(1)	OFF(1)	ON(0)	OFF(1)	11	$819000	11	$07B600
ON (0)	ON(0)	OFF(1)	OFF(1)	12	$801800	12	$078700
OFF(1)	ON(0)	OFF(1)	OFF(1)	13	$809800	13	$079700
ON (0)	OFF(1)	OFF(1)	OFF(1)	14	$811800	14	$07A700
OFF(1)	OFF(1)	OFF(1)	OFF(1)	15	$819800	15	$07B700

The conversion of a Turbo PMAC address ("*TurboAdr*") to a Power PMAC address offset ("*PowerAdr*") can be made using the following equation:

*PowerAdr = $800000 + (TurboAdr & $7BF8) * 8 + (TurboAdr & 7) * 4 + "X" * $20*

where "*X*" is 1 for a Turbo PMAC X address, and 0 for a Turbo PMAC Y address.

*The index numbers "*i*" given for the **Gate2[*i*]** data structures are strictly only valid for the case where the system is configured with the MACRO ICs located at the lowest possible address offset settings. (This configuration is strongly recommended.) In operation, the Power PMAC CPU auto-detects at power-up/reset all of the boards with these PMAC2-style MACRO cards and assigns the index number "0" to the IC with the lowest address offset, "1" to the next lowest, and so on.

The first MACRO IC on an ACC-5E is always located at the address offset shown in the table based on the card's DIP-switch setting. If the ACC-5E has a second MACRO IC installed, this second IC is always located at the next higher address offset, and is automatically assigned the next higher index number.

UMAC Addresses for PMAC2-Style I/O Cards

Power PMAC UMAC systems can accept several types of accessory boards that use the PMAC2-style "IOGATE" I/O IC. These include the ACC-11E*, ACC-14E, ACC-65E, ACC-66E, ACC-67E, and ACC-68E. Each board in this class must have a unique DIP-switch setting in a single UMAC rack; otherwise addressing conflicts will occur. The following table shows the possible switch settings, the resulting address offsets, and the **GateIo[*i*]** index values "***i***". (Note that the ACC-11E cannot be accessed through the **GateIo[*i*]** data structure; users should access its registers through M-variable pointers.)

ACC-28E high-resolution A/D converter boards also use this same address range; their registers can be accessed through the **Acc28E[*i*]** data structure. Other accessory boards, particularly for serial encoder interfaces, use this same address range as well, but do not have pre-defined data structures; users should access their registers through M-variable pointers.

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	Turbo PMAC "I/O IC" #	Turbo PMAC Base Address
ON (0)	ON(0)	ON(0)	ON(0)	0	$A00000	0	Y:$078C00
OFF(1)	ON(0)	ON(0)	ON(0)	1	$B00000	1	Y:$078D00
ON (0)	OFF(1)	ON(0)	ON(0)	2	$C00000	2	Y:$078E00
OFF(1)	OFF(1)	ON(0)	ON(0)	3	$D00000	3	Y:$078F00
ON (0)	ON(0)	OFF(1)	ON(0)	4	$A08000	4	Y:$079C00
OFF(1)	ON(0)	OFF(1)	ON(0)	5	$B08000	5	Y:$079D00
ON (0)	OFF(1)	OFF(1)	ON(0)	6	$C08000	6	Y:$079E00
OFF(1)	OFF(1)	OFF(1)	ON(0)	7	$D08000	7	Y:$079F00
ON (0)	ON(0)	ON(0)	OFF(1)	8	$A10000	8	Y:$07AC00
OFF(1)	ON(0)	ON(0)	OFF(1)	9	$B10000	9	Y:$07AD00
ON (0)	OFF(1)	ON(0)	OFF(1)	10	$C10000	10	Y:$07AE00
OFF(1)	OFF(1)	ON(0)	OFF(1)	11	$D10000	11	Y:$07AF00
ON (0)	ON(0)	OFF(1)	OFF(1)	12	$A18000	12	Y:$07BC00
OFF(1)	ON(0)	OFF(1)	OFF(1)	13	$B18000	13	Y:$07BD00
ON (0)	OFF(1)	OFF(1)	OFF(1)	14	$C18000	14	Y:$07BE00
OFF(1)	OFF(1)	OFF(1)	OFF(1)	15	$D18000	15	Y:$07BF00

The conversion of a Turbo PMAC address ("*TurboAdr*") to a Power PMAC address offset ("*PowerAdr*") can be made using the following equation:

$$PowerAdr = \$A00000 + (TurboAdr \,\&\, \$300) * \$1000 + (TurboAdr \,\&\, \$70F8) * 8 + (TurboAdr \,\&\, 7) * 4$$

*The older ACC-11E has the same rules for base address offset as a function of DIP-switch settings as the newer cards, but it does not have the circuitry necessary for auto-detection by the Power PMAC CPU, so it cannot use the **GateIo[*i*]** data structures. Its registers must be accessed using M-variables based on the address offsets of the registers.

UMAC Addresses for PMAC2-Style Shared Memory Cards

Power PMAC UMAC systems can accept accessory boards that provide a shared-memory interface between the processor and an external communications channel. The most important of these is the ACC-72E Fieldbus Interface Board. Each board in this class must have a unique DIP-switch setting in a single UMAC rack; otherwise addressing conflicts will occur. The following table shows the possible switch settings, and the resulting address offsets.

SW1-1	SW1-2 (MEMC S0 / 1)	SW1-3 (Address bit 15)	SW1-4 (Address bit 16)	Shared-Memory Board #	Power PMAC I/O Base Address Offset	Turbo PMAC Base Address
Don't care	ON(0)	ON(0)	ON(0)	0	$E00000	$06C000
Don't care	OFF(1)	ON(0)	ON(0)	1	$F00000	$074000
Don't care	ON(0)	OFF(1)	ON(0)	2	$E08000	$06D000
Don't care	OFF(1)	OFF(1)	ON(0)	3	$F08000	$075000
Don't care	ON(0)	ON(0)	OFF(1)	4	$E10000	$06E000
Don't care	OFF(1)	ON(0)	OFF(1)	5	$F10000	$076000
Don't care	ON(0)	OFF(1)	OFF(1)	6	$E18000	$06F000
Don't care	OFF(1)	OFF(1)	OFF(1)	7	$F18000	$077000

The conversion of a Turbo PMAC address ("*TurboAdr*") to a Power PMAC address offset ("*PowerAdr*") can be made using the following equation:

$$PowerAdr = \$E00000 + (TurboAdr \,\&\, \$10000) * \$10 + (TurboAdr \,\&\, \$3FF8) * 8 + (TurboAdr \,\&\, 7) * 4 + "X" * \$20$$

where "*X*" is 1 for a Turbo PMAC X address, and 0 for a Turbo PMAC Y address.

UMAC Addresses for PMAC3-Style Cards

Power PMAC UMAC systems can accept several types of accessory boards that use the PMAC3-style "DSPGATE3" Machine-Interface IC. These include the ACC-24E3, ACC-5E3, and ACC-59E3. Each board in this class must have a unique DIP-switch setting in a single UMAC rack; otherwise addressing conflicts will occur. The following table shows the possible switch settings, the resulting address offsets, and the **Gate3[*i*]** index values "***i***".

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 that Turbo PMAC CPUs cannot address boards with DSPGATE3 ICs.

UMAC Addressing Summary for Turbo and Power PMAC

The following table shows the addressing of all possible types of UMAC accessory cards when used with both the Turbo PMAC CPU and the Power PMAC CPU. The base addresses for both the functional part of the card, and for the "identification" IC are given for both CPU types.

Main CS #	Turbo IC #	Turbo Card Base Adr	Turbo ID Base Adr	Power IC Type & #	Power Card Base Adr Ofs	Power ID Base Adr Ofs	Power Cid #
	{reserved}			*{reserved}*			0
	{reserved}			*{reserved}*			1
CS2	Servo 2	$078200	$078F08	Gate1[4]	$600000	$D00040	2
CS3	Servo 3	$078300	$078F0C	Gate1[6]	$700000	$D00050	3
CS4	Macro 0	$078400	$078F10	Gate2[0]~	$800000	$D00080	4
CS4	Macro 4	$078500	$078F14	Gate2[4]~	$800800	$D00090	5
CS4	Macro 8	$078600	$078F18	Gate2[8]~	$801000	$D000C0	6
CS4	Macro 12	$078700	$078F1C	Gate2[12]~	$801800	$D000D0	7
MemCS0	Mem 0	$06C000	$078F20	Dpr[0]	$E00000	$D00100	8
MemCS1	Mem 1	$074000	$078F24	Dpr[1]	$F00000	$D00110	9
CS2	Servo 2*	$078220	$078F28	Gate1[5]	$600100	$D00140	10
CS3	Servo 3*	$078320	$078F2C	Gate1[7]	$700100	$D00150	11
CS10	I/O 0	$078C00	$078F30	GateIo[0]	$A00000	$D00180	12
CS12	I/O 1	$078D00	$078F34	GateIo[1]	$B00000	$D00190	13
CS14	I/O 2	$078E00	$078F38	GateIo[2]	$C00000	$D001C0	14
CS16	I/O 3	$078F00	$078F3C	GateIo[3]	$D00000	$D001D0	15
	{reserved}			*{reserved}*			16
	{reserved}			*{reserved}*			17
CS2	Servo 4	$079200	$079F08	Gate1[8]	$608000	$D08040	18
CS3	Servo 5	$079300	$079F0C	Gate1[10]	$708000	$D08050	19
CS4	Macro 1	$079400	$079F10	Gate2[1]~	$808000	$D08080	20
CS4	Macro 5	$079500	$079F14	Gate2[5]~	$808800	$D08090	21
CS4	Macro 9	$079600	$079F18	Gate2[9]~	$809000	$D080C0	22
CS4	Macro 13	$079700	$079F1C	Gate2[13]~	$809800	$D080D0	23
MemCS0	Mem 2	$06D000	$079F20	Dpr[2]	$E08000	$D08100	24
MemCS1	Mem 3	$075000	$079F24	Dpr[3]	$F08000	$D08110	25
CS2	Servo 4*	$079220	$079F28	Gate1[9]	$608100	$D08140	26
CS3	Servo 5*	$079320	$079F2C	Gate1[11]	$708100	$D08150	27
CS10	I/O 4	$079C00	$079F30	GateIo[4]	$A08000	$D08180	28
CS12	I/O 5	$079D00	$079F34	GateIo[5]	$B08000	$D08190	29
CS14	I/O 6	$079E00	$079F38	GateIo[6]	$C08000	$D081C0	30
CS16	I/O 7	$079F00	$079F3C	GateIo[7]	$D08000	$D081D0	31
	{reserved}			*{reserved}*			32
	{reserved}			*{reserved}*			33
CS2	Servo 6	$07A200	$07AF08	Gate1[12]	$610000	$D10040	34
CS3	Servo 7	$07A300	$07AF0C	Gate1[14]	$710000	$D10050	35
CS4	Macro 2	$07A400	$07AF10	Gate2[2]~	$810000	$D10080	36
CS4	Macro 6	$07A500	$07AF14	Gate2[6]~	$810800	$D10090	37
CS4	Macro 10	$07A600	$07AF18	Gate2[10]~	$811000	$D100C0	38
CS4	Macro 14	$07A700	$07AF1C	Gate2[14]~	$811800	$D100D0	39
MemCS0	Mem 4	$06E000	$07AF20	Dpr[4]	$E10000	$D10100	40
MemCS1	Mem 5	$076000	$07AF24	Dpr[5]	$F10000	$D10110	41
CS2	Servo 6*	$07A220	$07AF28	Gate1[13]	$610100	$D10140	42
CS3	Servo 7*	$07A320	$07AF2C	Gate1[15]	$710100	$D10150	43

CS10	I/O 8	$07AC00	$07AF30	GateIo[8]	$A10000	$D10180	44
CS12	I/O 9	$07AD00	$07AF34	GateIo[9]	$B10000	$D10190	45
CS14	I/O 10	$07AE00	$07AF38	GateIo[10]	$C10000	$D101C0	46
CS16	I/O 11	$07AF00	$07AF3C	GateIo[11]	$D10000	$D101D0	47
	{reserved}			*{reserved}*			48
	{reserved}			*{reserved}*			49
CS2	Servo 8	$07B200	$07BF08	Gate1[16]	$618000	$D18040	50
CS3	Servo 9	$07B300	$07BF0C	Gate1[18]	$718000	$D18050	51
CS4	Macro 3	$07B400	$07BF10	Gate2[3]~	$818000	$D18080	52
CS4	Macro 7	$07B500	$07BF14	Gate2[7]~	$818800	$D18090	53
CS4	Macro 11	$07B600	$07BF18	Gate2[11]~	$819000	$D180C0	54
CS4	Macro 15	$07B700	$07BF1C	Gate2[15]~	$819800	$D180D0	55
MemCS0	Mem 6	$06F000	$07BF20	Dpr[6]	$E18000	$D18100	56
MemCS1	Mem 7	$077000	$07BF24	Dpr[7]	$F18000	$D18110	57
CS2	Servo 8*	$07B220	$07BF28	Gate1[17]	$618100	$D18140	58
CS3	Servo 9*	$07B320	$07BF2C	Gate1[19]	$718100	$D18150	59
CS10	I/O 12	$07BC00	$07BF30	GateIo[12]	$A18000	$D18180	60
CS12	I/O 13	$07BD00	$07BF34	GateIo[13]	$B18000	$D18190	61
CS14	I/O 14	$07BE00	$07BF38	GateIo[14]	$C18000	$D181C0	62
CS16	I/O 15	$07BF00	$07BF3C	GateIo[15]	$D18000	$D181D0	63

~: These **Gate2[*i*]** index numbers are only valid for these addresses if all ACC-5E cards in the system are set for the lowest possible addresses that do not create conflict.

Power PMAC ASIC Register Element Addresses

This section documents the data structure elements for registers and parts of registers in the different types of ASICs that can be present in a Power PMAC system, and their addresses. Some of the elements are saved setup elements (shown in bold) that are documented in detail in that chapter of the manual. The addresses are primarily for reference, as most uses of the elements and registers do not require the user to know the numerical address of the register.

To calculate the address offset of a register element from the I/O base address:

1. Add the offset of the base address of the IC in question from the I/O base address, based on the IC type and index number. This offset value is found in the boxed table at the top of the section for each IC type.
2. If a channel-specific register, add the offset of the channel's base address from the IC's base address.
3. Add the offset of the register in question from the IC's base address, or for a channel-specific register, from the channel's base address.

This value can be used to declare "io" format M-variables.

To find the absolute numerical address of the register element in the Power PMAC memory map:

1. Find the starting address for I/O by querying the value of **Sys.piom**.
2. Add this to the address offset calculated above

This value will match the numerical value reported when the address of the element is queried.

Notes:

1. Bit numbers are given in "Intel" (little-endian) style, where the value of Bit *n* in the word is 2^n. All bit numbers are for a full 32-bit word, even if the hardware uses less than 32 bits.

2. Names given in bold font are those of saved setup elements. Names given in italics are not actual elements that can be used directly, but show components within full-word elements with distinct functions.

In the Power PMAC script-language environment, both with buffered program statements and on-line commands, elements that are less than 32 bits are accessed using the actual bit width of the element. For example **Gate1[*i*].Chan[*j*].Status** occupies bits 08 – 31 of a 32-bit word. In the script environment, it is treated as a 24-bit value. **Gate1[*i*].Chan[*j*].HomeFlag** occupies bit 24 of this same word. It can be accessed directly as a single-bit value in the script environment

In the C-language, IC registers can be accessed only as full 32-bit words, even if the registers do not occupy all 32-bit words. So **Gate1[*i*].Chan[*j*].Status** is treated as a 32-bit value, with the low 8 bits being meaningless. The single-bit value **Gate1[*i*].Chan[*j*].HomeFlag** cannot be accessed directly from a C program; it would have to be derived with an expression such as

(Gate1[*i*].Chan[*j*].Status & $01000000) >> 24

DSPGATE1 (PMAC2-Style Servo ASIC) Register Elements

The DSPGATE1 Servo ASIC is present on UMAC boards ACC-24E2, ACC-24E2A, ACC-24E2S, and ACC-51E. The data structure for the IC can be referred to by its generic name **Gate1[*i*]** or by the board name **Acc24E2[*i*]**, **Acc24E2A[*i*]**, **Acc24E2S[*i*]**, or **Acc51E[*i*]**, respectively.

Gate1[#]	4	5	6	7	8	9	10	11
IC Base Offset	$600000	$600100	$700000	$700100	$608000	$608100	$708000	$708100
Gate1[#]	12	13	14	15	16	17	18	19
IC Base Offset	$610000	$610100	$710000	$710100	$618000	$618100	$718000	$718100

Data Structure	Full-Word Element	Partial-Word Element	Offset from ASIC Base	Bits
Gate1[*i*].	**ClockCtrl**		$030	
		HardwareClockCtrl		08-19
		PhaseServoDir		20-21
		PhaseClockDiv		24-27
		ServoClockDiv		28-31
	DacStrobe		$070	08-31
	AdcStrobe		$0B0	08-31
	PwmCtrl		$0F0	08-31
		PwmDeadTime		08-15
		PwmPeriod		16-31
	Chan[0].		$000	
	Chan[1].		$040	
	Chan[2].		$080	
	Chan[3].		$0C0	

Chan[#]	0	1	2	3
Offset from IC Base	$000	$040	$080	$0C0

Data Structure	Full-Word Element	Partial-Word Element	Offset from Channel Base	Bits
Gate1[*i*].Chan[*j*].	TimeBetweenCts		$000	08-31
		SCLK cycles between		08-30
		Change of direction		31
	*Hardware 1/T values**			08-31
		*CompA fraction**		08-19
		*ServoCapt fraction**		20-31
	TimeSinceCts		$004	08-31
		SCLK cycles since last		08-30
		{reserved}		31
	*Hardware 1/T values**			08-31
		*CompB fraction**		08-19
		*HomeCapt fraction**		20-31
	Pwm[0] / Dac[0]		$008	08-31
	Pwm[1] / Dac[1]		$00C	08-31

```
Pwm[2] / Pfm                    $010        08-31
Adc[0]                          $014        08-31
Adc[1]                          $018        08-31
CompB                           $01C        08-31
Status                          $020        08-31
            HallState                       08-10
            CountError                      16
            Equ                             17
            PosCapt                         18-19
            ABC                             20-22
            Fault (Amp fault)               23
            HomeFlag                        24
            PlusLimit                       25
            MinusLimit                      26
            UserFlag                        27
            UVW                             28-30
            T                               31
PhaseCapt                       $024        08-31
ServoCapt                       $028        08-31
HomeCapt                        $02C        08-31

Ctrl                            $034        08-31
            EncCtrl                         08-11
            CaptCtrl                        12-15
            CaptFlagSel                     16-17
            PosClear                        18
            EquWrite                        19-20
            Equ1Ena                         21
            AmpEna                          22
            GatedIndexSel                   23
            IndexGateState                  24-25
            OneOverTEna                     26
            PfmDirPol                       27
            OutputPol                       28-29
            OutputMode                      30-31
CompAdd                         $038        08-31
CompA                           $03C        08-31
(* if OneOverTEna is set to 1)
```

Each UMAC card with a DSPGATE1 IC has an "ID chip" with information about the card. The base address offset of the ID chip is given in the table below, followed by information about the registers in chip.

Gate1[#]	4	5	6	7	8	9	10	11
Cid[#]	2	10	3	11	18	26	19	27
ID Chip Base Offset	$D00040	$D00140	$D00050	$D00150	$D08040	$D08140	$D08050	$D08150
Gate1[#]	12	13	14	15	16	17	18	19
Cid[#]	34	42	35	43	50	58	51	59
ID Chip Base Offset	$D10040	$D10140	$D10050	$D10150	$D18040	$D18140	$D18050	$D18150

Full-Word Register	Partial-Word Component	Offset from IC Base	Bits
Reg0		$000	08-15
	*Vendor ID bits 0-3**		08-11
	*Revision number***		08-11
	Clock buffer direction (1=out)		12
	1st chan encoder loss (0 = loss)		13
Reg1		$004	08-15
	*Vendor ID bits 4-7**		08-11
	*Card number bits 0-3***		08-11
	Bank select output		12
	2nd chan encoder loss (0 = loss)		13
Reg2		$008	08-15
	*Card options bits 0-4**		08-12
	*Card number bits 4-8***		08-12
	3rd chan encoder loss (0 = loss)		13
Reg3		$00C	08-15
	*Card options bits 5-9**		08-12
	*Card number bits 9-13***		08-12
	4th chan encoder loss (0 = loss)		13

(* when bank select output set to 0)
(** when bank select output set to 1)

DSPGATE2 (PMAC2-Style MACRO/IO ASIC) Register Elements

The DSPGATE2 MACRO IC is present on UMAC boards ACC-5E. The data structure for the IC can be referred to by its generic name **Gate2[*i*]** or by the board name **Acc5E[*i*]**. Note that the structure indices only match the listed base address offsets if the cards are set by DIP-switches to the lowest possible address offsets.

Gate2[#]	0	1	2	3	4	5	6	7
IC Base Offset	$800000	$800800	$801000	$801800	$808000	$808800	$809000	$809800
Gate2[#]	8	9	10	11	12	13	14	15
IC Base Offset	$810000	$810800	$811000	$811800	$818000	$818800	$819000	$819800

Data Structure	Full-Word Element	Partial-Word Element	Offset from ASIC Base	Bits
Gate2[*i*].	LowIoData		$000	08-31
	HighIoData		$004	08-23
	MuxData		$008	08-23
	DispData		$00C	08-19
	LowIoMode		$010	08-31
	HighIoMode		$014	08-15
	MuxMode		$018	08-23
	DispMode		$01C	08-19
	LowIoDir		$020	08-31
	HighIoDir		$024	08-15
	MuxDir		$028	08-23
	DispDir		$02C	08-19
	LowIoPol		$030	08-31
	HighIoPol		$034	08-15
	MuxPol		$038	08-23
	DispPol		$03C	08-19
	LowIoGray		$050	08-31
	IoGrayCtrl		$054	08-12
	MacroEnable		$058	08-31
	MacroMode		$05C	08-23
	DacStrobe		$070	08-31
	ClockCtrl		$07C	08-31
		HardwareClockCtrl		08-19
		PhaseServoDir		20-21
		PhaseClockDiv		24-27
		ServoClockDiv		28-31
	AdcStrobe		$0B0	08-31
	PwmCtrl		$0F0	08-31
		PwmDeadTime		08-15
		PwmPeriod		16-31
	Chan[0].		$080	
	Chan[1].		$0C0	
	Macro[*m*][*n*] (*m*=0 to 15, *n*=0 to 3)		*m**$10+*n**4+$100	

Chan[#]	0	1
Offset from IC Base	$080	$0C0

Data Structure	Full-Word Element	Partial-Word Element	Offset from Channel Base	Bits
Gate2[*i*].Chan[*j*].	TimeBetweenCts		$000	08-31
	TimeSinceCts		$004	08-31
	Pwm[0] / Dac[0]		$008	08-31
	Pwm[1] / Dac[1]		$00C	08-31
	Pwm[2] / Pfm		$010	08-31
	Adc[0]		$014	08-31
	Adc[1]		$018	08-31
	CompB		$01C	08-31
	Status		$020	08-31
		HallState		08-10
		CountError		16
		Equ		17
		PosCapt		18-19
		ABC		20-22
		Fault (*Amp fault*)		23
		HomeFlag		24
		PlusLimit		25
		MinusLimit		26
		UserFlag		27
		UVW		28-30
		T		31
	PhaseCapt		$024	08-31
	ServoCapt		$028	08-31
	HomeCapt		$02C	08-31
	Ctrl		$034	08-31
		EncCtrl		08-11
		CaptCtrl		12-15
		CaptFlagSel		16-17
		PosClear		18
		EquWrite		19-20
		Equ1Ena		21
		AmpEna		22
		GatedIndexSel		23
		IndexGateState		24-25
		OneOverTEna		26
		PfmDirPol		27
		OutputPol		28-29
		OutputMode		30-31
	CompAdd		$038	08-31
	CompA		$03C	08-31

Each UMAC card with a DSPGATE2 IC has an "ID chip" with information about the card. The base address offset of the ID chip is given in the table below, followed by information about the registers in chip.

Gate2[#]	0	1	2	3	4	5	6	7
Cid[#]	4	20	36	52	5	21	37	53
ID Chip Base Offset	$D00080	$D08080	$D10080	$D18080	$D00090	$D08090	$D10090	$D18090
Gate2[#]	8	9	10	11	12	13	14	15
Cid[#]	6	22	38	54	7	23	39	55
ID Chip Base Offset	$D000C0	$D080C0	$D100C0	$D180C0	$D000D0	$D080D0	$D100D0	$D180D0

Full-Word Register	Partial-Word Component	Offset from IC Base	Bits
Reg0		$000	08-15
	*Vendor ID bits 0-3**		08-11
	*Revision number***		08-11
	Clock buffer direction (1=out)		12
Reg1		$004	08-15
	*Vendor ID bits 4-7**		08-11
	*Card number bits 0-3***		08-11
	Bank select output		12
Reg2		$008	08-15
	*Card options bits 0-4**		08-12
	*Card number bits 4-8***		08-12
Reg3		$00C	08-15
	*Card options bits 5-9**		08-12
	*Card number bits 9-13***		08-12

(* when bank select output set to 0)
(** when bank select output set to 1)

DSPGATE3 (PMAC3-Style General ASIC) Register Elements

The DSPGATE3 machine-interface IC is present on UMAC boards ACC-24E3, ACC-5E3, and ACC-59E3. It is also present on the Power Clipper embedded controller and in the Power Brick controller/amplifier. The data structure for the IC can be referred to by its generic name **Gate3[*i*]** or by the board name **Acc24E3[*i*]**, **Acc5E3[*i*]**, **Acc59E3[*i*]**, **Clipper[*i*]**, or **PowerBrick[*i*]**, respectively.

Gate3[#]	0	1	2	3	4	5	6	7
IC Base Offset	$900000	$904000	$908000	$90C000	$910000	$914000	$918000	$91C000
Gate3[#]	8	9	10	11	12	13	14	15
IC Base Offset	$920000	$924000	$928000	$92C000	$930000	$934000	$938000	$93C000

Data Structure	Full-Word Element	Partial-Word Element	Offset from ASIC Base	Bits
Gate3[*i*].	Chan[0]		$000	
	Chan[1]		$080	
	Chan[2]		$100	
	Chan[3]		$180	
	PhaseServoClockCtrl		$200	00-31
		EncLatchDelay		00-07
		{reserved}		08-09
		PhaseServoDir		10-11
		PhaseClockDiv		12-13
		PhaseClockMult		14-15
		PhaseFreq		16-31
	HardwareClockCtrl		$204	00-31
		ServoClockDiv		00-03
		ClockPol		04-07
		FiltClockDiv		08-11
		EncClockDiv		12-15
		PfmClockDiv		16-19
		DacClockDiv		20-23
		AdcEncClockDiv		24-27
		AdcAmpClockDiv		28-31
	DacStrobe		$208	00-31
	AdcAmpCtrl		$20C	00-31
		AdcAmpHeaderBits		00-02
		AdcAmpUtoSConv		03
		AdcStrobeAmpDelay		04-07
		AdcStrobeAmpWord		08-31
	AdcEncCtrl		$210	00-31
		AdcEncHeaderBits		00-02
		AdcEncUtoSConv		03
		AdcStrobeEncDelay		04-07
		AdcStrobeEncWord		08-31
	ResolverCtrl		$214	00-31
		{reserved}		00-19
		ResolverExciteFreq		20-21
		ResolverExciteGain		22-23
		ResolverExcitePhase		24-31

SerialEncCtrl	$218	00-31
SerialProtocol		00-03
{reserved}		04-07
SerialTrigDelay		08-15
SerialTrigEdgeSel		16
SerialTrigClockSel		17
{reserved}		18-19
SerialClockNDiv		20-23
SerialClockMDiv		24-31
WpKey	$21C	00-31
ChipID	$220	00-31
IntCtrl	$224	00-31
InterruptStatus		00-07
Chan[0] Capture		00
Chan[1] Capture		01
Chan[2] Capture		02
Chan[3] Capture		03
Chan[0] Compare		04
Chan[1] Compare		05
Chan[2] Compare		06
Chan[3] Compare		07
InterruptSource(SignalStatus)		08-15
Chan[0] Capture		08
Chan[1] Capture		09
Chan[2] Capture		10
Chan[3] Capture		11
Chan[0] Compare		12
Chan[1] Compare		13
Chan[2] Compare		14
Chan[3] Compare		15
InterruptEnable		16-23
Chan[0] Capture		16
Chan[1] Capture		17
Chan[2] Capture		18
Chan[3] Capture		19
Chan[0] Compare		20
Chan[1] Compare		21
Chan[2] Compare		22
Chan[3] Compare		23
{reserved}		24-31
MacroError	$228	00-31
EEpromCtrl	$22C	00-31
EEpromReadEna		00
EEpromWriteEna		01
EEpromWP		02
EEpromError		03
EEpromSel		04-06
(reserved}		02-31
EEpromData[0]	$230	00-31
EEpromData[1]	$234	00-31
EEpromData[2]	$238	00-31
EEpromData[3]	$23C	00-31
GpioCtrl / GpioMode[0]	$240	00-31
GpioMode[1]	$244	00-31
GpioMode[2]	$248	00-31
GpioMode[3]	$24C	00-31
GpioData[0]	$250	00-31

GpioData[1]	$254	00-31
GpioData[2]	$258	00-31
GpioData[3]	$25C	00-31
GpioDir[0]	$260	00-31
GpioDir[1]	$264	00-31
GpioDir[2]	$268	00-31
GpioDir[3]	$26C	00-31
GpioPol[0]	$270	00-31
GpioPol[1]	$274	00-31
GpioPol[2]	$278	00-31
GpioPol[3]	$27C	00-31
ReadAlias[*n*] (*n* = 0 to 14)	*n**4+$280	00-31
WriteAlias[*n*] (*n* = 0 to 14)	*n**4+$2C0	00-31
MemAlias[*n*] (*n* = 0 to 59)	*n**4+$300	00-31
MacroEnableA	$3F0	00-23
MacroModeA	$3F4	00-15
MacroEnableB	$3F8	00-23
MacroModeB	$3FC	00-15
MacroInA[*m*][*n*] (*m* = 0 to 15, *n* = 0 to 3)	*m**$10+*n**4+$400	00-31
MacroOutA[*m*][*n*] (*m* = 0 to 15, *n* = 0 to 3)	*m**$10+*n**4+$500	00-31
MacroInB[*m*][*n*] (*m* = 0 to 15, *n* = 0 to 3)	*m**$10+*n**4+$600	00-31
MacroOutB[*m*][*n*] (*m* = 0 to 15, *n* = 0 to 3)	*m**$10+*n**4+$700	00-31

Chan[#]	0	1	2	3
Offset from IC Base	$000	$080	$100	$180

Data Structure	Full-Word Element	Partial-Word Element	Offset from Channel Base	Bits
Gate3[*i*].Chan[*j*].	Status		$000	00-31
		ABPins		00-03
		ABC		04-06
		Fault (*Amp fault*)		07
		HomeFlag		08
		PlusLimit		09
		MinusLimit		10
		UserFlag		11
		UVW		12-14
		T		15
		HallState		16-18
		DemuxInvalid		19
		PosCapt		20-21
		TrigState		22
		Equ		24
		EquOut		25

	LossStatus		28
	LossCapt		29
	CountError		30
	SosError		31
PhaseCapt		$004	00-31
ServoCapt		$008	00-31
AtanSumOfSqr		$00C	00-31
	SumOfSquares		00-15
	Atan		16-31
TimerA		$010	00-31
TimerB		$014	00-31
SerialEncDataA		$018	00-31
SerialEncDataB		$01C	00-31
AdcAmp[0]		$020	00-31
AdcAmp[1]		$024	00-31
AdcAmp[2]		$028	00-31
AdcAmp[3]		$02C	00-31
AdcEnc[0]		$030	00-31
AdcEnc[1]		$034	00-31
AdcEnc[2]		$038	00-31
AdcEnc[3]		$03C	00-31
Pwm[0] / Dac[0]		$040	00-31
Pwm[1] / Dac[1]		$044	00-31
Pwm[2] / Dac[2]		$048	00-31
Pwm[3] / Pfm		$04C	00-31
CompA		$050	00-31
CompB		$054	00-31
CompAdd		$058	00-31
SerialEncCmd		$05C	00-31
	SerialPosNumBits		00-05
	SerialStatusNumBits		06-09
	SerialDataReady		10
	SerialGrayBinConv		11
	SerialTrigEna		12
	SerialTrigMode		13
	SerialParityType		14-15
	SerialCmdWord		16-31
AdcOffset[0]		$060	00-31
AdcOffset[1]		$064	00-31
InCtrl		$068	00-31
	EncCtrl		00-03
	TimerMode		04-05
	CaptCtrl		06-09
	CaptFlagSel		10-11
	CaptFlagChan		12-13
	GatedIndexSel		14
	IndexGateState		15
	IndexDemuxEna		16
	FlagFilt2Ena		17
	AtanEna		18
	CountReset		19
	SerialEncEna		20
	PackInData		21-22
	{reserved}		23-31
OutCtrl		$06C	00-31
	EquOutMask		00-03
	EquOutPol		04

	Equ1Ena		05
	EquWrite		06-07
	AmpEna		08
	OutFlagB		09
	OutFlagC		10
	OutFlagD		11
	OutputMode		12-15
	OutputPol		16-17
	PfmDirPol		18
	PfmFormat		19
	PwmFreqMult		20-22
	PackOutData		23
	PwmDeadTime		24-31
PfmWidth		$070	00-11
HomeCapt		$074	00-31

Unlike other Power PMAC interfaces, those built around PMAC3-style ICs do not have a separate ID chip that is directly accessed by the processor. Instead, the ID information is stored in an EEPROM IC that is read through the DSPGATE3 IC in the **EEpromData[*k*]** elements in the IC.

General-Purpose I/O Register Elements

The Power PMAC supports a wide variety of general-purpose input/output accessories, both analog and digital. These are mapped into an I/O address space with offsets from $A00000 through $DFFFFF. The user can combine different kinds of accessories in this space, but must make sure that there are no addressing conflicts between any of the different types of these cards.

IOGATE (PMAC2-Style I/O ASIC) Register Elements

The IOGATE I/O ASIC is present on UMAC boards ACC-14E, ACC-65E, ACC-66E, ACC-67E, and ACC-68E. The data structure for the IC can be referred to by its generic name **GateIo[*i*]** or by the board name **Acc14E[*i*]**, **Acc65E[*i*]**, **Acc66E[*i*]**, **Acc67E[*i*]**, **Acc68E[*i*]**, respectively. The older ACC-11E I/O board also uses the IOGATE IC, but it does not contain the identification circuitry required for the CPU to be able to use the pre-defined data structure. ACC-11E registers must be access with "pointer" (ptr) M-variables.

GateIo[#]	0	1	2	3	4	5	6	7
IC Base Offset	$A00000	$B00000	$C00000	$D00000	$A08000	$B08000	$C08000	$D08000
GateIo[#]	8	9	10	11	12	13	14	15
IC Base Offset	$A10000	$B10000	$C10000	$D10000	$A18000	$B18000	$C18000	$D18000

Data Structure	Full-Word Element	Partial-Word Element	Offset from ASIC Base	Bits
GateIo[*i*].	DataReg[0]		$000	08-15
		DataReg[0].0.1		08
		DataReg[0].1.1		09
		DataReg[0].2.1		10
		DataReg[0].3.1		11
		DataReg[0].4.1		12
		DataReg[0].5.1		13
		DataReg[0].6.1		14
		DataReg[0].7.1		15
	DataReg[1]		$004	08-15
		{as for DataReg[0]}		
	DataReg[2]		$008	08-15
		{as for DataReg[0]}		
	DataReg[3]		$00C	08-15
		{as for DataReg[0]}		
	DataReg[4]		$010	08-15
		{as for DataReg[0]}		
	DataReg[5]		$014	08-15
		{as for DataReg[0]}		
	IntrReg		$018	08-15
	CtrlReg		$01C	08-15

Each modern UMAC card with an IOGATE IC (but not the older ACC-11E) has an "ID chip" with information about the card. The base address offset of the ID chip is given in the table below, followed by information about the registers in chip.

GateIo[#]	0	1	2	3	4	5	6	7
Cid[#]	12	13	14	15	28	29	30	31
ID Chip Base Offset	$D00180	$D00190	$D001C0	$D001D0	$D08180	$D08190	$D081C0	$D081D0
GateIo[#]	8	9	10	11	12	13	14	15
Cid[#]	44	45	46	47	60	61	62	63
ID Chip Base Offset	$D10180	$D10190	$D101C0	$D101D0	$D18180	$D18190	$D181C0	$D181D0

Full-Word Register	Partial-Word Component	Offset from IC Base	Bits
Reg0		$000	08-15
	*Vendor ID bits 0-3**		08-11
	*Revision number***		08-11
Reg1		$004	08-15
	*Vendor ID bits 4-7**		08-11
	*Card number bits 0-3***		08-11
	Bank select output		12
Reg2		$008	08-15
	*Card options bits 0-4**		08-12
	*Card number bits 4-8***		08-12
Reg3		$00C	08-15
	*Card options bits 5-9**		08-12
	*Card number bits 9-13***		08-12

(* when bank select output set to 0)
(** when bank select output set to 1)

ACC-28E (A/D Converter) Register Elements

The ACC-28E high-resolution A/D converter cards have 4 16-bit read-only registers that can be accessed either as unsigned ("U") or signed ("S") values through the choice of elements.

Acc28E[#]	0	1	2	3	4	5	6	7
IC Base Offset	$A00000	$B00000	$C00000	$D00000	$A08000	$B08000	$C08000	$D08000
Acc28E[#]	8	9	10	11	12	13	14	15
IC Base Offset	$A10000	$B10000	$C10000	$D10000	$A18000	$B18000	$C18000	$D18000

Data Structure	Full-Word Element	Offset from ASIC Base	Bits
Acc28E[*i*].	AdcUdata[0]	$000	16-31
	AdcSdata[0]	$000	16-31
	AdcUdata[1]	$004	16-31
	AdcSdata[1]	$004	16-31
	AdcUdata[2]	$008	16-31
	AdcSdata[2]	$008	16-31
	AdcUdata[3]	$00C	16-31
	AdcSdata[3]	$00C	16-31

Each ACC-28E has an "ID chip" with information about the card. The base address offset of the ID chip is given in the table below, followed by information about the registers in chip.

Acc28E[#]	0	1	2	3	4	5	6	7
Cid[#]	12	13	14	15	28	29	30	31
ID Chip Base Offset	$D00180	$D00190	$D001C0	$D001D0	$D08180	$D08190	$D081C0	$D081D0
Acc28E[#]	8	9	10	11	12	13	14	15
Cid[#]	44	45	46	47	60	61	62	63
ID Chip Base Offset	$D10180	$D10190	$D101C0	$D101D0	$D18180	$D18190	$D181C0	$D181D0

Full-Word Register	Partial-Word Component	Offset from IC Base	Bits
Reg0		$000	08-15
	*Vendor ID bits 0-3**		08-11
	*Revision number***		08-11
Reg1		$004	08-15
	*Vendor ID bits 4-7**		08-11
	*Card number bits 0-3***		08-11
	Bank select output		12
Reg2		$008	08-15
	*Card options bits 0-4**		08-12
	*Card number bits 4-8***		08-12
Reg3		$00C	08-15
	*Card options bits 5-9**		08-12
	*Card number bits 9-13***		08-12

```
(* when bank select output set to 0)
(** when bank select output set to 1)
```

ACC-36E (A/D Converter) Register Elements

The ACC-36E multiplexed A/D converter cards have a single 24-bit register that can be written to as a control register or read from as a data register, accessing the selected multiplexed ADCs either as unsigned ("U") or signed ("S") values through the choice of elements.

Acc36E[#]	0	1	2	3	4	5	6	7
IC Base Offset	$A00000	$B00000	$C00000	$D00000	$A08000	$B08000	$C08000	$D08000
Acc36E[#]	8	9	10	11	12	13	14	15
IC Base Offset	$A10000	$B10000	$C10000	$D10000	$A18000	$B18000	$C18000	$D18000

Data Structure	Full-Word Element	Partial-Word Element	Offset from ASIC Base	Bits
Acc36E[*i*].	ConvertCode (*write-only*)		$000	08-31
	ADCHighLow (*read-only*)		$000	08-31
		ADCsLow		08-19
		ADCuLow		08-19
		ADCsHigh		20-31
		ADCuHigh		20-31

Each ACC-36E has an "ID chip" with information about the card. The base address offset of the ID chip is given in the table below, followed by information about the registers in chip.

Acc36E[#]	0	1	2	3	4	5	6	7
Cid[#]	12	13	14	15	28	29	30	31
ID Chip Base Offset	$D00180	$D00190	$D001C0	$D001D0	$D08180	$D08190	$D081C0	$D081D0
Acc36E[#]	8	9	10	11	12	13	14	15
Cid[#]	44	45	46	47	60	61	62	63
ID Chip Base Offset	$D10180	$D10190	$D101C0	$D101D0	$D18180	$D18190	$D181C0	$D181D0

Full-Word Register	Partial-Word Component	Offset from IC Base	Bits
Reg0		$000	08-15
	*Vendor ID bits 0-3**		08-11
	ADCRdy*		13
	*Revision number***		08-11
Reg1		$004	08-15
	*Vendor ID bits 4-7**		08-11
	*Card number bits 0-3***		08-11
	Bank select output		12
Reg2		$008	08-15

```
                                Card options bits 0-4*        08-12
                                Card number bits 4-8**        08-12
            Reg3                                $00C          08-15
                                Card options bits 5-9*        08-12
                                Card number bits 9-13**       08-12

(* when bank select output set to 0)
(** when bank select output set to 1)
```

ACC-59E (A/D & D/A Converter) Register Elements

The ACC-59E converter cards have a single 24-bit register that can be written to as a control register or read from as a data register for the A/D converters, accessing the selected multiplexed ADCs either as unsigned ("U") or signed ("S") values through the choice of elements. They also have four 24-bit registers, each containing two 12-bit DAC values.

Acc59E[#]	0	1	2	3	4	5	6	7
IC Base Offset	$A00000	$B00000	$C00000	$D00000	$A08000	$B08000	$C08000	$D08000
Acc59E[#]	8	9	10	11	12	13	14	15
IC Base Offset	$A10000	$B10000	$C10000	$D10000	$A18000	$B18000	$C18000	$D18000

Data Structure	Full-Word Element	Partial-Word Element	Offset from ASIC Base	Bits
Acc59E[*i*].	ConvertCode (*write-only*)		$000	08-31
	ADCs (*read-only*)		$000	08-19
	ADCu (*read-only*)		$000	08-19
	DACHighLow[0]		$008	08-31
		DAC[0]		08-19
		DAC[4]		20-31
	DACHighLow[1]		$009	08-31
		DAC[1]		08-19
		DAC[5]		20-31
	DACHighLow[2]		$00A	08-31
		DAC[2]		08-19
		DAC[6]		20-31
	DACHighLow[3]		$00B	08-31
		DAC[3]		08-19
		DAC[7]		20-31

Each ACC-59E has an "ID chip" with information about the card. The base address offset of the ID chip is given in the table below, followed by information about the registers in chip.

Acc59E[#]	0	1	2	3	4	5	6	7
Cid[#]	12	13	14	15	28	29	30	31
ID Chip Base Offset	$D00180	$D00190	$D001C0	$D001D0	$D08180	$D08190	$D081C0	$D081D0
Acc59E[#]	8	9	10	11	12	13	14	15
Cid[#]	44	45	46	47	60	61	62	63
ID Chip Base Offset	$D10180	$D10190	$D101C0	$D101D0	$D18180	$D18190	$D181C0	$D181D0

Full-Word Register	Partial-Word Component	Offset from IC Base	Bits
Reg0		$000	08-15
	*Vendor ID bits 0-3**		08-11
	ADCRdy*		13
	*Revision number***		08-11
Reg1		$004	08-15
	*Vendor ID bits 4-7**		08-11
	*Card number bits 0-3***		08-11
	Bank select output		12
Reg2		$008	08-15
	*Card options bits 0-4**		08-12
	*Card number bits 4-8***		08-12
Reg3		$00C	08-15
	*Card options bits 5-9**		08-12
	*Card number bits 9-13***		08-12

(* when bank select output set to 0)
(** when bank select output set to 1)

ACC-84E (Serial Encoder Interface) Register Elements

The ACC-84E serial-encoder interface boards for UMAC systems have 1 multi-channel setup element and, for each of the four channels, 1 setup element and 4 status elements. The ACC-84C, ACC-84B, and ACC-84S boards have the same elements. All elements are 24 bits wide.

Note

The ACC-84C boards for Compact UMAC systems can take any of the 16 index values 0 – 15. The ACC-84B boards for Power Brick systems and the ACC-84S boards for Power Clipper systems can only take the index values 0 and 1.

Acc84E[#]	0	1	2	3	4	5	6	7
IC Base Offset	$A00000	$B00000	$C00000	$D00000	$A08000	$B08000	$C08000	$D08000
Acc84E[#]	8	9	10	11	12	13	14	15
IC Base Offset	$A10000	$B10000	$C10000	$D10000	$A18000	$B18000	$C18000	$D18000

Data Structure	Full-Word Element	Offset from ASIC Base	Bits
Acc84E[*i*].	**SerialEncCtrl**	$07C	08-31

Chan[#]	0	1	2	3
Offset from IC Base	$000	$010	$040	$050

Data Structure	Full-Word Element	Offset from Channel Base	Bits
Acc84E[*i*].Chan[*j*].SerialEncDataA		$000	08-31
Acc84E[*i*].Chan[*j*].SerialEncDataB		$004	08-31
Acc84E[*i*].Chan[*j*].SerialEncDataC		$008	08-31
Acc84E[*i*].Chan[*j*].SerialEncDataD		$00C	08-31
Acc84E[*i*].Chan[*j*].	**SerialEncCmd**	$020	08-31

Each ACC-84E has an "ID chip" with information about the card. The base address offset of the ID chip is given in the table below, followed by information about the registers in chip.

Acc84E[#]	0	1	2	3	4	5	6	7
Cid[#]	12	13	14	15	28	29	30	31
ID Chip Base Offset	$D00180	$D00190	$D001C0	$D001D0	$D08180	$D08190	$D081C0	$D081D0
Acc84E[#]	8	9	10	11	12	13	14	15
Cid[#]	44	45	46	47	60	61	62	63
ID Chip Base Offset	$D10180	$D10190	$D101C0	$D101D0	$D18180	$D18190	$D181C0	$D181D0

Full-Word Register	Partial-Word Component	Offset from IC Base	Bits
Reg0		$000	08-15
	*Vendor ID bits 0-3**		08-11
	*Revision number***		08-11
Reg1		$004	08-15
	*Vendor ID bits 4-7**		08-11
	*Card number bits 0-3***		08-11
	Bank select output		12
Reg2		$008	08-15
	*Card options bits 0-4**		08-12
	*Card number bits 4-8***		08-12
Reg3		$00C	08-15
	*Card options bits 5-9**		08-12
	*Card number bits 9-13***		08-12

(* when bank select output set to 0)
(** when bank select output set to 1)

POWER PMAC SUGGESTED I/O POINTER DECLARATIONS

Power PMAC's "M" (pointer) variables are a very useful method for accessing information of interest, particularly digital and analog I/O. The section provides suggestions for declarations of user names to these variables for commonly used information. Remember that these are merely suggestions. In many cases, application-specific variable names that denote what the I/O point is used for in the application will replace these names.

UMAC I/O Cards

UMAC rack-mounted controller systems provide a wide variety of digital and analog I/O boards. These boards map into the I/O space of the Power PMAC at address offsets between $A00000 and $DFFFFF from the base address of the I/O space. Up to 16 (total) of these boards may be used in a single UMAC rack. Each I/O board used must have a unique setting of the DIP switches SW1-1 to SW1-4.

The pointer-variable declarations for "digital I/O cards" in this section provide access to the I/O points and setup registers on UMAC digital I/O boards that use the IOGATE IC, including the ACC-11E, ACC-14E, ACC-65E, ACC-66E, ACC-67E, and ACC-68E.

The pointer-variable declarations for "ACC-28E A/D cards" in this section provide access to the four 16-bit A/D-converter registers on each ACC-28E high-resolution A/D converter board.

ACC-28E A/D Card 0

(SW1-1 ON, SW1-2 ON, SW1-3 ON, SW1-4 ON) (These are the default switch settings.)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard0Reg0->u.io:$A00000.16.16          // ACC-28E Card 0 ADC1
ptr AdcCard0Reg1->u.io:$A00004.16.16          // ACC-28E Card 0 ADC2
ptr AdcCard0Reg2->u.io:$A00008.16.16          // ACC-28E Card 0 ADC3
ptr AdcCard0Reg3->u.io:$A0000C.16.16          // ACC-28E Card 0 ADC4
```

ACC-28E A/D Card 1

(SW1-1 OFF, SW1-2 ON, SW1-3 ON, SW1-4 ON)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard1Reg0->u.io:$B00000.16.16          // ACC-28E Card 1 ADC1
ptr AdcCard1Reg1->u.io:$B00004.16.16          // ACC-28E Card 1 ADC2
ptr AdcCard1Reg2->u.io:$B00008.16.16          // ACC-28E Card 1 ADC3
ptr AdcCard1Reg3->u.io:$B0000C.16.16          // ACC-28E Card 1 ADC4
```

ACC-28E A/D Card 2

(SW1-1 ON, SW1-2 OFF, SW1-3 ON, SW1-4 ON)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard2Reg0->u.io:$C00000.16.16          // ACC-28E Card 2 ADC1
ptr AdcCard2Reg1->u.io:$C00004.16.16          // ACC-28E Card 2 ADC2
ptr AdcCard2Reg2->u.io:$C00008.16.16          // ACC-28E Card 2 ADC3
ptr AdcCard2Reg3->u.io:$C0000C.16.16          // ACC-28E Card 2 ADC4
```

ACC-28E A/D Card 3

(SW1-1 OFF, SW1-2 OFF, SW1-3 ON, SW1-4 ON)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard3Reg0->u.io:$D00000.16.16          // ACC-28E Card 2 ADC1
ptr AdcCard3Reg1->u.io:$D00004.16.16          // ACC-28E Card 2 ADC2
ptr AdcCard3Reg2->u.io:$D00008.16.16          // ACC-28E Card 2 ADC3
ptr AdcCard3Reg3->u.io:$D0000C.16.16          // ACC-28E Card 2 ADC4
```

ACC-28E A/D Card 4

(SW1-1 ON, SW1-2 ON, SW1-3 OFF, SW1-4 ON)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard4Reg0->u.io:$A08000.16.16          // ACC-28E Card 4 ADC1
ptr AdcCard4Reg1->u.io:$A08004.16.16          // ACC-28E Card 4 ADC2
ptr AdcCard4Reg2->u.io:$A08008.16.16          // ACC-28E Card 4 ADC3
ptr AdcCard4Reg3->u.io:$A0800C.16.16          // ACC-28E Card 4 ADC4
```

ACC-28E A/D Card 5

(SW1-1 OFF, SW1-2 ON, SW1-3 OFF, SW1-4 ON)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard5Reg0->u.io:$B08000.16.16          // ACC-28E Card 5 ADC1
ptr AdcCard5Reg1->u.io:$B08004.16.16          // ACC-28E Card 5 ADC2
ptr AdcCard5Reg2->u.io:$B08008.16.16          // ACC-28E Card 5 ADC3
ptr AdcCard5Reg3->u.io:$B0800C.16.16          // ACC-28E Card 5 ADC4
```

ACC-28E A/D Card 6

(SW1-1 ON, SW1-2 OFF, SW1-3 OFF, SW1-4 ON)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard6Reg0->u.io:$C08000.16.16          // ACC-28E Card 6 ADC1
ptr AdcCard6Reg1->u.io:$C08004.16.16          // ACC-28E Card 6 ADC2
ptr AdcCard6Reg2->u.io:$C08008.16.16          // ACC-28E Card 6 ADC3
ptr AdcCard6Reg3->u.io:$C0800C.16.16          // ACC-28E Card 6 ADC4
```

ACC-28E A/D Card 7

(SW1-1 OFF, SW1-2 OFF, SW1-3 OFF, SW1-4 ON)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard7Reg0->u.io:$D08000.16.16          // ACC-28E Card 7 ADC1
ptr AdcCard7Reg1->u.io:$D08004.16.16          // ACC-28E Card 7 ADC2
ptr AdcCard7Reg2->u.io:$D08008.16.16          // ACC-28E Card 7 ADC3
ptr AdcCard7Reg3->u.io:$D0800C.16.16          // ACC-28E Card 7 ADC4
```

ACC-28E A/D Card 8

(SW1-1 ON, SW1-2 ON, SW1-3 ON, SW1-4 OFF)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard8Reg0->u.io:$A10000.16.16          // ACC-28E Card 8 ADC1
ptr AdcCard8Reg1->u.io:$A10004.16.16          // ACC-28E Card 8 ADC2
ptr AdcCard8Reg2->u.io:$A10008.16.16          // ACC-28E Card 8 ADC3
ptr AdcCard8Reg3->u.io:$A1000C.16.16          // ACC-28E Card 8 ADC4
```

ACC-28E A/D Card 9

(SW1-1 OFF, SW1-2 ON, SW1-3 ON, SW1-4 OFF)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard9Reg0->u.io:$B10000.16.16          // ACC-28E Card 9 ADC1
ptr AdcCard9Reg1->u.io:$B10004.16.16          // ACC-28E Card 9 ADC2
ptr AdcCard9Reg2->u.io:$B10008.16.16          // ACC-28E Card 9 ADC3
ptr AdcCard9Reg3->u.io:$B1000C.16.16          // ACC-28E Card 9 ADC4
```

ACC-28E A/D Card 10

(SW1-1 ON, SW1-2 OFF, SW1-3 ON, SW1-4 OFF)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard10Reg0->u.io:$C10000.16.16         // ACC-28E Card 10 ADC1
ptr AdcCard10Reg1->u.io:$C10004.16.16         // ACC-28E Card 10 ADC2
ptr AdcCard10Reg2->u.io:$C10008.16.16         // ACC-28E Card 10 ADC3
ptr AdcCard10Reg3->u.io:$C1000C.16.16         // ACC-28E Card 10 ADC4
```

ACC-28E A/D Card 11

(SW1-1 OFF, SW1-2 OFF, SW1-3 ON, SW1-4 OFF)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard11Reg0->u.io:$D10000.16.16         // ACC-28E Card 11 ADC1
ptr AdcCard11Reg1->u.io:$D10004.16.16         // ACC-28E Card 11 ADC2
ptr AdcCard11Reg2->u.io:$D10008.16.16         // ACC-28E Card 11 ADC3
ptr AdcCard11Reg3->u.io:$D1000C.16.16         // ACC-28E Card 11 ADC4
```

ACC-28E A/D Card 12

(SW1-1 ON, SW1-2 ON, SW1-3 OFF, SW1-4 OFF)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard12Reg0->u.io:$A18000.16.16         // ACC-28E Card 12 ADC1
ptr AdcCard12Reg1->u.io:$A18004.16.16         // ACC-28E Card 12 ADC2
ptr AdcCard12Reg2->u.io:$A18008.16.16         // ACC-28E Card 12 ADC3
ptr AdcCard12Reg3->u.io:$A1800C.16.16         // ACC-28E Card 12 ADC4
```

ACC-28E A/D Card 13

(SW1-1 OFF, SW1-2 ON, SW1-3 OFF, SW1-4 OFF)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard13Reg0->u.io:$B18000.16.16        // ACC-28E Card 13 ADC1
ptr AdcCard13Reg1->u.io:$B18004.16.16        // ACC-28E Card 13 ADC2
ptr AdcCard13Reg2->u.io:$B18008.16.16        // ACC-28E Card 13 ADC3
ptr AdcCard13Reg3->u.io:$B1800C.16.16        // ACC-28E Card 13 ADC4
```

ACC-28E A/D Card 14

(SW1-1 ON, SW1-2 OFF, SW1-3 OFF, SW1-4 OFF)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard14Reg0->u.io:$C18000.16.16        // ACC-28E Card 14 ADC1
ptr AdcCard14Reg1->u.io:$C18004.16.16        // ACC-28E Card 14 ADC2
ptr AdcCard14Reg2->u.io:$C18008.16.16        // ACC-28E Card 14 ADC3
ptr AdcCard14Reg3->u.io:$C1800C.16.16        // ACC-28E Card 14 ADC4
```

ACC-28E A/D Card 15

(SW1-1 OFF, SW1-2 OFF, SW1-3 OFF, SW1-4 OFF)

```
// 16-bit variables used for accessing A/D values
ptr AdcCard15Reg0->u.io:$D18000.16.16        // ACC-28E Card 15 ADC1
ptr AdcCard15Reg1->u.io:$D18004.16.16        // ACC-28E Card 15 ADC2
ptr AdcCard15Reg2->u.io:$D18008.16.16        // ACC-28E Card 15 ADC3
ptr AdcCard15Reg3->u.io:$D1800C.16.16        // ACC-28E Card 15 ADC4
```

Digital I/O Card 0

(SW1-1 ON, SW1-2 ON, SW1-3 ON, SW1-4 ON) (These are the default switch settings. If there is a single digital I/O card in the UMAC rack, it will almost always be used at this base address.)

```
// Single-bit variables used for accessing I/O points
ptr IoCard0Pt00->u.io:$A00000.8.1            // I/O Card 0 I/O00
ptr IoCard0Pt01->u.io:$A00000.9.1            // I/O Card 0 I/O01
ptr IoCard0Pt02->u.io:$A00000.10.1           // I/O Card 0 I/O02
ptr IoCard0Pt03->u.io:$A00000.11.1           // I/O Card 0 I/O03
ptr IoCard0Pt04->u.io:$A00000.12.1           // I/O Card 0 I/O04
ptr IoCard0Pt05->u.io:$A00000.13.1           // I/O Card 0 I/O05
ptr IoCard0Pt06->u.io:$A00000.14.1           // I/O Card 0 I/O06
ptr IoCard0Pt07->u.io:$A00000.15.1           // I/O Card 0 I/O07
ptr IoCard0Pt08->u.io:$A00004.8.1            // I/O Card 0 I/O08
ptr IoCard0Pt09->u.io:$A00004.9.1            // I/O Card 0 I/O09
ptr IoCard0Pt10->u.io:$A00004.10.1           // I/O Card 0 I/O10
ptr IoCard0Pt11->u.io:$A00004.11.1           // I/O Card 0 I/O11
ptr IoCard0Pt12->u.io:$A00004.12.1           // I/O Card 0 I/O12
ptr IoCard0Pt13->u.io:$A00004.13.1           // I/O Card 0 I/O13
ptr IoCard0Pt14->u.io:$A00004.14.1           // I/O Card 0 I/O14
ptr IoCard0Pt15->u.io:$A00004.15.1           // I/O Card 0 I/O15
ptr IoCard0Pt16->u.io:$A00008.8.1            // I/O Card 0 I/O16
ptr IoCard0Pt17->u.io:$A00008.9.1            // I/O Card 0 I/O17
ptr IoCard0Pt18->u.io:$A00008.10.1           // I/O Card 0 I/O18
ptr IoCard0Pt19->u.io:$A00008.11.1           // I/O Card 0 I/O19
ptr IoCard0Pt20->u.io:$A00008.12.1           // I/O Card 0 I/O20
ptr IoCard0Pt21->u.io:$A00008.13.1           // I/O Card 0 I/O21
ptr IoCard0Pt22->u.io:$A00008.14.1           // I/O Card 0 I/O22
ptr IoCard0Pt23->u.io:$A00008.15.1           // I/O Card 0 I/O23
ptr IoCard0Pt24->u.io:$A0000C.8.1            // I/O Card 0 I/O24
ptr IoCard0Pt25->u.io:$A0000C.9.1            // I/O Card 0 I/O25
ptr IoCard0Pt26->u.io:$A0000C.10.1           // I/O Card 0 I/O26
ptr IoCard0Pt27->u.io:$A0000C.11.1           // I/O Card 0 I/O27
ptr IoCard0Pt28->u.io:$A0000C.12.1           // I/O Card 0 I/O28
ptr IoCard0Pt29->u.io:$A0000C.13.1           // I/O Card 0 I/O29
ptr IoCard0Pt30->u.io:$A0000C.14.1           // I/O Card 0 I/O30
ptr IoCard0Pt31->u.io:$A0000C.15.1           // I/O Card 0 I/O31
ptr IoCard0Pt32->u.io:$A00010.8.1            // I/O Card 0 I/O32
ptr IoCard0Pt33->u.io:$A00010.9.1            // I/O Card 0 I/O33
ptr IoCard0Pt34->u.io:$A00010.10.1           // I/O Card 0 I/O34
ptr IoCard0Pt35->u.io:$A00010.11.1           // I/O Card 0 I/O35
ptr IoCard0Pt36->u.io:$A00010.12.1           // I/O Card 0 I/O36
ptr IoCard0Pt37->u.io:$A00010.13.1           // I/O Card 0 I/O37
ptr IoCard0Pt38->u.io:$A00010.14.1           // I/O Card 0 I/O38
ptr IoCard0Pt39->u.io:$A00010.15.1           // I/O Card 0 I/O39
ptr IoCard0Pt40->u.io:$A00014.8.1            // I/O Card 0 I/O40
ptr IoCard0Pt41->u.io:$A00014.9.1            // I/O Card 0 I/O41
ptr IoCard0Pt42->u.io:$A00014.10.1           // I/O Card 0 I/O42
ptr IoCard0Pt43->u.io:$A00014.11.1           // I/O Card 0 I/O43
ptr IoCard0Pt44->u.io:$A00014.12.1           // I/O Card 0 I/O44
ptr IoCard0Pt45->u.io:$A00014.13.1           // I/O Card 0 I/O45
ptr IoCard0Pt46->u.io:$A00014.14.1           // I/O Card 0 I/O46
ptr IoCard0Pt47->u.io:$A00014.15.1           // I/O Card 0 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard0Reg0->u.io:$A00000.8.8     // I/O Card 0 I/O00-07 as byte
ptr IoCard0Reg1->u.io:$A00004.8.8     // I/O Card 0 I/O08-15 as byte
ptr IoCard0Reg2->u.io:$A00008.8.8     // I/O Card 0 I/O16-23 as byte
ptr IoCard0Reg3->u.io:$A0000C.8.8     // I/O Card 0 I/O24-31 as byte
ptr IoCard0Reg4->u.io:$A00010.8.8     // I/O Card 0 I/O32-39 as byte
ptr IoCard0Reg5->u.io:$A00014.8.8     // I/O Card 0 I/O40-47 as byte
ptr IoCard0Reg6->u.io:$A00018.8.8     // I/O Card 0 latch inputs
ptr IoCard0Reg7->u.io:$A0001C.8.8     // I/O Card 0 control register
```

Digital I/O Card 1

(SW1-1 OFF, SW1-2 ON, SW1-3 ON, SW1-4 ON)

```
// Single-bit variables used for accessing I/O points
ptr IoCard1Pt00->u.io:$B00000.8.1            // I/O Card 1 I/O00
ptr IoCard1Pt01->u.io:$B00000.9.1            // I/O Card 1 I/O01
ptr IoCard1Pt02->u.io:$B00000.10.1           // I/O Card 1 I/O02
ptr IoCard1Pt03->u.io:$B00000.11.1           // I/O Card 1 I/O03
ptr IoCard1Pt04->u.io:$B00000.12.1           // I/O Card 1 I/O04
ptr IoCard1Pt05->u.io:$B00000.13.1           // I/O Card 1 I/O05
ptr IoCard1Pt06->u.io:$B00000.14.1           // I/O Card 1 I/O06
ptr IoCard1Pt07->u.io:$B00000.15.1           // I/O Card 1 I/O07
ptr IoCard1Pt08->u.io:$B00004.8.1            // I/O Card 1 I/O08
ptr IoCard1Pt09->u.io:$B00004.9.1            // I/O Card 1 I/O09
ptr IoCard1Pt10->u.io:$B00004.10.1           // I/O Card 1 I/O10
ptr IoCard1Pt11->u.io:$B00004.11.1           // I/O Card 1 I/O11
ptr IoCard1Pt12->u.io:$B00004.12.1           // I/O Card 1 I/O12
ptr IoCard1Pt13->u.io:$B00004.13.1           // I/O Card 1 I/O13
ptr IoCard1Pt14->u.io:$B00004.14.1           // I/O Card 1 I/O14
ptr IoCard1Pt15->u.io:$B00004.15.1           // I/O Card 1 I/O15
ptr IoCard1Pt16->u.io:$B00008.8.1            // I/O Card 1 I/O16
ptr IoCard1Pt17->u.io:$B00008.9.1            // I/O Card 1 I/O17
ptr IoCard1Pt18->u.io:$B00008.10.1           // I/O Card 1 I/O18
ptr IoCard1Pt19->u.io:$B00008.11.1           // I/O Card 1 I/O19
ptr IoCard1Pt20->u.io:$B00008.12.1           // I/O Card 1 I/O20
ptr IoCard1Pt21->u.io:$B00008.13.1           // I/O Card 1 I/O21
ptr IoCard1Pt22->u.io:$B00008.14.1           // I/O Card 1 I/O22
ptr IoCard1Pt23->u.io:$B00008.15.1           // I/O Card 1 I/O23
ptr IoCard1Pt24->u.io:$B0000C.8.1            // I/O Card 1 I/O24
ptr IoCard1Pt25->u.io:$B0000C.9.1            // I/O Card 1 I/O25
ptr IoCard1Pt26->u.io:$B0000C.10.1           // I/O Card 1 I/O26
ptr IoCard1Pt27->u.io:$B0000C.11.1           // I/O Card 1 I/O27
ptr IoCard1Pt28->u.io:$B0000C.12.1           // I/O Card 1 I/O28
ptr IoCard1Pt29->u.io:$B0000C.13.1           // I/O Card 1 I/O29
ptr IoCard1Pt30->u.io:$B0000C.14.1           // I/O Card 1 I/O30
ptr IoCard1Pt31->u.io:$B0000C.15.1           // I/O Card 1 I/O31
ptr IoCard1Pt32->u.io:$B00010.8.1            // I/O Card 1 I/O32
ptr IoCard1Pt33->u.io:$B00010.9.1            // I/O Card 1 I/O33
ptr IoCard1Pt34->u.io:$B00010.10.1           // I/O Card 1 I/O34
ptr IoCard1Pt35->u.io:$B00010.11.1           // I/O Card 1 I/O35
ptr IoCard1Pt36->u.io:$B00010.12.1           // I/O Card 1 I/O36
ptr IoCard1Pt37->u.io:$B00010.13.1           // I/O Card 1 I/O37
ptr IoCard1Pt38->u.io:$B00010.14.1           // I/O Card 1 I/O38
ptr IoCard1Pt39->u.io:$B00010.15.1           // I/O Card 1 I/O39
ptr IoCard1Pt40->u.io:$B00014.8.1            // I/O Card 1 I/O40
ptr IoCard1Pt41->u.io:$B00014.9.1            // I/O Card 1 I/O41
ptr IoCard1Pt42->u.io:$B00014.10.1           // I/O Card 1 I/O42
ptr IoCard1Pt43->u.io:$B00014.11.1           // I/O Card 1 I/O43
ptr IoCard1Pt44->u.io:$B00014.12.1           // I/O Card 1 I/O44
ptr IoCard1Pt45->u.io:$B00014.13.1           // I/O Card 1 I/O45
ptr IoCard1Pt46->u.io:$B00014.14.1           // I/O Card 1 I/O46
ptr IoCard1Pt47->u.io:$B00014.15.1           // I/O Card 1 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard1Reg0->u.io:$B00000.8.8    // I/O Card 1 I/O00-07 as byte
ptr IoCard1Reg1->u.io:$B00004.8.8    // I/O Card 1 I/O08-15 as byte
ptr IoCard1Reg2->u.io:$B00008.8.8    // I/O Card 1 I/O16-23 as byte
ptr IoCard1Reg3->u.io:$B0000C.8.8    // I/O Card 1 I/O24-31 as byte
ptr IoCard1Reg4->u.io:$B00010.8.8    // I/O Card 1 I/O32-39 as byte
ptr IoCard1Reg5->u.io:$B00014.8.8    // I/O Card 1 I/O40-47 as byte
ptr IoCard1Reg6->u.io:$B00018.8.8    // I/O Card 1 latch inputs
ptr IoCard1Reg7->u.io:$B0001C.8.8    // I/O Card 1 control register
```

Digital I/O Card 2

(SW1-1 ON, SW1-2 OFF, SW1-3 ON, SW1-4 ON)

```
// Single-bit variables used for accessing I/O points
ptr IoCard2Pt00->u.io:$C00000.8.1            // I/O Card 2 I/O00
ptr IoCard2Pt01->u.io:$C00000.9.1            // I/O Card 2 I/O01
ptr IoCard2Pt02->u.io:$C00000.10.1           // I/O Card 2 I/O02
ptr IoCard2Pt03->u.io:$C00000.11.1           // I/O Card 2 I/O03
ptr IoCard2Pt04->u.io:$C00000.12.1           // I/O Card 2 I/O04
ptr IoCard2Pt05->u.io:$C00000.13.1           // I/O Card 2 I/O05
ptr IoCard2Pt06->u.io:$C00000.14.1           // I/O Card 2 I/O06
ptr IoCard2Pt07->u.io:$C00000.15.1           // I/O Card 2 I/O07
ptr IoCard2Pt08->u.io:$C00004.8.1            // I/O Card 2 I/O08
ptr IoCard2Pt09->u.io:$C00004.9.1            // I/O Card 2 I/O09
ptr IoCard2Pt10->u.io:$C00004.10.1           // I/O Card 2 I/O10
ptr IoCard2Pt11->u.io:$C00004.11.1           // I/O Card 2 I/O11
ptr IoCard2Pt12->u.io:$C00004.12.1           // I/O Card 2 I/O12
ptr IoCard2Pt13->u.io:$C00004.13.1           // I/O Card 2 I/O13
ptr IoCard2Pt14->u.io:$C00004.14.1           // I/O Card 2 I/O14
ptr IoCard2Pt15->u.io:$C00004.15.1           // I/O Card 2 I/O15
ptr IoCard2Pt16->u.io:$C00008.8.1            // I/O Card 2 I/O16
ptr IoCard2Pt17->u.io:$C00008.9.1            // I/O Card 2 I/O17
ptr IoCard2Pt18->u.io:$C00008.10.1           // I/O Card 2 I/O18
ptr IoCard2Pt19->u.io:$C00008.11.1           // I/O Card 2 I/O19
ptr IoCard2Pt20->u.io:$C00008.12.1           // I/O Card 2 I/O20
ptr IoCard2Pt21->u.io:$C00008.13.1           // I/O Card 2 I/O21
ptr IoCard2Pt22->u.io:$C00008.14.1           // I/O Card 2 I/O22
ptr IoCard2Pt23->u.io:$C00008.15.1           // I/O Card 2 I/O23
ptr IoCard2Pt24->u.io:$C0000C.8.1            // I/O Card 2 I/O24
ptr IoCard2Pt25->u.io:$C0000C.9.1            // I/O Card 2 I/O25
ptr IoCard2Pt26->u.io:$C0000C.10.1           // I/O Card 2 I/O26
ptr IoCard2Pt27->u.io:$C0000C.11.1           // I/O Card 2 I/O27
ptr IoCard2Pt28->u.io:$C0000C.12.1           // I/O Card 2 I/O28
ptr IoCard2Pt29->u.io:$C0000C.13.1           // I/O Card 2 I/O29
ptr IoCard2Pt30->u.io:$C0000C.14.1           // I/O Card 2 I/O30
ptr IoCard2Pt31->u.io:$C0000C.15.1           // I/O Card 2 I/O31
ptr IoCard2Pt32->u.io:$C00010.8.1            // I/O Card 2 I/O32
ptr IoCard2Pt33->u.io:$C00010.9.1            // I/O Card 2 I/O33
ptr IoCard2Pt34->u.io:$C00010.10.1           // I/O Card 2 I/O34
ptr IoCard2Pt35->u.io:$C00010.11.1           // I/O Card 2 I/O35
ptr IoCard2Pt36->u.io:$C00010.12.1           // I/O Card 2 I/O36
ptr IoCard2Pt37->u.io:$C00010.13.1           // I/O Card 2 I/O37
ptr IoCard2Pt38->u.io:$C00010.14.1           // I/O Card 2 I/O38
ptr IoCard2Pt39->u.io:$C00010.15.1           // I/O Card 2 I/O39
ptr IoCard2Pt40->u.io:$C00014.8.1            // I/O Card 2 I/O40
ptr IoCard2Pt41->u.io:$C00014.9.1            // I/O Card 2 I/O41
ptr IoCard2Pt42->u.io:$C00014.10.1           // I/O Card 2 I/O42
ptr IoCard2Pt43->u.io:$C00014.11.1           // I/O Card 2 I/O43
ptr IoCard2Pt44->u.io:$C00014.12.1           // I/O Card 2 I/O44
ptr IoCard2Pt45->u.io:$C00014.13.1           // I/O Card 2 I/O45
ptr IoCard2Pt46->u.io:$C00014.14.1           // I/O Card 2 I/O46
ptr IoCard2Pt47->u.io:$C00014.15.1           // I/O Card 2 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard2Reg0->u.io:$C00000.8.8    // I/O Card 2 I/O00-07 as byte
ptr IoCard2Reg1->u.io:$C00004.8.8    // I/O Card 2 I/O08-15 as byte
ptr IoCard2Reg2->u.io:$C00008.8.8    // I/O Card 2 I/O16-23 as byte
ptr IoCard2Reg3->u.io:$C0000C.8.8    // I/O Card 2 I/O24-31 as byte
ptr IoCard2Reg4->u.io:$C00010.8.8    // I/O Card 2 I/O32-39 as byte
ptr IoCard2Reg5->u.io:$C00014.8.8    // I/O Card 2 I/O40-47 as byte
ptr IoCard2Reg6->u.io:$C00018.8.8    // I/O Card 2 latch inputs
ptr IoCard2Reg7->u.io:$C0001C.8.8    // I/O Card 2 control register
```

Digital I/O Card 3

(SW1-1 OFF, SW1-2 OFF, SW1-3 ON, SW1-4 ON)

```
// Single-bit variables used for accessing I/O points
ptr IoCard3Pt00->u.io:$D00000.8.1            // I/O Card 3 I/O00
ptr IoCard3Pt01->u.io:$D00000.9.1            // I/O Card 3 I/O01
ptr IoCard3Pt02->u.io:$D00000.10.1           // I/O Card 3 I/O02
ptr IoCard3Pt03->u.io:$D00000.11.1           // I/O Card 3 I/O03
ptr IoCard3Pt04->u.io:$D00000.12.1           // I/O Card 3 I/O04
ptr IoCard3Pt05->u.io:$D00000.13.1           // I/O Card 3 I/O05
ptr IoCard3Pt06->u.io:$D00000.14.1           // I/O Card 3 I/O06
ptr IoCard3Pt07->u.io:$D00000.15.1           // I/O Card 3 I/O07
ptr IoCard3Pt08->u.io:$D00004.8.1            // I/O Card 3 I/O08
ptr IoCard3Pt09->u.io:$D00004.9.1            // I/O Card 3 I/O09
ptr IoCard3Pt10->u.io:$D00004.10.1           // I/O Card 3 I/O10
ptr IoCard3Pt11->u.io:$D00004.11.1           // I/O Card 3 I/O11
ptr IoCard3Pt12->u.io:$D00004.12.1           // I/O Card 3 I/O12
ptr IoCard3Pt13->u.io:$D00004.13.1           // I/O Card 3 I/O13
ptr IoCard3Pt14->u.io:$D00004.14.1           // I/O Card 3 I/O14
ptr IoCard3Pt15->u.io:$D00004.15.1           // I/O Card 3 I/O15
ptr IoCard3Pt16->u.io:$D00008.8.1            // I/O Card 3 I/O16
ptr IoCard3Pt17->u.io:$D00008.9.1            // I/O Card 3 I/O17
ptr IoCard3Pt18->u.io:$D00008.10.1           // I/O Card 3 I/O18
ptr IoCard3Pt19->u.io:$D00008.11.1           // I/O Card 3 I/O19
ptr IoCard3Pt20->u.io:$D00008.12.1           // I/O Card 3 I/O20
ptr IoCard3Pt21->u.io:$D00008.13.1           // I/O Card 3 I/O21
ptr IoCard3Pt22->u.io:$D00008.14.1           // I/O Card 3 I/O22
ptr IoCard3Pt23->u.io:$D00008.15.1           // I/O Card 3 I/O23
ptr IoCard3Pt24->u.io:$D0000C.8.1            // I/O Card 3 I/O24
ptr IoCard3Pt25->u.io:$D0000C.9.1            // I/O Card 3 I/O25
ptr IoCard3Pt26->u.io:$D0000C.10.1           // I/O Card 3 I/O26
ptr IoCard3Pt27->u.io:$D0000C.11.1           // I/O Card 3 I/O27
ptr IoCard3Pt28->u.io:$D0000C.12.1           // I/O Card 3 I/O28
ptr IoCard3Pt29->u.io:$D0000C.13.1           // I/O Card 3 I/O29
ptr IoCard3Pt30->u.io:$D0000C.14.1           // I/O Card 3 I/O30
ptr IoCard3Pt31->u.io:$D0000C.15.1           // I/O Card 3 I/O31
ptr IoCard3Pt32->u.io:$D00010.8.1            // I/O Card 3 I/O32
ptr IoCard3Pt33->u.io:$D00010.9.1            // I/O Card 3 I/O33
ptr IoCard3Pt34->u.io:$D00010.10.1           // I/O Card 3 I/O34
ptr IoCard3Pt35->u.io:$D00010.11.1           // I/O Card 3 I/O35
ptr IoCard3Pt36->u.io:$D00010.12.1           // I/O Card 3 I/O36
ptr IoCard3Pt37->u.io:$D00010.13.1           // I/O Card 3 I/O37
ptr IoCard3Pt38->u.io:$D00010.14.1           // I/O Card 3 I/O38
ptr IoCard3Pt39->u.io:$D00010.15.1           // I/O Card 3 I/O39
ptr IoCard3Pt40->u.io:$D00014.8.1            // I/O Card 3 I/O40
ptr IoCard3Pt41->u.io:$D00014.9.1            // I/O Card 3 I/O41
ptr IoCard3Pt42->u.io:$D00014.10.1           // I/O Card 3 I/O42
ptr IoCard3Pt43->u.io:$D00014.11.1           // I/O Card 3 I/O43
ptr IoCard3Pt44->u.io:$D00014.12.1           // I/O Card 3 I/O44
ptr IoCard3Pt45->u.io:$D00014.13.1           // I/O Card 3 I/O45
ptr IoCard3Pt46->u.io:$D00014.14.1           // I/O Card 3 I/O46
ptr IoCard3Pt47->u.io:$D00014.15.1           // I/O Card 3 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard3Reg0->u.io:$D00000.8.8    // I/O Card 3 I/O00-07 as byte
ptr IoCard3Reg1->u.io:$D00004.8.8    // I/O Card 3 I/O08-15 as byte
ptr IoCard3Reg2->u.io:$D00008.8.8    // I/O Card 3 I/O16-23 as byte
ptr IoCard3Reg3->u.io:$D0000C.8.8    // I/O Card 3 I/O24-31 as byte
ptr IoCard3Reg4->u.io:$D00010.8.8    // I/O Card 3 I/O32-39 as byte
ptr IoCard3Reg5->u.io:$D00014.8.8    // I/O Card 3 I/O40-47 as byte
ptr IoCard3Reg6->u.io:$D00018.8.8    // I/O Card 3 latch inputs
ptr IoCard3Reg7->u.io:$D0001C.8.8    // I/O Card 3 control register
```

Digital I/O Card 4

(SW1-1 ON, SW1-2 ON, SW1-3 OFF, SW1-4 ON)

```
// Single-bit variables used for accessing I/O points
ptr IoCard4Pt00->u.io:$A08000.8.1            // I/O Card 4 I/O00
ptr IoCard4Pt01->u.io:$A08000.9.1            // I/O Card 4 I/O01
ptr IoCard4Pt02->u.io:$A08000.10.1           // I/O Card 4 I/O02
ptr IoCard4Pt03->u.io:$A08000.11.1           // I/O Card 4 I/O03
ptr IoCard4Pt04->u.io:$A08000.12.1           // I/O Card 4 I/O04
ptr IoCard4Pt05->u.io:$A08000.13.1           // I/O Card 4 I/O05
ptr IoCard4Pt06->u.io:$A08000.14.1           // I/O Card 4 I/O06
ptr IoCard4Pt07->u.io:$A08000.15.1           // I/O Card 4 I/O07
ptr IoCard4Pt08->u.io:$A08004.8.1            // I/O Card 4 I/O08
ptr IoCard4Pt09->u.io:$A08004.9.1            // I/O Card 4 I/O09
ptr IoCard4Pt10->u.io:$A08004.10.1           // I/O Card 4 I/O10
ptr IoCard4Pt11->u.io:$A08004.11.1           // I/O Card 4 I/O11
ptr IoCard4Pt12->u.io:$A08004.12.1           // I/O Card 4 I/O12
ptr IoCard4Pt13->u.io:$A08004.13.1           // I/O Card 4 I/O13
ptr IoCard4Pt14->u.io:$A08004.14.1           // I/O Card 4 I/O14
ptr IoCard4Pt15->u.io:$A08004.15.1           // I/O Card 4 I/O15
ptr IoCard4Pt16->u.io:$A08008.8.1            // I/O Card 4 I/O16
ptr IoCard4Pt17->u.io:$A08008.9.1            // I/O Card 4 I/O17
ptr IoCard4Pt18->u.io:$A08008.10.1           // I/O Card 4 I/O18
ptr IoCard4Pt19->u.io:$A08008.11.1           // I/O Card 4 I/O19
ptr IoCard4Pt20->u.io:$A08008.12.1           // I/O Card 4 I/O20
ptr IoCard4Pt21->u.io:$A08008.13.1           // I/O Card 4 I/O21
ptr IoCard4Pt22->u.io:$A08008.14.1           // I/O Card 4 I/O22
ptr IoCard4Pt23->u.io:$A08008.15.1           // I/O Card 4 I/O23
ptr IoCard4Pt24->u.io:$A0800C.8.1            // I/O Card 4 I/O24
ptr IoCard4Pt25->u.io:$A0800C.9.1            // I/O Card 4 I/O25
ptr IoCard4Pt26->u.io:$A0800C.10.1           // I/O Card 4 I/O26
ptr IoCard4Pt27->u.io:$A0800C.11.1           // I/O Card 4 I/O27
ptr IoCard4Pt28->u.io:$A0800C.12.1           // I/O Card 4 I/O28
ptr IoCard4Pt29->u.io:$A0800C.13.1           // I/O Card 4 I/O29
ptr IoCard4Pt30->u.io:$A0800C.14.1           // I/O Card 4 I/O30
ptr IoCard4Pt31->u.io:$A0800C.15.1           // I/O Card 4 I/O31
ptr IoCard4Pt32->u.io:$A08010.8.1            // I/O Card 4 I/O32
ptr IoCard4Pt33->u.io:$A08010.9.1            // I/O Card 4 I/O33
ptr IoCard4Pt34->u.io:$A08010.10.1           // I/O Card 4 I/O34
ptr IoCard4Pt35->u.io:$A08010.11.1           // I/O Card 4 I/O35
ptr IoCard4Pt36->u.io:$A08010.12.1           // I/O Card 4 I/O36
ptr IoCard4Pt37->u.io:$A08010.13.1           // I/O Card 4 I/O37
ptr IoCard4Pt38->u.io:$A08010.14.1           // I/O Card 4 I/O38
ptr IoCard4Pt39->u.io:$A08010.15.1           // I/O Card 4 I/O39
ptr IoCard4Pt40->u.io:$A08014.8.1            // I/O Card 4 I/O40
ptr IoCard4Pt41->u.io:$A08014.9.1            // I/O Card 4 I/O41
ptr IoCard4Pt42->u.io:$A08014.10.1           // I/O Card 4 I/O42
ptr IoCard4Pt43->u.io:$A08014.11.1           // I/O Card 4 I/O43
ptr IoCard4Pt44->u.io:$A08014.12.1           // I/O Card 4 I/O44
ptr IoCard4Pt45->u.io:$A08014.13.1           // I/O Card 4 I/O45
ptr IoCard4Pt46->u.io:$A08014.14.1           // I/O Card 4 I/O46
ptr IoCard4Pt47->u.io:$A08014.15.1           // I/O Card 4 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard4Reg0->u.io:$A08000.8.8     // I/O Card 4 I/O00-07 as byte
ptr IoCard4Reg1->u.io:$A08004.8.8     // I/O Card 4 I/O08-15 as byte
ptr IoCard4Reg2->u.io:$A08008.8.8     // I/O Card 4 I/O16-23 as byte
ptr IoCard4Reg3->u.io:$A0800C.8.8     // I/O Card 4 I/O24-31 as byte
ptr IoCard4Reg4->u.io:$A08010.8.8     // I/O Card 4 I/O32-39 as byte
ptr IoCard4Reg5->u.io:$A08014.8.8     // I/O Card 4 I/O40-47 as byte
ptr IoCard4Reg6->u.io:$A08018.8.8     // I/O Card 4 latch inputs
ptr IoCard4Reg7->u.io:$A0801C.8.8     // I/O Card 4 control register
```

Digital I/O Card 5

(SW1-1 OFF, SW1-2 ON, SW1-3 OFF, SW1-4 ON)

```
// Single-bit variables used for accessing I/O points
ptr IoCard5Pt00->u.io:$B08000.8.1            // I/O Card 5 I/O00
ptr IoCard5Pt01->u.io:$B08000.9.1            // I/O Card 5 I/O01
ptr IoCard5Pt02->u.io:$B08000.10.1           // I/O Card 5 I/O02
ptr IoCard5Pt03->u.io:$B08000.11.1           // I/O Card 5 I/O03
ptr IoCard5Pt04->u.io:$B08000.12.1           // I/O Card 5 I/O04
ptr IoCard5Pt05->u.io:$B08000.13.1           // I/O Card 5 I/O05
ptr IoCard5Pt06->u.io:$B08000.14.1           // I/O Card 5 I/O06
ptr IoCard5Pt07->u.io:$B08000.15.1           // I/O Card 5 I/O07
ptr IoCard5Pt08->u.io:$B08004.8.1            // I/O Card 5 I/O08
ptr IoCard5Pt09->u.io:$B08004.9.1            // I/O Card 5 I/O09
ptr IoCard5Pt10->u.io:$B08004.10.1           // I/O Card 5 I/O10
ptr IoCard5Pt11->u.io:$B08004.11.1           // I/O Card 5 I/O11
ptr IoCard5Pt12->u.io:$B08004.12.1           // I/O Card 5 I/O12
ptr IoCard5Pt13->u.io:$B08004.13.1           // I/O Card 5 I/O13
ptr IoCard5Pt14->u.io:$B08004.14.1           // I/O Card 5 I/O14
ptr IoCard5Pt15->u.io:$B08004.15.1           // I/O Card 5 I/O15
ptr IoCard5Pt16->u.io:$B08008.8.1            // I/O Card 5 I/O16
ptr IoCard5Pt17->u.io:$B08008.9.1            // I/O Card 5 I/O17
ptr IoCard5Pt18->u.io:$B08008.10.1           // I/O Card 5 I/O18
ptr IoCard5Pt19->u.io:$B08008.11.1           // I/O Card 5 I/O19
ptr IoCard5Pt20->u.io:$B08008.12.1           // I/O Card 5 I/O20
ptr IoCard5Pt21->u.io:$B08008.13.1           // I/O Card 5 I/O21
ptr IoCard5Pt22->u.io:$B08008.14.1           // I/O Card 5 I/O22
ptr IoCard5Pt23->u.io:$B08008.15.1           // I/O Card 5 I/O23
ptr IoCard5Pt24->u.io:$B0800C.8.1            // I/O Card 5 I/O24
ptr IoCard5Pt25->u.io:$B0800C.9.1            // I/O Card 5 I/O25
ptr IoCard5Pt26->u.io:$B0800C.10.1           // I/O Card 5 I/O26
ptr IoCard5Pt27->u.io:$B0800C.11.1           // I/O Card 5 I/O27
ptr IoCard5Pt28->u.io:$B0800C.12.1           // I/O Card 5 I/O28
ptr IoCard5Pt29->u.io:$B0800C.13.1           // I/O Card 5 I/O29
ptr IoCard5Pt30->u.io:$B0800C.14.1           // I/O Card 5 I/O30
ptr IoCard5Pt31->u.io:$B0800C.15.1           // I/O Card 5 I/O31
ptr IoCard5Pt32->u.io:$B08010.8.1            // I/O Card 5 I/O32
ptr IoCard5Pt33->u.io:$B08010.9.1            // I/O Card 5 I/O33
ptr IoCard5Pt34->u.io:$B08010.10.1           // I/O Card 5 I/O34
ptr IoCard5Pt35->u.io:$B08010.11.1           // I/O Card 5 I/O35
ptr IoCard5Pt36->u.io:$B08010.12.1           // I/O Card 5 I/O36
ptr IoCard5Pt37->u.io:$B08010.13.1           // I/O Card 5 I/O37
ptr IoCard5Pt38->u.io:$B08010.14.1           // I/O Card 5 I/O38
ptr IoCard5Pt39->u.io:$B08010.15.1           // I/O Card 5 I/O39
ptr IoCard5Pt40->u.io:$B08014.8.1            // I/O Card 5 I/O40
ptr IoCard5Pt41->u.io:$B08014.9.1            // I/O Card 5 I/O41
ptr IoCard5Pt42->u.io:$B08014.10.1           // I/O Card 5 I/O42
ptr IoCard5Pt43->u.io:$B08014.11.1           // I/O Card 5 I/O43
ptr IoCard5Pt44->u.io:$B08014.12.1           // I/O Card 5 I/O44
ptr IoCard5Pt45->u.io:$B08014.13.1           // I/O Card 5 I/O45
ptr IoCard5Pt46->u.io:$B08014.14.1           // I/O Card 5 I/O46
ptr IoCard5Pt47->u.io:$B08014.15.1           // I/O Card 5 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard5Reg0->u.io:$B08000.8.8    // I/O Card 5 I/O00-07 as byte
ptr IoCard5Reg1->u.io:$B08004.8.8    // I/O Card 5 I/O08-15 as byte
ptr IoCard5Reg2->u.io:$B08008.8.8    // I/O Card 5 I/O16-23 as byte
ptr IoCard5Reg3->u.io:$B0800C.8.8    // I/O Card 5 I/O24-31 as byte
ptr IoCard5Reg4->u.io:$B08010.8.8    // I/O Card 5 I/O32-39 as byte
ptr IoCard5Reg5->u.io:$B08014.8.8    // I/O Card 5 I/O40-47 as byte
ptr IoCard5Reg6->u.io:$B08018.8.8    // I/O Card 5 latch inputs
ptr IoCard5Reg7->u.io:$B0801C.8.8    // I/O Card 5 control register
```

Digital I/O Card 6

(SW1-1 ON, SW1-2 OFF, SW1-3 OFF, SW1-4 ON)

```
// Single-bit variables used for accessing I/O points
ptr IoCard6Pt00->u.io:$C08000.8.1            // I/O Card 6 I/O00
ptr IoCard6Pt01->u.io:$C08000.9.1            // I/O Card 6 I/O01
ptr IoCard6Pt02->u.io:$C08000.10.1           // I/O Card 6 I/O02
ptr IoCard6Pt03->u.io:$C08000.11.1           // I/O Card 6 I/O03
ptr IoCard6Pt04->u.io:$C08000.12.1           // I/O Card 6 I/O04
ptr IoCard6Pt05->u.io:$C08000.13.1           // I/O Card 6 I/O05
ptr IoCard6Pt06->u.io:$C08000.14.1           // I/O Card 6 I/O06
ptr IoCard6Pt07->u.io:$C08000.15.1           // I/O Card 6 I/O07
ptr IoCard6Pt08->u.io:$C08004.8.1            // I/O Card 6 I/O08
ptr IoCard6Pt09->u.io:$C08004.9.1            // I/O Card 6 I/O09
ptr IoCard6Pt10->u.io:$C08004.10.1           // I/O Card 6 I/O10
ptr IoCard6Pt11->u.io:$C08004.11.1           // I/O Card 6 I/O11
ptr IoCard6Pt12->u.io:$C08004.12.1           // I/O Card 6 I/O12
ptr IoCard6Pt13->u.io:$C08004.13.1           // I/O Card 6 I/O13
ptr IoCard6Pt14->u.io:$C08004.14.1           // I/O Card 6 I/O14
ptr IoCard6Pt15->u.io:$C08004.15.1           // I/O Card 6 I/O15
ptr IoCard6Pt16->u.io:$C08008.8.1            // I/O Card 6 I/O16
ptr IoCard6Pt17->u.io:$C08008.9.1            // I/O Card 6 I/O17
ptr IoCard6Pt18->u.io:$C08008.10.1           // I/O Card 6 I/O18
ptr IoCard6Pt19->u.io:$C08008.11.1           // I/O Card 6 I/O19
ptr IoCard6Pt20->u.io:$C08008.12.1           // I/O Card 6 I/O20
ptr IoCard6Pt21->u.io:$C08008.13.1           // I/O Card 6 I/O21
ptr IoCard6Pt22->u.io:$C08008.14.1           // I/O Card 6 I/O22
ptr IoCard6Pt23->u.io:$C08008.15.1           // I/O Card 6 I/O23
ptr IoCard6Pt24->u.io:$C0800C.8.1            // I/O Card 6 I/O24
ptr IoCard6Pt25->u.io:$C0800C.9.1            // I/O Card 6 I/O25
ptr IoCard6Pt26->u.io:$C0800C.10.1           // I/O Card 6 I/O26
ptr IoCard6Pt27->u.io:$C0800C.11.1           // I/O Card 6 I/O27
ptr IoCard6Pt28->u.io:$C0800C.12.1           // I/O Card 6 I/O28
ptr IoCard6Pt29->u.io:$C0800C.13.1           // I/O Card 6 I/O29
ptr IoCard6Pt30->u.io:$C0800C.14.1           // I/O Card 6 I/O30
ptr IoCard6Pt31->u.io:$C0800C.15.1           // I/O Card 6 I/O31
ptr IoCard6Pt32->u.io:$C08010.8.1            // I/O Card 6 I/O32
ptr IoCard6Pt33->u.io:$C08010.9.1            // I/O Card 6 I/O33
ptr IoCard6Pt34->u.io:$C08010.10.1           // I/O Card 6 I/O34
ptr IoCard6Pt35->u.io:$C08010.11.1           // I/O Card 6 I/O35
ptr IoCard6Pt36->u.io:$C08010.12.1           // I/O Card 6 I/O36
ptr IoCard6Pt37->u.io:$C08010.13.1           // I/O Card 6 I/O37
ptr IoCard6Pt38->u.io:$C08010.14.1           // I/O Card 6 I/O38
ptr IoCard6Pt39->u.io:$C08010.15.1           // I/O Card 6 I/O39
ptr IoCard6Pt40->u.io:$C08014.8.1            // I/O Card 6 I/O40
ptr IoCard6Pt41->u.io:$C08014.9.1            // I/O Card 6 I/O41
ptr IoCard6Pt42->u.io:$C08014.10.1           // I/O Card 6 I/O42
ptr IoCard6Pt43->u.io:$C08014.11.1           // I/O Card 6 I/O43
ptr IoCard6Pt44->u.io:$C08014.12.1           // I/O Card 6 I/O44
ptr IoCard6Pt45->u.io:$C08014.13.1           // I/O Card 6 I/O45
ptr IoCard6Pt46->u.io:$C08014.14.1           // I/O Card 6 I/O46
ptr IoCard6Pt47->u.io:$C08014.15.1           // I/O Card 6 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard6Reg0->u.io:$C08000.8.8     // I/O Card 6 I/O00-07 as byte
ptr IoCard6Reg1->u.io:$C08004.8.8     // I/O Card 6 I/O08-15 as byte
ptr IoCard6Reg2->u.io:$C08008.8.8     // I/O Card 6 I/O16-23 as byte
ptr IoCard6Reg3->u.io:$C0800C.8.8     // I/O Card 6 I/O24-31 as byte
ptr IoCard6Reg4->u.io:$C08010.8.8     // I/O Card 6 I/O32-39 as byte
ptr IoCard6Reg5->u.io:$C08014.8.8     // I/O Card 6 I/O40-47 as byte
ptr IoCard6Reg6->u.io:$C08018.8.8     // I/O Card 6 latch inputs
ptr IoCard6Reg7->u.io:$C0801C.8.8     // I/O Card 6 control register
```

Digital I/O Card 7

(SW1-1 OFF, SW1-2 OFF, SW1-3 OFF, SW1-4 ON)

```
// Single-bit variables used for accessing I/O points
ptr IoCard7Pt00->u.io:$D08000.8.1            // I/O Card 7 I/O00
ptr IoCard7Pt01->u.io:$D08000.9.1            // I/O Card 7 I/O01
ptr IoCard7Pt02->u.io:$D08000.10.1           // I/O Card 7 I/O02
ptr IoCard7Pt03->u.io:$D08000.11.1           // I/O Card 7 I/O03
ptr IoCard7Pt04->u.io:$D08000.12.1           // I/O Card 7 I/O04
ptr IoCard7Pt05->u.io:$D08000.13.1           // I/O Card 7 I/O05
ptr IoCard7Pt06->u.io:$D08000.14.1           // I/O Card 7 I/O06
ptr IoCard7Pt07->u.io:$D08000.15.1           // I/O Card 7 I/O07
ptr IoCard7Pt08->u.io:$D08004.8.1            // I/O Card 7 I/O08
ptr IoCard7Pt09->u.io:$D08004.9.1            // I/O Card 7 I/O09
ptr IoCard7Pt10->u.io:$D08004.10.1           // I/O Card 7 I/O10
ptr IoCard7Pt11->u.io:$D08004.11.1           // I/O Card 7 I/O11
ptr IoCard7Pt12->u.io:$D08004.12.1           // I/O Card 7 I/O12
ptr IoCard7Pt13->u.io:$D08004.13.1           // I/O Card 7 I/O13
ptr IoCard7Pt14->u.io:$D08004.14.1           // I/O Card 7 I/O14
ptr IoCard7Pt15->u.io:$D08004.15.1           // I/O Card 7 I/O15
ptr IoCard7Pt16->u.io:$D08008.8.1            // I/O Card 7 I/O16
ptr IoCard7Pt17->u.io:$D08008.9.1            // I/O Card 7 I/O17
ptr IoCard7Pt18->u.io:$D08008.10.1           // I/O Card 7 I/O18
ptr IoCard7Pt19->u.io:$D08008.11.1           // I/O Card 7 I/O19
ptr IoCard7Pt20->u.io:$D08008.12.1           // I/O Card 7 I/O20
ptr IoCard7Pt21->u.io:$D08008.13.1           // I/O Card 7 I/O21
ptr IoCard7Pt22->u.io:$D08008.14.1           // I/O Card 7 I/O22
ptr IoCard7Pt23->u.io:$D08008.15.1           // I/O Card 7 I/O23
ptr IoCard7Pt24->u.io:$D0800C.8.1            // I/O Card 7 I/O24
ptr IoCard7Pt25->u.io:$D0800C.9.1            // I/O Card 7 I/O25
ptr IoCard7Pt26->u.io:$D0800C.10.1           // I/O Card 7 I/O26
ptr IoCard7Pt27->u.io:$D0800C.11.1           // I/O Card 7 I/O27
ptr IoCard7Pt28->u.io:$D0800C.12.1           // I/O Card 7 I/O28
ptr IoCard7Pt29->u.io:$D0800C.13.1           // I/O Card 7 I/O29
ptr IoCard7Pt30->u.io:$D0800C.14.1           // I/O Card 7 I/O30
ptr IoCard7Pt31->u.io:$D0800C.15.1           // I/O Card 7 I/O31
ptr IoCard7Pt32->u.io:$D08010.8.1            // I/O Card 7 I/O32
ptr IoCard7Pt33->u.io:$D08010.9.1            // I/O Card 7 I/O33
ptr IoCard7Pt34->u.io:$D08010.10.1           // I/O Card 7 I/O34
ptr IoCard7Pt35->u.io:$D08010.11.1           // I/O Card 7 I/O35
ptr IoCard7Pt36->u.io:$D08010.12.1           // I/O Card 7 I/O36
ptr IoCard7Pt37->u.io:$D08010.13.1           // I/O Card 7 I/O37
ptr IoCard7Pt38->u.io:$D08010.14.1           // I/O Card 7 I/O38
ptr IoCard7Pt39->u.io:$D08010.15.1           // I/O Card 7 I/O39
ptr IoCard7Pt40->u.io:$D08014.8.1            // I/O Card 7 I/O40
ptr IoCard7Pt41->u.io:$D08014.9.1            // I/O Card 7 I/O41
ptr IoCard7Pt42->u.io:$D08014.10.1           // I/O Card 7 I/O42
ptr IoCard7Pt43->u.io:$D08014.11.1           // I/O Card 7 I/O43
ptr IoCard7Pt44->u.io:$D08014.12.1           // I/O Card 7 I/O44
ptr IoCard7Pt45->u.io:$D08014.13.1           // I/O Card 7 I/O45
ptr IoCard7Pt46->u.io:$D08014.14.1           // I/O Card 7 I/O46
ptr IoCard7Pt47->u.io:$D08014.15.1           // I/O Card 7 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard7Reg0->u.io:$D08000.8.8     // I/O Card 7 I/O00-07 as byte
ptr IoCard7Reg1->u.io:$D08004.8.8     // I/O Card 7 I/O08-15 as byte
ptr IoCard7Reg2->u.io:$D08008.8.8     // I/O Card 7 I/O16-23 as byte
ptr IoCard7Reg3->u.io:$D0800C.8.8     // I/O Card 7 I/O24-31 as byte
ptr IoCard7Reg4->u.io:$D08010.8.8     // I/O Card 7 I/O32-39 as byte
ptr IoCard7Reg5->u.io:$D08014.8.8     // I/O Card 7 I/O40-47 as byte
ptr IoCard7Reg6->u.io:$D08018.8.8     // I/O Card 7 latch inputs
ptr IoCard7Reg7->u.io:$D0801C.8.8     // I/O Card 7 control register
```

Digital I/O Card 8

(SW1-1 ON, SW1-2 ON, SW1-3 ON, SW1-4 OFF)

```
// Single-bit variables used for accessing I/O points
ptr IoCard8Pt00->u.io:$A10000.8.1            // I/O Card 8 I/O00
ptr IoCard8Pt01->u.io:$A10000.9.1            // I/O Card 8 I/O01
ptr IoCard8Pt02->u.io:$A10000.10.1           // I/O Card 8 I/O02
ptr IoCard8Pt03->u.io:$A10000.11.1           // I/O Card 8 I/O03
ptr IoCard8Pt04->u.io:$A10000.12.1           // I/O Card 8 I/O04
ptr IoCard8Pt05->u.io:$A10000.13.1           // I/O Card 8 I/O05
ptr IoCard8Pt06->u.io:$A10000.14.1           // I/O Card 8 I/O06
ptr IoCard8Pt07->u.io:$A10000.15.1           // I/O Card 8 I/O07
ptr IoCard8Pt08->u.io:$A10004.8.1            // I/O Card 8 I/O08
ptr IoCard8Pt09->u.io:$A10004.9.1            // I/O Card 8 I/O09
ptr IoCard8Pt10->u.io:$A10004.10.1           // I/O Card 8 I/O10
ptr IoCard8Pt11->u.io:$A10004.11.1           // I/O Card 8 I/O11
ptr IoCard8Pt12->u.io:$A10004.12.1           // I/O Card 8 I/O12
ptr IoCard8Pt13->u.io:$A10004.13.1           // I/O Card 8 I/O13
ptr IoCard8Pt14->u.io:$A10004.14.1           // I/O Card 8 I/O14
ptr IoCard8Pt15->u.io:$A10004.15.1           // I/O Card 8 I/O15
ptr IoCard8Pt16->u.io:$A10008.8.1            // I/O Card 8 I/O16
ptr IoCard8Pt17->u.io:$A10008.9.1            // I/O Card 8 I/O17
ptr IoCard8Pt18->u.io:$A10008.10.1           // I/O Card 8 I/O18
ptr IoCard8Pt19->u.io:$A10008.11.1           // I/O Card 8 I/O19
ptr IoCard8Pt20->u.io:$A10008.12.1           // I/O Card 8 I/O20
ptr IoCard8Pt21->u.io:$A10008.13.1           // I/O Card 8 I/O21
ptr IoCard8Pt22->u.io:$A10008.14.1           // I/O Card 8 I/O22
ptr IoCard8Pt23->u.io:$A10008.15.1           // I/O Card 8 I/O23
ptr IoCard8Pt24->u.io:$A1000C.8.1            // I/O Card 8 I/O24
ptr IoCard8Pt25->u.io:$A1000C.9.1            // I/O Card 8 I/O25
ptr IoCard8Pt26->u.io:$A1000C.10.1           // I/O Card 8 I/O26
ptr IoCard8Pt27->u.io:$A1000C.11.1           // I/O Card 8 I/O27
ptr IoCard8Pt28->u.io:$A1000C.12.1           // I/O Card 8 I/O28
ptr IoCard8Pt29->u.io:$A1000C.13.1           // I/O Card 8 I/O29
ptr IoCard8Pt30->u.io:$A1000C.14.1           // I/O Card 8 I/O30
ptr IoCard8Pt31->u.io:$A1000C.15.1           // I/O Card 8 I/O31
ptr IoCard8Pt32->u.io:$A10010.8.1            // I/O Card 8 I/O32
ptr IoCard8Pt33->u.io:$A10010.9.1            // I/O Card 8 I/O33
ptr IoCard8Pt34->u.io:$A10010.10.1           // I/O Card 8 I/O34
ptr IoCard8Pt35->u.io:$A10010.11.1           // I/O Card 8 I/O35
ptr IoCard8Pt36->u.io:$A10010.12.1           // I/O Card 8 I/O36
ptr IoCard8Pt37->u.io:$A10010.13.1           // I/O Card 8 I/O37
ptr IoCard8Pt38->u.io:$A10010.14.1           // I/O Card 8 I/O38
ptr IoCard8Pt39->u.io:$A10010.15.1           // I/O Card 8 I/O39
ptr IoCard8Pt40->u.io:$A10014.8.1            // I/O Card 8 I/O40
ptr IoCard8Pt41->u.io:$A10014.9.1            // I/O Card 8 I/O41
ptr IoCard8Pt42->u.io:$A10014.10.1           // I/O Card 8 I/O42
ptr IoCard8Pt43->u.io:$A10014.11.1           // I/O Card 8 I/O43
ptr IoCard8Pt44->u.io:$A10014.12.1           // I/O Card 8 I/O44
ptr IoCard8Pt45->u.io:$A10014.13.1           // I/O Card 8 I/O45
ptr IoCard8Pt46->u.io:$A10014.14.1           // I/O Card 8 I/O46
ptr IoCard8Pt47->u.io:$A10014.15.1           // I/O Card 8 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard8Reg0->u.io:$A10000.8.8     // I/O Card 8 I/O00-07 as byte
ptr IoCard8Reg1->u.io:$A10004.8.8     // I/O Card 8 I/O08-15 as byte
ptr IoCard8Reg2->u.io:$A10008.8.8     // I/O Card 8 I/O16-23 as byte
ptr IoCard8Reg3->u.io:$A1000C.8.8     // I/O Card 8 I/O24-31 as byte
ptr IoCard8Reg4->u.io:$A10010.8.8     // I/O Card 8 I/O32-39 as byte
ptr IoCard8Reg5->u.io:$A10014.8.8     // I/O Card 8 I/O40-47 as byte
ptr IoCard8Reg6->u.io:$A10018.8.8     // I/O Card 8 latch inputs
ptr IoCard8Reg7->u.io:$A1001C.8.8     // I/O Card 8 control register
```

Digital I/O Card 9

(SW1-1 OFF, SW1-2 ON, SW1-3 ON, SW1-4 OFF)

```
// Single-bit variables used for accessing I/O points
ptr IoCard9Pt00->u.io:$B10000.8.1            // I/O Card 9 I/O00
ptr IoCard9Pt01->u.io:$B10000.9.1            // I/O Card 9 I/O01
ptr IoCard9Pt02->u.io:$B10000.10.1           // I/O Card 9 I/O02
ptr IoCard9Pt03->u.io:$B10000.11.1           // I/O Card 9 I/O03
ptr IoCard9Pt04->u.io:$B10000.12.1           // I/O Card 9 I/O04
ptr IoCard9Pt05->u.io:$B10000.13.1           // I/O Card 9 I/O05
ptr IoCard9Pt06->u.io:$B10000.14.1           // I/O Card 9 I/O06
ptr IoCard9Pt07->u.io:$B10000.15.1           // I/O Card 9 I/O07
ptr IoCard9Pt08->u.io:$B10004.8.1            // I/O Card 9 I/O08
ptr IoCard9Pt09->u.io:$B10004.9.1            // I/O Card 9 I/O09
ptr IoCard9Pt10->u.io:$B10004.10.1           // I/O Card 9 I/O10
ptr IoCard9Pt11->u.io:$B10004.11.1           // I/O Card 9 I/O11
ptr IoCard9Pt12->u.io:$B10004.12.1           // I/O Card 9 I/O12
ptr IoCard9Pt13->u.io:$B10004.13.1           // I/O Card 9 I/O13
ptr IoCard9Pt14->u.io:$B10004.14.1           // I/O Card 9 I/O14
ptr IoCard9Pt15->u.io:$B10004.15.1           // I/O Card 9 I/O15
ptr IoCard9Pt16->u.io:$B10008.8.1            // I/O Card 9 I/O16
ptr IoCard9Pt17->u.io:$B10008.9.1            // I/O Card 9 I/O17
ptr IoCard9Pt18->u.io:$B10008.10.1           // I/O Card 9 I/O18
ptr IoCard9Pt19->u.io:$B10008.11.1           // I/O Card 9 I/O19
ptr IoCard9Pt20->u.io:$B10008.12.1           // I/O Card 9 I/O20
ptr IoCard9Pt21->u.io:$B10008.13.1           // I/O Card 9 I/O21
ptr IoCard9Pt22->u.io:$B10008.14.1           // I/O Card 9 I/O22
ptr IoCard9Pt23->u.io:$B10008.15.1           // I/O Card 9 I/O23
ptr IoCard9Pt24->u.io:$B1000C.8.1            // I/O Card 9 I/O24
ptr IoCard9Pt25->u.io:$B1000C.9.1            // I/O Card 9 I/O25
ptr IoCard9Pt26->u.io:$B1000C.10.1           // I/O Card 9 I/O26
ptr IoCard9Pt27->u.io:$B1000C.11.1           // I/O Card 9 I/O27
ptr IoCard9Pt28->u.io:$B1000C.12.1           // I/O Card 9 I/O28
ptr IoCard9Pt29->u.io:$B1000C.13.1           // I/O Card 9 I/O29
ptr IoCard9Pt30->u.io:$B1000C.14.1           // I/O Card 9 I/O30
ptr IoCard9Pt31->u.io:$B1000C.15.1           // I/O Card 9 I/O31
ptr IoCard9Pt32->u.io:$B10010.8.1            // I/O Card 9 I/O32
ptr IoCard9Pt33->u.io:$B10010.9.1            // I/O Card 9 I/O33
ptr IoCard9Pt34->u.io:$B10010.10.1           // I/O Card 9 I/O34
ptr IoCard9Pt35->u.io:$B10010.11.1           // I/O Card 9 I/O35
ptr IoCard9Pt36->u.io:$B10010.12.1           // I/O Card 9 I/O36
ptr IoCard9Pt37->u.io:$B10010.13.1           // I/O Card 9 I/O37
ptr IoCard9Pt38->u.io:$B10010.14.1           // I/O Card 9 I/O38
ptr IoCard9Pt39->u.io:$B10010.15.1           // I/O Card 9 I/O39
ptr IoCard9Pt40->u.io:$B10014.8.1            // I/O Card 9 I/O40
ptr IoCard9Pt41->u.io:$B10014.9.1            // I/O Card 9 I/O41
ptr IoCard9Pt42->u.io:$B10014.10.1           // I/O Card 9 I/O42
ptr IoCard9Pt43->u.io:$B10014.11.1           // I/O Card 9 I/O43
ptr IoCard9Pt44->u.io:$B10014.12.1           // I/O Card 9 I/O44
ptr IoCard9Pt45->u.io:$B10014.13.1           // I/O Card 9 I/O45
ptr IoCard9Pt46->u.io:$B10014.14.1           // I/O Card 9 I/O46
ptr IoCard9Pt47->u.io:$B10014.15.1           // I/O Card 9 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard9Reg0->u.io:$B10000.8.8     // I/O Card 9 I/O00-07 as byte
ptr IoCard9Reg1->u.io:$B10004.8.8     // I/O Card 9 I/O08-15 as byte
ptr IoCard9Reg2->u.io:$B10008.8.8     // I/O Card 9 I/O16-23 as byte
ptr IoCard9Reg3->u.io:$B1000C.8.8     // I/O Card 9 I/O24-31 as byte
ptr IoCard9Reg4->u.io:$B10010.8.8     // I/O Card 9 I/O32-39 as byte
ptr IoCard9Reg5->u.io:$B10014.8.8     // I/O Card 9 I/O40-47 as byte
ptr IoCard9Reg6->u.io:$B10018.8.8     // I/O Card 9 latch inputs
ptr IoCard9Reg7->u.io:$B1001C.8.8     // I/O Card 9 control register
```

Digital I/O Card 10

(SW1-1 ON, SW1-2 OFF, SW1-3 ON, SW1-4 OFF)

```
// Single-bit variables used for accessing I/O points
ptr IoCard10Pt00->u.io:$C10000.8.1           // I/O Card 10 I/O00
ptr IoCard10Pt01->u.io:$C10000.9.1           // I/O Card 10 I/O01
ptr IoCard10Pt02->u.io:$C10000.10.1          // I/O Card 10 I/O02
ptr IoCard10Pt03->u.io:$C10000.11.1          // I/O Card 10 I/O03
ptr IoCard10Pt04->u.io:$C10000.12.1          // I/O Card 10 I/O04
ptr IoCard10Pt05->u.io:$C10000.13.1          // I/O Card 10 I/O05
ptr IoCard10Pt06->u.io:$C10000.14.1          // I/O Card 10 I/O06
ptr IoCard10Pt07->u.io:$C10000.15.1          // I/O Card 10 I/O07
ptr IoCard10Pt08->u.io:$C10004.8.1           // I/O Card 10 I/O08
ptr IoCard10Pt09->u.io:$C10004.9.1           // I/O Card 10 I/O09
ptr IoCard10Pt10->u.io:$C10004.10.1          // I/O Card 10 I/O10
ptr IoCard10Pt11->u.io:$C10004.11.1          // I/O Card 10 I/O11
ptr IoCard10Pt12->u.io:$C10004.12.1          // I/O Card 10 I/O12
ptr IoCard10Pt13->u.io:$C10004.13.1          // I/O Card 10 I/O13
ptr IoCard10Pt14->u.io:$C10004.14.1          // I/O Card 10 I/O14
ptr IoCard10Pt15->u.io:$C10004.15.1          // I/O Card 10 I/O15
ptr IoCard10Pt16->u.io:$C10008.8.1           // I/O Card 10 I/O16
ptr IoCard10Pt17->u.io:$C10008.9.1           // I/O Card 10 I/O17
ptr IoCard10Pt18->u.io:$C10008.10.1          // I/O Card 10 I/O18
ptr IoCard10Pt19->u.io:$C10008.11.1          // I/O Card 10 I/O19
ptr IoCard10Pt20->u.io:$C10008.12.1          // I/O Card 10 I/O20
ptr IoCard10Pt21->u.io:$C10008.13.1          // I/O Card 10 I/O21
ptr IoCard10Pt22->u.io:$C10008.14.1          // I/O Card 10 I/O22
ptr IoCard10Pt23->u.io:$C10008.15.1          // I/O Card 10 I/O23
ptr IoCard10Pt24->u.io:$C1000C.8.1           // I/O Card 10 I/O24
ptr IoCard10Pt25->u.io:$C1000C.9.1           // I/O Card 10 I/O25
ptr IoCard10Pt26->u.io:$C1000C.10.1          // I/O Card 10 I/O26
ptr IoCard10Pt27->u.io:$C1000C.11.1          // I/O Card 10 I/O27
ptr IoCard10Pt28->u.io:$C1000C.12.1          // I/O Card 10 I/O28
ptr IoCard10Pt29->u.io:$C1000C.13.1          // I/O Card 10 I/O29
ptr IoCard10Pt30->u.io:$C1000C.14.1          // I/O Card 10 I/O30
ptr IoCard10Pt31->u.io:$C1000C.15.1          // I/O Card 10 I/O31
ptr IoCard10Pt32->u.io:$C10010.8.1           // I/O Card 10 I/O32
ptr IoCard10Pt33->u.io:$C10010.9.1           // I/O Card 10 I/O33
ptr IoCard10Pt34->u.io:$C10010.10.1          // I/O Card 10 I/O34
ptr IoCard10Pt35->u.io:$C10010.11.1          // I/O Card 10 I/O35
ptr IoCard10Pt36->u.io:$C10010.12.1          // I/O Card 10 I/O36
ptr IoCard10Pt37->u.io:$C10010.13.1          // I/O Card 10 I/O37
ptr IoCard10Pt38->u.io:$C10010.14.1          // I/O Card 10 I/O38
ptr IoCard10Pt39->u.io:$C10010.15.1          // I/O Card 10 I/O39
ptr IoCard10Pt40->u.io:$C10014.8.1           // I/O Card 10 I/O40
ptr IoCard10Pt41->u.io:$C10014.9.1           // I/O Card 10 I/O41
ptr IoCard10Pt42->u.io:$C10014.10.1          // I/O Card 10 I/O42
ptr IoCard10Pt43->u.io:$C10014.11.1          // I/O Card 10 I/O43
ptr IoCard10Pt44->u.io:$C10014.12.1          // I/O Card 10 I/O44
ptr IoCard10Pt45->u.io:$C10014.13.1          // I/O Card 10 I/O45
ptr IoCard10Pt46->u.io:$C10014.14.1          // I/O Card 10 I/O46
ptr IoCard10Pt47->u.io:$C10014.15.1          // I/O Card 10 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard10Reg0->u.io:$C10000.8.8    // I/O Card 10 I/O00-07 as byte
ptr IoCard10Reg1->u.io:$C10004.8.8    // I/O Card 10 I/O08-15 as byte
ptr IoCard10Reg2->u.io:$C10008.8.8    // I/O Card 10 I/O16-23 as byte
ptr IoCard10Reg3->u.io:$C1000C.8.8    // I/O Card 10 I/O24-31 as byte
ptr IoCard10Reg4->u.io:$C10010.8.8    // I/O Card 10 I/O32-39 as byte
ptr IoCard10Reg5->u.io:$C10014.8.8    // I/O Card 10 I/O40-47 as byte
ptr IoCard10Reg6->u.io:$C10018.8.8    // I/O Card 10 latch inputs
ptr IoCard10Reg7->u.io:$C1001C.8.8    // I/O Card 10 control register
```

Digital I/O Card 11

(SW1-1 OFF, SW1-2 OFF, SW1-3 ON, SW1-4 OFF)

```
// Single-bit variables used for accessing I/O points
ptr IoCard11Pt00->u.io:$D10000.8.1           // I/O Card 11 I/O00
ptr IoCard11Pt01->u.io:$D10000.9.1           // I/O Card 11 I/O01
ptr IoCard11Pt02->u.io:$D10000.10.1          // I/O Card 11 I/O02
ptr IoCard11Pt03->u.io:$D10000.11.1          // I/O Card 11 I/O03
ptr IoCard11Pt04->u.io:$D10000.12.1          // I/O Card 11 I/O04
ptr IoCard11Pt05->u.io:$D10000.13.1          // I/O Card 11 I/O05
ptr IoCard11Pt06->u.io:$D10000.14.1          // I/O Card 11 I/O06
ptr IoCard11Pt07->u.io:$D10000.15.1          // I/O Card 11 I/O07
ptr IoCard11Pt08->u.io:$D10004.8.1           // I/O Card 11 I/O08
ptr IoCard11Pt09->u.io:$D10004.9.1           // I/O Card 11 I/O09
ptr IoCard11Pt10->u.io:$D10004.10.1          // I/O Card 11 I/O10
ptr IoCard11Pt11->u.io:$D10004.11.1          // I/O Card 11 I/O11
ptr IoCard11Pt12->u.io:$D10004.12.1          // I/O Card 11 I/O12
ptr IoCard11Pt13->u.io:$D10004.13.1          // I/O Card 11 I/O13
ptr IoCard11Pt14->u.io:$D10004.14.1          // I/O Card 11 I/O14
ptr IoCard11Pt15->u.io:$D10004.15.1          // I/O Card 11 I/O15
ptr IoCard11Pt16->u.io:$D10008.8.1           // I/O Card 11 I/O16
ptr IoCard11Pt17->u.io:$D10008.9.1           // I/O Card 11 I/O17
ptr IoCard11Pt18->u.io:$D10008.10.1          // I/O Card 11 I/O18
ptr IoCard11Pt19->u.io:$D10008.11.1          // I/O Card 11 I/O19
ptr IoCard11Pt20->u.io:$D10008.12.1          // I/O Card 11 I/O20
ptr IoCard11Pt21->u.io:$D10008.13.1          // I/O Card 11 I/O21
ptr IoCard11Pt22->u.io:$D10008.14.1          // I/O Card 11 I/O22
ptr IoCard11Pt23->u.io:$D10008.15.1          // I/O Card 11 I/O23
ptr IoCard11Pt24->u.io:$D1000C.8.1           // I/O Card 11 I/O24
ptr IoCard11Pt25->u.io:$D1000C.9.1           // I/O Card 11 I/O25
ptr IoCard11Pt26->u.io:$D1000C.10.1          // I/O Card 11 I/O26
ptr IoCard11Pt27->u.io:$D1000C.11.1          // I/O Card 11 I/O27
ptr IoCard11Pt28->u.io:$D1000C.12.1          // I/O Card 11 I/O28
ptr IoCard11Pt29->u.io:$D1000C.13.1          // I/O Card 11 I/O29
ptr IoCard11Pt30->u.io:$D1000C.14.1          // I/O Card 11 I/O30
ptr IoCard11Pt31->u.io:$D1000C.15.1          // I/O Card 11 I/O31
ptr IoCard11Pt32->u.io:$D10010.8.1           // I/O Card 11 I/O32
ptr IoCard11Pt33->u.io:$D10010.9.1           // I/O Card 11 I/O33
ptr IoCard11Pt34->u.io:$D10010.10.1          // I/O Card 11 I/O34
ptr IoCard11Pt35->u.io:$D10010.11.1          // I/O Card 11 I/O35
ptr IoCard11Pt36->u.io:$D10010.12.1          // I/O Card 11 I/O36
ptr IoCard11Pt37->u.io:$D10010.13.1          // I/O Card 11 I/O37
ptr IoCard11Pt38->u.io:$D10010.14.1          // I/O Card 11 I/O38
ptr IoCard11Pt39->u.io:$D10010.15.1          // I/O Card 11 I/O39
ptr IoCard11Pt40->u.io:$D10014.8.1           // I/O Card 11 I/O40
ptr IoCard11Pt41->u.io:$D10014.9.1           // I/O Card 11 I/O41
ptr IoCard11Pt42->u.io:$D10014.10.1          // I/O Card 11 I/O42
ptr IoCard11Pt43->u.io:$D10014.11.1          // I/O Card 11 I/O43
ptr IoCard11Pt44->u.io:$D10014.12.1          // I/O Card 11 I/O44
ptr IoCard11Pt45->u.io:$D10014.13.1          // I/O Card 11 I/O45
ptr IoCard11Pt46->u.io:$D10014.14.1          // I/O Card 11 I/O46
ptr IoCard11Pt47->u.io:$D10014.15.1          // I/O Card 11 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard11Reg0->u.io:$D10000.8.8    // I/O Card 11 I/O00-07 as byte
ptr IoCard11Reg1->u.io:$D10004.8.8    // I/O Card 11 I/O08-15 as byte
ptr IoCard11Reg2->u.io:$D10008.8.8    // I/O Card 11 I/O16-23 as byte
ptr IoCard11Reg3->u.io:$D1000C.8.8    // I/O Card 11 I/O24-31 as byte
ptr IoCard11Reg4->u.io:$D10010.8.8    // I/O Card 11 I/O32-39 as byte
ptr IoCard11Reg5->u.io:$D10014.8.8    // I/O Card 11 I/O40-47 as byte
ptr IoCard11Reg6->u.io:$D10018.8.8    // I/O Card 11 latch inputs
ptr IoCard11Reg7->u.io:$D1001C.8.8    // I/O Card 11 control register
```

Digital I/O Card 12

(SW1-1 ON, SW1-2 ON, SW1-3 OFF, SW1-4 OFF)

```
// Single-bit variables used for accessing I/O points
ptr IoCard12Pt00->u.io:$A18000.8.1           // I/O Card 12 I/O00
ptr IoCard12Pt01->u.io:$A18000.9.1           // I/O Card 12 I/O01
ptr IoCard12Pt02->u.io:$A18000.10.1          // I/O Card 12 I/O02
ptr IoCard12Pt03->u.io:$A18000.11.1          // I/O Card 12 I/O03
ptr IoCard12Pt04->u.io:$A18000.12.1          // I/O Card 12 I/O04
ptr IoCard12Pt05->u.io:$A18000.13.1          // I/O Card 12 I/O05
ptr IoCard12Pt06->u.io:$A18000.14.1          // I/O Card 12 I/O06
ptr IoCard12Pt07->u.io:$A18000.15.1          // I/O Card 12 I/O07
ptr IoCard12Pt08->u.io:$A18004.8.1           // I/O Card 12 I/O08
ptr IoCard12Pt09->u.io:$A18004.9.1           // I/O Card 12 I/O09
ptr IoCard12Pt10->u.io:$A18004.10.1          // I/O Card 12 I/O10
ptr IoCard12Pt11->u.io:$A18004.11.1          // I/O Card 12 I/O11
ptr IoCard12Pt12->u.io:$A18004.12.1          // I/O Card 12 I/O12
ptr IoCard12Pt13->u.io:$A18004.13.1          // I/O Card 12 I/O13
ptr IoCard12Pt14->u.io:$A18004.14.1          // I/O Card 12 I/O14
ptr IoCard12Pt15->u.io:$A18004.15.1          // I/O Card 12 I/O15
ptr IoCard12Pt16->u.io:$A18008.8.1           // I/O Card 12 I/O16
ptr IoCard12Pt17->u.io:$A18008.9.1           // I/O Card 12 I/O17
ptr IoCard12Pt18->u.io:$A18008.10.1          // I/O Card 12 I/O18
ptr IoCard12Pt19->u.io:$A18008.11.1          // I/O Card 12 I/O19
ptr IoCard12Pt20->u.io:$A18008.12.1          // I/O Card 12 I/O20
ptr IoCard12Pt21->u.io:$A18008.13.1          // I/O Card 12 I/O21
ptr IoCard12Pt22->u.io:$A18008.14.1          // I/O Card 12 I/O22
ptr IoCard12Pt23->u.io:$A18008.15.1          // I/O Card 12 I/O23
ptr IoCard12Pt24->u.io:$A1800C.8.1           // I/O Card 12 I/O24
ptr IoCard12Pt25->u.io:$A1800C.9.1           // I/O Card 12 I/O25
ptr IoCard12Pt26->u.io:$A1800C.10.1          // I/O Card 12 I/O26
ptr IoCard12Pt27->u.io:$A1800C.11.1          // I/O Card 12 I/O27
ptr IoCard12Pt28->u.io:$A1800C.12.1          // I/O Card 12 I/O28
ptr IoCard12Pt29->u.io:$A1800C.13.1          // I/O Card 12 I/O29
ptr IoCard12Pt30->u.io:$A1800C.14.1          // I/O Card 12 I/O30
ptr IoCard12Pt31->u.io:$A1800C.15.1          // I/O Card 12 I/O31
ptr IoCard12Pt32->u.io:$A18010.8.1           // I/O Card 12 I/O32
ptr IoCard12Pt33->u.io:$A18010.9.1           // I/O Card 12 I/O33
ptr IoCard12Pt34->u.io:$A18010.10.1          // I/O Card 12 I/O34
ptr IoCard12Pt35->u.io:$A18010.11.1          // I/O Card 12 I/O35
ptr IoCard12Pt36->u.io:$A18010.12.1          // I/O Card 12 I/O36
ptr IoCard12Pt37->u.io:$A18010.13.1          // I/O Card 12 I/O37
ptr IoCard12Pt38->u.io:$A18010.14.1          // I/O Card 12 I/O38
ptr IoCard12Pt39->u.io:$A18010.15.1          // I/O Card 12 I/O39
ptr IoCard12Pt40->u.io:$A18014.8.1           // I/O Card 12 I/O40
ptr IoCard12Pt41->u.io:$A18014.9.1           // I/O Card 12 I/O41
ptr IoCard12Pt42->u.io:$A18014.10.1          // I/O Card 12 I/O42
ptr IoCard12Pt43->u.io:$A18014.11.1          // I/O Card 12 I/O43
ptr IoCard12Pt44->u.io:$A18014.12.1          // I/O Card 12 I/O44
ptr IoCard12Pt45->u.io:$A18014.13.1          // I/O Card 12 I/O45
ptr IoCard12Pt46->u.io:$A18014.14.1          // I/O Card 12 I/O46
ptr IoCard12Pt47->u.io:$A18014.15.1          // I/O Card 12 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard12Reg0->u.io:$A18000.8.8    // I/O Card 12 I/O00-07 as byte
ptr IoCard12Reg1->u.io:$A18004.8.8    // I/O Card 12 I/O08-15 as byte
ptr IoCard12Reg2->u.io:$A18008.8.8    // I/O Card 12 I/O16-23 as byte
ptr IoCard12Reg3->u.io:$A1800C.8.8    // I/O Card 12 I/O24-31 as byte
ptr IoCard12Reg4->u.io:$A18010.8.8    // I/O Card 12 I/O32-39 as byte
ptr IoCard12Reg5->u.io:$A18014.8.8    // I/O Card 12 I/O40-47 as byte
ptr IoCard12Reg6->u.io:$A18018.8.8    // I/O Card 12 latch inputs
ptr IoCard12Reg7->u.io:$A1801C.8.8    // I/O Card 12 control register
```

Digital I/O Card 13

(SW1-1 OFF, SW1-2 ON, SW1-3 OFF, SW1-4 OFF)

```
// Single-bit variables used for accessing I/O points
M1300->u.io:$B18000.8.1                        // I/O Card 13 I/O00
ptr IoCard13Pt01->u.io:$B18000.9.1             // I/O Card 13 I/O01
ptr IoCard13Pt02->u.io:$B18000.10.1            // I/O Card 13 I/O02
ptr IoCard13Pt03->u.io:$B18000.11.1            // I/O Card 13 I/O03
ptr IoCard13Pt04->u.io:$B18000.12.1            // I/O Card 13 I/O04
ptr IoCard13Pt05->u.io:$B18000.13.1            // I/O Card 13 I/O05
ptr IoCard13Pt06->u.io:$B18000.14.1            // I/O Card 13 I/O06
ptr IoCard13Pt07->u.io:$B18000.15.1            // I/O Card 13 I/O07
ptr IoCard13Pt08->u.io:$B18004.8.1             // I/O Card 13 I/O08
ptr IoCard13Pt09->u.io:$B18004.9.1             // I/O Card 13 I/O09
ptr IoCard13Pt10->u.io:$B18004.10.1            // I/O Card 13 I/O10
ptr IoCard13Pt11->u.io:$B18004.11.1            // I/O Card 13 I/O11
ptr IoCard13Pt12->u.io:$B18004.12.1            // I/O Card 13 I/O12
ptr IoCard13Pt13->u.io:$B18004.13.1            // I/O Card 13 I/O13
ptr IoCard13Pt14->u.io:$B18004.14.1            // I/O Card 13 I/O14
ptr IoCard13Pt15->u.io:$B18004.15.1            // I/O Card 13 I/O15
ptr IoCard13Pt16->u.io:$B18008.8.1             // I/O Card 13 I/O16
ptr IoCard13Pt17->u.io:$B18008.9.1             // I/O Card 13 I/O17
ptr IoCard13Pt18->u.io:$B18008.10.1            // I/O Card 13 I/O18
ptr IoCard13Pt19->u.io:$B18008.11.1            // I/O Card 13 I/O19
ptr IoCard13Pt20->u.io:$B18008.12.1            // I/O Card 13 I/O20
ptr IoCard13Pt21->u.io:$B18008.13.1            // I/O Card 13 I/O21
ptr IoCard13Pt22->u.io:$B18008.14.1            // I/O Card 13 I/O22
ptr IoCard13Pt23->u.io:$B18008.15.1            // I/O Card 13 I/O23
ptr IoCard13Pt24->u.io:$B1800C.8.1             // I/O Card 13 I/O24
ptr IoCard13Pt25->u.io:$B1800C.9.1             // I/O Card 13 I/O25
ptr IoCard13Pt26->u.io:$B1800C.10.1            // I/O Card 13 I/O26
ptr IoCard13Pt27->u.io:$B1800C.11.1            // I/O Card 13 I/O27
ptr IoCard13Pt28->u.io:$B1800C.12.1            // I/O Card 13 I/O28
ptr IoCard13Pt29->u.io:$B1800C.13.1            // I/O Card 13 I/O29
ptr IoCard13Pt30->u.io:$B1800C.14.1            // I/O Card 13 I/O30
ptr IoCard13Pt31->u.io:$B1800C.15.1            // I/O Card 13 I/O31
ptr IoCard13Pt32->u.io:$B18010.8.1             // I/O Card 13 I/O32
ptr IoCard13Pt33->u.io:$B18010.9.1             // I/O Card 13 I/O33
ptr IoCard13Pt34->u.io:$B18010.10.1            // I/O Card 13 I/O34
ptr IoCard13Pt35->u.io:$B18010.11.1            // I/O Card 13 I/O35
ptr IoCard13Pt36->u.io:$B18010.12.1            // I/O Card 13 I/O36
ptr IoCard13Pt37->u.io:$B18010.13.1            // I/O Card 13 I/O37
ptr IoCard13Pt38->u.io:$B18010.14.1            // I/O Card 13 I/O38
ptr IoCard13Pt39->u.io:$B18010.15.1            // I/O Card 13 I/O39
ptr IoCard13Pt40->u.io:$B18014.8.1             // I/O Card 13 I/O40
ptr IoCard13Pt41->u.io:$B18014.9.1             // I/O Card 13 I/O41
ptr IoCard13Pt42->u.io:$B18014.10.1            // I/O Card 13 I/O42
ptr IoCard13Pt43->u.io:$B18014.11.1            // I/O Card 13 I/O43
ptr IoCard13Pt44->u.io:$B18014.12.1            // I/O Card 13 I/O44
ptr IoCard13Pt45->u.io:$B18014.13.1            // I/O Card 13 I/O45
ptr IoCard13Pt46->u.io:$B18014.14.1            // I/O Card 13 I/O46
ptr IoCard13Pt47->u.io:$B18014.15.1            // I/O Card 13 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard13Reg0->u.io:$B18000.8.8    // I/O Card 13 I/O00-07 as byte
ptr IoCard13Reg1->u.io:$B18004.8.8    // I/O Card 13 I/O08-15 as byte
ptr IoCard13Reg2->u.io:$B18008.8.8    // I/O Card 13 I/O16-23 as byte
ptr IoCard13Reg3->u.io:$B1800C.8.8    // I/O Card 13 I/O24-31 as byte
ptr IoCard13Reg4->u.io:$B18010.8.8    // I/O Card 13 I/O32-39 as byte
ptr IoCard13Reg5->u.io:$B18014.8.8    // I/O Card 13 I/O40-47 as byte
ptr IoCard13Reg6->u.io:$B18018.8.8    // I/O Card 13 latch inputs
ptr IoCard13Reg7->u.io:$B1801C.8.8    // I/O Card 13 control register
```

Digital I/O Card 14

(SW1-1 ON, SW1-2 OFF, SW1-3 OFF, SW1-4 OFF)

```
// Single-bit variables used for accessing I/O points
ptr IoCard14Pt00->u.io:$C18000.8.1           // I/O Card 14 I/O00
ptr IoCard14Pt01->u.io:$C18000.9.1           // I/O Card 14 I/O01
ptr IoCard14Pt02->u.io:$C18000.10.1          // I/O Card 14 I/O02
ptr IoCard14Pt03->u.io:$C18000.11.1          // I/O Card 14 I/O03
ptr IoCard14Pt04->u.io:$C18000.12.1          // I/O Card 14 I/O04
ptr IoCard14Pt05->u.io:$C18000.13.1          // I/O Card 14 I/O05
ptr IoCard14Pt06->u.io:$C18000.14.1          // I/O Card 14 I/O06
ptr IoCard14Pt07->u.io:$C18000.15.1          // I/O Card 14 I/O07
ptr IoCard14Pt08->u.io:$C18004.8.1           // I/O Card 14 I/O08
ptr IoCard14Pt09->u.io:$C18004.9.1           // I/O Card 14 I/O09
ptr IoCard14Pt10->u.io:$C18004.10.1          // I/O Card 14 I/O10
ptr IoCard14Pt11->u.io:$C18004.11.1          // I/O Card 14 I/O11
ptr IoCard14Pt12->u.io:$C18004.12.1          // I/O Card 14 I/O12
ptr IoCard14Pt13->u.io:$C18004.13.1          // I/O Card 14 I/O13
ptr IoCard14Pt14->u.io:$C18004.14.1          // I/O Card 14 I/O14
ptr IoCard14Pt15->u.io:$C18004.15.1          // I/O Card 14 I/O15
ptr IoCard14Pt16->u.io:$C18008.8.1           // I/O Card 14 I/O16
ptr IoCard14Pt17->u.io:$C18008.9.1           // I/O Card 14 I/O17
ptr IoCard14Pt18->u.io:$C18008.10.1          // I/O Card 14 I/O18
ptr IoCard14Pt19->u.io:$C18008.11.1          // I/O Card 14 I/O19
ptr IoCard14Pt20->u.io:$C18008.12.1          // I/O Card 14 I/O20
ptr IoCard14Pt21->u.io:$C18008.13.1          // I/O Card 14 I/O21
ptr IoCard14Pt22->u.io:$C18008.14.1          // I/O Card 14 I/O22
ptr IoCard14Pt23->u.io:$C18008.15.1          // I/O Card 14 I/O23
ptr IoCard14Pt24->u.io:$C1800C.8.1           // I/O Card 14 I/O24
ptr IoCard14Pt25->u.io:$C1800C.9.1           // I/O Card 14 I/O25
ptr IoCard14Pt26->u.io:$C1800C.10.1          // I/O Card 14 I/O26
ptr IoCard14Pt27->u.io:$C1800C.11.1          // I/O Card 14 I/O27
ptr IoCard14Pt28->u.io:$C1800C.12.1          // I/O Card 14 I/O28
ptr IoCard14Pt29->u.io:$C1800C.13.1          // I/O Card 14 I/O29
ptr IoCard14Pt30->u.io:$C1800C.14.1          // I/O Card 14 I/O30
ptr IoCard14Pt31->u.io:$C1800C.15.1          // I/O Card 14 I/O31
ptr IoCard14Pt32->u.io:$C18010.8.1           // I/O Card 14 I/O32
ptr IoCard14Pt33->u.io:$C18010.9.1           // I/O Card 14 I/O33
ptr IoCard14Pt34->u.io:$C18010.10.1          // I/O Card 14 I/O34
ptr IoCard14Pt35->u.io:$C18010.11.1          // I/O Card 14 I/O35
ptr IoCard14Pt36->u.io:$C18010.12.1          // I/O Card 14 I/O36
ptr IoCard14Pt37->u.io:$C18010.13.1          // I/O Card 14 I/O37
ptr IoCard14Pt38->u.io:$C18010.14.1          // I/O Card 14 I/O38
ptr IoCard14Pt39->u.io:$C18010.15.1          // I/O Card 14 I/O39
ptr IoCard14Pt40->u.io:$C18014.8.1           // I/O Card 14 I/O40
ptr IoCard14Pt41->u.io:$C18014.9.1           // I/O Card 14 I/O41
ptr IoCard14Pt42->u.io:$C18014.10.1          // I/O Card 14 I/O42
ptr IoCard14Pt43->u.io:$C18014.11.1          // I/O Card 14 I/O43
ptr IoCard14Pt44->u.io:$C18014.12.1          // I/O Card 14 I/O44
ptr IoCard14Pt45->u.io:$C18014.13.1          // I/O Card 14 I/O45
ptr IoCard14Pt46->u.io:$C18014.14.1          // I/O Card 14 I/O46
ptr IoCard14Pt47->u.io:$C18014.15.1          // I/O Card 14 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard14Reg0->u.io:$C18000.8.8    // I/O Card 14 I/O00-07 as byte
ptr IoCard14Reg1->u.io:$C18004.8.8    // I/O Card 14 I/O08-15 as byte
ptr IoCard14Reg2->u.io:$C18008.8.8    // I/O Card 14 I/O16-23 as byte
ptr IoCard14Reg3->u.io:$C1800C.8.8    // I/O Card 14 I/O24-31 as byte
ptr IoCard14Reg4->u.io:$C18010.8.8    // I/O Card 14 I/O32-39 as byte
ptr IoCard14Reg5->u.io:$C18014.8.8    // I/O Card 14 I/O40-47 as byte
ptr IoCard14Reg6->u.io:$C18018.8.8    // I/O Card 14 latch inputs
ptr IoCard14Reg7->u.io:$C1801C.8.8    // I/O Card 14 control register
```

Digital I/O Card 15

(SW1-1 OFF, SW1-2 OFF, SW1-3 OFF, SW1-4 OFF)

```
// Single-bit variables used for accessing I/O points
ptr IoCard15Pt00->u.io:$D18000.8.1           // I/O Card 15 I/O00
ptr IoCard15Pt01->u.io:$D18000.9.1           // I/O Card 15 I/O01
ptr IoCard15Pt02->u.io:$D18000.10.1          // I/O Card 15 I/O02
ptr IoCard15Pt03->u.io:$D18000.11.1          // I/O Card 15 I/O03
ptr IoCard15Pt04->u.io:$D18000.12.1          // I/O Card 15 I/O04
ptr IoCard15Pt05->u.io:$D18000.13.1          // I/O Card 15 I/O05
ptr IoCard15Pt06->u.io:$D18000.14.1          // I/O Card 15 I/O06
ptr IoCard15Pt07->u.io:$D18000.15.1          // I/O Card 15 I/O07
ptr IoCard15Pt08->u.io:$D18004.8.1           // I/O Card 15 I/O08
ptr IoCard15Pt09->u.io:$D18004.9.1           // I/O Card 15 I/O09
ptr IoCard15Pt10->u.io:$D18004.10.1          // I/O Card 15 I/O10
ptr IoCard15Pt11->u.io:$D18004.11.1          // I/O Card 15 I/O11
ptr IoCard15Pt12->u.io:$D18004.12.1          // I/O Card 15 I/O12
ptr IoCard15Pt13->u.io:$D18004.13.1          // I/O Card 15 I/O13
ptr IoCard15Pt14->u.io:$D18004.14.1          // I/O Card 15 I/O14
ptr IoCard15Pt15->u.io:$D18004.15.1          // I/O Card 15 I/O15
ptr IoCard15Pt16->u.io:$D18008.8.1           // I/O Card 15 I/O16
ptr IoCard15Pt17->u.io:$D18008.9.1           // I/O Card 15 I/O17
ptr IoCard15Pt18->u.io:$D18008.10.1          // I/O Card 15 I/O18
ptr IoCard15Pt19->u.io:$D18008.11.1          // I/O Card 15 I/O19
ptr IoCard15Pt20->u.io:$D18008.12.1          // I/O Card 15 I/O20
ptr IoCard15Pt21->u.io:$D18008.13.1          // I/O Card 15 I/O21
ptr IoCard15Pt22->u.io:$D18008.14.1          // I/O Card 15 I/O22
ptr IoCard15Pt23->u.io:$D18008.15.1          // I/O Card 15 I/O23
ptr IoCard15Pt24->u.io:$D1800C.8.1           // I/O Card 15 I/O24
ptr IoCard15Pt25->u.io:$D1800C.9.1           // I/O Card 15 I/O25
ptr IoCard15Pt26->u.io:$D1800C.10.1          // I/O Card 15 I/O26
ptr IoCard15Pt27->u.io:$D1800C.11.1          // I/O Card 15 I/O27
ptr IoCard15Pt28->u.io:$D1800C.12.1          // I/O Card 15 I/O28
ptr IoCard15Pt29->u.io:$D1800C.13.1          // I/O Card 15 I/O29
ptr IoCard15Pt30->u.io:$D1800C.14.1          // I/O Card 15 I/O30
ptr IoCard15Pt31->u.io:$D1800C.15.1          // I/O Card 15 I/O31
ptr IoCard15Pt32->u.io:$D18010.8.1           // I/O Card 15 I/O32
ptr IoCard15Pt33->u.io:$D18010.9.1           // I/O Card 15 I/O33
ptr IoCard15Pt34->u.io:$D18010.10.1          // I/O Card 15 I/O34
ptr IoCard15Pt35->u.io:$D18010.11.1          // I/O Card 15 I/O35
ptr IoCard15Pt36->u.io:$D18010.12.1          // I/O Card 15 I/O36
ptr IoCard15Pt37->u.io:$D18010.13.1          // I/O Card 15 I/O37
ptr IoCard15Pt38->u.io:$D18010.14.1          // I/O Card 15 I/O38
ptr IoCard15Pt39->u.io:$D18010.15.1          // I/O Card 15 I/O39
ptr IoCard15Pt40->u.io:$D18014.8.1           // I/O Card 15 I/O40
ptr IoCard15Pt41->u.io:$D18014.9.1           // I/O Card 15 I/O41
ptr IoCard15Pt42->u.io:$D18014.10.1          // I/O Card 15 I/O42
ptr IoCard15Pt43->u.io:$D18014.11.1          // I/O Card 15 I/O43
ptr IoCard15Pt44->u.io:$D18014.12.1          // I/O Card 15 I/O44
ptr IoCard15Pt45->u.io:$D18014.13.1          // I/O Card 15 I/O45
ptr IoCard15Pt46->u.io:$D18014.14.1          // I/O Card 15 I/O46
ptr IoCard15Pt47->u.io:$D18014.15.1          // I/O Card 15 I/O47
// Byte-wide variables used for power-on configuration
ptr IoCard15Reg0->u.io:$D18000.8.8    // I/O Card 15 I/O00-07 as byte
ptr IoCard15Reg1->u.io:$D18004.8.8    // I/O Card 15 I/O08-15 as byte
ptr IoCard15Reg2->u.io:$D18008.8.8    // I/O Card 15 I/O16-23 as byte
ptr IoCard15Reg3->u.io:$D1800C.8.8    // I/O Card 15 I/O24-31 as byte
ptr IoCard15Reg4->u.io:$D18010.8.8    // I/O Card 15 I/O32-39 as byte
ptr IoCard15Reg5->u.io:$D18014.8.8    // I/O Card 15 I/O40-47 as byte
ptr IoCard15Reg6->u.io:$D18018.8.8    // I/O Card 15 latch inputs
ptr IoCard15Reg7->u.io:$D1801C.8.8    // I/O Card 15 control register
```

ACC-84E Card 0

(SW1-1 ON, SW1-2 ON, SW1-3 ON, SW1-4 ON)

```
ptr EncCard0Ctrl->u.io:$A0007C.8.24          // ACC-84E Card 0 control reg
ptr EncCard0Chan0Cmd->u.io:$A00020.8.24      // ACC-84E Card 0 Chan 0 cmd reg
ptr EncCard0Chan0DataA->u.io:$A00000.8.24    // ACC-84E Card 0 Chan 0 data reg A
ptr EncCard0Chan0DataB->u.io:$A00004.8.24    // ACC-84E Card 0 Chan 0 data reg B
ptr EncCard0Chan0DataC->u.io:$A00008.8.24    // ACC-84E Card 0 Chan 0 data reg C
ptr EncCard0Chan0DataD->u.io:$A0000C.8.24    // ACC-84E Card 0 Chan 0 data reg D
ptr EncCard0Chan1Cmd->u.io:$A00030.8.24      // ACC-84E Card 0 Chan 1 cmd reg
ptr EncCard0Chan1DataA->u.io:$A00010.8.24    // ACC-84E Card 0 Chan 1 data reg A
ptr EncCard0Chan1DataB->u.io:$A00014.8.24    // ACC-84E Card 0 Chan 1 data reg B
ptr EncCard0Chan1DataC->u.io:$A00018.8.24    // ACC-84E Card 0 Chan 1 data reg C
ptr EncCard0Chan1DataD->u.io:$A0001C.8.24    // ACC-84E Card 0 Chan 1 data reg D
ptr EncCard0Chan2Cmd->u.io:$A00060.8.24      // ACC-84E Card 0 Chan 2 cmd reg
ptr EncCard0Chan2DataA->u.io:$A00040.8.24    // ACC-84E Card 0 Chan 2 data reg A
ptr EncCard0Chan2DataB->u.io:$A00044.8.24    // ACC-84E Card 0 Chan 2 data reg B
ptr EncCard0Chan2DataC->u.io:$A00048.8.24    // ACC-84E Card 0 Chan 2 data reg C
ptr EncCard0Chan2DataD->u.io:$A0004C.8.24    // ACC-84E Card 0 Chan 2 data reg D
ptr EncCard0Chan3Cmd->u.io:$A00070.8.24      // ACC-84E Card 0 Chan 3 cmd reg
ptr EncCard0Chan3DataA->u.io:$A00050.8.24    // ACC-84E Card 0 Chan 3 data reg A
ptr EncCard0Chan3DataB->u.io:$A00054.8.24    // ACC-84E Card 0 Chan 3 data reg B
ptr EncCard0Chan3DataC->u.io:$A00058.8.24    // ACC-84E Card 0 Chan 3 data reg C
ptr EncCard0Chan3DataD->u.io:$A0005C.8.24    // ACC-84E Card 0 Chan 3 data reg D
```

ACC-84E Card 1

(SW1-1 OFF, SW1-2 ON, SW1-3 ON, SW1-4 ON)

```
ptr EncCard1Ctrl->u.io:$B0007C.8.24          // ACC-84E Card 1 control reg
ptr EncCard1Chan0Cmd->u.io:$B00020.8.24      // ACC-84E Card 1 Chan 0 cmd reg
ptr EncCard1Chan0DataA->u.io:$B00000.8.24    // ACC-84E Card 1 Chan 0 data reg A
ptr EncCard1Chan0DataB->u.io:$B00004.8.24    // ACC-84E Card 1 Chan 0 data reg B
ptr EncCard1Chan0DataC->u.io:$B00008.8.24    // ACC-84E Card 1 Chan 0 data reg C
ptr EncCard1Chan0DataD->u.io:$B0000C.8.24    // ACC-84E Card 1 Chan 0 data reg D
ptr EncCard1Chan1Cmd->u.io:$B00030.8.24      // ACC-84E Card 1 Chan 1 cmd reg
ptr EncCard1Chan1DataA->u.io:$B00010.8.24    // ACC-84E Card 1 Chan 1 data reg A
ptr EncCard1Chan1DataB->u.io:$B00014.8.24    // ACC-84E Card 1 Chan 1 data reg B
ptr EncCard1Chan1DataC->u.io:$B00018.8.24    // ACC-84E Card 1 Chan 1 data reg C
ptr EncCard1Chan1DataD->u.io:$B0001C.8.24    // ACC-84E Card 1 Chan 1 data reg D
ptr EncCard1Chan2Cmd->u.io:$B00060.8.24      // ACC-84E Card 1 Chan 2 cmd reg
ptr EncCard1Chan2DataA->u.io:$B00040.8.24    // ACC-84E Card 1 Chan 2 data reg A
ptr EncCard1Chan2DataB->u.io:$B00044.8.24    // ACC-84E Card 1 Chan 2 data reg B
ptr EncCard1Chan2DataC->u.io:$B00048.8.24    // ACC-84E Card 1 Chan 2 data reg C
ptr EncCard1Chan2DataD->u.io:$B0004C.8.24    // ACC-84E Card 1 Chan 2 data reg D
ptr EncCard1Chan3Cmd->u.io:$B00070.8.24      // ACC-84E Card 1 Chan 3 cmd reg
ptr EncCard1Chan3DataA->u.io:$B00050.8.24    // ACC-84E Card 1 Chan 3 data reg A
ptr EncCard1Chan3DataB->u.io:$B00054.8.24    // ACC-84E Card 1 Chan 3 data reg B
ptr EncCard1Chan3DataC->u.io:$B00058.8.24    // ACC-84E Card 1 Chan 3 data reg C
ptr EncCard1Chan3DataD->u.io:$B0005C.8.24    // ACC-84E Card 1 Chan 3 data reg D
```

ACC-84E Card 2

(SW1-1 ON, SW1-2 OFF, SW1-3 ON, SW1-4 ON)

```
ptr EncCard2Ctrl->u.io:$C0007C.8.24          // ACC-84E Card 2 control reg
ptr EncCard2Chan0Cmd->u.io:$C00020.8.24      // ACC-84E Card 2 Chan 0 cmd reg
ptr EncCard2Chan0DataA->u.io:$C00000.8.24    // ACC-84E Card 2 Chan 0 data reg A
ptr EncCard2Chan0DataB->u.io:$C00004.8.24    // ACC-84E Card 2 Chan 0 data reg B
ptr EncCard2Chan0DataC->u.io:$C00008.8.24    // ACC-84E Card 2 Chan 0 data reg C
ptr EncCard2Chan0DataD->u.io:$C0000C.8.24    // ACC-84E Card 2 Chan 0 data reg D
ptr EncCard2Chan1Cmd->u.io:$C00030.8.24      // ACC-84E Card 2 Chan 1 cmd reg
ptr EncCard2Chan1DataA->u.io:$C00010.8.24    // ACC-84E Card 2 Chan 1 data reg A
ptr EncCard2Chan1DataB->u.io:$C00014.8.24    // ACC-84E Card 2 Chan 1 data reg B
ptr EncCard2Chan1DataC->u.io:$C00018.8.24    // ACC-84E Card 2 Chan 1 data reg C
ptr EncCard2Chan1DataD->u.io:$C0001C.8.24    // ACC-84E Card 2 Chan 1 data reg D
ptr EncCard2Chan2Cmd->u.io:$C00060.8.24      // ACC-84E Card 2 Chan 2 cmd reg
ptr EncCard2Chan2DataA->u.io:$C00040.8.24    // ACC-84E Card 2 Chan 2 data reg A
ptr EncCard2Chan2DataB->u.io:$C00044.8.24    // ACC-84E Card 2 Chan 2 data reg B
ptr EncCard2Chan2DataC->u.io:$C00048.8.24    // ACC-84E Card 2 Chan 2 data reg C
ptr EncCard2Chan2DataD->u.io:$C0004C.8.24    // ACC-84E Card 2 Chan 2 data reg D
ptr EncCard2Chan3Cmd->u.io:$C00070.8.24      // ACC-84E Card 2 Chan 3 cmd reg
ptr EncCard2Chan3DataA->u.io:$C00050.8.24    // ACC-84E Card 2 Chan 3 data reg A
ptr EncCard2Chan3DataB->u.io:$C00054.8.24    // ACC-84E Card 2 Chan 3 data reg B
ptr EncCard2Chan3DataC->u.io:$C00058.8.24    // ACC-84E Card 2 Chan 3 data reg C
ptr EncCard2Chan3DataD->u.io:$C0005C.8.24    // ACC-84E Card 2 Chan 3 data reg D
```

ACC-84E Card 3

(SW1-1 OFF, SW1-2 OFF, SW1-3 ON, SW1-4 ON)

```
ptr EncCard3Ctrl->u.io:$D0007C.8.24          // ACC-84E Card 3 control reg
ptr EncCard3Chan0Cmd->u.io:$D00020.8.24      // ACC-84E Card 3 Chan 0 cmd reg
ptr EncCard3Chan0DataA->u.io:$D00000.8.24    // ACC-84E Card 3 Chan 0 data reg A
ptr EncCard3Chan0DataB->u.io:$D00004.8.24    // ACC-84E Card 3 Chan 0 data reg B
ptr EncCard3Chan0DataC->u.io:$D00008.8.24    // ACC-84E Card 3 Chan 0 data reg C
ptr EncCard3Chan0DataD->u.io:$D0000C.8.24    // ACC-84E Card 3 Chan 0 data reg D
ptr EncCard3Chan1Cmd->u.io:$D00030.8.24      // ACC-84E Card 3 Chan 1 cmd reg
ptr EncCard3Chan1DataA->u.io:$D00010.8.24    // ACC-84E Card 3 Chan 1 data reg A
ptr EncCard3Chan1DataB->u.io:$D00014.8.24    // ACC-84E Card 3 Chan 1 data reg B
ptr EncCard3Chan1DataC->u.io:$D00018.8.24    // ACC-84E Card 3 Chan 1 data reg C
ptr EncCard3Chan1DataD->u.io:$D0001C.8.24    // ACC-84E Card 3 Chan 1 data reg D
ptr EncCard3Chan2Cmd->u.io:$D00060.8.24      // ACC-84E Card 3 Chan 2 cmd reg
ptr EncCard3Chan2DataA->u.io:$D00040.8.24    // ACC-84E Card 3 Chan 2 data reg A
ptr EncCard3Chan2DataB->u.io:$D00044.8.24    // ACC-84E Card 3 Chan 2 data reg B
ptr EncCard3Chan2DataC->u.io:$D00048.8.24    // ACC-84E Card 3 Chan 2 data reg C
ptr EncCard3Chan2DataD->u.io:$D0004C.8.24    // ACC-84E Card 3 Chan 2 data reg D
ptr EncCard3Chan3Cmd->u.io:$D00070.8.24      // ACC-84E Card 3 Chan 3 cmd reg
ptr EncCard3Chan3DataA->u.io:$D00050.8.24    // ACC-84E Card 3 Chan 3 data reg A
ptr EncCard3Chan3DataB->u.io:$D00054.8.24    // ACC-84E Card 3 Chan 3 data reg B
ptr EncCard3Chan3DataC->u.io:$D00058.8.24    // ACC-84E Card 3 Chan 3 data reg C
ptr EncCard3Chan3DataD->u.io:$D0005C.8.24    // ACC-84E Card 3 Chan 3 data reg D
```

ACC-84E Card 4

(SW1-1 ON, SW1-2 ON, SW1-3 OFF, SW1-4 ON)

```
ptr EncCard4Ctrl->u.io:$A0807C.8.24          // ACC-84E Card 4 control reg
ptr EncCard4Chan0Cmd->u.io:$A08020.8.24      // ACC-84E Card 4 Chan 0 cmd reg
ptr EncCard4Chan0DataA->u.io:$A08000.8.24    // ACC-84E Card 4 Chan 0 data reg A
ptr EncCard4Chan0DataB->u.io:$A08004.8.24    // ACC-84E Card 4 Chan 0 data reg B
ptr EncCard4Chan0DataC->u.io:$A08008.8.24    // ACC-84E Card 4 Chan 0 data reg C
ptr EncCard4Chan0DataD->u.io:$A0800C.8.24    // ACC-84E Card 4 Chan 0 data reg D
ptr EncCard4Chan1Cmd->u.io:$A08030.8.24      // ACC-84E Card 4 Chan 1 cmd reg
ptr EncCard4Chan1DataA->u.io:$A08010.8.24    // ACC-84E Card 4 Chan 1 data reg A
ptr EncCard4Chan1DataB->u.io:$A08014.8.24    // ACC-84E Card 4 Chan 1 data reg B
ptr EncCard4Chan1DataC->u.io:$A08018.8.24    // ACC-84E Card 4 Chan 1 data reg C
ptr EncCard4Chan1DataD->u.io:$A0801C.8.24    // ACC-84E Card 4 Chan 1 data reg D
ptr EncCard4Chan2Cmd->u.io:$A08060.8.24      // ACC-84E Card 4 Chan 2 cmd reg
ptr EncCard4Chan2DataA->u.io:$A08040.8.24    // ACC-84E Card 4 Chan 2 data reg A
ptr EncCard4Chan2DataB->u.io:$A08044.8.24    // ACC-84E Card 4 Chan 2 data reg B
ptr EncCard4Chan2DataC->u.io:$A08048.8.24    // ACC-84E Card 4 Chan 2 data reg C
ptr EncCard4Chan2DataD->u.io:$A0804C.8.24    // ACC-84E Card 4 Chan 2 data reg D
ptr EncCard4Chan3Cmd->u.io:$A08070.8.24      // ACC-84E Card 4 Chan 3 cmd reg
ptr EncCard4Chan3DataA->u.io:$A08050.8.24    // ACC-84E Card 4 Chan 3 data reg A
ptr EncCard4Chan3DataB->u.io:$A08054.8.24    // ACC-84E Card 4 Chan 3 data reg B
ptr EncCard4Chan3DataC->u.io:$A08058.8.24    // ACC-84E Card 4 Chan 3 data reg C
ptr EncCard4Chan3DataD->u.io:$A0805C.8.24    // ACC-84E Card 4 Chan 3 data reg D
```

ACC-84E Card 5

(SW1-1 OFF, SW1-2 ON, SW1-3 OFF, SW1-4 ON)

```
ptr EncCard5Ctrl->u.io:$B0807C.8.24          // ACC-84E Card 5 control reg
ptr EncCard5Chan0Cmd->u.io:$B08020.8.24      // ACC-84E Card 5 Chan 0 cmd reg
ptr EncCard5Chan0DataA->u.io:$B08000.8.24    // ACC-84E Card 5 Chan 0 data reg A
ptr EncCard5Chan0DataB->u.io:$B08004.8.24    // ACC-84E Card 5 Chan 0 data reg B
ptr EncCard5Chan0DataC->u.io:$B08008.8.24    // ACC-84E Card 5 Chan 0 data reg C
ptr EncCard5Chan0DataD->u.io:$B0800C.8.24    // ACC-84E Card 5 Chan 0 data reg D
ptr EncCard5Chan1Cmd->u.io:$B08030.8.24      // ACC-84E Card 5 Chan 1 cmd reg
ptr EncCard5Chan1DataA->u.io:$B08010.8.24    // ACC-84E Card 5 Chan 1 data reg A
ptr EncCard5Chan1DataB->u.io:$B08014.8.24    // ACC-84E Card 5 Chan 1 data reg B
ptr EncCard5Chan1DataC->u.io:$B08018.8.24    // ACC-84E Card 5 Chan 1 data reg C
ptr EncCard5Chan1DataD->u.io:$B0801C.8.24    // ACC-84E Card 5 Chan 1 data reg D
ptr EncCard5Chan2Cmd->u.io:$B08060.8.24      // ACC-84E Card 5 Chan 2 cmd reg
ptr EncCard5Chan2DataA->u.io:$B08040.8.24    // ACC-84E Card 5 Chan 2 data reg A
ptr EncCard5Chan2DataB->u.io:$B08044.8.24    // ACC-84E Card 5 Chan 2 data reg B
ptr EncCard5Chan2DataC->u.io:$B08048.8.24    // ACC-84E Card 5 Chan 2 data reg C
ptr EncCard5Chan2DataD->u.io:$B0804C.8.24    // ACC-84E Card 5 Chan 2 data reg D
ptr EncCard5Chan3Cmd->u.io:$B08070.8.24      // ACC-84E Card 5 Chan 3 cmd reg
ptr EncCard5Chan3DataA->u.io:$B08050.8.24    // ACC-84E Card 5 Chan 3 data reg A
ptr EncCard5Chan3DataB->u.io:$B08054.8.24    // ACC-84E Card 5 Chan 3 data reg B
ptr EncCard5Chan3DataC->u.io:$B08058.8.24    // ACC-84E Card 5 Chan 3 data reg C
ptr EncCard5Chan3DataD->u.io:$B0805C.8.24    // ACC-84E Card 5 Chan 3 data reg D
```

ACC-84E Card 6

(SW1-1 ON, SW1-2 OFF, SW1-3 OFF, SW1-4 ON)

```
ptr EncCard6Ctrl->u.io:$C0807C.8.24          // ACC-84E Card 6 control reg
ptr EncCard6Chan0Cmd->u.io:$C08020.8.24      // ACC-84E Card 6 Chan 0 cmd reg
ptr EncCard6Chan0DataA->u.io:$C08000.8.24    // ACC-84E Card 6 Chan 0 data reg A
ptr EncCard6Chan0DataB->u.io:$C08004.8.24    // ACC-84E Card 6 Chan 0 data reg B
ptr EncCard6Chan0DataC->u.io:$C08008.8.24    // ACC-84E Card 6 Chan 0 data reg C
ptr EncCard6Chan0DataD->u.io:$C0800C.8.24    // ACC-84E Card 6 Chan 0 data reg D
ptr EncCard6Chan1Cmd->u.io:$C08030.8.24      // ACC-84E Card 6 Chan 1 cmd reg
ptr EncCard6Chan1DataA->u.io:$C08010.8.24    // ACC-84E Card 6 Chan 1 data reg A
ptr EncCard6Chan1DataB->u.io:$C08014.8.24    // ACC-84E Card 6 Chan 1 data reg B
ptr EncCard6Chan1DataC->u.io:$C08018.8.24    // ACC-84E Card 6 Chan 1 data reg C
ptr EncCard6Chan1DataD->u.io:$C0801C.8.24    // ACC-84E Card 6 Chan 1 data reg D
ptr EncCard6Chan2Cmd->u.io:$C08060.8.24      // ACC-84E Card 6 Chan 2 cmd reg
ptr EncCard6Chan2DataA->u.io:$C08040.8.24    // ACC-84E Card 6 Chan 2 data reg A
ptr EncCard6Chan2DataB->u.io:$C08044.8.24    // ACC-84E Card 6 Chan 2 data reg B
ptr EncCard6Chan2DataC->u.io:$C08048.8.24    // ACC-84E Card 6 Chan 2 data reg C
ptr EncCard6Chan2DataD->u.io:$C0804C.8.24    // ACC-84E Card 6 Chan 2 data reg D
ptr EncCard6Chan3Cmd->u.io:$C08070.8.24      // ACC-84E Card 6 Chan 3 cmd reg
ptr EncCard6Chan3DataA->u.io:$C08050.8.24    // ACC-84E Card 6 Chan 3 data reg A
ptr EncCard6Chan3DataB->u.io:$C08054.8.24    // ACC-84E Card 6 Chan 3 data reg B
ptr EncCard6Chan3DataC->u.io:$C08058.8.24    // ACC-84E Card 6 Chan 3 data reg C
ptr EncCard6Chan3DataD->u.io:$C0805C.8.24    // ACC-84E Card 6 Chan 3 data reg D
```

ACC-84E Card 7

(SW1-1 OFF, SW1-2 OFF, SW1-3 OFF, SW1-4 ON)

```
ptr EncCard7Ctrl->u.io:$D0807C.8.24          // ACC-84E Card 7 control reg
ptr EncCard7Chan0Cmd->u.io:$D08020.8.24      // ACC-84E Card 7 Chan 0 cmd reg
ptr EncCard7Chan0DataA->u.io:$D08000.8.24    // ACC-84E Card 7 Chan 0 data reg A
ptr EncCard7Chan0DataB->u.io:$D08004.8.24    // ACC-84E Card 7 Chan 0 data reg B
ptr EncCard7Chan0DataC->u.io:$D08008.8.24    // ACC-84E Card 7 Chan 0 data reg C
ptr EncCard7Chan0DataD->u.io:$D0800C.8.24    // ACC-84E Card 7 Chan 0 data reg D
ptr EncCard7Chan1Cmd->u.io:$D08030.8.24      // ACC-84E Card 7 Chan 1 cmd reg
ptr EncCard7Chan1DataA->u.io:$D08010.8.24    // ACC-84E Card 7 Chan 1 data reg A
ptr EncCard7Chan1DataB->u.io:$D08014.8.24    // ACC-84E Card 7 Chan 1 data reg B
ptr EncCard7Chan1DataC->u.io:$D08018.8.24    // ACC-84E Card 7 Chan 1 data reg C
ptr EncCard7Chan1DataD->u.io:$D0801C.8.24    // ACC-84E Card 7 Chan 1 data reg D
ptr EncCard7Chan2Cmd->u.io:$D08060.8.24      // ACC-84E Card 7 Chan 2 cmd reg
ptr EncCard7Chan2DataA->u.io:$D08040.8.24    // ACC-84E Card 7 Chan 2 data reg A
ptr EncCard7Chan2DataB->u.io:$D08044.8.24    // ACC-84E Card 7 Chan 2 data reg B
ptr EncCard7Chan2DataC->u.io:$D08048.8.24    // ACC-84E Card 7 Chan 2 data reg C
ptr EncCard7Chan2DataD->u.io:$D0804C.8.24    // ACC-84E Card 7 Chan 2 data reg D
ptr EncCard7Chan3Cmd->u.io:$D08070.8.24      // ACC-84E Card 7 Chan 3 cmd reg
ptr EncCard7Chan3DataA->u.io:$D08050.8.24    // ACC-84E Card 7 Chan 3 data reg A
ptr EncCard7Chan3DataB->u.io:$D08054.8.24    // ACC-84E Card 7 Chan 3 data reg B
ptr EncCard7Chan3DataC->u.io:$D08058.8.24    // ACC-84E Card 7 Chan 3 data reg C
ptr EncCard7Chan3DataD->u.io:$D0805C.8.24    // ACC-84E Card 7 Chan 3 data reg D
```

ACC-84E Card 8

(SW1-1 ON, SW1-2 ON, SW1-3 ON, SW1-4 OFF)

```
ptr EncCard8Ctrl->u.io:$A1007C.8.24          // ACC-84E Card 8 control reg
ptr EncCard8Chan0Cmd->u.io:$A10020.8.24      // ACC-84E Card 8 Chan 0 cmd reg
ptr EncCard8Chan0DataA->u.io:$A10000.8.24    // ACC-84E Card 8 Chan 0 data reg A
ptr EncCard8Chan0DataB->u.io:$A10004.8.24    // ACC-84E Card 8 Chan 0 data reg B
ptr EncCard8Chan0DataC->u.io:$A10008.8.24    // ACC-84E Card 8 Chan 0 data reg C
ptr EncCard8Chan0DataD->u.io:$A1000C.8.24    // ACC-84E Card 8 Chan 0 data reg D
ptr EncCard8Chan1Cmd->u.io:$A10030.8.24      // ACC-84E Card 8 Chan 1 cmd reg
ptr EncCard8Chan1DataA->u.io:$A10010.8.24    // ACC-84E Card 8 Chan 1 data reg A
ptr EncCard8Chan1DataB->u.io:$A10014.8.24    // ACC-84E Card 8 Chan 1 data reg B
ptr EncCard8Chan1DataC->u.io:$A10018.8.24    // ACC-84E Card 8 Chan 1 data reg C
ptr EncCard8Chan1DataD->u.io:$A1001C.8.24    // ACC-84E Card 8 Chan 1 data reg D
ptr EncCard8Chan2Cmd->u.io:$A10060.8.24      // ACC-84E Card 8 Chan 2 cmd reg
ptr EncCard8Chan2DataA->u.io:$A10040.8.24    // ACC-84E Card 8 Chan 2 data reg A
ptr EncCard8Chan2DataB->u.io:$A10044.8.24    // ACC-84E Card 8 Chan 2 data reg B
ptr EncCard8Chan2DataC->u.io:$A10048.8.24    // ACC-84E Card 8 Chan 2 data reg C
ptr EncCard8Chan2DataD->u.io:$A1004C.8.24    // ACC-84E Card 8 Chan 2 data reg D
ptr EncCard8Chan3Cmd->u.io:$A10070.8.24      // ACC-84E Card 8 Chan 3 cmd reg
ptr EncCard8Chan3DataA->u.io:$A10050.8.24    // ACC-84E Card 8 Chan 3 data reg A
ptr EncCard8Chan3DataB->u.io:$A10054.8.24    // ACC-84E Card 8 Chan 3 data reg B
ptr EncCard8Chan3DataC->u.io:$A10058.8.24    // ACC-84E Card 8 Chan 3 data reg C
ptr EncCard8Chan3DataD->u.io:$A1005C.8.24    // ACC-84E Card 8 Chan 3 data reg D
```

ACC-84E Card 9

(SW1-1 OFF, SW1-2 ON, SW1-3 ON, SW1-4 OFF)

```
ptr EncCard9Ctrl->u.io:$B1007C.8.24          // ACC-84E Card 9 control reg
ptr EncCard9Chan0Cmd->u.io:$B10020.8.24      // ACC-84E Card 9 Chan 0 cmd reg
ptr EncCard9Chan0DataA->u.io:$B10000.8.24    // ACC-84E Card 9 Chan 0 data reg A
ptr EncCard9Chan0DataB->u.io:$B10004.8.24    // ACC-84E Card 9 Chan 0 data reg B
ptr EncCard9Chan0DataC->u.io:$B10008.8.24    // ACC-84E Card 9 Chan 0 data reg C
ptr EncCard9Chan0DataD->u.io:$B1000C.8.24    // ACC-84E Card 9 Chan 0 data reg D
ptr EncCard9Chan1Cmd->u.io:$B10030.8.24      // ACC-84E Card 9 Chan 1 cmd reg
ptr EncCard9Chan1DataA->u.io:$B10010.8.24    // ACC-84E Card 9 Chan 1 data reg A
ptr EncCard9Chan1DataB->u.io:$B10014.8.24    // ACC-84E Card 9 Chan 1 data reg B
ptr EncCard9Chan1DataC->u.io:$B10018.8.24    // ACC-84E Card 9 Chan 1 data reg C
ptr EncCard9Chan1DataD->u.io:$B1001C.8.24    // ACC-84E Card 9 Chan 1 data reg D
ptr EncCard9Chan2Cmd->u.io:$B10060.8.24      // ACC-84E Card 9 Chan 2 cmd reg
ptr EncCard9Chan2DataA->u.io:$B10040.8.24    // ACC-84E Card 9 Chan 2 data reg A
ptr EncCard9Chan2DataB->u.io:$B10044.8.24    // ACC-84E Card 9 Chan 2 data reg B
ptr EncCard9Chan2DataC->u.io:$B10048.8.24    // ACC-84E Card 9 Chan 2 data reg C
ptr EncCard9Chan2DataD->u.io:$B1004C.8.24    // ACC-84E Card 9 Chan 2 data reg D
ptr EncCard9Chan3Cmd->u.io:$B10070.8.24      // ACC-84E Card 9 Chan 3 cmd reg
ptr EncCard9Chan3DataA->u.io:$B10050.8.24    // ACC-84E Card 9 Chan 3 data reg A
ptr EncCard9Chan3DataB->u.io:$B10054.8.24    // ACC-84E Card 9 Chan 3 data reg B
ptr EncCard9Chan3DataC->u.io:$B10058.8.24    // ACC-84E Card 9 Chan 3 data reg C
ptr EncCard9Chan3DataD->u.io:$B1005C.8.24    // ACC-84E Card 9 Chan 3 data reg D
```

ACC-84E Card 10

(SW1-1 ON, SW1-2 OFF, SW1-3 ON, SW1-4 OFF)

```
ptr EncCard10Ctrl->u.io:$C1007C.8.24          // ACC-84E Card 10 control reg
ptr EncCard10Chan0Cmd->u.io:$C10020.8.24      // ACC-84E Card 10 Chan 0 cmd reg
ptr EncCard10Chan0DataA->u.io:$C10000.8.24    // ACC-84E Card 10 Chan 0 data reg A
ptr EncCard10Chan0DataB->u.io:$C10004.8.24    // ACC-84E Card 10 Chan 0 data reg B
ptr EncCard10Chan0DataC->u.io:$C10008.8.24    // ACC-84E Card 10 Chan 0 data reg C
ptr EncCard10Chan0DataD->u.io:$C1000C.8.24    // ACC-84E Card 10 Chan 0 data reg D
ptr EncCard10Chan1Cmd->u.io:$C10030.8.24      // ACC-84E Card 10 Chan 1 cmd reg
ptr EncCard10Chan1DataA->u.io:$C10010.8.24    // ACC-84E Card 10 Chan 1 data reg A
ptr EncCard10Chan1DataB->u.io:$C10014.8.24    // ACC-84E Card 10 Chan 1 data reg B
ptr EncCard10Chan1DataC->u.io:$C10018.8.24    // ACC-84E Card 10 Chan 1 data reg C
ptr EncCard10Chan1DataD->u.io:$C1001C.8.24    // ACC-84E Card 10 Chan 1 data reg D
ptr EncCard10Chan2Cmd->u.io:$C10060.8.24      // ACC-84E Card 10 Chan 2 cmd reg
ptr EncCard10Chan2DataA->u.io:$C10040.8.24    // ACC-84E Card 10 Chan 2 data reg A
ptr EncCard10Chan2DataB->u.io:$C10044.8.24    // ACC-84E Card 10 Chan 2 data reg B
ptr EncCard10Chan2DataC->u.io:$C10048.8.24    // ACC-84E Card 10 Chan 2 data reg C
ptr EncCard10Chan2DataD->u.io:$C1004C.8.24    // ACC-84E Card 10 Chan 2 data reg D
ptr EncCard10Chan3Cmd->u.io:$C10070.8.24      // ACC-84E Card 10 Chan 3 cmd reg
ptr EncCard10Chan3DataA->u.io:$C10050.8.24    // ACC-84E Card 10 Chan 3 data reg A
ptr EncCard10Chan3DataB->u.io:$C10054.8.24    // ACC-84E Card 10 Chan 3 data reg B
ptr EncCard10Chan3DataC->u.io:$C10058.8.24    // ACC-84E Card 10 Chan 3 data reg C
ptr EncCard10Chan3DataD->u.io:$C1005C.8.24    // ACC-84E Card 10 Chan 3 data reg D
```

ACC-84E Card 11

(SW1-1 OFF, SW1-2 OFF, SW1-3 ON, SW1-4 OFF)

```
ptr EncCard11Ctrl->u.io:$D1007C.8.24          // ACC-84E Card 11 control reg
ptr EncCard11Chan0Cmd->u.io:$D10020.8.24      // ACC-84E Card 11 Chan 0 cmd reg
ptr EncCard11Chan0DataA->u.io:$D10000.8.24    // ACC-84E Card 11 Chan 0 data reg A
ptr EncCard11Chan0DataB->u.io:$D10004.8.24    // ACC-84E Card 11 Chan 0 data reg B
ptr EncCard11Chan0DataC->u.io:$D10008.8.24    // ACC-84E Card 11 Chan 0 data reg C
ptr EncCard11Chan0DataD->u.io:$D1000C.8.24    // ACC-84E Card 11 Chan 0 data reg D
ptr EncCard11Chan1Cmd->u.io:$D10030.8.24      // ACC-84E Card 11 Chan 1 cmd reg
ptr EncCard11Chan1DataA->u.io:$D10010.8.24    // ACC-84E Card 11 Chan 1 data reg A
ptr EncCard11Chan1DataB->u.io:$D10014.8.24    // ACC-84E Card 11 Chan 1 data reg B
ptr EncCard11Chan1DataC->u.io:$D10018.8.24    // ACC-84E Card 11 Chan 1 data reg C
ptr EncCard11Chan1DataD->u.io:$D1001C.8.24    // ACC-84E Card 11 Chan 1 data reg D
ptr EncCard11Chan2Cmd->u.io:$D10060.8.24      // ACC-84E Card 11 Chan 2 cmd reg
ptr EncCard11Chan2DataA->u.io:$D10040.8.24    // ACC-84E Card 11 Chan 2 data reg A
ptr EncCard11Chan2DataB->u.io:$D10044.8.24    // ACC-84E Card 11 Chan 2 data reg B
ptr EncCard11Chan2DataC->u.io:$D10048.8.24    // ACC-84E Card 11 Chan 2 data reg C
ptr EncCard11Chan2DataD->u.io:$D1004C.8.24    // ACC-84E Card 11 Chan 2 data reg D
ptr EncCard11Chan3Cmd->u.io:$D10070.8.24      // ACC-84E Card 11 Chan 3 cmd reg
ptr EncCard11Chan3DataA->u.io:$D10050.8.24    // ACC-84E Card 11 Chan 3 data reg A
ptr EncCard11Chan3DataB->u.io:$D10054.8.24    // ACC-84E Card 11 Chan 3 data reg B
ptr EncCard11Chan3DataC->u.io:$D10058.8.24    // ACC-84E Card 11 Chan 3 data reg C
ptr EncCard11Chan3DataD->u.io:$D1005C.8.24    // ACC-84E Card 11 Chan 3 data reg D
```

ACC-84E Card 12

(SW1-1 ON, SW1-2 ON, SW1-3 OFF, SW1-4 OFF)

```
ptr EncCard12Ctrl->u.io:$A1807C.8.24          // ACC-84E Card 12 control reg
ptr EncCard12Chan0Cmd->u.io:$A18020.8.24      // ACC-84E Card 12 Chan 0 cmd reg
ptr EncCard12Chan0DataA->u.io:$A18000.8.24    // ACC-84E Card 12 Chan 0 data reg A
ptr EncCard12Chan0DataB->u.io:$A18004.8.24    // ACC-84E Card 12 Chan 0 data reg B
ptr EncCard12Chan0DataC->u.io:$A18008.8.24    // ACC-84E Card 12 Chan 0 data reg C
ptr EncCard12Chan0DataD->u.io:$A1800C.8.24    // ACC-84E Card 12 Chan 0 data reg D
ptr EncCard12Chan1Cmd->u.io:$A18030.8.24      // ACC-84E Card 12 Chan 1 cmd reg
ptr EncCard12Chan1DataA->u.io:$A18010.8.24    // ACC-84E Card 12 Chan 1 data reg A
ptr EncCard12Chan1DataB->u.io:$A18014.8.24    // ACC-84E Card 12 Chan 1 data reg B
ptr EncCard12Chan1DataC->u.io:$A18018.8.24    // ACC-84E Card 12 Chan 1 data reg C
ptr EncCard12Chan1DataD->u.io:$A1801C.8.24    // ACC-84E Card 12 Chan 1 data reg D
ptr EncCard12Chan2Cmd->u.io:$A18060.8.24      // ACC-84E Card 12 Chan 2 cmd reg
ptr EncCard12Chan2DataA->u.io:$A18040.8.24    // ACC-84E Card 12 Chan 2 data reg A
ptr EncCard12Chan2DataB->u.io:$A18044.8.24    // ACC-84E Card 12 Chan 2 data reg B
ptr EncCard12Chan2DataC->u.io:$A18048.8.24    // ACC-84E Card 12 Chan 2 data reg C
ptr EncCard12Chan2DataD->u.io:$A1804C.8.24    // ACC-84E Card 12 Chan 2 data reg D
ptr EncCard12Chan3Cmd->u.io:$A18070.8.24      // ACC-84E Card 12 Chan 3 cmd reg
ptr EncCard12Chan3DataA->u.io:$A18050.8.24    // ACC-84E Card 12 Chan 3 data reg A
ptr EncCard12Chan3DataB->u.io:$A18054.8.24    // ACC-84E Card 12 Chan 3 data reg B
ptr EncCard12Chan3DataC->u.io:$A18058.8.24    // ACC-84E Card 12 Chan 3 data reg C
ptr EncCard12Chan3DataD->u.io:$A1805C.8.24    // ACC-84E Card 12 Chan 3 data reg D
```

ACC-84E Card 13

(SW1-1 OFF, SW1-2 ON, SW1-3 OFF, SW1-4 OFF)

```
ptr EncCard13Ctrl->u.io:$B1807C.8.24          // ACC-84E Card 13 control reg
ptr EncCard13Chan0Cmd->u.io:$B18020.8.24      // ACC-84E Card 13 Chan 0 cmd reg
ptr EncCard13Chan0DataA->u.io:$B18000.8.24    // ACC-84E Card 13 Chan 0 data reg A
ptr EncCard13Chan0DataB->u.io:$B18004.8.24    // ACC-84E Card 13 Chan 0 data reg B
ptr EncCard13Chan0DataC->u.io:$B18008.8.24    // ACC-84E Card 13 Chan 0 data reg C
ptr EncCard13Chan0DataD->u.io:$B1800C.8.24    // ACC-84E Card 13 Chan 0 data reg D
ptr EncCard13Chan1Cmd->u.io:$B18030.8.24      // ACC-84E Card 13 Chan 1 cmd reg
ptr EncCard13Chan1DataA->u.io:$B18010.8.24    // ACC-84E Card 13 Chan 1 data reg A
ptr EncCard13Chan1DataB->u.io:$B18014.8.24    // ACC-84E Card 13 Chan 1 data reg B
ptr EncCard13Chan1DataC->u.io:$B18018.8.24    // ACC-84E Card 13 Chan 1 data reg C
ptr EncCard13Chan1DataD->u.io:$B1801C.8.24    // ACC-84E Card 13 Chan 1 data reg D
ptr EncCard13Chan2Cmd->u.io:$B18060.8.24      // ACC-84E Card 13 Chan 2 cmd reg
ptr EncCard13Chan2DataA->u.io:$B18040.8.24    // ACC-84E Card 13 Chan 2 data reg A
ptr EncCard13Chan2DataB->u.io:$B18044.8.24    // ACC-84E Card 13 Chan 2 data reg B
ptr EncCard13Chan2DataC->u.io:$B18048.8.24    // ACC-84E Card 13 Chan 2 data reg C
ptr EncCard13Chan2DataD->u.io:$B1804C.8.24    // ACC-84E Card 13 Chan 2 data reg D
ptr EncCard13Chan3Cmd->u.io:$B18070.8.24      // ACC-84E Card 13 Chan 3 cmd reg
ptr EncCard13Chan3DataA->u.io:$B18050.8.24    // ACC-84E Card 13 Chan 3 data reg A
ptr EncCard13Chan3DataB->u.io:$B18054.8.24    // ACC-84E Card 13 Chan 3 data reg B
ptr EncCard13Chan3DataC->u.io:$B18058.8.24    // ACC-84E Card 13 Chan 3 data reg C
ptr EncCard13Chan3DataD->u.io:$B1805C.8.24    // ACC-84E Card 13 Chan 3 data reg D
```

ACC-84E Card 14

(SW1-1 ON, SW1-2 OFF, SW1-3 OFF, SW1-4 OFF)

```
ptr EncCard14Ctrl->u.io:$C1807C.8.24          // ACC-84E Card 14 control reg
ptr EncCard14Chan0Cmd->u.io:$C18020.8.24      // ACC-84E Card 14 Chan 0 cmd reg
ptr EncCard14Chan0DataA->u.io:$C18000.8.24    // ACC-84E Card 14 Chan 0 data reg A
ptr EncCard14Chan0DataB->u.io:$C18004.8.24    // ACC-84E Card 14 Chan 0 data reg B
ptr EncCard14Chan0DataC->u.io:$C18008.8.24    // ACC-84E Card 14 Chan 0 data reg C
ptr EncCard14Chan0DataD->u.io:$C1800C.8.24    // ACC-84E Card 14 Chan 0 data reg D
ptr EncCard14Chan1Cmd->u.io:$C18030.8.24      // ACC-84E Card 14 Chan 1 cmd reg
ptr EncCard14Chan1DataA->u.io:$C18010.8.24    // ACC-84E Card 14 Chan 1 data reg A
ptr EncCard14Chan1DataB->u.io:$C18014.8.24    // ACC-84E Card 14 Chan 1 data reg B
ptr EncCard14Chan1DataC->u.io:$C18018.8.24    // ACC-84E Card 14 Chan 1 data reg C
ptr EncCard14Chan1DataD->u.io:$C1801C.8.24    // ACC-84E Card 14 Chan 1 data reg D
ptr EncCard14Chan2Cmd->u.io:$C18060.8.24      // ACC-84E Card 14 Chan 2 cmd reg
ptr EncCard14Chan2DataA->u.io:$C18040.8.24    // ACC-84E Card 14 Chan 2 data reg A
ptr EncCard14Chan2DataB->u.io:$C18044.8.24    // ACC-84E Card 14 Chan 2 data reg B
ptr EncCard14Chan2DataC->u.io:$C18048.8.24    // ACC-84E Card 14 Chan 2 data reg C
ptr EncCard14Chan2DataD->u.io:$C1804C.8.24    // ACC-84E Card 14 Chan 2 data reg D
ptr EncCard14Chan3Cmd->u.io:$C18070.8.24      // ACC-84E Card 14 Chan 3 cmd reg
ptr EncCard14Chan3DataA->u.io:$C18050.8.24    // ACC-84E Card 14 Chan 3 data reg A
ptr EncCard14Chan3DataB->u.io:$C18054.8.24    // ACC-84E Card 14 Chan 3 data reg B
ptr EncCard14Chan3DataC->u.io:$C18058.8.24    // ACC-84E Card 14 Chan 3 data reg C
ptr EncCard14Chan3DataD->u.io:$C1805C.8.24    // ACC-84E Card 14 Chan 3 data reg D
```

ACC-84E Card 15

(SW1-1 OFF, SW1-2 OFF, SW1-3 OFF, SW1-4 OFF)

```
ptr EncCard15Ctrl->u.io:$D1807C.8.24          // ACC-84E Card 15 control reg
ptr EncCard15Chan0Cmd->u.io:$D18020.8.24      // ACC-84E Card 15 Chan 0 cmd reg
ptr EncCard15Chan0DataA->u.io:$D18000.8.24    // ACC-84E Card 15 Chan 0 data reg A
ptr EncCard15Chan0DataB->u.io:$D18004.8.24    // ACC-84E Card 15 Chan 0 data reg B
ptr EncCard15Chan0DataC->u.io:$D18008.8.24    // ACC-84E Card 15 Chan 0 data reg C
ptr EncCard15Chan0DataD->u.io:$D1800C.8.24    // ACC-84E Card 15 Chan 0 data reg D
ptr EncCard15Chan1Cmd->u.io:$D18030.8.24      // ACC-84E Card 15 Chan 1 cmd reg
ptr EncCard15Chan1DataA->u.io:$D18010.8.24    // ACC-84E Card 15 Chan 1 data reg A
ptr EncCard15Chan1DataB->u.io:$D18014.8.24    // ACC-84E Card 15 Chan 1 data reg B
ptr EncCard15Chan1DataC->u.io:$D18018.8.24    // ACC-84E Card 15 Chan 1 data reg C
ptr EncCard15Chan1DataD->u.io:$D1801C.8.24    // ACC-84E Card 15 Chan 1 data reg D
ptr EncCard15Chan2Cmd->u.io:$D18060.8.24      // ACC-84E Card 15 Chan 2 cmd reg
ptr EncCard15Chan2DataA->u.io:$D18040.8.24    // ACC-84E Card 15 Chan 2 data reg A
ptr EncCard15Chan2DataB->u.io:$D18044.8.24    // ACC-84E Card 15 Chan 2 data reg B
ptr EncCard15Chan2DataC->u.io:$D18048.8.24    // ACC-84E Card 15 Chan 2 data reg C
ptr EncCard15Chan2DataD->u.io:$D1804C.8.24    // ACC-84E Card 15 Chan 2 data reg D
ptr EncCard15Chan3Cmd->u.io:$D18070.8.24      // ACC-84E Card 15 Chan 3 cmd reg
ptr EncCard15Chan3DataA->u.io:$D18050.8.24    // ACC-84E Card 15 Chan 3 data reg A
ptr EncCard15Chan3DataB->u.io:$D18054.8.24    // ACC-84E Card 15 Chan 3 data reg B
ptr EncCard15Chan3DataC->u.io:$D18058.8.24    // ACC-84E Card 15 Chan 3 data reg C
ptr EncCard15Chan3DataD->u.io:$D1805C.8.24    // ACC-84E Card 15 Chan 3 data reg D
```

POWER PMAC TURBO I-VARIABLE EQUIVALENTS

This section is intended to facilitate the conversion of a working Turbo PMAC application to the Power PMAC with minimum changes. It documents the Power PMAC equivalents to Turbo PMAC setup I-variables and describes whether changes to these settings are required or not. Note that a direct conversion by straight application of these tables usually will not utilize Power PMAC's capabilities to its fullest. However, it may provide a good starting point for further enhancements.

Notes on Equivalents

1. Addressing variables in Power PMAC can, and usually will, be set to ***{data structure element name}*.a**. Numerical values of addresses do not need to be known, but will be different from equivalents in Turbo PMAC.

2. Position units for this variable in Power PMAC are "motor units", which can be, but do not have to be "counts" of the encoder. In applications where the simplest possible porting from Turbo PMAC is desired, motor units should be set at "counts" of the encoder or equivalent.

3. This variable is an integer (or at least a fixed-point value) in Turbo PMAC, but a floating-point value in Power PMAC, which provides increased resolution, and often range.

4. Motor position scale factors work significantly differently in the floating-point motor mathematics of the Power PMAC compared to the fixed-point motor mathematics. Changing the Power PMAC scale factors actually changes the working units of the motors (which permits the motors to be scaled in engineering units such as millimeters or degrees). However, for the simplest conversion of a Turbo PMAC application, it is recommended that the motor be left in units of counts of the feedback device, so **Motor[*x*].PosSf** and **Motor[*x*].Pos2Sf** should be set to their default values of 1.0 in these applications.

5. Turbo PMAC I-variable number can be used as "legacy" equivalent for matching Power PMAC setup element (for Motors 1 – 32, Coordinate Systems 1 – 16, MACRO ICs 0 – 3, and Servo ICs 0 – 9).

Global I-Variables

This section lists the "global" Turbo PMAC I-variables and their Power PMAC equivalents.

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
I0	--	No serial port host communications on Power PMAC	
I1	--	No serial port host communications on Power PMAC	
I2	--	No control panel port on Turbo PMAC2 or Power PMAC	
I3	--	Fixed handshaking characters in Power PMAC	
I4	--	Communications integrity handled at TCP/IP level	
I5	--	Power PMAC always at equivalent of I5 = 3 (all PLCs can be enabled). Commands in file `pp_startup.txt` can be used to control which PLC programs enabled at startup	
I6	--	Fixed error reporting protocol in Power PMAC	
I7	**Sys.PhaseCycleExt**	Same value provides compatible operation	
I8	**Sys.RtIntPeriod**	Same value provides compatible operation	
I9 bit 0	**echo** command value bits 0 – 2	**echo** value provides short-form/long-form response option separately for different query types	
I9 bit 1	**echo** value bit 3	**echo** bit polarity is opposite from I9 bit; = 0 for hex response, = 1 for decimal	
I10	**Sys.ServoPeriod**	Set **ServoPeriod** = I10/8388608 for compatible operation	
I11	--	Motion programs always start execution as soon as possible after run command in Power PMAC	
I12	--	Improved lookahead algorithms in Power PMAC eliminate need for this variable	
I13	--	All in-position checking is done in foreground in Power PMAC	
I14	--	Buffers can be defined in pp_startup.txt file commands	
I15	--	Trigonometric functions with "d" ending (e.g. **sind**, **atan2d**) use degrees; those without (e.g. **sin**, **atan2**) use radians	
I16 – I18	--	Program buffer handshaking handled automatically by OS utilities	
I19	--	ASIC clock source specified by individual **Gate*n*[*i*].PhaseServoDir** saved elements	

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
I20 – I23	--	MACRO IC numbers automatically assigned based on addresses; not limited to 4 ICs in Power PMAC	
I24	--	No DPRAM in Power PMAC	
I26	--	Only fiber MACRO automatically supported by Power PMAC	
I27	**MuxIo.InBit**	**InBit** = 8 for I27 = 0; **InBit** = 15 for I27 = 1	
I28	--	No display port support in Power PMAC	
I29	**MuxIo.pIn** **MuxIo.pOut**	No default address in Power PMAC	1
I30	**CompTable[*m*].Ctrl**	Rollover individually controllable for each dimension of each table	
I36	--	Abort and enable functions always separate in Power PMAC (equivalent of I36 = 1)	
I37	**Sys.BusCtrl[*n*]**	No memory wait state control; I/O wait state control is significantly different	
I38	--	Improved Power PMAC standard functionality eliminates need for this	
I39	--	Configuration information presented differently in Power PMAC	
I40	**Sys.WDTReset**	Default settings of 0 provide effectively compatible operation	
I41	--	Lockout function not supported in Power PMAC	
I42	--	Improved Power PMAC standard functionality eliminates need for this	
I43	--	No serial port host communications supported on Power PMAC	
I44	--	Ladder logic program executed as independent application on Power PMAC	
I45	--	Improved Power PMAC standard functionality eliminates need for this	
I46	--	No battery-backed RAM in Power PMAC	
I47 – I50	--	No DPRAM in Power PMAC	
I51	**Sys.CompEnable**	Setting **CompEnable** > maximum table number enables all tables	
I52	--	CPU operating frequency automatically set	
I53 – I54	--	Serial port baudrates set in OS file etc/inittab; default to 115,200	
I55 – I58		No DPRAM in Power PMAC	
I59	--	No control panel on Turbo PMAC2 or Power PMAC	

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
I60 – I61	--	Filtered velocity sample period fixed at 16 servo cycles in Power PMAC	
I62	**Sys.SendFileMode**	Operation is somewhat different in Power PMAC	
I63 – I64	--	Improved Power PMAC standard functionality eliminates need for this	
I65	--	Power PMAC project management files provide other methods of showing user configuration	
I67	**Modbus[*i*]** structure	No need to specify data structure address in Power PMAC	
I68	**Sys.MaxCoords** **Coord[*x*].SyncOps**	Default Power PMAC settings provide compatible operation with any Turbo PMAC settings	
I69	--	No automatic Modbus control panel structure in Power PMAC	
I70, I72, I74, I76	--	Improved Power PMAC standard functionality eliminates need for this	
I71, I73, I75, I77	--	Only Type 1 MACRO protocol supported in Power PMAC	
I78	**Macro.IOTimeout**	I78 is in servo cycles, **IOTimeout** is in milliseconds	
I79	**Macro.IOTimeout**	I79 is in servo cycles, **IOTimeout** is in milliseconds	
I80	**Macro.TestPeriod**	Same value provides compatible operation	
I81	**Macro.TestMaxErrors**	Same value provides compatible operation	
I82	**Macro.TestReqdSynchs**	Same value provides compatible operation	
I83	--	Automatically set in Power PMAC	
I84	--	Automatically set in Power PMAC	
I85	**Macro.Station**	Automatically detected	
I90 – I99	--	No VME bus interface in Power PMAC	

Motor I-Variables

This section lists the motor-specific Turbo PMAC I-variables and their Power PMAC equivalents. The Power PMAC motor structure index "***x***" is equivalent to the "hundreds" value of the Turbo PMAC I-variable number (e.g. **Motor[17].ServoCtrl** is equivalent to I1700).

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
Ixx00	**Motor[*x*].ServoCtrl**	Same value provides compatible operation	5
Ixx01	**Motor[*x*].PhaseCtrl**	Set to 4 (not 1) in Power PMAC to commutate with PMAC2-style ICs	5
Ixx02	**Motor[*x*].pDac**	Default value in Power PMAC usually equivalent to default value in Turbo PMAC system	1,5
Ixx03	**Motor[*x*].pEnc**	Default value in Power PMAC usually functionally equivalent to default value in Turbo PMAC system	1,5
Ixx04	**Motor[*x*].pEnc2**	Default value in Power PMAC usually functionally equivalent to default value in Turbo PMAC system	1,5
Ixx05	**Motor[*x*].pMasterEnc**	Must be set to encoder table entry address	1,5
Ixx06	**Motor[*x*].MasterCtrl**	Same value provides compatible operation	5
Ixx07	**Motor[*x*].MasterPosSf**	Floating-point value; specifies following "gear ratio" by itself in motor units per master position unit (= Ixx07/Ixx08 in Turbo PMAC)	5
Ixx08	**Motor[*x*].PosSf**	Must set **PosSf** to 1.0 to set units of motor to counts of the feedback device	4,5
Ixx09	**Motor[*x*].Pos2Sf**	Must set **Pos2Sf** to 1.0 to set units of motor to counts of the feedback device	4,5
Ixx10	**Motor[*x*].pAbsPos**	Address specified by element name	1,5
Ixx11	**Motor[*x*].FatalFeLimit**	Units are whole motor units, not 1/16 count; set **FatalFeLimit =** Ixx11/16 for compatible operation when motor units are counts	2,3,5
Ixx12	**Motor[*x*].WarnFeLimit**	Units are whole motor units, not 1/16 count; set **WarnFeLimit =** Ixx12/16 for compatible operation when motor units are counts	2,3,5
Ixx13	**Motor[*x*].MaxPos**	Same value provides compatible operation if motor units are counts	2,3,5
Ixx14	**Motor[*x*].MinPos**	Same value provides compatible operation if motor units are counts	2,3,5
Ixx15	**Motor[*x*].AbortTa**	To specify deceleration rate (not time) on abort, **AbortTa** must be set < 0, expressing an inverse rate, so set **AbortTa** = -1/Ixx15 for compatible operation if motor units are counts	2,5

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
Ixx16	**Motor[*x*].MaxSpeed**	Same value provides compatible operation if motor units are counts	2,5
Ixx17	**Motor[*x*].InvAmax** **Motor[*x*].InvDmax**	As inverse, Power PMAC value is reciprocal of equivalent Turbo PMAC value. Set **InvAmax** = 1/Ixx17 for compatible operation if motor units are counts. **InvDmax** used instead for final deceleration of a sequence.	2,5
Ixx19	**Motor[*x*].JogTa**	Jog-move acceleration rate can be specified (not just limited) by setting **JogTa** to negative value, expressing an inverse rate.	
Ixx20	**Motor[*x*].JogTa**	Total accel time is Ta+Ts, not just Ta, when Ta > Ts	5
Ixx21	**Motor[*x*].JogTs**	Total accel time is Ta+Ts, not just Ta, when Ta > Ts	5
Ixx22	**Motor[*x*].JogSpeed**	Same value provides compatible operation if motor units are counts	2,5
Ixx23	**Motor[*x*].HomeVel**	Same value provides compatible operation if motor units are counts	2,5
Ixx24 bit 23	**Motor[*x*].AmpFaultLevel**	Same value provides compatible operation	
Ixx24 bits 22-21	**Motor[*x*].FaultMode**	**FaultMode** bit 0 controls whether other motors in C.S. are killed or aborted on killing fault of this motor. Motors in other C.S.'s are never affected	
Ixx24 bit 20	**Motor[*x*].pAmpFault**	Set **pAmpFault** to 0 to disable amplifier fault check	
Ixx24 bits 19-18	--	No need to specify special mode for MACRO flags in Power PMAC	
Ixx24 bit 17	**Motor[*x*].pLimits**	Set **pLimits** to 0 to disable hardware overtravel limit check	
Ixx24 bit 16	**Motor[*x*].pAmpEnable**	Set **pAmpEnable** to 0 to disable output of amplifier enable signal	
Ixx24 bit 15	--	Instantaneous desired positions always checked against software overtravel limits	
Ixx24 bit 14	**Motor[*x*]. SoftLimitStopDis**	Same value provides compatible operation	
Ixx24 bit 13	--	Function not supported in Turbo PMAC2 or Power PMAC	
Ixx24 bit 12	**Motor[*x*]. CaptPosRightShift** **Motor[*x*]. CaptPosLeftShift**	Can shift out captured fractional count value or not with **CaptPosRightShift** and **CaptPosLeftShift**	

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
Ixx24 bit 11	**Motor[*x*]. CaptPosRightShift** **Motor[*x*]. CaptPosLeftShift**	Can use **CaptPosRightShift** and **CaptPosLeftShift** to match resolution of captured position with that of servo position	
Ixx24 bit 10	**Motor[*x*].PhaseMode** bit 0	Same value provides compatible operation	
Ixx24 bit 8	**Motor[*x*].FaultMode** bit 2	Same value provides compatible operation	
Ixx24 bit 1	**Motor[*x*].AmpFaultBit** **Motor[*x*].AmpEnableBit** **Motor[*x*].LimitBits** **Motor[*x*].CaptFlagBit**	In Power PMAC, each flag is specified by register address and bit number within that register for maximum flexibility; not just limited to one style of IC for all flags	
Ixx25	**Motor[*x*].pCaptFlag**	Default value in Power PMAC usually functionally equivalent to default value in Turbo PMAC system	1
Ixx26	**Motor[*x*].HomeOffset**	Units are whole motor units, not 1/16 count; set **HomeOffset** = Ixx26/16 for compatible operation when motor units are counts	2,3,5
Ixx27	**Coord[*x*].PosRollover[*i*]**	Rotary axis rollover calculations are done for the axis, not the motor, in Power PMAC. Units are axis units.	
Ixx28	**Motor[*x*].InPosBand**	Units are whole motor units, not 1/16 count; set **InPosBand** = Ixx28/16 for compatible operation when motor units are counts	2,3,5
Ixx29	**Motor[*x*].DacBias** **Motor[*x*].IaBias**	Use **DacBias** when not commutating motor in Power PMAC. Use **IaBias** when commutating motor in Power PMAC. Same value provides compatible operation	
Ixx30	**Motor[*x*].Servo.Kp**	See *Equations for Servo Gains*, below	3,5
Ixx31	**Motor[*x*].Servo.Kvfb**	See *Equations for Servo Gains*, below	3,5
Ixx32	**Motor[*x*].Servo.Kvff**	See *Equations for Servo Gains*, below	3,5
Ixx33	**Motor[*x*].Servo.Ki**	See *Equations for Servo Gains*, below	3,5
Ixx34	**Motor[*x*].Servo.SwZvInt**	Same value provides compatible operation	5
Ixx35	**Motor[*x*].Servo.Kaff**	See *Equations for Servo Gains*, below	3,5
Ixx36	**Motor[*x*].Servo.Kc1**	Same value provides compatible operation	3,5
Ixx37	**Motor[*x*].Servo.Kc2**	Same value provides compatible operation	3,5
Ixx38	**Motor[*x*].Servo.Kd1**	Same value provides compatible operation	3,5
Ixx39	**Motor[*x*].Servo.Kd2**	Same value provides compatible operation	3,5
Ixx40	**Motor[*x*].Pn0** **Motor[*x*].Pd1** **Motor[*x*].PreFilterEna**	For compatible operation, Set **Pn0** = Ixx40, **Pd1** = -Ixx40 **PreFilterEna** = 1	

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
Ixx41	**Motor[*x*].SoftLimitOffset**	Same value provides compatible operation if motor units are counts	2,3,5
Ixx42	**Motor[*x*].pAmpEnable** **Motor[*x*].pAmpFault**	If set to 0 in Power PMAC, disables functionality of flag (does not revert to Ixx25 address as in Turbo PMAC)	1
Ixx43	**Motor[*x*].pLimits**	If set to 0 in Power PMAC, disables functionality of flag (does not revert to Ixx25 address as in Turbo PMAC)	1
Ixx44	**Motor[*x*].pMotorNode** **Motor[*x*].MotorMode**	Address and mode of operation split into two variables in Power PMAC.	1
Ixx55	**Motor[*x*].pSineTable** **Motor[*x*].pVoltSineTable**	For default Ixx55 of 0, set these to default of **Sys.SineTable[0].a**. For non-default addresses, refer to manual.	1
Ixx56	**Motor[*x*].AdvGain**	Set **AdvGain** = Ixx56*Ixx09*32 for compatible operation	3,5
Ixx57	**Motor[*x*].I2tSet**	Same value provides compatible operation	3,5
Ixx58	**Motor[*x*].I2tTrip**	Set **I2tTrip** = Ixx58*2^{30}***Sys.ServoPeriod**/1000 for compatible operation	3,5
Ixx59	**Motor[*x*].Ctrl**	In Power PMAC, **Motor[*x*].Ctrl** specifies the address of the selected algorithm, permitting a choice among multiple built-in or customer algorithms	
Ixx60	**Motor[*x*].Stime**	Same value provides compatible operation	5
Ixx61	**Motor[*x*].IiGain**	Set **IiGain** = 8*Ixx61 for compatible operation	3,5
Ixx62	**Motor[*x*].IpfGain**	Set **IpfGain** = 4*Ixx62 for compatible operation	3,5
Ixx63	**Motor[*x*].Servo.MaxInt**	See *Equations for Servo Gains*, below. Negative values for **MaxInt** not supported in Power PMAC	5
Ixx64	**Motor[*x*].Servo.Kbreak**	Set **Kbreak** = (Ixx64+16)/16 for compatible operation	3,5
Ixx65	**Motor[*x*].Servo. BreakPosErr**	Units are whole motor units, not 1/16 count. Set **BreakPosErr** = Ixx65/16 for compatible operation	2,3,5
Ixx66	**Motor[*x*].PwmSf**	Same value provides compatible operation	5
Ixx67	**Motor[*x*].Servo. MaxPosErr**	Units are whole motor units, not 1/16 count; set **MaxPosErr** = Ixx67/16 for compatible operation when motor units are counts	5
Ixx68	**Motor[*x*].Servo.Kfff**	Same value provides compatible operation	3,5
Ixx69	**Motor[*x*].MaxDac**	Same value provides compatible operation	3,5
Ixx70	**Motor[*x*].PhasePosSf**	Set **PhasePosSf** = 2048/(256*Ixx71/Ixx70) for compatible operation	3

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
Ixx71	**Motor[*x*].PhasePosSf**	Set **PhasePosSf** = 2048/(256*Ixx71/Ixx70) for compatible operation	3,5
Ixx72	**Motor[*x*].PhaseOffset**	Same value provides compatible operation	5
Ixx73	**Motor[*x*]. PhaseFindingDac**	Same value provides compatible operation	5
Ixx74	**Motor[*x*]. PhaseFindingTime**	For "guess" method, same value provides compatible operation. For "stepper" method, multiply Ixx74 value by 256 for compatible operation	5
Ixx75	**Motor[*x*]. AbsPhasePosOffset** **Motor[*x*]. AbsPhasePosForce**	For absolute position read, set **AbsPhasePosOffset** = Ixx75*Ixx70 for compatible operation. For correction, set **AbsPhasePosForce** = Ixx75*2048/(Ixx71/Ixx70) for compatible operation	
Ixx76	**Motor[*x*].IpbGain**	Set **IpbGain** = 4*Ixx76 for compatible operation	3,5
Ixx77	**Motor[*x*].IdCmd**	Same value provides compatible operation	3,5
Ixx78	**Motor[*x*].DtOverRotorTc**	Same value provides compatible operation	3,5
Ixx79	**Motor[*x*].IbBias**	Same value provides compatible operation	3,5
Ixx80	**Motor[*x*].PowerOnMode** **Motor[*x*]. PhaseFindingTime**	Bit 0 of Ixx80 split into bits 0 &1 of **PowerOnMode**. Bit 1 of Ixx80 uses threshold of **PhaseFindingTime** for method selection.	
Ixx81	**Motor[*x*].pAbsPhasePos**	Address specified by element name	1,5
Ixx82	**Motor[*x*].pAdc**	Default value in Power PMAC usually functionally equivalent to default value in Turbo PMAC system. Specifies 1st, not 2nd of pair of registers.	1,5
Ixx83	**Motor[*x*].pPhaseEnc**	Default value in Power PMAC usually functionally equivalent to default value in Turbo PMAC system	1,5
Ixx84	**Motor[*x*].AdcMask**	32-bit mask in Power PMAC vs. 24-bit in Turbo PMAC; set **AdcMask =** Ixx84*$100 (*256) for compatible operation.	5
Ixx85	**Motor[*x*].BlSlewRate**	Units are whole motor units per real-time interrupt, not 1/16 count per background cycle	2,5
Ixx86	**Motor[*x*].BlSize**	Units are whole motor units, not 1/16 count; set **BlSize =** Ixx86/16 for compatible operation when motor units are counts	2,5

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments on Power PMAC Implementation	Notes
Ixx87	**Motor[*x*].BlHysteresis**	Units are whole motor units, not 1/16 count; set **BlHysteresis =** Ixx87/16 for compatible operation when motor units are counts	2,5
Ixx88	**Motor[*x*].InPosTime**	Units of servo cycles, not background cycles	5
Ixx90	**Motor[*x*].RapidSpeedSel**	Same value provides compatible operation	5
Ixx91	**Motor[*x*]. AbsPhasePosFormat**	This variable is structured very differently in Power PMAC – refer to detailed description	5
Ixx92	--	Jog moves are always calculated and started in the next servo cycle in Power PMAC	
Ixx95	**Motor[*x*].AbsPosFormat**	This variable is structured very differently in Power PMAC – refer to detailed description	5
Ixx96 bit 0	**Motor[*x*].PhaseMode** bit 1	Same value provides compatible operation	
Ixx96 bit 1	**Motor[*x*].Servo.SwFffInt** **Motor[*x*]. Servo.Kvifb** **Motor[*x*].Servo.Kviff**	Same value of **SwFffInt** provides compatible operation of friction feedforward. For compatible operation with Ixx96 bit 1 = 0, use **Kvfb** and **Kvff** velocity gains; For compatible operation with Ixx96 bit 1 = 1, use **Kvifb** and **Kviff** velocity gains	
Ixx97	**Motor[*x*].CaptureMode**	If Ixx97 = 0 or 1, set **CaptureMode** = Ixx97 for compatible operation. If Ixx97 = 3, set **CaptureMode** = 2 for compatible operation.	5
Ixx98	--	Automatic calculation of absolute position from geared resolvers not supported in Power PMAC	
Ixx99	--	Automatic calculation of absolute position from geared resolvers not supported in Power PMAC	

Equations for Servo Gains

The basic Power PMAC servo loop is structured and scaled differently from the Turbo PMAC servo loop. To calculate the required Power PMAC servo element values to match the operation of Turbo PMAC I-variables, the following equations can be used:

$$Motor[x].Servo.Kp = Ixx30 * \frac{Ixx08}{2^{19} * Motor[x].PosSf}$$

$$Motor[x].Servo.Kvfb = Ixx31 * \frac{Ixx09 * Ixx30}{2^{26} * Motor[x].Pos2Sf}$$

$$Motor[x].Servo.Kvff = Ixx32 * \frac{Ixx08 * Ixx30}{2^{26} * Motor[x].PosSf}$$

$$Motor[x].Servo.Ki = \frac{Ixx33}{2^{23}}$$

$$Motor[x].Servo.Kaff = Ixx35 * \frac{Ixx08 * Ixx30}{2^{26} * Motor[x].PosSf}$$

$$Motor[x].Servo.MaxInt = Ixx63 * \frac{Ixx08 * Ixx30}{2^{23}}$$

Notes:

- Those users wishing the simplest possible conversion from Turbo PMAC to Power PMAC will scale their Power PMAC motors in counts of the feedback device, so will leave **Motor[*x*].PosSf** and **Motor[*x*].Pos2Sf** at their default values of 1.0.

- If Turbo PMAC variable Ixx96 bit 1 (value 2) is set to 1 to place the integrator in the velocity loop instead of the position loop, Power PMAC elements **Motor[*x*].Servo.Kvifb** and **Motor[*x*].Servo.Kviff** should be used instead of **Motor[*x*].Servo.Kvfb** and **Motor[*x*].Servo.Kvff**, respectively, in the above equations.

Data Gathering I-Variables

Note that the data gathering setup elements in Power PMAC cannot be saved to flash memory.

Turbo PMAC I-Var	Equivalent Power PMAC Setup Element	Comments	Notes
I5000 Bit 0	--	Power PMAC has no DPRAM, so gathering is always to main memory. Uploading data while gathering is active is possible in Power PMAC.	
I5000 Bit 1	**Gather.Enable**	**Enable** = 2 sets gathering without rollover **Enable** = 3 sets gathering with rollover	
I5001 – I5048	**Gather.Addr[*i*]**	Typically specified by name of element, not numerical address value	1
I5049	**Gather.Period**	Same value provides compatible operation	
I5050 – I5051	**Gather.Items**	**Items** is not a mask like I5050 and I5051; sources from **Addr[0]** to **Addr[*Items*-1]** are gathered	

ADC De-multiplexing I-Variables

These setup variables govern the processing of multiplexed ADCs in ACC-36E and ACC-59E.

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments	Notes
I5060	**AdcDemux.Enable**	**Enable** controls both ring size and on/off functionality	
I5061 – I5076	**AdcDemux.Address[*i*]**	Typically specified by name of element, not numerical address value	1
I5080	**AdcDemux.Enable**	**Enable** controls both ring size and on/off functionality	
I5081 – I5096	**AdcDemux. ConvertCode[*i*]**	In Power PMAC, both of a pair of ADCs must be used the same way (unipolar or bipolar	

Coordinate System I-Variables

This section lists the coordinate-system-specific Turbo PMAC I-variables and their Power PMAC equivalents. The Power PMAC coordinate-system structure index "***x***" is equivalent to the "hundreds" value of the Turbo PMAC I-variable number minus 50 (e.g. **Coord[3].SegMoveTime** is equivalent to I5313).

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments	Notes
Isx11	--	Not directly supported in Power PMAC. Use status element **Sys.ServoCount** and user variable for closest equivalent. **Sys.CdTimer[*i*]** (in seconds) available in V2.2 and newer	
Isx12	--	Not directly supported in Power PMAC. Use status element **Sys.ServoCount** and user variable for closest equivalent. **Sys.CdTimer[*i*]** (in seconds) available in V2.2 and newer	
Isx13	**Coord[*x*].SegMoveTime**	Same value provides compatible operation	3,5
Isx14	--	(*Not supported in Power PMAC*)	
Isx15	**Coord[*x*].SegOverride**	In Power PMAC, equivalent value is 1.0 greater than in Turbo PMAC (1.0 is real time, not 0.0). Not saved in Power PMAC	5
Isx16	**Coord[*x*].SegOverrideSlew**	Same value provides compatible operation	5
Isx20	**Coord[*x*].LHDistance**	Same value provides compatible operation	5
Isx21	--	Program direct commands **`lh\`**, **`lh<`**, and **`lh>`** are used to change lookahead state quickly	
Isx50	--	If kinematics routines are present for a coordinate system in Power PMAC, they are automatically executed.	
Isx53	**Coord[*x*].StepMode**	Same value provides compatible operation	
Isx78	**Coord[*x*].MaxCirAccel**	Same value provides compatible operation	5
Isx79	**Coord[*x*].RapidVelCtrl**	Same value provides compatible operation	5
Isx81	**Coord[*x*].InPosTimeOut**	Same value provides compatible operation	
Isx82	**Coord[*x*]. AddedDwellTime**	Same value provides compatible operation	5
Isx83	**Coord[*x*].CornerBlendBp**	Same value provides compatible operation	5
Isx84	**Coord[*x*].CCCtrl** bit 0	Same value provides compatible operation	5
Isx85	**Coord[*x*].CornerDwellBp**	Same value provides compatible operation	5
Isx86	**Coord[*x*].AltFeedRate**	Same value provides compatible operation	5
Isx87	**Coord[*x*].Ta** **Coord[*x*].Td**	Total accel time is **Ta**+**Ts**, not just **Ta**, when **Ta** > **Ts**. **Td** used for final deceleration of a blended move sequence	

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments	Notes
Isx88	**Coord[*x*].Ts**	Total accel time is **Ta**+**Ts**, not just **Ta**, when **Ta** > **Ts**	5
Isx89	**Coord[*x*].Tm**	Feedrate/move-time distinction set by sign of this element, not by separate bit (negative is feedrate). Note that the value of this element is usually set automatically by program commands	5
Isx90	**Coord[*x*].FeedTime**	Same value provides compatible operation	5
Isx91	--	No support for control panel in Turbo PMAC2 or Power PMAC	
Isx92	**Coord[*x*].NoBlend**	Same value provides compatible operation	5
Isx93	**Coord[*x*].pDesTimeBase**	Default value in Power PMAC usually equivalent to default value in Turbo PMAC system	1,5
Isx94	**Coord[*x*].TimeBaseSlew**	Set **TimeBaseSlew** to Isx94/2^{23} for compatible operation	5
Isx95	**Coord[*x*].FeedHoldSlew**	Set **FeedHoldSlew** to Isx95/2^{23} for compatible operation	5
Isx96	**Coord[*x*]. RadiusErrorLimit**	Same value provides compatible operation	5
Isx97	**Coord[*x*].MinArcLen**	If Isx97 > 0, set **MinArcLen** to π*Isx97 for compatible operation If Isx97 = 0, set **MinArcLen** to π*2^{-20} for compatible operation	5
Isx98	**Coord[*x*].MaxFeedRate**	Same value provides compatible operation	5
Isx99	**Coord[*x*].CCAddedArcBp**	Same value provides compatible operation	5

PMAC2-Style MACRO IC I-Variables

Because the I-variables for setup of the ICs are in hardware registers, they are identical in function between Turbo PMAC and Power PMAC. Only their names have been changed.

In most cases, the Power PMAC MACRO IC structure index "***i***" is equivalent to the Turbo PMAC MACRO IC number (0 – 3). The Power PMAC channel sub-structure index "***j***" is equivalent to the tens digit "*n*" of the Turbo PMAC I-variable number minus one (e.g. **Gate2[0].Chan[0].EncCtrl** is equivalent to I6810).

Often, an "alias" for the **Gate2[*i*]** data structure representing the name of the hardware utilizing the IC is used instead, e.g. **Acc5E[*i*]**.

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments	Notes
I6m00/ I6m50	**Gate2[*i*].PwmPeriod**	Same value provides compatible operation	5
I6m01/ I6m51	**Gate2[*i*].PhaseClockDiv**	Same value provides compatible operation	5
I6m02/ I6m52	**Gate2[*i*].ServoClockDiv**	Same value provides compatible operation	5
I6m03/ I6m53	**Gate2[*i*]. HardwareClockCtrl**	Same value provides compatible operation	5
I6m04/ I6m54	**Gate2[*i*].PwmDeadTime**	Same value provides compatible operation	5
I6m05/ I6m55	**Gate2[*i*].DacStrobe**	Same value provides compatible operation	5
I6m06/ I6m56	**Gate2[*i*].AdcStrobe**	Same value provides compatible operation	5
I6m07/ I6m57	**Gate2[*i*].PhaseServoDir**	Same value provides compatible operation	5
I6mn0	**Gate2[*i*].Chan[*j*].EncCtrl**	Same value provides compatible operation	5
I6mn1	**Gate2[*i*].Chan[*j*].Equ1Ena**	Same value provides compatible operation	5
I6mn2	**Gate2[*i*].Chan[*j*].CaptCtrl**	Same value provides compatible operation	5
I6mn3	**Gate2[*i*].Chan[*j*].CaptFlag**	Same value provides compatible operation	5
I6mn4	**Gate2[*i*].Chan[*j*]. GatedIndexSel**	Same value provides compatible operation	5
I6mn5	**Gate2[*i*].Chan[*j*]. IndexGateState**	Same value provides compatible operation	5
I6mn6	**Gate2[*i*].Chan[*j*]. OutputMode**	Same value provides compatible operation	5
I6mn7	**Gate2[*i*].Chan[*j*]. OutputPol**	Same value provides compatible operation	5
I6mn8	**Gate2[*i*].Chan[*j*]. PfmDirPol**	Same value provides compatible operation	5
I6mn9	**Gate2[*i*].Chan[*j*]. OneOverTEna**	Same value provides compatible operation	5
I6m40/	**Gate2[*i*].MacroMode**	Same value provides compatible operation	5

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments	Notes
I6m90			
I6m41/ I6m91	**Gate2[*i*].MacroEnable**	Same value provides compatible operation	5

PMAC2-Style Servo IC I-Variables

Because the I-variables for setup of the Servo ICs are in hardware registers, they are identical in function between Turbo PMAC and Power PMAC. Only their names have been changed.

The Power PMAC servo IC structure index "***i***" is equivalent to two times the hundreds digit "*m*" of the Turbo PMAC I-variable number (e.g. **Gate1[4].PwmPeriod** is equivalent to I7200). The Power PMAC channel sub-structure index "***j***" is equivalent to the tens digit "*n*" of the Turbo PMAC I-variable number minus one (e.g. **Gate1[6].Chan[0].EncCtrl** is equivalent to I7310).

Often, an "alias" for the **Gate1[*i*]** data structure representing the name of the hardware utilizing the IC is used instead, e.g. **Acc24E2[*i*]**, **Acc24E2A[*i*]**, **Acc24E2S[*i*]**, or **Acc51E[*i*]**.

Turbo PMAC I-Var	Equivalent Power PMAC Saved Setup Element	Comments	Notes
I7m00	**Gate1[*i*].PwmPeriod**	Same value provides compatible operation	5
I7m01	**Gate1[*i*].PhaseClockDiv**	Same value provides compatible operation	5
I7m02	**Gate1[*i*].ServoClockDiv**	Same value provides compatible operation	5
I7m03	**Gate1[*i*]. HardwareClockCtrl**	Same value provides compatible operation	5
I7m04	**Gate1[*i*].PwmDeadTime**	Same value provides compatible operation	5
I7m05	**Gate1[*i*].DacStrobe**	Same value provides compatible operation	5
I7m06	**Gate1[*i*].AdcStrobe**	Same value provides compatible operation	5
I7m07	**Gate1[*i*].PhaseServoDir**	Same value provides compatible operation	5
I7mn0	**Gate1[*i*].Chan[*j*].EncCtrl**	Same value provides compatible operation	5
I7mn1	**Gate1[*i*].Chan[*j*].Equ1Ena**	Same value provides compatible operation	5
I7mn2	**Gate1[*i*].Chan[*j*].CaptCtrl**	Same value provides compatible operation	5
I7mn3	**Gate1[*i*].Chan[*j*].CaptFlag**	Same value provides compatible operation	5
I7mn4	**Gate1[*i*].Chan[*j*]. GatedIndexSel**	Same value provides compatible operation	5
I7mn5	**Gate1[*i*].Chan[*j*]. IndexGateState**	Same value provides compatible operation	5
I7mn6	**Gate1[*i*].Chan[*j*]. OutputMode**	Same value provides compatible operation	5
I7mn7	**Gate1[*i*].Chan[*j*]. OutputPol**	Same value provides compatible operation	5
I7mn8	**Gate1[*i*].Chan[*j*]. PfmDirPol**	Same value provides compatible operation	5
I7mn9	**Gate1[*i*].Chan[*j*]. OneOverTEna**	Same value provides compatible operation	5

Encoder Conversion Table I-Variables

The encoder conversion table (ECT) is structured very differently in Power PMAC than in Turbo PMAC. In Turbo PMAC, the setup consists of a series of I-variables (I8000 – I8191), with each entry comprising 1 to 5 I-variables.

In Power PMAC, the table takes the form of a series of indexed data structures, with each l data structure comprising an entry in the table that produces one (primary) result for servo use. Each entry in the Power PMAC ECT has an identical structure, although not all of the elements in the structure are used in every conversion method. Power PMAC supports up to 768 **EncTable[*n*]** entries (***n*** = 0 to 767).

In Power PMAC, the motor servo-loop setup variables that specify use of conversion table results – **Motor[*x*].pEnc**, **Motor[*x*].pEnc2**, and **Motor[*x*].pMasterEnc** are set to the base address of the ECT entry containing the desired result (**EncTable[*n*].a**).

Turbo PMAC Method Digit $0: Software 1/T Encoder Interpolation

EncTable[*n*].type = 3
EncTable[*n*].pEnc = **Gate1[*i*].Chan[*j*].ServoCapt.a**
EncTable[*n*].pEnc1 = **Gate1[*i*].Chan[*j*].TimeBetweenCts.a**
EncTable[*n*].index1 = 0
EncTable[*n*].index2 = 0
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].ScaleFactor = 1/512

Turbo PMAC Method Digit $1: Acc-28E 16-bit ADC Conversion

EncTable[*n*].type = 1
EncTable[*n*].pEnc = **Acc28E.AdcSdata[*j*].a**
EncTable[*n*].index1 = 16
EncTable[*n*].index2 = 16
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].ScaleFactor = 1/65536

Turbo PMAC Method Digit $2: Y-Register Parallel Read, No Maximum-Change Filtering

EncTable[*n*].type = 1
EncTable[*n*].pEnc = ***{source element}*.a**
EncTable[*n*].index1 = 32 - *Turbo Width Value*
EncTable[*n*].index2 = *Turbo Offset Value* + 8
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0

EncTable[*n*].MaxDelta= 0
EncTable[*n*].ScaleFactor = $1/2^{(32\text{-}TurboWidthValue)}$ (For result in LSBs of source)
EncTable[*n*].ScaleFactor = $1/2^{(27\text{-}TurboWidthValue)}$ (For result in 1/32 LSB of source)

Turbo PMAC Method Digit $3: Y-Register Parallel Read, with Maximum-Change Filtering

EncTable[*n*].type = 1
EncTable[*n*].pEnc = ***{source element}*.a**
EncTable[*n*].index1 = 32 - *Turbo Width Value*
EncTable[*n*].index2 = *Turbo Offset Value* + 8
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= *Turbo Max Change Value* (In source LSBs per servo cycle)
EncTable[*n*].ScaleFactor = $1/2^{(32\text{-}TurboWidthValue)}$ (For result in LSBs of source)
EncTable[*n*].ScaleFactor = $1/2^{(27\text{-}TurboWidthValue)}$ (For result in 1/32 LSB of source)

Turbo PMAC Method Digit $4: Time Base Conversion

This section provides the Power PMAC settings for "time base" conversion using an incremental encoder with software 1/T extension from a PMAC2-style "DSPGATE1" servo IC. Note that it is the same type of entry as for processing the position value of the encoder for feedback. The coordinate system points to a different register for its time-base source than a motor's servo algorithms use for feedback or master position.

EncTable[*n*].type = 3
EncTable[*n*].pEnc = **Gate1[*i*].Chan[*j*].ServoCapt.a**
EncTable[*n*].pEnc1 = **Gate1[*i*].Chan[*j*].TimeBetweenCts.a**
EncTable[*n*].index1 = 0
EncTable[*n*].index2 = 0
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].ScaleFactor = 1/(512 * RTIF[cts/msec])

Coord[*x*].pDesTimeBase = **EncTable[*n*].DeltaPos.a**

Turbo PMAC Method Digit $5: Integrated Acc-28E 16-bit ADC Conversion

EncTable[*n*].type = 1
EncTable[*n*].pEnc = **Acc28E.AdcSdata[*j*].a**
EncTable[*n*].index1 = 16
EncTable[*n*].index2 = 16
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 1
EncTable[*n*].index5 = 0
EncTable[*n*].PrevDelta = 65536 × *{16-bit bias value}*
EncTable[*n*].ScaleFactor = 1/65536

Turbo PMAC Method Digit $6: Y/X-Register Parallel Read, No Maximum-Change Filtering

EncTable[*n*].type = 1
EncTable[*n*].pEnc = ***{source element}*.a**
EncTable[*n*].index1 = 32 - *Turbo Width Value*
EncTable[*n*].index2 = *Turbo Offset Value* – 16*
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= 0
EncTable[*n*].ScaleFactor = $1/2^{(32-TurboWidthValue)}$ (For result in LSBs of source)
EncTable[*n*].ScaleFactor = $1/2^{(27-TurboWidthValue)}$ (For result in 1/32 LSB of source)

*Assuming all data used was in an X-register in Turbo PMAC.

Turbo PMAC Method Digit $7: Y/X-Register Parallel Read, with Maximum-Change Filtering

EncTable[*n*].type = 1
EncTable[*n*].pEnc = ***{source element}*.a**
EncTable[*n*].index1 = 32 - *Turbo Width Value*
EncTable[*n*].index2 = *Turbo Offset Value* – 16*
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= *Turbo Max Change Value* (In source LSBs per servo cycle)
EncTable[*n*].ScaleFactor = $1/2^{(32-TurboWidthValue)}$ (For result in LSBs of source)
EncTable[*n*].ScaleFactor = $1/2^{(27-TurboWidthValue)}$ (For result in 1/32 LSB of source)

*Assuming all data used was in an X-register in Turbo PMAC.

Turbo PMAC Method Digit $8: Parallel Extension of Incremental Encoder

This conversion method is not supported in any Turbo PMAC system that could be upgraded to Power PMAC, or in any Power PMAC system.

Turbo PMAC Method Digit $9/$A/$B: Triggered Time Base Conversion

This section provides the Power PMAC settings for "triggered time base" conversion using an incremental encoder with software 1/T extension from a PMAC2-style "DSPGATE1" servo IC.

Frozen State (Turbo PMAC Method Digit = $9):

EncTable[*n*].type = 10
EncTable[*n*].pEnc = **Gate1[*i*].Chan[*j*].ServoCapt.a**
EncTable[*n*].pEnc1 = **Gate1[*i*].Chan[*j*].TimeBetweenCts.a**
EncTable[*n*].index1 = 0
EncTable[*n*].index2 = 0
EncTable[*n*].index3 = 0

EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].ScaleFactor = 1/(512 × RTIF[cts/msec])

Coord[*x*].pDesTimeBase = **EncTable[*n*].PrevDelta.a**

Armed State (Turbo PMAC Method Digit = $B):

EncTable[*n*].pEnc1 = **Gate1[*i*].Chan[*j*].Status.a**
EncTable[*n*].index1 = 2
(Trigger condition defined by **Gate1[*i*].Chan[*j*].CaptCtrl** and **Gate1[*i*].Chan[*j*].FlagCtrl**)

Armed State (Turbo PMAC Method Digit = $A):

On occurrence of the specified trigger, Power PMAC automatically changes the entry to the "running" state equivalent to "untriggered" time base, by setting:

EncTable[*n*].type = 3
EncTable[*n*].pEnc1 = **Gate1[*i*].Chan[*j*].TimeBetweenCts.a**
EncTable[*n*].index1 = 0

Turbo PMAC Method Digit $C: No Extension of Incremental Encoder

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*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= 0
EncTable[*n*].ScaleFactor = 1/256

Turbo PMAC Method Digit $D: Low-Pass Filter

In the Turbo PMAC encoder conversion table, the low-pass filter (simple exponential filter without an integrator, or a tracking filter with an integrator) almost always processed the result of another table entry that processed the actual source data. This section shows how to do this in the Power PMAC table. The Power PMAC table has more flexibility to do the filtering in the same entry that processes the source data – refer to the Power PMAC User's Manual chapter on the ECT for more details.

For an exponential filter (Turbo PMAC 1st setup word bit 19 = 0), the Power PMAC settings to process the result of a previous entry ***m*** would be:

EncTable[*n*].type = 1
EncTable[*n*].pEnc = **EncTable[*m*].PrevEnc.a**
EncTable[*n*].index1 = 0
EncTable[*n*].index2 = 256 – (*Turbo Kp Gain* / 32768)
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0

EncTable[*n*].MaxDelta= *Turbo Max Change value* × 2^N *
EncTable[*n*].ScaleFactor = 1/256

* N is the bit number of the LSB of the source data after the shifting

For tracking filter (Turbo PMAC 1st setup word bit 19 = 1), the Power PMAC settings to process the result of a previous entry ***m*** would be:

EncTable[*n*].type = 1
EncTable[*n*].pEnc = **EncTable[*m*].PrevEnc.a**
EncTable[*n*].index1 = *Turbo Ki Gain / 32768***
EncTable[*n*].index2 = 256 – (*Turbo Kp Gain* / 32768)
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0 **
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= *Turbo Max Change value* × 2^N *
EncTable[*n*].ScaleFactor = 1/256

** If the formula for **index1** gives a value less than 32, increase **index4** by 1 and multiply **index1** by 2 until **index1** is 32 or greater.

Note that **index1** and **index2**must be integer values in Power PMAC, so rounding may be required.

Turbo PMAC Method Digit $E: Addition or Subtraction of Entries

In the Turbo PMAC encoder conversion table, addition or subtraction of data sources *must* use the results of previous entries as the sources to be added or subtracted. In the Power PMAC ECT, the results of previous entries are most commonly the sources for addition or subtraction, but this is not required. This section shows how to add or subtract the results of other entries in the Power PMAC table. For adding or subtracting other sources, refer to the Power PMAC User's Manual chapter on the ECT for more details.

For adding the results of two entries (Turbo PMAC 1st setup word bit 17 = 0), the Power PMAC settings to add the results of previous entries ***m*** and ***p*** would be:

EncTable[*n*].type = 8
EncTable[*n*].pEnc = **EncTable[*m*].PrevEnc.a**
EncTable[*n*].pEnc1 = **EncTable[*p*].PrevEnc.a**
EncTable[*n*].index1 = 0
EncTable[*n*].index2 = 0
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= 0
EncTable[*n*].ScaleFactor = 1.0

For subtracting the results of two entries (Turbo PMAC 1st setup word bit 17 = 0), the Power PMAC settings to subtract the result of previous entry ***p*** from previous entry ***m*** would be:

EncTable[*n*].type = 9
EncTable[*n*].pEnc = **EncTable[*m*].PrevEnc.a**

EncTable[*n*].pEnc1 = **EncTable[*p*].PrevEnc.a**
EncTable[*n*].index1 = 0
EncTable[*n*].index2 = 0
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= 0
EncTable[*n*].ScaleFactor = 1.0

Turbo PMAC Method Digit $F/$0: High-Resolution Sinusoidal Encoder Interpolation

EncTable[*n*].type = 4
EncTable[*n*].pEnc = **Gate1[*i*].Chan[*j*].Status.a**
EncTable[*n*].pEnc1 = **Gate1[*i*].Chan[*j*].Adc[0].a**
EncTable[*n*].index1 = 0
EncTable[*n*].index2 = 0
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].SinBias = 16 × *Turbo Sine Bias Term*
EncTable[*n*].CosBias = 16 × *Turbo Cosine Bias Term*
EncTable[*n*].ScaleFactor = 1/(512 × RTIF[cts/msec])

With the above settings, no low-pass filtering is performed in this ECT entry. This type of filtering, whether in this entry, or a separate entry, is often recommended for sine encoder conversion.

Turbo PMAC Method Digit $F/$1: High-Resolution Sinusoidal Encoder Interpolation Diagnostics

In the Power PMAC, the standard high-resolution sinusoidal encoder interpolation entry (**type** = 4, shown immediately above), automatically computes the (square of the) vector magnitude of encoder signal every servo cycle; this can be read in status element **EncTable[*n*].SumOfSqr**.

The Power PMAC does not have an automatic method in the ECT for estimating the offsets of the sine and cosine inputs of a sinusoidal encoder.

Turbo PMAC Method Digit $F/$2: Byte-Wide Parallel Read, No Maximum-Change Filtering

EncTable[*n*].type = 5
EncTable[*n*].pEnc = **GateIo[*i*].DataReg[*j*].a**
EncTable[*n*].pEnc1 = **GateIo[*i*].DataReg[*j*+*1*].a**
EncTable[*n*].index1 = 32 - *Turbo Width Value*
EncTable[*n*].index2 = *Turbo Offset Value* + 8
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= *Turbo Max Change Value* (In source LSBs per servo cycle)
EncTable[*n*].ScaleFactor = $1/2^{(32-TurboWidthValue)}$

Turbo PMAC Method Digit $F/$3: Byte-Wide Parallel Read, with Maximum-Change Filtering

EncTable[*n*].type = 5
EncTable[*n*].pEnc = **GateIo[*i*].DataReg[*j*].a**
EncTable[*n*].pEnc1 = **GateIo[*i*].DataReg[*j*+*1*].a**
EncTable[*n*].index1 = 32 - *Turbo Width Value*
EncTable[*n*].index2 = *Turbo Offset Value* + 8
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].MaxDelta= 0
EncTable[*n*].ScaleFactor = $1 / 2^{(32-TurboWidthValue)}$

Turbo PMAC Method Digit $F/$4: Resolver Arctangent Conversion

EncTable[*n*].type = 6
EncTable[*n*].pEnc = **Gate1[*i*].a** + 256
EncTable[*n*].pEnc1 = **Gate1[*i*].Chan[*j*].Adc[0].a**
EncTable[*n*].index1 = 0
EncTable[*n*].index2 = 0
EncTable[*n*].index3 = 0
EncTable[*n*].index4 = 0
EncTable[*n*].index5 = 0
EncTable[*n*].SinBias = 16 × *Turbo Sine Bias Term*
EncTable[*n*].CosBias = 16 × *Turbo Cosine Bias Term*
EncTable[*n*].ScaleFactor = 1 / 4

With the above settings, no low-pass filtering is performed in this ECT entry. This type of filtering, whether in this entry, or a separate entry, is strongly recommended for resolver conversion.

POWER PMAC TURBO SUGGESTED M-VARIABLE EQUIVALENTS

This section is intended to facilitate the conversion of a working Turbo PMAC application to the Power PMAC with minimum changes. It documents the Power PMAC equivalents to Turbo PMAC suggested M-variable definitions and describes whether changes to these settings are required or not. (Unlike I-variables, these Turbo PMAC M-variable definitions are only suggested, but they are very widely used.)

Note that a direct conversion by straight application of these tables usually will not utilize Power PMAC's capabilities to its fullest. However, it may provide a good starting point for further enhancements.

Notes on Equivalents

1. Position units for this variable in Turbo PMAC are in units such as 1/[Ixx08*32] or 1/[Ixx09*32] of a count to provide sufficient resolution in a fixed-point format. In Power PMAC's floating-point equivalent, fractional resolution is provided with "whole" units.

2. Position units for this variable in Power PMAC are "motor units", which can be, but do not have to be "counts" of the encoder. In applications where the simplest possible porting from Turbo PMAC is desired, motor units should be set a "counts" of the encoder or equivalent.

3. In Turbo PMAC, this position value is relative to the motor home position if a position referencing operation has been done since power-on/reset. In Power PMAC, this position value is always relative to the power-on/reset position. The difference between the motor home position and the power-on/reset position is held in **Motor[*x*].HomePos**, which should be subtracted from the Power PMAC value to get the equivalent of the Turbo PMAC value.

Miscellaneous Global Variables

This section lists variables that are used to monitor processor time spent in different tasks.

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
M70	Time between phase interrupts (CPU cycles/2)	**Sys.PhaseDeltaTime**	**PhaseDeltaTime** in microseconds	
M71	Time for phase tasks (CPU cycles/2)	**Sys.PhaseTime**, **Sys.FltrPhaseTime**	**PhaseTime**, **FltrPhaseTime** in microseconds	
M72	Time for servo tasks (CPU cycles/2)	**Sys.ServoTime**, **Sys.FltrServoTime**	**ServoTime**, **FltrServoTime** in microseconds	
M73	Time for RTI tasks (CPU cycles/2)	**Sys.RtIntTime**, **Sys.FltrRtIntTime**	**RtIntTime**, **FltrRtIntTime** in microseconds	

Servo Cycle Counter

The servo cycle counter increments once per servo interrupt, and can be used to determine the time elapsed in a process. Note the longer range of the Power PMAC counter. In both cases, a robust algorithm will account for the possibility of rollover.

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
M100	24-bit servo-cycle counter	**Sys.ServoCount**	**ServoCount** is 32-bit integer	

PMAC2 Servo IC Registers

This section lists key registers and control/status bits in the "DSPGATE1" PMAC2-style Servo IC. In the Power PMAC Script environment, the full-register elements are 24-bit values, even though they are accessed over a 32-bit bus.

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
Mxx01	24-bit counter position	**Gate1[*i*].Chan[*j*].PhaseCapt**		
Mxx02	Phase A command value, DAC or PWM	**Gate1[*i*].Chan[*j*].Dac[0]**, **Gate1[*i*].Chan[*j*].Pwm[0]**	**Dac[0]** and **Pwm[0]** are same register; have 24-bit, not 16-bit, range	
Mxx03	Captured position	**Gate1[*i*].Chan[*j*].HomeCapt**		
Mxx04	Phase B command value, DAC or PWM	**Gate1[*i*].Chan[*j*].Dac[1]**, **Gate1[*i*].Chan[*j*].Pwm[1]**	**Dac[1]** and **Pwm[1]** are same register; have 24-bit, not 16-bit, range	
Mxx05	Phase A ADC input value	**Gate1[*i*].Chan[*j*].Adc[0]**	**Adc[0]** has 24-bit, not 16-bit, range	
Mxx06	Phase B ADC input value	**Gate1[*i*].Chan[*j*].Adc[1]**	**Adc[1]** has 24-bit, not 16-bit, range	
Mxx07	Phase C command value, PFM or PWM	**Gate1[*i*].Chan[*j*].Pfm**, **Gate1[*i*].Chan[*j*].Pwm[2]**	**Pfm** and **Pwm[1]** are same register; have 24-bit, not 16-bit, range	
Mxx08	Compare A position	**Gate1[*i*].Chan[*j*].CompA**		
Mxx09	Compare B position	**Gate1[*i*].Chan[*j*].CompB**		
Mxx10	Compare auto-increment value	**Gate1[*i*].Chan[*j*].CompAdd**		
Mxx11	Compare initial state write enable	**Gate1[*i*].Chan[*j*].EquWrite** bit 0		
Mxx12	Compare initial state	**Gate1[*i*].Chan[*j*].EquWrite** bit 1		
Mxx14	AENA output status	**Gate1[*i*].Chan[*j*].AmpEna**		
Mxx15	User flag input status	**Gate1[*i*].Chan[*j*].UserFlag**		
Mxx16	Compare output value	**Gate1[*i*].Chan[*j*].Equ**		
Mxx17	Capture flag	**Gate1[*i*].Chan[*j*].PosCapt**		
Mxx18	Count error flag	**Gate1[*i*].Chan[*j*]. CountError**		
Mxx19	Encoder C channel	**Gate1[*i*].Chan[*j*].ABC**		

	input status	bit 2		
Mxx20	Home flag input status	**Gate1[*i*].Chan[*j*].HomeFlag**		
Mxx21	Plus limit input status	**Gate1[*i*].Chan[*j*].PlusLimit**		
Mxx22	Minus limit input status	**Gate1[*i*].Chan[*j*]. MinusLimit**		
Mxx23	Fault flag input status	**Gate1[*i*].Chan[*j*].Fault**		
Mxx24	W flag input status	**Gate1[*i*].Chan[*j*].UVW** bit 0		
Mxx25	V flag input status	**Gate1[*i*].Chan[*j*].UVW** bit 1		
Mxx26	U flag input status	**Gate1[*i*].Chan[*j*].UVW** bit 2		
Mxx27	T flag input status	**Gate1[*i*].Chan[*j*].T**		
Mxx28	TUVW flag inputs as 4-bit value	8*__Gate1[*i*].Chan[*j*].T__ + **Gate1[*i*].Chan[*j*].UVW**		

Motor Status Bits

This section lists the most commonly used Turbo PMAC motor status bits and their Power PMAC equivalents. The Power PMAC motor structure index "***x***" is equivalent to the "hundreds" value of the Turbo PMAC suggested M-variable number.

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
Mxx30	Stopped on position limit	**Motor[*x*].LimitStop**, **Motor[*x*].SoftLimit**	Power PMAC has separate bits for HW and SW limits	
Mxx31	Positive end limit set	**Motor[*x*].PlusLimit**, **Motor[*x*].SoftPlusLimit**	Power PMAC has separate bits for HW and SW limits	
Mxx32	Negative end limit set	**Motor[*x*].MinusLimit**, **Motor[*x*].SoftMinusLimit**	Power PMAC has separate bits for HW and SW limits	
Mxx33	Desired velocity zero	**Motor[*x*].DesVelZero**		
Mxx35	Dwell in progress	**Motor[*x*].Desired.Dwell**		
Mxx37	Running program (move timer active)	**Motor[*x*].Desired. TimerEnabled**		
Mxx38	Open-loop mode	**Motor[*x*].ClosedLoop**	Opposite logical polarity in Power PMAC	
Mxx39	Amplifier enabled	**Motor[*x*].AmpEna**		
Mxx40	Background in-position	**Motor[*x*].InPos**	All in-position checking in Power PMAC is foreground	
Mxx41	Warning following error	**Motor[*x*].FeWarn**		
Mxx42	Fatal following error	**Motor[*x*].FeFatal**		
Mxx43	Amplifier fault	**Motor[*x*].AmpFault**		
Mxx44	Foreground in-position	**Motor[*x*].InPos**	Foreground in-position checking in Power PMAC has scan count	
Mxx45	Home complete	**Motor[*x*].HomeComplete**		
Mxx46	Integrated following error fault	--	No integrated following error fault in Power PMAC	
Mxx47	I2T fault	**Motor[*x*].I2tFault**		

Mxx48	Phase error fault	**Motor[*x*].PhaseFound**	Opposite logical polarity in Power PMAC	
Mxx49	Phase search in progress	**Motor[*x*].PhaseFindingStep**	Enumeration, not Boolean, in Power PMAC, but non-zero during phasing search	
--	Home search in progress	**Motor[*x*].HomeInProgress**		
--	Trigger move in process	**Motor[*x*].TriggerMove**		
--	Assigned to C.S.	**Motor[*x*].Coord**	**Coord** specifies C.S. # motor is assigned to. Usually motors not assigned to active C.S. are assigned to C.S. 0.	

Motor Move Registers

This section lists commonly used Turbo PMAC motor move registers – positions, velocities, and servo commands – and their Power PMAC equivalents. The Power PMAC motor structure index "***x***" is equivalent to the "hundreds" value of the Turbo PMAC suggested M-variable number.

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
Mxx61	Commanded position	**Motor[*x*].DesPos**	**DesPos** is net desired position, including master and compensation	1,2,3
Mxx62	Actual position	**Motor[*x*].ActPos**	**ActPos** is net actual position, including backlash and actual compensation	1,2,3
Mxx63	Target (end) position	**Motor[*x*].MoveDesPos**		1,2,3
Mxx64	Position bias	**Motor[*x*].CoordSf[32]**, **Tdata[*i*].Bias[*m*]**	Different strategies for offsetting axis origin from motor origin usually used	
Mxx66	Actual velocity	**Motor[*x*].ActVel**, **Motor[*x*].ActVel2**	Mxx66 is inner-loop velocity; **ActVel** is outer-loop velocity; **ActVel2** is inner-loop velocity	1,2
Mxx67	Master position	**Motor[*x*].MasterPos**, **Motor[*x*].ActiveMasterPos**	**ActiveMasterPos** can be rate limited and is used directly by servo	1,2
Mxx68	Filter output	**Motor[*x*].IqCmd**	Floating-point in Power PMAC, equivalent units	
Mxx69	Compensation correction	**Motor[*x*].CompDesPos**, **Motor[*x*].CompPos**	Mxx69 is added to desired position; **CompDesPos** is added to desired position; **CompPos** is added to actual position, so has opposite direction sense	1,2
Mxx70	Present phase position (including fraction)	**Motor[*x*].PhasePos**	**PhasePos** in commutation units (1/2048 cycle)	1
Mxx71	Present phase position	**Motor[*x*].PhasePos**	**PhasePos** in commutation units (1/2048 cycle)	

Mxx72	Variable jog position/distance	**Motor[*x*].ProgJogPos**		
Mxx73	Encoder home capture position	**Motor[*x*].HomePos**	**HomePos** always in motor units	
Mxx74	Averaged actual velocity	**Motor[*x*].FltrVel**, **Motor[*x*].FltrVel2**	Mxx74 is inner-loop velocity; **FltrVel** is outer-loop velocity; **FltrVel2** is inner-loop velocity	1,2
Mxx75	Actual quadrature current	**Motor[*x*].IqMeas**	Floating-point in Power PMAC, equivalent units	
Mxx76	Actual measured current	**Motor[*x*].IdMeas**	Floating-point in Power PMAC, equivalent units	
Mxx77	Quadrature current loop integrator output	**Motor[*x*].IqVolts**	Floating-point in Power PMAC, equivalent units	
Mxx78	Direct current-loop integrator output	**Motor[*x*].IdVolts**	Floating-point in Power PMAC, equivalent units	
Mxx79	PID internal filter result	**Motor[*x*].ServoOut**	Floating-point in Power PMAC, equivalent units	

Motor Axis Definition Registers

This section lists the Turbo PMAC suggested M-variables for the axis-definition coefficients for motors and their Power PMAC equivalents. The Power PMAC motor structure index "***x***" is equivalent to the "hundreds" value of the Turbo PMAC suggested M-variable number (e.g. **Motor[12].CoordSf[6]** is equivalent to M1291).

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
Mxx91	X/U/A/B/C-Axis scale factor	**Motor[*x*].CoordSf[6]** (X), **Motor[*x*].CoordSf[3]** (U), **Motor[*x*].CoordSf[0]** (A), **Motor[*x*].CoordSf[1]** (B), **Motor[*x*].CoordSf[2]** (C)	Power PMAC has separate element for each axis	
Mxx92	Y/V-Axis scale factor	**Motor[*x*].CoordSf[7]** (Y), **Motor[*x*].CoordSf[4]** (V)	Power PMAC has separate element for each axis	
Mxx93	Z/W-Axis scale factor	**Motor[*x*].CoordSf[8]** (Z), **Motor[*x*].CoordSf[5]** (W)	Power PMAC has separate element for each axis	
Mxx94	Axis offset	**Motor[*x*].CoordSf[32]**		

Demultiplexed ADC Registers

This section lists the Turbo PMAC suggested M-variables for the results of the automatic demultiplexing of A/D converters on ACC-36 or ACC-59 boards

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
M5061 – M5076	Demuxed low ADC register from I5061 – I5076	**AdcDemux.ResultLow[*i*]**	Index ***i*** ranges from 0 to 15 to match M5061 to M5076	

M5081 – M5096	Demuxed high ADC register from I5061 – I5076	**AdcDemux.ResultHigh[*i*]**	Index ***i*** ranges from 0 to 15 to match M5081 to M5096	

Coordinate System Timers

Power PMAC does not have direct equivalents to Turbo PMAC's dedicated countdown timers. But it is easy to create equivalent logic in Power PMAC using the **Sys.ServoCount** status register.

To create the logic equivalent to the Turbo PMAC logic:

```
Msx11 == MyDelay
while (Msx11 > 0) endwhile
```

you can use the Power PMAC logic:

```
MyEndtime = Sys.ServoCount + MyDelay;
while (Sys.ServoCount < MyEndTime) { }
```

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
Msx11	&x Isx11 timer (for synchronous assignments)	(*no direct equivalent*)	Use servo counter and general-purpose variable as explained above	
Msx12	&x Isx12 timer (for synchronous assignments)	(*no direct equivalent*)	Use servo counter and general-purpose variable as explained above	

Coordinate System End-of-Calculated-Move Registers

This section lists the Turbo PMAC suggested M-variables for the axis destination values for the most recently calculated move (which may not be the currently executing move) and their Power PMAC equivalents. The Power PMAC motor structure index "***x***" is equivalent to the "hundreds" value of the Turbo PMAC suggested M-variable number minus 50 (e.g. **Coord[1].CdPos[6]** is equivalent to M5147).

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
Msx41	A-axis target position	**Coord[*x*].CdPos[0]**		
Msx42	B-axis target position	**Coord[*x*].CdPos[1]**		
Msx43	C-axis target position	**Coord[*x*].CdPos[2]**		
Msx44	U-axis target position	**Coord[*x*].CdPos[3]**		
Msx45	V-axis target position	**Coord[*x*].CdPos[4]**		
Msx46	W-axis target position	**Coord[*x*].CdPos[5]**		
Msx47	X-axis target position	**Coord[*x*].CdPos[6]**		
Msx48	Y-axis target position	**Coord[*x*].CdPos[7]**		
Msx49	Z-axis target position	**Coord[*x*].CdPos[8]**		

Coordinate System Status Bits

This section lists the most commonly used Turbo PMAC coordinate-system status bits and their Power PMAC equivalents. The Power PMAC motor structure index "***x***" is equivalent to the "hundreds" value of the Turbo PMAC suggested M-variable number minus 50 (e.g. **Coord[2].RunTimeError** is equivalent to M5282).

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
Msx80	Program-running bit	**Coord[*x*].ProgRunning**	**ProgRunning** stays true until all moves calculated by the program have finished executing	
Msx81	Circle radius error	**Coord[*x*].RadiusError**		
Msx82	Run-time error	**Coord[*x*].RunTimeError**		
Msx84	Continuous motion request	**Coord[*x*].ContMotion**		
Msx87	In-position bit (AND of motors)	**Coord[*x*].InPos**		
Msx88	Warning following error (OR of motors)	**Coord[*x*].FeWarn**		
Msx89	Fatal following error (OR of motors)	**Coord[*x*].FeFatal**		
Msx90	Amp-fault error (OR of motors	**Coord[*x*].AmpFault**		

Coordinate System Time Base Variables

This section lists the coordinate-system time base variables. The Power PMAC motor structure index "***x***" is equivalent to the "hundreds" value of the Turbo PMAC suggested M-variable number minus 50 (e.g. **Coord[4].TimeBase** is equivalent to M5498).

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
Msx97	Host-commanded time base	**Coord[*x*].DesTimeBase**	**DesTimeBase** has units of milliseconds, not 2^{-23} msec	
Msx98	Present time base	**Coord[*x*].TimeBase**	**TimeBase** has units of milliseconds, not 2^{-23} msec	

UMAC I/O Accessory M-Variables

This section lists the I/O point M-variables for the first I/O card on the UMAC backplane. It is common in Power PMAC applications to assign an M-variable through a "ptr" variable declaration to an I/O point to give it a name that is meaningful to the application.

Turbo PMAC M-Var	Description	Equivalent Power PMAC Element	Comments on Power PMAC Implementation	Notes
M7000 .. M7007	MI/O00 .. MI/O07	**GateIo[0].DataReg[0].0 .. GateIo[0].DataReg[0].7**		
M7008 .. M7015	MI/O08 .. MI/O15	**GateIo[0].DataReg[1].0 .. GateIo[0].DataReg[1].7**		
M7016 .. M7023	MI/O16 .. MI/O23	**GateIo[0].DataReg[2].0 .. GateIo[0].DataReg[2].7**		
M7024 .. M7031	MI/O24 .. MI/O31	**GateIo[0].DataReg[3].0 .. GateIo[0].DataReg[3].7**		
M7000 .. M7007	MI/O32 .. MI/O39	**GateIo[0].DataReg[4].0 .. GateIo[0].DataReg[4].7**		
M7000 .. M7007	MI/O40 .. MI/O47	**GateIo[0].DataReg[5].0 .. GateIo[0].DataReg[5].7**		

POWER PMAC UBUS32 SPECIFICATION

Backplane Connector Pinout

Pin #	Row A	Row B	Row C
1	+5V	+5V	+5V
2	GND	GND	GND
3	BD09 (*BD01*)	BD00 (*rsvrd.*)	BD08 (*BD00*)
4	BD11 (*BD03*)	BD01 (*rsvrd.*)	BD10 (*BD02*)
5	BD13 (*BD05*)	BD02 (*rsvrd.*)	BD12 (*BD04*)
6	BD15 (*BD07*)	BD03 (*rsvrd.*)	BD14 (*BD06*)
7	BD17 (*BD09*)	BD04 (*rsvrd.*)	BD16 (*BD08*)
8	BD19 (*BD11*)	BD05 (*rsvrd.*)	BD18 (*BD10*)
9	BD21 (*BD13*)	BD06 (*rsvrd.*)	BD20 (*BD12*)
10	BD23 (*BD15*)	BD07 (*rsvrd.*)	BD22 (*BD14*)
11	BD25 (*BD17*)	rsrvd	BD24 (*BD16*)
12	BD27 (*BD19*)	rsrvd	BD26 (*BD18*)
13	BD29 (*BD21*)	rsrvd	BD28 (*BD20*)
14	BD31 (*BD23*)	BCSDIR(*rsvrd.*)	BD30 (*BD22*)
15	rsrvd	BCS0-(*rsvrd.*)*	rsrvd
16	BA01	BCS1-(*rsvrd.*)*	BA00
17	BA04 (*BA03*)	BCS5-(*rsvrd.*)*	BA02
18	BA03 (*BX/Y*)	BA15 (*BA14*)	BA05 (*BA04*)
19	BCS3-	BA07 (*BA06*)	BCS2-
20	BA06 (*BA05*)	BA08 (*BA07*)	BCS4-
21	BCS12-	BA09 (*BA08*)	BCS10-
22	BCS16-	BA10 (*BA09*)	BCS14-
23	BA14 (*BA13*)	BA11 (*BA10*)	BA13 (*BA12*)
24	BRD-	BA12 (*BA11*)	BWR-
25	rsrvd	DPRCS1-	rsrvd
26	WAIT-	VMECS1-	BRESET
27	PHASE+	UMAC_INT-	SERVO+
28	PHASE-	INT1- (*EQU1-*)	SERVO-
29	AGND	INT2- (*EQU2-*)	AGND
30	A-15V	PWM_ENA	A+15V
31	GND	GND	GND
32	+5V	+5V	+5V

Notes:

1. Names in italics refer to use of this pin with Turbo PMAC CPU on UBUS backplane.
2. “rsvrd” means “reserved for future use”
3. “B” as the first letter means “buffered”; these signals should be passed through buffer ICs within 25mm of the backplane connector on any card using this signal.
4. (*) On prototype boards from 2007, BCS0- was on pin B11, BCS1- was on B12, and BCS2- was on B13.

FIRMWARE UPDATE HISTORY

This section lists the additions and corrections provided in each revision of Power PMAC firmware.

V2.2 Firmware Updates (July 2016)

Updates from 2.1 release:

1. Added support for ARM CPU used in Micro Power PMAC and new UMAC CPU board.

2. Added support for Omron IPC running Windows and Linux under Hypervisor.

3. Added support for Acontis EtherCAT stack network interface software with new saved setup element **Sys.EcatType** set to 1. (Users of Etherlabs EtherCAT stack can continue to use it on their versions of Power PMAC with **Sys.EcatType** = 0.)

4. Increased default value of **Motor[*x*].EcatAmpFaultLimit** from 100 to 1000 to support EtherCAT drives that take longer to fully enable after command.

5. Added 256 Script user countdown timer elements **Sys.CdTimer[*i*]** (***i*** = 0 to 255), scaled in seconds. Primarily useful for delays in Script PLC programs.

6. Added saved setup element **Coord[*x*].FProtect**. If set to 1, feedrate parameter **Coord[*x*].Tm** is automatically set to 0.0 when a motion program is started from the top, and if a move is commanded with a zero or negative feedrate value (**Coord[*x*].Tm** >= 0.0), program is stopped with a run time error (**Coord[*x*].ErrorStatus** = 21).

7. Added non-saved setup element **Coord[*x*].OnceNoBlend** to make it easier to implement "one-shot" exact-stop functionality, as with G09 CNC code. If greater than 0 when a move is calculated, blending is disabled at the end of that move and the element value is decremented by 1.

8. Added status elements for buffered PLC-style I/O forcing functions **Sys.BgForceInOr**, **Sys.BgForceOutOr**, **Sys.FgForceInOr**, **Sys.FgForceOutOr**, and **Sys.ForceOr**. These 32-bit elements are the bit-by-bit logical OR words of the matching individual forcing words. They are intended to make it easy for the user to determine whether any of the buffered inputs or outputs is being forced.

9. Added on-line command **bufioforceclear** to permit the user to clear all of the buffered PLC-style I/O forcing words (background and foreground, input and output) with a single command.

V2.1 Firmware Updates (March 2016)

Updates from 2.0.2 maintenance release:

10. Added optional second vector-feedrate axis set with **`frax2`** program command and **Coord[*x*].FR2Axes** status element. If second set is declared, then on feedrate-specified blended moves, Power PMAC compares time for primary feedrate axis set to time for secondary set, and uses longer time.

11. Added auxiliary velocity and acceleration (torque) feedforward output registers for **Sys.PosCtrl** position-output servo algorithm with new **Motor[*x*].Servo.pVelOut** and **Motor[*x*].Servo.pAccOut** addressing elements (present but not documented in V2.0 release). Intended mainly to permit direct commanding of "velocity offset" and "torque offset" registers in EtherCAT cyclic position-mode drives.

12. Added capability to include **Motor[*x*].CompDac** value (usually from torque compensation table) in auxiliary acceleration (torque) feedforward output register specified by **Motor[*x*].Servo.pAccOut** in **Sys.PosCtrl** position-output servo algorithm.

13. Added capability to filter feedforward trajectory values in **Sys.PosCtrl** algorithm by adding "F" polynomial filter with its elements **Motor[*x*].Servo.Kf1** and **Kf2** to algorithm.

14. Disabled post-servo adjustments to servo-output command from **Sys.PosCtrl** algorithm (as not appropriate for this mode).

15. Added auxiliary acceleration (torque) feedforward output register for **Sys.PidCtrl**, **Sys.ServoCtrl**, and **Sys.AdaptiveCtrl** servo algorithms with new **Motor[*x*].Servo.pAccOut** addressing element. (Present but not documented in V2.0 release.) Feedforward terms that would normally add into torque node of these algorithms for inclusion in the standard command output instead are used for separate feedforward output. Intended mainly to permit direct commanding of torque/acceleration feedforward values on MACRO and EtherCAT networked drives in velocity mode.

16. Added ability to directly cascade multiple servo loops with new saved setup element **Motor[*x*].pCascadeCmd**, which specifies the address of a double-precision floating-point register that is a position or torque input to another motor's servo loop, and **Motor[*x*].CascadeMode**, which determines whether this value is integrated or not. This eliminates the need to process the command through a compensation table or position-following algorithm.

17. Added functionality for scanned buffered PLC-style I/O with new saved setup elements **Sys.BufIoEnable**, **Sys.MaxRtBufIn**, **Sys.MaxRtBufOut**, **BufIo[*i*].pIn**, **BufIo[*i*].InScans**, and **BufIo[*i*].pOut**. User can override input and output values with **BufIo[*i*].ForceInOn**, **BufIo[*i*].ForceInOff**, **BufIo[*i*].ForceOutOn**, and **BufIo[*i*].ForceOutOff**. User can read scanned input state in **BufIo[*i*].In**, and input edges in **BufIo[*i*].FallIn**, **BufIo[*i*].FallInLatch**, **BufIo[*i*].RiseIn**, and **BufIo[*i*].RiseInLatch**. User can write to output holding elements **BufIo[*i*].Out** for later automatic transfer to actual output registers.

18. Improved flexibility of specifying the two hardware overtravel limit input bits by extending range of saved setup element **Motor[*x*].LimitBits**. In pre-existing range of 0 – 31, **LimitBits** specifies bit # of positive limit, and negative limit is next higher bit. In new range of 32 – 63, (**LimitBits** – 32) specifies a single limit input bit for both ends. In new range of 64 – 95, (**LimitBits** – 64) specifies bit # of negative limit, and positive limit is next higher bit. In new range of 96 – 127, (**LimitBits** – 96) specifies bit # of positive limit, and negative limit is specified by **Motor[*x*].pAuxFault** and **Motor[*x*].AuxFaultBit**. If bit 7 (value 128) is set, a 0 in the input bit signifies the limit has been hit instead of the default 1.

19. Added ability to set trigger condition of selected bit for triggered moves as 1-to-0 transition of specified bit with new saved setup element **Motor[*x*].CaptFlagInvert**.

20. Added ability to set trigger condition of selected bit for triggered moves as first transition found after start of move with new saved setup element **Motor[*x*].CaptToggle**.

21. Added ability to perform specific arming of external trigger circuitry for triggered moves with new saved setup elements **Motor[*x*].pCaptEna**, **CaptEnaBit**, and **CaptEnaInvert**. Primarily to set up hardware capture in EtherCAT drives.

22. Added ability when executing circle-mode move with 2D cutter-radius compensation to have tool-edge speed (rather than tool-center speed) to be at specified feedrate by setting new bit 3 (value 8) of **Coord[*x*].CCCtrl** to 1.

23. Added ability to automatically append a delay time at the end of a continuous move sequence with new saved setup element **Coord[*x*].EndDelay** to permit trajectory filters in inverse-kinematic subroutine or motor pre-filters to converge on final programmed position.

24. Added "iterative learning control" algorithms for setting up torque compensation tables with new saved setup elements **CompTable[*m*].DacEnable**, **DacTarget**, **DacGain**, **MaxDac**, and **MinPosError**.

25. Added ability to store gathered data to potentially larger user shared memory buffer instead of fixed (1 MB) dedicated gather buffer with new elements **Gather.UserBufStart**, **UserBufSize**, **PhaseUserBufStart**, and **PhaseUserBufSize**.

26. Added ability to upload full data gathering buffer that has wrapped around with "`-r`" option on "`gather`" application. Added **`clear gather`** and **`clear phase gather`** on-line and buffered program commands to set up for wraparound operation so this option will work even if the gathering has not wrapped around.

27. Added **`rotfreeall`** on-line query command that will report size of both available blocks of memory in the buffer (at the start and at the end) if these are separate. The already-existing **`rotfree`** command only reports the larger of the two of these blocks.

28. Added informational saved setup elements **Motor[*x*].PosUnit**, **Motor[*x*].Pos2Unit**, and **Motor[*x*].CurrentScale**. These have no active function in the Power PMAC, but can be used by the IDE motor setup utilities to store this information for future reference.

29. Added Modbus "keep alive" saved setup elements **Modbus[*x*].Config.KeepAliveCnt**, **KeepAliveEnable**, **KeepAliveIdle**, **KeepAliveInterval**, and **KeepAliveTimeOut**.

30. Removed Script write-protection from **Acc84*x*[*i*].Chan[*j*].SerialEncDataA**, **SerialEncDataB**, **SerialEncDataC**, and **SerialEncDataD** elements to better support non-encoder uses of this accessory family. (With encoder-interface logic installed, a write operation will have no effect on the register, but no error will be reported.)

31. Removed Script write-protection from **Gate1[*i*].Chan[*j*].CountError** and **Gate2[*i*].Chan[*j*].CountError** status bits to allow user to clear the error bit directly. (This does not remove the error in the counter value.)

32. In gantry leader/follower control, a follower motor in closed-loop control is automatically disabled ("killed") if leader motor is disabled for any reason (even if not a fault). If disabled by command, no special status bits are set for either motor for reason of disabling.

33. In gantry leader/follower control, if a follower motor hits a hardware or software overtravel limit, leader motor is stopped just as if it had hit this limit itself. Appropriate status bits for both follower and leader motors are set.

34. In "network slave" torque mode (**Motor[*x*].MotorMode** = 3), changed action so the slave motor does not react to overtravel limits itself. It still passes the limit information back to the master motor, which can take the appropriate action.

35. Changed **ECAT[*i*].Enable** from saved to non-saved setup element that is always 0 at power-on/reset. User application must explicitly set to 1 to activate network. Prevents network initialization problems if Power PMAC is ready before slave devices.

36. Tightened control of servo period adjustment in EtherCAT distributed clock mode to follow external frequency more closely.

37. Improved timing margins on **MuxIo** serial data transfers to better support 3rd-party modules.

38. Increased efficiency and robustness of encoder conversion table **type** = 4 software arctangent interpolation for sinusoidal encoders by locking in relationship between A/B quadrature state and counter LSBs at initialization.

39. If **Coord[*x*].AltFeedRate** is set to 0.0, on a feedrate-specified move where the vector distance of feedrate axes is exactly zero, the non-feedrate axes will move at the speed specified by the numerical value from the F-command.

40. Corrected reporting of **Coord[*x*].Nsync** value and move axis target positions when in reverse (negative time base) execution through trace buffers.

41. Fixed problem with corner-size calculations from **Coord[*x*].CornerRadius** and **Coord[*x*].CornerAccel** with certain **Ts** S-curve times.

42. Fixed problem with 2D cutter compensation interference checking when checking two blended circle moves of different radii.

43. Fixed problem with triggered-move hardware capture for motors with servo loop closed in phase interrupt and single feedback (**Motor[*x*].pPhaseLoadEnc** = 0).

V2.0.2 Maintenance Release Updates (May 2015)

Updates from 2.0.0 release:

1. Moved EtherCAT amplifier-enable control logic from background thread to real-time interrupt.

2. Added new data structures to support modified EtherCAT amplifier-enable control logic: **Motor[*x*].EcatAmpEnable** and **Motor[*x*].EcatAmpFaultLimit**.

3. Permits partial-word access to EtherCAT I/O data structures (**ECAT[*i*].IO[*j*].data.*m.n***)

4. Allocated more memory to execute multi-motor commands from buffered program commands (e.g. **`jog/1..20`**).

5. Checks for adequate flash memory storage before starting a **`save`** operation. Sets **Sys.FlashSizeErr** status bit if insufficient space is found.

6. Reports start of rebooting process in response to **`reboot`** command.

7. Added appware and data structure support for hardware control panels: **CtrlPanel[*i*].Enable**, **Output[*j*]**, **Input[*j*]**, **ip[*j*]**, **Error**.

8. Added **Sys.ZeroVelSetPoint** to permit specification of velocity magnitude threshold for **Coord[*x*].DesVelZero** when trajectory pre-filter is used.

9. Added "legacy" servo algorithm in same arrangement as Turbo PMAC algorithm for ease of conversion. Selected by setting **Motor[*x*].Ctrl** to **Sys.LegacyCtrl**.

10. Increased permissible number of rollover cycles for compensation tables and cam tables.

11. Fixed operation of MuxIO data transfers with ACC-34 and compatible boards when using **Gate3[*i*].Gpio[*j*]** data registers for transfers.

12. Fixed erroneous early clearing of Power Brick AC amplifier faults.

13. Fixed erroneous allocation of local variables in kinematic subroutines. (Bug introduced in 2.0.0 firmware.)

14. Fixed operation of **`fsave`** command.

15. Fixed automatic power-on identification of Power Clipper hardware by extending ID chip range check, corrected support for IDE intellisense with Power Clipper.

16. Changed end-of-line character in MACRO backup files from <CR> to <LF> for compatibility with Linux computers.

17. Fixed intermittent problems with MacroRingOrderRepair and MacroRingOrderRestore functions.

18. Changed end-of-transmission response in MACRO ASCII communications from <CR> <LF> to <ACK><LF>.

19. Changed default value of **Sys.PhaseOverServoPeriod** for x86 configuration from 0.25 to 1.0 to reflect that default phase and servo periods are the same here.

20. Changed default values of **Sys.MaxMotors** and **Sys.MaxEncoders** for x86 configuration to 32.

21. Never back up saved setup element values of "undetected" ASICs for x86 configuration.

22. Added non-saved setup element **CamTable[*m*].Disable** for gradual removal of cam command in disabling process.

V2.0 Firmware Updates (January 2015)

1. Added support for AMCC465 (86*xxx*) CPUs, single-core and dual-core.

2. Added support for multi-core x86 CPUs for "Soft Power PMAC" with both Microsoft Windows™ and Linux running in other cores.

3. Added **Clipper[*i*]** alias for **Gate3[*i*]** data structure. Supports new Power Clipper embedded controller.

4. Added more complete support for new Power Brick product line.

5. Added data structure support for Compact UMAC ACC-84C serial-encoder interface board with **Acc84C[*i*]** data structure equivalent to existing **Acc84E[*i*]** data structure.

6. Added ability to manually specify presence of accessories on "I/O" chip select so older accessories without ID chip for auto-identification can be used with data structure access. Accomplished by providing write access to for **GateIo[*i*]** ID status elements **PartNum**, **PartOpt**, **PartRev**, and **PartType**. Primarily to support ACC-84S Clipper serial encoder board and ACC-11E UMAC I/O board.

7. Added data structure support for Clipper ACC-84S serial-encoder interface board with **Acc84S[*i*]** data structure equivalent to existing **Acc84E[*i*]** data structure.

8. Added "auxiliary channel" structure elements to **Acc84*x*[*i*]** data structure to support ACC-84S boards with two encoder-interface ICs.

9. Added data structure support for UMAC ACC-11E digital I/O board with **Acc11E[*i*]** as alias for **GateIo[*i*]** data structure.

10. Added saved setup element **CamTable[*m*].PosBias** for user specification of target motor offset from table values.

11. Added saved setup elements **CamTable[*m*].DacGain**, **CamTable[*m*].MinPosError**, and **CamTable[*m*].MaxDac** to support iterative learning control for optimizing torque outputs.

12. Added check of motor brake state when motor move is commanded. If brake is not fully disengaged (even if disengaging), move command will be rejected with an error.

13. Added capability for motors using "PosCtrl" position output mode to operate with **Motor[*x*].PosSf** values other than default of 1.0.

14. Added Script read access to **Motor[*x*].BrakeTimer** so user can monitor progress during **BrakeOffDelay** and **BrakeOnDelay** periods.

15. Added ability to provide "damping" in double integration in encoder conversion table entries to enhance stability. Extended maximum value of **EncTable[*n*].index4** from 2 to 15 in this mode, with values above 2 providing double integration with an exponential filter of weighting $1/(2^{16-index4})$. Higher **index4** values provide quicker damping.

16. Added new Type 12 method to encoder conversion table for single-register position read with separate error register. Intended mainly for serial encoder protocols.

17. Added new **EncTable[*n*].Status** element with bit 0 being automatically set to 1 in servo cycles where **MaxDelta** limit exceeded or Type 12 error detected. **Motor[*x*].pEncLoss** can use this register.

18. Added bit 2 (value 4) to **Motor[*x*].PhaseMode**. When set to 1, enables control of "tri-level" amplifier stage in direct-PWM mode using two ASIC channels.

19. Added capability to report hardware captured position to MACRO master when operating as a MACRO slave.

20. Added new "echo mode" for backup/save with composite full-word elements broken up into constituent partial-word elements. Enabled by setting bit 6 (value 64) of echo command value to 1.

21. Program **`read`** command now provides bit field for successfully read letter values in local variable **D54** as well as **D0**. Can be used to prevent rare occurrence of **D0** being overwritten by another task before it can be used by commands after **`read`**.

22. Increased maximum permitted velocity during current-loop auto-nulling process from **Motor[*x*].MaxSpeed** / 10,000 to **Motor[*x*].MaxSpeed** / 1,000.

23. Extended range of local ("L") variable numbers that can be used for 2D compensation table data points from L0 – L1 to L0 – L1023. Extended local variable numbers that can be used for 3D compensation table data points from L0 – L2 to L0 – L31 for first two indices and L0 – L63 for third index. Slight reduction of maximum constant index value for these tables.

24. Eliminated possible anomaly of a motor in closed-loop but amplifier-disabled state (which could be created by interrupt occurring in the middle of a state transition).

25. Modified coordinate system fault-sharing logic so that if a motor faults (e.g. overtravel limit, amplifier fault) when not executing a motion program, other motors in the coordinate system are not affected.

26. Changed default value of **Coord[*x*].AltFeedrate** from 0.0 to 1.0 to enable default use of alternate feedrate control when vector axes in feedrate moves have little or no distance compared to non-vector axes.

27. Fixed operation of automatic power-on absolute read of both servo and phase position (**Motor[*x*].PowerOnMode** = 6).

28. Fixed operation of auxiliary fault detection when **Motor[*x*].AuxFaultLevel** set to 1.

29. Fixed error message reported when issuing a control command to an inactive motor.

30. Fixed reporting of **C(*i*)**, **D(*i*)**, and **R(*i*)** array variable forms when listing programs.

31. Fixed response to **`list apc`** command for a never-executed program.

32. Fixed operation of trigonometric functions using radians at "exactly" 7π/4 radians.

33. Fixed direct transition from rapid to spline move mode.

34. Added software port numbers 5, 6, and 7 for the **send** command.

35. Added buffered program **nop*{expression}*** command so the returned value of functions do not have to be assigned to a variable.

36. Added **Sys.ioIdata[*i*]** and **Sys.ioUdata[*i*]** as unified alternate means of accessing I/O registers.

37. Added I/O elements **DPR[*i*].LinIdata16[*j*]**, **DPR[*i*].LinIdata32[*j*]**, **DPRLinUdata16[*j*]**, and **DPR[*i*].LinUdata32[*j*]** to support third-party shared-memory accessories with linear addressing scheme (not Turbo-PMAC compatible addressing).

V1.6.1 Maintenance Release Updates (May 2014)

Updates from V1.6.0. release:

1. Fixed operation of Encoder Conversion Table entry type 4 sine encoder when used with ACC-51E (PMAC2-style IC, **index5** = 0) – bug introduced in V1.6.0.

2. Fixed automatic read at power-up/reset of both phase and servo absolute position read (**Motor[*x*].PowerOnMode** = 6, **Motor[*x*].pAbsPhasePos** > 0, **Motor[*x*].pAbsPos** > 0).

3. Raised maximum permitted velocity during current-loop auto-null (**Motor[*x*].CurrentNullPeriod** > 0) from **Motor[*x*].MaxSpeed** / 10,000 to **Motor[*x*].MaxSpeed** / 1,000.

4. Modified "fault sharing" within a coordinate system so that when a motor in the coordinate system faults (e.g. following error, overtravel limit, amplifier fault) when a motion program is *not* running in the C.S. (as in a jogging or homing move), other motors in the coordinate system are not affected.

5. Introduced background check for motor "closed loop" status when "amplifier enabled" is not set – which can happen if an interrupt disrupts a state transition – and automatically clear "closed loop" status if this is found.

6. Fixed EtherCAT distributed-clock synchronization – bug introduced in V1.6.0.

7. In a coordinate system executing "negative time base", permitted operation to suspend when this reversal reached beginning of motor trace buffer, so can resume forward operation on next positive time base, rather than aborting program operation. (Trace buffer must not have wrapped.)

8. Permitted **PosCtrl** servo algorithm to output commanded positions properly when **Motor[*x*].PosSf** not equal to default of 1.0, so supports motor scaling in engineering units.

9. Permitted write operations to **EncTable[*n*].PrevDelta**. Write permission to this element name was accidentally removed in V1.6.0 when writeable alias **EncTable[*n*].EncBias** was introduced for integrating entries.

V1.6 Firmware Updates (February 2014)

1. Added support for new Power PC 465 CPU, single-core and dual-core.

2. Added cam table functionality through **CamTable[*m*]** data structure.

3. Added option for how motor position is reported when there are "offset" components (master position in offset mode, cam table position offsets) with new **Motor[*x*].PosReportMode** Boolean saved setup element.

4. Added capability for an additional fault bit for each motor with new saved setup elements **Motor[*x*].pAuxFault**, **Motor[*x*].AuxFaultBit**, **Motor[*x*].AuxFaultLevel**, **Motor[*x*].AuxFaultLimit**. Auxiliary fault functionality identical to that of "encoder loss" functionality. Can be used for secondary encoder loss or motor thermal fault, among other uses.

5. Added capability for a "global" abort input with new saved setup elements **Sys.pAbortAll**, **Sys.AbortAllBit**, **Sys.AbortAllLimit**, and **Coord[*x*].AbortAllMode**.

6. Added capability for a hybrid abort mode with the new coordinate system **`adisable`** command (on-line and programmed). All motors in the coordinate system are first brought to a controlled stop in the same manner as on an **`abort`** command. As each motor reaches a desired-velocity-zero status, a "delayed disable" is performed on the motor, with immediate brake engagement if used, followed by a "kill" after **Motor[*x*].BrakeOnDelay** time.

7. Added new saved setup element **Coord[*x*].AbortTimeBase** to permit the specification of the minimum time base percentage at which abort deceleration profiles will be executed.

8. Permitted more flexibility in how often motors are checked for status and safety updates with new **Sys.MotorsPerRtInt** saved setup element. If changed from its (backward-compatible) default value of 0, settings of **Motor[*x*].BlSlewRate**, **Motor[*x*].BrakeOffDelay**, **Motor[*x*].BrakeOnDelay**, **Motor[*x*].PhaseFindingTime**, and **Motor[*x*].I2tTrip** must be changed to maintain compatible operation.

9. Added ability for dual (motor/load) feedback when closing the servo loop in the phase interrupt with new saved setup elements **Motor[*x*].pPhaseLoadEnc**, **Motor[*x*].PhaseLoadEncLeftShift**, and **Motor[*x*].PhaseLoadEncRightShift**.

10. Provided more flexibility in single-step operation of motion programs with **Coord[*x*].StepMode**. In addition to existing single-move mode (= 0), supports CNC-style single-line mode (= 1), and debugging modes (= 2, 3).

11. Provided the capability to download addressing setup element values specifying non-present hardware without errors with **Sys.SimConfigOk**. This permits using a project configuration on a reduced or simulated system.

12. On system re-initialization, if no clock-producing ASICs are found, **Sys.CpuTimerIntr** is automatically set to 1 so the processor is generating its own interrupts internally. This automatically sets up Power PMAC for no-ASIC operation for networks such as EtherCAT, or for simulation mode.

13. Provided the capability to specify error magnitude of blended path at a corner relative to unblended corner port with new saved setup element **Coord[*x*].CornerError**.

14. Added the capability to maintain unscaled vector feedrate and 2D tool compensation radius when the XYZ Cartesian space is rescaled by transformation matrices with new non-saved setup element **Coord[*x*].TxyzScale**. New saved setup element **Coord[*x*].AutoTxyzScale** control bit permits the automatic calculation of **TxyzScale** based on the 3x3 XYZ portion of the transformation matrix.

15. Added new status bit **Motor[*x*].SoftLimitDir** to indicate which software overtravel limit caused move to be stopped, modified, or rejected (0 is positive limit, 1 is negative limit).

16. Provided protection against inadvertent excessive negative time base with new saved setup element **Sys.MaxTimedUnderflow**. If the accumulated negative time base for any motor goes more than this time back past the beginning of the stored equation span, motion will be aborted.

17. Provided the capability to buffer past motor trajectory information in motor "trace" buffers for reverse execution under negative time base values. Buffer size is set by new saved setup element **Motor[*x*].TraceSize**.

18. Provided the capability for the Power PMAC to automatically limit the slew rate of time base changes so that no motor acceleration limit is violated. This mode is enabled by setting **Coord[*x*].TimeBaseSlew** and **Coord[*x*].FeedHoldSlew** to negative values.

19. Supported new extended-resolution auto-correcting sinusoidal interpolator with new value of 7 for **Motor[*x*].EncType** and new encoder conversion table method (**EncTable[*n*].type** = 7) for extended hardware interpolation (65,536x interpolation).

20. Supported new extended-resolution auto-correcting sinusoidal interpolator with new value of 7 for **Motor[*x*].EncType** and new encoder conversion table method (**EncTable[*n*].type** = 7) for extended hardware interpolation (65,536x interpolation).

21. Provided the capability to ban single-character on-line "action" commands with **Sys.NoShortCmds**. This mode lessens the chance of issuing such a command by accident. (This was actually added in the V1.5 firmware.)

22. Provided new position-capture monitoring functionality with non-saved setup control element **Motor[*x*].CapturePos**, and status element **Motor[*x*].CapturedPos** (in motor units).

23. Added the capability for compensation tables to use source motor actual position instead of desired position to compute corrections with new saved setup element **CompTable[*m*].SourceCtrl**.

24. Permitted variable-time **spline** mode to operate like Turbo PMAC with new **Coord[*x*].SplineTimeRotate** Boolean saved setup element.

25. Added the capability for running Power PMAC motors as MACRO slaves with new elements **Motor[*x*].pMotorNode**, **Motor[*x*].MotorNodeOffset**, and **Motor[*x*].MotorMode**.

26. Provided the capability to combine cross-coupled and adaptive servo control algorithms by permitting motors with **Motor[*x*].Ctrl** = **Sys.GantryXCtrl** to use adaptive control gains for each motor. (Note that in the unlikely situation that an existing user of the **GantryXCtrl** algorithms had [unused] adaptive gains not equal to zero, this change would not be fully backward compatible.)

27. Provided more flexibility in adaptive servo algorithm with **Motor[*x*].Servo.MinW**, **Motor[*x*].Servo.MaxW**, **Motor[*x*].Servo.MinDR**, and **Motor[*x*].MaxDr**, permitting controlled variation in closed-loop natural frequency (W) and damping ratio (DR) over the range of detected inertias.

28. Provided support for new Power Brick products with **BrickAC** and **BrickLV** data structures, **PowerBrick[*i*]** alias for **Gate3[*i*]** data structure, and **`brickacver`**, **`bricklvver`** on-line query commands.

29. Added **Acc72EX[*i*]** data structure to support new ACC-72EX fieldbus interface board.

30. Provided more robustness in the calculation of very high-block rate applications. If requested calculation of next programmed move block would overflow pending buffer, it is delayed one real-time interrupt rather than immediately causing run-time error. New status element **Coord[*x*].BufferWarn** indicates if this condition occurs, which helps in optimizing the setup of **Sys.PreCalc** variable for these applications.

31. Corrected action when move calculation reaches last move in rotary motion program buffer or when fixed motion program flow jumps back more than **Coord[*x*].GoBack**+1 times. Execution now suspends so it can automatically resume (it had aborted program execution).

32. Permitted calculated move equations to be loaded from queue into active execution registers when coordinate system time base is exactly 0.0. This improves the position reference accuracy of triggered time-base starts.

33. Changed default value of **Coord[*x*].Ta** from 0.0 to 100.0 (msec). Changed default value of **Coord[*x*].Td** from 0.0 to 100.0 (msec). Change default value of **Coord[*x*].Ts** from 0.0 to 50.0 (msec). Helps beginning users make controlled moves.

34. Motor addressing variables **Motor[*x*].pLimits** and **Motor[*x*].pAmpFault** set to 0 on re-initialization for motors not assigned to auto-detected hardware interface channels. This disables hardware overtravel limit and amplifier fault checking for these motors, making it easier to set up virtual motors.

35. Made the following modal commands inoperative ("no-ops") in script PLC programs: **`pvt`**, **`spline`**, **`ta`**, **`tm`**, **`ts`**, **`td`**,and **`F`**. These commands still permitted to be entered into PLC programs, but do not do anything when executed. It is still possible to get the effect of these commands by writing directly to the data structure elements these commands would affect.

36. Added new saved setup element **EncTable[*n*].EncBias** to hold offset term added to source position before any numerical integration is performed. Shares register with status element **EncTable[*n*].PrevDelta**, which was previously used for this purpose.

V1.5.8 Maintenance Release Updates (Oct 2012)

Updates from V1.5.3.0 release:

1. Eliminated corruption of program buffers when same program loaded twice in one project.
2. Corrected operation of cutter radius compensation when using inverse kinematic transformation in a coordinate system.
3. Corrected operation of **`begin:0`** buffered program command so program counter properly points to rotary program buffer for the coordinate system.
4. Raised maximum command line length from 1024 characters to 8192 characters.
5. Corrected operation of "nanosleep" in independent C applications.
6. Corrected operation of EtherCAT distributed clocks to maintain proper lock with processor-based timer (**Sys.CpuTimerIntr**=1).

V1.5 Firmware Updates (June 2012)

1. Added new element **Motor[*x*].CompDesPos** as possible target register for compensation tables. Provides component of net desired position along with trajectory and master positions. Permits tables to be used as electronic cam tables.

2. Added ability to use user shared memory buffer array variables for Script vector and matrix functions by setting **Ldata.Control** to 2 for program.

3. Improved jog-style (jog, home, rapid) move blending on the fly algorithms.

4. Added auto-detection of Compact UMAC accessory boards: ACC-11C, ACC-24C2, ACC-24C2A, ACC-51C.

5. Added automatic calculation of filtered (moving average) task computation times with results in new elements **Sys.FltrPhaseTime**, **Sys.FltrServoTime**, **Sys.FltrRtiTime**, and **Sys.FltrBgTime**.

6. Added "inverse time" mode programming for linear and circle mode moves with new non-saved setup element **Coord[*x*].InvTimeMode**.

7. Added new saved setup element **Coord[*x*].HomeRequired**. If set to 1, all motors assigned to axes in coordinate system must have established position reference to permit a motion program to run in the coordinate system.

8. Added encoder conversion table entry type 11 to process floating-point register **Motor[*x*].IqCmd** as simulated feedback. This method supports virtual motors and virtual feedback for open-loop direct microstepping.

9. Added new correction elements **EncTable[*n*].CoverSerror** and **EncTable[*n*].TanHalfPhi** for sinusoidal encoder conversion (type 4). Added new element **EncTable[*n*].index5** to permit software arctangent conversion for DSPGATE3 interface to sinusoidal encoders.

10. Added current-feedback auto-nulling capability with new saved setup element **Motor[*x*].CurrentNullPeriod**. Automatically sets **Motor[*x*].IaBias** and **Motor[*x*].IbBias** terms on enabling of motor.

11. Added new bit 5 (value 32) to echo mode parameter. When set to 1, backup of motors and coordinate systems includes inactive motors and coordinate systems, facilitating fuller compare of different configurations.

12. Added capability to get absolute position data over MACRO ring using new settings of **Motor[*x*].AbsPosFormat** and **Motor[*x*].AbsPhasePosFormat**.

13. Added capability to use absolute position data from consecutive registers with a "gap" in data between first and second registers using new settings of **Motor[*x*].AbsPosFormat**.

14. Modified in-position check function so that in-position cannot be reported as true during the constant-velocity portion of indefinite jogs and homing-search moves.

15. Corrected checking of software overtravel limits during position following. Added saved setup element **Motor[*x*].SoftLimitOffset** to permit difference between calculation-time software limits and execution-time software limits.

16. Corrected calculation of very short spline moves to prevent buffer overflow.

17. Corrected implementation of servo output deadband: changed sign of integrator "seed"; added integrator seed when desired-velocity zero state reached with following error between inner and outer deadband limits.

V1.4 Firmware Updates (September 2011)

1. Added saved setup element **Sys.BgSleepTime** to permit user to specify the amount of time between background cycles for independent C applications, in microseconds. Formerly fixed at 1.0 milliseconds.

2. Added "spindle axis" definitions for motors (**#*x*->S**, **#*x*->S0**, **#*x*->S1**) to facilitate changing the motor between a rotary positioning axis and a velocity-mode spindle without have to delete and redefine the lookahead buffer for a different number of positioning motors in the coordinate system. Added status elements **Motor[*x*].SpindleMotor** and **Coord[*x*].LHMotorSlots** to store present status of motor and coordinate system in this regard.

3. Added saved setup elements **Motor[*x*].PwmDbComp** and **Motor[*x*].PwmDbI** to permit compensation in direct-PWM control mode for current deadband due to the required PWM deadtime.

4. Added saved setup elements in the new **GateIo[*i*].Init** sub-structure so the setup can be saved without affecting operation of the I/O itself.

5. Added data structure **Acc53E[*i*]** for ACC-53E UMAC SSI Encoder Interface board.

6. Added **PartType** status elements to accessory data structures **Gate1[*i*]** (and its aliases), **Gate2[*i*]** (and its aliases), **GateIo[*i*]** (and its aliases), **Acc28E[*i*]**, **Acc36E[*i*]**, **Acc59E[*i*]**, **Acc72E[*i*]**, **Acc84E[*i*]**, and **DPR[*i*]**, facilitating setup utilities.

7. Added **Sys.Uhex[*i*]** non-saved data structure elements in the user shared memory buffer; equivalent to **Sys.Udata[*i*]** except values are reported in hexadecimal format when queried.

8. Added cross-coupled gantry servo control algorithm, selected by setting **Motor[*x*].Ctrl** to new status address element **Sys.GantryXCtrl**. Adds new saved gain terms **Motor[*x*].Servo.Kxpg**, **Motor[*x*].Servo.Kxig**, and **Motor[*x*].Kxvg**.

9. Added status bit **Coord[*x*].ProgProceeding**, similar to existing status bit **Coord[*x*].ProgRunning**, except it goes to zero when stopped by a feed hold.

10. Added ability to use "-" as a unary negation operator on a data structure element.

11. Added "bit field" access to **Gate3[*i*].GpioData[*j*]** 32-bit I/O elements: **Gate3[*i*].GpioData[*j*].*m*.*n*** specifies the use of ***n*** bits starting at bit ***m*** in the element.

12. Added new bit (bit 1, value 2) to **Motor[*x*].PhaseMode** that, when set, disables integrator in direct current loop; facilitates direct-PWM control of brush DC motors.

13. Added capability for 32-bit position output of **PosCtrl** servo algorithm to roll over.

14. Added USB-to-RS232 converter capability to serial driver in kernel.

15. Added auto-detection of monitor resolution to video driver software. Had been fixed at 1024x768.

16. Added capability to write network settings to installed USB memory stick when powering up. Permits the discovery of the Power PMAC's IP address without the need for a communications link.

17. Changed calculation of "transparent" software overtravel limit status bits; now depend on motor desired position instead of actual position, preventing toggling when stopped at limit.

18. Ensured that independent C applications are automatically stopped on **$$$** or **$$$***** commands.

19. After **$$$***** re-initialization, the file `pp_default.cfg` is automatically created in directory `ftp://usrflash/Project/Configuration`, containing factory default settings.

20. Fixed reporting of address element **Motor[*x*].pEncLoss** when set to **Gate1[*i*].Chan[*j*].EncLossN.a** or equivalent address **Cid[*n*].PartData[*j*].a** for ACC-24E2*x* quadrature loss bit.

21. Fixed operation of "sign is direction" rotary axis rollover mode (**Coord[*x*].PosRollover[*i*]** < 0).

22. Fixed error trapping of invalid system commands by `gpascii` utility.

23. Fixed memory leak in real-time kernel semaphore creation.

24. Added saved software substructure **GateIo.Init** to permit the automatic configuration of the hardware **GateIo** setup elements at power-on/reset.

V1.3 Firmware Updates (January 2011)

1. Position following enhancements: New saved setup element **Motor[*x*].SlewMasterPosSf** permits automatic gradual changes in following gear ratio. New saved setup elements **Motor[*x*].MasterMaxSpeed** and **Motor[*x*].MasterMaxAccel** permit automatic saturation of following speed and acceleration, with option of re-synchronization to speed or position when coming out of saturation.

2. Automatic encoder-loss detection and resulting shutdown enabled with new saved setup elements **Motor[*x*].pEncLoss**, **Motor[*x*].EncLossBit**, **Motor[*x*].EncLossLevel**, and **Motor[*x*].EncLossLimit**. Encoder-loss fault is treated like fatal following error fault or amplifier fault.

3. The ability to disable ("kill") a motor on hitting a hardware limit (if the software limit has not already been hit) instead of bringing it to a closed-loop stop ("aborting") has been provided with new bit 2 (value 4) of existing saved setup element **Motor[*x*].FaultMode**. This mode is suggested to handle cases of uncontrolled excursions into the hardware limits, usually due to some kind of feedback failure.

4. Implemented multiple-move buffering of axis target positions with new saved setup elements **Coord[*x*].TPSize** and **Coord[*x*].TPCoords** for reporting of target positions with new on-line command **t** and new buffered program command **tread**, and reporting of "distance to go" with new on-line command **g** and new buffered program command **dtogread**.

5. Implemented buffered program axis query commands **dread** (for desired positions), **fread** (for following errors), **pread** (for actual positions), and **vread** (for filtered actual velocities) that place the returned values into the local D-variable array for the program.

6. String-variable and character-array manipulation functions provided with **Sys.Cdata[*i*]** character arrays in the user shared memory buffer, **memset** and **memcpy** character-array manipulation functions, and **sprintf**, **strcpy**, **strncpy**, **strtolower**, **strtoupper**, **strcat**, **strncat**, **strchr**, **strrchr**, **strcmp**, **strncmp**, **strspn**, **strcspn**, **strlen**, **strpbrk**, **strstr**, and **strtod** string-variable manipulation functions. Ability to use string variables added to existing program buffer commands **cmd**, **send**, and **system**.

7. Automatic adaptive servo control algorithm for systems with changing inertia provided with new setting **Sys.AdaptiveCtrl** for existing saved setup element **Motor[*x*].Ctrl**, and new saved setup elements **Motor[*x*].EstTime**, **Motor[*x*].EstMinDac**, **Motor[*x*].Servo.NominalGain**, **Motor[*x*].Servo.MinGainFactor**, and **Motor[*x*].MaxGainFactor**.

8. Implemented "timer-assisted software capture" functionality in triggered moves to improve the capture accuracy for feedback types not processed through an ASIC counter with new setting of 3 for existing saved setup element **Motor[*x*].CaptureMode**. Requires that the trigger (but not position feedback) be processed through PMAC3-style

"DSPGATE3" ASIC, as it uses the timer circuitry in the ASIC to interpolate between servo-cycle positions. New saved setup element **Motor[*x*].ServoCaptTimeOffset** is used to set the interpolation timing properly.

9. Provided capability for user to implement custom interrupt routine in C for interrupts generated from a PMAC3-style "DSPGATE3" ASIC with real-time C routine `CaptCompISR`. Routine is enabled by setting new saved setup element **UserAlgo.CaptCompIntr** to 1.

10. Added capability for extended (256-servo-cycle) rotary buffering of up to 8 servo-loop actual position registers with new saved setup element **Motor[*x*].pBufPos** and **Motor[*x*].pBufPos2**.

11. Added capability to perform shift operations on commutation position feedback data with new saved setup elements **Motor[*x*].PhaseEncRightShift** and **Motor[*x*].PhaseEncLeftShift**. This permits registers with MSB of the feedback data not in bit 31 of the register, and registers with "garbage" data in low bits, to be used for commutation feedback.

12. Implemented data structures for direct access to hardware registers in several UMAC accessories: **Acc59E[*i*]**, **Acc59E3[*i*]**, **Acc72E[*i*]**, and **Acc84E[*i*]**.