

GE  
Automation & Controls  
Programmable Control Products

# PACSystems\*

## RX7i, RX3i and RSTi-EP

### CPU Programmer's

# Reference Manual

GFK-2950D

November 2018







**Warning**

Warning notices are used in this publication to emphasize that hazardous voltages, currents, temperatures, or other conditions that could cause personal injury exist in this equipment or may be associated with its use. In situations where inattention could cause either personal injury or damage to equipment, a Warning notice is used.



**Caution**

Caution notices are used where equipment might be damaged if care is not taken.

**Note:** Notes merely call attention to information that is especially significant to understanding and operating the equipment.

These instructions do not purport to cover all details or variations in equipment, nor to provide for every possible contingency to be met during installation, operation, and maintenance. The information is supplied for informational purposes only, and GE makes no warranty as to the accuracy of the information included herein. Changes, modifications, and/or improvements to equipment and specifications are made periodically and these changes may or may not be reflected herein. It is understood that GE may make changes, modifications, or improvements to the equipment referenced herein or to the document itself at any time. This document is intended for trained personnel familiar with the GE products referenced herein.

GE may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not provide any license whatsoever to any of these patents.

GE PROVIDES THE FOLLOWING DOCUMENT AND THE INFORMATION INCLUDED THEREIN AS-IS AND WITHOUT WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED STATUTORY WARRANTY OF MERCHANTABILITY OR FITNESS FOR PARTICULAR PURPOSE.

\* indicates a trademark of General Electric Company and/or its subsidiaries.  
All other trademarks are the property of their respective owners.

©Copyright 2014-2018 General Electric Company.  
All Rights Reserved

If you purchased this product through an Authorized Channel Partner, please contact the seller directly.

**General Contact Information**

Online technical support and GlobalCare	<a href="http://www.geautomation.com/support">www.geautomation.com/support</a>
Additional information	<a href="http://www.geautomation.com">www.geautomation.com</a>
Solution Provider	<a href="mailto:solutionprovider.ip@ge.com">solutionprovider.ip@ge.com</a>

**Technical Support**

If you have technical problems that cannot be resolved with the information in this manual, please contact us by telephone or email, or on the web at [www.geautomation.com/support](http://www.geautomation.com/support)

**Americas**

Phone	1-800-433-2682
International Americas Direct Dial	1-780-420-2010 (if toll free 800 option is unavailable)
Customer Care Email	<a href="mailto:digitalsupport@ge.com">digitalsupport@ge.com</a>
Primary language of support	English

**Europe, the Middle East, and Africa**

Phone	+800-1-433-2682
EMEA Direct Dial	+ 420-296-183-331 (if toll free 800 option is unavailable or if dialing from a mobile telephone)
Customer Care Email	<a href="mailto:digitalsupport.emea@ge.com">digitalsupport.emea@ge.com</a>
Primary languages of support	English, French, German, Italian, Czech, Spanish

**Asia Pacific**

Phone	+86-400-820-8208
	+86-21-3877-7006 (India, Indonesia, and Pakistan)
Customer Care Email	<a href="mailto:digitalsupport.apac@ge.com">digitalsupport.apac@ge.com</a>
Primary languages of support	Chinese, Japanese, English

# Table of Contents

---

<b>RX7i, RX3i and RSTi-EP CPU Programmer's Reference Manual GFK-2950D .....</b>	<b>iii</b>
<b>Table of Contents.....</b>	<b>i</b>
<b>Table of Figures.....</b>	<b>xii</b>
<b>Chapter 1 Introduction .....</b>	<b>1</b>
<b>1.1 Revisions in this Manual.....</b>	<b>2</b>
<b>1.2 PACSystems Programming and Configuration.....</b>	<b>3</b>
<b>1.3 Migrating Series 90 Applications to PACSystems.....</b>	<b>3</b>
<b>1.4 PACSystems Documentation .....</b>	<b>4</b>
<b>Chapter 2 Program Organization.....</b>	<b>5</b>
<b>2.1 Structure of a PACSystems Application Program .....</b>	<b>6</b>
2.1.1 Blocks .....	6
2.1.2 Functions and Function Blocks .....	6
2.1.3 How Blocks Are Called.....	7
2.1.4 Nested Calls.....	7
2.1.5 Types of Blocks.....	8
2.1.6 Local Data.....	18
2.1.7 Parameter Passing Mechanisms .....	19
2.1.8 Languages.....	21
<b>2.2 Controlling Program Execution.....</b>	<b>24</b>
<b>2.3 Interrupt-Driven Blocks .....</b>	<b>25</b>
2.3.1 Interrupt Handling.....	26
2.3.2 Timed Interrupts .....	27
2.3.3 I/O Interrupts .....	27
2.3.4 Module Interrupts.....	27
2.3.5 Interrupt Block Scheduling.....	28
<b>Chapter 3 Program Data .....</b>	<b>29</b>
<b>3.1 Variables.....</b>	<b>30</b>
3.1.1 Mapped Variables .....	30
3.1.2 Symbolic Variables.....	31
3.1.3 I/O Variables .....	32
3.1.4 Arrays .....	35
3.1.5 Variable Indexes and Arrays.....	35

<b>3.2</b>	<b>Reference Memory.....</b>	<b>38</b>
3.2.1	Word (Register) References .....	38
3.2.2	Bit (Discrete) References .....	40
<b>3.3</b>	<b>User Reference Size and Default .....</b>	<b>41</b>
3.3.1	%G User References and CPU Memory Locations.....	41
<b>3.4</b>	<b>Genius Global Data .....</b>	<b>42</b>
<b>3.5</b>	<b>Transitions and Overrides .....</b>	<b>43</b>
<b>3.6</b>	<b>Retentiveness of Logic and Data .....</b>	<b>44</b>
<b>3.7</b>	<b>Data Scope.....</b>	<b>45</b>
<b>3.8</b>	<b>System Status References.....</b>	<b>46</b>
3.8.1	%S References.....	47
3.8.2	%SA, %SB, and %SC References.....	48
3.8.3	Fault References.....	50
<b>3.9</b>	<b>How Program Functions Handle Numerical Data .....</b>	<b>52</b>
3.9.1	Data Types.....	52
3.9.2	Floating Point Numbers .....	54
<b>3.10</b>	<b>User Defined Types (UDTs) .....</b>	<b>56</b>
3.10.1	Working with UDTs .....	56
3.10.2	UDT Properties .....	56
3.10.3	UDT Limits .....	57
3.10.4	RUN Mode Store of UDTs.....	57
3.10.5	UDT Operational Notes .....	58
<b>3.11</b>	<b>Operands for Instructions .....</b>	<b>59</b>
<b>3.12</b>	<b>Word-for-Word Changes .....</b>	<b>61</b>
3.12.1	Exception: Symbolic Variables .....	61
<b>Chapter 4</b>	<b>Ladder Diagram (LD) Programming .....</b>	<b>63</b>
<b>4.1</b>	<b>Advanced Math Functions .....</b>	<b>64</b>
4.1.1	Exponential/Logarithmic Functions .....	65
4.1.2	Square Root .....	66
4.1.3	Trig Functions .....	67
4.1.4	Inverse Trig – ASIN, ACOS, and ATAN.....	68
<b>4.2</b>	<b>Bit Operation Functions .....</b>	<b>69</b>
4.2.1	Data Lengths for the Bit Operation Functions.....	70
4.2.2	Bit Position .....	71
4.2.3	Bit Sequencer.....	72
4.2.4	Bit Set, Bit Clear .....	75

4.2.5	Bit Test .....	76
4.2.6	Logical AND, Logical OR, and Logical XOR .....	77
4.2.7	Logical NOT .....	79
4.2.8	Masked Compare .....	80
4.2.9	Rotate Bits .....	83
4.2.10	Shift Bits .....	84
<b>4.3</b>	<b>Coils.....</b>	<b>86</b>
4.3.1	Coil Checking.....	86
4.3.2	Graphical Representation of Coils .....	87
4.3.3	Set Coil, Reset Coil.....	88
4.3.4	Transition Coils .....	89
<b>4.4</b>	<b>Contacts .....</b>	<b>93</b>
4.4.1	Continuation Contact .....	94
4.4.2	Fault Contact .....	95
4.4.3	High and Low Alarm Contacts .....	96
4.4.4	No Fault Contact .....	97
4.4.5	Normally Closed and Normally Open Contacts.....	98
4.4.6	Transition Contacts .....	99
<b>4.5</b>	<b>Control Functions .....</b>	<b>104</b>
4.5.1	Do I/O.....	105
4.5.2	Edge Detectors.....	108
4.5.3	Drum .....	110
4.5.4	For Loop.....	114
4.5.5	Mask I/O Interrupt .....	116
4.5.6	Read Switch Position.....	117
4.5.7	Scan Set IO .....	118
4.5.8	Suspend I/O .....	119
4.5.9	Suspend or Resume I/O Interrupt .....	121
<b>4.6</b>	<b>Conversion Functions.....</b>	<b>122</b>
4.6.1	Convert Angles .....	123
4.6.2	Convert UINT or INT to BCD4 .....	124
4.6.3	Convert DINT to BCD8 .....	125
4.6.4	Convert BCD4, UINT, DINT, or REAL to INT .....	126
4.6.5	Convert BCD4, INT, DINT, or REAL to UINT .....	128
4.6.6	Convert BCD8, UINT, INT, REAL or LREAL to DINT .....	130
4.6.7	Convert BCD4, BCD8, UINT, INT, DINT, and LREAL to REAL .....	132
4.6.8	Convert REAL to LREAL .....	134
4.6.9	Convert DINT to LREAL.....	134
4.6.10	Truncate .....	135
<b>4.7</b>	<b>Counters.....</b>	<b>136</b>
4.7.1	Data Required for Counter Function Blocks .....	136
4.7.2	Down Counter .....	138

4.7.3	Up Counter .....	139
<b>4.8</b>	<b>Data Move Functions.....</b>	<b>141</b>
4.8.1	Array Size .....	143
4.8.2	Array Size Dimension Function Blocks.....	144
4.8.3	Block Clear .....	146
4.8.4	Block Move .....	147
4.8.5	BUS_ Functions .....	148
4.8.6	Communication Request (COMMREQ) .....	154
4.8.7	Data Initialization.....	159
4.8.8	Data Initialize ASCII .....	160
4.8.9	Data Initialize Communications Request .....	161
4.8.10	Data Initialize DLAN .....	162
4.8.11	Move.....	163
4.8.12	Move Data.....	165
4.8.13	Move Data Explicit .....	166
4.8.14	Move From Flat .....	167
4.8.15	Move to Flat .....	169
4.8.16	Shift Register .....	171
4.8.17	Size Of.....	173
4.8.18	Swap.....	174
<b>4.9</b>	<b>Data Table Functions.....</b>	<b>175</b>
4.9.1	Array Move .....	177
4.9.2	Array Range.....	179
4.9.3	FIFO Read.....	181
4.9.4	FIFO Write .....	183
4.9.5	LIFO Read.....	185
4.9.6	LIFO Write.....	186
4.9.7	Search.....	187
4.9.8	Sort.....	189
4.9.9	Table Read .....	190
4.9.10	Table Write.....	191
<b>4.10</b>	<b>Math Functions .....</b>	<b>192</b>
4.10.1	Overflow .....	193
4.10.2	Absolute Value .....	194
4.10.3	Add .....	195
4.10.4	Divide .....	197
4.10.5	Modulus .....	199
4.10.6	Multiply .....	200
4.10.7	Scale .....	202
4.10.8	Subtract .....	203
<b>4.11</b>	<b>Program Flow Functions.....</b>	<b>204</b>
4.11.1	Argument Present .....	205

4.11.2	Call .....	206
4.11.3	Comment.....	209
4.11.4	JumpN.....	210
4.11.5	Master Control Relay/End Master Control Relay .....	211
4.11.6	Wires.....	213
<b>4.12</b>	<b>Relational Functions.....</b>	<b>214</b>
4.12.1	Compare.....	215
4.12.2	Equal, Not Equal, Greater or Equal, Greater Than, Less or Equal, Less Than.....	216
4.12.3	EQ_DATA.....	217
4.12.4	Range.....	218
<b>4.13</b>	<b>Timers .....</b>	<b>219</b>
4.13.1	Timed Contacts.....	219
4.13.2	Timer Function Blocks .....	220
4.13.3	Standard Timer Function Blocks.....	231
<b>Chapter 5</b>	<b>Function Block Diagram (FBD) .....</b>	<b>237</b>
<b>5.1</b>	<b>Note on Reentrancy .....</b>	<b>238</b>
<b>5.2</b>	<b>Advanced Math Functions .....</b>	<b>239</b>
5.2.1	EXPT Function.....	241
<b>5.3</b>	<b>Bit Operation Functions .....</b>	<b>242</b>
5.3.1	Logical AND, Logical OR, and Logical XOR.....	244
5.3.2	Logical NOT.....	246
<b>5.4</b>	<b>Comments.....</b>	<b>247</b>
5.4.1	Text Block .....	247
<b>5.5</b>	<b>Comparison Functions .....</b>	<b>248</b>
5.5.1	Equal, Not Equal, Greater or Equal, Greater Than, Less or Equal, Less Than.....	250
<b>5.6</b>	<b>Control Functions .....</b>	<b>251</b>
<b>5.7</b>	<b>Counters.....</b>	<b>253</b>
<b>5.8</b>	<b>Data Move Functions .....</b>	<b>254</b>
5.8.1	Fan Out .....	258
5.8.2	Move Data .....	259
<b>5.9</b>	<b>Math Functions.....</b>	<b>261</b>
5.9.1	Overflow.....	262
5.9.2	Add.....	263
5.9.3	Divide.....	264
5.9.4	Modulus.....	265
5.9.5	Multiply .....	266

5.9.6	Negate .....	267
5.9.7	Subtract .....	268
<b>5.10</b>	<b>Program Flow Functions .....</b>	<b>270</b>
<b>5.11</b>	<b>Timers.....</b>	<b>271</b>
5.11.1	Built-in Timer Function Blocks .....	271
5.11.2	Standard Timer Function Blocks.....	272
<b>5.12</b>	<b>Type Conversion Functions .....</b>	<b>273</b>
5.12.1	Convert WORD to INT .....	275
5.12.2	Convert WORD to UINT .....	276
5.12.3	Convert DWORD to DINT.....	277
5.12.4	Convert INT or UINT to WORD .....	278
5.12.5	Convert DINT to DWORD.....	279
<b>Chapter 6</b>	<b>Service Request Function.....</b>	<b>281</b>
<b>6.1</b>	<b>Operation of SVC_REQ Function.....</b>	<b>283</b>
6.1.1	Ladder Diagram.....	283
6.1.2	Function Block Diagram .....	284
<b>6.2</b>	<b>SVC_REQ 1: Change/Read Constant Sweep Timer .....</b>	<b>285</b>
6.2.1	To disable Constant Sweep mode: .....	285
6.2.2	To enable Constant Sweep mode and use the old timer value:.....	285
6.2.3	To enable Constant Sweep mode and use a new timer value:.....	285
6.2.4	To change the timer value without changing the selection for sweep mode state: .....	285
6.2.5	To read the current timer state and value without changing either: .....	286
<b>6.3</b>	<b>SVC_REQ 2: Read Window Modes and Time Values.....</b>	<b>287</b>
<b>6.4</b>	<b>SVC_REQ 3: Change Controller Communications Window Mode.....</b>	<b>288</b>
6.4.1	To disable the controller communications window: .....	288
6.4.2	To re-enable or change the controller communications window mode:.....	288
<b>6.5</b>	<b>SVC_REQ 4: Change Backplane Communications Window Mode and Timer Value</b>	<b>289</b>
6.5.1	To disable the Backplane Communications window: .....	289
6.5.2	To enable the Backplane Communications window mode:.....	289
<b>6.6</b>	<b>SVC_REQ 5: Change Background Task Window Mode and Timer Value .....</b>	<b>290</b>
6.6.1	To disable the Background Task window: .....	290
6.6.2	To enable the Background Task window mode: .....	290
<b>6.7</b>	<b>SVC_REQ 6: Change/Read Number of Words to Checksum .....</b>	<b>291</b>
6.7.1	To read the word count:.....	291
6.7.2	To set a new word count: .....	291

<b>6.8</b>	<b>SVC_REQ 7: Read or Change the Time-of-Day Clock .....</b>	<b>293</b>
6.8.1	Parameter Block Formats .....	293
<b>6.9</b>	<b>SVC_REQ 8: Reset Watchdog Timer.....</b>	<b>301</b>
<b>6.10</b>	<b>SVC_REQ 9: Read Sweep Time from Beginning of Sweep.....</b>	<b>302</b>
<b>6.11</b>	<b>SVC_REQ 10: Read Target Name .....</b>	<b>303</b>
<b>6.12</b>	<b>SVC_REQ 11: Read Controller ID .....</b>	<b>304</b>
<b>6.13</b>	<b>SVC_REQ 12: Read Controller Run State.....</b>	<b>305</b>
<b>6.14</b>	<b>SVC_REQ 13: Shut Down (STOP) CPU .....</b>	<b>306</b>
<b>6.15</b>	<b>SVC_REQ 14: Clear Controller or I/O Fault Table .....</b>	<b>307</b>
<b>6.16</b>	<b>SVC_REQ 15: Read Last-Logged Fault Table Entry .....</b>	<b>308</b>
<b>6.17</b>	<b>SVC_REQ 16: Read Elapsed Time Clock.....</b>	<b>311</b>
<b>6.18</b>	<b>SVC_REQ 17: Mask/Unmask I/O Interrupt .....</b>	<b>313</b>
6.18.1	Masking/Unmasking Module Interrupts .....	313
<b>6.19</b>	<b>SVC_REQ 18: Read I/O Forced Status .....</b>	<b>315</b>
<b>6.20</b>	<b>SVC_REQ 19: Set Run Enable/Disable.....</b>	<b>316</b>
<b>6.21</b>	<b>SVC_REQ 20: Read Fault Tables .....</b>	<b>317</b>
6.21.1	Non-Extended Formats.....	318
6.21.2	Extended Formats .....	321

<b>6.22</b>	<b>SVC_REQ 21: User-Defined Fault Logging .....</b>	<b>326</b>
<b>6.23</b>	<b>SVC_REQ 22: Mask/Unmask Timed Interrupts .....</b>	<b>328</b>
<b>6.24</b>	<b>SVC_REQ 23: Read Master Checksum .....</b>	<b>329</b>
<b>6.25</b>	<b>SVC_REQ 24: Reset Module .....</b>	<b>330</b>
<b>6.26</b>	<b>SVC_REQ 25: Disable/Enable EXE Block and Standalone C Program Checksums....</b>	<b>331</b>
<b>6.27</b>	<b>SVC_REQ 29: Read Elapsed Power Down Time.....</b>	<b>332</b>
<b>6.28</b>	<b>SVC_REQ 32: Suspend/Resume I/O Interrupt .....</b>	<b>333</b>
<b>6.29</b>	<b>SVC_REQ 45: Skip Next I/O Scan.....</b>	<b>334</b>
<b>6.30</b>	<b>SVC_REQ 50: Read Elapsed Time Clock .....</b>	<b>335</b>
<b>6.31</b>	<b>SVC_REQ 51: Read Sweep Time from Beginning of Sweep .....</b>	<b>337</b>
<b>6.32</b>	<b>SVC_REQ 56: Logic Driven Read of Nonvolatile Storage .....</b>	<b>338</b>
6.32.1	Discrete Memory .....	338
6.32.2	Storage Disabled Conditions .....	338
6.32.3	Maximum of One Active Instruction .....	338
6.32.4	ENO and Power Flow To The Right .....	338
6.32.5	Parameter Block .....	339
<b>6.33</b>	<b>SVC_REQ 57: Logic Driven Write to Nonvolatile Storage.....</b>	<b>342</b>
6.33.1	Length of Data Written .....	342
6.33.2	Write Frequency .....	342
6.33.3	Erase Cycles.....	343
6.33.4	Discrete Memory .....	343
6.33.5	Retentiveness .....	343
6.33.6	Maximum of One Active Instruction .....	343
6.33.7	Storage Disabled Conditions .....	343
6.33.8	Error Checking .....	343
6.33.9	Fragmentation.....	344
6.33.10	When nonvolatile storage is full.....	344
6.33.11	Equality.....	345
6.33.12	Redundancy.....	345
6.33.13	ENO and Power Flow to the Right .....	345
6.33.14	Parameter Block for SVC_REQ 57 .....	346
<b>Chapter 7</b>	<b>PID Built-In Function Block .....</b>	<b>351</b>
<b>7.1</b>	<b>Operands of the PID Function .....</b>	<b>352</b>
7.1.1	Operands for LD Version of PID Function Block .....	352
7.1.2	Operands for FBD Version of PID Function Block .....	353

<b>7.2</b>	<b>Reference Array for the PID Function.....</b>	<b>354</b>
7.2.1	Scaling Input and Outputs .....	354
7.2.2	Reference Array Parameters .....	355
<b>7.3</b>	<b>Operation of the PID Function.....</b>	<b>362</b>
7.3.1	Automatic Operation .....	362
7.3.2	Manual Operation.....	362
7.3.3	Time Interval for the PID Function.....	363
<b>7.4</b>	<b>PID Algorithm Selection (PIDISA or PIDIND) and Gain Calculations .....</b>	<b>364</b>
7.4.1	Derivative Term .....	365
7.4.2	Error Term Mode .....	365
7.4.3	Derivative Action on PV Bit .....	365
7.4.4	Combined Operation of Error Term and Derivative Action Modes .....	365
7.4.5	CV Bias Term .....	366
7.4.6	CV Amplitude and Rate Limits.....	366
7.4.7	Sample Period and PID Function Block Scheduling .....	367
<b>7.5</b>	<b>Determining the Process Characteristics .....</b>	<b>368</b>
<b>7.6</b>	<b>Setting Tuning Loop Gains .....</b>	<b>369</b>
7.6.1	Basic Iterative Tuning Approach.....	369
7.6.2	Setting Loop Gains Using the Ziegler and Nichols Tuning Approach .....	370
7.6.3	Ideal Tuning Method.....	371
<b>7.7</b>	<b>PID Example .....</b>	<b>372</b>
7.7.1	Reference Array Initialization using %M00006 .....	372
<b>Chapter 8</b>	<b>Structured Text (ST) Programming .....</b>	<b>375</b>
<b>8.1</b>	<b>Language Overview.....</b>	<b>375</b>
8.1.1	Statements .....	375
8.1.2	Expressions.....	375
8.1.3	Operators.....	376
8.1.4	Structured Text Syntax.....	377
<b>8.2</b>	<b>Statement Types .....</b>	<b>378</b>
8.2.1	Assignment Statement .....	379
8.2.2	Function Call .....	380
8.2.3	RETURN Statement.....	383
8.2.4	IF Statement.....	384
8.2.5	CASE Statement .....	385
8.2.6	FOR ... DO Statements .....	387
8.2.7	WHILE Statement.....	389
8.2.8	REPEAT Statement.....	390
8.2.9	ARG_PRESENT Statement.....	391
8.2.10	Exit Statement .....	392

<b>Chapter 9</b>	<b>Diagnostics.....</b>	<b>393</b>
<b>9.1</b>	<b>Fault Handling Overview .....</b>	<b>394</b>
9.1.1	System Response to Faults .....	394
9.1.2	Fault Tables.....	394
9.1.3	Fault Actions and Fault Action Configuration .....	395
<b>9.2</b>	<b>Using the Fault Tables.....</b>	<b>396</b>
9.2.1	Controller Fault Table .....	396
9.2.2	I/O Fault Table.....	398
<b>9.3</b>	<b>System Handling of Faults.....</b>	<b>400</b>
9.3.1	System Fault References.....	401
9.3.2	Using Fault Contacts .....	404
9.3.3	Using Point Faults .....	406
9.3.4	Using Alarm Contacts .....	406
<b>9.4</b>	<b>Controller Fault Descriptions and Corrective Actions .....</b>	<b>407</b>
9.4.1	Controller Fault Groups .....	407
9.4.2	Loss of or Missing Rack (Group 1).....	408
9.4.3	Loss of or Missing Option Module (Group 4) .....	409
9.4.4	Addition of, or Extra Rack (Group 5).....	409
9.4.5	Reset of, Addition of, or Extra Option Module (Group 8) .....	410
9.4.6	System Configuration Mismatch (Group 11).....	411
9.4.7	System Bus Error (Group 12) .....	417
9.4.8	CPU Hardware Failure (Group 13).....	418
9.4.9	Module Hardware Failure (Group 14) .....	419
9.4.10	Option Module Software Failure (Group 16) .....	420
9.4.11	Program or Block Checksum Failure (Group 17) .....	421
9.4.12	Battery Status (Group 18) .....	422
9.4.13	Constant Sweep Time Exceeded (Group 19).....	423
9.4.14	System Fault Table Full (Group 20).....	423
9.4.15	I/O Fault Table Full (Group 21) .....	423
9.4.16	User Application Fault (Group 22).....	424
9.4.17	CPU Over-Temperature (Group 24).....	426
9.4.18	Power Supply Fault (Group 25).....	426
9.4.19	No User Program on Power-Up (Group 129).....	426
9.4.20	Corrupted User Program on Power-Up (Group 130).....	427
9.4.21	Window Completion Failure (Group 131).....	427
9.4.22	Password Access Failure (Group 132).....	428
9.4.23	Null System Configuration for RUN Mode (Group 134).....	428
9.4.24	CPU System Software Failure (Group 135) .....	429
9.4.25	Communications Failure During Store (Group 137) .....	431
9.4.26	Non-Critical CPU Software Event (Group 140).....	432

<b>9.5</b>	<b>I/O Fault Descriptions and Corrective Actions.....</b>	<b>434</b>
9.5.1	Fault Extra Data.....	434
9.5.2	I/O Fault Groups .....	434
9.5.3	I/O Fault Categories .....	435
9.5.4	Circuit Faults (Category 1) .....	439
9.5.5	Loss of Block (Category 2).....	444
9.5.6	Addition of Block (Category 3) .....	445
9.5.7	I/O Bus Fault (Category 6) .....	446
9.5.8	Module Fault (Category 8).....	447
9.5.9	Addition of IOC (Category 9).....	448
9.5.10	Loss of or Missing IO Controller (Category 10) .....	448
9.5.11	IOC (I/O Controller) Software Fault (Category 11).....	449
9.5.12	Forced and Unforced Circuit (Categories 12 and 13).....	449
9.5.13	Loss of or Missing I/O Module (Category 14).....	450
9.5.14	Addition of I/O Module (Category 15).....	450
9.5.15	Extra I/O Module (Category 16).....	450
9.5.16	Extra Block (Category 17).....	451
9.5.17	IOC Hardware Failure (Category 18).....	451
9.5.18	GBC Stopped Reporting Faults (Category 19) .....	451
9.5.19	GBC Software Exception (Category 21).....	452
9.5.20	Block Switch (Category 22).....	453
9.5.21	Reset of IOC (Category 27) .....	453
<b>9.6</b>	<b>Diagnostic Logic Blocks (DLBs).....</b>	<b>454</b>
9.6.1	DLB Operation.....	455
9.6.2	Executing DLBs.....	457
9.6.3	Diagnostic Logic Block (DLB) Example .....	461

## Table of Figures

Figure 1: Conditional Block Call	8
Figure 2: Block Call with Parameters	9
Figure 3: Defining Member Variables for a User-Defined Function Block	11
Figure 4: Creating a User-Defined Function Block	12
Figure 5: Use of User-Defined Function Block in Ladder Logic	12
Figure 6: Display of Instance Data Structures	12
Figure 7: Calling an External Block in Ladder Logic	15
Figure 8: Relationship of %L & %P to Program Blocks	18
Figure 9: Local Data (%L) Usage by Program Blocks	18
Figure 10: Parameter Passing Example	19
Figure 11: Explanation of Ladder Diagram Rung	21
Figure 12: Illustration of Function Block Diagram	22
Figure 13: Conflict Avoidance when using Interrupt-Driven Blocks	26
Figure 14: PID in Ladder Diagram	351
Figure 15: PID in Function Block Diagram	351
Figure 16: PID_IND Diagram	364
Figure 17: PID Example Logic	373
Figure 18: Controller Fault Table Display	396
Figure 19: Detail Information for Controller Fault Entry	397
Figure 20: I/O Fault Table Display	398
Figure 21: I/O Fault Table Fault Entry Detail Display	399
Figure 22: Diagnostic Logic Blocks (DLBs) assigned to Target in MPE	455
Figure 23: Properties of Diagnostic Logic Block (DLB)	457
Figure 24: DLB Heartbeat Setting	457
Figure 25: Drag DLB from Toolchest and Drop in Active Blocks Node	463
Figure 26: Set DLB Execution Mode to Sweep (Properties Tab)	463
Figure 27: Start DLB Execution	463
Figure 28: Initialize Local Symbolic Variables	464
Figure 29: DLB Icon and Status Bar after Execution has Commenced	464
Figure 30: Data Watch for DLB Variables	464

# Chapter 1 Introduction

---

This manual contains general information about programming a PACSystems CPU. It also provides detailed descriptions of specific programming requirements.

For a general introduction to the PACSystems family of products, including new features, product overviews, and specifications, see *PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual*, GFK-2222.

**Programming Features** are described in Chapter 2 through Chapter 8.

- Elements of an Application Program: Chapter 2
- Program Data: Chapter 3
- Ladder Diagram (LD) instruction set reference: Chapter 4
- Function Block Diagram (FBD) instruction set reference: Chapter 5
- The Service Request Function: Chapter 6
- The PID Function: Chapter 7
- Structured Text (ST): Chapter 8

**Diagnostics**, including Fault Handling and Diagnostic Logic Blocks are described in Chapter 9.

## 1.1 Revisions in this Manual

Rev	Date	Description
D	Nov-2018	CPE330/CPE400/CPL410 increased block count from 512 to 768 including _Main
C	Feb-2018	Updated for CPE302 throughout. Updated SVC_REQ 20 for newly-implemented feature that makes it possible to uniquely identify remote PROFINET IO faults recorded in the IO Fault Table by Remote Rack, Remote Slot, Remote Sub-Slot, and Device ID. Requires RX3i firmware version 9.40 or later.
B	Oct-2017	Added Redundancy and FA_OK System Bits (%S) Section 3.8.1.
A	May-2017	Changed the document Title and the contact information. Updated the Titles of the GFK's wherever applicable.
-	May-2015	<i>PACSystems RX7i and RX3i CPU Reference Manual</i> GFK-2222U Chapters 5-11 & Chapter 14 form the content of this new manual, the <i>PACSystems RX7i and RX3i CPU Programmer's Reference Manual</i> , GFK-2950. GFK-2222V and later versions defer to GFK-2950 for CPU programming content.

## 1.2 PACSystems Programming and Configuration

Proficy\* Machine Edition (PME) programming software provides a universal engineering development environment for all programming, configuration and diagnostics of PACSystems. A PACSystems CPU is programmed and configured using the programming software to perform process and discrete automation for various applications. The supported programming languages are documented in this manual.

## 1.3 Migrating Series 90 Applications to PACSystems

The PACSystems control system provides cost-effective expansion of existing systems. Support for existing Series 90 modules, expansion racks and remote racks protects your hardware investment. You can upgrade on your timetable without disturbing panel wiring.

- The RX3i supports most Series 90-30 modules, expansion racks, and remote racks. For a list of supported I/O, Communications, Motion, and Intelligent modules, see the *PACSystems RX3i System Manual*, GFK-2314.
- The RX7i supports most existing Series 90-70 modules, expansion racks and Genius networks. For a list of supported I/O, Communications, and Intelligent modules, see the *PACSystems RX7i Installation Manual*, GFK-2223.
- Conversion of Series 90-70 and Series 90-30 programs preserves existing development effort.
- Conversion of VersaPro and Logicmaster applications to Machine Edition allows smooth transition to PACSystems.

## 1.4 PACSystems Documentation

### PACSystems Manuals

<i>PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual</i>	GFK-2222
<i>PACSystems RX7i, RX3i and RSTi-EP CPU Programmer's Reference Manual</i>	GFK-2950
<i>PACSystems RX7i, RX3i and RSTi-EP TCP/IP Ethernet Communications User Manual</i>	GFK-2224
<i>PACSystems TCP/IP Ethernet Communications Station Manager User Manual</i>	GFK-2225
<i>C Programmer's Toolkit for PACSystems</i>	GFK-2259
<i>PACSystems Memory Xchange Modules User's Manual</i>	GFK-2300
<i>PACSystems Hot Standby CPU Redundancy User Manual</i>	GFK-2308
<i>PACSystems Battery and Energy Pack Manual</i>	GFK-2741
<i>Proficy Machine Edition Logic Developer Getting Started</i>	GFK-1918
<i>Proficy Process Systems Getting Started Guide</i>	GFK-2487
<i>PACSystems RXi, RX3i, RX7i and RSTi-EP Controller Secure Deployment Guide</i>	GFK-2830
<i>PACSystems RX3i &amp; RSTi-EP PROFINET I/O Controller Manual</i>	GFK-2571

### RX3i Manuals

<i>PACSystems RX3i System Manual</i>	GFK-2314
<i>DSM324i Motion Controller for PACSystems RX3i and Series 90-30 User's Manual</i>	GFK-2347
<i>PACSystems RX3i PROFIBUS Modules User's Manual</i>	GFK-2301
<i>PACSystems RX3i Max-On Hot Standby Redundancy User's Manual</i>	GFK-2409
<i>PACSystems RX3i Ethernet Network Interface Unit User's Manual</i>	GFK-2439
<i>PACMotion Multi-Axis Motion Controller User's Manual</i>	GFK-2448
<i>PACSystems RX3i PROFINET Scanner Manual</i>	GFK-2737
<i>PACSystems RX3i CEP PROFINET Scanner User Manual</i>	GFK-2883
<i>PACSystems RX3i Serial Communications Modules User's Manual</i>	GFK-2460
<i>PACSystems RX3i Genius Communications Gateway User Manual</i>	GFK-2892
<i>PACSystems RX3i DNP3 Outstation Module IC695EDS001 User's Manual</i>	GFK-2911
<i>PACSystems RX3i IEC 104 Server Module IC695EIS001 User's Manual</i>	GFK-2949

### RX7i Manuals

<i>PACSystems RX7i Installation Manual</i>	GFK-2223
<i>PACSystems RX7i User's Guide to Integration of VME Modules</i>	GFK-2235
<i>Series 90-70 Genius Bus Controller User's Manual</i>	GFK-2017

### Series 90 Manuals

<i>Series 90-30 Genius Bus Controller User's Manual</i>	GFK-1034
---	----------

### Distributed I/O Systems Manuals

<i>Genius I/O System User's Manual</i>	GEK-90486-1
<i>Genius I/O Analog and Discrete Blocks User's Manual</i>	GEK-90486-2

In addition to these manuals, datasheets and product update documents describe individual modules and product revisions. The most recent PACSystems documentation is available on the GE Automation & Control support website <http://geautomation.com/support>.

## **Chapter 2 Program Organization**

---

This chapter provides information about the operation of application programs in a PACSystems CPU.

- Structure of the Application Program
- Controlling Program Execution
- Interrupt-Driven Blocks

## 2.1 Structure of a PACSystems Application Program

A PACSystems application consists of one block-structured application program. The application program contains all the logic needed to control the operations of the CPU and the modules in the system. Application programs are created using the programming software and transferred to the CPU. Programs are stored in the CPU's non-volatile memory.

During the CPU Sweep, the CPU reads input data from the modules in the system and stores the data in its configured input memory locations. The CPU then executes the entire application program once, using this fresh input data. Executing the application program creates new output data that is placed in the configured output memory locations.

After the application program completes its execution, the CPU writes the output data to modules in the system. This completes the CPU Sweep.

A block-structured program always includes a \_MAIN block. Program execution begins with the \_MAIN block. Counting the \_MAIN block, the CPE330, CPE400 and CPL410 support up to 768 blocks with firmware release 9.70 or later. All other CPU models support up to 512 blocks. Note that PME 9.50 SIM 13 or later is also required for supporting a block count of up to 768.

### 2.1.1 Blocks

A block is a named section of executable logic that can be downloaded to and run on the target controller. The logic in a block can include functions, function blocks and calls to other blocks.

### 2.1.2 Functions and Function Blocks

A function is a type of instruction that has no internal storage (instance data). Therefore, it produces the same result for the same set of input values every time it executes.

A function block defines data as a set of inputs and output parameters that can be used as software connections to other blocks and internal variables. It has an algorithm that runs every time the function block is executed. Because a function block has instance data, that is it can store values, it has a defined state.

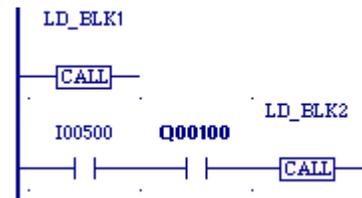
The following table describes the types of instructions that make up the PACSystems instruction set.

Instruction Type	Instance Data	Examples
Functions	None	BIT_SEQ, ADD, RANGE
Built-in function blocks	WORD array.	TMR, PID_IND, PID_ISA
Standard function blocks	Structure variable. (Refer to <i>Instance Data Structures</i> .)	TP, TOF, TON

**Note:** A user defined function block (UDFB) is a block of logic that can be called in your program logic to create multiple instances of the block, allowing you to create a block of logic once and reuse it as if it was a standard function block instruction. For additional information, refer to *Types of Blocks* and *User-Defined Function Blocks* (UDFBs).

### 2.1.3 How Blocks Are Called

A block executes when called from the program logic in the `_MAIN` block or another block. In this example, `LD_BLK1` is always called. Conditional logic can be used to control calling a block. For `LD_BLK2` to be called, input `%I00500` and output `%Q00100` must be ON. For details on using the Call function, refer to Chapter 4 (LD programming), Chapter 5 (FBD programming) or Chapter 8 (ST programming).



### 2.1.4 Nested Calls

The CPU allows nested block calls as long as there is enough execution stack space to support the call. If there is not enough stack space to support a given block call, an *Application Stack Overflow* fault is logged. In these circumstances, the CPU cannot execute the block. Instead, it sets all of the block's Boolean outputs to `FALSE`, and resumes execution at the point after the block call instruction.

**Note:** To halt the CPU when there is not enough stack space to execute a block, there are two choices. The best method is to add logic to detect the occurrence of any User Application Fault by testing the diagnostic bit `%SA38`, and then call `SVC_REQ 13` to halt the CPU. An alternative method is to add logic that tests for a negative `OK` value coming out of the block and then call `SVC_REQ 13` to halt the CPU.

A call depth of eight levels or more can be expected, except in rare cases where several of the called blocks have very large numbers of parameters. The actual call depth achieved depends on several factors, including the amount of data (non-Boolean) flow used in the blocks, the particular functions called by the blocks, and the number and types of parameters defined for the blocks. If blocks use less than the maximum amount of stack resources, more than eight nested calls may be possible. The call level nesting counts the `_MAIN` block as level 1.

### 2.1.5 Types of Blocks

PACSystems supports four types of blocks.

Block Type	Local Data	Programming Languages	Size Limit	Parameters
<b>Block</b>	Has its own local data	LD FBD ST	128 KB	0 inputs 1 output
<b>Parameterized Block</b>	Inherits local data from caller	LD FBD ST	128 KB	63 inputs 64 outputs
<b>User Defined Function Block (UDFB)</b>	Has its own local data	LD FBD ST	128 KB	63 inputs 64 outputs Unlimited internal member variables
<b>External Block</b>	Inherits local data from caller	C	user memory size limit (10 MB)	63 inputs 64 outputs

All PACSystems block types automatically provide an OK output parameter. The name used to reference the OK parameter within a block is Y0. Logic within the block can read and write the Y0 parameter. When a block is called, its Y0 parameter is automatically initialized to TRUE. This will result in a positive power flow out of the block call instruction when the block completes execution, unless Y0 is set to FALSE within the logic of the block.

For all block types, the maximum number of input parameters is one less than the maximum number of output parameters. This is because the EN input to the block call is not considered to be an input parameter to the block. It is used in LD language to determine whether or not to call the block, but is not passed into the block if the block is called.

#### Program Blocks

Any block can be a program block. The `_MAIN` block is automatically declared when you create a block-structured program. When you declare any other block, you must assign it a unique block name. A block is automatically configured with no input parameters and one output parameter (OK).

When a block-structured program is executed, the `_MAIN` block is automatically executed. Other blocks execute when called from the program logic in the `_MAIN` block, another block, or itself. In the following example, if `%M00001` is ON, the block named `ProcessEGD` will be executed:

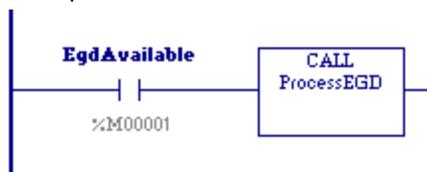


Figure 1: Conditional Block Call

#### Program Blocks and Local Data

Program blocks support the use of `%P` global data. In addition, each block, except `_MAIN`, has its own `%L` local data. Blocks do not inherit `%L` local data from their callers.

## Using Parameters with a Program Block

Every block is automatically defined to have one formal 'power flow' (or OK) output parameter, named Y0. Y0 is a BOOL parameter of LENGTH 1, passed by initial-value result. It indicates successful execution of the block. It can be read and written to by the logic within the block.

## Parameterized Blocks

Any block except `_MAIN` can be a parameterized block. When you declare a parameterized block, you must assign it a unique block name. A parameterized block can be configured with up to 63 input and 64 output parameters.

A parameterized block executes when called from the program logic in the `_MAIN` block, another block, or itself. In the following example, if `%I00001` is set, the parameterized block named `LOAD_41` will be executed.

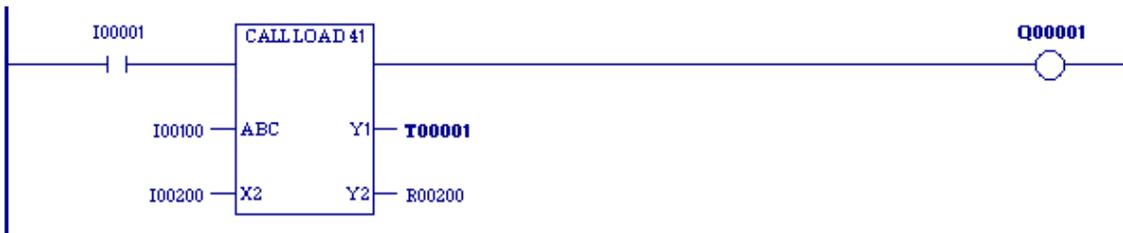


Figure 2: Block Call with Parameters

## Parameterized Blocks and Local Data

Parameterized blocks support the use of `%P` global data. Parameterized blocks do not have their own `%L` data, but instead inherit the `%L` data of their calling blocks. Parameterized blocks also inherit the `FST_EXE` system reference and *time-stamp* data that is used to update timer functions from their calling blocks. If `%L` references are used within a parameterized block and the block is called by `_MAIN`, `%L` references will be inherited from the `%P` references wherever encountered in the parameterized block (for example, `%L0005 = %P0005`).

**Note:** It is possible, by using Online Editing in the programming software to cause a parameterized block to use `%L` higher than allowed because of the way it inherits data. Using a word-for-word change to restore this reference to a valid address does not correct the block because the variable still exists in the variable list. Deleting the variable from the variable list does not cause an update to the CPU, so the parameterized block still sees the reference out of range fault. To correct this condition, you must remove the unused variables from the variable list after deleting them from the logic.

## Using Parameters with a Parameterized Block

A parameterized block may be defined to have between 0 and 63 formal input parameters, and between 1 and 64 formal output parameters. A 'power-flow out' (or OK) parameter, named Y0, is automatically defined for every parameterized block. It is a BOOL parameter of LENGTH 1, and indicates the successful execution of the parameterized block. It can be read and written to by the parameterized block's logic.

The following table lists the TYPEs, LENGTHs, and parameter-passing mechanisms allowed for parameterized block parameters. (For definitions of the parameter passing types, refer to *Parameter Passing Mechanisms*.)

Type	Length	Default Parameter Passing Mechanism
<b>BOOL</b>	1 to 256	INPUTS: by reference
		OUTPUTS: by value result; except Y0, which is by initial-value result
<b>BYTE</b>	1 to 1024	INPUTS: by reference
		OUTPUTS: by reference
<b>INT, UINT, and WORD</b>	1 to 512	INPUTS: by reference
		OUTPUTS: by reference
<b>DINT, REAL, and DWORD</b>	1 to 256	INPUTS: by reference
		OUTPUTS: by reference
<b>LREAL</b>	1 to 128	INPUTS: by reference
		OUTPUTS: by reference
<b>function block<sup>1</sup></b>	1	INPUTS: by reference
		OUTPUTS: not allowed
<b>UDFB<sup>1</sup></b>	1	INPUTS: by reference
		OUTPUTS: not allowed
<b>User Defined Type (UDT)</b>	1 to 1024	INPUTS: by reference
		OUTPUTS: not allowed

The PACSystems default parameter passing mechanisms correspond to the way that parameterized subroutine block (PSB) parameters are passed on 90-70 controllers. The parameter passing mechanisms of formal parameters cannot be changed from their default values.

Arguments, or *actual parameters*, are passed into a parameterized block whenever a parameterized block call is executed. In general, arguments to formal parameters may come from any memory type, may be data flow, and may be constants (when the formal parameter's LENGTH is 1). The following list contains the restrictions on arguments relative to this general rule:

- %S memory addresses cannot be used as arguments to any output parameter. This is because user logic is not allowed to write to %S memory.
- Indirect references used as arguments are resolved immediately before the parameterized block is called, and the corresponding direct reference is passed into the block. For example, where %R1 contains the value 10 and @R1 is used as an argument to a call, immediately before calling the block, @R1 is resolved to be %R10, and %R10 is passed in as the argument to the block. During execution of the block, the argument remains as %R10, regardless of whether the value in %R1 changes.

In general, formal parameters within a parameterized block may be used with any instruction or with any block call, as long as their TYPE and LENGTH are compatible with what the instruction, function, or block call requires. The following list contains the restrictions on formal parameters relative to this general rule:

- Formal parameters cannot be used on legacy transitional contacts or coils, or on FAULT, NOFLT, HIALM, or LOALM contacts. However, formal parameters can be used on IEC transitional contacts and coils.
- Formal BOOL input parameters cannot be used on coils or as output arguments to a function or to a block call.
- Formal parameters cannot be used with the DO I/O function.
- Formal parameters cannot be used with indirect referencing.

<sup>1</sup> A maximum of 16 input parameters can be of type function block or UDFB.

## User-Defined Function Blocks (UDFBs)

Users can define their own blocks, which have parameters and instance data, instead of being limited to the standard and built-in function blocks provided in the PACSystems instruction set. In many cases, the use of this feature results in a reduction in total program size.

Once defined, multiple instances of a UDFB can be created by calling it within the program logic. Each instance has its own unique copy of the function block's *instance data*, which consists of the function block's internal member variables and all of its input and output parameters except those that are passed by reference. When a UDFB is called on a given instance, the UDFB's logic operates on that instance's copy of the instance data. The values of the instance data persist from one execution of the UDFB to the next.

Note: A member variable is not passed into or out of a UDFB as a parameter. A member variable is used only within the logic of that function block.

A UDFB cannot be triggered by an interrupt.

UDFB logic is created using FBD, LD or ST. UDFB logic can make calls to all the other types of PACSystems blocks (blocks, parameterized blocks, external blocks and other UDFBs). Blocks, parameterized blocks, and other UDFBs can make calls to UDFBs.

Unless otherwise stated, the PACSystems implementation of UDFBs meets the IEC 61131-3 requirements for user defined function blocks.

### Defining a UDFB

To create a UDFB in the programming software, create an LD, FBD or ST block in the Program Blocks folder. In the Properties for the block, select Function Block.

To define instance data for a UDFB, select Parameters in the block's properties. Input and output parameters are defined in the same way as for parameterized blocks. In the following example, three internal member variables are defined: **temp**, **speed**, and **modelno**.

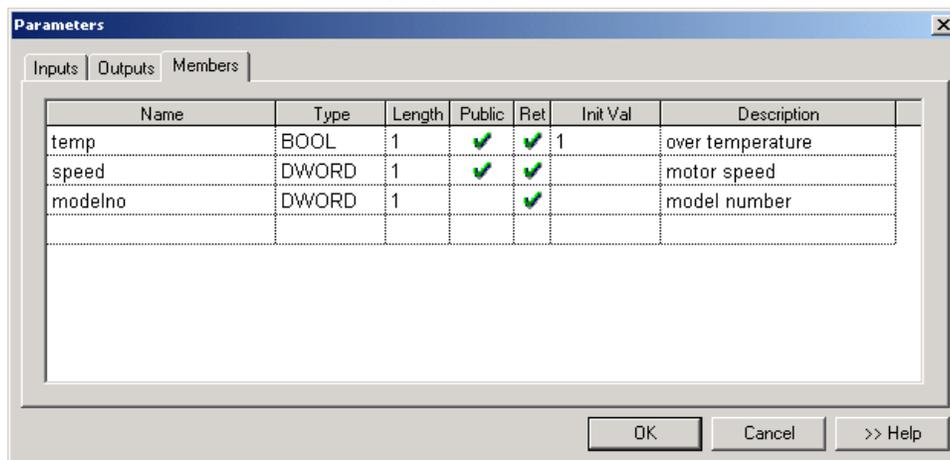


Figure 3: Defining Member Variables for a User-Defined Function Block

## Creating UDFB Instances

You create an instance of a UDFB by calling it in your logic and assigning an instance name in the function properties.

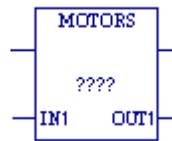


Figure 4: Creating a User-Defined Function Block

In the following LD example, the first rung creates two instances of the UDFB, Motors. The instance variables associated with the instances are `motors.motor1` and `motors.motor2`. The second rung uses the two instances of the internal variable **temp** in logic.

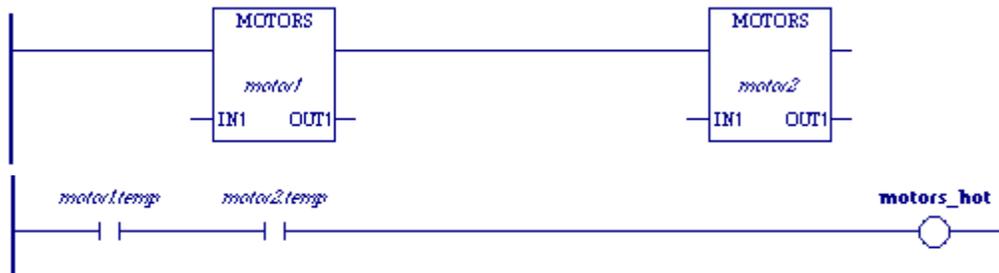


Figure 5: Use of User-Defined Function Block in Ladder Logic

## Instance Data Structures

A variable with the format **function\_block\_name.instance\_name** is automatically created for each instance of a UDFB. The instance data makes up a single composite variable that is of a structure type. The example to the right shows the variable structures associated with two instances of the UDFB named Motors. Each instance variable has elements corresponding to parameters **In1**, **Out1**, and **YO**, and internal variables **modelno**, **speed**, and **temp**.

Instances are created as symbolic variables, never as mapped variables. This ensures that instance data is only referenced by the instance name and not by a memory address, which means that no aliases can be created for the UDFB data elements. The indirect reference operator cannot be used on an instance variable because indirect references are not permitted on symbolic variables.

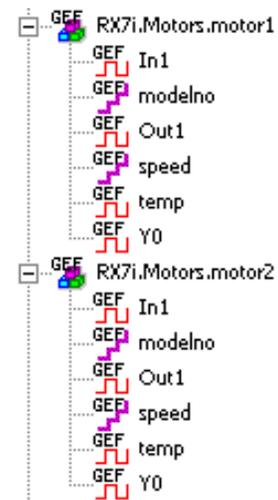


Figure 6: Display of Instance Data Structures

## UDFBs and Scope

Unlike a parameterized subroutine, a UDFB has its own %L memory.

By default, internal variables of a UDFB have local scope, making them visible only to the logic inside the UDFB. They cannot be read or written by any external logic or by the hardware configuration. An internal variable can be made visible outside the UDFB by changing its scope to global. Logic outside the UDFB can read but cannot write to internal variables whose scope is global.

**Note:** If you give internal variables global scope, your application will not conform to IEC requirements.

## Using Parameters with UDFBs

UDFBs support up to 63 inputs and up to 64 outputs.

Each UDFB has a predefined Boolean output parameter, Y0, which the CPU sets to true upon each invocation of the block. Y0 can be controlled by logic within the block and provides the output status of the block.

The following table lists the TYPEs, LENGTHs, and parameter-passing mechanisms allowed for UDFB parameters. For additional information on parameter passing, refer to *Parameter Passing Mechanisms*.

Type	Length	Parameter Passing Mechanism	Retentiveness of Instance Data for Parameters
<b>BOOL</b>	1 to 256	INPUTS: by reference, constant reference, value, or value result. (Default: value)	Not Applicable if passed by reference, since not stored in instance data. Can be retentive (default) or non-retentive for value or value result.
		OUTPUTS: by result; except Y0, which is by initial-value result	Retentive (default) or Non-retentive
<b>BYTE</b>	1 to 1024	INPUTS: by reference, constant reference, value, or value result. (Default: value)	Retentive for value or value result. Not applicable for reference
		OUTPUTS: by result	
<b>INT, UINT, and WORD</b>	1 to 512	INPUTS: by reference, constant reference, value, or value result. (Default: value)	Retentive for value or value result. Not applicable for reference
		OUTPUTS: by result	
<b>DINT, REAL, and DWORD</b>	1 to 256	INPUTS: by reference, constant reference, value, or value result. (Default: value)	Retentive for value or value result. Not applicable for reference
		OUTPUTS: by result	
<b>LREAL</b>	1 to 128	INPUTS: by reference, constant reference, value, or value result. (Default: value)	Retentive for value or value result. Not applicable for reference
		OUTPUTS: by result	
<b>Function block (standard or PACMotion)</b>	1	INPUTS: by reference, constant reference, (Default: reference)	Not applicable since passed by reference
		OUTPUTS: by result	
<b>UDFB<sup>2</sup></b>	1	INPUTS: by reference, constant reference, friend	Not applicable since passed by reference
		OUTPUTS: not allowed	
<b>UDT</b>	1 to 1024	INPUTS: by reference, constant reference	Not applicable since passed by reference
		OUTPUTS: not allowed	

<sup>2</sup> A maximum of 16 input parameters can be of type UDFB.

If an input parameter is passed by reference or by value result, it requires an argument. All other parameters of a UDFB are optional. That is, they do not have to be given arguments on each instance of the UDFB. If no argument is given for an optional parameter, the variable element associated with the parameter retains the value it previously had.

UDFB outputs cannot be passed as arguments to input parameters that are passed by reference or passed by value result. This restriction prevents modification of a UDFB output.

### Using Internal Member Variables with UDFBs

A UDFB can have any number of internal member variables. The values of internal variables are not passed via the input and output parameters. An internal variable cannot have the same name as a parameter of the UDFB it is defined in.

An internal variable can be:

- Any basic type supported by PACSystems (BOOL, INT, UINT, DINT, REAL, LREAL, BYTE, WORD, and DWORD).
- A UDFB type. Such member variables are known as nested instances. For example, the function block *Motor* can have an internal variable of type *Valve*, where *Valve* is a UDFB type. Note that defining a member variable as a UDFB type does not create an instance.

A nested instance cannot be of the same type as the UDFB being defined because this would set up an infinitely recursive definition. Nor can any level of a nested instance be of the same type as the parent UDFB being defined. For example, the UDFB *Motor* cannot have an internal variable of type *Valve*, if the *Valve* UDFB contains an internal variable of type *Motor*.

- A UDT: a structured, user-defined data type consisting of elements of other selected data types.
- A one-dimensional array.

Internal variables of TYPE BOOL can be retentive (default) or non-retentive. All other TYPEs must be retentive.

Member variables corresponding to a UDFB's input parameters cannot be read or written outside of the UDFB. (This is more restrictive than the IEC 61131-3 requirements for user defined function blocks.) Member variables corresponding to the UDFB's output parameters can be read but not written outside the UDFB.

Internal member variables that have basic types may be given initial values. The same initial values apply to all instances of a UDFB. If an initial value isn't given, the internal member variable is set to zero when the application transitions to RUN mode for the first time.

An internal member variable that is a nested instance has initial values as specified by its UDFB type definition.

Initial values are not stored during a RUN mode store. They will not take effect until a STOP Mode Store is performed.

## UDFB Logic

An instance of a BOOL parameter or internal variable can be forced ON or OFF, or used with transition-detecting instructions. The exception to this is that BOOL input parameters passed by reference cannot be forced or used with the Series 90-70 legacy transition-detecting instructions (POSCOIL, NEGCOIL, POSCON and NEGCON) because their values are not stored in instance data.

All input parameters to a UDFB, and their corresponding instance data elements, can be read by the logic of that particular UDFB.

Input parameters that are passed by reference or passed by value result to a UDFB can be written to by their UDFB's logic. Input parameters passed by value **cannot** be written to by their UDFB logic. Note that the restriction on writing to input parameters passed by value does not apply to other types of blocks.

All UDFB output parameters can be both read and written to by their logic.

## UDFB Operation with Other Blocks

A UDFB instance that is of global scope can be invoked by another UDFB's logic or any other block's logic.

A UDFB instance that is passed (by reference) as an argument to a UDFB can be invoked by the UDFB's logic.

A UDFB instance that is passed (by reference) as an argument to a parameterized block can be invoked by the parameterized block's logic.

The output parameters, and their corresponding instance data elements, of a UDFB instance that is passed as an argument can be read but not modified by the receiving block's logic. The input parameters of a UDFB instance that is passed as an argument cannot be read or modified by the receiving block's logic. The internal variables of a UDFB instance that is passed as an argument cannot be modified by the receiving block's logic. They can be read if their scope is global, but not if their scope is local.

## External Blocks

External blocks are developed using external development tools as well as the C Programmer's Toolkit for PACSystems. Refer to the *C Programmer's Toolkit for PACSystems*, GFK-2259 for detailed information regarding external blocks.

Any block except `_MAIN` can be an external block. When you declare an external block, you must assign it a unique block name. It can be configured with up to 63 input parameters and 64 output parameters.

An external block executes when called from the program logic in the `_MAIN` block or from the logic in another block, parameterized block, or UDFB. External blocks themselves cannot call any other block. In the following example, if `%I00001` is set, the external block named `EXT_11` is executed.



Figure 7: Calling an External Block in Ladder Logic

**Note:** Unlike other block types, external blocks cannot call other blocks.

### External Blocks and Local Data

External blocks support the use of %P global data. External blocks do not have their own %L data, but instead inherit the %L data of their calling blocks. They also inherit the FST\_EXE system reference and the *time-stamp* data that is used to update timer function blocks from their calling blocks. If %L references are used within an external block and the block is called by \_MAIN, %L references will be inherited from the %P references wherever encountered in the external block (for example, %L0005 = %P0005).

### Initialization of C Variables

When an external block is stored to the CPU, a copy of the initial values for its global and static variables is saved. However, if static variables are declared without an initial value, the initial value is undefined and must be initialized by the C application. (Refer to *Global Variable Initialization* and *Static Variable* in the *C Programmer's Toolkit for PACSystems*, GFK-2259). The saved initial values are used to re-initialize the block's global and static variables whenever the CPU transitions from STOP Mode to RUN Mode.

## Using Parameters with an External Block

An external block may be defined to have between zero and 63 formal input parameters and between one and 64 formal output parameters. A ‘power-flow out’ (or OK) parameter, named Y0, is automatically defined for every external block. Y0 is a BOOL parameter of LENGTH 1, and indicates the successful execution of the block. It can be read and written to by the external block’s logic.

The following table gives the TYPES, LENGTHs, and parameter-passing mechanisms allowed for external block parameters.

Type	Length	Default Parameter Passing Mechanism
<b>BOOL</b>	1 to 256	INPUTS: by reference
		OUTPUTS: by reference; except Y0, which is by initial-value result
<b>BYTE</b>	1 to 1024	INPUTS: by reference
		OUTPUTS: by reference
<b>INT, UINT, and WORD</b>	1 to 512	INPUTS: by reference
		OUTPUTS: by reference
<b>DINT, REAL, and DWORD</b>	1 to 256	INPUTS: by reference
		OUTPUTS: by reference
<b>LREAL</b>	1 to 128	INPUTS: by reference
		OUTPUTS: by reference
<b>UDT<sup>3</sup></b>	1 to 128	INPUTS: by reference
		OUTPUTS: not allowed

The PACSystems default parameter passing mechanisms correspond to the way that external block parameters are passed on 90-70 controllers. The parameter passing mechanisms of formal parameters cannot be changed from their default values.

You must define a name for each formal input and output parameter.

Arguments, or *actual parameters*, are passed into an external block whenever an external block call is executed.

Arguments may be any valid reference address including an indirect reference, may be flow, or may be a constant if the corresponding parameter’s LENGTH is 1.

<sup>3</sup> To use a UDT, you must include the UDT definition as a C structure in the external block. For details, refer to *Using a UDT as a C block input parameter data type* in the online help.

### 2.1.6 Local Data

Each block or UDFB in a block-structured program has an associated local data block. `_MAIN`'s data block memory is referenced by `%P`; all other data block memories are referenced by `%L`.

The size of the data block is dependent on the highest reference in its block for `%L` and in all blocks for `%P`.

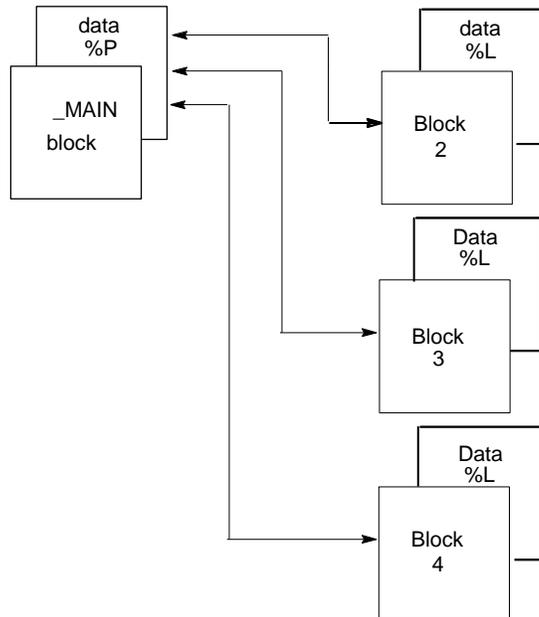


Figure 8: Relationship of `%L` & `%P` to Program Blocks

All blocks within the program can use data associated with the `_MAIN` block (`%P`). Blocks and UDFBs can use their own `%L` data as well as the `%P` data that is available to all blocks. The `_MAIN` block cannot use `%L`.

External blocks and parameterized blocks can use the Local Data (`%L`) of their calling block as well as the `%P` data of the `_MAIN` block. If a parameterized block or external block is called by `MAIN`, all `%L` references in the parameterized block or external block will actually be references to corresponding `%P` references (for example, `%L0005 = %P0005`). In addition to inheriting the Local Data of their calling blocks, parameterized blocks and external blocks inherit the `FST_EXE` status of their calling blocks.

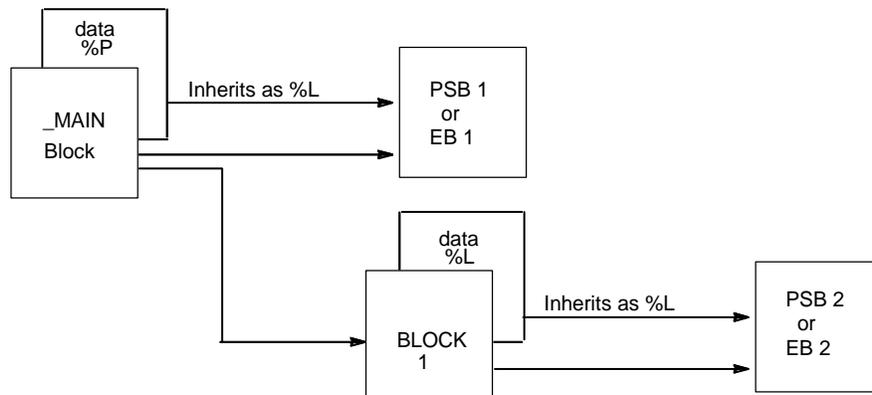


Figure 9: Local Data (`%L`) Usage by Program Blocks

### 2.1.7 Parameter Passing Mechanisms

All blocks (except `_MAIN`) have at least one parameter and thus are affected by parameter passing mechanisms. A *parameter passing mechanism* describes the way that data is passed from an argument in a calling block to a parameter in the called block, and from the parameter in the called block back to the argument in the calling block.

PACSystems supports the following parameter-passing mechanisms: pass by reference, pass by constant reference, pass by value, pass by value result, pass by result and pass by initial-value result. An additional type, pass by friend, is available when the input Data Type is a UDFB. A parameter is defined by its TYPE, LENGTH, and parameter passing mechanism.

- When a parameter is passed by **reference**, the address of its argument is passed into the function block instance or parameterized block. All logic within the called block that reads or writes to the parameter directly reads or writes to the actual argument.
- When a parameter is passed by **constant reference**, the CPU passes a reference address pointer, symbolic variable pointer, or I/O variable pointer into the function block instance or parameterized block. The instance or block can only read the reference address or variable.
- When a parameter is passed by **friend** (UDFB inputs only), the CPU passes a UDFB instance variable pointer into the function block instance or parameterized block. The instance or block can write to any output or member, whether public or private, of the UDFB instance variable passed as a friend.

**Tip:** In the logic of a UDFB, when you want to pass the UDFB as a friend, assign the pseudo-variable `#This` to the input that expects an instance variable of that UDFB type. In the following example, the `In2` input of the `LDPSB` parameterized block expects a UDFB instance variable friend of the `ABC` data type. Inside the logic of `ABC`, assign `#This` to `In2` in the call to `LDPSB`.



LDPSB Parameters							
Inputs							
	Name	Data Type	Length	Pass By	Retentive	Initial Value	Description
▶	In1	BOOL	1	Value			
	In2	ABC	1	Friend			
*					<input type="checkbox"/>		

Figure 10: Parameter Passing Example

- When a parameter is passed by **value (UDFB inputs only)**, the value of its argument is copied into a local stack memory associated with the called block. All logic within the called block that reads or writes to the parameter is reading or writing to this stack memory. Thus, no changes are ever made to the actual argument.

- When a parameter is passed by **value result (UDFB inputs only)**, the value of its argument is copied into a local stack memory associated with the called block, and the address of its argument is saved. All logic within the called block that reads or writes to the parameter is reading or writing to this stack memory. When the called block completes its execution, the value in the stack memory is copied back to the actual argument's address. Thus, no changes are made to the actual argument while the called block is executing, but when it completes execution, the actual argument is updated.

## 2.1.8 Languages

### Ladder Diagram (LD)

Logic written in Ladder Diagram language consists of a sequence of rungs that execute from top to bottom. The logic execution is thought of as *power flow*, which proceeds down along the left *rail* of the ladder, and from left to right along each rung in sequence.

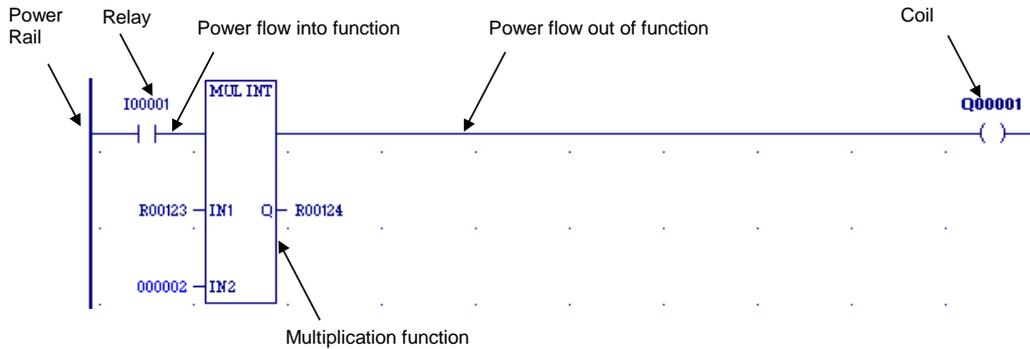


Figure 11: Explanation of Ladder Diagram Rung

The flow of logical power through each rung is controlled by a set of simple program instructions that work like mechanical relays and output coils. Whether or not a relay passes logical power flow along the rung depends on the content of a memory location with which the relay has been associated in the program. For instance, a relay might pass positive power flow if its associated memory location contains the value 1. The same relay passes negative power flow if the memory location contains the value 0.

Usually an instruction that receives negative power flow does not execute and propagates the negative power flow on to the next instruction in the rung. However, some instructions such as timers and counters execute even when they receive negative power flow, and may even pass positive power flow out. Once a rung completes execution, with either positive or negative power flow, power flows down along the left rail to the next rung.

Within a rung, there are many complex functions that are part of the standard function library and can be used for operations like moving data stored in memory, performing math operations, and controlling communications between the CPU and other devices in the system. Some program functions, such as the Jump function and Master Control Relay, can be used to control the execution of the program itself. Together, this large group of Ladder Diagram instructions and standard library functions makes up the *instruction set* of the CPU.

## Function Block Diagram

Function Block Diagram (FBD) is an IEC 61131-3 graphical programming language that represents the behavior of functions, function blocks and programs as a set of interconnected graphical blocks.

FBD depicts a system in terms of the flow of signals between processing elements, in a manner very similar to signal flows depicted in electronic circuit diagrams. Instructions are shown with inputs entering from the left and outputs exiting on the right. A function block type name is always shown within the element and the name of the function block instance is shown above the element.

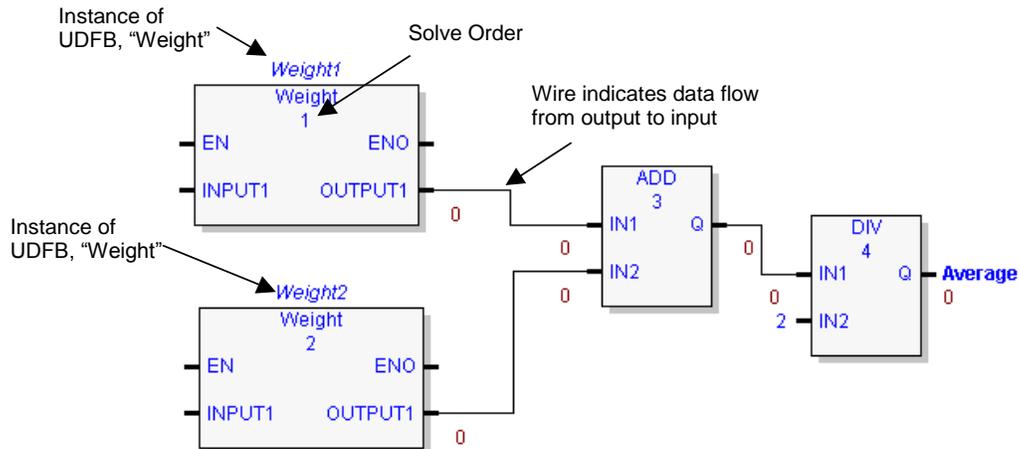


Figure 12: Illustration of Function Block Diagram

The order of execution of instructions in an FBD is determined by the following:

- a) The display position of the instruction in the FBD editor
- b) Whether the inputs to the FBD instruction are resolved.

To determine the order of execution of FBD instructions in the FBD editor, the FBD compiler performs the following steps:

1. The FBD compiler scans the instructions in the FBD editor, beginning from left to right, and top to bottom. When an instruction is encountered, the compiler attempts to resolve the instruction, that is, the inputs are known. If the inputs are known, the instruction is solved, and scanning continues for the next instruction.
2. If the current instruction cannot be resolved, that is, the inputs are not known, then the compiler scans for the previous instruction, using the wire connecting the output of the previous instruction to the input of the current instruction.
3. If the previous instruction can be resolved, the compiler calculates the output. The output of the previous instruction then becomes the input to the current instruction, the current instruction is resolved, and scanning continues for the next instruction.
4. If the previous instruction cannot be resolved, that is, the inputs are not known, then step 2 is repeated until an instruction is encountered, which can be resolved.

## Structured Text

The Structured Text (ST) programming language is an IEC 1131-3 textual programming language. A structured text program consists of a series of statements, which are constructed from expressions and language keywords. A statement directs the PLC to perform a specified action. Statements provide variable assignments, conditional evaluations, iteration, and the ability to call other blocks. For details on ST statements, parameters, keywords, and operators supported by PACSystems, refer to *Structured Text (ST) Programming* in Chapter 8.

Blocks, parameterized blocks, and UDFBs can be programmed in ST. The `_MAIN` program block can also be programmed in ST.

A block programmed in ST can call blocks, parameterized blocks, and UDFBs.

## 2.2 Controlling Program Execution

There are many ways in which program execution can be controlled to meet the system's timing requirements. The PACSystems CPU instruction set contains several powerful control functions that can be included in an application program to limit or change the way the CPU executes the program and scans I/O. For details on using these functions, refer to Chapter 4.

The following is a partial list of the commonly used methods:

- The Jump (JUMPN) function can be used to cause program execution to move either forward or backward in the logic. When a JUMPN function is active, the coils in the part of the program that is skipped are left in their previous states (not executed with negative power flow, as they are with a Master Control Relay). Jumps cannot span blocks.
- The nested Master Control Relay (MCRN) function can be used to execute a portion of the program logic with negative power flow. Logic is executed in a forward direction and coils in that part of the program are executed with negative power flow. Master Control Relay functions can be nested to 255 levels deep.
- The Suspend I/O function can be used to stop both the input scan and output scan for one sweep. I/O can be updated, as necessary, during the logic execution through the use of DO I/O instructions.
- The Service Request function can be used to suspend or change the time allotted to the window portions of the sweep.
- Program logic can be structured so that blocks are called more or less frequently, depending on their importance and on timing constraints. The CALL function can be used to cause program execution to go to a specific block. Conditional logic placed before the Call function controls the circumstances under which the CPU executes the block logic. After the block execution is finished, program execution resumes at the point in the logic directly after the CALL instruction.

## 2.3 Interrupt-Driven Blocks

Three types of interrupts can be used to start a block's execution:

- **Timed Interrupts** are generated by the CPU based on a user-specified time interval with an initial delay (if specified) applied on STOP Mode to RUN Mode transition of the CPU.
- **I/O Interrupts** are generated by I/O modules to indicate discrete input state changes (rising/falling edge), analog range limits (low/high alarms), and high-speed signal counting events.
- **Module Interrupts** are generated by VME modules. A single interrupt is supported per module.



### Caution

**Interrupt-driven block execution can interrupt the execution of non-interrupt-driven logic. Unexpected results may occur if the interrupting logic and interrupted logic access the same data. If necessary, Service Request #17 or Service Request # 32 can be used to temporarily mask I/O and Timed Interrupt-driven logic from executing when shared data is being accessed.**

---

### 2.3.1 Interrupt Handling

An I/O, Module, or Timed interrupt can be associated with any block except `_MAIN`, as long as the block has no parameters other than an OK output. After an interrupt has been associated with a block, that block executes each time the interrupt trigger occurs. A given block can have multiple timed, I/O, and module interrupt triggers associated with it. It is executed each time any one of its associated interrupts triggers. For details on how interrupt blocks are prioritized, refer to *Interrupt Block Scheduling*.

If a parameterized block or external block is triggered by an interrupt, it inherits `%P` data as its `%L` local data. For example, a `%L00005` reference in the parameterized block or C block actually references `%P00005`.

**Note:** Timer function blocks do not accumulate time if used in a block that is executed as a result of an interrupt.

Blocks that are triggered by interrupts can make calls to other blocks. The application stack used during interrupt-driven execution is different from the stack used during normal block-structured program execution. In particular, the nested call limit is different from the limit described for calls from the `_MAIN` block. If a call results in insufficient stack space to complete the call, the CPU logs an *Application Stack Overflow* fault.

**Note:** We strongly recommend that interrupt-driven blocks not be called from the `_MAIN` block or other non-interrupt driven blocks because the interrupt and non-interrupt driven blocks could be reading and writing the same global memories at indeterminate times relative to each other. In the following example (Figure 13) `INT1`, `INT2`, `BLOCK5`, and `PB1` should not be called from `_MAIN`, `BLOCK2`, `BLOCK3`, or `BLOCK4`.

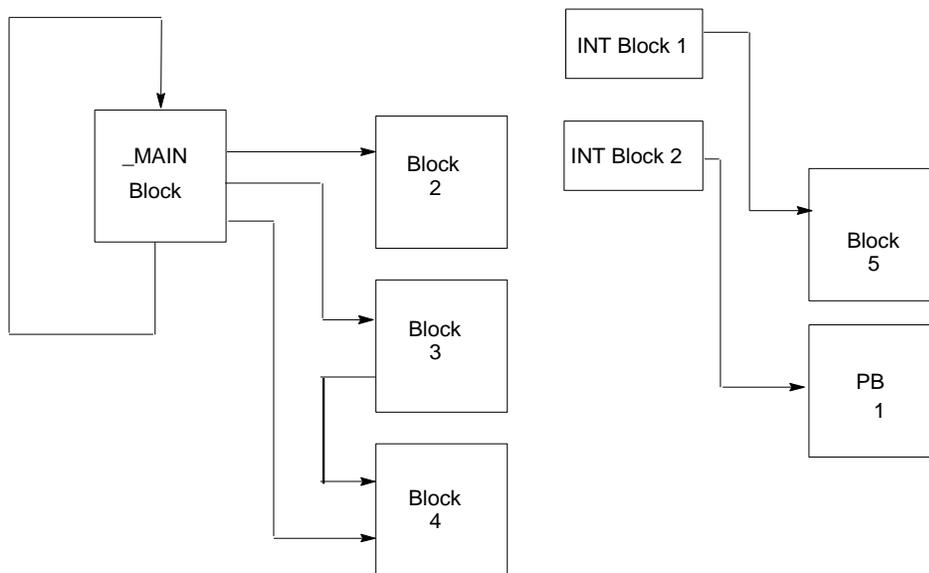


Figure 13: Conflict Avoidance when using Interrupt-Driven Blocks

### 2.3.2 Timed Interrupts

A block can be configured to execute on a specified time interval with an initial delay (if specified) applied on a STOP Mode to RUN Mode transition of the CPU.

To configure a timed interrupt block, specify the following parameters in the scheduling properties for the block:

<b>Time Base</b>	The smallest unit of time that you can specify for Interval and Delay. The time base can be 1.0 second, 0.10 second, or 0.01 second, or 0.001 second.
<b>Interval</b>	Specifies how frequently the block executes in multiples of the time base.
<b>Delay</b>	(Optional) Specifies an additional delay for the first execution of the block in multiples of the time base.

The first execution of a Timed Interrupt block will occur at  $((\text{delay} * \text{time base}) + (\text{interval} * \text{time base}))$  after the CPU is placed in RUN Mode.

### 2.3.3 I/O Interrupts

A block can be triggered by an interrupt input from certain hardware modules. For example, on the 32-Circuit 24 Vdc Input Module (IC697MDL650), the first input can be configured to generate an interrupt on either the rising or falling edge of the input signal. If the interrupt is enabled in the module configuration, that input can serve as a trigger to cause the execution of a block.

To configure an I/O interrupt, specify a trigger in the scheduling properties for the block. The trigger must be a global variable in %I, %AI or %AQ memory, or an I/O variable. (An I/O variable is a form of symbolic variable that is mapped to a module I/O point in hardware configuration.)

PACSystems modules that can trigger user interrupt logic always send the interrupt to the CPU when configured to do so. If the CPU is in STOP mode when it receives the interrupt, it does not run the user interrupt block. The CPU does not run the user interrupt block when it transitions from STOP Mode to RUN Mode.

### 2.3.4 Module Interrupts

A block can be triggered by an interrupt from a module that supports I/O interrupts if the Interrupt parameter is enabled in the module's hardware configuration.

To configure a module interrupt, specify the module by rack/slot/interrupt ID as the Trigger in the scheduling properties for the block.

### 2.3.5 Interrupt Block Scheduling

You can select one of two types of interrupt block scheduling at the target level:

- **Normal block scheduling** allows you to associate a maximum of 64 I/O and Module Interrupts and 16 Timed Interrupts. With normal block scheduling, all interrupt-triggered blocks have equal priority. This is the default scheduling mode.
- **Preemptive block scheduling** allows you to associate a maximum of 32 interrupt triggers. With preemptive block scheduling, each trigger can be assigned a relative priority.

#### Normal Block Scheduling

Interrupt-driven logic has the highest priority of any user logic in the system. The execution of a block triggered from an interrupt preempts the execution of the normal CPU sweep activities. Execution of the normal CPU sweep activities is resumed after the interrupt-driven block execution completes.

If the CPU receives one or more interrupts while executing an interrupt block, it places the incoming interrupts into the queue while it finishes executing the current interrupt block. Timed interrupt driven blocks are queued ahead of I/O or Module driven blocks. I/O or Module interrupt driven blocks are queued in the order in which the interrupts are received. If an interrupt driven block is already in the queue, additional interrupts that occur for this block are ignored.

#### Preemptive Block Scheduling

Preemptive scheduling allows you to assign a priority to each interrupt trigger. The priority values range from 1 to 16, with 1 being the highest. A single block can have multiple interrupts with different priorities or the same priorities.

An incoming interrupt is handled according to its priority compared to that of the currently executing block as follows:

- If an incoming interrupt has a higher priority than the interrupt associated with the block that is currently executing, the currently executing block is stopped and put in the interrupt queue. The block associated with the incoming interrupt begins executing.
- If an incoming interrupt has the same priority as the interrupt trigger associated with the block that is currently executing, that block continues to execute and the incoming interrupt is placed in the queue.
- If an incoming interrupt has a lower priority than the interrupt associated with the block that is currently executing, the incoming interrupt is placed in the queue.

When the CPU completes the execution of an interrupt block, the block associated with the interrupt trigger that has the highest priority in the queue begins execution — or resumes execution if the block's execution was preempted by another interrupt block and was placed in the queue.

If multiple blocks in the queue have the same interrupt priority, their execution order is not deterministic.

**Note:** Certain functions, such as DOIO, BUS\_RD, BUS\_WRT, COMMREQ, SCAN\_SET\_IO, and some SVC\_REQs may cause a block to yield to another queued block that has the same priority

## Chapter 3 Program Data

---

This chapter describes the types of data that can be used in an application program, and explains how that data is stored in the PACSystems CPU's memory.

- *Variables*
- *Reference Memory*
- *User Reference Size and Default*
- *Genius Global Data*
- *Transitions and Overrides*
- *Retentiveness of Logic and Data*
- *Data Scope*
- *System Status References*
- *How Program Functions Handle Numerical Data*
- *User Defined Types (UDTs)*
- *Operands for Instructions*
- *Word-for-Word Changes*

## 3.1 Variables

A variable is a named storage space for data values. It represents a memory location in the target PACSystems CPU.

A variable can be mapped to a reference address (for example, %R00001). If you do not map a variable to a specific reference address, it is considered a *symbolic variable*. The programming software handles the mapping for symbolic variables in a special portion of PACSystems user space memory.

The kinds of values a variable can store depend on its data type. For example, variables with a UINT data type store unsigned whole numbers with no fractional part. Data types are described in *How Program Functions Handle Numerical Data*.

In the programming software, all variables in a project are displayed in the Variables tab of the Navigator. You create, edit, and delete variables in the Variables tab. Some variables are also created automatically by certain components (such as TIMER variables when you add a Timer instruction to ladder logic). The data type and other properties of a variable, such as reference address are configured in the Inspector.

For more information about *system variables*, which are created when you create a target in the programming software, refer to *System Status References*.

### 3.1.1 Mapped Variables

Mapped (manually located) variables are assigned a specific reference address. For details on the types of Reference Memory and their uses, refer to *Reference Memory*.

### 3.1.2 Symbolic Variables

Symbolic variables are variables for which you do not specify a reference address (similar to a variable in a typical high-level language). Except as noted in this section, you can use these in the same ways that you use mapped variables.

In the programming software, a symbolic variable is displayed with a blank address. You can change a mapped variable to a symbolic variable by removing the reference address from the variable's properties. Similarly, you can change a symbolic variable into a mapped variable by specifying a reference address for the variable in its properties.

The memory required to support symbolic variables counts against user space. The amount of space reserved for these variables is configured on the Memory tab in the CPU hardware configuration.

#### Restrictions on the Use of Symbolic Variables

- Symbolic variables cannot be used with indirect references (for example, @Name). For a full description, refer to *Indirect References*.
- Only global scope Symbolic variables can be used in EGD pages.
- A variable must be globally scoped and published (internal or external) to be used in a C block.
- Symbolic variables cannot be used in the COMMREQ status word.
- Use of symbolic variables is not supported on web pages.
- Symbolic Boolean variables are not allowed on non-BOOL parameters.
- Symbolic non-discrete variables cannot be used on Series 90-70 style Transition contacts and coils. (Symbolic discrete variables are supported.)
- Overrides and Forces cannot be used on symbolic non-discrete variables. (Symbolic discrete variables are supported.)
- Arrays of the following data types are not supported:
  - Arrays of user defined function block (UDFB) instance variables.
  - Arrays of PACMotion function block instance variables.
  - Arrays of TON, TOF, or TP instance variables.
  - Arrays of reference ID variables (RIVs) that contain one or more linked RIV elements.

**Note:** An RIV array is supported when none of its elements is linked.

### 3.1.3 I/O Variables

An I/O variable is a symbolic variable that is mapped to a terminal in the hardware configuration. A terminal can be one of the following: Physical discrete or analog I/O point on a PACSystems module or on a Genius device, a discrete or analog status returned from a PACSystems module, or Global Data. The use of I/O variables allows you to configure hardware modules without having to specify the reference addresses to use when scanning their inputs and outputs. Instead, you can directly associate variable names with a module's inputs and outputs.

As with symbolic variables, memory required to support I/O variables counts against user space. You can configure the space available for I/O variables in the Memory tab of the PACSystems CPU.

For a given module or Genius bus, you must use either I/O variables or manually located mapped variables: you cannot use both in combination. It is not necessary to map all points on a module. Points that are disconnected or unused can be skipped. When points are skipped, space is reserved in user memory for that point (that is, a 32-point discrete module will always use 32 bits of memory).

The hardware configuration (HWC) and logic become coupled in a PACSystems target on your computer as soon as you do one of the following: Enable I/O variables for a module or Genius bus (even if you don't create any I/O variables), use one or more symbolic variables in the Ethernet Global Data (EGD) component, or upload a coupled HWC and logic from a PACSystems PLC. The HWC and logic become coupled in a PACSystems controller when coupled HWC and logic are downloaded to it.

Effects of coupled HWC and logic:

- Whether the HWC and logic are coupled in the PACSystems target on your computer or in the PACSystems controller, you cannot download or upload the HWC and logic independently.
- When the HWC and logic are coupled in the PACSystems controller, you cannot clear the HWC and the logic independently.
- As for any download, you cannot *RUN Mode Store* (RMS) the HWC and logic independently.
- The HWC must be completely equal for you to make word-for-word changes, launch the Online Test mode of Test Edit, or accept the edits of Test Edit.

I/O variables can be used any place that other symbolic variables are supported, such as in logic as parameters to built-in function blocks, user defined function blocks, parameterized function blocks, C blocks, bit-in-word references, and transition contacts and coils.

#### Restrictions on the Use of I/O Variables

- Since I/O variables are a form of symbolic variable, the same restrictions that apply to other symbolic variables of the same data type and array bounds apply to I/O variables.
- Only a global variable can become an I/O variable. A local variable cannot become an I/O variable.
- You can map only a discrete variable to a discrete terminal.
- You can map only a non-discrete variable to an analog terminal.
- Arrays and UDT variables must fit on the number of terminals in the reference address node counting from and including the terminal where you enter the array head or UDT variable. For example, if you have 32 analog terminals and you have a WORD array of 12 elements, you can map it to terminal 21 or any terminal before it (1 through 20).
- You can map a discrete array only to a terminal  $8n+1$ , where  $n = 0, 1, 2$ , and so on. The "+1" is included because the terminals are numbered beginning with 1. If you map it to a terminal other than  $8n+1$ , an error occurs upon validation.
- An I/O variable cannot be mapped to more than one location in hardware configuration.
- For the DO\_IO function block, if an I/O variable is assigned to the ST parameter, then the same I/O variable must also be assigned to the END parameter, and the entire module is scanned.

- Some I/O modules do not support the use of I/O variables. For a list of modules that support I/O variables, please refer to the *Important Product Information* for Logic Developer – PLC programming software.

### I/O Variable Format

To map an I/O variable, use the format **%vdr.s.[z.]g.t:**

**v** = I (input) or Q (output)

**d** = data type: X (discrete) or W (analog).

**r** = rack number

**s** = slot number

**[z]** = sub-slot number. This element and the period that follows it appear only if there is a sub-slot, for example, the SBA number of a Genius device. For an Ethernet daughterboard, set this value to 0.

**g** = segment number or number of the reference address node. Set to 0 for the first reference address node on the Terminals tab, 1 for the second reference node, and so on.

**t** = terminal number. One-based, that is, the numbering begins at 1.

### Supported I/O Variable Types

Data Type Mnemonic	Supported Data Types	Number of Consecutive Terminals Required
X	BOOL variable	1
	BOOL array	Number of elements in array.
	BYTE variable	8
	BYTE array	8n, where n is the number of array elements.
W	DINT variable	2
	DINT array	Number of elements in array times 2
	DWORD variable	2
	DWORD array	Number of elements in array times 2
	INT variable	1
	INT array	Number of elements in array
	LREAL variable	4
	LREAL array	Number of elements in array times 4
	REAL variable	2
	REAL array	Number of elements in array times 2
	UINT variable	1
	UINT array	Number of elements in array
	WORD variable	1
WORD array	Number of elements in array	

### I/O Variable Examples

 QW1                      Sample\_IO\_Variable                      %QW0.8.0.1

The I/O variable, Sample\_IO\_Variable is mapped to a non-discrete (W) output point (Q) on the module located in rack 0, slot 8. The variable is mapped to the first point in the first group of non-discrete output reference addresses.

 I2                              IO\_VAR\_EXAMPLE                      %IX0.5.2.2

The I/O variable, IO\_VAR\_EXAMPLE, is mapped to a discrete (X) input point (I) on the module located in rack 0, slot 5. The point is located in the module's third group of discrete input points and is point 2 in that group.

### 3.1.4 Arrays

An array is a complex data type composed of a series of variable elements with identical data types. Any variable can become an array, except for another array, a variable element, or a UDFB. In Machine Edition, you can create single-dimensional arrays and two-dimensional arrays.

In the controller CPU, each element of an array is treated as a separate variable with a separate, read-only reference address. The *root* node of the array variable also has a reference address that is editable. When you set or change the reference address of the *root* node of an array variable, the reference addresses of its elements are filled in with a range of addresses starting at that reference address and incremented for each element so as to create contiguous non-overlapping memory.

### 3.1.5 Variable Indexes and Arrays

PACSystems CPUs with firmware version 6.00 or later support variable indexes for arrays. With a variable index, when logic is executed, the value of the variable is evaluated and the corresponding array element is accessed.

**Note:** The numbering of array elements is zero-based.

For example, to access an element of the array named ABC, you could write ABC[DEF] in logic. When logic is executed, if the value of DEF is 5, then ABC[DEF] is equivalent to ABC[5], and the sixth element of array ABC is accessed.

If the value of the variable index exceeds the array boundary, a non-fatal fault is logged to the CPU fault table. In LD, the instruction for which this occurred does not pass power to the right.

#### Requirements and Support

An index variable must be of the INT, UINT, or DINT data type.

The valid range of values for an index variable is 0 through Y, where Y = [the number of array elements in the array] - 1. Refer to *Ensuring that a Variable Index does not Exceed the Upper Boundary of an Array*.

An index variable can be one of the following:

- Symbolic variable
- I/O variable
- Variable mapped to % memory areas such as %R
- Structure element
- Array element with a constant index
- Array element with a variable index
- Alias variable
- In the logic of a UDFB or parameterized block: formal parameter

The following support a variable index:

- Array elements of any data type except STRING
- Parameter array elements of any data type
- Alias variables

Dimensional support:

- One-dimensional (1D) formal parameter arrays in the logic of a UDFB or parameterized block
- 2D support for the top level of an array of structures and 1D support for a structure element that is an array. For example:

PQR[a, b].STRU[y].Zed,

where Zed is an element of the array of structures STRU, which itself is an element of the 2D array of structures PQR.

- 1D and 2D arrays for other variables

Other features:

- An array with a variable index supports a bit reference, for example  
MyArray[nIndex].X[4],  
where .X[4] is the fifth bit of the value stored in MyArray[nIndex]. The bit reference itself, [4] in the example, must be a constant.
- In LD, the following word-for-word changes are supported for array elements with variable indexes:

Replacing an index variable with another index variable

Replacing an index variable with a constant

Replacing a constant with an index variable

In LD, Diagnostic Logic Blocks support the use of array elements with variable indexes.

### Where Array Elements with Variable Indexes are Not Supported:

The following do not support array elements with variable indexes:

- Indirect references
- EGD variables
- Reference ID variables (RIVs) and I/O variables when accessed in the Hardware Configuration

**Note:** In logic, RIVs and I/O variables support variable indexes.

- STRING variables

A variable index cannot be one of the following:

- A math expression. For example, ABC[GH+1] is not supported.
- An indirect reference. For example, W[@XYZ] is not supported.
- A bit reference. For example, ABC[DEF.X[3]] is not supported.

**Note:** You can use a bit reference on an array element designated by a variable index. For example, ABC[DEF].X[3] is supported.

- An array head. For example, if MNP and QRS are arrays, MNP[QRS] is not supported, but MNP[QRS[3]] and MNP[QRS[TUV]] are, where TUV is an index variable.
- A negative index. This generates a run-time non-fatal CPU fault.
- A value greater than Y, where Y = [number of array elements] - 1. This generates a run-time non-fatal CPU fault.

## Ensuring that a Variable Index does not Exceed the Upper Boundary of an Array

### One-Dimensional Array

1. Once per scan, execute ARRAY\_SIZE\_DIM1 to count the number of elements in the array.

**Note:** The array size of a variable can be changed in a RUN Mode Store but it will not be changed while logic is executing.

ARRAY\_SIZE\_DIM1 places the count value in the variable associated with its output Q.

2. Before executing an instruction that uses a variable index, compare the value of the index variable with the number of elements in the array.

**Tip:** In LD, use a RANGE instruction.

**Notes** Checking before executing each instruction that uses an indexed variable is recommended in case logic has modified the index value beyond the array size or in case the array size has been reduced before the scan to less than the value of an index variable that has not been reduced accordingly since.

Valid range of an index variable: 0 through (n-1), where n is the number of array elements.  
Array indexes are zero-based.

### Two-Dimensional Array

- Execute both ARRAY\_SIZE\_DIM1 and ARRAY\_SIZE\_DIM2 to count the number of elements in respectively the first and second dimensions of the array.

## 3.2 Reference Memory

The CPU stores program data in bit memory and word memory. Both types of memory are divided into different types with specific characteristics. By convention, each type is normally used for a specific type of data, as explained below. However, there is great flexibility in actual memory assignment.

Memory locations are indexed using alphanumeric identifiers called references. The reference's letter prefix identifies the memory area. The numerical value is the offset within that memory area, for example %AQ0056.

### 3.2.1 Word (Register) References

Type	Description
%AI	The prefix %AI represents an analog input register. An analog input register holds the value of one analog input or other non-discrete value.
%AQ	The prefix %AQ represents an analog output register. An analog output register holds the value of one analog output or other non-discrete value.
%R	Use the prefix %R to assign system register references that will store program data such as the results of calculations.
%W	Retentive Bulk Memory Area, which is referenced as %W (WORD memory).
%P	Use the prefix %P to assign program register references that will store program data with the _MAIN block. This data can be accessed from all program blocks. The size of the %P data block is based on the highest %P reference in all blocks. %P addresses are available only to the LD program they are used in, including C blocks called from LD blocks; they are not system-wide.

**Note:** All register references are retained across a power cycle to the CPU.

### Indirect References

An indirect reference allows you to treat the contents of a variable assigned to an LD instruction operand as a pointer to other data, rather than as actual data. Indirect references are used only with word memory areas (%R, %W, %AI, %AQ, %P, and %L). An indirect reference in %W requires two %W locations as a DWORD indirect index value. For example, @%W0001 would use the %W2:W1 as a DWORD index into the %W memory range. The DWORD index is required because the %W size is greater than 65K.

Indirect references cannot be used with symbolic variables.

To assign an indirect reference, type the @ character followed by a valid reference address or variable name. For example, if %R00101 contains the value 1000, @R00101 instructs the CPU to use the data location of %R01000.

Indirect references can be useful when you want to perform the same operation to many word registers. Use of indirect references can also be used to avoid repetitious logic within the application program. They can be used in loop situations where each register is incremented by a constant or by a value specified until a maximum is reached.

### Bit in Word References

Bit in word referencing allows you to specify individual bits in a word reference type as inputs and outputs of Boolean expressions, functions, and calls that accept bit parameters (such as parameterized blocks). This feature is restricted to word references in retentive memory. The bit number in the bit within word construct must be a constant.

You can use the programmer or an HMI to set an individual bit on or off within a word, or monitor a bit within a word. Also, C blocks can read, modify, and write a bit within a word.

Bit in Word references can be used in the following situations:

- In retentive 16-bit memory (AI, AQ, R, W, P, and L) and symbolics.
- On all contacts and coils **except** legacy transition contacts (POSCON/NEGCON) and transition coils (POSCOIL/NEGCOIL).
- On all functions and call parameters that accept single or unaligned bit parameters.

Functions that accept unaligned discrete references	Parameters
ARRAY MOVE (BIT)	SR and DS
ARRAY RANGE (BIT)	Q
MOVE (BIT)	IN and Q
SHFR (BIT)	IN, ST and Q

### Restrictions

The use of Bit in Word references has the following restrictions:

- Bit in Word references cannot be used on legacy transition contacts (POSCON/NEGCON) and transition coils (POSCON/NEGCON).
- The bit number (index) must be a constant; it cannot be a variable.
- Bit addressing is not supported for a constant.
- Indirect references cannot be used to address bits in 16-bit memory.
- You cannot force a bit within 16-bit memory.

### Examples:

%R2.X [0] addresses the first (least significant) bit of %R2

%R2.X [1] addresses the second bit of %R2. In the examples

In the examples [0] and [1] are the bit indexes. Valid bit indexes for the different variable types are:

BYTE variable	[0] through [7]
WORD, INT, or UINT variable	[0] through [15]
DWORD or DINT variable	[0] through [31]

### 3.2.2 Bit (Discrete) References

Type	Description
<b>%I</b>	Represents input references. %I references are located in the input status table, which stores the state of all inputs received from input modules during the last input scan. A reference address is assigned to discrete input modules using your programming software. Until a reference address is assigned, no data will be received from the module. %I memory is always retentive.
<b>%Q</b>	Represents physical output references. The coil check function checks for multiple uses of %Q references with relay coils or outputs on functions. You can select the level of coil checking desired (Single, Warn Multiple, or Multiple).  %Q references are located in the output status table, which stores the state of the output references as last set by the application program. This output status table's values are sent to output modules at the end of the program scan. A reference address is assigned to discrete output modules using your programming software. Until a reference address is assigned, no data is sent to the module. A particular %Q reference may be either retentive or non-retentive.
<b>%M</b>	Represents internal references. The coil check function of your programming software checks for multiple uses of %M references with relay coils or outputs on functions. A particular %M reference may be either retentive or non-retentive.
<b>%T</b>	Represents temporary references. These references are never checked for multiple coil use and can, therefore, be used many times in the same program even when coil use checking is enabled—this is not a recommended practice because it makes subsequent trouble-shooting more difficult. %T may be used to prevent coil use conflicts while using the cut/paste and file write/include functions. Because this memory is intended for temporary use, it is cleared on STOP Mode to RUN Mode transitions and cannot be used with retentive coils.
<b>%S</b> <b>%SA</b> <b>%SB</b> <b>%SC</b>	Represent system status references. These references are used to access special CPU data such as timers, scan information, and fault information. For example, the %SC0012 bit can be used to check the status of the CPU fault table. Once the bit is set on by an error, it will not be reset until after the sweep. %S, %SA, %SB, and %SC can be used on any contacts. <ul style="list-style-type: none"> <li>▪ %SA, %SB, and %SC can be used on retentive coils -(M)-.</li> </ul> <p><b>Note:</b> Although the programming software forces the logic to use retentive coils with %SA, %SB, and %SC references, most of these references are not preserved across power cycles regardless of the state of the battery or Energy Pack.</p> <p>%S can be used as word or bit-string input arguments to functions or function blocks. %SA, %SB, and %SC can be used as word or bit-string input or output arguments to functions and function blocks. For a description of the behavior of each bit, refer to <i>System Status References</i>.</p>
<b>%G</b>	Represents global data references. These references are used to access data shared among several control systems.

**Note:** For details on retentiveness, refer to *Retentiveness of Logic and Data*.

### 3.3 User Reference Size and Default

Maximum user references and default reference sizes are listed in the table below.

Item	Range	Default
<b>Reference Points</b>		
%I reference	32768 bits	32768 bits
%Q reference	32768 bits	32768 bits
%M reference	32768 bits	32768 bits
%S total (S, SA, SB, SC)	512 bits (128 each)	512 bits (128 each)
%T reference	1024 bits	1024 bits
%G	7680 points	7680 points
Total Reference Points	107520	107520
<b>Reference Words</b>		
%AI reference	0–32640 words	64 words
%AQ reference	0–32640 words	64 words
%R, 1K word increments	0–32640 words	1024 words
%W	0—maximum available user RAM	0 words
Total Reference Words	0—maximum available user RAM	1152 words
%L (per block)	8192 words	8192 words
%P (per program)	8192 words	8192 words
<b>Managed Memory</b>		
Symbolic Discrete	0–83,886,080 (bits)	32768
Symbolic Non-Discrete	0–5,242,880 (words)	65536
I/O Discrete	0 through 83,886,080	0
I/O Non-Discrete	0 through 5,242,880.	0
Total Symbolic	0–42,088,704 bytes (This is the total memory available for the combined total of symbolic memory. This also includes other user memory use, program etc.)	143360

#### 3.3.1 %G User References and CPU Memory Locations

The CPU contains one data space for all of the global data references (%G). The internal CPU memory for this data is 7680 bits long. For Series 90-70 systems, the programming software subdivides this range using %G, %GA, %GB, %GC, %GD, and %GE prefixes—allowing each of these prefixes to be used with bit offsets in the range 1–1280. For PACSystems, these ranges are converted to %G.

### 3.4 Genius Global Data

PACSystems supports the sharing of data among multiple control systems that share a common Genius I/O bus. This mechanism provides a means for the automatic and repeated transfer of %G, %I, %Q, %AI, %AQ, and %R data. No special application programming is required to use global data since it is integrated into the I/O scan. All devices that have Genius I/O capability can send and receive global data from a PACSystems CPU.

Using *I/O Variables*, you can directly associate variable names to a module's Genius global data that is scanned as part of an input/output scan.

### 3.5 Transitions and Overrides

The %I, %Q, %M, and %G user references, and symbolic variables of type BOOL, have associated transition and override bits. %T, %S, %SA, %SB, and %SC references have transition bits but not override bits. The CPU uses transition bits for counters, transition contacts, and transitional coils. Note that counters do not use the same kind of transition bits as contacts and coils. Transition bits for counters are stored within the locating reference.

The transition bit for a reference tells whether the most recent value (ON, OFF) written to the reference is the same as the previous value of the reference. Therefore when a reference is written and its new value is the same as its previous value, its transition bit is turned OFF. When its new value is different from its previous value, its transition bit is turned ON. The transition bit for a reference is affected every time the reference is written to. The source of the write is immaterial; it can result from a coil execution, an executed function's output, the updating of reference memory after an input scan, etc.

When override bits are set, the associated references cannot be changed from the program or the input device; they can only be changed on command from the programmer. Overrides do not protect transition bits. If an attempted write occurs to an overridden memory location, the corresponding transition bit is cleared.

### 3.6 Retentiveness of Logic and Data

Data is defined as retentive if it is saved by the CPU when the CPU transitions from STOP Mode to RUN Mode.

The following items are retentive:

- program logic
- fault tables and diagnostics
- checksums for program logic
- overrides and output forces
- word data (%R, %W, %L, %P, %AI, %AQ)
- bit data (%I, %G, fault locating references, and reserved bits)
- %Q and %M variables that are configured as retentive (%T data is non-retentive and therefore not saved on STOP Mode to RUN Mode transitions).
- symbolic variables that have a data type other than BOOL
- symbolic variables of BOOL type that are configured as retentive
- Retentive data is also preserved during power-cycles of the CPU with battery backup or Energy Pack backup. Exceptions to this rule include the fault locating references and most of the %S, %SA, %SB, and %SC references. These references are initialized to zero at power-up regardless of the state of the battery or Energy Pack. (For a description of the behavior of each, refer to *System Status References*.)

When %Q or %M variables are configured as retentive, the contents are retained through power loss and Run-to-Stop-to-Run transitions.

### 3.7 Data Scope

Each of the user references has *scope*; that is, it may be available throughout the system, available to all programs, restricted to a single program, or restricted to local use within a block.

User Reference Type	Range	Scope
%I, %Q, %M, %T, %S, %SA, %SB, %SC, %G, %R, %W, %AI, %AQ, convenience references, fault locating references	Global	From any program, block, or host computer. Variables defined in these registers have system (global) scope by default. However, variables with local scope can also be assigned in these registers.
Symbolic variable	Global	From any program, block, or host computer. Symbolic variables have system (global) scope by default. However, symbolic variables with local scope can be created using the naming conventions for local variables.
I/O variable	Global	From any program, block, or host computer.
%P	Program	From any block, but not from other programs (also available to a host computer).
%L	Local	From within a block only (also available to a host computer).

In an LD block:

- %P should be used for program references that are shared with other blocks.
- %L are local references that can be used to restrict the use of register data to that block. These local references are not available to other parts of the program.
- %I, %Q, %M, %T, %S, %SA, %SB, %SC, %G, %R, %W, %AI, and %AQ references are available throughout the system.

## 3.8 System Status References

System status references in the CPU are assigned to %S, %SA, %SB, and %SC memory. The four timed contacts (time tick references) include #T\_10MS, #T\_100MS, #T\_SEC, and #T\_MIN. Examples of other system status references include #FST\_SCN, #ALW\_ON, and #ALW\_OFF

**Note:** %S bits are read-only bits; do not write to these bits. However, you can write to %SA, %SB, and %SC bits.

Listed below are available system status references that can be used in an application program. When entering logic, either the reference or the nickname can be used. Refer to Chapter 9 for detailed fault descriptions and information on correcting faults.

### 3.8.1 %S References

Reference	System Variable	Definition
%S0001	#FST_SCN	Current sweep is the first sweep in which the LD executed. Set the first time the user program is executed after a STOP Mode to RUN Mode transition and cleared upon completion of its execution.
%S0002	#LST_SCN	Set when the CPU transitions to RUN Mode; cleared when the CPU is performing its final sweep. The CPU clears this bit and then performs one more complete sweep before transitioning to STOP or STOP Faulted mode. If the number of last scans set to 0, %S0002 will be cleared after the CPU is stopped and user logic will not see this bit cleared.
%S0003	#T_10MS	0.01 second timed contact.
%S0004	#T_100MS	0.1 second timed contact.
%S0005	#T_SEC	1.0 second timed contact.
%S0006	#T_MIN	1.0 minute timed contact.
%S0007	#ALW_ON	Always ON.
%S0008	#ALW_OFF	Always OFF.
%S0009	#SY_FULL	Set when the CPU fault table fills up (size configurable with a default of 16 entries). Cleared when an entry is removed from the CPU fault table and when the CPU fault table is cleared.
%S0010	#IO_FULL	Set when the I/O Fault Table fills up (size configurable with a default of 32 entries). Cleared when an entry is removed from the I/O Fault Table and when the I/O Fault Table is cleared.
%S0011	#OVR_PRE	Set when an override exists in %I, %Q, %M, or %G, or symbolic BOOL memory.
%S0012	#FRC_PRE	Set when force exists on a Genius point.
%S0013	#PRG_CHK	Set when background program check is active.
%S0014	#PLC_BAT	CPUs with batteries, including CPU310, CPU315, CPU/CRU320 and NIU001 <ul style="list-style-type: none"> <li>▪ If the battery is disconnected, this contact is set to 1.</li> <li>▪ Whenever a Smart Battery fails during operation, this contact is set to 1. If used in conjunction with a legacy (non-smart) battery, this indication is not reliable.</li> </ul> Battery-less CPUs, including CPE302, CPE305, CPE310 and CPE330: <ul style="list-style-type: none"> <li>▪ Energy Pack is connected and functioning = 0</li> <li>▪ Energy Pack is not connected or has failed = 1</li> </ul>
%S0033	#PRI_UNT	Set to 1 if the local unit is configured as the Primary CPU: otherwise it is cleared. For any given local unit, if PRI_UNT is set, SEC_UNT cannot be set.
%S0034	#SEC_UNT	Set to 1 if the local unit is configured as the Secondary CPU: otherwise it is cleared. For any given local unit, if SEC_UNT is set, PRI_UNT cannot be set.
%S0035	#LOC_RDY	Set to 1 if local unit is in Run mode with outputs enabled. Otherwise set to 0.
%S0036	#LOC_ACT	Set to 1 if local unit is currently the Active unit; otherwise it is cleared. For any given local unit, if LOC_ACT is set, REM_ACT cannot be set.
%S0037	#REM_RDY	Set to 1 if remote unit is in Run mode with outputs enabled. Otherwise set to 0.
%S0038	#REM_ACT	Set to 1 if remote unit is currently the Active unit; otherwise it is cleared. For any given local unit, if REM_ACT is set, LOC_ACT cannot be set.
%S0039	#LOGICEQ	Set to 1 if the application logic for both units in the redundant system is the same. Otherwise set to 0.
%S0041	#RDN_COMM_AVAIL	Redundancy Communication Link Available: 1 indicates that the two CPUs can communicate with each other and will be able to synchronize when required.
%S0042	#RDN_P1_LINK_UP	Redundancy Ethernet Port 1 on LAN3 has link on its PHY.
%S0043	#RDN_P2_LINK_UP	Redundancy Ethernet Port 2 on LAN3 has link on its PHY.
%S0049	#FA_OK	Field Agent OK: 1 indicates Field Agent running and connected to cloud.

**Note:** The #FST\_EXE name is not associated with a %S address, it must be referenced by the name #FST\_EXE only. This bit is set when transitioning from STOP Mode to RUN Mode and indicates that the current sweep is the first time this block has been called.

### 3.8.2 %SA, %SB, and %SC References

**Note:** %SA, %SB, and %SC contacts are not set or reset until the input scan phase of the sweep following the occurrence of the fault or a clearing of the fault table(s). %SA, %SB, and %SC contacts can also be set or reset by user logic and CPU monitoring devices.

Reference	System Variable	Definition
%SA0001	#PB_SUM	Set when a checksum calculated on the application program does not match the reference checksum. If the fault was due to a temporary failure, the condition can be cleared by again storing the program to the CPU. If the fault was due to a hard RAM failure, then the CPU must be replaced. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0002	#OV_SWP	Set when the CPU detects that the previous sweep took longer than the time specified by the user. To clear this bit, clear the CPU fault table or power cycle the CPU. Only occurs if the CPU is in Constant Sweep mode.
%SA0003	#APL_FLT	Set when an application fault occurs. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0009	#CFG_MM	Set when a configuration mismatch fault is logged in the fault tables. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0008	#OVR_TMP	Set when the operating temperature of the CPU exceeds the normal operating temperature, 58°C. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0010	#HRD_CPU	Set when the diagnostics detects a problem with the CPU hardware. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0011	#LOW_BAT	The low battery indication is <b>not</b> supported for all CPU modules. For details, refer to <i>Battery Status (Group 18)</i> in Chapter 9. The CPU may set this contact when an I/O module or special-purpose module has reported a low battery. In this case, a fault will be reported in the I/O Fault Table. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0012	#LOS_RCK	Set when an expansion rack stops communicating with the CPU. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0013	#LOS_IOC	Set when a Bus Controller stops communicating with the CPU. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0014	#LOS_IOM	Set when an I/O module stops communicating with the CPU. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0015	#LOS_SIO	Set when an option module stops communicating with the CPU. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0017	#ADD_RCK	Set when an expansion rack is added to the system. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SA0018	#ADD_IOC	Set when a Bus Controller is added to a rack. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0019	#ADD_IOM	Set when an I/O module is added to a rack. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0020	#ADD_SIO	Set when an intelligent option module is added to a rack. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0022	#IOC_FLT	Set when a Bus Controller reports a bus fault, a global memory fault, or an IOC hardware fault. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0023	#IOM_FLT	Set when an I/O module reports a circuit or module fault. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0027	#HRD_SIO	Set when a hardware failure is detected in an option module. To clear this bit, clear the I/O Fault Table or power cycle the CPU.

Reference	System Variable	Definition
%SA0029	#SFT_IOC	Set when there is a software failure in the I/O Controller. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0030	#PNIO_ALARM	A PROFINET alarm has been received and an I/O fault has been logged in group 28. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0031	#SFT_SIO	Set when an option module detects an internal software error. To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0032	#SBUS_ER	Set when a bus error occurs on the VME bus backplane To clear this bit, clear the I/O Fault Table or power cycle the CPU.
%SA0081 – %SA0112		Set when a user-defined fault is logged in the CPU fault table. To clear these bits, clear the CPU fault table or power cycle the CPU. For more information, see discussion of SVC_REQ 21: User-Defined Fault Logging in Chapter 7.
%SB0001	#WIND_ER	Set when there is not enough time to start the Programmer Window in Constant Sweep mode. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SB0009	#NO_PROG	Set when the CPU powers up with memory preserved, but no user program is present. Cleared when the CPU powers up with a program present or by clearing the CPU fault table.
%SB0010	#BAD_RAM	Set when the CPU detects corrupted RAM memory at power-up. Cleared when the CPU detects that RAM memory is valid at power-up or by clearing the CPU fault table.
%SB0011	#BAD_PWD	Set when a password access violation occurs. Cleared when the CPU fault table is cleared or when the CPU is power cycled.
%SB0012	#NUL_CFG	Set when an attempt is made to put the CPU in RUN Mode when there is no configuration data present. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SB0013	#SFT_CPU	Set when the CPU detects an error in the CPU operating system software. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SB0014	#STOR_ER	Set when an error occurs during a programmer store operation. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SB0016	#MAX_IOC	Set when more than 32 IOCs are configured for the system. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SB0017	#SBUS_FL	Set when the CPU fails to gain access to the bus. To clear this bit, clear the CPU fault table or power cycle the CPU.
%SC0009	#ANY_FLT	Set when any fault occurs that causes an entry to be placed in the CPU or I/O Fault Table. Cleared when both fault tables are cleared or when the CPU is power cycled.
%SC0010	#SY_FLT	Set when any fault occurs that causes an entry to be placed in the CPU fault table. Cleared when the CPU fault table is cleared or when the CPU is power cycled.
%SC0011	#IO_FLT	Set when any fault occurs that causes an entry to be placed in the I/O Fault Table. Cleared when the I/O Fault Table is cleared or when the CPU is power cycled.
%SC0012	#SY_PRES	Set as long as there is at least one entry in the CPU fault table. Cleared when the CPU fault table is cleared.
%SC0013	#IO_PRES	Set as long as there is at least one entry in the I/O Fault Table. Cleared when the I/O Fault Table is cleared.
%SC0014	#HRD_FLT	Set when a hardware fault occurs. Cleared when both fault tables are cleared or when the CPU is power cycled.
%SC0015	#SFT_FLT	Set when a software fault occurs. Cleared when both fault tables are cleared or when the CPU is power cycled.

### 3.8.3 Fault References

The fault references are discussed in Chapter 9 of this manual but are also listed here for your convenience.

#### System Fault References

System Fault Ref	Description
#ANY_FLT	Any new fault in either table since the last power-up or clearing of the fault tables
#SY_FLT	Any new system fault in the CPU fault table since the last power-up or clearing of the fault tables
#IO_FLT	Any new fault in the I/O Fault Table since the last power-up or clearing of fault tables
#SY_PRESENCE	Indicates that there is at least one entry in the CPU fault table
#IO_PRESENCE	Indicates that there is at least one entry in the I/O Fault Table
#HRD_FLT	Any hardware fault
#SFT_FLT	Any software fault

#### Configurable Fault References

Configurable Faults (Default Action)	Description
#SBUS_ER (diagnostic)	System bus error. (The BSERR signal was generated on the VME system bus.)
#SFT_IOC (diagnostic)	Non-recoverable software error in a Genius Bus Controller.
#LOS_RCK (diagnostic)	Loss of rack (BRM failure, loss of power) or missing a configured rack.
#LOS_IOC (diagnostic)	Loss of Bus Controller missing a configured Bus Controller.
#LOS_IOM (diagnostic)	Loss of I/O module (does not respond) or missing a configured I/O module.
#LOS_SIO (diagnostic)	Loss of intelligent option module (does not respond) or missing a configured module.
#IOC_FLT (diagnostic)	Non-fatal bus or Bus Controller error—more than 10 bus errors in 10 seconds (error rate is configurable).
#CFG_MM (fatal)	Wrong module type detected during power-up, store of configuration, or RUN Mode. The CPU does not check the configuration parameters set up for individual modules such as Genius I/O blocks.

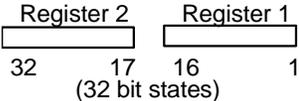
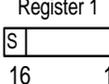
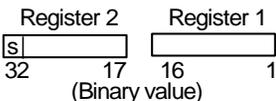
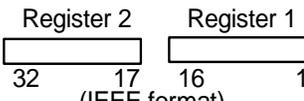
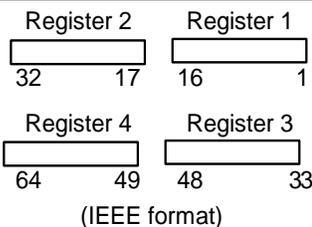
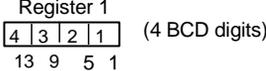
## Non-Configurable Faults

Non-Configurable Faults (Action)	Description
#SBUS_FL (fatal)	System bus failure. The CPU was not able to access the VME bus. BUSGRT-NMI error.
#HRD_CPU (fatal)	CPU hardware fault, such as failed memory device or failed serial port.
#HRD_SIO (diagnostic)	Non-fatal hardware fault on any module in the system.
#SFT_SIO (diagnostic)	Non-recoverable software error in a LAN interface module.
#PB_SUM (fatal)	Program or block checksum failure during power-up or in RUN Mode.
#LOW_BAT (diagnostic)	The low battery indication is not supported for all CPU modules. For details, refer to <i>Battery Status (Group 18)</i> in Chapter 9. The CPU may set this contact when an I/O module or special-purpose module has reported a low battery. In this case, a fault will be reported in the I/O Fault Table. To clear this bit, clear the CPU fault table or power cycle the CPU.
#OV_SWP (diagnostic)	Constant sweep time exceeded.
#SY_FULL, IO_FULL (diagnostic)	CPU fault table full I/O Fault Table full
#IOM_FLT (diagnostic)	Point or channel on an I/O module—a partial failure of the module.
#APL_FLT (diagnostic)	Application fault.
#ADD_RCK (diagnostic)	New rack added, extra, or previously faulted rack has returned.
#ADD_IOC (diagnostic)	Extra I/O Bus Controller or reset of I/O Bus Controller.
#ADD_IOM (diagnostic)	Previously faulted I/O module is no longer faulted or extra I/O module.
#ADD_SIO (diagnostic)	New intelligent option module is added, extra, or reset.
#NO_PROG (information)	No application program is present at power-up. Should only occur the first time the CPU is powered up or if the user memory is not retained.
#BAD_RAM (fatal)	Corrupted program memory at power-up. Program could not be read and/or did not pass checksum tests.
#WIND_ER (information)	Window completion error. Servicing of Programmer or Logic Window was skipped. Occurs in Constant Sweep mode.
#BAD_PWD (information)	Change of privilege level request to a protection level was denied; bad password.
#NUL_CFG (fatal)	No configuration present upon transition to RUN Mode. Running without a configuration is similar to suspending the I/O scans.
#SFT_CPU (fatal)	CPU software fault. A non-recoverable error has been detected in the CPU. May be caused by Watchdog Timer expiring.
#MAX_IOC (fatal)	The maximum number of bus controllers has been exceeded. The CPU supports 32 bus controllers.
#STOR_ER (fatal)	Download of data to CPU from the programmer failed; some data in CPU may be corrupted.

### 3.9 How Program Functions Handle Numerical Data

Regardless of where data is stored in memory – in one of the bit memories or one of the word memories – the application program can handle it as different data types.

#### 3.9.1 Data Types

Type	Name	Description	Data Format
BOOL	Boolean	The smallest unit of memory. It has two states: 1 or 0. A BOOL array may have length N.	
BYTE	Byte	Has an 8-bit value. Has 256 values (0–255). A BYTE array may have length N.	
WORD	Word	Uses 16 consecutive bits of data memory. The valid range of word values is 0000 hex to FFFF hex.	 (16 bit states) 16      1
DWORD	Double Word	Has the same characteristics as a single word data type, except that it uses 32 consecutive bits in data memory instead of only 16 bits.	 (32 bit states) 32      17      16      1
UINT	Unsigned Integer	Uses 16-bit memory data locations. They have a valid range of 0 to +65535 (FFFF hex).	 (Binary value) 16      1
INT	Signed Integer	Uses 16-bit memory data locations, and are represented in 2's complement notation. The valid range of an INT data type is –32768 to +32767.	 (Two's Complement value) 16      1 s=sign bit (0=positive, 1=negative)
DINT	Double Precision Integer	Stored in 32-bit data memory locations (two consecutive 16-bit memory locations). Always signed values (bit 32 is the sign bit). The valid range of a DINT data type is –2147483648 to +2147483647	 (Binary value) 32      17      16      1 s=sign bit (0=positive, 1=negative)
REAL	Floating Point	Uses 32 consecutive bits (two consecutive 16-bit memory locations). The range of numbers that can be stored in this format is from ±1.401298E-45 to ±3.402823E+38. For the IEEE format, refer to <i>Floating Point Numbers</i> .	 (IEEE format) 32      17      16      1
LREAL	Double Precision Floating Point	Uses 64 consecutive bits (four consecutive 16-bit memory locations). The range of numbers that can be stored in this format is from ±2.2250738585072020E-308 to ±1.7976931348623157E+308. For the IEEE format, refer to <i>Floating Point Numbers</i> .	 (IEEE format) 32      17      16      1 64      49      48      33
BCD-4	Four-Digit BCD	Uses 16-bit data memory locations. Each binary coded decimal (BCD) digit uses four bits and can represent numbers between 0 and 9. This BCD coding of the 16 bits has a legal value range of 0 to 9999.	 (4 BCD digits) 4    3    2    1 13   9   5   1

Type	Name	Description	Data Format								
BCD-8	Eight-Digit BCD	Uses two consecutive 16-bit data memory locations (32 consecutive bits). Each BCD digit uses 4 bits per digit to represent numbers from 0 to 9. The complete valid range of the 8-digit BCD data type is 0 to 99999999.	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">           Register 2  <table border="1" style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="padding: 2px;">8</td><td style="padding: 2px;">7</td><td style="padding: 2px;">6</td><td style="padding: 2px;">5</td> </tr> </table> </div> <div style="text-align: center;">           Register 1  <table border="1" style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="padding: 2px;">4</td><td style="padding: 2px;">3</td><td style="padding: 2px;">8</td><td style="padding: 2px;">1</td> </tr> </table> </div> </div> <p style="text-align: center; margin-top: 5px;">32 29 25 21 17 16 13 9 5 1 (8 BCD digits)</p>	8	7	6	5	4	3	8	1
8	7	6	5								
4	3	8	1								
MIXED	Mixed	Available only with the MUL and DIV functions. The MUL function takes two integer inputs and produces a double integer result. The DIV function takes a double integer dividend and an integer divisor to product an integer result.	<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;">16 □</div> <div style="text-align: center;">16 □</div> <div style="font-size: 2em;">=</div> <div style="text-align: center;">32 □</div> </div> <div style="display: flex; justify-content: space-around; align-items: center; margin-top: 5px;"> <div style="text-align: center;">32 □</div> <div style="text-align: center;">16 □</div> <div style="font-size: 2em;">=</div> <div style="text-align: center;">16 □</div> </div>								
ASCII	ASCII	Eight-bit encoded characters. A single word reference is required to make two (packed) ASCII characters. The first character of the pair corresponds to the low byte of the reference word. The remaining 7 bits in each section are converted.									

**Note:** Using functions that are not explicitly bit-typed will affect transitions for all bits in the written byte/word/dword. For information about using floating point numbers, refer to *Floating Point Numbers*.

### 3.9.2 Floating Point Numbers

Floating point numbers are stored in one of two IEEE 754 standard formats that uses adjacent 16-bit words: 32-bit single precision or 64-bit double precision.

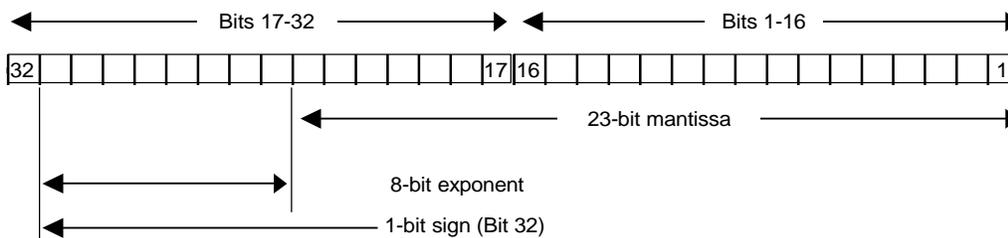
The REAL data type represents single precision floating point numbers. The LREAL data type represents double precision floating point numbers. REAL and LREAL variables are typically used to store data from analog I/O devices, calculated values, and constants.

#### Types of Floating Point Variables

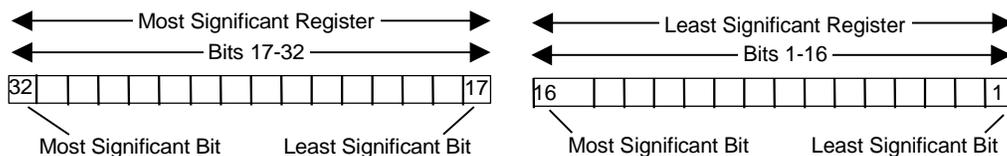
Data Type	Precision and Range
REAL	Limited to 6 or 7 significant digits, with a range of approximately $\pm 1.401298 \times 10^{-45}$ through $\pm 3.402823 \times 10^{38}$ .
LREAL	Limited to 17 significant digits, with a range of approximately $\pm 2.2250738585072020 \times 10^{-308}$ to $\pm 1.7976931348623157 \times 10^{308}$ .

**Note:** The programming software allows 32-bit and 64-bit arguments (DWORD, DINT, REAL, and LREAL) to be placed in discrete memories such as %I, %M, and %R in the PACSystems target. This is not allowed on Series 90-70 targets. (Note that any bit reference address that is passed to a non-bit parameter must be byte-aligned. This is the same as the Series 90-70 CPU.)

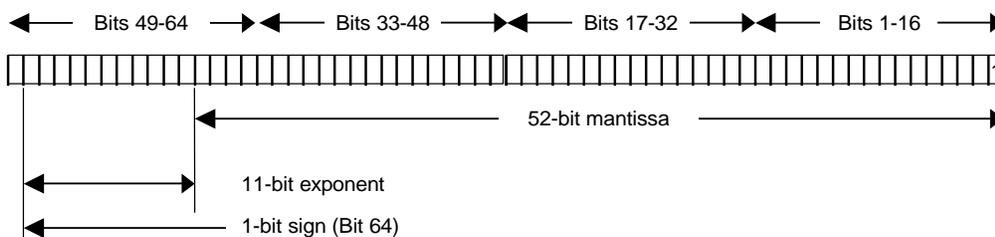
#### Internal Format of REAL Numbers



Register use by a single floating point number is diagrammed below. For example, if the floating point number occupies registers R5 and R6, R5 is the least significant register and R6 is the most significant register.



#### Internal Format of LREAL Numbers



## Errors in Floating Point Numbers and Operations

Overflow occurs when a REAL or LREAL function generates a number outside the allowed range. When this occurs, the Enable Out output of the function is set Off, and the result is set to positive infinity (for a number greater than the upper limit) or negative infinity (for a number less than the lower limit). You can determine where this occurs by testing the sense of the Enable Out output.

Binary representations of Infinity and NaN values have exponents that contain all 1s.

### IEEE 754 Infinity Representations

	REAL	LREAL
POS_INF (positive infinity)	= 7F800000h	= 7FF0000000000000h
NEG_INF (negative infinity)	= FF800000h	= 7FF0000000000001h

If the infinities produced by overflow are used as operands to other REAL or LREAL functions, they may cause an undefined result. This undefined result is referred to as a NaN (Not a Number). For example, the result of adding positive infinity to negative infinity is undefined. When the ADD\_REAL function is invoked with positive infinity and negative infinity as its operands, it produces a NaN. If any operand of a function is a NaN, the result will be some NaN.

**Note:** For NaN, the Enable Out output is Off (not energized).

### IEEE 754 Representations of NaN values:

REAL	LREAL
7F800001 through 7FFFFFFF	7FF8000000000001 through 7FFFFFFFFFFFFFFF
FF800001 through FFFFFFFF	FFF0000000000001 through FFFFFFFFFFFFFFFF

**Note:** For releases 5.0 and greater, the CPU may return slightly different values for NaN compared to previous releases. In some cases, the result is a special type of NaN displayed as #IND in Machine Edition. In these cases, for example, EXP(-infinity), power flow out of the function is identical to that in previous releases.

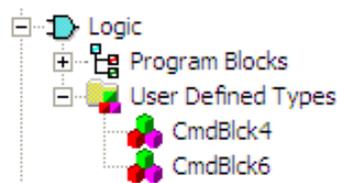
### 3.10 User Defined Types (UDTs)

A UDT is a structured data type consisting of elements of other selected data types. Each top-level UDT element can be one of the following:

Top-level UDT Element	Example
Simple data type, except STRING	INT
Another UDT, except any in which the current UDT is nested at any level. <b>Note:</b> A UDT cannot be nested within itself.	A UDT named UDT_ABC has a top-level element whose data type is another UDT, named UDT_2.
Array of a simple data type	LREAL array of length 8.
Array of UDTs <b>Note:</b> A UDT cannot be nested within itself.	A UDT named UDT_ABC has a top-level element that is an array whose data type is another UDT, named UDT_row.

#### 3.10.1 Working with UDTs

1. In Machine Edition, add a UDT as a node under a target in the Project tab of the Navigator. A UDT will be saved with the target in which it is used.
2. Edit the UDT properties and define the elements in the UDT's structure.
3. Create a variable whose data type is the UDT. By default, the variable resides in symbolic memory. You can convert the symbolic variable to an I/O variable by assigning it to an I/O terminal.
4. Use the variable in logic.



#### 3.10.2 UDT Properties

**Name:** The UDT's name. Maximum length: 32 characters.

**Description:** The user-defined description of the UDT.

**Memory Type:** The type of symbolic or I/O variable memory in which a variable of this UDT resides.

**Non-Discrete:** (Default) Word-oriented memory organized in groups of 16 contiguous bits.

**Discrete:** Bit-oriented memory.

**Notes:** You cannot nest a UDT of one memory type in a UDT of a different memory type. Changing the memory type propagates to existing variables of this UDT only after target validation.

**Is Fixed Size:** If set to True, you can increase the Size (Bytes) value to a maximum of 65,535 bytes to create a buffer at the end of the UDT. The buffer is included in the memory allocated to every downloaded variable of that UDT data type. Use of a buffer may allow RUN Mode store of a UDT when the size of the UDT definition has changed. For details, refer to *RUN Mode Store of UDTs*.

If set to False (default), the Size (Bytes) value is read-only and does not include a buffer at the end of the UDT.

**Size (bytes):** (Read-only when Is Fixed Size is set to False.) The total number of bytes required to store a structure variable of the user-defined data type (UDT).

**Bytes Remaining:** (Read-only; displayed if Is Fixed Size is set to True.) The UDT's buffer size; the number of bytes available before the actual size of the UDT reaches the value of the Size (bytes) property.

### 3.10.3 UDT Limits

- Maximum number of UDTs per target: 2048
- Maximum UDT size: 65,535 bytes

**Note:** Bit spares created to line up the end of a section of BOOL variables or arrays with the end of a byte will count toward the maximum size.

- Maximum number of top-level UDT elements: 1024
- Maximum array size of a top-level UDT element: 1024 array elements
- UDTs **do not** support the following:
  - Two-dimensional arrays
  - Function block data types
  - Enumerated data types
- You cannot nest a UDT of one memory type in a UDT of a different memory type.
- You cannot alias a variable to a UDT variable or UDT variable element.
- A FAULT contact supports a BOOL element of a UDT I/O variable, but not a BOOL element of a UDT parameter in a UDFB or parameterized block.
- POSCON and NEGCON do not support BOOL elements of UDT parameters in parameterized blocks or UDFBs.

### 3.10.4 RUN Mode Store of UDTs

An RMS can be performed on a target that contains a variable of a UDT, unless:

- An operation in the UDT editor modifies the offset or bit mask of an element that has the same name before and after the operation.
- The size of the UDT definition increases.
- Array length increases.
- The memory type of the UDT definition changes.
- There is a data type change in the UDT definition, except for the following interchangeable data types:
  - WORD, INT, UINT
  - DWORD, DINT
- The UDT definition is renamed.

### 3.10.5 UDT Operational Notes

- By default, a UDT variable resides in symbolic memory. You can convert the symbolic variable to an I/O variable.
- All UDT elements are public and, therefore, readable and writeable.
- Properties of elements of UDT variables:

The Input Transfer List and Output Transfer List properties are read-only and set to False.

The Retentive property is editable only for BOOLs and only if the UDT Memory Type is discrete. For UDTs whose Memory Type is non-discrete, a BOOL variable has its Retentive property set to True during validation.

- UDT variables are supported in LD, FBD, and ST blocks, as well as in Diagnostic Logic Blocks.

For additional operational notes, refer to the programmer Help.

#### Example

You want to set up six COMMREQ commands to send values to a series of six identical intelligent modules that require individualized data of the same data types in the same format, specified by the manual for the intelligent module. This data contains header information and several words of data. You could proceed as follows:

1. Add a UDT named COMMREQ6 and edit it to contain the data in the required data types and sequence.
2. Create an array of length 6, named ABC, of the COMMREQ6 data type.
3. The array resides in symbolic memory. You can convert the symbolic variable to an I/O variable.
4. Populate the variable. If the value of an element needs to be the same for all six COMMREQ6 elements, you can set up an ST for loop that uses a variable index to populate each element with the same data, for example:

```
for i = 1 to 6 do
  ABC[i].WaitFlag := 0;
end_for;
```
5. Just before issuing one or more COMMREQs, use the Move to Flat instruction to *flatten* the COMMREQ6 array or one or more of its top-level elements from a structure to a *flat* series of contiguous registers in an area of % memory supported by COMMREQ.
6. Issue the COMMREQs based on the % memory registers that you just populated with the Move to Flat instruction.

Although you can populate the memory registers directly without a UDT and Move to Flat, there are advantages when working with UDT variables:

- UDT variables reside in symbolic or I/O variable memory, which protects them from memory overlaps and offers more protection against overwriting, whereas reference memory areas offer no such protection. It is best to use reference memory just before issuing a COMMREQ.
- You can work with meaningful structure variable names and structure element names.
- You can set up loops with variable indexes to populate some of the values.

### 3.11 Operands for Instructions

The operands for PACSystems instructions can be in the following forms:

- Constants
- Variables that are located in any of the PACSystems memory areas (%I, %Q, %M, %T, %G, %S, %SA, %SB, %SC, %R, %W, %L, %P, %AI, %AQ)
- Symbolic variables, including I/O variables
- Parameters of a Parameterized block or C block
- Power flow
- Data flow
- Computed references such as indirect references or bit-in-word references
- BOOL arrays

An operand's type and length must be compatible with that of the parameter it is being passed into. PACSystems instructions and functions have the following operand restrictions:

- Constants cannot be used as operands to output parameters because output values cannot be written to constants.
- Variables located in %S memory cannot be used as operands to output parameters because %S memory is read-only.
- Variables located in %S, %SA, %SB, and %SC memories cannot be used as operands to numerical parameters such as INTs, DINTs, REALs, LREALs, etc.
- Data flow is prohibited on some input parameters of some functions. This occurs when the function, during the course of its execution, actually writes a value to the input parameter. Data flow is prohibited in these cases because data flow is stored in a temporary memory and any updated value assigned to it would be inaccessible to the user application.
- The arguments to EN, OK, and many other BOOLEAN input and output parameters are restricted to be power flow.
- Restrictions on using Parameterized block or External block parameters as operands to instructions or functions are documented in Chapter 2.
- References in discrete memory (I, Q, M, and T) must be byte-aligned.

Note the following:

- Indirect references, which are available for all WORD-oriented memories (%R, %W, %P, %L, %AI, %AQ), can be used as arguments to instructions wherever located variables in the corresponding WORD-oriented memory are allowed. Note that indirect references are converted into their corresponding direct references immediately before they are passed into an instruction or function.
- Bit-in-word references are generally allowed on contact and coil instructions other than legacy transition contacts and coils (POSCON, NEGCON, POSCOIL and NEGCOIL). They are also allowed as arguments to function parameters that accept single or unaligned bits.

BOOL arrays can be used as parameters to an instruction instead of variables of other data types. The array must be of sufficient length to replace the given data type. For example, instead of using a 16-bit INT variable, you could use a BOOL array of length 16 or more.

The following conditions must be met:

- The BOOL array must be byte-aligned, that is, the reference address of the first element of the BOOL array must be  $8n + 1$ , where  $n = 0, 1, 2, 3$ , and so on. For example, %M00033 is byte-aligned, because  $33 = (8 \times 4) + 1$ .
- The parameter in question must support discrete memory reference addresses.
- The instruction in question must not have a Length parameter. (The Length parameter is displayed as ?? in the LD editor until a value has been assigned.)
- The data type to be replaced with a BOOL array must be one of the following:

Data Type	Minimum Length
BYTE	8
INT, UINT, WORD	16
DINT, DWORD, REAL	32
REAL	64

- Excess bits are ignored. For example, if you use a BOOL array of length 12 instead of an 8-bit BYTE, the last four bits of the BOOL array are ignored.

## 3.12 Word-for-Word Changes

Many changes to the program that do not modify the size of the program are considered word-for-word changes. Examples include changing the type of contact or coil, or changing a reference address used for an existing function block.

The following are word-for-word changes:

- Switching between two symbolic variables
- Switching between a symbolic variable and a mapped variable
- Switching between a constant and a symbolic variable

### 3.12.1 Exception: Symbolic Variables

Creating, deleting, or modifying a symbolic variable definition is not a word-for-word change.



## Chapter 4 Ladder Diagram (LD) Programming

---

This chapter describes the programming instructions that can be used to create ladder logic programs for the PACSystems control system.

For an overview of the types of operands that can be used with instructions, refer to *Operands for Instructions* in Chapter 3.

The ladder logic implementation of the PACSystems instruction set includes the following categories:

- *Advanced Math Functions*
- *Bit Operation Functions*
- *Coils*
- *Contacts*
- *Control Functions*
- *Conversion Functions*
- *Counters*
- *Data Move Functions*
- *Data Table Functions*
- *Math Functions*
- *Program Flow Functions*
- *Relational Functions*
- *Timers*
- *Motion Functions and Function Blocks*  
RX3i CPUs support PLCopen compliant motion functions and function blocks. Details of these function blocks can be found in the *PACMotion Multi-Axis Motion Controller User's Manual*, GFK-2448.
- *PROFINET I/O Communication*  
Consists of the PNIO\_DEV\_COMM function. For details, refer to the *PACSystems RX3i & RSTi-EP PROFINET I/O Controller Manual*, GFK-2571.

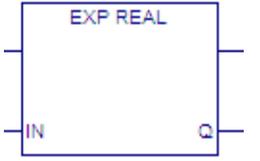
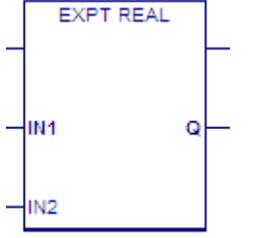
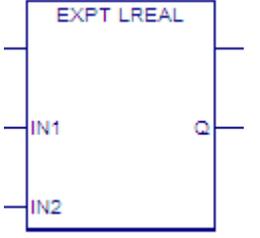
## 4.1 Advanced Math Functions

The Advanced Math functions perform logarithmic, exponential, square root, trigonometric, and inverse trigonometric operations.

Function	Mnemonic	Description
Exponential	EXP_REAL EXP_LREAL	Raises <b>e</b> to the value specified in IN ( $e^{IN}$ ). Calculates the inverse natural logarithm of the IN operand.
	EXPT_REAL EXPT_LREAL	Calculates IN1 to the IN2 power ( $IN1^{IN2}$ ).
Inverse Trig	ACOS_REAL ACOS_LREAL	Calculates the inverse cosine of the IN operand and expresses the result in radians.
	ASIN_REAL ASIN_LREAL	Calculates the inverse sine of the IN operand and expresses the result in radians.
	ATAN_REAL ATAN_LREAL	Calculates the inverse tangent of the IN operand and expresses the result in radians.
Logarithmic	LN_REAL LN_LREAL	Calculates the natural logarithm of the operand IN.
	LOG_REAL LOG_LREAL	Calculates the base 10 logarithm of the operand IN.
Square Root	SQRT_DINT	Calculates the square root of the operand IN, a double-precision integer, and stores in Q the double-precision integer portion of the square root of the input IN.
	SQRT_INT	Calculates the square root of the operand IN, a single-precision integer, and stores in Q the single-precision integer portion of the square root of the input IN.
	SQRT_REAL SQRT_LREAL	Calculates the square root of the operand IN, a real number, and stores the real-number result in Q
Trig	COS_REAL COS_LREAL	Calculates the cosine of the operand IN, where IN is expressed in radians.
	SIN_REAL SIN_LREAL	Calculates the sine of the operand IN, where IN is expressed in radians.
	TAN_REAL TAN_LREAL	Calculates the tangent of the operand IN, where IN is expressed in radians.

### 4.1.1 Exponential/Logarithmic Functions

When an exponential or logarithmic function receives power flow, it performs the appropriate operation on the REAL or LREAL input value(s) and places the result in output Q.

<p>The inverse natural log (EXP) function raises e to the power specified by IN.</p>		
<p>The Power of X (EXPT) function raises the value of input IN1 to the power specified by the value IN2.</p>		
<p>The Base 10 Logarithm (LOG) function calculates the base 10 logarithm of IN.</p>		
<p>The Natural Logarithm (LN) function calculates the logarithm of IN.</p>		

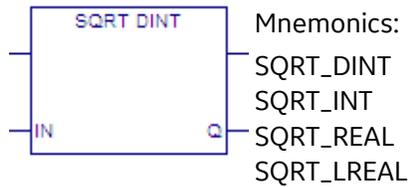
The power flow output is energized when the function is performed, unless *Overflow* or one of the following invalid conditions occurs:

- IN < 0, for LOG or LN
- IN1 < 0, for EXPT
- IN is negative infinity, for EXP
- IN, IN1, or IN2 is a NaN (Not a Number)

#### Operands of the Exponential/Logarithmic Functions

Parameter	Description	Allowed Operands	Optional
IN or IN1	For EXP, LOG, and LN, IN contains the REAL or LREAL value to be operated on. The EXPT function has two inputs, IN1 and IN2. For EXPT, IN1 is the base value and IN2 is the exponent.	All except variables located in %S—%SC	No
IN2 (EXPT)	The REAL or LREAL exponent for EXPT.	All except variables located in %S—%SC	No
Q	Contains the REAL or LREAL logarithmic/exponential value of IN or of IN1 and IN2.	All except constants and variables located in %S—%SC	No

### 4.1.2 Square Root



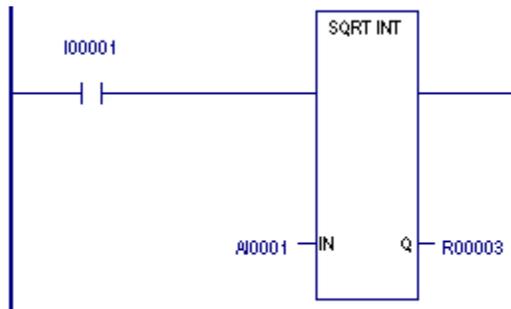
When the Square Root function receives power flow, it finds the square root of IN and stores the result in Q. The output Q must be the same data type as IN.

The power flow output is energized when the function is performed without *Overflow*, unless one of these invalid REAL operations occurs:

- If  $IN < 0$ , Q is set to 0 and ENO is set FALSE.
- If IN is a NaN (Not a Number), Q will also be a NaN value and ENO will be set false.

#### Example

The square root of the integer number located at %AI0001 is placed into %R0003 when %I0001 is ON.



#### Operands for the Square Root Function

Parameter	Description	Allowed Operands	Optional
IN	The value to calculate the square root of. If $IN < 0$ , the function does not pass power flow.	All except variables located in %S - %SC	No
Q	The calculated square root.	All except constants and variables located in %S - %SC	No

### 4.1.3 Trig Functions



Mnemonics:  
 SIN\_REAL  
 SIN\_LREAL  
 COS\_REAL  
 COS\_LREAL  
 TAN\_REAL  
 TAN\_LREAL

The SIN, COS, and TAN functions are used to find the trigonometric sine, cosine, and tangent, respectively, of an input whose units are radians. When one of these functions receives power flow, it computes the sine (or cosine or tangent) of IN and stores the result in output Q.

The SIN, COS, and TAN functions accept a broad range of input values, where  $-2^{63} < IN < 2^{63}$ , ( $2^{63}$  is approximately  $9.22 \times 10^{18}$ ). Input values outside this range will produce incorrect results.

The power flow output is energized unless the following invalid condition occurs:

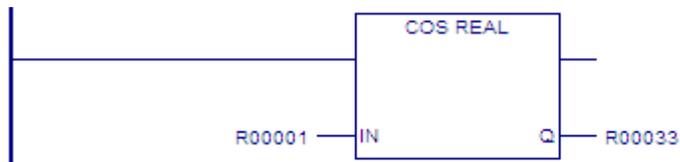
- IN or Q is a NaN (Not a Number)

#### Operands of Trig Functions

Parameter	Description	Allowed Operands	Optional
IN	Number of radians. $-2^{63} < IN < 2^{63}$	All except variables located in %S—%SC	No
Q	Trigonometric value of IN (REAL or LREAL)	All except constants and variables located in %S—%SC	No

#### Example

The COS of the value in V\_R0001 is placed in V\_R0033.



### 4.1.4 Inverse Trig – ASIN, ACOS, and ATAN



Mnemonics:  
 ASIN\_REAL  
 ASIN\_LREAL  
 ACOS\_REAL  
 ACOS\_LREAL  
 ATAN\_REAL  
 ATAN\_LREAL

When an Inverse Sine (ASIN), Inverse Cosine (ACOS), or Inverse Tangent (ATAN) function receives power flow, it respectively computes the inverse sine, inverse cosine or inverse tangent of IN and stores the result in radians in output Q.

The ASIN and ACOS functions accept a narrow range of input values, where  $-1 \leq IN \leq 1$ . Given a valid value for the IN parameter, the ASIN function produces a result Q such that:

$$ASIN(IN) = -\frac{\pi}{2} \leq Q \leq \frac{\pi}{2}$$

The ACOS function produces a result Q such that:

$$ACOS(IN) = 0 \leq Q \leq \pi$$

The ATAN function accepts the broadest range of input values, where  $-\infty \leq IN \leq +\infty$ . Given a valid value for the IN parameter, the ATAN function produces a result Q such that:

$$ATAN(IN) = -\frac{\pi}{2} \leq Q \leq \frac{\pi}{2}$$

The power flow output is energized unless one of the following invalid conditions occurs:

- IN is outside the valid range for ASIN, ACOS, or ATAN
- IN is a NaN (Not a Number)

#### Operands of Inverse Trig Functions

Parameter	Description	Allowed Operands	Optional
IN	The REAL or LREAL value to process. ASIN and ACOS: $-1 \leq IN \leq 1$ ATAN: $-\infty \leq IN \leq +\infty$	All except variables located in %S - %SC	No
Q	Trigonometric value of IN. REAL or LREAL value expressed in radians. ASIN: $(-\pi/2) \leq Q \leq (\pi/2)$ ACOS: $0 \leq Q \leq \pi$ ATAN: $(-\pi/2) \leq Q \leq (\pi/2)$	All except constants and variables located in %S - %SC	No

## 4.2 Bit Operation Functions

The Bit Operation functions perform comparison, logical, and move operations on bit strings.

Function	Mnemonics	Description
Bit Position	BIT_POS_DWORD BIT_POS_WORD	Bit Position. Locates a bit set to 1 in a bit string.
Bit Sequencer	BIT_SEQ	Bit Sequencer. Sequences a string of bit values, starting at ST. Performs a bit sequence shift through an array of bits. The maximum length allowed is 256 words.
Bit Set, Clear	BIT_SET_DWORD BIT_SET_WORD	Bit Set. Sets a bit in a bit string to 1.
	BIT_CLR_DWORD BIT_CLR_WORD	Bit Clear. Clear a bit within a string by setting that bit to 0.
Bit Test	BIT_TEST_DWORD BIT_TEST_WORD	Bit Test. Tests a bit within a bit string to determine whether that bit is currently 1 or 0.
Logical AND	AND_DWORD AND_WORD	Compares the bit strings IN1 and IN2 bit by bit. When a pair of corresponding bits are both 1, places a 1 in the corresponding location in output string Q; otherwise, places a 0 in the corresponding location in Q.
Logical NOT	NOT_DWORD NOT_WORD	Logical invert. Sets the state of each bit in output bit string Q to the opposite state of the corresponding bit in bit string IN1.
Logical OR	OR_DWORD OR_WORD	Compares the bit strings IN1 and IN2 bit by bit. When a pair of corresponding bits are both 0, places a 0 in the corresponding location in output string Q; otherwise, places a 1 in the corresponding location in Q.
Logical XOR	XOR_DWORD XOR_WORD	Compares the bit strings IN1 and IN2 bit by bit. When a pair of corresponding bits are different, places a 1 in the corresponding location in the output bit string Q; when a pair of corresponding bits are the same, places a 0 in Q.
Masked Compare	MASK_COMP_DWORD MASK_COMP_WORD	Masked Compare. Compares the contents of two separate bit strings with the ability to mask selected bits.
Rotate Bits	ROL_DWORD ROL_WORD	Rotate Left. Rotates all the bits in a string a specified number of places to the left.
	ROR_DWORD ROR_WORD	Rotate Right. Rotates all the bits in a string a specified number of places to the right.
Shift Bits	SHIFTL_DWORD SHIFTL_WORD	Shift Left. Shifts all the bits in a word or string of words to the left by a specified number of places.
	SHIFTR_DWORD SHIFTR_WORD	Shift Right. Shifts all the bits in a word or string of words to the right by a specified number of places.

### 4.2.1 Data Lengths for the Bit Operation Functions

The Bit Operation functions operate on a single WORD or DWORD of data or up to 256 WORDs or DWORDs that occupy adjacent memory locations.

Bit Operation functions treat the WORD or DWORD data as a continuous string of bits, with bit 1 of the first WORD or DWORD being the Least Significant Bit (LSB). The last bit of the last WORD or DWORD is the Most Significant Bit (MSB). For example, if you specify three WORDs of data beginning at reference %R0100, they are treated as 48 contiguous bits.

%R0100	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	← bit 1 (LSB)
%R0101	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	
%R0102	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	
	↑																
	(MSB)																

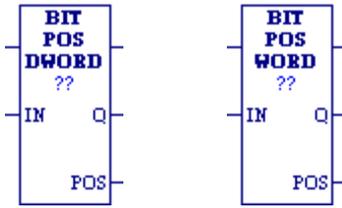


#### Warning

**Overlapping input and output reference address ranges in multiword functions is not recommended, as it can produce unexpected results.**

Note that for all functions (Bit Test, Bit Set, Bit Clear, and Bit Position) that return a bit position indicator as an output parameter (POS), bit position numbering starts at 1, not 0, as shown in the diagram above.

### 4.2.2 Bit Position



The Bit Position function locates a bit set to 1 in a bit string.

Each scan that power is received, the function scans the bit string starting at IN. When the function stops scanning, either a bit equal to 1 has been found or the entire length of the string has been scanned.

POS is set to the position within the bit string of the first non-zero bit; POS is set to zero if no non-zero bit is found.

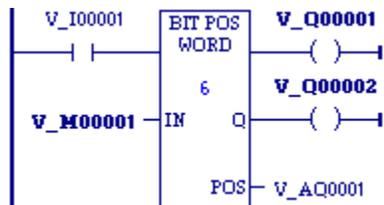
A string length of 1 to 256 WORDs or DWORDs can be selected. The function passes power flow to the right whenever it receives power.

#### Operands of Bit Position

Parameter	Description	Allowed Operands	Optional
Length (displayed as ??)	The number of WORDs or DWORDs in the bit string. $1 \leq \text{Length} \leq 256$ .	Constants	No
IN	The data to operate on	All. Constants may only be used when Length is 1.	No
Q	Energized if a bit set to 1 is found	Flow	Yes
POS	An unsigned integer giving the position of the first nonzero bit found, or zero if no non-zero bit is found	All except constants and variables located in %S - %SC	No

#### Examples

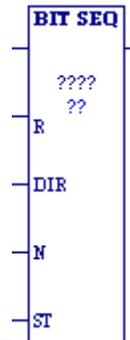
When V\_I00001 is set, the bit string starting at V\_M00001 is searched until a bit equal to 1 is found, or 6 words have been searched. Coil V\_Q00001 is turned on. If a bit equal to 1 is found, its location within the bit string is written to V\_AQ0001 and V\_Q00002 is turned on. For example, if V\_00001 is set, bit V\_M00001 is 0, and bit V\_M0002 is 1, the value written to V\_AQ0001 is 2.



### 4.2.3 Bit Sequencer

The Bit Sequencer (BIT\_SEQ) function performs a bit sequence shift through a series of contiguous bits.

The operation of BIT\_SEQ depends on the value of the reset input (R), and both the current value and previous value of the enabling power flow input (EN):



R Current Execution	EN Previous Execution	EN Current Execution	Bit Sequencer Execution
ON	ON/OFF	ON/OFF	Bit sequencer resets
OFF	OFF	ON	Bit sequencer increments/decrements by 1
		OFF	Bit sequencer does not execute
	ON	ON/OFF	Bit sequencer does not execute

The reset input (R) overrides the enabling power flow (EN) and always resets the sequencer. When R is active, the current step number is set to the value of the optional N operand. If you did not specify N, the step number is set to 1. All bits in the bit sequencer, ST, are set to 0, except for the bit pointed to by the current step, which is set to 1.

When EN is active and R is not active, and the previous EN was OFF, the bit pointed to by the current step number is cleared. The current step number is incremented or decremented, based on the direction (DIR) operand. Then the bit pointed to by the new step number is set to 1.

- When the step number is being incremented and it goes outside the range of  $(1 \leq \text{step number} \leq \text{Length})$ , it is set back to 1.
- When the step number is being decremented and it goes outside the range of  $(1 \leq \text{step number} \leq \text{Length})$ , it is set to Length.

The parameter ST is optional. If it is not used, BIT\_SEQ operates as described above, except that no bits are set or cleared. The function just cycles the current step number through its allowed range.

BIT\_SEQ passes power to the right whenever it receives power.

**Note:** Before using the BIT\_SEQUENCER function block, the current step number (Word 1 in the control block) must be set to an integer value between 1 and the length, as defined in the function block properties. Failure to properly initialize the step number in the BIT\_SEQUENCER function block may result in the CPU going to STOP-HALT mode.

Asserting the Reset parameter (R), before using the BIT SEQUENCER function block assures that the current step number is set to a valid value.

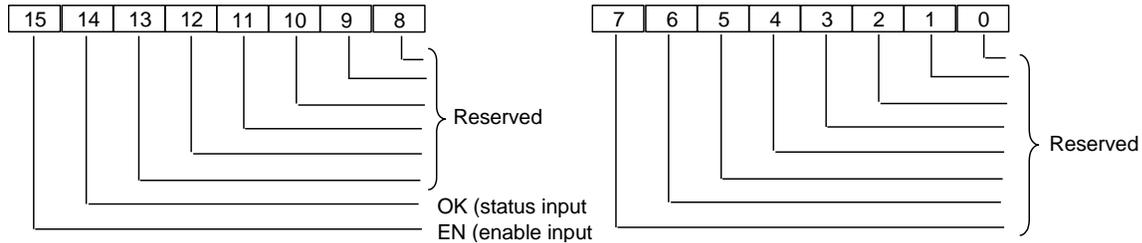
### Memory Required for Bit Sequencer

Each bit sequencer uses a three-word array of control block information. The control block can be a symbolic variable or it can be located in %R, %W, %L, or %P memory:

Word 1	current step number
Word 2	length of sequence (in bits)
Word 3	control word

**Note:** Do not write to the control block memory registers from other functions.

Word 3 (the control word) stores the state of the Boolean inputs and outputs of its associated function in the following format:



Notes:

Bits 0 through 13 are not used.

In the N operand, bits are entered as 1 through 16, not 0 through 15.

### Operands for Bit Sequencer



#### Warning

**Do not write to the Control Block memory with other instructions. Overlapping references may cause erratic operation of BIT\_SEQ.**

Parameter	Description	Allowed Operands	Optional
Address (????)	Beginning address of the Control Block, which is a three-word array: Word 1: current step number Word 2: length of sequence in bits Word 3: control word, which tracks the status of the last enabling power flow and the status of the power flow to the right.	Symbolic variables, variables located in %R, %W, %P, or %L	No
Length (??)	The number of bits in the bit sequencer, ST, that BIT_SEQ will step through. $1 \leq \text{Length} \leq 256$ .	Constants	No
R	When R is energized, the step number of BIT_SEQ is set to the value in N (default = 1), and the bit sequencer, ST, is filled with zeroes, except for the current step number bit.	Flow	No
DIR	(Direction) When DIR is energized, the step number of BIT_SEQ is incremented prior to the shift. Otherwise, it is decremented.	Flow	No

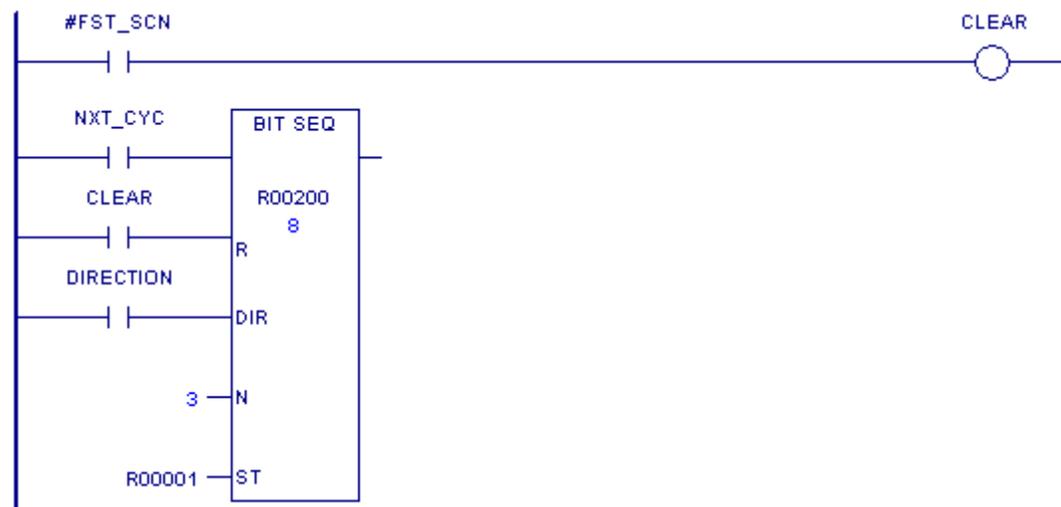
Parameter	Description	Allowed Operands	Optional
N	The value that the step number is set to when R is energized. Default value is 1. $1 \leq N \leq \text{Length}$ . If $N < 1$ , the step number will be reset to 1 when R is energized. If $N > \text{Length}$ , the step number will be reset to Length. Must be an integer variable or constant.	All except variables located in %S - %SC	Yes
ST	Contains the first word of the bit sequencer. If ST is not used, the Bit Sequencer function operates as described above, except that no bits are set or cleared. The function just cycles the current step number (in word 1 of the control block) through its allowed range. If ST is in %M memory and the Length is 3, the bit sequencer occupies 3 bits; the other 5 bits of the byte are not used. If ST is in %R memory, and the Length is 17, the bit sequencer uses 4 bytes, all of %R1 and %R2.	All except constants, flow, and variables located in %S	Yes

**Example**

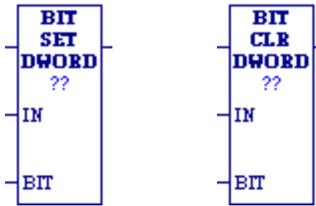
In the following example, a #FST\_SCN system variable is used to set CLEAR to ON for one scan. This sets the step number in Word 1 of the Bit Sequencer’s control block to an initial value of 3.

The Bit Sequencer operates on register memory %R00001. Its control block is stored in registers %R0010, %R0011, and %R0012. When CLEAR is active, the sequencer is reset and the current step is set to step number 3, as specified in N. The third bit of %R00001 is set to one and the other seven bits are set to zero.

When NXT\_CYC is active and CLEAR is not active, the bit for step number 3 is cleared and the bit for step number 2 or 4 (depending on whether DIRECTION is energized) is set.



### 4.2.4 Bit Set, Bit Clear



- Mnemonics  
 BIT\_SET\_DWORD  
 BIT\_SET\_WORD  
 BIT\_CLR\_DWORD  
 BIT\_CLR\_WORD

The Bit Set (BIT\_SET\_DWORD and BIT\_SET\_WORD) function sets a bit in a bit string to 1. The Bit Clear (BIT\_CLR\_DWORD and BIT\_CLR\_WORD) function clears a bit in a string by setting the bit to 0.

Each scan that power is received; the function sets or clears the specified bit. If a variable rather than a constant is used to specify the bit number, the same function can set or clear different bits on successive scans. Only one bit is set or cleared, and the transition information for that bit is updated. The transition status of all the other bits in the bit string is not affected.

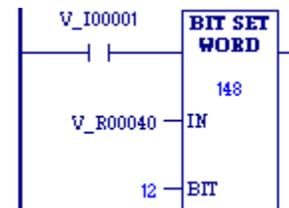
The function passes power flow to the right, unless the value for BIT is outside the specified range.

#### Operands for Bit Set, Bit Clear

Parameter	Description	Allowed Operands	Optional
Length (??)	The number of WORDs or DWORDs in the bit string. $1 \leq \text{Length} \leq 256$ .	Constants	No
IN	The first WORD or DWORD of the data to process	All except constants, flow, and variables located in %S	No
BIT	The number of the bit to set or clear in IN. $1 \leq \text{BIT} \leq (16 \times \text{Length})$ for WORD. $1 \leq \text{BIT} \leq (32 \times \text{length})$ for DWORD	All except variables located in %S - %SC	No

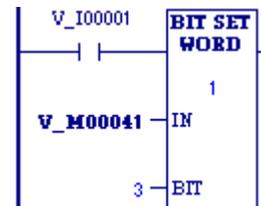
#### Example 1

Whenever input V\_I0001 is set, bit 12 of the string beginning at reference %R00040 (as specified by variable V\_R0040) is set to 1.



#### Example 2

Whenever V\_I00001 is set, %M00043, the third bit of the string beginning at %M00041, is set to 1. Note that neither the status nor the transition value of any of the other bits in the same byte as %M00043 (e.g., %M00041, %M00042, %M00044, etc.) is affected by the BIT\_SET function



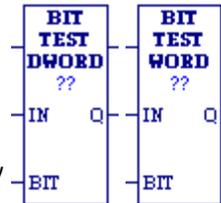
### 4.2.5 Bit Test

When the Bit Test function receives power flow, it tests a bit within a bit string to determine whether that bit is currently 1 or 0. The result of the test is placed in output Q.

Each scan that power is received, the Bit Test function sets its output Q to the same state as the specified bit. If a register rather than a constant is used to specify the bit number, the same function can test different bits on successive sweeps. If the value of BIT is outside the range ( $1 \leq \text{BIT} \leq (16 \times \text{length})$  for a WORD and  $1 \leq \text{BIT} \leq (32 \times \text{length})$  for a DWORD), then Q is set OFF.

You can specify a string length of 1 to 256 WORDs or DWORDs.

**Note:** When using the Bit Test function, the bits are numbered 1 through 16 for a WORD, *not* 0 through 15. They are numbered 1 through 32 for a DWORD.



#### Operands for Bit Test

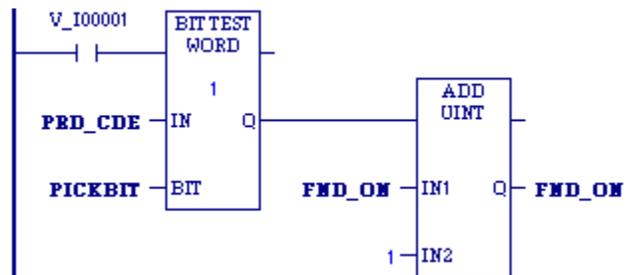
Parameter	Description	Allowed Operands	Optional
Length (??)	The number of WORDs or DWORDs in the data string to test. $1 \leq \text{Length} \leq 256$ .	Constant	No
IN	The first WORD or DWORD in the data to test	All	No
BIT	The number of the bit to test in IN. $1 \leq \text{BIT} \leq (16 \times \text{Length})$ .	All except variables located in %S - %SC	No
Q	The state of the specific bit tested; Q is energized if the bit tested is a 1.	Flow	No

#### Example 1

When input V\_I0001 is set, the bit at the location contained in reference PICKBIT is tested.

The bit is part of string PRD\_CDE.

If it is 1, output Q passes power flow to the ADD function, causing 1 to be added to the current value of the ADD function input IN1.

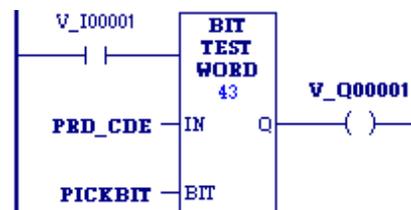


#### Example 2

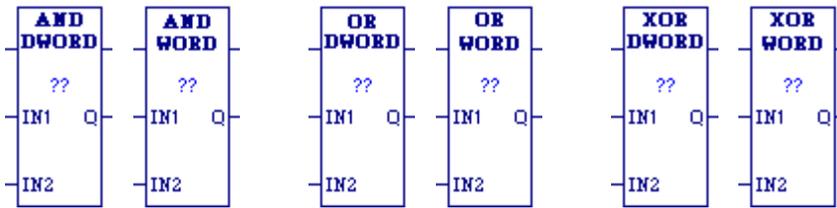
When input V\_I0001 is set, the bit at the location contained in reference PICKBIT is tested.

The bit is part of string PRD\_CDE.

If it is 1, output Q passes power flow and the coil V\_Q0001 is turned on



## 4.2.6 Logical AND, Logical OR, and Logical XOR



Each scan that power is received, the Logical function examines each bit in bit string IN1 and the corresponding bit in bit string IN2, beginning with the least significant bit in each. You can specify a string length of 1 to 256 WORDs or DWORDs. The IN1 and IN2 bit strings specified may overlap.

### Logical AND

If both bits examined by the Logical AND function are 1, AND places a 1 in the corresponding location in output string Q. If either bit is 0 or both bits are 0, AND places a 0 in string Q in that location.

AND passes power flow to the right whenever it receives power.

**Tip:** You can use the Logical AND function to build masks or screens, where only certain bits are passed (the bits opposite a 1 in the mask), and all other bits are set to 0.

### Logical OR

If either bit examined by the Logical OR function is 1, OR places a 1 in the corresponding location in output string Q. If both bits are 0, Logical OR places a 0 in string Q in that location. The function passes power flow to the right whenever it receives power.

Tips:

- You can use the Logical OR function to combine strings or to control many outputs with one simple logical structure. The Logical OR function is the equivalent of two relay contacts in parallel multiplied by the number of bits in the string.
- You can use the Logical OR function to drive indicator lamps directly from input states or to superimpose blinking conditions on status lights.

### Logical XOR

When the Exclusive OR (XOR) function receives power flow, it compares each bit in bit string IN1 with the corresponding bit in string IN2. If the bits are different, a 1 is placed in the corresponding position in the output bit string.

For each pair of bits examined, if only one bit is 1, then XOR places a 1 in the corresponding location in bit string Q. XOR passes power flow to the right whenever it receives power.

Tips for Logical XOR

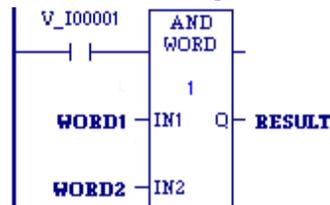
- If string IN2 and output string Q begin at the same reference, a 1 placed in string IN1 will cause the corresponding bit in string IN2 to alternate between 0 and 1, changing state with each scan as long as power is received.
- You can program longer cycles by pulsing the power flow to the function at twice the desired rate of flashing. The power flow pulse should be one scan long (one-shot type coil or self-resetting timer).
- You can use XOR to quickly compare two bit strings, or to blink a group of bits at the rate of one ON state per two scans.
- XOR is useful for transparency masks.

### Operands for Logical AND, OR, and XOR

Parameter	Description	Allowed Operands	Optional
Length (??)	The number of words in the bit string on which to perform the logical operation. $1 \leq \text{Length} \leq 256$ .	Constant	No
IN1	The first WORD or DWORD of the first string operate on.	All	No
IN2 (Must be the same data type as IN1.)	The first WORD or DWORD of the second string to operate on.	All	No
Q (Must be the same data type as IN1.)	The first WORD or DWORD of the operation's result.	All except constants and variables located in %S memory	No

#### Example: Logical AND

When input v\_I0001 is set, the 16-bit strings represented by variables WORD1 and WORD2 are examined. The logical AND places the results in output string RESULT.

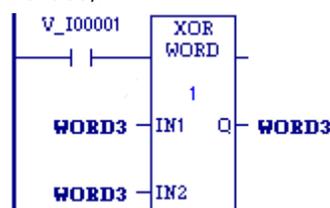


<b>WORD1</b>	0	0	0	1	1	1	1	1	1	1	0	0	1	0	0	0
<b>WORD2</b>	1	1	0	1	1	1	0	0	0	0	0	0	1	1	1	1

<b>RESULT</b>	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0
---------------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

#### Example: Logical XOR

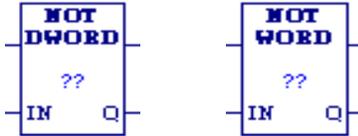
Whenever v\_I0001 is set, the bit string represented by the variable WORD3 is cleared (set to all zeroes).



<b>I1 (WORD3)</b>	0	0	0	1	1	1	1	1	1	1	0	0	1	0	0	0
<b>I2 (WORD3)</b>	0	0	0	1	1	1	1	1	1	1	0	0	1	0	0	0

<b>Q (WORD3)</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
------------------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

### 4.2.7 Logical NOT



When the Logical Not or Logical Invert (NOT) function receives power flow, it sets the state of each bit in the output bit string Q to the opposite of the state of the corresponding bit in bit string IN1.

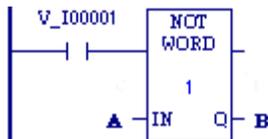
All bits are altered on each scan that power is received, making output string Q the logical complement of input string IN1. Logical NOT passes power flow to the right whenever it receives power. You can specify a string length of 1 to 256 WORDs or DWORDs

#### Operands for Logical NOT

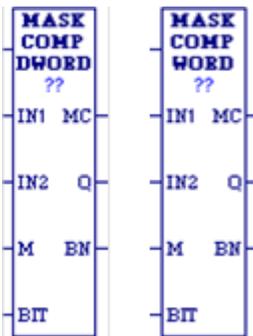
Parameter	Description	Allowed Operands	Optional
Length (??)	The number of WORDs or DWORDs in the bit string to NOT. $1 \leq \text{Length} \leq 256$ .	Constant	No
IN1	The first WORD or DWORD of the input string to NOT.	All	No
Q (Must be the same data type as IN1)	The first WORD or DWORD of the NOT's result.	All except constants and variables located in %S memory	No

#### Example

When input V\_I0001 is set, the bit string represented by the variable A is negated. Logical NOT stores the resulting inverse bit string in variable B. Variable A retains its original bit string value.



### 4.2.8 Masked Compare



The Masked Compare (MASK\_COMP\_DWORD and MASK\_COMP\_WORD) function compares the contents of two bit strings. It provides the ability to mask selected bits

**Tip:** Input string 1 might contain the states of outputs such as solenoids or motor starters. Input string 2 might contain their input state feedback, such as limit switches or contacts.

When the function receives power flow, it begins comparing the bits in the first string with the corresponding bits in the second string. Comparison continues until a mismatch is found or until the end of the string is reached.

The BIT input stores the bit number where the next comparison should start. Ordinarily, this is the same as the number where the last mismatch occurred. Because the bit number of the last mismatch is stored in output BN, the same reference can be used for both BIT and BN. The comparison actually begins 1 bit following BIT; therefore, the initial value of BIT should be 1 less first bit to be compared (for example, zero (0) to begin comparison at %I00001). Using the same reference for BIT and BN causes the compare to start at the next bit position after a mismatch; or, if all bits compared successfully upon the next invocation of the function, the compare starts at the beginning.

**Tip:** If you want to start the next comparison at some other location in the string, you can enter different references for BIT and BN. If the value of BIT is a location that is beyond the end of the string, BIT is reset to 0 before starting the next comparison.

The function passes power flow whenever it receives power. The other outputs of the function depend on the state of the corresponding mask bit.

**If all corresponding bits in strings IN1 and IN2 match**, the function sets the mismatch output MC to 0 and BN to the highest bit number in the input strings. The comparison then stops. On the next invocation of a Masked Compare, it is reset to 0.

**If a Mismatch is found**, that is, if the two bits being compared are not the same, the function checks the correspondingly numbered bit in string M (the mask).

If the mask bit is a 1, the comparison continues until it reaches another mismatch or the end of the input strings.

If a mismatch is detected and the corresponding mask bit is a 0, the function does the following:

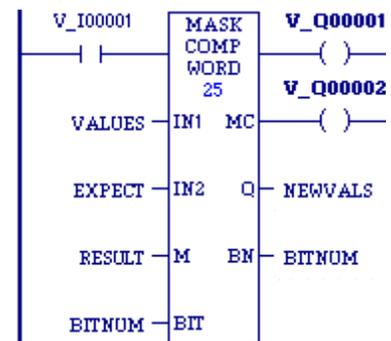
1. Sets the corresponding mask bit in M to 1.
2. Sets the mismatch (MC) output to 1.
3. Updates the output bit string Q to match the new content of mask string M.
4. Sets the bit number output (BN) to the number of the mismatched bit.
5. Stops the comparison.

### Operands for Masked Compare Function

Parameter	Description	Allowed Operands	Optional
Length (??)	The number of DWORDs or WORDs in the two compared strings. DWORD: $1 \leq \text{Length} \leq 2,048$ WORD: $1 \leq \text{Length} \leq 4,096$	Constant	No
IN1	The first bit string to be compared	All. Constants are legal only when Length is 1	No
IN2	The second bit string to be compared	All. Constants are legal only when Length is 1	No
M	The bit string mask containing the ongoing status of the compare	All except flow or variables in %S memory. Constants are legal only when Length is 1	No
BIT	BIT+1=the bit number where the next comparison starts	All except variables in %S - %SC memories	No
Q	The output copy of the compare mask bit string	All except constants	No
BN	The number of the bit where the latest mismatch occurred, or the highest bit number in the inputs if no mismatch occurred	All except constants and variables in %S memory	No
MC	Can be used to determine if a mismatch has occurred.	flow	Yes

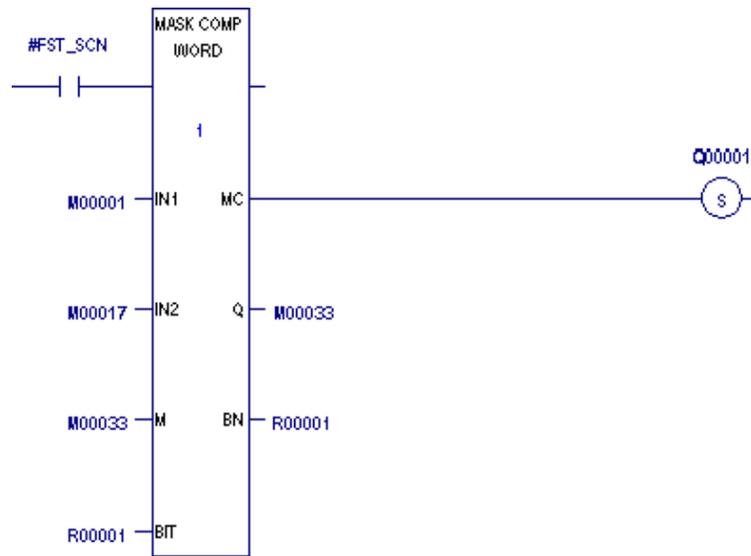
#### Masked Compare Example 1

When %I00001 is set, MASK\_COMP\_WORD compares the bits represented by the reference VALUES against the bits represented by the reference EXPECT. Comparison begins at BITNUM+1. If an unmasked mismatch is detected, the comparison stops. The corresponding bit is set in the mask RESULT. BITNUM is updated to contain the bit number of the mismatched bit. In addition, the output string NEWVALS is updated with the new value of RESULT, and coil %Q00002 is turned on. Coil %Q00001 is turned on whenever MASK\_COMP\_WORD receives power flow.



**Masked Compare Example 2**

On the first scan, the Masked Compare Word function executes. %M0001 through %M0016 is compared with %M0017 through %M0032. %M0033 through %M0048 contains the mask value. The value in %R0001 determines the bit position in the two input strings where the comparison starts.



Before the function is executed, the contents of the above references are:

(I1) - %M0001 = 6C6Ch =

0	1	1	0	1	1	0	0	0	0	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(I2) - %M0017 = 606Fh =

0	1	1	0	1	1	0	1	0	1	1	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(M/Q) - %M0033 = 000Fh =

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(BIT/BN) - %R0001 = 0

(MC) - %Q0001 = OFF

The contents of these references after the function block is executed are as follows:

(I1) - %M0001 =

0	1	1	0	1	1	0	0	0	1	1	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(I2) - %M0017 =

0	1	1	0	1	1	0	1	0	1	1	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(M/Q) - %M0033 =

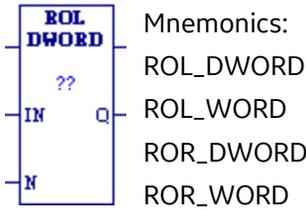
0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(BIT/BN) - %R0001 = 8

(MC) - %Q0001 = ON

The #FST\_SCN contact forces one and only one execution; otherwise, the function would repeat with possibly unexpected results.

### 4.2.9 Rotate Bits



When receiving power flow, the Rotate Bits Right (ROR\_DWORD and ROR\_WORD) and Rotate Bits Left (ROL\_DWORD and ROL\_WORD) functions rotate all the bits in a string of WORDs or DWORDs N positions respectively to the right or to the left. When rotation occurs, the specified number of bits is rotated out of the input string respectively to the right or to the left and back into the string on the other side.

The Rotate Bits function passes power flow to the right, unless the number of bits to rotate is less than 0, or is greater than the total length of the string. The result is placed in output string Q. If you want the input string to be rotated, the output parameter Q must use the same memory location as the input parameter IN. The entire rotated string is written on each scan that power is received.

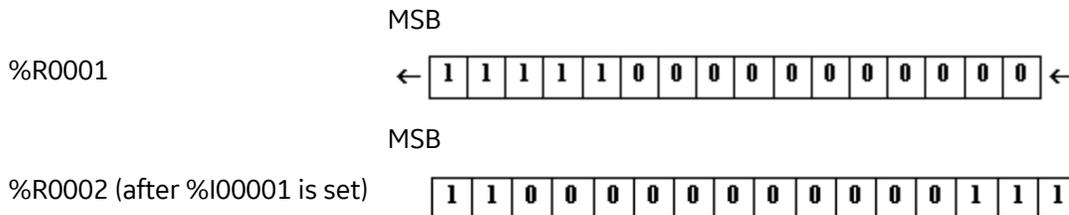
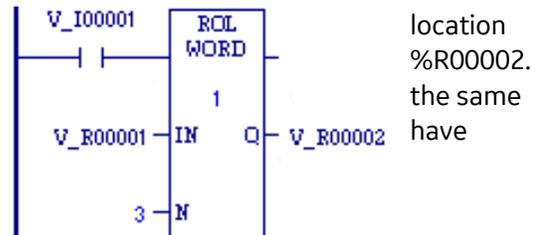
A string length of 1 to 256 words or double words can be specified.

#### Operands for Rotate Bits

Parameter	Description	Allowed Operands	Optional
Length (??)	The number of WORDs or DWORDs in the string to be rotated. $1 \leq \text{Length} \leq 256$ .	Constant	No
IN	The string to rotate	All. Constants are legal when Length is 1	No
N	The number of positions to rotate. $0 \leq N \leq \text{Length}$ .	All except variables in %S - %SC memories	No
Q	The resulting rotated string	All except constants and variables in %S memory	No

#### Example

Whenever input V\_I0001 is set, the input bit string in %R0001 is rotated left 3 bits and the result is placed in %R0002. The actual input bit string %R0001 is left unchanged. If reference had been used for IN and Q, a rotation would have occurred in place.



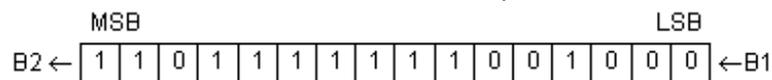
### 4.2.10 Shift Bits



Mnemonics:  
 SHIFTL\_DWORD  
 SHIFTL\_WORD  
 SHIFTR\_DWORD  
 SHIFTR\_WORD

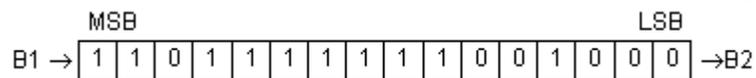
#### Shift Left

When the Shift Left (SHIFTL\_WORD) function receives power flow, it shifts all the bits in a word or group of words to the left by a specified number of places, N. When the shift occurs, the specified number of bits is shifted out of the output string to the left. As bits are shifted out of the high end of the string (Most Significant Bit (MSB)), the same number of bits is shifted in at the low end (Least Significant Bit (LSB)). The SHIFTL\_DWORD function operates in a similar manner on DWORDs instead of WORDs.



#### Shift Right

When the Shift Right (SHIFTR\_WORD) function receives power flow, it shifts all the bits in a word or group of words a specified number of places to the right (N). When the shift occurs, the specified number of bits is shifted out of the output string to the right. As bits are shifted out of the low end of the string (LSB), the same number of bits is shifted in at the high end (MSB).



#### Shift Left and Shift Right

A string length (Length) of 1 to 256 words can be specified.

The bits being shifted into the beginning of the string are specified via input parameter B1. If the value of N is greater than 1, each bit is filled with the same value (0 or 1). This can be:

- The Boolean output of another program function.
- All 1s. To do this, use the #AWL\_ON (always on) system bit (in memory location %S7), as a permissive to input B1.
- All 0s. To do this, use the #ALW\_OFF (always off) system bit (in memory location %S8), as a permissive to input B1.

The Shift Bits function passes power flow to the right, unless the number of bits specified to shift is zero or is greater than the array size.

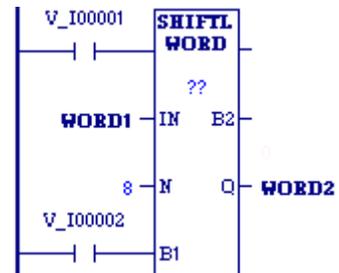
Output Q is the shifted copy of the input string. If you want the input string to be shifted, the output parameter Q must use the same memory location as the input parameter IN. The entire shifted string is written on each scan that power is received. Output B2 is the last bit shifted out. For example, if four bits were shifted, B2 would be the fourth bit shifted out.

**Operands for Shift Left, Shift Right, Shift Left and Shift Right**

Parameter	Description	Allowed Operands	Optional
Length (??)	The number of WORDs or DWORDs in the string. $1 \leq \text{Length} \leq 256$ .	Constants.	No
IN	The string of WORDs or DWORDs to shift	All. Constants are legal only when Length = 1.	No
N	The number of places (bits) to shift the array. $0 \leq N \leq \text{Length}$ If N is 0, no shift occurs, but power flow is generated. If N is greater than the number of bits in the string (Length), all bits in Q are set to the value B1, OK is set FALSE, and B2 is set to B1.	All except variables in %S— %SC memories	No
B1	The bit value to shift into the array	flow	No
B2	The bit value of the last bit shifted out of the array.	flow	Yes
Q (Must be the same data type as IN)	The first WORD or DWORD of the shifted array	All except constants and variables in %S memory.	No

**Example**

Whenever input V\_I0001 is set, the bits in the input string that begins at WORD1 are copied to the output bit string that starts at WORD2. WORD2 is left-shifted by 8 bits, as specified by the input N. The resulting open bits at the beginning of the output string are set to the value of V\_I0002.



## 4.3 Coils

Coils are used to control the discrete (BOOL) references assigned to them. Conditional logic must be used to control the flow of power to a coil. Coils cause action directly. They do not pass power flow to the right. If additional logic in the program should be executed as a result of the coil condition, you can use an internal reference for the coil or a continuation coil/contact combination.

A continuation coil does not use an internal reference. It must be followed by a continuation contact at the beginning of any rung following the continuation coil.

Coils are always located at the rightmost position of a line of logic.

### 4.3.1 Coil Checking

The level of coil checking is set to *Show as error* by default. If you want a coil conflict to result in a warning instead of this error, or if you want no warning at all, edit the Controller option: **Multiple Coil Use Warning** in the programming software.

The *Show as warning* option enables you to use any coil reference with multiple Coils, Set Coils, and Reset Coils, but you will be warned at validation time every time you do so. With both the *Show as warning* and the *no warning* options, a reference can be set ON by either a Set Coil or a normal Coil and can be set OFF by a Reset Coil or by a normal Coil.

### 4.3.2 Graphical Representation of Coils

The programming software displays the COIL, NCCOIL, SETCOIL, and RESETCOIL instructions differently depending on the retentive state of the BOOL variables assigned to them. Examples are provided in the discussion of each type of coil. For a discussion of retentiveness, refer to *Retentiveness of Logic and Data* in Chapter 3.

#### Coil (Normally Open)



A retentive variable is assigned to the coil



A non-retentive variable is assigned to the coil

When a COIL receives power flow, it sets its associated BOOL variable ON (1). When it receives no power flow, it sets the associated BOOL variable OFF (0). COIL can be assigned a retentive variable or a non-retentive variable.

Valid memory areas: %I, %Q, %M, %T, %SA - %SC, and %G. Symbolic discrete variables are permitted. Bit-in-word references on any word-oriented memory except %AI, including symbolic non-discrete memory, are also permitted.

#### Continuation Coil



A continuation coil instructs the PLC to continue the present rung's LD logic power flow value (TRUE or FALSE) at the continuation contact on a following rung.

The flow state of the continuation coil is passed to the continuation contact.

#### Notes:

- If the flow of logic does not execute a continuation coil before it executes a continuation contact, the state of the continuation contact is no flow (FALSE).
- The continuation coil and the continuation contact do not use parameters and do not have associated variables.
- You can have multiple rungs with continuation contacts after a single continuation coil.
- You can have multiple rungs with continuation coils before one rung with a continuation contact.

#### Negated Coil



A retentive variable is assigned to the negated coil



A non-retentive variable is assigned to the negated coil

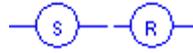
When it does *not* receive power flow, a negated coil (NCCOIL) sets a discrete reference ON. When it does receive power flow, NCCOIL sets a discrete reference OFF. NCCOIL can be assigned a retentive variable or a non-retentive variable.

Valid memory areas: %I, %Q, %M, %T, %SA - %SC, and %G. Symbolic discrete variables are permitted. Bit-in-word references on any word-oriented memory except %AI, including symbolic non-discrete memory, are also permitted.

### 4.3.3 Set Coil, Reset Coil



Set Coil and Reset Coil with a retentive variable assigned



Set Coil and Reset Coil with a non-retentive variable assigned

The SET and RESET coils can be used to keep (i.e. *latch*) the state of a reference either ON or OFF.



#### Warning

**SET / RESET coils write an undefined result to the transition bit for the given reference. This result differs from that written by Series 90-70 CPUs and could change for future PACSystems CPU models.**

**Because they write an undefined result to transition bits, do not use SET or RESET coils with references used on POSCON or NEGCON transition contacts.**

When a SET coil receives power flow, it sets its discrete reference ON. When a SET coil does not receive power flow, it does not change the value of its discrete reference. Therefore, whether or not the coil itself continues to receive power flow, the reference stays ON until the reference is reset by other logic, such as a RESET coil.

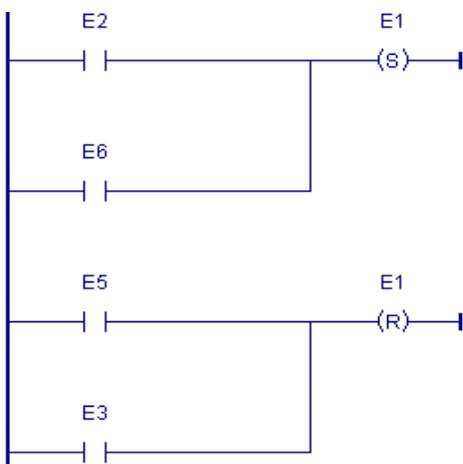
When a RESET coil receives power flow, it resets a discrete reference to OFF. When a RESET coil does not receive power flow, it does not change the value of its discrete reference. Therefore, its reference remains OFF until it is set ON by other logic, such as a SET coil.

The last solved SET coil or RESET coil of a pair takes precedence.

The SET and RESET coils can be assigned a retentive variable or a non-retentive variable.

Valid memory areas: %I, %Q, %M, %T, %SA - %SC, and %G. Symbolic discrete variables are permitted. Bit-in-word references on any word-oriented memory except %AI, including symbolic non-discrete memory, are also permitted.

#### Example of Set Coil, Reset Coil



The coil represented by E1 is turned ON when reference E2 or E6 is ON and is turned OFF when reference E5 or E3 is ON.

### 4.3.4 Transition Coils

PACSystems controllers provide four transition coils: PTCOIL, NTCOIL, POSCOIL, and NEGCOIL. POSCOIL and NEGCOIL are updated every time they are called.

PTCOIL and NTCOIL are updated once per CPU scan.

For examples showing the differences in the operation of the two types of transition coils, see *Examples Comparing PTCOIL and POSCOIL*

#### POSCOIL and NEGCOIL

##### Warning



- **These transition coil instructions should not be used in a parameterized block or user-defined function block (UDFB) with a parameter or member. In these cases, an R\_TRIG or F\_TRIG should be used instead.**
- **Do not override a transition coil by putting a force on its reference bit. If a transition coil is overridden, the coil has no effect on the bit, and if the override is then removed, the coil might be set ON for one sweep. . This can cause unexpected behavior in the Controller logic and in field devices attached to the Controller.**
- **Do not write to the reference bit of a transition coil using any other instruction or from an external device. Doing so will destroy the coil's one-shot nature and the coil may not behave as described.**
- **Do not use a transition contact with the same reference address used on a transition coil because the value of the transition bit, which stores the power flow value into the coil, will be affected.**

Positive Transition Coil (POSCOIL) 	Negative Transition Coil (NEGCOIL) 
<p>If:</p> <ul style="list-style-type: none"> <li>■ the <i>transition</i> bit is OFF, and</li> <li>■ the input power flow is ON,</li> </ul> <p>the POSCOIL sets the <i>reference</i> bit of its associated variable ON until the coil is executed again. When the coil is executed again, it sets its reference bit OFF.</p> <p><b>Note:</b> When the Positive Transition Coil sets its <i>reference</i> bit ON, it also sets its <i>transition</i> bit to ON. The next time the Positive Transition coil executes, it finds its transition bit set to ON and sets its reference bit to OFF.</p>	<p>If:</p> <ul style="list-style-type: none"> <li>■ the <i>transition</i> bit is OFF, and</li> <li>■ the input power flow input is OFF,</li> </ul> <p>the NEGCOIL sets the <i>reference</i> bit of its associated variable ON until the coil is executed again. When the coil is executed again, it sets its reference bit OFF.</p> <p><b>Note:</b> When the Negative Transition Coil sets its <i>reference</i> bit ON, it also sets its <i>transition</i> bit to ON. The next time the Negative Transition Coil executes, it finds the transition bit set to ON and sets its reference bit to OFF.</p>

### Operands for POSCOIL and NEGCOIL

Parameter	Description	Allowed Operands	Optional
BOOL_V	The variable associated with POSCOIL or NEGCOIL	BOOL variable: I, Q, M, T, G, SA, SB, SC, symbolic discrete variables, and I/O variable. Bit reference in BOOL variable: I, Q, M, T, G, SA, SB, SC	No

#### Example for POSCOIL and NEGCOIL

When reference E1 goes from OFF to ON, coils E2 and E3 receive power flow, turning E2 ON. When E1 goes from ON to OFF, power flow is removed from E2 and E3, turning coil E3 ON.



### PTCOIL and NTCOIL

Because the behavior of PTCOILs and NTCOILs is determined only by the current power flow into the coil and the previous power flow into the coil (i.e., the transition bit), it is not affected by writes to its associated BOOL variable by other coils or instructions in the logic. Therefore, many of the cautions that apply to POSCOILs and NEGCOILs do not apply to PTCOILs and NTCOILs.

#### Warning



- **PTCOIL and NTCOIL instructions should not be used in a parameterized block or user-defined function block (UDFB) with a parameter or member. In these cases, an R\_TRIG or F\_TRIG should be used instead.**
- **The transition bit of a given PTCOIL or NTCOIL is changed only once per CPU scan. Therefore, using a PTCOIL or NTCOIL in a block that can be called multiple times per scan can have adverse effects on all calls after the first one because the PTCOIL or NTCOIL cannot detect the transition on the second and subsequent calls.**
- **Do not override a transition coil by putting a force on its reference bit. If a transition coil is overridden, the coil has no effect on the bit, and if the override is then removed, the coil might be set ON for one sweep. . This can cause unexpected consequences in the Controller logic and in field devices attached to the Controller.**
- **Do not use a transition contact with the same reference address used on a transition coil because the value of the transition bit, which stores the power flow value into the coil, will be affected.**

	
Positive Transition Coil (PTCOIL)	Negative Transition Coil (NTCOIL)
<p>If:</p> <ul style="list-style-type: none"> <li>■ the <i>transition</i> bit is OFF, and</li> <li>■ the input power flow is ON</li> </ul> <p>the PTCOIL sets the <i>reference</i> bit and transition bit of its associated variable ON.</p>	<p>If:</p> <ul style="list-style-type: none"> <li>■ the <i>transition</i> bit is OFF, and</li> <li>■ the input power flow is OFF</li> </ul> <p>the NTCOIL sets the reference bit and transition bit of its associated variable ON.</p>
The <i>transition bit</i> depends on the value of the input power flow the last time the PTCOIL or NTCOIL was executed.	

**Notes:**

- As soon as a PTCOIL or NTCOIL is set to ON or OFF, it updates its transition bit.
- Multiple instances of PTCOIL and/or NTCOIL can be associated with the same BOOL variable, but the transition status of each instance of the PTCOIL or NTCOIL associated with the BOOL variable is unique, that is, it is tracked independently.
- The transition bit is non-retentive, that is, it is cleared to OFF when the CPU transitions from STOP Mode to RUN Mode. As a result, the first time a PTCOIL executes with its input power flow set to ON its associated BOOL variable will be set to ON.

**Operands for PTCOIL and NTCOIL**

Parameter	Description	Allowed Operands	Optional
BOOL_V	The variable associated with PTCOIL or NTCOIL	Variables in I, Q, M, T, SA, SB, SC, or G memories as well as symbolic discrete variables. In addition, bit-in-word references on any non-discrete memory (e.g., %R) or on symbolic non-discrete variables are allowed.	No

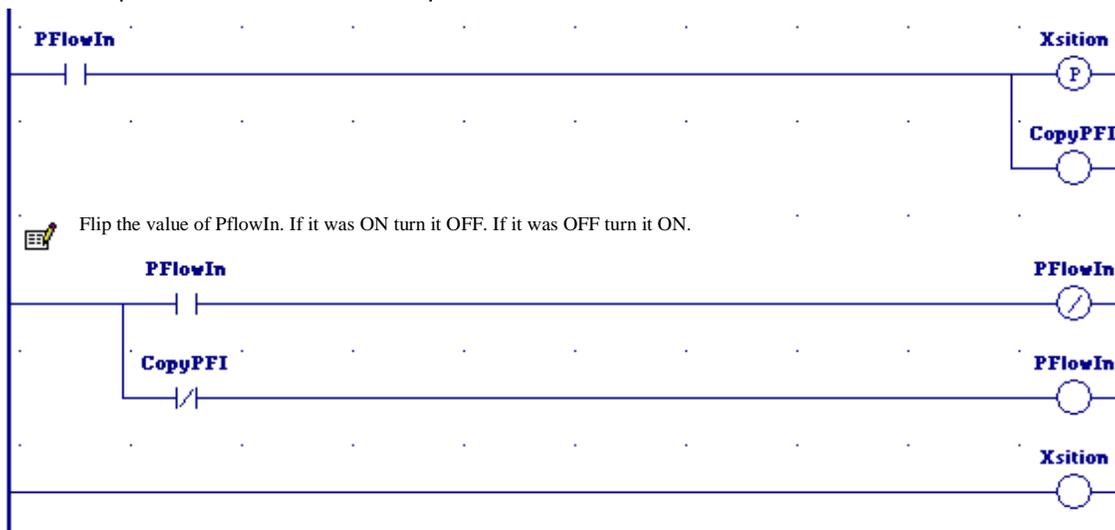
## Examples Comparing PTCOIL and POSCOIL

### PTCOIL

In the example below, the power flow into the PTCOIL alternates between OFF and ON. On the first sweep the power flow in is OFF, on the second sweep it is ON, and so forth. Each time the power flow into the PTCOIL changes from OFF to ON, the value of Xsition is turned ON. Therefore, on the first sweep, the PTCOIL turns Xsition OFF, on the second sweep it turns it ON, on the third sweep it turns it OFF, and so forth. Notice that the behavior of the PTCOIL is **not** affected by the presence of the fourth rung, which also writes to Xsition. PTCOIL behaves the same way when the fourth rung is removed.

### POSCOIL

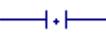
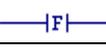
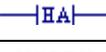
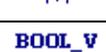
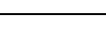
If a POSCOIL is used in place of the PTCOIL in the example below (keeping the rest of the logic identical and same alternation of power flow into the POSCOIL), the behavior of the logic will be different. The behavior of the POSCOIL **is** affected by the execution of the fourth rung, which writes to Xsition and changes both its value and its transition bit. In this example, the POSCOIL never turns Xsition ON. If the fourth rung is removed, POSCOIL will behave exactly as the PTCOIL behaves, turning Xsition OFF on the first sweep, ON on the second sweep, and so forth.



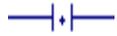
## 4.4 Contacts

A contact is used to monitor the state of a reference address. Whether the contact passes power flow depends on positive power flow into the contact, the state or status of the reference address being monitored, and the contact type.

A reference address is ON if its state is 1; it is OFF if its state is 0.

Contact	Display	Mnemonic	Contact Passes Power to Right...
Continuation Contact		CONTCON	if the preceding continuation coil is set ON
Fault Contact	<b>BWVAR</b> 	FAULT	if its associated BOOL or WORD variable has a point fault
High Alarm Contact	<b>WORDV</b> 	HIALR	if the high alarm bit associated with the analog (WORD) reference is ON
Low Alarm Contact	<b>WORDV</b> 	LOALR	if the low alarm bit associated with the analog (WORD) reference is ON
No Fault Contact	<b>BWVAR</b> 	NOFLT	if its associated BOOL or WORD variable does not have a point fault
Normally Closed Contact	<b>BOOLV</b> 	NCCON	if associated BOOL variable is OFF
Normally Open Contact	<b>BOOLV</b> 	NOCON	if associated BOOL variable is ON
Transition Contacts	<b>BOOLV</b> 	NEGCON	(negative transition contact) if BOOL reference transitions from ON to OFF. Updated every time it is called.
	<b>BOOL_V</b> 	NTCON	(negative transition contact) if BOOL reference transitions from ON to OFF. Updated once per scan.
	<b>BOOLV</b> 	POSCON	(positive transition contact) if BOOL reference transitions from OFF to ON. Updated every time it is called.
	<b>BOOL_V</b> 	PTCON	(positive transition contact) if BOOL reference transitions from OFF to ON. Updated once per scan.

### 4.4.1 Continuation Contact



A continuation contact continues the LD logic from the last previously-executed rung in the block that contained a continuation coil.

The flow state of the continuation contact is the same as the preceding executed continuation coil. A continuation contact has no associated variable.

Notes:

- If the flow of logic does not execute a continuation coil before it executes a continuation contact, the state of the continuation contact is no flow.
- The state of the continuation contact is cleared (set to no flow) each time a block begins execution.
- The continuation coil and the continuation contact do not use parameters and do not have associated variables.
- You can have multiple rungs with continuation contacts after a single continuation coil.
- You can have multiple rungs with continuation coils before one rung with a continuation contact.

## 4.4.2 Fault Contact

### BWVAR



A Fault contact (FAULT) detects faults in discrete or analog reference addresses, or locates faults (rack, slot, bus, module).

- To guarantee correct indication of module status, use the reference address (%I, %Q, %AI, %AQ) with the FAULT/NOFLT contacts.
- To locate a fault, use the rack, slot, bus, module fault locating system variable with a FAULT/NOFLT contact.

**Note:** The fault indication of a given module is cleared when the associated fault is cleared from the fault table.

- For I/O point fault reporting, you must enable point fault references in Hardware Configuration.

FAULT passes power flow if its associated variable or location has a point fault.

### Operands

Parameter	Description	Allowed Operands	Optional
BWVAR	The variable associated with the FAULT contact	variables in %I, %Q, %AI, and %AQ memories, and predefined fault-locating references	No

### 4.4.3 High and Low Alarm Contacts



The **high alarm contact** (HIALR) is used to detect a high alarm associated with an analog reference. Use of this contact and the low alarm contact must be enabled during CPU configuration.

A high alarm contact passes power flow if the high alarm bit associated with the analog reference is ON.

The **low alarm contact** (LOALR) detects a low alarm associated with an analog reference. Use of this contact must be enabled during CPU configuration.

A low alarm contact passes power flow if the low alarm bit associated with the analog reference is ON.

#### Operands

Parameter	Description	Allowed Operands	Optional
WORDV	The variable associated with the HIALR or LOALR contact	variables in AI and AQ memories	No

#### 4.4.4 No Fault Contact

##### **BWVAR**

—|NF|—

A No Fault (NOFLT) contact detects faults in discrete or analog reference addresses, or locates faults (rack, slot, bus, module). NOFLT passes power flow if its associated variable or location does not have a point fault.

- To guarantee correct indication of module status, use the reference address (%I, %Q, %AI, %AQ) with the FAULT/NOFLT contacts.
- To locate a fault, use the rack, slot, bus, module fault locating system variables with a FAULT/NOFLT contact.
- For I/O point fault reporting, you must configure your Hardware Configuration (HWC) to enable the PLC point faults.

**Note:** The fault indication of a given module is cleared when the associated fault is cleared from the fault table.

#### Operands

Parameter	Description	Allowed Operands	Optional
BWVAR	The variable associated with the NOFLT contact	variables in %I, %Q, %AI, and %AQ memories, and predefined fault-locating references	No

### 4.4.5 Normally Closed and Normally Open Contacts



A **normally closed contact** (NCCON) acts as a switch that passes power flow if the BOOLV operand is OFF (false, 0).

A **normally open contact** (NOCON) acts as a switch that passes power flow if the BOOLV operand is ON (true, 1).

#### Operands

Parameter	Description	Allowed Operands	Optional
BOOLV	<p>BOOLV may be a predefined system variable or a user-defined variable.</p> <p>NCCON:                      If BOOLV is ON, the normally closed contact does not pass power flow.                      If BOOLV is OFF, the contact passes power flow.</p> <p>NOCON:                      If BOOLV is ON, the normally open contact passes power flow.                      If BOOLV is OFF, the contact does not pass power flow.</p>	discrete variables in I, Q, M, T, S, SA, SB, SC, and G memories; symbolic discrete variables; bit-in-word references on variables in any non-discrete memory (e.g., %L) or on symbolic non-discrete variables.	No

### 4.4.6 Transition Contacts

PACSystems controllers provide four transition contacts: POSCON, NEGCON, PTCON and NTCON.

- The power flow out of the POSCON and NEGCON transition contacts is determined by the last write to the BOOL variable associated with the contact. The associated transition bit is updated every time the function is called.
- The power flow out of the PTCON and NTCON transition contacts is determined by the value that the associated BOOL variable had the last time the contact was executed. The associated transition bit is updated once per scan.

For an example showing the differences in the operation of the two types of transition contacts, see *Examples Comparing PTCON and POSCON*.

#### POSCON and NEGCON

---

#### Warning



- **These transition contact instructions should not be used in a parameterized block or user-defined function block (UDFB) with a parameter or member. In these cases, an R\_TRIG or F\_TRIG should be used instead.**
  - **Do not use POSCON or NEGCON transition contacts for references used with transition coils (also called one-shot coils) or with SET and RESET coils.**
  - **If a SETCOIL or RESETCOIL receives positive power flow and its associated variable is *not* overridden, the SETCOIL or RESETCOIL writes the expected result to the transition bit for the associated variable (that is, the transition bit is set if the variable's value is set from ON to OFF or is set from OFF to ON, and is cleared when its value remains the same). However, if the SETCOIL or RESETCOIL receives positive power flow and its associated variable is overridden, the SETCOIL or RESETCOIL causes the transition bit to be cleared.**
  - **Do not use a transition contact with the same reference address used on a transition coil because the value of the transition bit, which stores the power flow value into the coil, will be affected.**
-

<b>BOOLV</b> — ↑ — Positive Transition Contact POSCON	<b>BOOLV</b> — ↓ — Negative Transition Contact NEGCON
<p>POSCON starts passing power flow and continues passing power flow to the right only when all of the following conditions are met:</p> <ul style="list-style-type: none"> <li>■ the input power flow to POSCON is ON,</li> <li>■ the value of the associated variable is ON, and</li> <li>■ the <i>transition</i> bit for the associated variable is ON</li> </ul> <p>The POSCON's transition bit is set to ON when the variable associated with the POSCON transitions from OFF to ON.</p>	<p>NEGCON starts passing power flow and continues passing power flow to the right only when all of the following conditions are met:</p> <ul style="list-style-type: none"> <li>■ the input power flow to NEGCON is ON</li> <li>■ the value of the associated variable is OFF, and</li> <li>■ the <i>transition</i> bit for the associated variable is ON</li> </ul> <p>The NEGCON's transition bit is set to ON when the variable associated with the NEGCON transitions from ON to OFF.</p>
<p>The transition bit is set to OFF when the associated variable is written to while the POSCON or NEGCON contact is passing power flow, regardless of whether the value written is ON or OFF. Power flow stops when the transition bit is set to OFF.</p>	

Depending on the logic flow, writes to the POSCON's or NEGCON's associated variable can occur at different intervals within the Controller scan:

- multiple times during a Controller scan, resulting in the transition bit being ON for only a portion of the scan.
- several Controller scans apart, resulting in the transition bit being ON for more than one scan.
- once per scan, for example if the POSCON or NEGCON's associated variable is a %I input bit.

The source of the write is immaterial; it can be an output coil, a function block output, the input scan, an input interrupt, a data change from the program, or external communications. When the variable is written, the transition bit is immediately affected. The scan does not affect the transition bit. The only way to clear the transition bit is to write to the associated variable.

### Overrides

Overrides do not protect transition bits. If a write is attempted to an overridden point, the point's transition bit is cleared. As a result, any associated POSCON or NEGCON contacts will stop passing power flow.

### Transition to RUN Mode

- Variables that are non-retentive and not overridden will have values and transitions cleared to 0.
- Variables that are non-retentive and overridden will retain their values and transition bits.
- Variables that are retentive will retain their values and transition bits.

### Operands for POSCON and NEGCON

Parameter	Description	Allowed Operands	Optional
BOOLV	The variable associated with the transition contact	<p>BOOL variable: I, Q, M, T, S, SA, SB, SC, and G, symbolic discrete variables, I/O variables</p> <p>Bit reference in BOOL variable: I, Q, M, T, S, SA, SB, SC.</p>	No

**POSCON and NEGCON Example 1**

Coil E2 is turned ON when the value of the variable E1 transitions from OFF to ON. It stays ON until E1 is written to again, causing the POSCON to stop passing power flow.

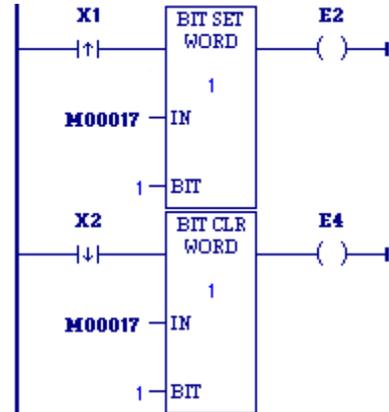
Coil E4 is turned ON when the value of the variable E3 transitions from ON to OFF. It stays ON until E3 is written to again, causing the NEGCON to stop passing power flow.



**POSCON and NEGCON Example 2**

Bit %M00017 is set by a BIT\_SET function and then cleared by a BIT\_CLR function. The positive transition contact X1 activates the BIT\_SET, and the negative transition X2 activates the BIT\_CLR.

The positive transition associated with bit %M00017 will be on until %M00017 is reset by the BIT\_CLR function. This occurs because the bit is only written when contact X1 goes from OFF to ON. Similarly, the negative transition associated with bit %M00017 will be ON until %M00017 is set by the BIT\_SET function.



## PTCON and NTCN



### Warning

**PTCON or NTCN instructions should not be used in a parameterized block or user-defined function block with a parameter or member. In these cases, an R\_TRIG or F\_TRIG should be used instead.**

**The transition bit of a given PTCON or NTCN is updated only once per CPU scan. Therefore, using a PTCON or NTCN in a block that can be called multiple times per scan may have adverse effects on all calls after the first one because the PTCON or NTCN cannot detect the transition on the second and subsequent calls.**

<b>BOOL_V</b> — P — Positive Transition Contact PTCON	<b>BOOL_V</b> — N — Negative Transition Contact NTCN
PTCON passes power flow to the right only when all of the following conditions are met: <ul style="list-style-type: none"> <li>■ The input power flow to PTCON is ON.</li> <li>■ The value of the BOOL variable associated with PTCON is ON.</li> <li>■ The transition bit associated with the PTCON is OFF</li> </ul>	NTCN passes power flow to the right only when all of the following conditions are met: <ul style="list-style-type: none"> <li>■ The input power flow to NTCN is ON.</li> <li>■ The value of the BOOL variable associated with NTCN is OFF.</li> <li>■ The transition bit associated with the NTCN is ON</li> </ul>
The transition bit depends on the value of the BOOL variable associated with this PTCON or NTCN when it was last executed.	

### Notes:

- As soon as a PTCON or NTCN is set to ON or OFF, it updates its transition bit.
- Multiple instances of PTCON and/or NTCN can be associated with the same BOOL variable, but the instance data of each instance of the PTCON or NTCN associated with the BOOL variable is unique, that is, it is tracked independently.
- Transition data is non-retentive, that is, it is cleared to OFF when the CPU transitions from STOP Mode to RUN Mode. As a result, the first time a PTCON executes with its input power flow set to ON and its associated BOOL variable also set to ON, it passes power flow to the right.

### Operands for PTCON and NTCN

Parameter	Description	Allowed Operands	Optional
BOOL_V	The variable associated with PTCON or NTCN contact	BOOL variable: I, Q, M, T, S, SA, SB, SC, and G memories, symbolic discrete variables, I/O variables. Bit reference in non-BOOL variable: R, AI, AQ, L, P, W, and on symbolic non-discrete variables.	No

Examples Comparing PTCON and POSCON

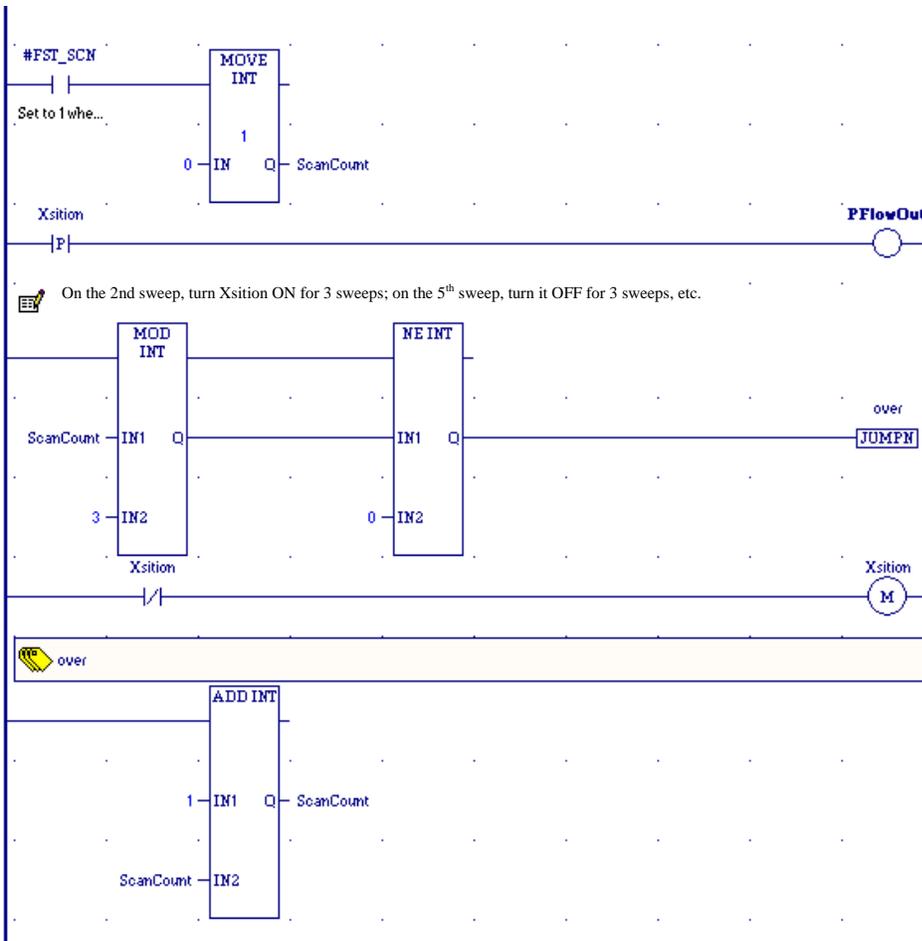
PTCON

The logic in the following example starts execution with all variables set to 0. Before the second sweep begins, the Xsition variable used on the PTCON instruction is set to 1. It retains that value for sweeps 2, 3, and 4. Then it is reset back to 0 before sweep 5 begins and retains its 0 value for sweeps 5, 6, and 7. This pattern repeats. The PTCON instruction in rung two passes power flow on the 2nd sweep, the 8<sup>th</sup> sweep, the 14<sup>th</sup> sweep, and so on. These are sweeps where the Xsition variable's value becomes a 1, after having been a 0 on the previous sweep. On all other sweeps, the PTCON instruction does not pass power flow.

POSCON

If a POSCON is used in place of the PTCON in the following example (keeping the rest of the logic identical), the same alternation of the Xsition variable's value occurs. The POSCON instruction passes power flow on sweeps 2, 3, and 4; then again on sweeps 8, 9, and 10; and so forth. The POSCON's behavior is dependent on Xsition's transition bit. Since Xsition's value is written once and then simply retained for three sweeps, its transition bit retains its same value for three sweeps. Thus the POSCON will pass or not pass power flow for three sweeps in a row. Note that if Xsition's value is actually written on each sweep, the POSCON and the PTCON behave identically.

Logic Example Using PTCON

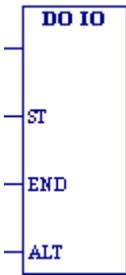


## 4.5 Control Functions

The control functions limit program execution and change the way the CPU executes the application program.

Function	Mnemonic	Description
Do I/O	DO_IO	For one scan, immediately services a specified range of inputs or outputs. (All inputs or outputs on a module are serviced if any reference locations on that module are included in the DO I/O function. Partial I/O module updates are not performed.) Optionally, a copy of the scanned I/O can be placed in internal memory, rather than at the real input points.
Drum	DRUM	Provides predefined On/Off patterns to a set of 16 discrete outputs in the manner of a mechanical drum sequencer.
Edge Detectors	F_TRIG R_TRIG	Detect the changing state of a Boolean signal.
For Loop	FOR_LOOP EXIT_FOR END_FOR	For loop. Repeats the logic between the FOR_LOOP instruction and END_FOR instruction a specified number of times or until EXIT_FOR is encountered.
Mask I/O Interrupt	MASK_IO_INTR	Mask or unmask an interrupt from an I/O module when using I/O variables. If not using I/O variables, use SVC_REQ 17: Mask/Unmask I/O Interrupt, described in Chapter 6.
Proportional Integral Derivative Control	PID_ISA PID_IND	Provides two PID (Proportional/Integral/Derivative) closed-loop control algorithms: <p style="margin-left: 40px;">Standard ISA PID algorithm (PID_ISA)  Independent term algorithm (PID_IND)</p> <b>Note:</b> For details, refer to Chapter 7.
Read Switch Position	SWITCH_POS	Reads position of the Run/Stop switch and the mode for which the switch is configured.
Scan Set IO	SCAN_SET_IO	Scans the IO of a specified scan set.
Service Request	SVC_REQ	Requests a special PLC service. <b>Note:</b> For details, refer to Chapter 6.
Suspend IO	SUS_IO	Suspends for one sweep all normal I/O updates, except those specified by DO I/O instructions.
Suspend or Resume I/O Interrupt	SUSP_IO_INTR	Suspend or resume an I/O interrupt when using I/O variables. If not using I/O variables, use SVC_REQ 32: Suspend/Resume I/O Interrupt, described in Chapter 6.

### 4.5.1 Do I/O



When the DO I/O (DO\_IO) function receives power flow, it updates inputs or outputs for one scan while the program is running. You can also use DO\_IO to update selected I/O during the program in addition to the normal I/O scan.

You can use DO\_IO in conjunction with a Suspend I/O (SUS\_IO) function, which stops the normal I/O scan. For details, refer to *Suspend I/O*.

If input references are specified, DO\_IO allows the most recent values of inputs to be obtained for program logic. If output references are specified, DO I/O updates outputs based on the most current values stored in I/O memory. I/O is serviced in increments of entire I/O modules; the PLC adjusts the references, if necessary, while DO\_IO executes. DO\_IO does not scan I/O modules that are not configured.

DO\_IO continues to execute until all inputs in the selected range have reported or all outputs have been serviced on the I/O modules. Program execution then returns to the function that follows the DO\_IO.

If the range of references includes an option module (HSC, APM, etc.), all the input data (%I and %AI) or all the output data (%Q and %AQ) for that module are scanned. The ALT parameter is ignored while scanning option modules.

DO\_IO passes power to the right whenever it receives power unless:

- Not all references of the type specified are present within the selected range.
- The CPU is not able to properly handle the temporary list of I/O created by the function.
- The range specified includes I/O modules that are associated with a *Loss of I/O* fault.



#### Warning

**If DO\_IO is used with timed or I/O interrupts, transition contacts associated with scanned inputs may not operate as expected.**

**Note:** The Do I/O function skips modules that do not support DO\_IO scanning:

IC693BEM331	90-30 Genius Bus Controller
IC694BEM331	RX3i Genius Bus Controller
IC693BEM341	90-30 2.5 GHz FIP Bus Controller
IC693DNM200	90-30 DeviceNet Master
IC695PBM300	RX3i PROFIBUS Master
IC695PBS301	RX3i PROFIBUS Slave
IC687BEM731	90-70 Genius Bus Controller
IC697BEM731	90-70 Standard Width Genius Bus Controller

### Do I/O for Inputs

When DO\_IO receives power flow and input references are specified, the PLC scans input points from the starting reference (ST) to the ending reference (END). If a reference is specified for ALT, a copy of the new input values is placed in memory beginning at that reference, and the real input values are not updated. ALT must be the same size as the reference type scanned. If a discrete reference is used for ST and END, ALT must also be discrete.

If no reference is specified for ALT, the real input values are updated. This allows inputs to be scanned one or more times during the program execution portion of the CPU scan.

### Do I/O for Outputs

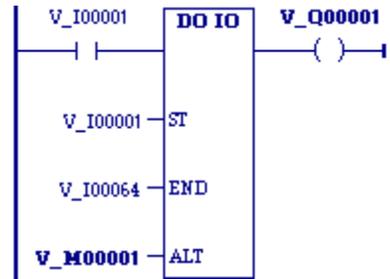
When DO\_IO receives power flow and output references are specified, the PLC writes to the output points. If no value is specified in ALT, the range of outputs written to the output modules is specified by the starting reference (ST) and the ending reference (END). If outputs should be written to the output points from internal memory other than %Q or %AQ, the beginning reference is specified for ALT and the end reference is automatically calculated from the length of the END—ST range.

### Operands

Parameter	Description	Allowed Operands	Optional
ST	<p>The starting address of the set of input or output points or words to be serviced. ST and END must be in the same memory area.</p> <ul style="list-style-type: none"> <li>■ If ST and END are placed in BOOL memory, ST must be byte-aligned. That is, its reference address must start at (8n+1), for example, %I01, %Q09, %Q49.</li> <li>■ If ST and END are mapped to analog memory, they can have the same reference address.</li> <li>■ If ST is mapped to an I/O variable, the same I/O variable must also be assigned to the END parameter, and the entire module is scanned.</li> </ul>	I, Q, AI, AQ, I/O Variable	No
END	<p>The address of the end bit of input or output points or words to be serviced. Must be in the same memory area as ST.</p> <ul style="list-style-type: none"> <li>■ If ST and END are placed in BOOL memory, END's reference address must be 8n, for example, %I08, %Q16.</li> <li>■ If ST and END are mapped to analog memory, they can have the same reference address.</li> <li>■ If ST is mapped to an I/O variable, the same I/O variable must also be assigned to the END parameter, and the entire module is scanned.</li> </ul>	I, Q, AI, AQ, I/O Variable	No
ALT	<p>For an input scan, ALT specifies the address to store scanned input point/word values. For an output scan, ALT specifies the address to get output point/word values from, to send to the I/O modules.</p> <p><b>Note:</b> ALT can be a WORD only if ST and END are in analog memory.</p>	I, Q, M, T, G, R, AI, AQ	Yes

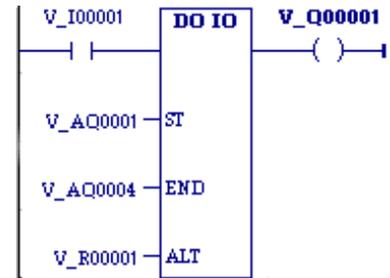
**Example - Do I/O for Inputs**

When DO\_IO receives power flow, the PLC scans references %I0001–64 and %Q0001 is turned on. A copy of the scanned inputs is placed in internal memory from %M0001-64. Because a reference is specified for ALT, the real inputs are not updated. This allows the current values of inputs to be compared with their values at the beginning of the scan. This form of DO\_IO allows input points to be scanned one or more times during the program execution portion of the CPU scan.

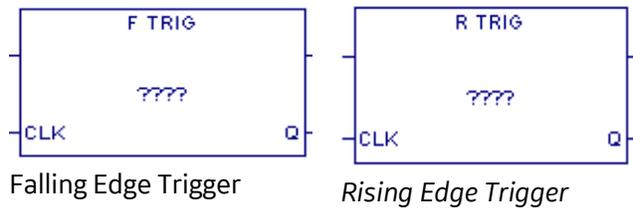


**Example - Do I/O for Outputs**

Because a reference is entered for ALT, the values at %AQ001–004 are *not* written to output modules. When DO\_IO receives power flow, the PLC writes the values from references %R0001-0004 to the analog output modules and %Q0001 is turned on.



### 4.5.2 Edge Detectors



These function blocks detect the changing state of a Boolean signal and produce a single pulse when an edge is detected.

When transitional instructions, such as *Transition Coils* or *Transition Contacts*, are used inside a function block, there is a problem when the same function block is called more than once per scan. The first call executes the transition correctly but subsequent calls do not because they see the state as adjusted from the first call. The rising and falling edge trigger instructions solve this problem. These instructions have their own instance data that can be a member or an input of the function block so that the transition state follows that of the function block instance and not the function block.

If an edge detector function block is used within a UDFB, its instance data must be a member variable of the UDFB.

#### Operands

Parameter	Description	Allowed Operands	Optional
????	Instance data for function block. This is a structure variable, described below.	F_TRIG, R_TRIG	No
CLK	Input to be monitored for a change in state.	All	Yes
Q	Edge detection output.	Must be flow in LD. In other languages all types allowed except S, SA, SB, SC and constants.	Yes

#### Instance Data Structure

These elements cannot be published or written to.

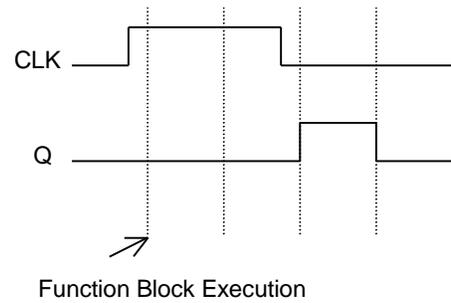
Element Name	Type	Description
CLK	BOOL	Edge detection input. Not accessible in user logic.
Q	BOOL	Edge detection output. Accessible in user logic. Read only.
STATE	BOOL	Internal value. Not accessible in user logic.
ENO	BOOL	Enable Output. User logic can access as read-only.

### F\_TRIG Operation

When the CLK input goes from true to false, the output Q is true for one function block instance execution. The output Q then remains false until a new falling edge is detected.

When the Controller transitions from STOP Mode to RUN Mode and the CLK input is false and the instance memory is non-retentive, the output Q is true after the function block's first execution. After the next execution, the output is false.

The F\_TRIG output Q will be true for one function block instance execution at a STOP Mode to RUN Mode transition after the first download, whether or not instance memory is retentive.

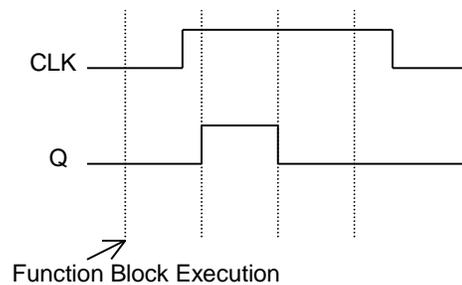


### R\_TRIG Operation

When the CLK input transitions from false to true, the output Q is true for one function block execution. The output Q then remains false until a new rising edge is detected.

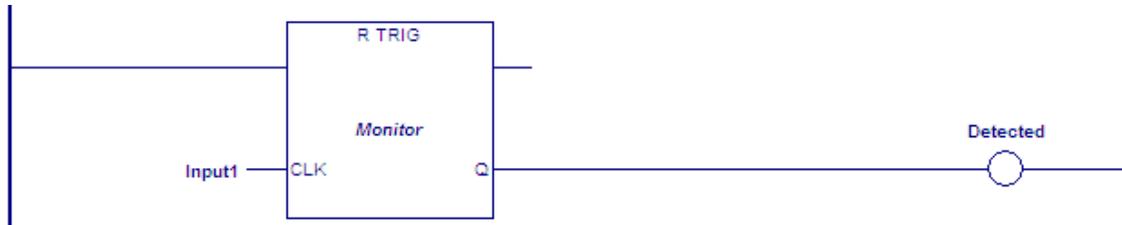
When the Controller transitions from STOP Mode to RUN Mode and the CLK input is true and the instance memory is non-retentive, the output Q is set to true after the function block's first execution. After the second execution, the output is false.

If the CLK input is initialized on, the R\_TRIG output Q will be true for one function block instance execution at a STOP Mode to RUN Mode transition after the first download, whether or not instance memory is retentive.

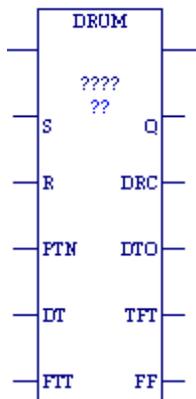


### Example

In the following example, when Input1 transitions from false to true, the coil, Detected, is set ON for one function block execution. The output Q remains false until a new rising edge is detected.



### 4.5.3 Drum



The Drum function operates like a mechanical drum sequencer, which steps through a set of potential output bit patterns and selects one based on inputs to the function. The selected value is copied to a group of 16 discrete output references.

When the Drum function receives power flow, it copies the contents of a selected reference to the Q reference.

Power flow to the R (Reset) input or to the S (Step) input selects the reference to be copied.

The function passes power to the right only if it receives power from the left and no error condition is detected.

The DTO (Dwell Timeout Output) bit is cleared the first time the drum is in a new step. This is true:

- Whether the drum is introduced to a new step by changing the Active Step or by using the S (Step) Input.
- Regardless of the DT (Dwell Time array) value associated with the step (even if it is 0).
- During the first sweep the Active Step is initialized.

#### Using Drum in Parameterized Blocks

The Drum dwell and fault timer features use an internal timer that is implemented in the same manner as for the OFDT, ONDTR, and TMR timers. Therefore, special care must be taken when programming Drum in parameterized blocks. Drum functions in parameterized blocks can be programmed to track true real-time as long as the guidelines and rules below are followed. If the guidelines and rules described here are not followed, the operation of the Drum function in parameterized blocks is undefined.

**Note:** These rules are not enforced by the programming software. It is your responsibility to ensure these rules are followed.

The best use of a Drum function is to invoke it with a particular reference address exactly one time each scan. With parameterized blocks, it is important to use the appropriate reference memory with the Drum function and to call the parameterized block an appropriate number of times.

#### Finding the Source Block

The source block is either the `_MAIN` block or the lowest logic block of type Block that appears above the parameterized block in the call tree. To determine the source block for a given parameterized block, determine which block invoked that parameterized block. If the calling block is `_MAIN` or of type Block, it is the source block. If the calling block is any other type (parameterized block or function block), apply the same test to the block that invoked this block. Continue back up the call tree until the `_MAIN` block or a block of type Block is found. This is the source block for the parameterized block.

## Programming Drum in Parameterized Blocks

Different guidelines and rules apply depending on whether you want to use the parameterized block in more than one place in your program logic.

### Parameterized block called from one block

If your parameterized block that contains a Drum function will be called from only one logic block, follow these rules:

1. Call the parameterized block exactly one time per execution of its source block.
2. Choose a reference address for the Drum control block that will not be manipulated anywhere else. The reference address may be %R, %P, %L, %W, or symbolic.

**Note:** %L memory is the same %L memory available to the source block of type Block. %L memory corresponds to %P memory when the source block is `_MAIN`.

### Parameterized block called from multiple blocks

When calling the parameterized block from multiple blocks, it is imperative to separate the Drum reference memory used by each call to the parameterized block. Follow these rules and guidelines:

1. Call the parameterized block exactly one time per execution of each source block that it appears in.
2. Choose a %L reference or parameterized block formal parameter for the Drum control block. Do not use a %R, %P, %W, or symbolic memory reference.

#### Notes:

- The strongly recommended choice is a %L location, which is inherited from the parameterized block's source block. Each source block has its own %L memory space except the `_MAIN` block, which has a %P memory area instead. When the `_MAIN` block calls another block, the %P mappings from the `_MAIN` block are accessed by the called block as %L mappings.
- If you use a parameterized block formal parameter (word array passed-by-reference), the actual parameter that corresponds to this formal parameter must be a %L, %R, %P, %W, or symbolic reference. If the *actual parameter* is a %R, %P, %W, or symbolic reference, a unique reference address must be used by each source block.

## Recursion

If you use recursion (that is, if you have a block call itself either directly or indirectly) and your parameterized block contains a Drum function, you must follow two additional rules:

- Program the source block so that it invokes the parameterized block before making any recursive calls to itself.
- Do not program the parameterized block to call itself directly.

## Using Drum in UDFBs

UDFBs are user-defined logic blocks that have parameters and instance data. For details on these and other types of blocks, refer to Chapter 2.

When a Drum function is present inside a UDFB, and a member variable is used for the control block of a Drum function, the behavior of the Drum function may not match your expectations. If multiple instances of the UDFB are called during a logic sweep, only the first-executed instance will update the timer in the Drum function. If a different instance is then executed, the timer value will remain unchanged.

In the case of multiple calls to a UDFB during a logic scan, only the first call will add elapsed time to its timer functions. This behavior matches the behavior of the Drum function timer in a normal program block.

**Example**

A UDFB is defined that uses a member variable for a Drum function block. Two instances of the function block are created: Drum\_A and Drum\_B. During each logic scan, both Drum\_A and Drum\_B are executed. However, only the member variable in Drum\_A is updated and the member variable in Drum\_B always remains at 0.

**Operands for Drum**

Parameter	Description	Allowed Operands	Optional
????	(Control Block) The beginning address of a five-word array that contains the Drum Sequencer's control block. The contents of the control block are described below.	R, P, L, W, Symbolic	No
??	(Length) Value between 1 and 128 that specifies the number of steps.	Constant	No
S	Step input. Used to go one step forward in the sequence. When the function receives power flow and S makes an OFF to ON transition, the Drum Sequencer moves one step. When R (Reset) is active, the function ignores S.	flow	No
R	Reset input. Used to select a specific step in the sequence. When the DRUM function and Reset both receive power flow, DRUM copies the Preset Step value in the Control Block to the Active Step reference in the Control Block. Then the function copies the value in the Preset Step reference to the Q reference bits. When R is active, the function ignores S.	flow	No
PTN	(Pattern) The starting address of an array of words. The number of words is specified by the Length (??) operand. Each word represents one step of the Drum Sequencer. The value of each word represents the desired combination of outputs for a particular value of the Active Step word in the control block. The first element corresponds to an Active Step value of 1; the last element corresponds to an Active Step value of Length. The programming software does not create an array for you. You must ensure you have enough memory for PTN.	All except constant and S, SA—SC numerical data.	No
DT	(Dwell Time ) If you use the DT operand, you must also use the DTO operand and vice-versa. The DT operand is the starting address of Length words of memory, where Length is the number of steps. Each DT word corresponds to one word of PTN. The value of each word represents the dwell time for the corresponding step of the Drum Sequencer in 0.1 second units. When the dwell time expires for a given step the DTO bit is set. If a Dwell Time is specified, the drum cannot sequence into its next step until the Dwell Time has expired. The programming software does not create an array for you. You must ensure you allocate enough memory for DT.	All except S, SA, SB, SC and constant	Yes

Parameter	Description	Allowed Operands	Optional
FTT	(Fault Timeout) If you use the FTT operand, you must also use the TFT operand, and vice-versa. The FTT operand is the starting address of Length words of memory, where Length is the number of steps. Each FTT word corresponds to one word of PTN. The value of each word represents the fault timeout for the corresponding step of the Drum Sequencer in 0.1 second units. When the fault timeout has expired the Fault Timeout bit is set. The programming software does not create an array for you. You must ensure you allocate enough memory for FTT.	All except S, SA, SB, SC and constant	Yes
Q	A word of memory containing the element of the PTN that corresponds to the current Active Step.	All except S and constant	No
DRC	(Drum Coil) Bit reference that is set whenever the function is enabled and Active Step is <b>not</b> equal to Preset Step.	All except S	Yes
DTO	(Dwell Timeout) If you use the DTO operand, you must also use DT and vice-versa. This bit reference is set if the dwell time for the current step has expired.	All except S and constant	Yes
TFT	(Timeout Fault ) If you use the TFT operand, you must also use the FTT operand and vice-versa. Bit reference that is set if the drum has been in a particular step longer than the step's specified Fault Timeout.	All except S and constant	Yes
FF	(First Follower ) The starting address of (Length/8+1) bytes of memory, where Length is the number of steps. If MOD (Length/8+1)>0, FF has (Length/8+1) bytes. Each bit in the bytes of FF corresponds to one word of PTN. No more than one bit in the FF bytes is ON at any time, and that bit corresponds to the value of the Active Step. The first bit corresponds to an Active Step value of one. The last used bit corresponds to an Active Step value of Length.	All except S and constant	Yes

### Control Block for the Drum Sequencer Function

The control block for the Drum Sequencer function contains information needed to operate the Drum Sequencer.

address	Active Step
address + 1	Preset Step
address + 2	Step Control
address + 3	Timer Control

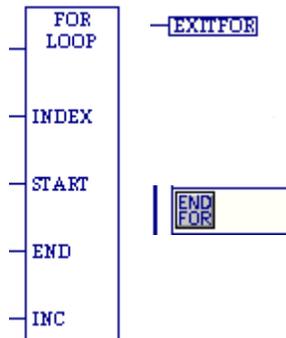
**Active Step** The active step value specifies the element in the Pattern array to copy to the Out output memory location. This is used as the array index into the Pattern, Dwell Time, Fault Timeout, and First Follower arrays.

**Preset Step** A word input that is copied to the Active Step output when the Reset is On.

**Step Control** A word that is used to detect Off to On transitions on both the Step input and the Enable input. The Step Control word is reserved for use by the function, and **must not be written to**.

**Timer Control** Two words of data that hold values needed to run the timer. These values are reserved for use by the function and **must not be written to**.

### 4.5.4 For Loop



A FOR loop repeats rung logic a specified number of times while varying the value of the INDEX variable in the loop.

A FOR loop begins with a FOR\_LOOP instruction and ends with an END\_FOR instruction.

The logic to be repeated must be placed between the FOR and END\_FOR instructions.

The optional EXIT\_FOR instruction enables you to exit the loop if a condition is met before the FOR loop ends normally.

When FOR\_LOOP receives power flow, it saves the START, END, and INC (Increment) operands and uses them to evaluate the number of times the rungs between the FOR\_LOOP and its END\_FOR instructions are executed. Changing the START and END operands while the FOR loop is executing does not affect its operation.

When an END\_FOR receives power flow, the FOR loop is terminated and power flow jumps directly to the statement following the END\_FOR instruction.

There can be nothing after the FOR\_LOOP instruction in the rung and the FOR\_LOOP instruction must be the last instruction to be executed in the rung. An EXIT\_FOR statement can be placed only between a FOR instruction and an END\_FOR instruction. The END\_FOR statement must be the only instruction in its rung.

A FOR\_LOOP can assign decreasing values to its index variable by setting the increment to a negative number. For example, if the START value is 21, the END value is 1, and the increment value is -5, the statements of the FOR loop are executed five times, and the index variable is decremented by 5 in each pass. The values of the index variable will be 21, 16, 11, 6, and 1.

When the START and END values are set equal, the statements of the FOR loop are executed only once.

When START cannot be incremented or decremented to reach the END, the statements within the FOR loop are not executed. For example, if the value of START is 10, the value of END is 5, and the INCREMENT is 1, power flow jumps directly from the FOR statement to the statement after the END\_FOR statement.

**Note:** If the FOR\_LOOP instruction has power flow when it is first tested, the rungs between the FOR and its corresponding END\_FOR statement are executed the number of times initially specified by START, END, and INCREMENT. This repeated execution occurs on a single sweep of the PLC and may cause the watchdog timer to expire if the loop is long.

Nesting of FOR loops is allowed, but it is restricted to five FOR/END\_FOR pairs. Each FOR instruction must have a matching END\_FOR statement following it.

Nesting with JUMPs and MCRs is allowed, provided that they are properly nested. MCRs and ENDMCRs must be completely within or completely outside the scope of a FOR\_LOOP/END\_FOR pair. JUMPs and LABEL instructions must also be completely within or completely outside the scope of a FOR\_LOOP/END\_FOR pair. Jumping into or out of the scope of a FOR/END\_FOR is not allowed.

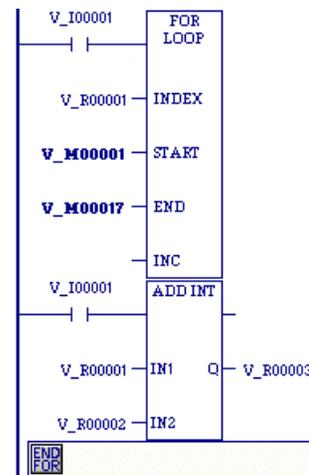
## Operands

Only the FOR\_LOOP function requires operands.

Parameter	Description	Allowed Operands	Optional
INDEX	The index variable. When the loop has completed, this value is undefined. <b>Note:</b> Changing the value of the index variable within the scope of the FOR loop is not recommended.	All except constants, flow, and variables in %S - %SC	No
START	The index start value.	All except variables in %S - %SC	No
END	The index end value.	All except variables in %S - %SC	No
INC	The increment value. (Default: 1.)	Constants	Yes

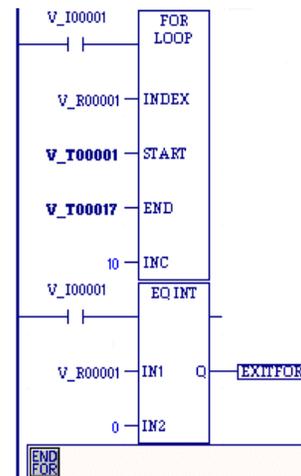
### For Loop Example 1

The value for %M00001 (START) is 1 and the value for %M00017 (END) is 10. The INDEX (%R00001) increments by the value of the INC operand (which is assumed to be 1 when omitted) starting at 1 until it reaches the ending value 10. The ADD function of the loop is executed 10 times, adding the current value of I1 (%R00001), which will vary from 1 to 10, to the value of I2 (%R00002).

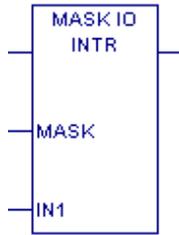


### For Loop Example 2

The value for %T00001 (START) is -100 and the value for %T00017 (END) is 100. The INDEX (%R00001) increments by tens, starting at -100 until it reaches its end value of +100. The EQ function of the loop tries to execute 21 times, with the INDEX (%R00001) being equal to -100, -90, -80, -70, -60, -50, -40, -30, -20, -10, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100. However, when the INDEX (%R00001) is 0, the EXIT statement is enabled and power flow jumps directly to the statement after the END\_FOR statement.



### 4.5.5 Mask I/O Interrupt



Mask or unmask an interrupt from an I/O board when using I/O variables. If not using I/O variables, use SVC\_REQ 17.

When the interrupt is masked, the CPU processes the interrupt but does not schedule the associated logic for execution. When the interrupt is unmasked, the CPU processes the interrupt and schedules the associated logic for execution.

When the CPU transitions from STOP Mode to RUN Mode, the interrupt is unmasked

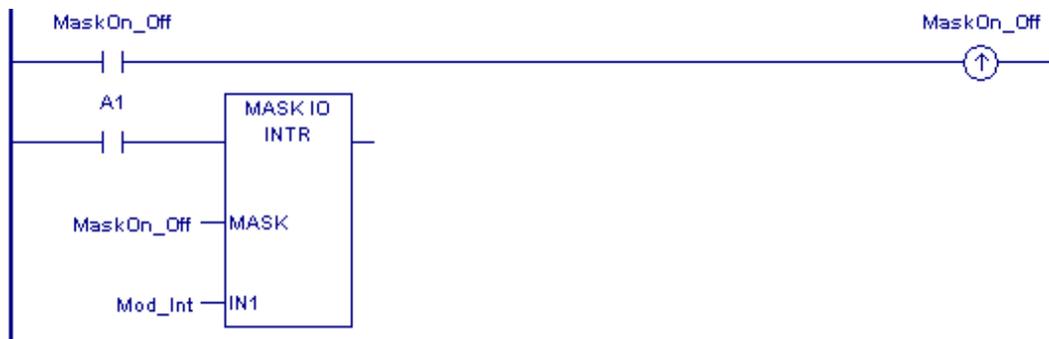
The function passes power to the right when it executes successfully.

#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
MASK	Selects unmask or mask operation. Unmask=0; Mask=1	BOOL variable or Bit reference in non-discrete memory	data flow, I, Q, M, T, G, S, SA, SB, SC, R, P, L, AI, AQ, W, symbolic, I/O variable	No
IN1	The interrupt trigger to be masked or unmasked. <ul style="list-style-type: none"> <li>■ The I/O board must be a supported input module.</li> <li>■ The reference address specified must correspond to a valid interrupt trigger reference.</li> <li>■ The interrupt for the specified channel must be enabled in the configuration.</li> </ul>	BOOL or WORD variable	I, Q, M, T, G, R, P, L, AI, AQ, W, I/O variable	No

#### Example

In the following example, the variable Mod\_Int is mapped to an I/O point on a hardware module and is configured as an I/O interrupt to a program block. When the BOOL variable MaskOn\_Off transitions from OFF to ON and A1 is set to ON, the interrupt Mod\_Int is masked (not executed) for one scan.



### 4.5.6 Read Switch Position

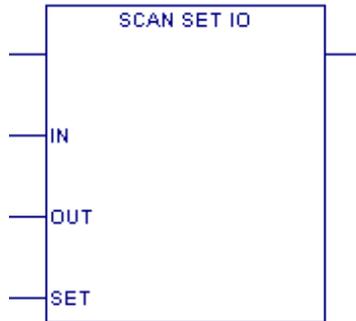


Read Switch Position (SWITCH\_POS) allows the logic to read the current position of the RUN/STOP switch, as well as the mode for which the switch is configured.

#### Operands

Parameter	Description	Allowed Operands	Optional
POS	Memory location at which to write current switch position value. 1 - RUN I/O Enabled 2 - RUN Outputs Disabled 3 - STOP Mode	All except S, SA, SB, SC	No
MODE	Memory location to which switch configuration value is written. 0 - Switch configuration not supported 1 - Switch controls RUN/STOP mode 2 - Switch not used, or is used by the user application 3 - Switch controls both memory protection and RUN/STOP mode 4 - Switch controls memory protection	All except S, SA, SB, SC	No

### 4.5.7 Scan Set IO



The Scan\_Set\_IO function scans the I/O of a specified scan set number. (Modules can be assigned to scan sets in hardware configuration.) You can specify whether the Inputs and/or Outputs of the associated scan set will be scanned.

Execution of this function block does not affect the normal scanning process of the corresponding scan set. If the corresponding scan set is configured for non-default Number of Sweeps or Output Delay settings, they remain in effect regardless of how many executions of the Scan Set IO function occur in any given sweep.

The Scan Set IO function skips those modules that do not support *Do I/O* scanning.

#### Operands for SCAN\_SET\_IO

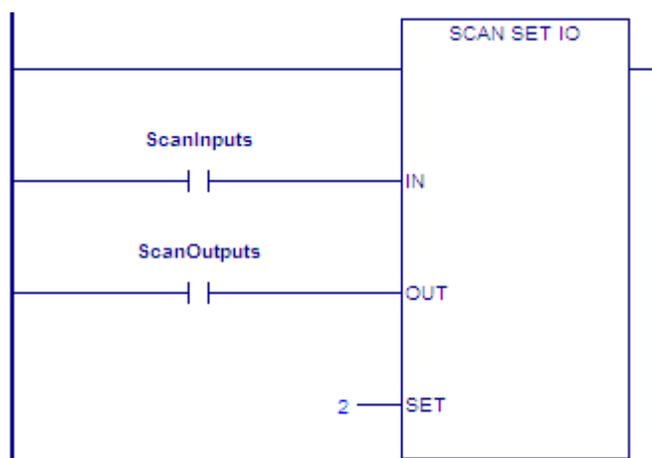
Parameter	Description	Allowed Types	Allowed Operands	Optional
IN	If true the inputs will be scanned.	BOOL variable or bit reference in a non-BOOL variable	Power flow	No
OUT	If true the outputs will be scanned.	BOOL variable or bit reference in a non-BOOL variable	Power flow	No
SET	Number of the scan set to be scanned. Scan sets are specified in the CPU hardware configuration and assigned to modules in the module hardware configuration.	UINT	All except %S memory types.	No
ENO	Energized when all arguments to the function are valid and there are no errors in scanning.	BOOL variable or bit reference in a non-BOOL variable	Power flow.	Yes

#### Example

By using the Scan Set IO function block in an interrupt block, you can create a custom I/O scan. For example, two Scan Set IO function blocks can be used in an interrupt block to scan the inputs of a scan set at the beginning of the block and the outputs of the same scan set at the end of the block.

In the example at right:

- When ScanInputs is ON, input data for all I/O modules assigned to Scan Set 2 is updated.
- When ScanOutputs is ON, output data for all I/O modules assigned to Scan Set 2 is updated



### 4.5.8 Suspend I/O



The Suspend I/O (SUS\_IO) function stops normal I/O scans from occurring for one CPU sweep. During the next output scan, all outputs are held at their current states. During the next input scan, the input references are not updated with data from inputs. However, during the input scan portion of the sweep, the CPU verifies that Genius bus controllers have completed their previous output updates.

**Note:** The PACSystems SUS\_IO function suspends analog and discrete I/O, whether integrated I/O or Genius I/O. It does not suspend Ethernet Global Data. For details, refer to *PACSystems RX7i, RX3i and RSTi-EP TCP/IP Ethernet Communications User Manual*, GFK-2224.

When SUS\_IO receives power flow, all I/O servicing stops except that provided by DO\_IO functions.



#### Warning

**If SUS\_IO were placed at the left rail of the ladder, without enabling logic to regulate its execution, no regular I/O scan would ever be performed.**

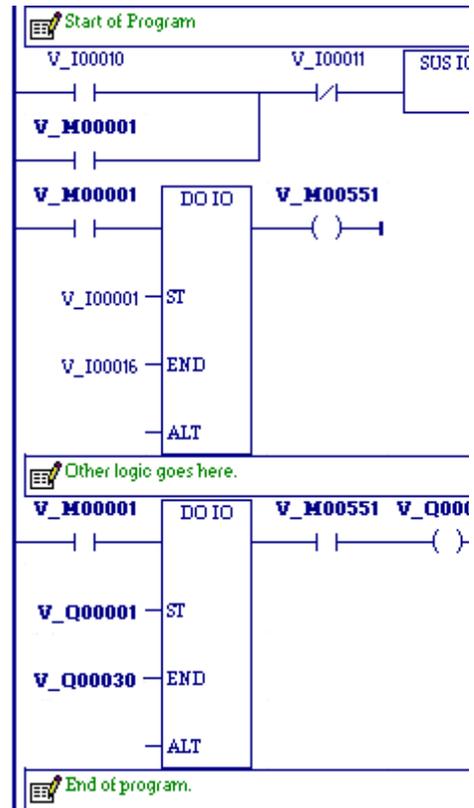
**SUS\_IO passes power flow to the right whenever it receives power.**

**Example**

The example at right shows a SUS\_IO function and a DO\_IO function used to stop I/O scans, then cause certain I/O to be scanned from the program.

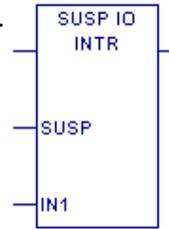
Inputs %I00010 and %I00011 form a latch circuit with the contact from %M00001. This keeps the SUS\_IO function active on each sweep until %I00011 goes on. If this input were not scanned by DO\_IO after SUS\_IO went active, SUS\_IO could only be disabled by powering down the PLC. Output %Q00002 is set when both DO\_IO functions execute successfully. The rung is constructed so that both DO\_IO functions execute even if one does not set its OK output. With normal I/O suspended, output %Q00002 is not updated until a DO\_IO function with %Q00002 in its range executes. This does not occur until the sweep after the setting of %Q00002. Outputs that are set after a DO\_IO function executes are not updated until another DO\_IO function executes, typically in the next sweep. Because of this delay, most programs that use SUS\_IO and DO\_IO place the SUS\_IO function in the first rung of the program, the DO\_IO function that processes inputs in the next rung, and the DO\_IO function that processes outputs in the last rung.

The range of the DO\_IO function doing outputs is %Q00001 through %Q00030. If the module in this range were a 32-point module, the DO\_IO function would actually perform a scan of the entire module. A DO\_IO function will not break the scan in the middle of an I/O module.



### 4.5.9 Suspend or Resume I/O Interrupt

Suspend or resume an I/O interrupt when using I/O variables.  
 If not using I/O variables, use SVC\_REQ 32.



The function executes successfully and passes power to the right unless:

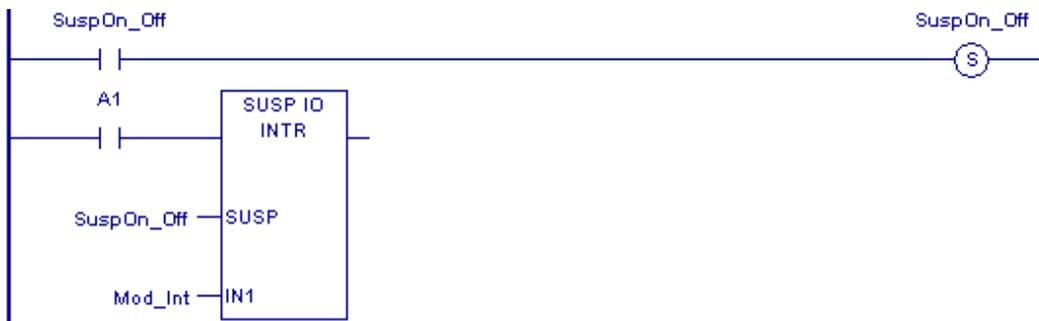
- The I/O module associated with the interrupt trigger specified in IN1 is not supported.
- The reference address specified does not correspond to a valid interrupt trigger reference.
- The specified channel does not have its interrupt enabled in the configuration.

#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
SUSP	Selects a suspend or resume operation. 1 (ON)=suspend 0 (OFF)=resume	BOOL variable or bit reference in a non-BOOL variable	data flow, I, Q, M, T, G, S, SA, SB, SC, R, P, L, discrete symbolic, I/O variable	No
IN1	The interrupt trigger to be suspended or resumed.	BOOL or WORD variable	I, Q, M, T, G, R, P, L, AI, AQ, W, I/O variable	No

#### Example

In the following example, the variable Mod\_Int is mapped to an I/O point on a hardware module and is configured as an I/O interrupt to a program block. When the BOOL variable SuspOn\_Off is set to ON and A1 is set to ON, interrupts from Mod\_Int are suspended until SuspOn\_Off is reset.



## 4.6 Conversion Functions

The Conversion functions change a data item from one number format (data type) to another. Many programming instructions, such as math functions, must be used with data of one type. As a result, data conversion is often required before using those instructions.

Function	Description
<b>Convert Angles</b>	
DEG_TO_RAD	Converts degrees to radians
RAD_TO_DEG	Converts radians to degrees
<b>Convert to BCD4 (4-digit Binary-Coded-Decimal)</b>	
UINT_TO_BCD4	Converts UINT (16-bit unsigned integer) to BCD4
INT_TO_BCD4	Converts INT (16-bit signed integer) to BCD4
<b>Convert to BCD8 (8-digit Binary-Coded-Decimal)</b>	
DINT_TO_BCD8	Converts DINT (32-bit signed integer) to BCD8
<b>Convert to INT (16-bit signed integer)</b>	
BCD4_TO_INT	Converts BCD4 to INT
UINT_TO_INT	Converts UINT to INT
DINT_TO_INT	Converts DINT to INT
REAL_TO_INT	Converts REAL to INT
<b>Convert to UINT (16-bit unsigned integer)</b>	
BCD4_TO_UINT	Converts BCD4 to UINT
INT_TO_UINT	Converts INT to UINT
DINT_TO_UINT	Converts DINT to UINT
REAL_TO_UINT	Converts REAL to UINT
<b>Convert to DINT (32-bit signed integer)</b>	
BCD8_TO_DINT	Converts 8-digit Binary-Coded-Decimal (BCD8) to DINT
UINT_TO_DINT	Converts UINT to DINT
INT_TO_DINT	Converts INT to DINT
REAL_TO_DINT	Converts REAL (32-bit signed real or floating-point values) to DINT
LREAL_TO_DINT	Converts REAL (64-bit signed real or floating-point values) to DINT
<b>Convert to REAL (32-bit signed real or floating-point values)</b>	
BCD4_TO_REAL	Converts BCD4 to REAL
BCD8_TO_REAL	Converts BCD8 to REAL
UINT_TO_REAL	Converts UINT to REAL
INT_TO_REAL	Converts INT to REAL
DINT_TO_REAL	Converts DINT to REAL
LREAL_TO_REAL	Converts LREAL to REAL
<b>Convert to LREAL (64-bit signed real or floating-point values)</b>	
DINT_TO_LREAL	Converts DINT to LREAL
REAL_TO_LREAL	Converts REAL to LREAL
<b>Truncate</b>	
TRUNC_DINT	Rounds a REAL number down to a DINT (32-bit signed integer) number
TRUNC_INT	Rounds a REAL number down to an INT (16-bit signed integer) number

### 4.6.1 Convert Angles



Mnemonics:  
 DEG\_TO\_RAD\_REAL  
 DEG\_TO\_RAD\_LREAL  
 RAD\_TO\_DEG\_REAL  
 RAD\_TO\_DEG\_LREAL

When the Degrees to Radians (DEG\_TO\_RAD) or the Radians to Degrees (RAD\_TO\_DEG) function receives power flow, it performs the appropriate angle conversion on the REAL or LREAL value in input IN and places the result in output Q.

DEG\_TO\_RAD and RAD\_TO\_DEG pass power flow to the right when they execute, unless IN is NaN (Not a Number).

#### Operands

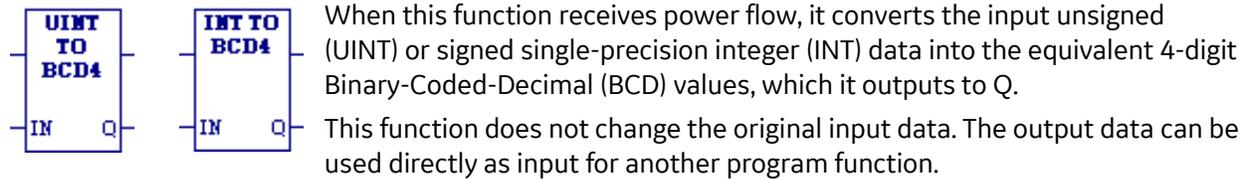
Parameter	Description	Allowed Operands	Optional
IN	The value to convert.	All except S, SA, SB, and SC	No
Q	The converted value.	All except S, SA, SB, and SC	No

#### Example

A value of +1500 radians is converted to degrees. The result is placed in %R00001 and %R00002.



### 4.6.2 Convert UINT or INT to BCD4



The function passes power flow when power is received, unless the conversion would result in a value that is outside the range 0 to 9,999.

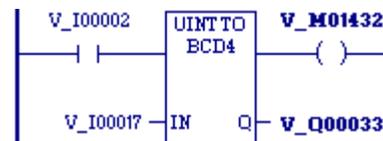
**Tip:** Data can be converted to BCD format to drive BCD-encoded LED displays or presets to external devices such as high-speed counters.

#### Operands

Parameter	Description	Allowed Operands	Optional
IN	The UINT or INT value to convert to BCD4.	All except S, SA, SB, and SC	No
Q	The BCD4 equivalent value of the original UINT or INT value in IN.	All except S, SA, SB, and SC	No

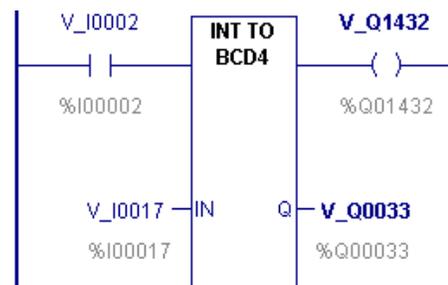
#### Example - UINT to BCD4

Whenever input %I0002 is set and no errors exist, the UINT at input location %I00017 through %I00032 is converted to four BCD digits and the result is stored in memory locations %Q00033 through %Q00048. Coil %M01432 is used to check for successful conversion.



#### Example - INT to BCD4

Whenever input %I0002 is set and no errors exist, the INT values at input locations %I0017 through %I0032 are converted to four BCD digits, and the result is stored in memory locations %Q0033 through %Q0048. Coil %Q1432 is used to check for successful conversion.



### 4.6.3 Convert DINT to BCD8



When DINT\_TO\_BCD8 receives power flow, it converts the input signed double-precision integer (DINT) data into the equivalent 8-digit Binary-Coded-Decimal (BCD) values, which it outputs to Q. DINT\_TO\_BCD8 does not change the original DINT data.

**Note:** The output data can be used directly as input for another program function.

The function passes power flow when power is received, unless the conversion would result in a value that is outside the range 0 to 99,999,999.

#### Operands

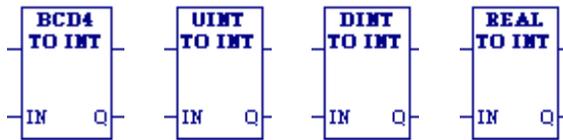
Parameter	Description	Allowed Operands	Optional
IN	The DINT value to convert to BCD8	All except S, SA, SB, and SC	No
Q	The BCD8 equivalent value of the original DINT value in IN	All except S, SA, SB, and SC	No

#### Example

Whenever input %I00002 is set and no errors exist, the double-precision signed integer (DINT) at input location %AI0003 is converted to eight BCD digits and the result is stored in memory locations %L00001 through %L00002.



#### 4.6.4 Convert BCD4, UINT, DINT, or REAL to INT



##### BDC4, UINT, and DINT

When this function receives power flow, it converts the input data into the equivalent single-precision signed integer (INT) value, which it outputs to Q. This function does not change the original input data. The output data can be used directly as input for another program function, as in the examples.

The function passes power flow when power is received, unless the data is out of range.

##### REAL

When REAL\_TO\_INT receives power flow, it rounds the input REAL data up or down to the nearest single-precision signed integer (INT) value, which it outputs to Q. REAL\_TO\_INT does not change the original REAL data.

**Note:** The output data can be used directly as input for another program function.

The function passes power flow when power is received, unless the data is out of range or NaN (Not a Number).



##### Warning

**Converting from REAL to INT may result in *Overflow*. For example, REAL 7.4E15, which equals  $7.4 \times 10^{15}$ , converts to INT OVERFLOW.**

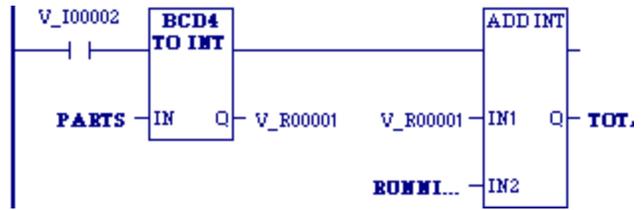
**Tip:** To truncate a REAL value and express the result as an INT, i.e., to remove the fractional part of the REAL number and express the remaining integer value as an INT, use TRUNC\_INT.

##### Operands

Parameter	Description	Allowed Operands	Optional
IN	The value to convert to INT.	All except S, SA, SB, and SC	No
Q	The INT equivalent value of the original value in IN.	All except S, SA, SB, and SC	No

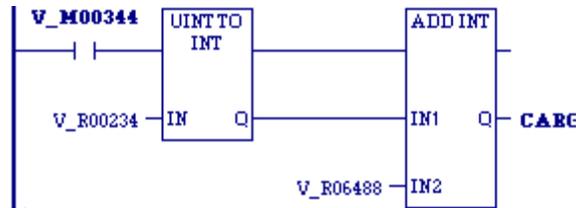
**Example: BCD4 to INT**

Whenever input %I0002 is set, the BCD-4 value in PARTS is converted to a signed integer (INT) and passed to the ADD\_INT function, where it is added to the INT value represented by the reference RUNNING. The sum is output by ADD\_INT to the reference TOTAL.



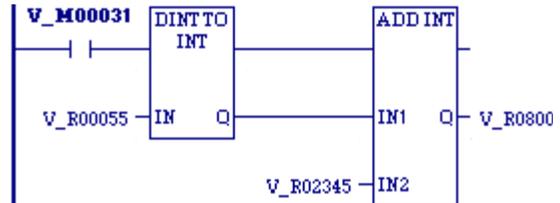
**Example: UINT to INT**

Whenever input %M00344 is set, the UINT value in %R00234 is converted to a signed integer (INT) and passed to the ADD function, where it is added to the INT value in %R06488. The sum is output by the ADD function to the reference CARGO.

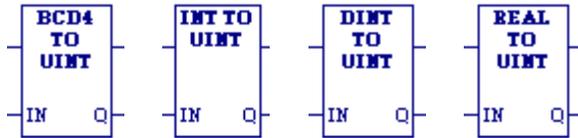


**Example: DINT to INT**

Whenever input %M00031 is set, the DINT value in %R00055 is converted to a signed integer (INT) and passed to the ADD function, where it is added to the INT at %R02345. The sum is output by the ADD function to %R08004.



### 4.6.5 Convert BCD4, INT, DINT, or REAL to UINT



When this function receives power flow, it converts the input data into the equivalent single-precision unsigned integer (UINT) value, which it outputs to Q.

The conversion to UINT does not change the original data. The output data can be used directly as input for another program function, as in the example.

The function passes power flow when power is received, unless the resulting data is outside the range 0 to +65,535.



#### Warning

**Converting from REAL to UINT may result in *Overflow*. For example, REAL 7.2E17, which equals  $7.2 \times 10^{17}$ , converts to UINT OVERFLOW.**

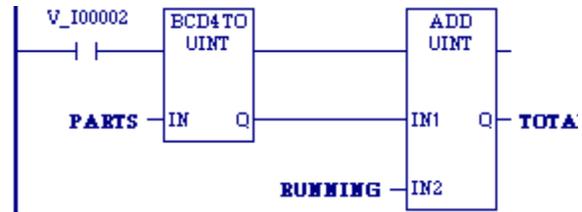
#### Operands

Parameter	Description	Allowed Operands	Optional
IN	The value to convert to UINT.	All except S, SA, SB, and SC	No
Q	The UINT equivalent value of the original input value in IN.	All except S, SA, SB, and SC	No

**Example: BCD4 to UINT**

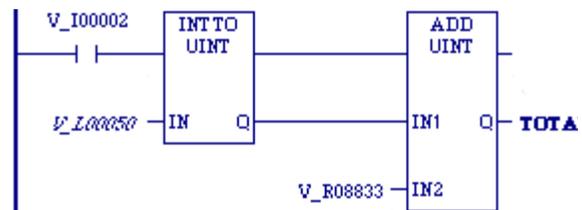
**Tip:** One use of BCD4\_TO\_UINT is to convert BCD data from the I/O structure into integer data and store it in memory. This can provide an interface to BCD thumbwheels or external BCD electronics, such as high-speed counters and position encoders.

In the example at right, whenever input %I0002 is set, the BCD4 value in PARTS is converted to an unsigned single-precision integer (UINT) and passed to the ADD\_UINT function, where it is added to the UINT value represented by the reference RUNNING. The sum is output by ADD\_UINT to the reference TOTAL.



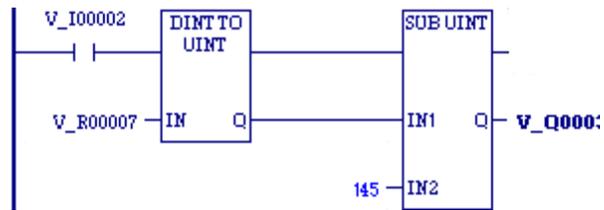
**Example: INT to UINT**

Whenever input %I0002 is set, the INT value in %L00050 is converted to an unsigned single-precision integer (UINT) and passed to the ADD\_UINT function, where it is added to the UINT value in %R08833. The sum is output by ADD\_UINT to the reference TOTAL.



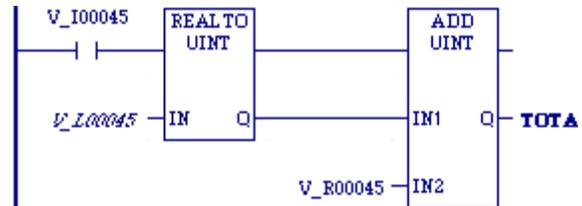
**Example: DINT to UINT**

Whenever input %I00002 is set and no errors exist, the double precision signed integer (DINT) at input location %R00007 is converted to an unsigned integer (UINT) and passed to the SUB function, where the constant value 145 is subtracted from it. The result of the subtraction is stored in the output reference location

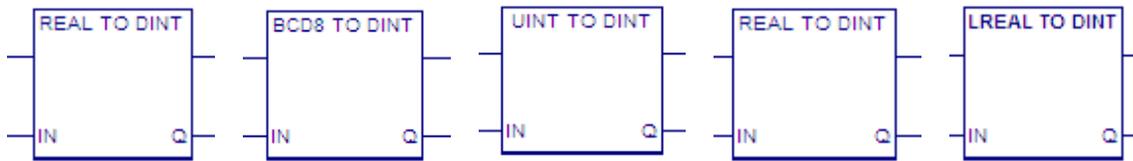


**Example: REAL to UINT**

Whenever input %I00045 is set, the REAL value in %L00045 is converted to an unsigned single-precision integer (UINT) and passed to the ADD\_UINT function, where it is added to the UINT value in %R00045. The sum is output by ADD\_UINT to the reference TOTAL.



### 4.6.6 Convert BCD8, UINT, INT, REAL or LREAL to DINT



#### BCD8, UINT, and INT

When this function receives power flow, it converts the data into the equivalent signed double-precision integer (DINT) value, which it outputs to Q. The conversion to DINT does not change the original data.

The output data can be used directly as input for another program function. The function passes power flow when power is received, unless the data is out of range.

#### REAL and LREAL

When REAL\_TO\_DINT or LREAL\_TO\_DINT receives power flow, it rounds the input data to the nearest double-precision signed integer (DINT) value, which it outputs to Q. These functions do not change the original REAL or LREAL data.

The output data can be used directly as input for another program function. The function passes power flow when power is received, unless the conversion would result in an out-of-range DINT value.



#### Warning

**Converting from LREAL or REAL to DINT may result in Overflow. For example, REAL 5.7E20, which equals  $5.7 \times 10^{20}$ , converts to DINT OVERFLOW.**

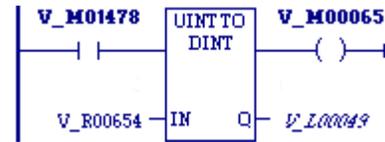
**Tip:** To truncate a REAL value and express the result as a DINT, i.e., to remove the fractional part of the REAL number and express the remaining integer value as a DINT, use TRUNC\_DINT.

#### Operands

Parameter	Description	Allowed Operands	Optional
IN	The value to convert to DINT.	All except S, SA, SB, and SC	No
Q	The DINT equivalent value of the original input value in IN.	All except S, SA, SB, and SC	No

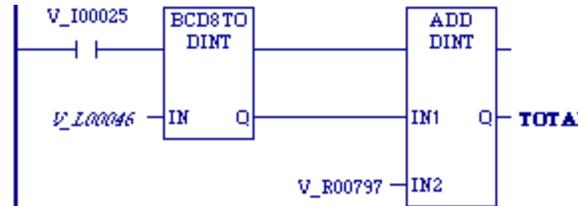
**Example: UINT to DINT**

Whenever input %M01478 is set, the unsigned single-precision integer (UINT) value at input location %R00654 is converted to a double-precision signed integer (DINT) and the result is placed in location %L00049. The output %M00065 is set whenever the function executes successfully.



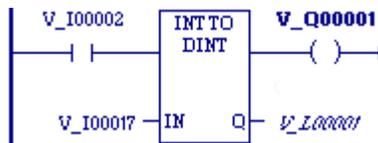
**Example: BCD8 to DINT**

Whenever input %I00025 is set, the BCD-8 value in %L00046 is converted to a signed double-precision integer (DINT) and passed to the ADD\_DINT function, where it is added to the DINT value in %R00797. The sum is output by ADD\_DINT to the reference TOTAL.



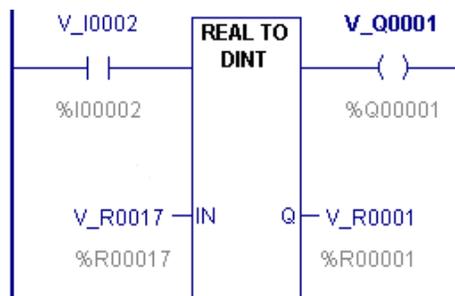
**Example: INT to DINT**

Whenever input %I00002 is set, the signed single-precision integer (INT) value at input location %I00017 is converted to a double-precision signed integer (DINT) and the result is placed in location %L00001. The output %Q01001 is set whenever the function executes successfully.

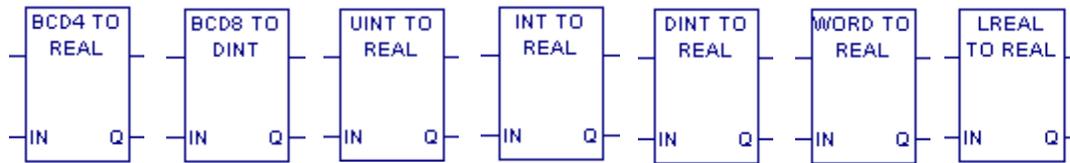


**Example: REAL to DINT**

Whenever input %I0002 is set, the REAL value at input location %R0017 is converted to a double precision signed integer (DINT) and the result is placed in location %R0001. The output %Q1001 is set whenever the function executes successfully.



### 4.6.7 Convert BCD4, BCD8, UINT, INT, DINT, and LREAL to REAL



When this function receives power flow, it converts the input data into the equivalent 32-bit floating-point (REAL) value, which it outputs to Q. The conversion to REAL does not change the original input data.

The output data can be used directly as input for another program function.

The function passes power flow when power is received, unless the conversion would result in a value that is out of range.

#### Warning



**Converting from BCD8 to REAL may result in the loss of significant digits.**

**This is because a BCD8 value is stored in a DWORD, which uses 32 bits to store a value, whereas a REAL (32-bit IEEE floating point number) uses 8 bits to store the exponent and the sign and only 24 bits to store the mantissa.**

#### Warning



**Converting from DINT to REAL may result in the loss of significant digits for numbers with more than 7 significant base-10 digits.**

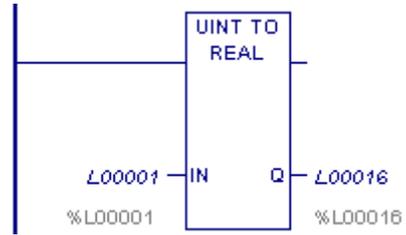
**This is because a DINT value uses 32 bits to store a value, which is the equivalent of up to 10 significant base-10 digits, whereas a REAL (32-bit IEEE floating point number) uses 8 bits to store the exponent and the sign and only 24 bits to store the mantissa, which is the equivalent of 7 or 8 significant base-10 digits. When the REAL result is displayed as a base-10 number, it may have up to 10 digits, but these are converted from the rounded 24-bit mantissa, so that the last 2 or 3 digits may be inaccurate.**

### Operands

Parameter	Description	Allowed Operands	Optional
IN	The value to convert to REAL.	All except S, SA, SB, and SC	No
Q	The REAL equivalent value of the original input value in IN.	All except S, SA, SB, and SC	No

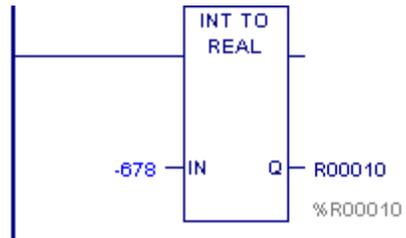
**Example: UINT to REAL**

The unsigned integer value in %L00001 is 825. The value placed in %L00016 is 825.000.



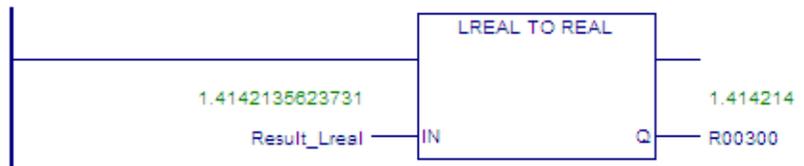
**Example: INT to REAL**

The integer value of input IN is -678. The value placed in %R00010 is -678.000.

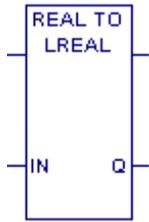


**Example: LREAL to REAL**

The double-precision floating point value of the square root of 2 is rounded to the nearest single-precision floating point value and placed in R00300.



### 4.6.8 Convert REAL to LREAL



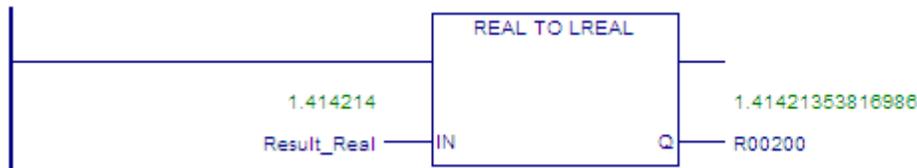
When REAL\_TO\_LREAL receives power flow, it converts the 32-bit single precision floating point REAL data to the equivalent 64-bit double-precision floating point data. REAL\_TO\_LREAL does not change the original REAL data.

#### Operands

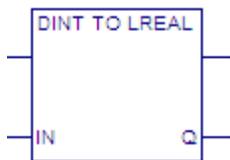
Parameter	Description	Allowed Operands	Optional
IN	The REAL value to convert to LREAL.	All except S, SA, SB, and SC	No
Q	The LREAL equivalent value of the original REAL value.	All except S, SA, SB, and SC	No

#### Example

The REAL value of the square root of 2 is converted to the LREAL data type and placed in R00200. Because the actual precision of the data in Result\_Real is seven decimal places, the additional decimal places in the data in R00200 are not valid.

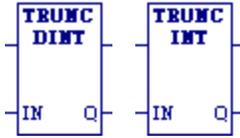


### 4.6.9 Convert DINT to LREAL



When DINT\_TO\_LREAL receives power flow, it converts the double-precision input data to 64-bit double-precision floating point data.

### 4.6.10 Truncate



When power is received, the Truncate functions TRUNC\_DINT and TRUNC\_INT round a floating-point (REAL) value down respectively to the nearest signed double-precision signed integer (DINT) or signed single-precision integer (INT) value. TRUNC\_DINT and TRUNC\_INT output the converted value to Q. The original data is not changed.

**Note:** The output data can be used directly as input for another program function.

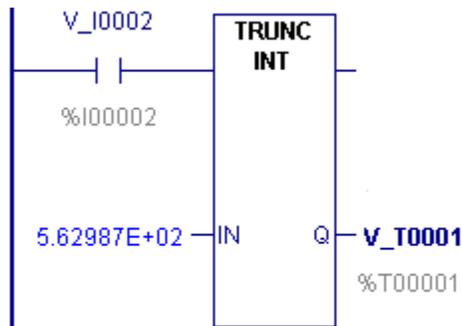
TRUNC\_DINT and TRUNC\_INT pass power flow when power is received, unless the specified conversion would result in a value that is out of range or unless IN is NaN (Not a Number).

#### Operands

Parameter	Description	Allowed Operands	Optional
IN	The REAL value whose copy is to be converted and truncated. The original is left intact.	All except S, SA, SB, and SC	No
Q	The truncated value of the original REAL value in IN.	All except S, SA, SB, and SC	No

#### Example

The displayed constant is truncated and the integer result 562 is placed in %T0001.



## 4.7 Counters

Function	Mnemonic	Description
Down Counter	DNCTR	Counts down from a preset value. The output is ON whenever the Current Value is $\leq 0$ .
Up Counter	UPCTR	Counts up to a designated value. The output is ON whenever the Current Value is $\geq$ the Preset Value.

### 4.7.1 Data Required for Counter Function Blocks



#### Warning

**Do not use two consecutive words (registers) as the starting addresses of two counters. Logic Developer PLC does not check or warn you if register blocks overlap. Timers will not work if you place the current value of a second timer on top of the preset value for the previous timer.**

---

Each counter uses a one-dimensional, three-word array of %R, %W, %P, %L, or symbolic memory to store the following information:

**Current value Word 1 (CV)**



**Warning**

**The first word (CV) can be read but should not be written to, or the function may not work properly.**

**Preset value (PV) Word 2** When the Preset Value (PV) operand is a variable, it is normally set to a different location than word 2 in the timer’s or counter’s three-word array.

- If you use a different address and you change word 2 directly, your change will have no effect, as PV will overwrite word 2.
- If you use the same address for the PV operand and word 2, you can change the Preset Value in word 2 while the timer or counter is running and the change will be effective.

**Control word Word 3**

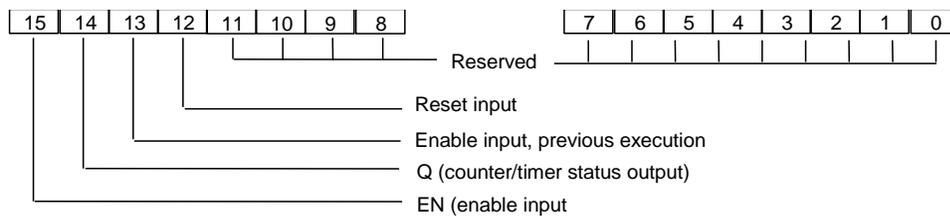
The control word stores the state of the Boolean inputs and outputs of its associated timer or counter, as shown in the following diagram:



**Warning**

**The third word (Control) can be read but should not be written to; otherwise, the function will not work.**

**Word 3: Control Word Structure**



**Note:** Bits 0 through 13 are not used for counters.

### 4.7.2 Down Counter



The Down Counter (DNCTR) function counts down from a preset value. The minimum Preset Value (PV) is zero; the maximum PV is +32,767 counts. When the Current Value (CV) reaches the minimum value, -32,768, it stays there until reset. When DNCTR is reset, CV is set to PV. When the power flow input transitions from OFF to ON, CV is decremented by one. The output is ON whenever  $CV \leq 0$ .

The output state of DNCTR is retentive on power failure; no automatic initialization occurs at power-up.



#### Warning

**Do not use the Address of the down counter with other instructions. Overlapping references cause erratic counter operation.**

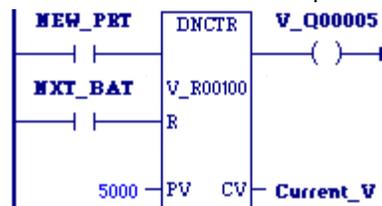
**Note:** For DNCTR to function properly, you must provide an initial reset to set the CV to the value in PV. If DNCTR is not initially reset, CV will decrement from 0 and the output of DNCTR will be set to ON immediately.

#### Operands

Parameter	Description	Allowed Operands	Optional
Address (????)	The beginning address of a three-word WORD array: Word 1: Current Value (CV) Word 2: Preset Value (PV)% Word 3: Control word	R, W, P, L, symbolic	No
R	When R receives power flow, it resets the counter's CV to PV.	Power flow	No
PV	Preset Value to copy into word 2 of the counter's address when the counter is enabled or reset. $0 \leq PV \leq 32,767$ . If PV is out of range, word 2 cannot be reset.	All except S, SA, SB, SC	No
CV	The current value of the counter	All except S, SA, SB, SC and constant	No

#### Example - Down Counter

DNCTR counts 5000 new parts before energizing output %Q00005.



### 4.7.3 Up Counter



The Up Counter (UPCTR) function counts up to the Preset Value (PV). The range is 0 to +32,767 counts. When the Current Value (CV) of the counter reaches 32,767, it remains there until reset. When the UPCTR reset is ON, CV resets to 0. Each time the power flow input transitions from OFF to ON, CV increments by 1. CV can be incremented past the Preset Value (PV). The output is ON whenever  $CV \geq PV$ . The output (Q) stays ON until the R input receives power flow to reset CV to zero.

The state of UPCTR is retentive on power failure; no automatic initialization occurs at power-up.



#### Warning

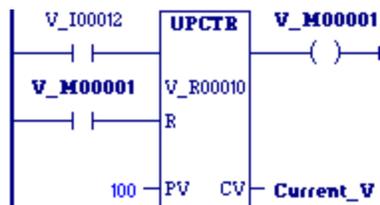
**Do not use the Address of the up counter with other instructions. Overlapping references cause erratic counter operation.**

#### Operands

Parameter	Description	Allowed Operands	Optional
Address (????)	The beginning address of a three-word WORD array: Word 1: Current Value (CV) Word 2: Preset Value (PV) Word 3: Control word	R, W, P, L, symbolic	No
R	When R is ON, it resets the counter's CV to 0.	Power flow	No
PV	Preset Value to copy into word 2 of the counter's address when the counter is enabled or reset. $0 \leq PV \leq 32,767$ . If PV is out of range, it does not affect word 2.	All except S, SA, SB, and SC	No
CV	The current value of the counter	All except S, SA, SB, SC and constant	No

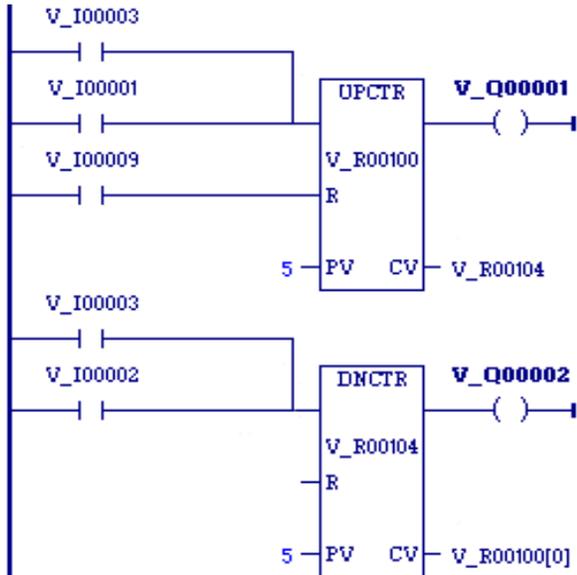
#### Example - Up Counter

Every time input %I0012 transitions from OFF to ON, the Up Counter counts up by 1; internal coil %M0001 is energized whenever 100 parts have been counted. Whenever %M0001 is ON, the accumulated count is reset to zero.



**Example - Up Counter and Down Counter**

This example uses an up/down counter pair with a shared register for the accumulated or current value. When the parts enter the storage area, the up counter increments by 1, increasing the current value of the parts in storage by a value of 1. When a part leaves the storage area, the down counter decrements by 1, decreasing the inventory storage value by 1. To avoid conflict with the shared register, both counters use different register addresses but each has a current value (CV) address that is the same as the accumulated value for the other register.



## 4.8 Data Move Functions

The Data Move functions provide basic data move capabilities.

Function	Mnemonics	Description
Array Size	ARRAY_SIZE	Counts the number of elements in an array.
Array Size Dimension 1	ARRAY_SIZE_DIM1	Returns the value of the Array Dimension 1 property of a one- or two-dimensional array.
Array Size Dimension 2	ARRAY_SIZE_DIM2	Returns the value of the Array Dimension 2 property of a two-dimensional array.
Block Clear	BLK_CLR_WORD	Replaces all the contents of a block of data with zeroes. Can be used to clear an area of WORD or analog memory.
Block Move	BLKMOV_DINT BLKMOV_DWORD BLKMOV_INT BLKMOV_REAL BLKMOV_UINT BLKMOV_WORD	Copies a block of seven constants to a specified memory location. The constants are input as part of the function.
Bus Read	BUS_RD_BYTE BUS_RD_DWORD BUS_RD_WORD	Reads data from a module on the bus.
Bus Read Modify Write	BUS_RMW_BYTE BUS_RMW_DWORD BUS_RMW_WORD	Uses a read/modify/write cycle to update a data element in a module on the bus.
Bus Test and Set	BUS_TS_BYTE BUS_TS_WORD	Handles semaphores on the bus.
Bus Write	BUS_WRT_BYTE BUS_WRT_DWORD BUS_WRT_WORD	Writes data to a module on the bus.
Communication Request	COMMREQ	Allows the program to communicate with an intelligent module, such as a Genius Bus Controller or a High Speed Counter.
Data Initialization	DATA_INIT_DINT DATA_INIT_DWORD DATA_INIT_INT DATA_INIT_REAL DATA_INIT_LREAL DATA_INIT_UINT DATA_INIT_WORD	Copies a block of constant data to a reference range. The mnemonic specifies the data type.
Data Initialize ASCII	DATA_INIT_ASCII	Copies a block of constant ASCII text to a reference range.
Data Initialize DLAN	DATA_INIT_DLAN	Used with a DLAN Interface module.
Data Initialize Communications Request	DATA_INIT_COMM	Initializes a COMMREQ function with a block of constant data. The length should equal the size of the COMMREQ function's entire command block.

Function	Mnemonics	Description
Move	MOVE_BOOL MOVE_DATA MOVE_DINT MOVE_DWORD MOVE_INT MOVE_REAL MOVE_LREAL MOVE_UINT MOVE_WORD	Copies data as individual bits, so the new location does not have to be the same data type. Data can be moved into a different data type without prior conversion.
Move Data Explicit	MOVE_DATA_EX	Provides an input that allows for data coherency by locking symbolic memory being written to during the copy operation.
Move from Flat	MOVE_FROM_FLAT	Copies reference memory data to a UDT variable or UDT array. Provides the option of locking the symbolic or I/O variable memory area being written to during the copy operation.
Move to Flat	MOVE_TO_FLAT	Copies data from symbolic or I/O variable memory to reference memory. Copies across mismatching data types.
Shift Register	SHFR_BIT SHFR_DWORD SHFR_WORD	Shifts one or more data bits, data WORDs or data DWORDs from a reference location into a specified area of memory. Data already in the area is shifted out.
Size Of	SIZE_OF	Counts the number of bits used by a variable.
Swap	SWAP_DWORD SWAP_WORD	Swaps two BYTES of data within a WORD or two WORDs within a DWORD.

### 4.8.1 Array Size



Counts the number of elements in the array assigned to input IN and writes the number to output Q.

In an array of structure variables, the number of structure variables is written to Q; the elements in the structure variables are not counted.

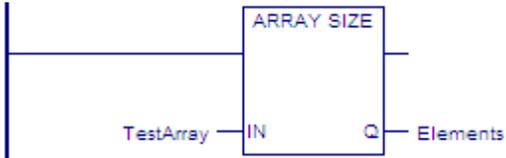
**Tip:** If the array assigned to input IN of ARRAY\_SIZE is passed to a parameterized C block for processing, also pass the value of output Q to the block. In the C block logic, use the value of output Q to ensure all array elements are processed without exceeding the end of the array. For a two-dimensional array, this method works only if all elements are treated identically; for example, all are initialized to the same value.

#### Operands

Parameter	Description	Allowed Operands	Optional
IN	Array of any data type whose elements are counted. If a non-array variable is assigned to IN, the value of Q is 1.	Data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	No
Q	Number of elements in the array assigned to input IN.	DINT or DWORD variable. Data flow, I, Q, M, T, G, R, P, L, AI, AQ, W, symbolic, I/O variable	No

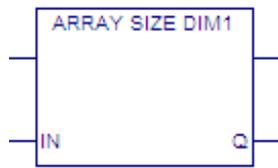
#### Example

The two-dimensional array TestArray has its Array Dimension 1 property set to 4 and its Array Dimension 2 property set to 3. ARRAY\_SIZE calculates  $4 \times 3$  and writes the value 12 to the variable Elements.



## 4.8.2 Array Size Dimension Function Blocks

### Array Size Dimension 1



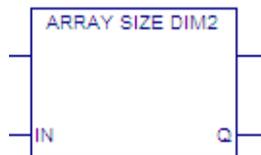
Returns the value of the Array Dimension 1 property of an array and writes the value to output Q. If a non-array variable is assigned to IN, the value of Q is 0.

In an LD or ST block that is not a parameterized block or a User Defined Function Block (UDFB), you can use the output Q value to ensure that a loop using a variable index to access array elements does not exceed the array's first dimension.

### Operands

Parameter	Description	Allowed Operands	Optional
IN	Array of any data type.	Data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	No
Q	The value of the Array Dimension 1 property of the array assigned to input IN. The value is set to 0 if a non-array is assigned to IN. <b>Note:</b> Because the index of the first element of an array is zero, the index of the last element is one less than the value assigned to Q.	DINT or DWORD variable. Data flow, I, Q, M, T, G, R, P, L, AI, AQ, W, symbolic, I/O variable	No

### Array Size Dimension 2



Returns the value of the Array Dimension 2 property of an array and writes the value to output Q. If a non-array variable is assigned to IN, the value of Q is 0.

In an LD or ST block that is not a parameterized block or a User Defined Function Block (UDFB), you can use the output Q value to ensure that a loop using a variable index to access array elements does not exceed the array's second dimension.

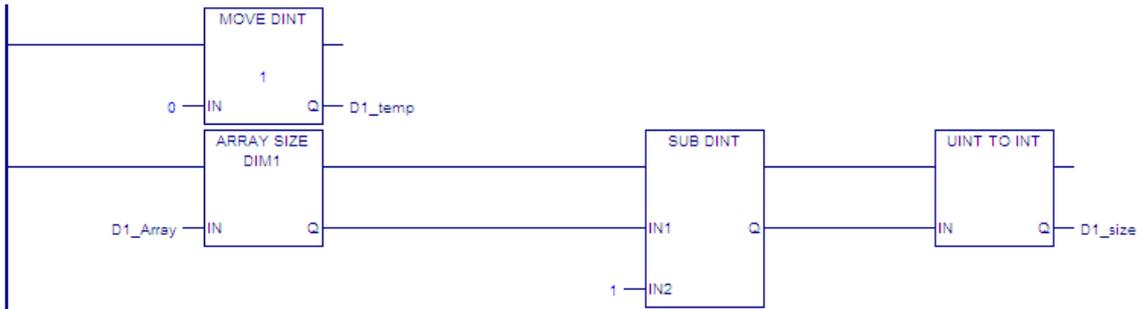
### Operands

Parameter	Description	Allowed Operands	Optional
IN	Array of any data type.	Data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	No
Q	The value of the Array Dimension 2 property of the array assigned to input IN. The value is set to 0 if a non-array is assigned to IN. <b>Note:</b> Because the index of the first element of an array is zero, the index of the last element is one less than the value assigned to Q.	DINT or DWORD variable. Data flow, I, Q, M, T, G, R, P, L, AI, AQ, W, symbolic, I/O variable	No

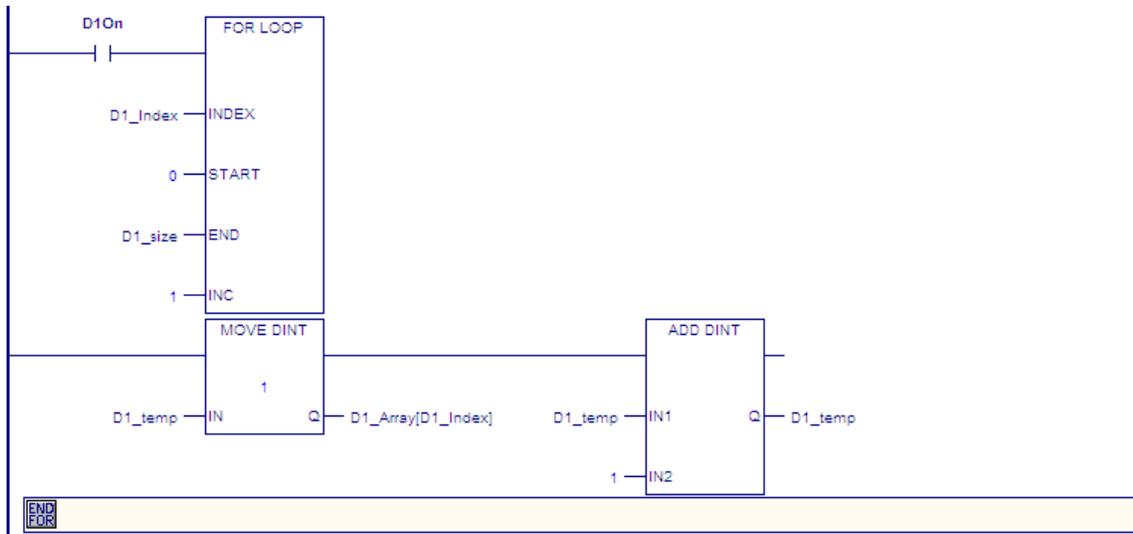
**Example - FOR\_LOOP that Iterates Through Dimension 1 of an Array**

To use a FOR\_LOOP to access array elements by means of a variable index, you must ensure that the FOR\_LOOP does not iterate beyond the last element of the array.

In the following logic, MOVE\_DINT initializes the variable D1\_temp to 0. ARRAY\_SIZE\_DIM1 counts the number of elements of a one-dimensional array named D1\_Array and outputs the result to output Q. Because the index of the first element of an array is zero, the loop must iterate (Q - 1) times. SUB\_DINT performs the subtraction and the result is converted to an INT value and assigned to variable D1\_size.



In the following rungs, the FOR\_LOOP executes when D1ON is set to On. The variable index D1\_Index increments by 1 from 0 through D1\_size, the value calculated by ARRAY\_SIZE\_DIM1 and SUB\_DINT. In each loop, the value of D1\_temp is assigned to the element D1\_Array[D1\_Index] and D1\_temp is increased by 1.



You can use a FOR\_LOOP to iterate through an array's second dimension in a method similar to this example. You can also use nested FOR\_LOOPS to ensure that operations on elements using two variable indexes each do not exceed their array dimension. For additional examples, refer to the online help.

### 4.8.3 Block Clear

**BLK CLR WORD** ??  
IN

When the Block Clear (BLKCLR\_WORD) function receives power flow, it fills the specified block of data with zeroes, beginning at the reference specified by IN. When the data to be cleared is from BOOL (discrete) memory (%I, %Q, %M, %G, or %T), the transition information associated with the references is updated. BLKCLR\_WORD passes power to the right whenever it receives power.

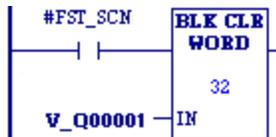
**Note:** The input parameter IN is not included in coil checking.

#### Operands

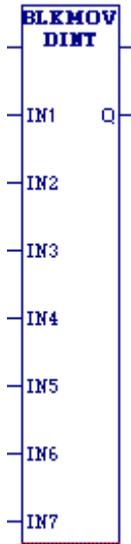
Parameter	Description	Allowed Operands	Optional
Length (??)	The number of words to clear, starting at the IN location. $1 \leq \text{Length} \leq 256$ words.	Constant	No
IN	The first WORD of the memory block to clear to 0.	All except %S and data flow.	No

#### Example

At power-up, 32 words of %Q memory (512 points) beginning at %Q0001 are filled with zeroes. The transition information associated with these references will also be updated.



### 4.8.4 Block Move



When the Block Move (BLKMOV) function receives power flow, it copies a block of seven constants into consecutive locations beginning at the destination specified in output Q. BLKMOV passes power to the right whenever it receives power.

Mnemonics:  
 BLKMOV\_DINT  
 BLKMOV\_DWORD  
 BLKMOV\_INT  
 BLKMOV\_REAL  
 BLKMOV\_UINT  
 BLKMOV\_WORD

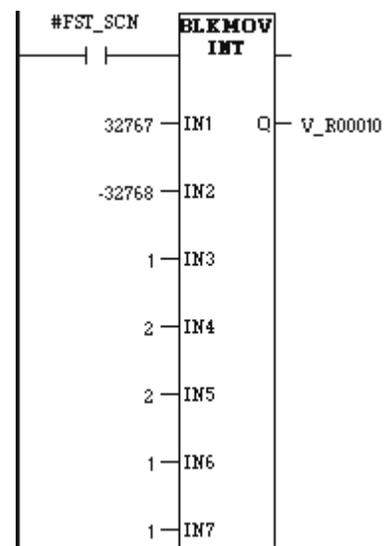
#### Operands

**Note:** For each mnemonic, use the corresponding data type for the Q operand. For example, BLKMOV\_DINT requires Q to be a DINT variable.

Parameter	Description	Allowed Operands	Optional
IN1 to IN7	The seven constant values to move.	Constants. Constant type must match function type.	No
Q	The first memory location of the destination for the moved values. IN1 is moved to Q.	All except %S. %SA, SB, SC are also prohibited on BLKMOV_REAL, BLK_MOV_INT, and BLK_MOV_UINT.	No

#### Example

When the enabling input represented by the name #FST\_SCN is ON, BLKMOV\_INT copies the seven input constants into memory locations %R0010 through %R0016.



### 4.8.5 BUS\_ Functions

Four program functions allow the PACSystems CPU to communicate with modules in the system.

- Bus Read (BUS\_RD)
- Bus Write (BUS\_WRT)
- Bus Read/Modify/Write (BUS\_RMW)
- Bus Test and Set (BUS\_TS)

These functions use the same parameters to specify which module on the bus will exchange data with the CPU.

**Note:** Additional information related to addressing modules is required to use the BUS\_ functions. For open VME modules in an RX7i system, refer to the *PACSystems RX7i User's Guide to Integration of VME Modules*, GFK-2235. For other modules, refer to the product documentation provided by the manufacturer.

#### Rack, Slot, Subslot, Region, and Offset Parameters

The rack and slot parameters refer to a module in the hardware configuration. The region parameter refers to a memory region configured for that module. The sub-slot is ordinarily set to 0. The offset is a 0-based number that the function adds to the module's base address (which is part of the memory region configuration) to compute the address to be read or written.

#### BUS Read

<div style="border-bottom: 1px solid black; padding-bottom: 5px;">                 BUS RD                  BYTE             </div> <div style="border-bottom: 1px solid black; padding: 5px; text-align: center;">                 ??             </div> <div style="border-bottom: 1px solid black; padding: 5px;">                 R    ST             </div> <div style="border-bottom: 1px solid black; padding: 5px;">                 S    Q             </div> <div style="border-bottom: 1px solid black; padding: 5px;">                 SS             </div> <div style="border-bottom: 1px solid black; padding: 5px;">                 RGN             </div> <div style="padding: 5px;">                 OFF             </div>	<p>The BUS_RD function reads data from the bus.</p> <p>This function should be executed before the data is needed in the program. If the amount of data to be read is greater than 32767 BYTES, WORDS, or DWORDS, use multiple instructions to read the data.</p> <p>When BUS_RD receives power flow, it accesses the module at the specified rack (R), slot (S), subslot (SS), address region (RGN) and offset (OFF). BUS_RD copies the specified number (Length) of data units (DWORDS, WORDs or BYTEs) from the module to the CPU, beginning at output reference (Q).</p> <p>The function passes power to the right when its operation is successful. The status of the operation is reported in the status location (ST).</p> <p><b>Note:</b> For each BUS_RD function type, use the corresponding data type for the Q operand. For example, BUS_RD_BYTE requires Q to be a BYTE variable.</p> <p><b>Note:</b> An interrupt block can preempt the execution of a BUS_RD function. On the bus, only 256 bytes are read coherently (i.e., read without being preempted by an interrupt).</p>	<p>Mnemonics:</p> <p>BUS_RD_DINT</p> <p>BUS_RD_DWORD</p> <p>BUS_RD_WORD</p>
---	--	---

**Operands for BUS READ**

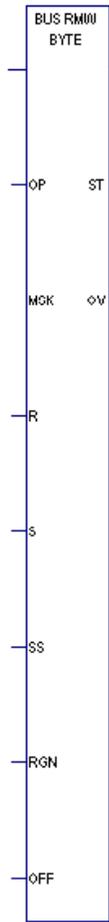
Parameter	Description	Allowed Operands	Optional
Length (??)	The number of BYTEs, DWORDs, or WORDs. 1 to 32,767.	Constant	No
R	Rack number. UINT constant or variable.	All except %S—%SC	No
S	Slot number. UINT constant or variable.	All except %S—%SC	No
SS	Subslot number (defaults to 0). UINT constant or variable.	All except %S—%SC	Yes
RGN	Region (defaults to 1). WORD constant or variable.	All except %S—%SC	Yes
OFF	The offset in bytes. DWORD constant or variable.	All except %S—%SC	No
ST	The status of the operation. WORD variable.	All except variables located in %S—%SC, and constants	Yes
Q	Reference for data read from the module. DWORD variable.	All except variables located in %S—%SC, and constants	No

**BUS\_RD Status in the ST Output**

The BUS\_RD function returns one of the following values to the ST output:

0	Operation successful.
1	Bus error
2	Module does not exist at rack/slot location.
3	Module at rack/slot location is an invalid type.
4	Start address outside the configured range.
5	End address outside the configured address range.
6	Absolute address even but interface configured as odd byte only
8	Region not enabled
10	Function parameter invalid.

### BUS Read Modify Write



The BUS\_RMW function updates one byte, word, or double word of data on the bus. This function locks the bus while performing the read-modify-write operation.

Other mnemonic:  
BUS\_RMW\_WORD

When the BUS\_RMW function receives power flow through its enable input, the function reads a dword, word or byte of data from the module at the specified rack (R), slot (S), subslot (SS) and optional address region (RGN) and offset (OFF). The original value is stored in parameter (OV).

The function combines the data with the data mask (MSK). The operation performed (AND / OR) is selected with the OP parameter. The mask value is dword data. When operating on a word of data, only the lower 16 bits are used. When operating on a byte of data, only the lower 8 bits of the mask data are used. The result is then written back to the same address from which it was read.

The BUS\_RMW function passes power to the right when its operation is successful, and returns a status value to the ST output.

### Operands for BUS\_RMW

For BUS\_RMW\_WORD, the absolute bus address must be a multiple of 2. For BUS\_RMW\_DWORD, it must be a multiple of 4.

The absolute bus address is equal to the base address plus the offset value.

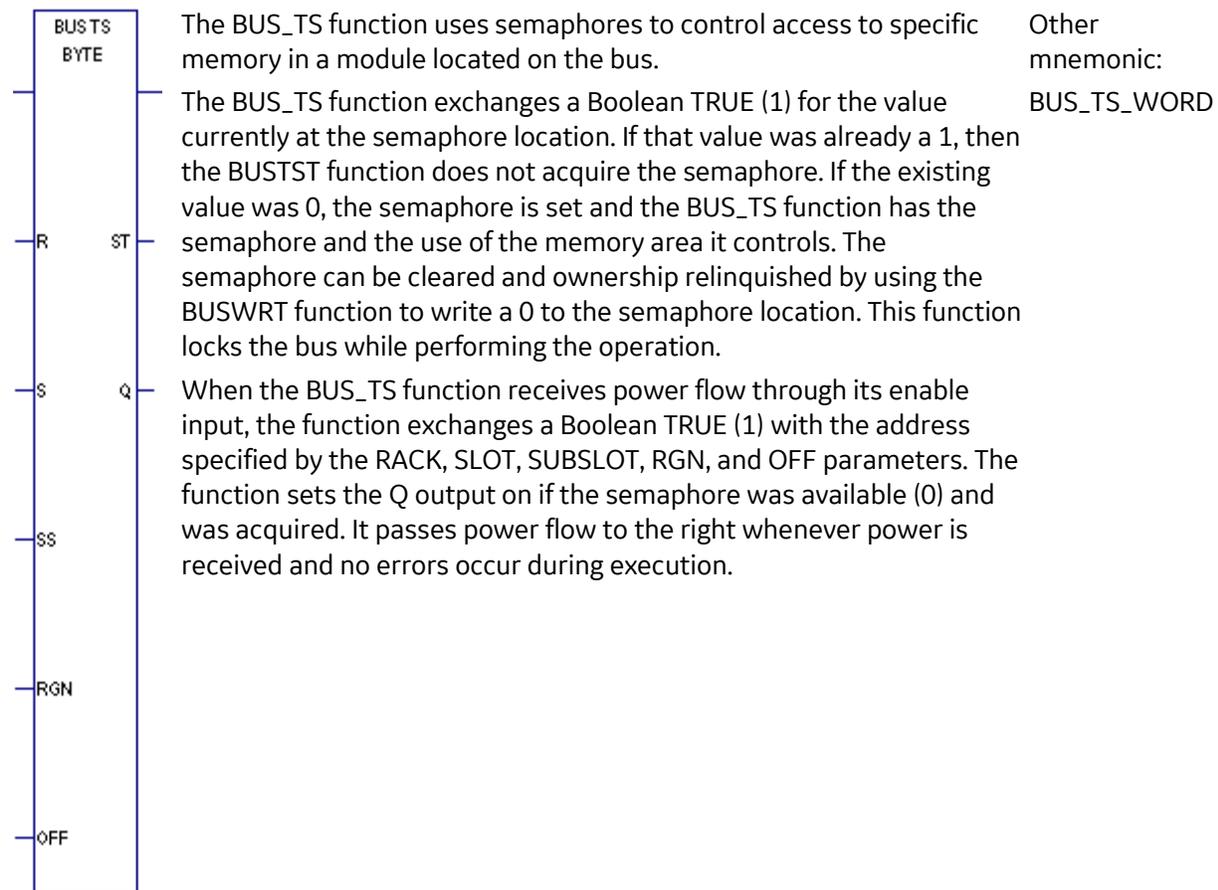
Parameter	Description	Allowed Operands	Optional
OP	Type of operation: 0 = AND 1 = OR	Constant	No
MSK	The data mask. DWORD constant or variable.	All except %S—%SC	No
R	Rack number. UINT constant or variable.	All except %S—%SC	No
S	Slot number. UINT constant or variable.	All except %S—%SC	No
SS	Subslot number (optional, defaults to 0). UINT constant or variable.	All except %S—%SC	Yes
RGN	Region (defaults to 1). WORD constant or variable.	All except %S—%SC	Yes
OFF	The offset in bytes. DWORD constant or variable.	All except %S—%SC	No
ST	The status of the operation. WORD variable.	All except variables located in %S—%SC, and constants	Yes
OV	Original value. DWORD variable.	All except variables located in %S—%SC, and constants	Yes

### BUS\_RMW Status in the ST Output

The BUS\_RMW function returns one of the following values to the ST output:

0	Operation successful.
1	Bus error
2	Module does not exist at rack/slot location.
3	Module at rack/slot location is an invalid type.
4	Start address outside the configured range.
5	End address outside the configured address range.
6	Absolute address even but interface configured as odd byte only
7	For WORD type, absolute bus address is not a multiple of 2. For DWORD type, absolute bus address is not a multiple of 4.
8	Region not enabled
9	Function type too large for configured access type.
10	Function parameter invalid.

## BUS Test and Set



### Operands for BUS Test and Set

BUS\_TS can be programmed as BUS\_TS\_BYTE or BUS\_TS\_WORD. For BUS\_TS\_WORD, the absolute address of the module must be a multiple of 2. The absolute address is equal to the base address plus the offset value.

Parameter	Description	Allowed Operands	Optional
R	Rack number. UINT constant or variable.	All except %S—%SC	No
S	Slot number. UINT constant or variable.	All except %S—%SC	No
SS	Subslot number (defaults to 0). UINT constant or variable.	All except %S—%SC	Yes
RGN	Region (defaults to 1). WORD constant or variable.	All except %S—%SC	Yes
OFF	The offset in bytes. DWORD constant or variable.	All except %S—%SC	No
ST	The status of the bus test and set operation. WORD variable.	All except variables located in %S—%SC, and constant	Yes
Q	Output set on if the semaphore was available (0). Otherwise, Q is set off.	Power flow	Yes

## BUS Write



When the BUS\_WRT function receives power flow through its enable input, it writes the data located at reference (IN) to the module at the specified rack (R), slot (S), subslot (SS) and optional address region (RGN) and offset (OFF). BUSWRT writes the specified length (LEN) of data units (DWORDS, WORDS or BYTES).

Mnemonics:  
 BUS\_WRT\_DINT  
 BUS\_WRT\_DWORD  
 BUS\_WRT\_WORD

The BUS\_WRT function passes power to the right when its operation is successful. The status of the operation is reported in the status location (ST).

**Note:** For each BUS\_WRT function type, use the corresponding data type for the IN operand. For example, BUS\_WRT\_BYTE requires IN to be a BYTE variable.

**Note:** An interrupt block can preempt the execution of a BUS\_WRT function. On the bus, only 256 bytes are written coherently (i.e., written without being preempted by an interrupt).

## Operands for Bus Write

Parameter	Description	Allowed	Optional
Length (??)	Length. The number of BYTEs, DWORDs, or WORDs. 1 to 32,767.	Constant	No
IN	Reference for data to be written to the module. DWORD variable.	All except variables located in %S–%SC, and constant	No
R	Rack number. UINT constant or variable.	All except %S–%SC	No
S	Slot number. UINT constant or variable.	All except %S–%SC	No
SS	Subslot number (defaults to 0) UINT constant or variable.	All except %S–%SC	Yes
RGN	Region. (defaults to 1) WORD constant or variable.	All except %S–%SC	Yes
OFF	The offset in bytes. DWORD constant or variable.	All except %S–%SC	No
ST	The status of the operation. WORD variable.	All except variables located in %S–%SC, and constant	Yes

### 4.8.6 Communication Request (COMMREQ)



The Communication Request (COMMREQ) function communicates with an intelligent module, such as a Genius Communications Module or High Speed Counter.

**Notes:**

- The information presented in this section shows only the basic format of the COMMREQ function. Many types of COMMREQs have been defined. You will need additional information to program the COMMREQ for each type of device. Programming requirements for each module that uses the COMMREQ function are described in the specialty module's user documentation.
- If you are using the COMMREQ to conduct serial communications, refer to the *Serial I/O, SNP and RTU Protocols* section in *PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual, GFK-2222*.
- If you are using the COMMREQ to interface with an intelligent module (such as Genius Communications Gateway), refer to that product's user manual for operational details.
- A COMMREQ instruction inside an interrupt block being executed may cause the block to be preempted when a new, incoming interrupt has the same priority.

When COMMREQ receives power flow, it sends the command block of data specified by the IN operand to the communications TASK in the intelligent or specialty module, at the rack/slot location specified by the SYSID operand. The command block contents are sent to the receiving device and the program execution resumes immediately. (Because PACSystems does not support WAIT mode COMMREQs, the timeout value is ignored.)

The COMMREQ passes power flow unless the following fault conditions exist. The Function Faulted (FT) output may be set ON if:

- Control block is invalid
- Destination is invalid (target module is not present or is faulted)
- Target module cannot receive mail because its queue is full

The Function Faulted output may have these states:

Enable	Error?	Function Faulted Output
active	no	OFF
active	yes	ON
not active	no execution	OFF

#### Command Block

The command block provides information to the intelligent module on the command to be performed. The command block starts at the reference specified by the operand IN. This address may be in any word-oriented area of memory (%R, %P, %L, %W, %AI, %AQ, or symbolic non-discrete variables). The length of the command block depends on the amount of data sent to the device.

The Command Block contains the data to be communicated to the other device, plus information related to the execution of the COMMREQ. Information required for the command block can be placed

in the designated memory area using a programming function such as MOVE, BLKMOV, or DATA\_INIT\_COMM.

### Command Block Structure

Address	Data Block Length (in words)	The number of data words starting with the data at address+6 to the end of the command block, inclusive. The data block length ranges from 1 to 128 words. Each COMMREQ command has its own data block length. When entering the data block length, you must ensure that the command block fits within the register limits
Address + 1	Wait/No Wait Flag	Must be set to 0 (No Wait)
Address + 2	Status Pointer Memory Type	Specifies the memory type for the location where the COMMREQ status word (CSR) returned by the device will be written when the COMMREQ completes.
Address + 3	Status Pointer Offset	The word at address + 3 contains the offset for the status word within the selected memory type. <b>Note:</b> The status pointer offset is a zero-based value. For example, %R00001 is at offset zero in the register table.
Address + 4	Idle Timeout Value	This parameter is ignored in No Wait mode.
Address + 5	Maximum Communication Time	This parameter is ignored in No Wait mode.
Address + 6 to Address + 133	Data Block	The data block contains the command's parameters. The data block begins with a command number in address + 6, which identifies the type of communications function to be performed. Refer to the specific device manual for COMMREQ command formats.

### Status Pointer Memory Type

Status pointer memory type contains a numeric code that specifies the user reference memory type for the status word. The table below shows the code for each reference type:

For this memory type		Enter this decimal value
%I	Discrete input table (BIT mode)	70
%Q	Discrete output table (BIT mode)	72
%I	Discrete input table (BYTE mode)	16
%Q	Discrete output table (BYTE mode)	18
%R	Register memory	8
%W	Word memory	196
%AI	Analog input table	10
%AQ	Analog output table	12

Notes:

- The value entered determines the mode. For example, if you enter the %I bit mode is 70, then the offset will be viewed as that bit. On the other hand, if the %I value is 16, then the offset will be viewed as that byte.
- The high byte at address + 2 should contain zero.

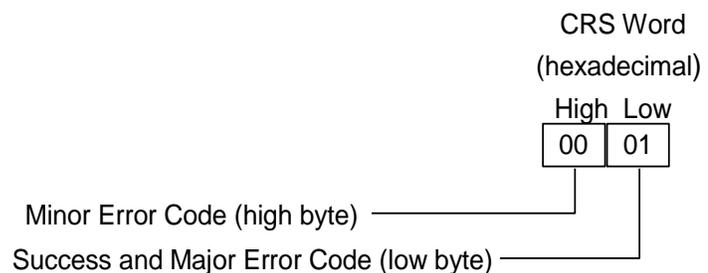
### Operands for COMMREQ

Parameter	Description	Allowed Operands	Optional
IN	The reference of the first WORD of the command block.	Variables in %R, %P, %L, %AI, %AQ, %W, and symbolic non-discrete variables	No
SYSID	The rack number (most significant byte) and slot number (least significant byte) of the target device (intelligent module). <b>Note:</b> For systems that do not have expansion racks, SYSID must be zero for the main rack.	All except flow and variables in %S - %SC	No
TASK	The task ID of the process on the target device	Constants; variables in %R, %P, %L, %AI, %AQ, %W, and symbolic non-discrete variables	No
FT	Function Faulted output. FT is energized if an error is detected processing the COMMREQ: <ul style="list-style-type: none"> <li>▪ This is a WAIT mode COMMREQ and the CPU does not support it</li> <li>▪ The specified target address (SYSID operand) is not present.</li> <li>▪ The specified task (TASK operand) is not valid for the device.</li> <li>▪ The data length is 0.</li> <li>▪ The devices status pointer address (part of the command block) does not exist. This may be due to an incorrect memory type selection, or an address within that memory type that is out of range.</li> </ul>	Power flow	Yes

### COMMREQ Status Word

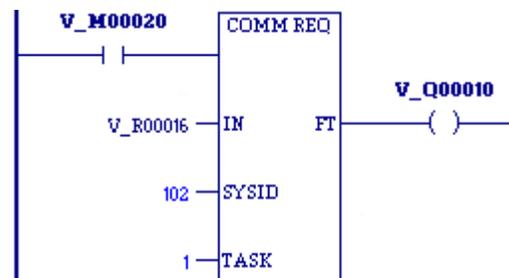
The CRS word consists of two byte values, a major code and a minor code.

Refer to the specific device manual for CRS major and minor codes used by COMMREQ commands at that device.



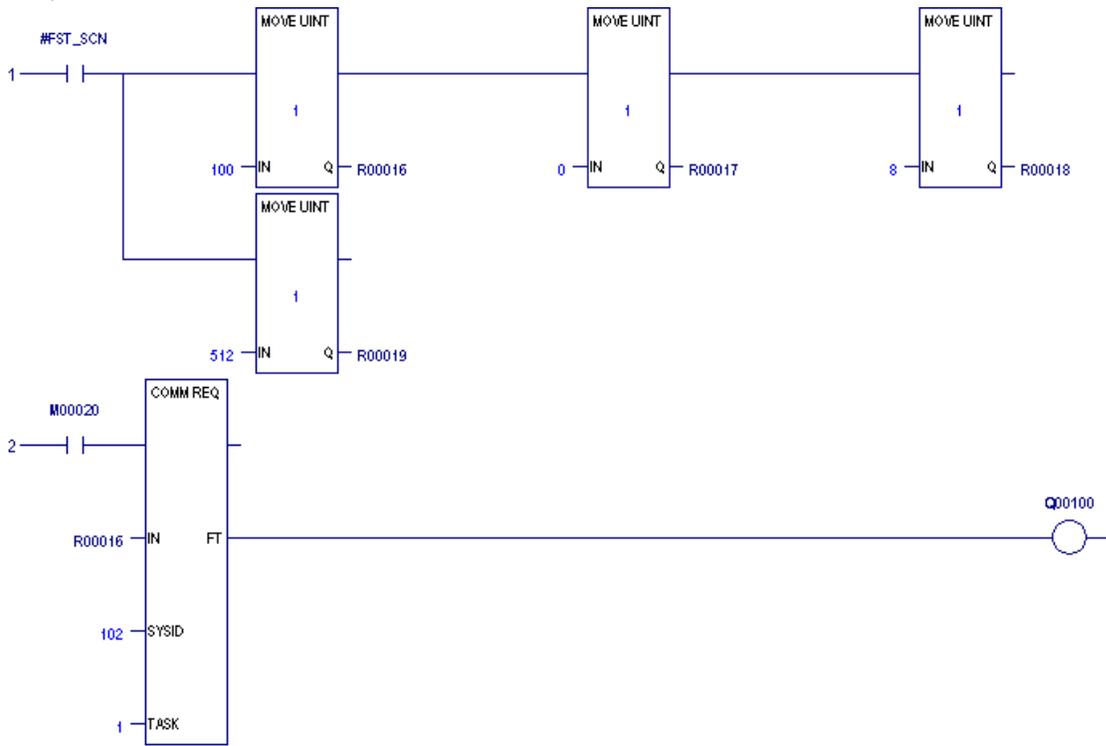
### COMMREQ Example 1

When enabling input %M0020 is ON, a command block starting at %R0016 is sent to communications task 1 in the device located at rack 1, slot 2 of the PLC. If an error occurs processing the COMMREQ, %Q0100 is set.



**COMMREQ Example 2**

The MOVE function can be used to enter the command block contents for the COMMREQ described in example 1.



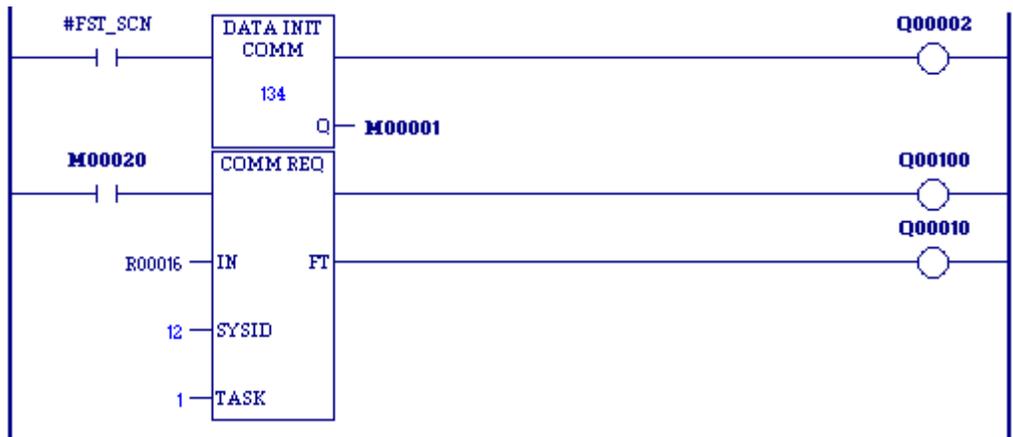
Input IN of the COMMREQ specifies %R00016 as the beginning reference for the command block. Successive references contain the following:

%R00016	Data Block Length
%R00017	Wait/No Wait Flag
%R00018	Status Pointer Memory Type
%R00019	Status Pointer Offset
%R00020	Idle Timeout Value (Because this parameter is ignored in NO WAIT mode, no value is input).
%R00021	Maximum Communication Time Value (Because this parameter is ignored in NO WAIT mode, no value is input).
%R00022 to end of data	Data Block

MOVE functions supply the following command block data for the COMMREQ.

- The first MOVE function places the length of the data being communicated in %R00016.
- The second MOVE function places the constant 0 in %R00017. This specifies NO WAIT mode.
- The third MOVE function places the constant 8 in %R00018. This specifies the register table as the location for the status pointer.
- The fourth MOVE function places the constant 512 in reference %R00019. Therefore, the status pointer is located at %R00513.

The programming logic displayed in example 2 can be simplified by replacing the six MOVE functions with one DATA\_INIT\_COMM function.



### 4.8.7 Data Initialization



The Data Initialization (DATA\_INIT) function copies a block of constant data to a reference range.

When the DATA\_INIT instruction is first programmed, the constants are initialized to zeroes. To specify the constant data to copy, double-click the DATA\_INIT instruction in the LD editor.

**Note:** The mnemonics DATA\_INIT\_ASCII (refer to *Data Initialize ASCII*) and DATA\_INIT\_COMM (refer to *Data Initialize Communications Request*) operate differently from the other six functions.

- Mnemonics:
- DATA\_INIT\_DWORD
  - DATA\_INIT\_DWORD
  - DATA\_INIT\_INT
  - DATA\_INIT\_UINT
  - DATA\_INIT\_REAL
  - DATA\_INIT\_LREAL
  - DATA\_INIT\_WORD

When DATA\_INIT receives power flow, it copies the constant data to output Q. DATA\_INIT's constant data length (LEN) specifies how much constant data of the function type is copied to consecutive reference addresses starting at output Q. DATA\_INIT passes power to the right whenever it receives power.

Notes:

- The output parameter is not included in coil checking.
- If you replace one DATA\_INIT instruction (except DATA\_INIT\_ASCII or DATA\_INIT\_COMM) with another (except DATA\_INIT\_ASCII or DATA\_INIT\_COMM), Logic Developer - PLC attempts to keep the same data. For example, configuring a DATA\_INIT\_INT with eight rows and then replacing the instruction with a DATA\_INIT\_DINT would keep the data for the eight rows. Some precision may be lost when replacing a DATA\_INIT\_ instruction, and a warning message will be displayed when this case is detected.

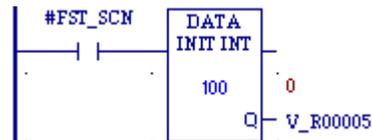
#### Operands

**Note:** For each mnemonic, use the corresponding data type for the Q operand. For example, DATA\_INIT\_DINT requires Q to be a DINT variable.

Parameter	Description	Allowed Operands	Optional
Length	The quantity (default 1) of constant data copied to consecutive reference addresses starting at output Q.	Constants	No
Q	The beginning address of the area to which the data is copied.	All, except %S. SA, SB, and SC are not allowed for REAL, LREAL, INT, and UINT versions.	No

#### Example

On the first scan (as restricted by the #FST\_SCN system variable), 100 words of initial data are copied to %R00005 through %R00104.



### 4.8.8 Data Initialize ASCII



The Data Initialize ASCII (DATA\_INIT\_ASCII) function copies a block of constant ASCII text to a reference range.

When DATA\_INIT\_ASCII is first programmed, the constants are initialized to zeroes. To specify the constant data to copy, double-click the DATA\_INIT\_ASCII instruction in the LD editor.

When DATA\_INIT\_ASCII receives power flow, it copies the constant data to output Q.

DATA\_INIT\_ASCII's constant data length (LEN) specifies how many bytes of constant text are copied to consecutive reference addresses starting at output Q. LEN must be an even number. DATA\_INIT\_ASCII passes power to the right whenever it receives power.

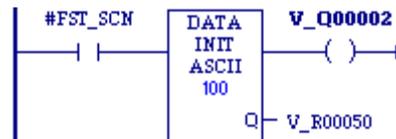
**Note:** The output parameter is not included in coil checking.

#### Operands

Parameter	Description	Allowed Operands	Optional
Length	The number (default 1) of bytes of constant text copied to consecutive reference addresses starting at output Q. LEN must be an even number.	Constants	No
Q	The beginning address of the area where the data is copied.	All except %S.	No

#### Example

On the first scan (as restricted by the #FST\_SCN system variable) the decimal equivalent of 100 bytes of ASCII text is copied to %R00050 through %R00149. %Q00002 receives power.



### 4.8.9 Data Initialize Communications Request



The Data Initialize Communications Request (DATA\_INIT\_COMM) function initializes a COMMREQ function with a block of constant data. The IN parameter of the COMMREQ must correspond with output Q of this DATA\_INIT\_COMM function.

When DATA\_INIT\_COMM is first programmed, the constants are initialized to zeroes. To specify the constant data to copy, double-click the DATA\_INIT\_COMM instruction in the LD editor.

When DATA\_INIT\_COMM receives power flow, it copies the constant data to output Q.

DATA\_INIT\_COMM's constant data length operand specifies how many words of constant data to copy to consecutive reference addresses starting at output Q. The length should be equal to the size of the COMMREQ function's entire command block. DATA\_INIT\_COMM passes power to the right whenever it receives power.

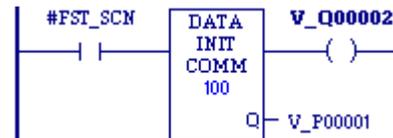
**Note:** The output parameter is not included in coil checking.

#### Operands

Parameter	Description	Allowed Operands	Optional
Length	The number of WORDs (default 7) of constant data copied to consecutive reference addresses starting at output Q. Must equal the size of the COMMREQ function's entire command block, including the header (words 0-5).	Constant	No
Q	The beginning address of the area where the data is copied.	R, W, P, L, AI, AQ, and symbolic non-discrete variables	No

#### Example

On the first scan (as restricted by the #FST\_SCN system variable), a command block consisting of 100 words of data, including the 6 header words, is copied to %P00001 through %P00100. %Q00002 receives power.



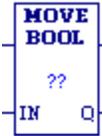
### 4.8.10 Data Initialize DLAN

The Data Initialize DLAN (DATA\_INIT\_DLAN) function is used with a DLAN Interface module, which is a limited availability, specialty system. If you have a DLAN system, refer to the *DLAN/DLAN+ Interface Module User's Manual*, GFK-0729, for details.

#### Operands

Parameter	Description	Allowed Operands	Optional
Q	The beginning address of the area where the data is copied.	Flow, R, W, P, L, AI, AQ, and symbolic non-discrete variables	No

### 4.8.11 Move



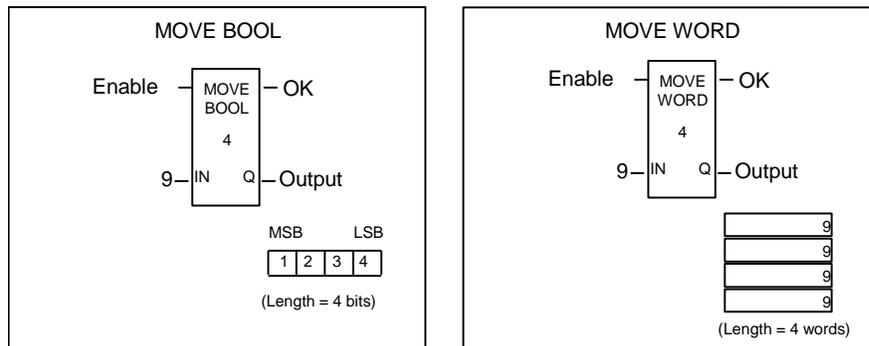
When the MOVE function receives power flow, it copies data as individual bits from one location in PLC memory to another. Because the data is copied in bit format, the new location does not need to be the same data type as the original.

- Mnemonics:
- MOVE\_BOOL
  - MOVE\_DINT
  - MOVE\_DWORD
  - MOVE\_INT
  - MOVE\_REAL
  - MOVE\_UINT
  - MOVE\_WORD

The MOVE function copies data from input operand IN to output operand Q as bits. If data is moved from one location in BOOL (discrete) memory to another, for example, from %I memory to %T memory, the transition information associated with the BOOL memory elements is updated to indicate whether or not the MOVE operation caused any BOOL memory elements to change state. Data at the input operand does not change unless there is an overlap in the source and destination.

**Note:** If an array of BOOL-type data specified in the Q operand does not include all the bits in a byte, the transition bits associated with that byte (which are not in the array) are cleared when the Move function receives power flow. The input IN can be either a variable providing a reference for the data to be moved or a constant. If a constant is specified, then the constant value is placed in the location specified by the output reference. For example, if a constant value of 4 is specified for IN, then 4 is placed in the memory location specified by Q. If the length is greater than 1 and a constant is specified, then the constant is placed in the memory location specified by Q and the locations following, up to the length specified. Do not allow overlapping of IN and Q operands.

The result of the MOVE depends on the data type selected for the function, as shown below. For example, if the constant value 9 is specified for IN and the length is 4, then 9 is placed in the bit memory location specified by Q and the three locations following:



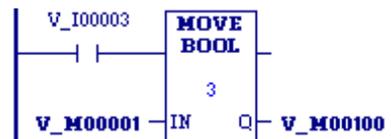
The MOVE function passes power to the right whenever it receives power.

### MOVE Operands

Parameter	Description	Allowed Operands	Optional
Length (??)	The length of IN; the number of bits, words, or double words to copy. If IN is a constant and Q is BOOL, then $1 \leq \text{Length} \leq 16$ ; otherwise, $1 \leq \text{Length} \leq 256$ . $1 \leq \text{Length} \leq 32,767$	Constant	No
IN	The location of the first data item to copy. For MOVE_BOOL, any discrete reference may be used. It does not need to be byte-aligned. However, 16 bits beginning with the reference address specified are displayed online. If IN is a constant, it is treated as an array of bits. The value of the least significant bit is copied into the memory location specified by Q. If Length is greater than one, the bits are copied in order from the least significant to the most significant into successive memory locations, up to the length specified.	All. %S, %SA, %SB, %SC allowed only for WORD, DWORD, BOOL types.	No
Q	The location of the first destination data item. For MOVE_BOOL, any discrete reference may be used. It does not need to be byte-aligned. However, 16 bits beginning with the reference address specified are displayed online.	All except %S. Also no %SA, SB, SC except for WORD, DWORD, BOOL types.	No

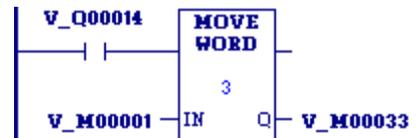
#### MOVE\_BOOL Example

When %I00003 is set, the three bits %M00001, %M00002, and %M00003 are moved to %M00100, %M00101, and %M00102, respectively. Coil %Q00001 is turned on.

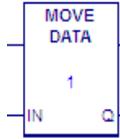


#### MOVE\_WORD Example

V\_M00001 and V\_M00033 are both WORD arrays of length 3, for a total of 48 bits in each array. Since PLCs do not recognize arrays, Length has to be set at 3, for the total number of WORDs to be moved. When enabling input V\_Q0014 is ON, MOVE\_WORD moves 48 bits from the memory location %M00001 to memory location %M00033. Even though the destination overlaps the source for 16 bits, the move is done correctly.



### 4.8.12 Move Data

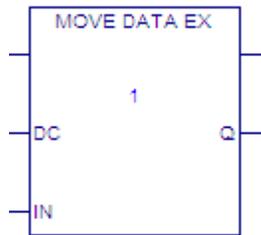


The MOVE\_DATA function copies the variable assigned to the input, IN to the variable assigned to the output, Q. If the constant 0 is assigned to IN, the variable assigned to Q is initialized to its default value. Mnemonic: MOVE\_DATA

#### MOVE\_DATA Operands

Parameter	Description	Allowed Operands	Optional
Length (??)	The length of IN; the number of elements to copy. $1 \leq \text{Length} \leq 32,767$	Constant	No
IN	The location of the data item to copy. If IN is 0, Q is set to its default value.	Enumerated variable, structure variable, or array of these types; the constant 0. For details, refer to <i>Data Types and Structures</i> in the <i>PACMotion Multi-Axis Motion Controller User's Manual</i> , GFK-2448.	No
Q	The location of the data copied from IN. Q must be the same data type as IN, unless IN is the constant 0.	Enumerated variable, structure variable, or array of these types.	No

### 4.8.13 Move Data Explicit



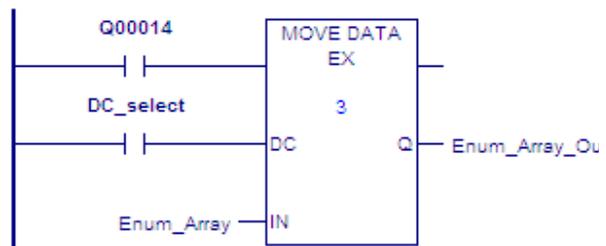
MOVE\_DATA\_EX provides optional data coherency by locking the symbolic memory being written to during the copy operation. This allows data to be copied coherently when accessed by multiple logic threads (i.e. interrupt blocks). Note that copying large amounts of data with coherency enabled can increase interrupt latency.

#### MOVE\_DATA\_EX Operands

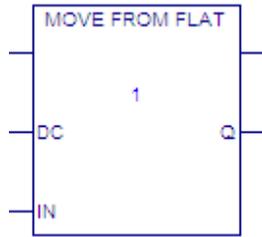
Parameter	Description	Allowed Operands	Optional
Length (??)	The length of IN; the number of elements to copy. $1 \leq \text{Length} \leq 32,767$	Constant	No
DC	Data coherency. If True memory being written to is locked, enabling coherent copying of data from one Controller memory area to another. If False (default), data is copied normally from one Controller memory area to another <i>without</i> data coherency. <ul style="list-style-type: none"> <li>The input DC should be used only when using interrupt blocks and is required only when the same memory is used in more than one interrupt block, or in the main program and an interrupt block.</li> <li>If DC is True, an interrupt block cannot preempt the copy operation.</li> <li>If DC is False or not present, then interrupts can preempt the copy.</li> <li>Using DC can impact interrupt latency if the amount of data copied is large.</li> </ul>	Data flow.	Yes
IN	The location of the data item to copy. If IN is 0 (LD only), length is assigned the constant 1 and the variable or structure assigned to Q is set to its default value.	Enumerated variable or structure variable, or array of these types; the constant 0.	No
Q	Variable or array to which IN is copied. Q must be the same data type as IN, unless IN is the constant 0.	Enumerated variable or structure variable, or array of these types.	No

#### Example

Enum\_Array and Enum\_Array\_Out are arrays of enumerated variables, with three elements each. To copy all elements in Enum\_Array, input Length should be 3. When the enabling input Q00014 is on, MOVE\_DATA\_EX copies three elements from memory location Enum\_Array to memory location Enum\_Array\_Out.



### 4.8.14 Move From Flat



MOVE\_FROM\_FLAT copies reference memory data to a User-defined Data Type (UDT) variable or UDT array.

MOVE\_FROM\_FLAT provides optional data coherency by locking the data being written to during the copy operation. This allows data to be copied coherently when accessed by multiple logic threads (i.e. interrupt blocks). Note that copying large amounts of data with coherency enabled can increase interrupt latency.

#### Operation

##### Copying arrays and array elements

The constant value assigned to input LEN (Length) determines the number of UDT array elements to be filled by copying data from reference memory to output Q.

##### Example:

If constant value 6 is assigned to input LEN (Length), there should be a UDT array of at least six elements assigned to output Q. During logic execution,  $n$  bytes of data are copied from reference memory to the first six UDT array elements, where  $n$  is the length of the UDT array element (in bytes) times six.

##### Copying to specified array elements

For output Q, a single element of a UDT array can be specified, for example, myUDT\_array[4] (5th element of myUDT\_array). In this case, the input LEN (Length) operand applies to the array elements starting from and including myUDT\_array[4].

##### Example:

myUDT\_array is a UDT array of ten elements, of which each element is a UDT variable, and myUDT\_array[4] is assigned to output Q. This restricts the value of input LEN (Length) to six or less because there are six remaining UDT array elements that can be filled in myUDT\_array.

#### Notes:

- Length determines how many UDT variable elements to overwrite in Q.
- If an array head is assigned to input IN, the Length determines how many UDT array elements assigned to Q are filled by copying data from reference memory.

### MOVE\_FROM\_FLAT Operands

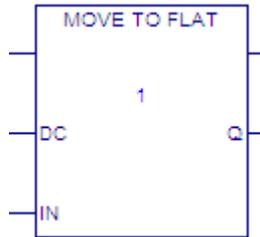
Parameter	Description	Allowed Operands	Optional
Length (??)	Determines the number of UDT array elements to be filled by copying data from reference memory to output Q. $1 \leq \text{Length} \leq 32,767$	Constant	No
DC	Data coherency. If True, memory being written to is locked, enabling coherent copying of data from one Controller memory area to another. If False (default), data is copied normally from one Controller memory area to another; that is <i>without</i> data coherency. <ul style="list-style-type: none"> <li>▪ The input DC should be used only when using interrupt blocks and is required only when the same memory is used in more than one interrupt block, or in the main program and an interrupt block.</li> <li>▪ If DC is True, an interrupt block cannot preempt the copy operation.</li> <li>▪ If DC is False or not present, then interrupts can preempt the copy.</li> <li>▪ Using DC can impact interrupt latency if the amount of data copied is large.</li> </ul>	Data flow.	Yes
IN	Reference memory data being copied to UDT variable elements in output Q as determined by the Length.	All except %S, symbolic, or I/O variable.	No
Q	UDT variable or UDT array to which IN is copied.	Discrete or non-discrete symbolic, discrete or non-discrete I/O variable.	No

#### Example

A WORD variable mapped to %R1 is assigned to input IN and a value of 1 is assigned to Length. A UDT variable or UDT array is assigned to output Q.

When MOVE\_FROM\_FLAT executes,  $n$  bytes of data are copied, starting at %R1 to a UDT variable or UDT array, where  $n$  is the UDT array element length (in bytes). If a UDT array is assigned to output Q,  $n$  bytes of data are copied to the first UDT array element.

### 4.8.15 Move to Flat



MOVE\_TO\_FLAT instruction copies data from symbolic or I/O variable memory to reference memory. MOVE\_TO\_FLAT copies across mismatched data types for an operation such as a Modbus transfer.

MOVE\_TO\_FLAT provides optional data coherency by locking the reference memory being written to during the copy operation. This allows data to be copied coherently when accessed by multiple logic threads (i.e. interrupt blocks). Note that copying large amounts of data with coherency enabled can increase interrupt latency.

Notes:

- The Data Coherency (DC) input should be used only when using interrupt blocks and is required only when the same memory is used in more than one interrupt block, or in the main program and an interrupt block.
- If DC is True, an interrupt block cannot preempt the copy operation.
- If DC is False or not present, then interrupts can preempt the copy.
- Using DC can impact interrupt latency if the amount of data copied is large.

#### Copying Arrays and Array Elements

The Length determines the number of UDT array elements to be copied to the reference memory of the variable assigned to output Q.

**Example:** If the value 6 is assigned to Length, there should be a UDT array of at least six elements assigned to input IN. When logic executes,  $n$  bytes of data are copied from the UDT array elements to the reference memory of the variable assigned to output Q, where  $n$  is the length of the UDT array element (in bytes) times six.

## MOVE\_TO\_FLAT Operands

Parameter	Description	Allowed Operands	Optional
Length (??)	The length of IN; the number of elements to copy. $1 \leq \text{Length} \leq 32,767$	Constant	No
DC	Data coherency. If True, the memory being written to is locked. This enables a coherent copy of a UDT to reference memory. If False (default), data is copied normally from one Controller memory area to another; that is <i>without</i> data coherency. <ul style="list-style-type: none"> <li>▪ DC should be used only when using interrupt blocks and is required only when the same memory is used in more than one interrupt block, or in the main program and an interrupt block.</li> <li>▪ If DC is True, an interrupt block cannot preempt the copy operation.</li> <li>▪ If DC is False or not present, interrupts can preempt the copy.</li> <li>▪ Using DC can impact interrupt latency if the amount of data copied is large.</li> </ul>	Data flow.	Yes
IN	UDT variable or UDT array. The data copied to the reference memory mapped to the variable assigned to Q. If IN is 0, length is assigned the constant 1 and the variable or structure assigned to Q is set to its default value.	Discrete or non-discrete symbolic, discrete or non-discrete I/O variable.	No
Q	Variable or array to which IN is copied. The amount of data copied is determined by the constant value assigned to input LEN (Length).	All memory areas except %S, discrete symbolic, discrete I/O variable. <ul style="list-style-type: none"> <li>▪ Indirect referencing is available for all register references (%R, %P, %L, %W, %AI, and %AQ).</li> <li>▪ BYTE arrays must be packed; that is, they must be in discrete memory.</li> </ul>	No

**Example**

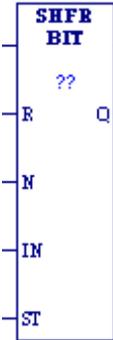
A UDT variable or UDT array is assigned to input IN.

The constant value 8 is assigned to input LEN (Length).

A DWORD variable mapped to %R1 is assigned to output Q.

If the constant value 8 is assigned to LEN (length), there should be a UDT array of at least eight elements assigned to IN. When MOVE\_TO\_FLAT executes,  $n$  bytes of data are copied from the UDT variable or array to %R memory, starting at %R1 in the example, where  $n$  is the length of a UDT array element (in bytes) times eight.

### 4.8.16 Shift Register



When the Shift Register (SHFR\_BIT, SHFR\_DWORD, or SHFR\_WORD) function receives power and the R operand does not, SHFR shifts one or more data BITS, data DWORDS, or data WORDs from a reference location into a specified area of memory. A contiguous section of memory serves as a shift register. For example, one word might be shifted into an area of memory with a specified length of five words. As a result of this shift, another word of data would be shifted out of the end of the memory area.

Mnemonics:  
SHFR\_BIT  
SHFR\_DWORD  
SHFR\_WORD



#### Warning

**The use of overlapping input and output reference address ranges in multiword functions is not recommended, as it may produce unexpected results.**

The reset input (R) takes precedence over the function enable input. When the reset is active, all references beginning at the shift register (ST) up to the length specified, are filled with zeroes.

If the function receives power flow and R is not active, each BIT, DWORD, or WORD of the shift register is moved to the next highest reference. The elements shifted out of ST are shifted into Q. The highest reference of IN is shifted into the vacated element starting at ST.

**Note:** The contents of the shift register are accessible throughout the program because they are overlaid on absolute locations in logic addressable memory.

The function passes power to the right whenever it receives power flow and the R operand does not.

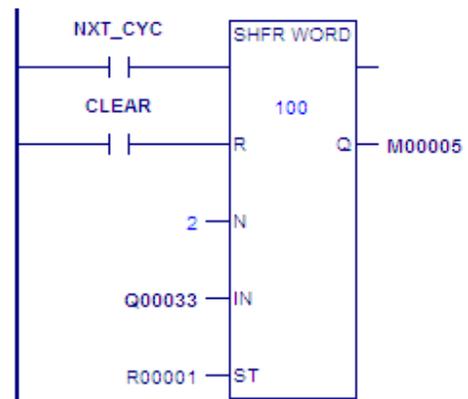
### Operands for Shift Register

Parameter	Description	Allowed Operands	Optional
Length (??)	The number of data items in the shift register, ST. $1 \leq \text{Length} \leq 256$		No
R	Reset. When R is ON, the shift register located at ST is filled with zeroes.	Power flow	No
N	The number of data items to shift into ST.	Constants	No
IN	The value to shift into the first data item of ST. <b>SHFR_BIT:</b> For %I, %Q, %M and %T memory, any BOOL reference may be used; it does not need to be byte-aligned. However, 1 bit, beginning with the reference address specified, is displayed online.	All	No
ST	The first data item of the shift register. <b>Note:</b> For %I, %Q, %M and %T memory, any BOOL reference may be used; it does not need to be byte-aligned. However, 16 bits, beginning with the reference address specified, are displayed online.	All except data flow, constants, S	No
Q	The data shifted out of ST. The same number of data items will be shifted into Q as were shifted out of ST. <b>SHFR_BIT:</b> For %I, %Q, %M and %T memory, any BOOL reference may be used; it does not need to be byte-aligned. However, 1 bit, beginning with the reference address specified, is displayed online.	All except S	No

#### Example

SHFR\_WORD operates on register memory locations %R0001 through %R0100. When the reset reference CLEAR is active, the Shift Register words are set to zero.

When the NXT\_CYC reference is active and CLEAR is not, the two words at the starting address V\_Q00033 are shifted into the Shift Register at %R0001. The words shifted out of the Shift Register from %R0100 are stored in output %M0005. Note that, for this example, the length specified and the amount of data to be shifted (N) are not the same.



### 4.8.17 Size Of



Counts the number of bits used by the variable assigned to input IN and writes the number of bits to output Q.

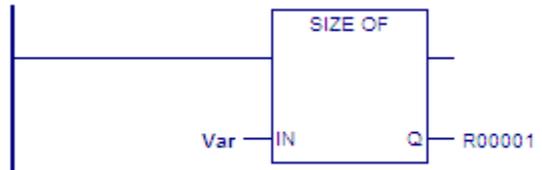
Mnemonics:  
SIZE\_OF

#### Operands

Parameter	Description	Allowed Operands	Optional
IN	The variable whose size in bits is calculated.	Variable of any data type except BYTE arrays in non-discrete memory and double-segment structures.	No
Q	The number of bits used by the variable assigned to input IN.	DINT or DWORD variable. ST also supports INT and WORD variables.	No

#### Example

The single-segment structure named Var assigned to input IN contains eight BOOL elements ( $8 \times 1 = 8$  bits) and twelve WORD elements ( $12 \times 16 = 192$  bits). SIZE\_OF outputs the value  $8 + 192 = 200$  to the variable R00001 assigned to output Q.



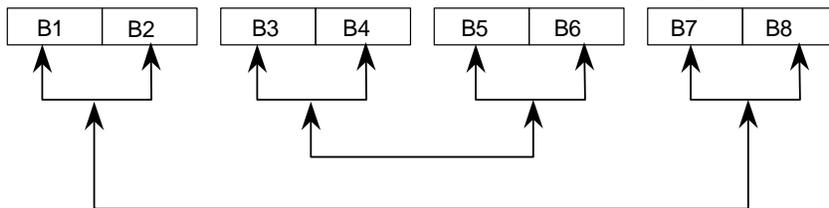
### 4.8.18 Swap

 The SWAP function is used to swap two bytes within a word (SWAP WORD) or two words within a double word (SWAP DWORD). The SWAP can be performed over a wide range of memory by specifying a length greater than 1. If that is done, the data in each word or double word within the specified length is swapped.

When the SWAP function receives power flow, it swaps the data in reference IN and places the swapped data into output reference Q. The function passes power to the right whenever it receives power.

PACSystems CPUs use the Intel convention for storing word data in bytes. They store the least significant byte of a word in address n and the most significant byte in address n+1. Many VME modules follow the Motorola convention of storing the most significant byte in address n and the least significant byte in address n+1.

The PACSystems CPU assigns byte address 1 to the same storage location regardless of the byte convention used by the other device. However, because of the difference in byte significance, word and multiword data, for example, 16-bit integers (INT, UINT), 32-bit integers (DINT) or floating point (REAL) numbers, must be adjusted when being transferred to or from Motorola-convention modules. In these cases, the two bytes in each word must be swapped, either before or after the transfer. In addition, for multiword data items, the words must be swapped end-for-end on a word basis. For example, a 64-bit real number transferred to the PACSystems CPU from a Motorola-convention module must be byte-swapped and word-reversed, either before or after reading, as shown below:



Character (ASCII) strings or BCD data require no adjustment since the Intel and Motorola conventions for storage of character strings are identical.

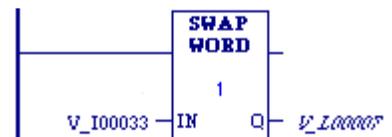
#### Operands for Swap

The two parameters, IN and Q, must both be the same type, WORD or DWORD.

Parameter	Description	Allowed Operands	Optional
Length (??)	The number of WORDs or DWORDs to operate on. $1 \leq \text{Length} \leq 256$	Constant	No
IN	Reference for data to be swapped. (must be the same type as Q)	All	No
Q	Reference for swapped data. (must be the same type as IN)	All except S	No

#### Example for Swap

Two bytes located in bits %I00033 through %I00048 are swapped. The result is stored in %L00007.



## 4.9 Data Table Functions

Function	Mnemonic	Description
Array Move	ARRAY_MOVE_BOOL ARRAY_MOVE_BYTE ARRAY_MOVE_DINT ARRAY_MOVE_INT ARRAY_MOVE_WORD	Copies a specified number of data elements from a source memory block to a destination memory block. <b>Note:</b> The memory blocks do not need to be defined as arrays. You must supply a starting address and the number of contiguous registers to use for the move.
Array Range	ARRAY_RANGE_DINT ARRAY_RANGE_DWORD ARRAY_RANGE_INT ARRAY_RANGE_UINT ARRAY_RANGE_WORD	Determines if a value is between the range specified in two tables
FIFO Read	FIFO_RD_DINT FIFO_RD_DWORD FIFO_RD_INT FIFO_RD_UINT FIFO_RD_WORD	Removes the entry at the bottom of the First In First Out (FIFO) table, and decrements the pointer by one
FIFO Write	FIFO_WRT_DINT FIFO_WRT_DWORD FIFO_WRT_INT FIFO_WRT_UINT FIFO_WRT_WORD	Increments the table pointer and writes data to the bottom of the FIFO table
LIFO Read	LIFO_RD_DINT LIFO_RD_DWORD LIFO_RD_INT LIFO_RD_UINT LIFO_RD_WORD	Removes the entry at the pointer location in the LIFO (Last In First Out) table, and decrements the pointer by one
LIFO Write	LIFO_WRT_DINT LIFO_WRT_DWORD LIFO_WRT_INT LIFO_WRT_UINT LIFO_WRT_WORD	Increments the LIFO table's pointer and writes data to the table
Search	SEARCH_EQ_BYTE SEARCH_EQ_DINT SEARCH_EQ_DWORD SEARCH_EQ_INT SEARCH_EQ_UINT SEARCH_EQ_WORD	Searches for all array values equal to a specified value
	SEARCH_GE_BYTE SEARCH_GE_DINT SEARCH_GE_DWORD SEARCH_GE_INT SEARCH_GE_UINT SEARCH_GE_WORD	Searches for all array values greater than or equal to a specified value

Function	Mnemonic	Description
	SEARCH_GT_BYTE SEARCH_GT_DINT SEARCH_GT_DWORD SEARCH_GT_INT SEARCH_GT_UINT SEARCH_GT_WORD	Searches for all array values greater than a specified value
	SEARCH_LE_BYTE SEARCH_LE_DINT SEARCH_LE_DWORD SEARCH_LE_INT SEARCH_LE_UINT SEARCH_LE_WORD	Searches for all array values less than or equal to a specified value
	SEARCH_LT_BYTE SEARCH_LT_DINT SEARCH_LT_DWORD SEARCH_LT_INT SEARCH_LT_UINT SEARCH_LT_WORD	Searches for all array values less than a specified value
	SEARCH_NE_BYTE SEARCH_NE_DINT SEARCH_NE_DWORD SEARCH_NE_INT SEARCH_NE_UINT SEARCH_NE_WORD	Searches for all array values not equal to a specified value
Sort	SORT_INT SORT_UINT SORT_WORD	Sorts a memory block in ascending order
Table Read	TBL_RD_DINT TBL_RD_DWORD TBL_RD_INT TBL_RD_UINT TBL_RD_WORD	Copies a value from a specified table location to an output reference
Table Write	TBL_WRT_DINT TBL_WRT_DWORD TBL_WRT_INT TBL_WRT_UINT TBL_WRT_WORD	Copies a value from an input reference to a specified table location

### 4.9.1 Array Move



When the Array Move function receives power flow, it copies a specified number of elements from a source memory block to a destination memory block. Starting at the indexed location (SR+SNX-1) of the input memory block, it copies N elements to the output memory block, starting at the indexed location (DS+DNX-1) of the output memory block.

Mnemonics:  
 ARRAY\_MOVE\_BOOL  
 ARRAY\_MOVE\_BYTE  
 ARRAY\_MOVE\_DINT  
 ARRAY\_MOVE\_DWORD  
 ARRAY\_MOVE\_INT  
 ARRAY\_MOVE\_UINT  
 ARRAY\_MOVE\_WORD

**Note:** For ARRAY\_MOVE\_BOOL, when 16-bit registers are selected for the operands of the source memory block and/or destination memory block starting address, the least significant bit of the specified 16-bit register is the first bit of the memory block. The value displayed contains 16 bits, regardless of the length of the memory block.

The indices in an Array Move instruction are 1-based. In using an Array Move, no element outside either the source or destination memory blocks (as specified by their starting address and length) may be referenced.

The function passes power flow unless one of the following conditions occurs:

- It receives no power flow.
- (N + SNX - 1) is greater than Length.
- (N + DNX - 1) is greater than Length.

**Note:** For each mnemonic, use the corresponding data type for the SR and DS operands. For example, ARRAY\_MOVE\_BYTE requires SR and DS to be BYTE variables.

#### Operands for Array Move

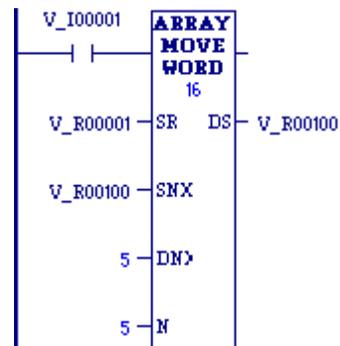
Parameter	Description	Allowed Operands	Optional
Length (??)	The length of each memory block (source and destination); the number of elements in each memory block. $1 \leq \text{Length} \leq 32,767$ .	Constant	No
SR (must be the same data type as DS)	The starting address of the source memory block. <b>Note:</b> For an Array Move with the data type BOOL, any reference may be used; it does not need to be byte-aligned. Sixteen bits, beginning with the reference address specified, are displayed online.	All except constants. %S - %SC allowed only for BYTE, WORD, DWORD types.	No
SNX	The index of the source memory block	All except variables in %S - %SC.	No
DNX	The index of the destination memory block	All except variables in %S - %SC.	No
N	Count indicator	All except variables in %S - %SC	No

Parameter	Description	Allowed Operands	Optional
DS (must be the same data type as SR)	The starting address of the destination memory block. <b>Note:</b> For an Array Move with the data type BOOL, any reference may be used; it does not need to be byte-aligned. Sixteen bits, beginning with the reference address specified, are displayed online.	All, except S and constants. %SA - %SC allowed only for BYTE, WORD, DWORD types	No

### Array Move Example 1

To define the input memory block %R0001 - %R0016 and the output memory block %R0100 - %R0115, SR is set as %R0001, DS is set as %R0100, and Length is set to 16.

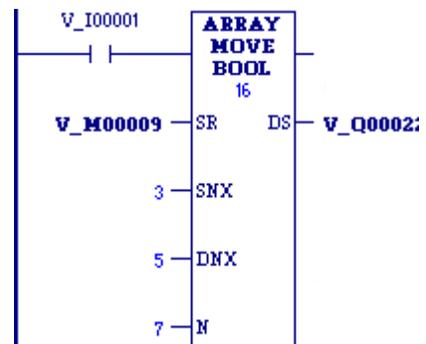
To copy the five registers %R0003 - %R0007 to the registers %R0104 - %R0108, N is set to 5, SNX=%R0100 is set to 3 (to designate the third register, %R0003, of the block starting at %R0001), and DNX is set to 5 (to designate the fifth register, %R0104, of the block starting at %R0100).



### Array Move Example 2

Using bit memory blocks, the input block starts at SR=%M0009, the output block starts at %Q0022, and the length of both blocks is 16 one-bit registers (Length=16).

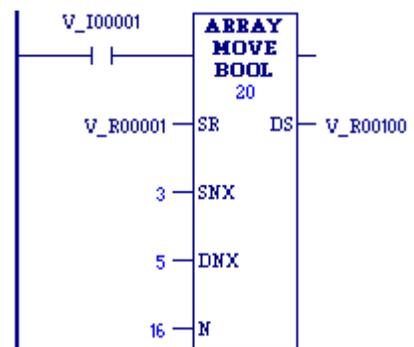
To copy the seven registers %M0011 - %M0017 to %Q0026 - %Q0032, N is set to 7, SNX is set to 3 (to designate the third register, %M0011, of the block starting at %M0009), and DNX is set to 5 (to designate the fifth register, %Q0026, of the block starting at %Q0022).



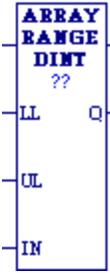
### Array Move Example 3

Sixteen (=N) bits that are not byte-aligned are moved from the two 16-bit registers that start at %R00001 (SR) to the two 16-bit registers that begin at %R00100 (DS). For the purposes of this Boolean move, Length is set to 20, because the other 12 bits in either memory block are not considered.

By setting SNX to 3, N to 16, and DNX to 5, the third (SNX) least significant bit of %R0001 through the second least significant bit of %R0002 (for a total of 16 bits=N) are written into the fifth (DNX) least significant bit of %R0100 through the fourth least significant bit of %R0101 (for the same total of 16 bits).



### 4.9.2 Array Range



The ARRAY\_RANGE function compares a single input value against two arrays of delimiters that specify an upper and lower bound to determine if the input value falls within the range specified by the delimiters. The output is an array of bits that is set ON (1) when the input value is greater than or equal to the lower limit and less than or equal to the upper limit. The output is set OFF (0) when the input is outside this range or when the range is invalid, as when the lower limit exceeds the upper limit.

Mnemonics:  
 ARRAY\_RANGE\_DINT  
 ARRAY\_RANGE\_DWORD  
 ARRAY\_RANGE\_INT  
 ARRAY\_RANGE\_UINT  
 ARRAY\_RANGE\_WORD

The ARRAY\_RANGE function compares a single input value against two arrays of delimiters that specify an upper and lower bound to determine if the input value falls within the range specified by the delimiters. The output is an array of bits that is set ON (1) when the input value is greater than or equal to the lower limit and less than or equal to the upper limit. The output is set OFF (0) when the input is outside this range or when the range is invalid, as when the lower limit exceeds the upper limit.

When ARRAY\_RANGE receives power, it compares the value in input parameter IN against each range specified by the array element values of LL and UL. Output Q sets a bit ON (1) for each corresponding array element where the value of IN is greater than or equal to the value of LL and is less than or equal to the value of UL. Output Q sets a bit OFF (0) for each corresponding array element where the value of IN is not within this range or when the range is invalid, as when the value of LL exceeds the value of UL. If the operation is successful, ARRAY\_RANGE passes power flow to the right.

#### Operands for Array Range

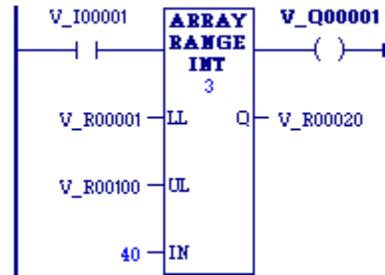
##### Notes:

- For each mnemonic, use the corresponding data type for the LL, UL, and Q operands. For example, ARRAY\_RANGE\_DINT requires LL, UL, and Q to be DINT variables.
- Q is not aligned. It is displayed in bit format. It displays either a 1 (ON) or a 0 (OFF) for the first array element. For BOOL references, it represents the reference displayed. For other references, it represents the low order bit of the reference displayed.

Parameter	Description	Allowed Operands	Optional
Length (??)	The number of elements in each array.	Constant	No
LL	The lower limit of the range	All except constants and %S - %SC for INT, DINT.	No
UL	The upper limit of the range	All except constants and %S - %SC for INT, DINT.	No
IN	The value to compare against each range specified by LL and UL	All except constants and %S - %SC for INT, DINT.	No
Q	Energized when the value in IN is within the range specified by LL and UL, inclusive.	All except S	No

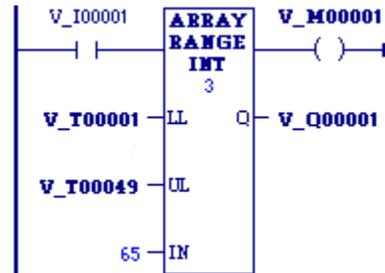
**Array Range Example 1**

The lower limit (LL) values of %R00001 through %R00008 are 1, 20, 30, 100, 25, 50, 10, and 200. The upper limit (UL) values of %R00100 through %R00108 are 40, 50, 150, 2, 45, 90, 250, and 47. The resulting Q values will be placed in the first 8 bits of %R00200. The bit values low order to high are: 1, 1, 1, 0, 1, 0, 1, and 0. The bit value displayed will be set ON (1) for the low order bit of %R00200. The ok output will be set ON (1).



**Array Range Example 2**

The lower limit (LL) array contains %T00001 through %T00016, %T00017 through %T00032, and %T00033 through %T00048. The lower limit values are 100, 65, and 1. The upper limit (UL) values are 29, 165, and 2. The resulting Q values of 0, 1, and 0 will be placed in %Q00001 through %Q00003. The bit value displayed will be 0 (OFF), representing the value of %Q00001. The power output will be set ON (1).



### 4.9.3 FIFO Read



The First-In-First-Out (FIFO) Read (FIFO\_RD) function moves data out of tables. Values are always moved out of the bottom of the table. If the pointer reaches the last location and the table becomes full, FIFO\_RD must be used to remove the entry at the pointer location and decrement the pointer by one. FIFO\_RD is used in conjunction with the FIFO\_WRT function, which increments the pointer and writes entries into the table.

Mnemonics:  
 FIFO\_RD\_DINT  
 FIFO\_RD\_DWORD  
 FIFO\_RD\_INT  
 FIFO\_RD\_UINT  
 FIFO\_RD\_WORD

1. FIFO\_RD copies the top location (entry 0) of the table to output parameter Q. Additional program logic must then be used to place the data in the input reference.
2. The remaining items in the table are copied to a lower numbered position in the table.
3. FIFO\_RD decrements the pointer by one.
4. Steps 1, 2, and 3 are repeated each time FIFO\_RD is executed, until the table is empty (PTR = 0).

The pointer does not wrap around when the table is full.

When FIFO\_RD receives power flow, the data at the first location of the table is copied to output Q. Next, each item in the table is moved down to the next lower location. This begins with item 2 in the table, which is moved into position 1. Finally, the pointer is decremented. If this causes the pointer location to become 0, the output EM is set ON, i.e., EM indicates whether or not the table is empty.

FIFO\_RD passes power to the right if the pointer is greater than zero and less than the value specified for LEN.

**Note:** A FIFO table is a queue. A LIFO table is a stack.

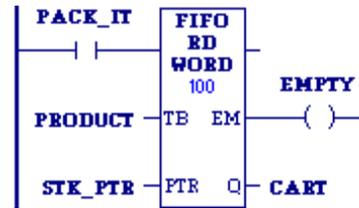
#### Operands for FIFO Read

**Note:** For each mnemonic, use the corresponding data type for the TB and Q operands. For example, FIFO\_RD\_DINT requires TB and Q to be DINT variables.

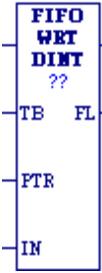
Parameter	Description	Allowed Operands	Optional
Length (??)	$1 \leq \text{Length} \leq 32,767$ .	Constants	No
TB (must be the same type as Q)	The elements in the FIFO table	All except constants	No
PTR	Pointer. Index of the last element of the FIFO table.	All except constants, data flow, and variables in %S-%SC	No
EM	Energized when the last element of the table is read	Flow	No
Q (must be the same type as TB)	The element read from the FIFO table	All except constants, S; SA, SB, SC allowed only for WORD, DWORD	No

**Example for FIFO Read**

PRODUCT is a FIFO table with 100 word-sized elements. When the enabling input PACK\_IT is ON, the PRODUCT data item in the table location pointed to by STK\_PTR is copied to the reference location specified in CART. This table location pointed to would be the bottom, or oldest data item in the table. The number in STK\_PTR is then decremented. A copy of the oldest data item in the PRODUCT table is left behind in each table location as the current data is copied out during successive PACK\_IT triggers. Output node EM passes power when the PTR = 0, firing the coil EMPTY. No further data from the PRODUCT table can be read without first copying data in using the FIFO\_WRT function.



### 4.9.4 FIFO Write



The First-In-First-Out (FIFO) Write (FIFO\_WRT) function moves data into tables. The function increments the table pointer by one and adds an entry at the new pointer location in a FIFO table. Values are always moved in at the bottom of the table. If the pointer reaches the last location and the table becomes full, FIFO\_WRT can add no further values. The FIFO\_RD function must then be used to remove the entry at the pointer location and decrement the pointer by one.

Mnemonics:  
 FIFO\_WRT\_DINT  
 FIFO\_WRT\_DWORD  
 FIFO\_WRT\_INT  
 FIFO\_WRT\_UINT  
 FIFO\_WRT\_WORD

1. FIFO\_WRT increments the pointer by one.
2. FIFO\_WRT copies data from input parameter IN to the position in the table indicated by the pointer. (It writes over any value currently at that location.) Additional program logic must then be used to place the data in the input reference.
3. Steps 1 and 2 are repeated each time FIFO\_WRT is executed, until the table is full (PTR=0).

The pointer does not wrap around when the table is full.

When FIFO\_WRT receives power flow, the pointer is incremented by 1. Then, input data is written into the table at the pointer location. If the pointer was already at the last location in the table, no data is written and FIFO\_WRT does not pass power to the right. The pointer always indicates the last item entered into the table. If the table becomes full, it is not possible to add more entries to it.

FIFO\_WRT passes power to the right after a successful execution (PTR < LEN).

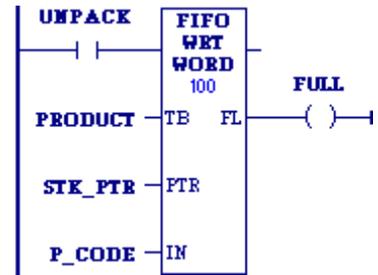
#### Operands for FIFO Write

**Note:** For each mnemonic, use the corresponding data type for the TB and IN operands. For example, FIFO\_WRT\_DINT requires TB and IN to be DINT variables.

Parameter	Description	Allowed Operands	Optional
Length (??)	$1 \leq \text{Length} \leq 32,767$ .	Constants	No
TB (must be the same data type as IN)	The elements in the FIFO table	All except constants, data flow, and S. SA - SC allowed only for WORD, DWORD types	No
PTR	Pointer. Index of the last element of the FIFO table.	All except constants, data flow, S - SC.	No
IN (must be the same data type as TB)	The element to write to the FIFO table	All. S - SC allowed only for WORD, DWORD types.	No
FL	Energized when IN is written to the last element of the table	Power flow	No

**Example for FIFO Write**

PRODUCT is a FIFO table with 100 word-sized elements. When the enabling input UNPACK is ON, a data item from P\_CODE is copied to the table location pointed to by the value in STK\_PTR. Output node FL passes power when PTR = LEN, firing the FULL coil. No further data from P\_CODE can be added to the table without first copying data out, using the FIFO\_RD function.



### 4.9.5 LIFO Read



The Last-In-First-Out (LIFO) Read (LIFO\_RD) function moves data out of tables. Values are always moved out of the top of the table. If the pointer reaches the last location and the table becomes full, LIFO\_RD must be used to remove the entry at the pointer location and decrement the pointer by one. LIFO\_RD is used in conjunction with the LIFO\_WRT function, which increments the pointer and writes entries into the table.

Mnemonics:  
 LIFO\_RD\_DINT  
 LIFO\_RD\_DWORD  
 LIFO\_RD\_INT  
 LIFO\_RD\_UINT  
 LIFO\_RD\_WORD

1. LIFO\_RD copies data indicated by the pointer to output parameter Q. Additional program logic must then be used to place the data in the input reference.
2. LIFO\_RD decrements the pointer by one.
3. Steps 1 and 2 are repeated each time the instruction is executed, until the table is empty (PTR = LEN).

The pointer does not wrap around when the table is full.

When LIFO\_RD receives power flow, the data at the pointer location is copied to output Q, then the pointer is decremented. If this causes the pointer location to become 0, the output EM is set ON, i.e., EM indicates whether or not the table is empty. If the table is empty when LIFO\_RD receives power flow, no read occurs. The pointer always indicates the last item entered into the table.

LIFO\_RD passes power to the right if the pointer was in range for an element to be read.

**Note:** A LIFO table is a stack. A FIFO table is a queue.

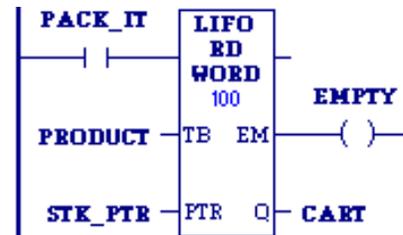
#### Operands for LIFO Read

**Note:** For each mnemonic, use the corresponding data type for the TB and Q operands. For example, LIFO\_RD\_DINT requires TB and Q to be DINT variables.

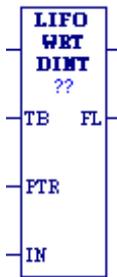
Parameter	Description	Allowed Operands	Optional
Length (??)	$1 \leq \text{Length} \leq 32,767$ .	Constant	No
TB (must be the same type as Q)	The elements in the table	All except constants	No
PTR	Pointer. Index of the next element to read.	All except constants, S - SC, and data flow	No
EM	Energized when the last element of the table is read	Power flow	No
Q (must be the same type as TB)	The element read from the table	All except constants and S. SA, SB, SC allowed only for WORD, DWORD.	No

#### Example for LIFO Read

PRODUCT is a LIFO table with 100 word-sized elements. When the enabling input PACK\_IT is ON, the data item at the top of the table is copied into the reference indicated by the nickname CART. The reference identified by STK\_PTR contains the table pointer. Output coil EMPTY indicates when the table is empty.



### 4.9.6 LIFO Write



The Last-In-First-Out (LIFO) Write (LIFO\_WRT) function increments the table pointer by one and then adds an entry at the new pointer location in a table. Values are always moved in at the top of the table. If the pointer reaches the last location and the table becomes full, LIFO\_WRT cannot add further values. LIFO\_RD must then be used to remove the entry at the pointer location and decrement the pointer by one.

Mnemonics:  
 LIFO\_WRT\_DINT  
 LIFO\_WRT\_DWORD  
 LIFO\_WRT\_INT  
 LIFO\_WRT\_UINT  
 LIFO\_WRT\_WORD

1. LIFO\_WRT increments the table pointer by one.
2. LIFO\_WRT copies data from input parameter IN to the position in the table indicated by the pointer. (It writes over any value currently at that location.) Additional program logic must then be used to place the data in the input reference.
3. Steps 1 and 2 are repeated each time LIFO\_WRT is executed, until the table is full (PTR=LEN).

The pointer does not wrap around when the table is full.

When LIFO\_WRT receives power flow, the pointer increments by 1; then the new data is written at the pointer location. If the pointer was already at the last location in the table, no data is written and LIFO\_WRT does not pass power to the right. The pointer always indicates the last item entered into the table. If the table is full, it is not possible to add more entries to it.

LIFO\_WRT passes power to the right after a successful execution (PTR < LEN).

**Note:** A LIFO table is a stack. A FIFO table is a queue.

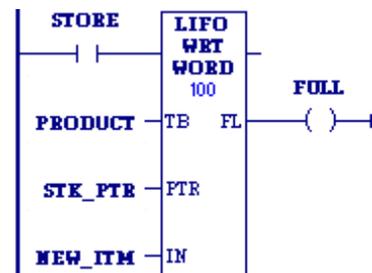
#### Operands for LIFO Write

**Note:** For each mnemonic, use the corresponding data type for the TB and IN operands. For example, LIFO\_WRT\_DINT requires TB and Q to be DINT variables.

Parameter	Description	Allowed Operands	Optional
Length (??)	$1 \leq \text{Length} \leq 32,767$ .	Constants	No
TB (must be the same type as IN)	The elements in the table	All except constants, S, data flow. SA - SC allowed only for WORD, DWORD.	No
PTR	Pointer. Index of the next element to write.	All except constants, S - SC, and data flow	No
IN (must be the same type as TB)	The element to write to the table	All. S - SC allowed only for WORD, DWORD	No
FL	Energized when IN is written to the last element of the table	All	No

#### Example for LIFO Write

PRODUCT is a LIFO table with 100 word-sized elements. When the enabling input STORE is ON, a data item from NEW\_ITEM is copied to the table location pointed to by the value in STK\_PTR. Output FL passes power when PTR = LEN, firing the FULL coil. No further data from NEW\_ITEM can be added to the table without first copying data out, using the LIFO\_RD function.



### 4.9.7 Search



When the Search function receives power, it searches the specified memory block for a value that satisfies the search criteria. For example, SEARCH\_GE\_DWORD searches for a DWORD that is greater than or equal to the specified value (the IN operand).

Search can evaluate six different relationships for six data types, for a total of thirty-six mnemonics.

#### Search Relationships:

- SEARCH\_EQ\_ searches for a value of the specified data type **equal** to the IN operand.
- SEARCH\_GE\_ searches for a value of the specified data type **greater than or equal** to IN.
- SEARCH\_GT\_ searches for a value of the specified data type **greater than** IN.
- SEARCH\_LE\_ searches for a value of the specified data type **less than or equal** to IN.
- SEARCH\_LT\_ searches for a value of the specified data type **less than** IN.
- SEARCH\_NE\_ searches for a value of the specified data type that is **not equal** to IN.

Data types:

BYTE, DINT, DWORD, INT, UINT, WORD

Searching begins at AR+INX, where AR is the starting address and INX is the index value into the memory block. The search continues either until a register that satisfies the search criteria is found or until the end of the memory block is reached.

- If a register is found, the Found Indication (FD) is set ON and the Output Index (ONX) is set to the relative position of this register within the block.
- If no register is found before the end of the block is reached, the Found Indication (FD) is set OFF and the Output Index (ONX) is set to zero.

The input index (INX) is zero-based, that is, 0 means first reference, whereas the output index (ONX) is one-based, that is, 1 means the first reference.

The valid values for INX are 0 to (Length - 1). The valid values for ONX are 1 to Length.

INX should be set to zero to begin searching at the memory block's first register. This value increments by one at the time of execution. If the value of input INX is out-of-range, (< 0 or > Length-1), INX is set to the default value of zero.

SEARCH passes power flow to the right when it performs without error. If INX is out of range, SEARCH does not pass power flow to the right.

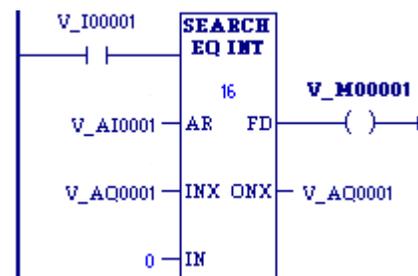
### Operands for the Search Function

**Note:** For each mnemonic, use the corresponding data type for the AR and IN operands. For example, SEARCH\_EQ\_BYTE requires AR and IN to be BYTE variables.

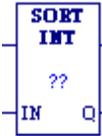
Parameter	Description	Allowed Operands	Optional
Length (??)	The number of registers starting at AR that make up the memory block to search. $1 \leq \text{Length} \leq 32,767$ 8-bit or 16-bit registers.	Constants	No
AR (must be the same type as IN)	The starting address of the memory block to search; the address of the first register in the memory block.	All except constants	No
INX	The zero-based index into the memory block at which to begin the search. Zero points to the first reference. Valid range: $0 \leq \text{INX} \leq (\text{Length}-1)$ . If INX is out of range, it is set to the default value of 0.	All except constants	No
IN (must be the same type as AR)	The value that the search is based on. For example: SEARCH_GT_DINT searches for a DINT value that is greater than IN. SEARCH_NE_UINT searches for a UINT value that is not equal to IN. SEARCH_GE_WORD searches for a WORD value that is greater than or equal to IN.	All	No
ONX	The one-based position within the memory block of the search target. A value of 1 points to the first reference. Valid range: $1 \leq \text{ONX} \leq \text{Length}$	data flow, I, Q, M, T, G, R, P, L, AI, AQ	No
FD	Found indicator. This power flow indicator is energized when a register that satisfies the search criteria is found and the function was successful.	Power flow	No

### Example for the Search Function

To search the memory block %AI00001 - %AI00016, AR is set as %AI00001 and Length is set as 16. The values of the 16 registers are 100, 20, 0, 5, 90, 200, 0, 79, 102, 80, 24, 34, 987, 8, 0, and 500. Initially, the search index into AR, %AQ0001, is 5. When power flow input is ON, each scan searches the memory block looking for a match to the IN value of 0. The first scan starts searching at %AI00006 and finds a match at %AI00007, so FD turns ON and %AQ00001 becomes 7. The second scan starts searching at %AI00008 and finds a match at %AI00015, so FD remains ON and %AQ00001 becomes 15. The next scan starts at %AI00016. Since the end of the memory block is reached without a match, FD is set OFF and %AQ00001 is set to zero. The next scan starts searching at the beginning of the memory block.



### 4.9.8 Sort



When it receives power flow, the SORT function sorts the elements of the memory block 'IN' in ascending order. The output memory block Q contains integers that give the index that the sorted elements had in the original memory block or list. Q is exactly the same size as IN. It also has a specification (LEN) of the number of elements to be sorted.

Mnemonics:  
 SORT\_INT  
 SORT\_UINT  
 SORT\_WORD

SORT operates on memory blocks of no more than 64 elements. When EN is ON, all of the elements of IN are sorted into ascending order, based on their data type. The array Q is also created, giving the original position that each sorted element held in the unsorted array. OK is always set ON.

**Notes:** The SORT function is executed each scan it is enabled.

Do not use the SORT function in a timed or triggered input program block.

#### Operands

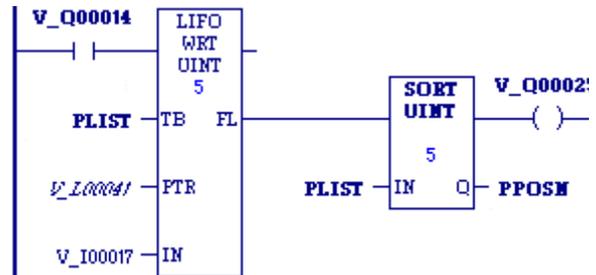
**Note:** For each mnemonic, use the corresponding data type for the IN and Q operands. For example, SORT\_INT requires IN and Q to be INT variables.

Parameter	Description	Allowed Operands	Optional
Length (??)	The number (1–64) of elements that make up the memory block to sort.	Constants	No
IN	The memory block that contains the elements to sort. After the sort, IN contains the elements in the sorted order.	All except data flow, S, constants. SA – SC valid only for WORD type	No
Q (must be the same type as IN)	An array of indexes that gives the position of the sorted elements in the original memory block	All except S - SC and constants	No

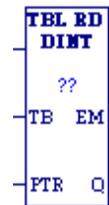
#### Example

New part numbers (%I00017 - %I00032) are pushed onto a parts array PLIST every time %Q00014 is ON. When the array is filled, it is sorted and the output %Q00025 is turned on. The array PPOSN then contains the original position that the now-sorted elements held before the sort was done on PLIST.

If PLIST were an array of five elements and contained the values 25, 67, 12, 35, 14 before the sort, then after the sort it would contain the values 12, 14, 25, 35, 67. PPOSN would contain the values 3, 5, 1, 4, 2.



### 4.9.9 Table Read



The Table Read (TBL\_RD) function sequentially reads values in a table. When the pointer reaches the end of the table, it wraps around to the beginning of the table. (TBL\_RD is like FIFO\_RD with a wrap-around.)

Mnemonics:  
 TBL\_RD\_DINT  
 TBL\_RD\_DWORD  
 TBL\_RD\_INT  
 TBL\_RD\_UINT  
 TBL\_RD\_WORD

When TBL\_RD receives power flow:

1. TBL\_RD increments the pointer by one.
2. TBL\_RD copies data indicated by the pointer to output parameter Q. Additional program logic must then be used to capture the data from the output reference.
3. Steps 1 and 2 are repeated each time the instruction is executed, until the end of the table is reached (PTR=length specified in Length). When the end of the table is reached, the pointer wraps around to the beginning of the table.

When TBL\_RD receives power flow, the pointer (PTR) increments by one. If this new pointer location is the last item in the table, the output EM is set ON. The next time TBL\_RD executes, PTR is automatically set back to 1. After PTR is incremented, the content at the new pointer location is copied to output Q.

TBL\_RD always passes power to the right when it receives power.

**Note:** The TBL\_RD and TBL\_WRT functions can operate on the same or different tables. By specifying a different reference for the pointer, these functions can access the same data table at different locations or at different rates.

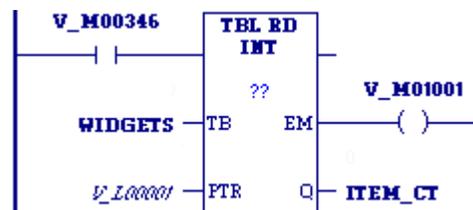
#### Operands

**Note:** For each mnemonic, use the corresponding data type for the TB and Q operands. For example, TBL\_RD\_DINT requires TB and Q to be DINT variables.

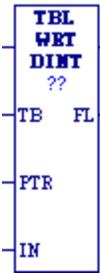
Parameter	Description	Allowed Operands	Optional
Length	$1 \leq \text{Length} \leq 32,767$	Constants	No
TB (must be the same type as Q)	The elements in the table	All except constants	No
PTR	Pointer. Index of the next element.	All except data flow, S - SC, constants	No
EM	Energized when the last element of the table is read	Power flow	No
Q (must be the same type as TB)	The element read from the table	All except constants, S, SA, SB, SC allowed only for WORD, DWORD	No

#### Table Read Example

WIDGETS is a table with 20 integer elements. When the enabling input %M00346 is ON, the pointer increments and the contents of the next element of the table are copied into ITEM\_CT. %L00001 functions as the pointer into the data table. %M01001 is used to signal when all items of the data table have been accessed.



### 4.9.10 Table Write



The Table Write (TBL\_WRT) function sequentially updates values in a table that never becomes full. When the pointer (PTR) reaches the end of the table, it automatically returns to the beginning of the table.

Mnemonics:  
 TBL\_WRT\_DINT  
 TBL\_WRT\_DWORD  
 TBL\_WRT\_INT  
 TBL\_WRT\_UINT  
 TBL\_WRT\_WORD

1. TBL\_WRT increments the pointer by one.
2. TBL\_WRT copies data from input parameter IN to the position in the table indicated by the pointer. (It writes over any value currently at that location.) Additional program logic must then be used to place the data in the input reference.
3. Steps 1 and 2 are repeated each time the instruction is executed, until the table is full (PTR=LEN). When the table is full, the pointer wraps around to the beginning of the table.

**Note:** The TBL\_WRT and TBL\_RD functions can operate on the same or different tables. By specifying a different reference for the pointer, these functions can access the same data table at different locations or at different rates.

When TBL\_WRT receives power flow, the pointer (PTR) increments by 1. If this new pointer location is the last item in the table, the output FL is set to ON. The next time TBL\_WRT executes, PTR is automatically set back to 1. After incrementing PTR, TBL\_WRT writes the content of the input reference to the current pointer location, overwriting data already stored there.

TBL\_WRT always passes power to the right when it receives power.

**Note:** TBL\_WRT is like FIFO\_WRT with a wrap-around.

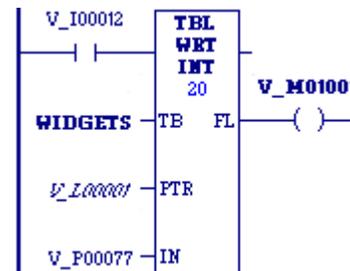
#### Operands

**Note:** For each mnemonic, use the corresponding data type for the TB and IN operands. For example, TBL\_WRT\_DINT requires TB and IN to be DINT variables.

Parameter	Description	Allowed Operands	Optional
Length	$1 \leq \text{Length} \leq 32,767$ .	Constants	No
TB (must be the same data type as IN)	The elements in the table	All except S, constants, data flow. SA - SC allowed only for WORD, DWORD	No
PTR	Pointer. Index of the next element.	All except constants, data flow, %S - %SC	No
IN (must be the same data type as TB)	The element to write to the table	All. %S - %SC allowed only for WORD, DWORD	No
FL	Energized when IN is written to the last element of the table	Power flow	No

#### Table Write Example

WIDGETS is a table with 20 integer elements. When the enabling input %I00012 is ON, the pointer increments and the contents of %P00077 are written into the table at the pointer location. %L00001 functions as the pointer into the data table.



## 4.10 Math Functions

Your program may need to include logic to convert data to a different data type before using a Math or Numerical function. The description of each function includes information about appropriate data types. Refer to the *Conversion Functions* section to understand how to convert one data type to a different data type.

Function	Mnemonics	Description
Absolute Value	ABS_DINT, ABS_INT, ABS_REAL, ABS_LREAL	Finds the absolute value of a double-precision integer (DINT), signed single-precision integer (INT), or floating-point (REAL or LREAL) value. The mnemonic specifies the value's data type.
Add	ADD_DINT, ADD_INT, ADD_REAL, ADD_LREAL, ADD_UINT	Addition. Adds two numbers.
Divide <sup>4</sup>	DIV_DINT, DIV_INT, DIV_MIXED, DIV_REAL, DIV_LREAL, DIV_UINT	Division. Divides one number by another and outputs the quotient. <b>Note:</b> Take care to avoid <i>Overflow</i> conditions when performing divisions.
Modulus	MOD_DINT, MOD_INT, MOD_UINT	Modulo Division. Divides one number by another and outputs the remainder.
Multiply <sup>4</sup>	MUL_DINT, MUL_INT, MUL_MIXED, MUL_REAL, MUL_LREAL, MUL_UINT	Multiplication. Multiplies two numbers. <b>Note:</b> Take care to avoid <i>Overflow</i> conditions when performing multiplications.
Scale	SCALE	Scales an input parameter and places the result in an output location.
Subtract	SUB_DINT, SUB_INT, SUB_REAL, SUB_LREAL, SUB_UINT	Subtraction. Subtracts one number from another.

<sup>4</sup> To avoid *Overflows* when multiplying or dividing 16-bit numbers, use the *Conversion Functions* to convert the numbers to a 32-bit data type.

### 4.10.1 Overflow

When an operation results in overflow, there is no power flow.

If an operation on signed operands (INT, DINT, REAL) results in overflow, the output reference is set to its largest possible value for the data type. For signed numbers, the sign is set to show the direction of the overflow. If signed or double precision integers are used, the sign of the result for DIV and MUL functions depends on the signs of I1 and I2.

Maximum Values	MAXINT16	Maximum signed 16-bit	7FFF hex	32,767
	MAXUINT16	Maximum unsigned 16-bit	FFFF hex	65,535
	MAXINT32	Maximum signed 32-bit	7FFFFFFF hex	2,147,483,647
Minimum Values	MININT16	Minimum signed 16-bit	8000 hex	-32,768
	MININT32	Minimum signed 32-bit	80000000 hex	-2,147,483,648

If an operation on unsigned operands (UINT) results in overflow or underflow, the output value wraps around. For example the ADD\_UINT operation, 65535+16, yields a result of 15.

### 4.10.2 Absolute Value



When the function receives power flow, it places the absolute value of input IN into output Q.

Mnemonics:  
 ABS\_DINT  
 ABS\_INT  
 ABS\_REAL  
 ABS\_LREAL

The function outputs power flow, unless one of the following conditions occurs:

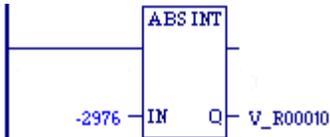
- For INT type, IN is  $-32,768$ .
- For DINT type, IN is  $-2,147,483,648$ .
- For REAL or LREAL type, IN is NaN (Not a Number).

#### Operands

Parameter	Description	Allowed Operands	Optional
IN (must be same type as Q)	The value to process.	All except S, SA, SB, SC	No
Q (must be same type as IN)	The absolute value of IN.	All except S, SA, SB, SC and constant	No

#### Example

The absolute value of  $-2,976$ , which is  $2,976$ , is placed in %R00010:



### 4.10.3 Add



When the ADD function receives power flow, it adds the two operands IN1 and IN2 of the same data type and stores the sum in the output variable assigned to Q, also of the same data type.

Mnemonics:  
 ADD\_DINT  
 ADD\_INT  
 ADD\_REAL  
 ADD\_LREAL  
 ADD\_UINT

The power flow output is energized when ADD is performed, unless an invalid operation or *Overflow* occurs. (For more information, refer to the section on *Overflow*.)

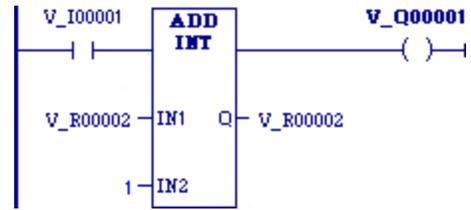
Mnemonic	Operation	Displays as
ADD_INT	$Q(16\text{-bit}) = IN1(16\text{-bit}) + IN2(16\text{-bit})$	base 10 number with sign, up to 5 digits long
ADD_DINT	$Q(32\text{-bit}) = IN1(32\text{-bit}) + IN2(32\text{-bit})$	base 10 number with sign, up to 10 digits long
ADD_REAL	$Q(32\text{-bit}) = IN1(32\text{-bit}) + IN2(32\text{-bit})$	base 10 number, sign and decimals, up to 8 digits long (excluding the decimals)
ADD_LREAL	$Q(64\text{-bit}) = IN1(64\text{-bit}) + IN2(64\text{-bit})$	base 10 number, sign and decimals, up to 17 digits long (excluding the decimals)
ADD_UINT	$Q(16\text{-bit}) = IN1(16\text{-bit}) + IN2(16\text{-bit})$	base 10 number, unsigned, up to 5 digits long

#### Operands of the ADD Function

Operand	Description	Allowed Operands	Optional
IN1	The value to the left of the plus sign (+) in the equation $IN1+IN2=Q$ .	All except S, SA, SB, SC	No
IN2	The value to the right of the plus sign (+) in the equation $IN1+IN2=Q$ .	All except S, SA, SB, SC	No
Q	The result of $IN1+IN2$ . If an ADD of signed operands results in <i>Overflow</i> , Q is set to the largest possible value and there is no power flow. If an ADD_UINT operation results in <i>Overflow</i> , Q wraps around.	All except S, SA, SB, SC and constant.	No

**Example1 for ADD**

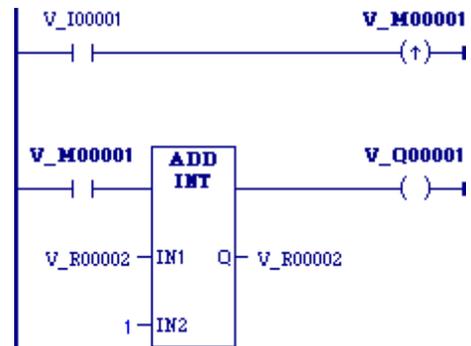
The first example is a failed attempt to create a counter circuit that would count the number of times switch %I00001 closes. The running total is stored in register %R00002. The intent of this design is that when %I0001 closes, the ADD instruction should add one to the value in %R00002 and place the new value right back into %R00002. The problem with this design is that the ADD instruction executes once every PLC scan while %I0001 is closed. For example, if %I0001 stays closed for five scans, the output increments five times, even though %I00001 only closed once during that period.



**Example2 for ADD**

To correct the above problem, the enable input to the ADD instruction should come from a transition (one-shot) coil, as shown below. In the improved circuit, the %I0001 input switch controls a transition coil, %M0001, whose contact turns on the enable input of the ADD function for only one scan each time contact %I00001 closes. In order for the %M00001 contact to close again, contact %I0001 has to open and close again.

**Note:** If IN1 and/or IN2 is NaN (Not a Number), ADD\_REAL passes no power flow.



### 4.10.4 Divide



When the DIV function receives power flow, it divides the operand IN1 by the operand IN2 of the same data type as IN1 and stores the quotient in the output variable assigned to Q, also of the same data type as IN1 and IN2.

The power flow output is energized when DIV is performed, unless an invalid operation or *Overflow* occurs. (For more information, refer to the section on *Overflow*.)

Mnemonics:  
 DIV\_DINT  
 DIV\_INT  
 DIV\_MIXED  
 DIV\_REAL  
 DIV\_LREAL  
 DIV\_UINT

Notes:

- DIV rounds down; it does not round to the closest integer. For example, 24 DIV 5 = 4.
- DIV\_MIXED uses mixed data types.
- Be careful to avoid overflows.

The following REAL and LREAL operations are invalid for DIV:

- Any number divided by 0. This operation yields a result of 65535.
- ∞ divided by ∞
- I1 and/or I2 is NaN (Not a Number)

Mnemonic	Operation	Displays as
DIV_UINT	$Q(16\text{-bit}) = IN1(16\text{-bit}) / IN2(16\text{-bit})$	base 10 number, unsigned, up to 5 digits long
DIV_INT	$Q(16\text{-bit}) = IN1(16\text{-bit}) / IN2(16\text{-bit})$	base 10 number with sign, up to 5 digits long
DIV_DINT	$Q(32\text{-bit}) = IN1(32\text{-bit}) / IN2(32\text{-bit})$	base 10 number with sign, up to 10 digits long
DIV_MIXED	$Q(16\text{-bit}) = IN1(32\text{-bit}) / IN2(16\text{-bit})$	base 10 number with sign, up to 5 digits long
DIV_REAL	$Q(32\text{-bit}) = IN1(32\text{-bit}) / IN2(32\text{-bit})$	base 10 number, sign and decimals, up to 8 digits long (excluding the decimals)
DIV_LREAL	$Q(64\text{-bit}) = IN1(64\text{-bit}) / IN2(64\text{-bit})$	base 10 number, sign and decimals, up to 17 digits long (excluding the decimals)

#### Operands for the DIV Function

Parameter	Description	Allowed Operands	Optional
IN1	Dividend: the value to be divided; shown to the left of <i>DIV</i> in the equation $IN1 \text{ DIV } IN2=Q$ .	All except S, SA, SB, SC	No
IN2	Divisor: the value to divide into IN1; shown to the right of <i>DIV</i> in the equation $IN1 \text{ DIV } IN2=Q$ .	All except S, SA, SB, SC	No
Q	The quotient of $IN1/IN2$ . If a DIV operation on signed operands results in <i>Overflow</i> , Q is set to the largest possible value and there is no power flow. If a DIV_UINT operation results in <i>Overflow</i> , Q wraps around.	All except S, SA, SB, SC and constant	No

### DIV\_MIXED Operands

Parameter	Description	Allowed Operands	Optional
IN1	Dividend: the value to be divided; shown to the left of <i>DIV</i> in the equation $IN1 \text{ DIV } IN2=Q$ .	All except S, SA, SB, SC	No
IN2	Divisor: the value to divide into IN1; shown to the right of <i>DIV</i> in the equation $IN1 \text{ DIV } IN2=Q$ .	All except S, SA, SB, SC	No
Q	The quotient of $IN1/IN2$ . If an <i>Overflow</i> occurs, the result is the largest value with the proper sign and no power flow.	All except S, SA, SB, SC and constant	No

### DIV\_MIXED Example

DIV\_DINT can be used in conjunction with a MUL\_DINT function to scale a  $\pm 10$  volt input to  $\pm 25,000$  engineering units. Refer to *Example – Scaling Analog Input Values*.

### 4.10.5 Modulus



When the Modulo Division (MOD) function receives power flow, it divides input IN1 by input IN2 and outputs the remainder of the division to Q.

Mnemonics:  
 MOD\_DINT  
 MOD\_INT  
 MOD\_UINT

All three operands must be of the same data type. The sign of the result is always the same as the sign of input parameter IN1. Output Q is calculated using the formula:

$$Q = IN1 - ((IN1 \text{ DIV } IN2) \times IN2)$$

where DIV produces an integer number.

The power flow output is always ON when the function receives power flow, unless there is an attempt to divide by zero. In that case, the power flow output is set to OFF.

#### Operands for Modulus Function

Parameter	Description	Allowed Operands	Optional
IN1	Dividend: the value to be divided to obtain the remainder; shown to the left of <i>MOD</i> in the equation $IN1 \text{ MOD } IN2=Q$ .	All except S, SA, SB, SC	No
IN2	Divisor: the value to divide into IN1; shown to the right of <i>MOD</i> in the equation $IN1 \text{ MOD } IN2=Q$ .	All except S, SA, SB, SC	No
Q	The remainder of $IN1/IN2$ .	All except S, SA, SB, SC and constant	No

### 4.10.6 Multiply



When the MUL function receives power flow, it multiplies the two operands IN1 and IN2 of the same data type and stores the result in the output variable assigned to Q, also of the same data type.

The power flow output is energized when the function is performed, unless an invalid operation or *Overflow* occurs. (For more information, refer to the section on *Overflow*.)

Mnemonics:  
 MUL\_DINT  
 MUL\_INT  
 MUL\_MIXED  
 MUL\_REAL  
 MUL\_LREAL  
 MUL\_UINT

**Note:** MUL\_MIXED uses mixed data types. Be careful to avoid overflows.

The following REAL and LREAL operations are invalid for MUL:

- $0 \times \infty$
- I1 and/or I2 is NaN (Not a Number).

Mnemonic	Operation	Displays as
MUL_INT	$Q(16\text{-bit}) = IN1(16\text{-bit}) \times IN2(16\text{-bit})$	base 10 number with sign, up to 5 digits long
MUL_DINT	$Q(32\text{-bit}) = IN1(32\text{-bit}) \times IN2(32\text{-bit})$	base 10 number with sign, up to 10 digits long
MUL_REAL	$Q(32\text{-bit}) = IN1(32\text{-bit}) \times IN2(32\text{-bit})$	base 10 number, sign and decimals, up to 8 digits long (excluding the decimals)
MUL_LREAL	$Q(64\text{-bit}) = IN1(64\text{-bit}) \times IN2(64\text{-bit})$	base 10 number, sign and decimals, up to 17 digits long (excluding the decimals)
MUL_UINT	$Q(16\text{-bit}) = IN1(16\text{-bit}) \times IN2(16\text{-bit})$	base 10 number, unsigned, up to 5 digits long
MUL_MIXED	$Q(32\text{-bit}) = IN1(16\text{-bit}) \times IN2(16\text{-bit})$	base 10 number with sign, up to 10 digits long

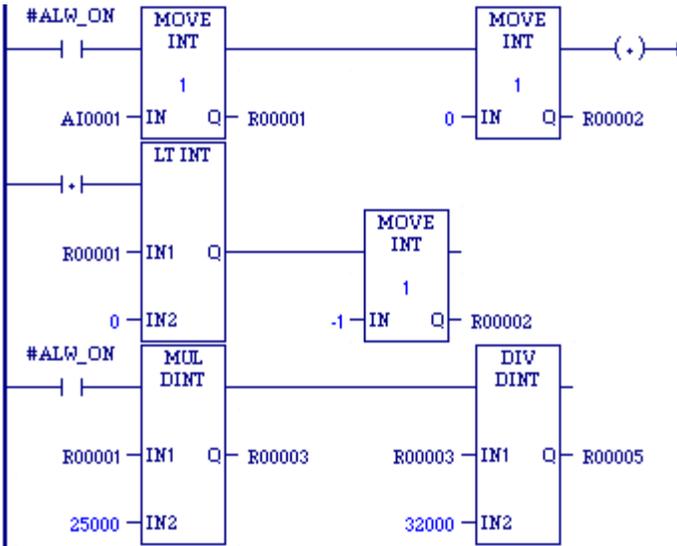
#### Operands for Multiply

Parameter	Description	Allowed Operands	Optional
IN1	The first value to multiply; the value to the left of the multiply sign (×) in the equation $IN1 \times IN2=Q$ .	All except S, SA, SB, SC	No
IN2	The second value to multiply; the value to the right of the multiply sign (×) in the equation $IN1 \times IN2=Q$ .	All except S, SA, SB, SC	No
Q	The result of $IN1 \times IN2$ . If a MUL operation on signed operands results in <i>Overflow</i> , Q is set to the largest possible value and there is no power flow. If a MUL_UINT operation results in <i>Overflow</i> , Q wraps around.	All except S, SA, SB, SC and constant	No

**Example – Scaling Analog Input Values**

A common application is to scale analog input values with a MUL operation followed by a DIV and possibly an ADD operation. A 0 to  $\pm 10$  volt analog input will place values of 0 to  $\pm 32,000$  in its corresponding %AI input register. Multiplying this input register using an MUL\_INT function will result in an *Overflow* since an INT type instruction has an input and output range of 32,767 to  $-32,768$ . Using the %AI value as in input to a MUL\_DINT also does not work as the 32-bit IN1 will combine 2 analog inputs at the same time. To solve this problem, you can move the analog input to the low word of a double register, then test the sign and set the second register to 0 if the sign tests positive or  $-1$  if negative. Then use the double register just created with a MUL\_DINT which gives a 32-bit result, and which can be used with a following DIV\_DINT function.

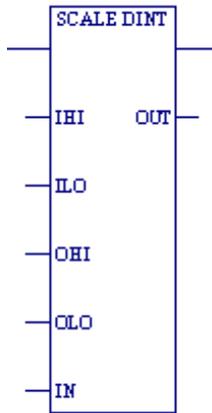
For example, the following logic could be used to scale a  $\pm 10$  volt input %AI1 to  $\pm 25000$  engineering units in %R5.



An alternate, but less accurate, way of programming this circuit using INT values involves placing the DIV\_DINT instruction first, followed by the MUL\_DINT instruction. The value of IN2 for the DIV instruction would be 32, and the value of IN2 for the MUL would be 25. This maintains the scaling proportion of the above circuit and keeps the values within the working range of the INT type instructions. However, the DIV instruction inherently discards any remainder value, so when the DIV output is multiplied by the MUL instruction, the error introduced by a discarded remainder is multiplied. The percent of error is non-linear over the full range of input values and is greater at lower input values.

By contrast, in the example above, the results are more accurate because the DIV operation is performed last, so the discarded remainder is not multiplied. If even greater precision is required, substitute REAL type math instructions in this example so that the remainder is not discarded.

### 4.10.7 Scale



When the SCALE function receives power flow, it scales the input operand IN and places the result in the output variable assigned to output operand OUT. The power flow output is energized when SCALE is performed without *Overflow*.

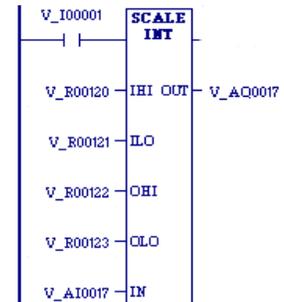
Mnemonics:  
SCALE\_DINT  
SCALE\_INT  
SCALE\_DINT  
SCALE\_UINT

### Operands

Parameter	Description	Allowed Operands	Optional
IHI	(Inputs High) Maximum input value (module-related). The upper limit of the unscaled data. IHI is used with ILO, OHI and OLO to calculate the scaling factor applied to the input value IN.	All except S, SA, SB, SC	No
ILO	(Inputs Low) Minimum input value (module-related). The lower limit of the unscaled data. Must be the same data type as IHI.	All except S, SA, SB, SC	No
OHI	(Outputs High) Maximum output value. The upper limit of the scaled data. Must be the same data type as IHI. When the IN input is at the IHI value, the OUT value is the same as the OHI value.	All except S, SA, SB, SC	No
OLO	(Outputs Low) Minimum output value. The lower limit of the scaled data. Must be the same data type as IHI. When the IN input is at the ILO value, the OUT value is the same as the OLO value.	All except S, SA, SB, SC	No
IN	(INput value) The value to be scaled. Must be the same data type as IHI.	All except S, SA, SB, SC	No
OUT	(OUTput value) The scaled equivalent of the input value. Must be the same data type as IHI.	All except S, SA, SB, SC	No

### Example

In the example at right, the registers %R0120 through %R0123 are used to store the high and low scaling values. The input value to be scaled is analog input %AI0017. The scaled output data is used to control analog output %AQ0017. The scaling is performed whenever %I0001 is ON.



### 4.10.8 Subtract



When the SUB function receives power flow, it subtracts the operand IN2 from the operand IN1 of the same data type as IN2 and stores the result in the output variable assigned to Q, also of the same data type.

Mnemonics:  
 SUB\_DINT  
 SUB\_INT  
 SUB\_REAL  
 SUB\_LREAL  
 SUB\_UINT

The power flow output is energized when SUB is performed, unless an invalid operation or *Overflow* occurs. (For more information, refer to the section on *Overflow*.)

If a SUB\_UINT operation results in a negative number, Q wraps around, yielding a result that is the highest possible value (65535) minus the absolute value of the difference -1.

The following REAL and LREAL operations are invalid for SUB:

- $(\pm \infty) - (\pm \infty)$
- I1 and/or I2 is NaN (Not a Number)

Mnemonic	Operation	Displays as
SUB_INT	$Q(16\text{-bit}) = IN1(16\text{-bit}) - IN2(16\text{-bit})$	base 10 number with sign, up to 5 digits long
SUB_DINT	$Q(32\text{-bit}) = IN1(32\text{-bit}) - IN2(32\text{-bit})$	base 10 number with sign, up to 10 digits long
SUB_REAL	$Q(32\text{-bit}) = IN1(32\text{-bit}) - IN2(32\text{-bit})$	base 10 number, sign and decimals, up to 8 digits long (excluding the decimals)
SUB_LREAL	$Q(64\text{-bit}) = IN1(64\text{-bit}) - IN2(64\text{-bit})$	base 10 number, sign and decimals, up to 17 digits long (excluding the decimals)
SUB_UINT	$Q(16\text{-bit}) = IN1(16\text{-bit}) - IN2(16\text{-bit})$	base 10 number, unsigned, up to 5 digits long

### Operands for Subtract

Parameter	Description	Allowed Operands	Optional
IN1	The value to subtract from; the value to the left of the minus sign (-) in the equation $IN1-IN2=Q$ .	All except S, SA, SB, SC	No
IN2	The value to subtract from IN1; the value to the right of the minus sign (-) in the equation $IN1-IN2=Q$ .	All except S, SA, SB, SC	No
Q	The result of $IN1-IN2$ . If a SUB operation on signed operands results in underflow, Q is set to the smallest possible value and there is no power flow.  If a SUB_UINT operation results in <i>Overflow</i> , Q wraps around. For example, The SUB_UINT operation $600 - 601 = -1$ sets Q to 65535 The SUB_UINT operation $600 - 602 = -2$ sets Q to 65534	All except S, SA, SB, SC and constant	No

## 4.11 Program Flow Functions

The program flow functions limit program execution or change the way the CPU executes the application program.

Function	Mnemonic	Description
Argument Present	ARG_PRES	Determines whether an input or output parameter value was present when the function block instance of the parameter was invoked. For example, a parameter can be optional (pass by value).
Call	CALL	Causes program execution to go to a specified block.
Comment	COMMENT	Places a text explanation in the program.
End Master Control Relay	ENDMCRN	Nested End Master Control Relay. Indicates that the subsequent logic is to be executed with normal power flow.
End of Logic	END	Provides an unconditional end of logic. The program executes from the first rung to the last rung or the END instruction, whichever is encountered first.
Jump	JUMPN	Nested jump. Causes program execution to jump to a specified location indicated by a LABELN. JUMPN/LABELN pairs can be nested within one another. Multiple JUMPNs can share the same LABELN.
Label	LABELN	Nested label. Specifies the target location of a JUMPN instruction.
Master Control Relay	MCRN	Nested Master Control Relay. Causes all rungs between the MCR and its subsequent ENDMCRN to be executed without power flow. Up to MCRN/ENDMCRN pairs can be nested within one another. All the MCRNs share the same ENDMCRN.
Wires	H_WIRE	Horizontally connects elements of a line of LD logic, to complete the power flow.
	V_WIRE	Vertically connects elements of a line of LD logic, to complete the power flow.

### 4.11.1 Argument Present



The ARG\_PRES function determines whether an input parameter value was present when the function block instance of the parameter was invoked. This may be necessary if the parameter is optional.

This function must be called from a function block instance or a parameterized block.

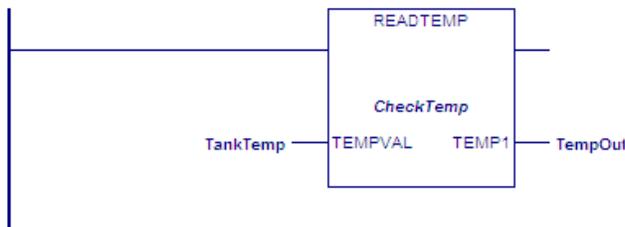
The standard output parameter ENO is false only when EN is false.

#### Operands for ARG\_PRES

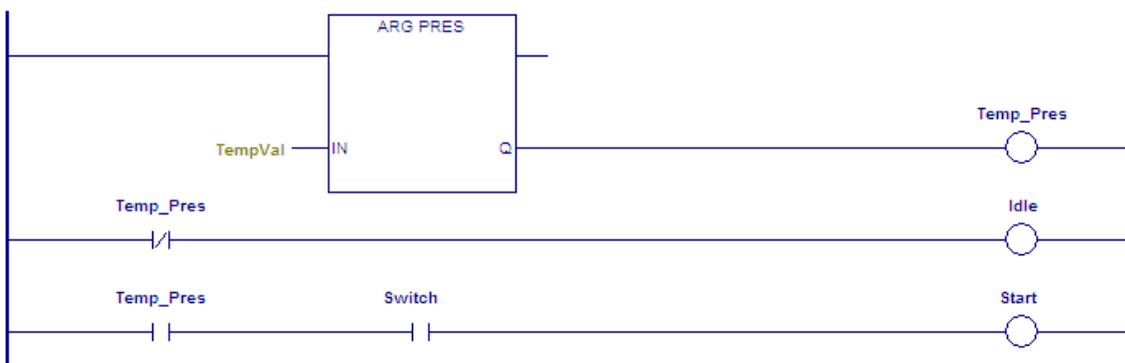
Parameter	Description	Allowed Operands	Optional
IN	Parameter name. Must be a parameter of the function block that contains the ARG_PRES instruction. Cannot be an array element or structure element. An alias to a parameter should resolve only to the parameter name.	All except flow and constants.	No
Q	True if the parameter is present, otherwise false.	Must be flow in LD. In other languages all types allowed except S, SA, SB, SC and constants.	No

#### Example for ARG\_PRES

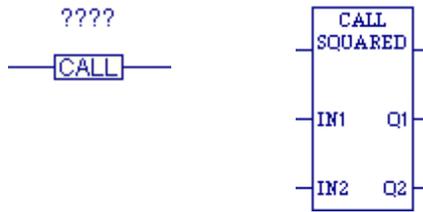
The following sample rung calls the user defined function block, ReadTemp, which has two parameters, TempVal and Temp1.



The function block ReadTemp contains the following logic, which uses an ARG\_PRES function to determine whether a value for TempVal is present. If TempVal does not have a value, Temp\_Pres is OFF and Idle is ON. If a value exists for TempVal, the ARG\_PRES function sets Temp\_Pres ON. When Temp\_Pres and Switch are both ON, Start is set ON.



### 4.11.2 Call



**Non-parameterized** **Parameterized.** May call a parameterized external block or a parameterized block. May have up to 7 input and 8 output parameters.

When the CALL function receives power flow, it causes the logic execution to go immediately to the designated program block, external C block (parameterized or not), or parameterized block and execute it. After the block's execution is complete, control returns to the point in the logic immediately following the CALL instruction.

Notes:

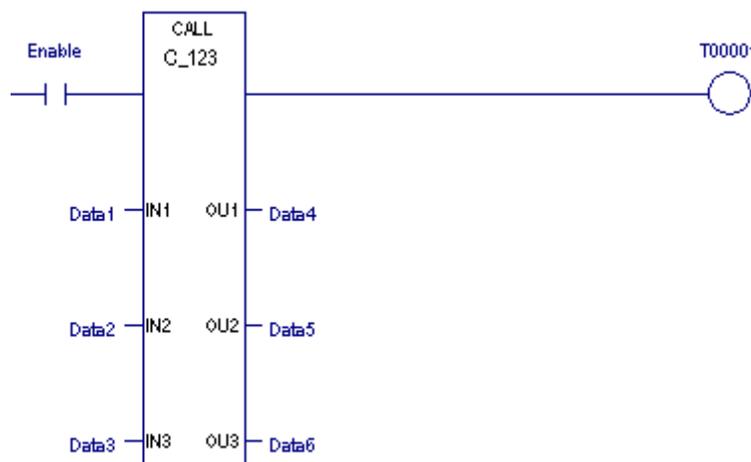
- A CALL function can be used in any program block, including the \_MAIN block, or a parameterized block. It cannot be used in an external block.
- You cannot call a \_MAIN block.
- The called block must exist in the target before making the call.
- There is no limit to the number of calls that can be made from or to a given block.
- You can set up recursive subroutines by having a block call itself. When stack size is configured to be the default (64K), the PLC guarantees a minimum of eight nested calls before an *Application Stack Overflow* fault is logged.
- Each block has a predefined parameter, Y0, which the CPU sets to 1 upon each invocation of the block. Y0 can be controlled by logic within the block and provides the output status of the block. When the Y0 parameter of a Program Block, parameterized block, or external C block returns ON, the CALL passes power to the right; when it returns OFF, the CALL does not pass power to the right.

## Operands for Call

Parameter	Description
Block Name (????)	Block name; the name of the block to transfer to. You cannot CALL the _MAIN block. A program block or a parameterized block can call itself.
(Parameterized calls only) Input parameters (0 – 7) Output parameters (1 – 8)	<p>Notes for External (C) blocks:</p> <ul style="list-style-type: none"> <li>You must define the TYPE, LENGTH, and NAME for each external C block parameter.</li> <li>The valid data type, value range, and memory area for each parameter are stated in the external block's written documentation.</li> <li>Data flow is permitted for any parameter.</li> <li>For additional information, see the section on <i>External Blocks</i> in Chapter 2.</li> </ul> <p>Notes for Parameterized Blocks:</p> <ul style="list-style-type: none"> <li>You must define the TYPE, LENGTH, and NAME for each parameter. Valid operands on the CALL instruction include variables, flow, and indirect references. Input operands can also be constants.</li> <li>If a formal parameter is an array of BOOL type and has a length evenly divisible by 16, then a variable or array residing in word-oriented memory can be passed on to the parameterized block as an operand. For example, if a parameterized block has a formal parameter Y1 of data type BIT and length 48, you can pass a WORD array of length 3 to Y1.</li> <li>The BOOL parameter Y0 is automatically defined for all parameterized blocks and can be used in the parameterized block's logic. When the parameterized block stops executing and Y0 is ON, the CALL passes power flow to the right. If Y0 is OFF, the CALL passes no power flow.</li> <li>A parameterized block is not required to have the same number of inputs and outputs.</li> <li>For additional information, refer to <i>Using Parameters with a Parameterized Block</i> in Chapter 2.</li> </ul>

## Example 1 for Call

In the example at right, if Enable is set, the C block named C\_123 is executed. C\_123 operates on the input data located at reference addresses Data1, Data2, and Data3, and produces values located at reference addresses Data4, Data5, and Data6. Logic within C\_123 controls the power flow output.



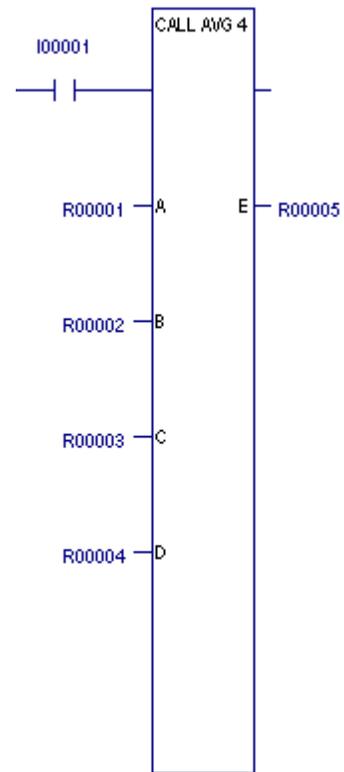
**Example 2 for Call**

Parameterized blocks are useful for building libraries of user-defined functions. For example, if you have an equation such as:

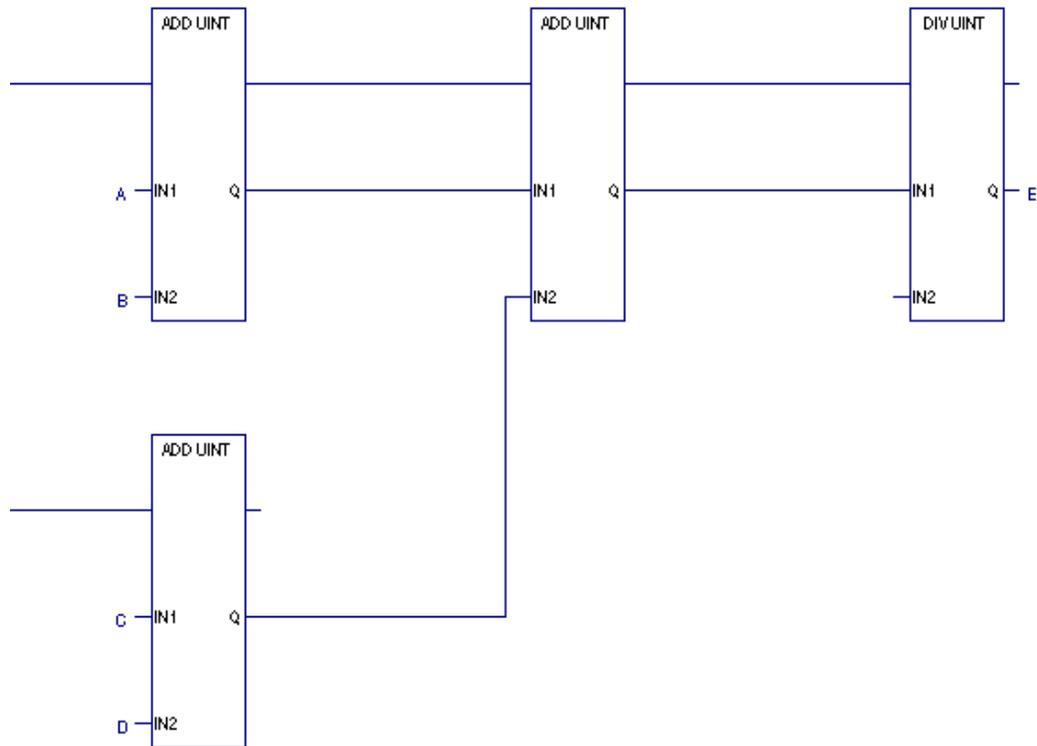
$E=(A+B+C+D)/4$ , a parameterized block named AVG\_4 could be called as shown in the example to the right.

In this example, the average of the values in R00001, R00002, R00003, and R00004 would be placed in R00005.

The logic within the parameterized block would be defined as shown below.



**Logic for AVG\_4 Parameterized Block**



### 4.11.3 Comment



The Comment function is used to enter a text explanation in the program. When you insert a Comment instruction into the LD logic, it displays ????. After you key in a comment, the first few words are displayed.



You can set the Comment mode option to Brief or Full.

Notes:

- Editing a comment makes the Programmer lose equality.
- Comment text is downloaded to the controller and retrieved upon Logic Upload.

### 4.11.4 JumpN



**Description**

Nested form of Jump instruction.

**Always associated with...**

a LABELN instruction

**Mnemonic**

JUMPN

A JUMPN instruction causes a portion of the program logic to be bypassed. Program execution continues at the LABELN specified in the same block. Power flow jumps directly from the JUMPN to the rung with the named LABELN.

When the Jump is active, any functions between the jump and the label are not executed. All coils between JUMPN and its associated LABELN are left at their previous states. This includes coils associated with timers, counters, latches, and relays.

Any JUMPN can be either a forward or a backward jump, i.e., its LABELN can be either in a further or previous rung. The LABELN must be in the same block.

**Note:** To avoid creating an endless loop with forward and backward JUMPN instructions, a backward JUMPN should contain a way to make it conditional.

A JUMPN and its associated LABELN can be placed anywhere in a program, as long as the JUMPN / LABELN range:

- does not overlap the range of a MCRN / ENDMCRN pair.
- does not overlap the range of a FOR\_LOOP / END\_FOR pair.

Nothing can be connected to the right side of a JUMPN instruction.

#### Operands

Parameter	Description	Optional
Label (????)	Label name; the name assigned to the destination LABEL(N).	No

### 4.11.5 Master Control Relay/End Master Control Relay



Description	Always associated with...	Mnemonics
Nested form of the Master Control Relay	an ENDMCRN instruction	MCRN
Nested End Master Control Relay	an MCRN instruction	ENDMCRN

#### MCRN

An MCRN instruction marks the beginning of a section of logic that will be executed with no power flow. The end of an MCRN section must be marked with an ENDMCRN having the same name as the MCRN. ENDMCRNs must follow their corresponding MCRNs in the logic.

All rungs between an active MCRN and its corresponding ENDMCRN are executed with negative power flow from the power rail. The ENDMCRN function associated with the MCRN causes normal program execution to resume, with positive power flow coming from the power rail.

With a Master Control Relay, functions within the scope of the Master Control Relay are executed *without power flow, and coils are turned off*.

Block calls within the scope of an active Master Control Relay will not execute. However, any timers in the block will continue to accumulate time.

A rung may not contain anything after an MCRN.

Unlike JUMP instructions, MCRNs can only move forward. An ENDMCRN instruction must appear after its corresponding MCRN instruction in a program.

The following controls are imposed by an MCRN:

- Timers do not increment or decrement. TMR types are reset. For an ONDTR function, the accumulator holds its value.
- Normal outputs are off; negated outputs are on.

**Note:** When an MCRN is energized, the logic it controls is scanned and contact status is displayed, but no outputs are energized. If you are not aware that an MCRN is controlling the logic being observed, this might appear to be a faulty condition.

An MCRN and its associated ENDMCRN can be placed anywhere in a program, as long as the MCRN / ENDMCRN range:

- Is completely nested within another MCRN / ENDMCRN range, up to a maximum 255 levels of nesting, or is completely outside of the range of another MCRN / ENDMCRN range.
- Is completely nested within a FOR\_LOOP / END\_FOR range or is completely outside of the range of a FOR\_LOOP / END\_FOR.

### EndMCRN

The End Master Control Relay instruction marks the end of a section of logic begun with a Master Control Relay instruction. When the MCRN associated with the ENDMCRN is active, the ENDMCRN causes program execution to resume with normal power flow. When the MCRN associated with the ENDMCRN is not active, the ENDMCRN has no effect.

ENDMCRN must be tied to the power rail; there can be no logic before it in the rung; execution cannot be conditional.

ENDMCRN has a name that identifies it and associates it with the corresponding MCRN(s). The ENDMCRN function has no outputs; there can be nothing after an ENDMCR instruction in a rung.

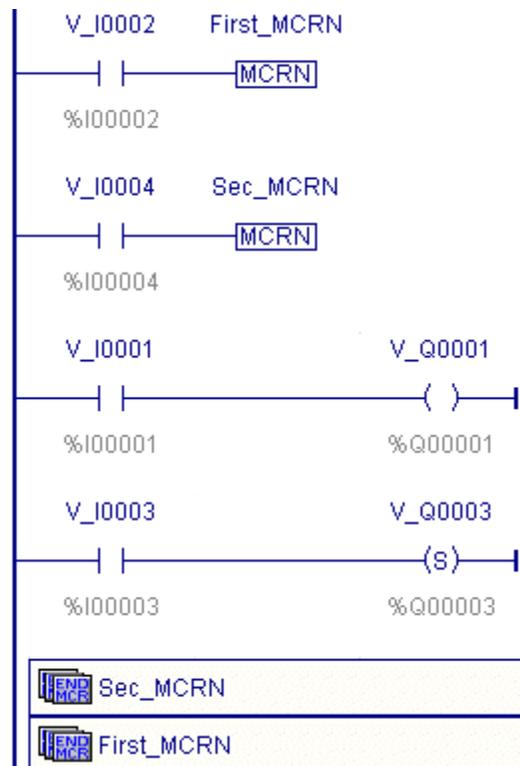
### Operands for MCRN/ENDMCRN

The Master Control Relay function has a single operand, a name that identifies the MCRN. This name is used again with an ENDMCRN instruction. The MCRN has no output.

Parameter	Description	Optional
Name (????)	The name associated with the MCRN that starts the section of logic.	No

### Example of MCRN/ENDMCRN

The example at right an MCRN named *Sec\_MCRN* nested inside the MCRN named *First\_MCRN*. Whenever the V\_I0002 contact allows power flow into the MCRN function, program execution will continue without power flow to the coils until the associated ENDMCRN is reached. If the V\_I0001 and V\_I0003 contacts are ON, the V\_Q0001 coil is turned OFF and the SET coil V\_Q0003 maintains its current state.



### 4.11.6 Wires

Horizontal and vertical wires (H\_WIRE and V\_WIRE) are used to connect elements of a line of LD logic between functions. Their purpose is to complete the flow of logic (*power*) from left to right in a line of logic.

A horizontal wire transmits the BOOLEAN ON/OFF state of the element on its immediate left to the element on its immediate right.

A vertical wire may intersect with one or more horizontal wires on each side. The state of the vertical wire is the inclusive OR of the ON states of the horizontal wires on its left side. The state of the vertical wire is copied to all of the attached horizontal wires on its right side.

**Note:** Wires can be used for data flow, but you cannot route data flow leftwards. Nor can two separate data flow lines come into the left side of the same vertical wire.

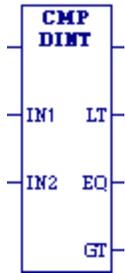


## 4.12 Relational Functions

Relational functions compare two values of the same data type or determine whether a number lies within a specified range. The original values are unaffected.

Function	Mnemonic	Description
Compare	CMP_DINT CMP_INT CMP_REAL CMP_LREAL CMP_UINT	Compares two numbers, IN1 and IN2, of the data type specified by the mnemonic. <ul style="list-style-type: none"> <li>■ If IN1 &lt; IN2, the LT output is turned ON.</li> <li>■ If IN1 = IN2, the EQ output is turned ON.</li> <li>■ If IN1 &gt; IN2, the GT output is turned ON.</li> </ul>
Equal	EQ_DATA EQ_DINT EQ_INT EQ_REAL EQ_LREAL EQ_UINT	Tests two numbers for equality
Greater or Equal	GE_DINT GE_INT GE_REAL GE_LREAL GE_UINT	Tests whether one number is greater than or equal to another
Greater Than	GT_DINT GT_INT GT_REAL GT_LREAL GT_UINT	Tests whether one number is greater than another
Less or Equal	LE_DINT LE_INT LE_REAL LE_LREAL LE_UINT	Tests whether one number is less than or equal to another
Less Than	LT_DINT LT_INT LT_REAL LT_LREAL LT_UINT	Tests whether one number is less than another
Not Equal	NE_DINT NE_INT NE_REAL NE_LREAL NE_UINT	Tests two numbers for inequality
Range	RANGE_DINT RANGE_DWORD RANGE_INT RANGE_UINT RANGE_WORD	Tests whether one number is within the range defined by two other supplied numbers

### 4.12.1 Compare



When the Compare (CMP) function receives power flow, it compares the value IN1 to the value IN2.

- If IN1 < IN2, CMP energizes the LT (Less Than) output.
- If IN1 = IN2, CMP energizes the EQ (Equal) output.
- If IN1 > IN2, CMP energizes the GT (Greater Than) output.

Mnemonics:  
 CMP\_DINT  
 CMP\_INT  
 CMP\_REAL  
 CMP\_LREAL  
 CMP\_UINT

IN1 and IN2 must be the same data type.

CMP compares data of the following types: DINT, INT, REAL, LREAL, and UINT.

**Tip:** To compare values of different data types, first use conversion functions to make the types the same.

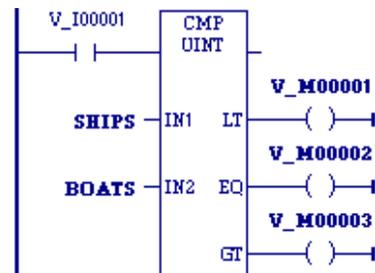
When it receives power flow, CMP always passes power flow to the right, unless IN1 and/or IN2 is NaN (Not a Number).

#### Operands

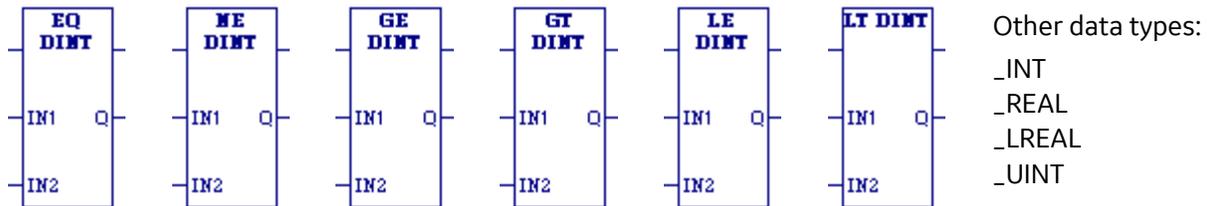
Parameter	Description	Allowed Operands	Optional
IN1	The first value to compare.	All except S, SA, SB, SC	No
IN2	The second value to compare.	All except S, SA, SB, SC	No
LT	Output LT is energized when I1 < I2.	Power flow	No
EQ	Output EQ is energized when I1 = I2.	Power flow	No
GT	Output GT is energized when I1 > I2.	Power flow	No

#### Example

When %I00001 is ON, the integer variable SHIPS is compared with the variable BOATS. Internal coils %M0001, %M0002, and %M0003 are set to the results of the compare.



### 4.12.2 Equal, Not Equal, Greater or Equal, Greater Than, Less or Equal, Less Than



When the relational function receives power flow, it compares input IN1 to input IN2. These operands must be the same data type. If inputs IN1 and IN2 are equal, the function passes power to the right, unless IN1 and/or IN2 is NaN (Not a Number). The following relational functions can be used to compare two numbers:

Function	Definition	Relational Statement
EQ	Equal	IN1=IN2
NE	Not Equal	IN1≠IN2
GE	Greater Than or Equal	IN1≥IN2
GT	Greater Than	IN1>IN2
LE	Less Than or Equal	IN1≤IN2
LT	Less Than	IN1<IN2

**Note:** If an *Overflow* occurs with a `_UINT` operation, the result wraps around – refer to the section on *Overflow*.

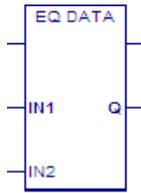
If the `_DINT` or `_INT` operations are fed the largest possible value with any sign, they cannot determine if it is an overflow value. The power flow output of the previous operation would need to be checked. If an overflow occurred on a previous `DINT`, or `INT` operation, the result was the largest possible value with the proper sign and no power flow.

**Tip:** To compare values of different data types, first use conversion functions to make the types the same. The relational functions require data to be one of the following types: `DINT`, `INT`, `REAL`, `LREAL`, or `UINT`.

#### Operands

Parameter	Description	Allowed Operands	Optional
IN1	The first value to be compared; the value on the left side of the relational statement.	All except S, SA, SB, SC	No
IN2	The second value to be compared; the value on the right side of the relational statement. IN2 must be the same data type as IN1.	All except S, SA, SB, SC	No
Q	The power flow. If the relational statement is true, Q is energized, unless IN1 or IN2 is NaN.	Power flow	No

### 4.12.3 EQ\_DATA



The EQ\_DATA function compares two input variables, IN1 and IN2 of the same data type. If IN1 and IN2 are equal, output Q is energized. If they are not equal, Q is cleared.

Mnemonic:  
EQ\_DATA

#### Operands

Parameter	Description	Allowed Operands	Optional
IN1	The first value to be compared; the value on the left side of the relational statement.	PACMotion ENUM variable or structure variable.  For details, refer to <i>Data Types and Structures</i> in the <i>PACMotion Multi-Axis Motion Controller User's Manual</i> , GFK-2448.	No
IN2	The second value to be compared; the value on the right side of the relational statement. IN2 must be the same data type as IN1.	PACMotion ENUM variable or structure variable.	No
Q	If IN1 or IN2 is true, Q is energized.	Power flow	No

### 4.12.4 Range



When the Range function is enabled, it compares the value of input IN against the range delimited by operands L1 and L2. Either L1 or L2 can be the high or low limit. When  $L1 \leq IN \leq L2$  or  $L2 \leq IN \leq L1$ , output parameter Q is set ON (1). Otherwise, Q is set OFF (0).

If the operation is successful, it passes power flow to the right.

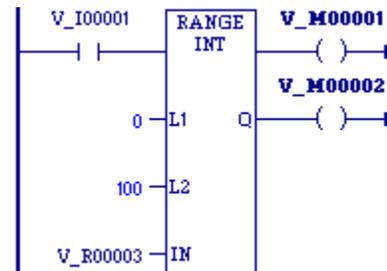
- Mnemonics:  
 RANGE\_DINT  
 RANGE\_DWORD  
 RANGE\_INT  
 RANGE\_UINT  
 RANGE\_WORD

#### Operands

Parameter	Description	Allowed Operands	Optional
IN	The value to compare against the range delimited by L1 and L2. Must be the same data type as L1 and L2.	All except S, SA, SB, SC	No
L1	The start point of the range. May be the upper limit or the lower limit. Must be the same data type as IN and L2.	All except S, SA, SB, SC	No
L2	The end point of the range. May be the lower or upper limit. Must be the same data type as IN and L1.	All except S, SA, SB, SC	No
Q	If $L1 \leq IN \leq L2$ or $L2 \leq IN \leq L1$ , Q is energized; otherwise, Q is off.	Power flow	No

#### Example

When RANGE\_INT receives power flow from the normally open contact %I0001, it determines whether the value in %R00003 is within the range 0 to 100 inclusively. Output coil %M00002 is ON only if  $0 \leq \%AI0050 \leq 100$ .



## 4.13 Timers

This section describes the PACSystems timed contacts and timer function blocks that are implemented in the LD language.

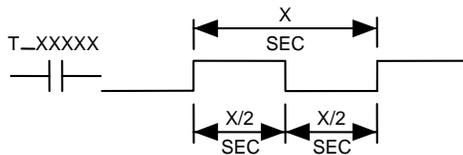
### 4.13.1 Timed Contacts

The PACSystems has four timed contacts that can be used to provide regular pulses of power flow to other program functions. Timed contacts cycle on and off, in square-wave form, every 0.01 second, 0.1 second, 1.0 second, and 1 minute. Timed contacts can be read by an external communications device to monitor the state of the CPU and the communications link. Timed contacts are also often used to blink pilot lights and LEDs.

The timed contacts are referenced as T\_10MS (0.01 second), T\_100MS (0.1 second), T\_SEC (1.0 second), and T\_MIN (1 minute). These contacts represent specific locations in %S memory:

```
#T_10MS   0.01 second timed contact  %S0003
#T_100MS  0.1 second timed contact   %S0004
#T_SEC    1.0 second timed contact    %S0005
#T_MIN    1.0 minute timed contact    %S0006
```

These contacts provide a pulse having an equal on and off time duration. The following timing diagram illustrates the on/off time duration of these contacts.



#### Caution

**Do not use timed contacts for applications requiring accurate measurement of elapsed time. Timers, time-based subroutines, and PID blocks are preferred for these types of applications.**

**The CPU updates the timed contact references based on a free-running timer that has no relationship to the start of the CPU sweep. If the sweep time remains in phase with the timed contact clock, the contact will always appear to be in the same state. For example, if the CPU is in constant sweep mode with a sweep time setting of 100ms, the T\_10MS and T\_100MS bits will never toggle.**



### 4.13.2 Timer Function Blocks

Function	Function Block Type	Mnemonic	Description
Off Delay Timer	Built-in (instance data is WORD array) See <i>Built-In Timer Function Blocks</i> below.	OFDT_HUNDS OFDT_SEC OFDT_TENTHS OFDT_THOUS	The Current Value (CV) of the timer resets to zero when power flow input is on. CV increments while power flow is off. When CV=PV (Preset Value), power flow is no longer passed to the right until power flow input is on again.
On Delay Stopwatch Timer		ONDTR_HUNDS ONDTR_SEC ONDTR_TENTHS ONDTR_THOUS	Retentive on delay timer. Increments while it receives power flow and holds its value when power flow stops.
On Delay Timer		TMR_HUNDS TMR_SEC TMR_TENTHS TMR_THOUS	Simple on delay timer. Increments while it receives power flow and resets to zero when power flow stops.
Timer Off Delay	Standard (instance data is a structure variable) See <i>Standard Timer Function Blocks</i> .	TOF	When the input IN transitions from ON to OFF, the timer starts timing until a specified period of time has elapsed, then sets the output Q to OFF.
Timer On Delay		TON	When the input IN transitions from OFF to ON, the timer starts timing until a specified period of time has elapsed, then sets the output Q to ON.
Timer Pulse		TP	When the input IN transitions from OFF to ON, the timer sets the output Q to ON for a specified time interval.

#### Built-In Timer Function Blocks

**Note:** Special care must be taken when programming timers in program blocks that are not called every sweep, and in parameterized blocks and UDFBs. For details, refer to:

- *Using OFDT, ONDTR and TMR in Program Blocks not Called Every Sweep*
- *Timers that are Skipped by the Jump Instruction*
- *Using OFDT, ONDTR and TMR in Parameterized Blocks, and*
- *Using OFDT, ONDTR and TMR in UDFBs.*

#### Data Required for Built-in Timer Function Blocks

The data associated with these functions is retentive through power cycles. Each timer uses a three-word array of %R, %W, %P, %L or symbolic memory to store the following information:

Current value (CV) Word 1  
Preset value (PV) Word 2  
Control word Word 3



**Warning**

**Do not use two consecutive words (registers) as the starting addresses of two timers. Logic Developer - PLC does not check or warn you if register blocks overlap. Timers will not work if you place the current value of a second timer on top of the preset value for the previous timer.**

**Word 1: Current value (CV)**



**Warning**

**The first word (CV) can be read but should not be written to, or the function may not work properly.**

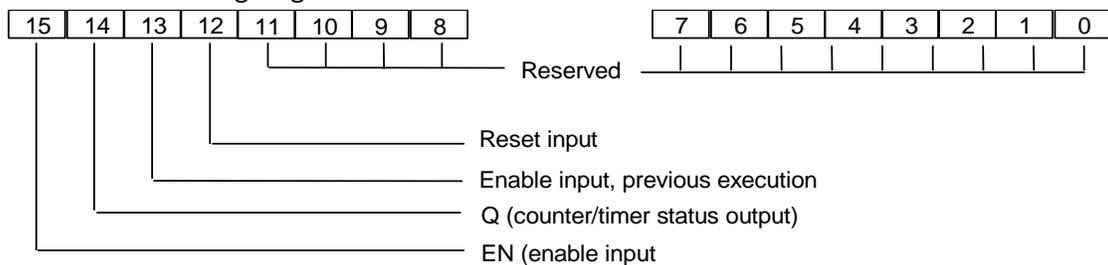
**Word 2: Preset value (PV)**

When the Preset Value (PV) operand is a variable, it is normally set to a different location than word 2 in the timer's or counter's three-word array.

- If you use a different address and you change word 2 directly, your change will have no effect, as PV will overwrite word 2.
- If you use the same address for the PV operand and word 2, you can change the Preset Value in word 2 while the timer or counter is running and the change will be effective.

**Word 3: Control word**

The control word stores the state of the Boolean inputs and outputs of its associated timer or counter, as shown in the following diagram:



**Warning**

**The third word (Control) can be read but should not be written to; otherwise, the function will not work.**

**Note:** Bits 0 through 13 are used for timer accuracy.

### Using OFDT, ONDTR and TMR in Program Blocks not Called Every Sweep

Care should be taken when timers (ONDTR, TMR, and OFDTR) are used in program blocks that are **not** called every sweep. The timers accumulate time across calls to the sub-block unless they are reset. This means that they function like timers operating in a program with a much slower sweep than the timers in the main program block. For program blocks that are inactive for large periods of time, the timers should be programmed in such a manner as to account for this catch up feature.

### Timers that are Skipped by the Jump Instruction

You should not program a Jump around an instance of OFDT, ONDTR or TMR. Timers that are skipped will **not** catch up and will therefore not accumulate time in the same manner as if they were executed every sweep.

**Note:** Timer function blocks do not accumulate time if used in a block that is executed as a result of an interrupt.

### Using OFDT, ONDTR and TMR in Parameterized Blocks

Special care must be taken when programming timers in PACSystems parameterized blocks. Timers in parameterized blocks can be programmed to track true real-time as long as the guidelines and rules below are followed. If the guidelines and rules described here are not followed, the operation of the timer functions in parameterized blocks is undefined.

**Note:** These rules are not enforced by the programming software. It is your responsibility to ensure these rules are followed.

The best use of a timer function is to invoke it with a particular reference address exactly one time each scan. With parameterized blocks, it is important to use the appropriate reference memory with the timer function and to call the parameterized block an appropriate number of times.

### Finding the Source Block

The source block is either the `_MAIN` block or the lowest logic block of type Block that appears above the parameterized block in the call tree. To determine the source block for a given parameterized block, determine which block invoked that parameterized block. If the calling block is `_MAIN` or of type Block, it is the source block. If the calling block is any other type (parameterized block or function block), apply the same test to the block that invoked this block. Continue back up the call tree until the `_MAIN` block or a block of type Block is found. This is the source block for the parameterized block.

### Programming OFDT, ONDTR and TMR in Parameterized Blocks

Different guidelines and rules apply depending on whether you want to use the parameterized block in more than one place in your program logic.

### Parameterized block called from one block

If your parameterized block that contains a timer will be called from only one logic block, follow these rules:

1. Call the parameterized block exactly one time per execution of its source block.
2. Choose a reference address for the timer that will not be manipulated anywhere else. The reference address may be `%R`, `%P`, `%L`, `%W`, or symbolic.

**Note:** `%L` memory is the same `%L` memory available to the source block of type Block. `%L` memory corresponds to `%P` memory when the source block is `_MAIN`.

### Parameterized block called from multiple blocks

When calling the parameterized block from multiple blocks, it is imperative to separate the timer reference memory used by each call to the parameterized block. Follow these rules and guidelines:

1. Call the parameterized block exactly one time per execution of each source block in which it appears.
2. Choose a %L reference or parameterized block formal parameter for the timer reference memory. Do not use a %R, %P, %W, or symbolic memory reference.

#### Notes:

- The strongly recommended choice is a %L location, which is inherited from the parameterized block's source block. Each source block has its own %L memory space except the \_MAIN block, which has a %P memory area instead. When the \_MAIN block calls another block, the %P mappings from the \_MAIN block are accessed by the called block as %L mappings.
- If you use a parameterized block formal parameter (word array passed-by-reference), the actual parameter that corresponds to this formal parameter must be a %L, %R, %P, %W, or symbolic reference. If the *actual parameter* is a %R, %P, %W, or symbolic reference, a unique reference address must be used by each source block.

### Recursion

If you use recursion (that is, if you have a block call itself either directly or indirectly) and your parameterized block contains an OFDT, ONDTR, or TMR, you must follow two additional rules:

- Program the source block so that it invokes the parameterized block before making any recursive calls to itself.
- Do not program the parameterized block to call itself directly.

### Using OFDT, ONDTR and TMR in UDFBs

UDFBs are user-defined logic blocks that have parameters and instance data. For details on these and other types of blocks, refer to Chapter 2.

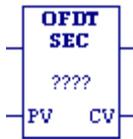
When a timer function is present inside a UDFB, and a member variable is used for the control block of a timer, the behavior of the timer may not match your expectations. If multiple instances of the UDFB are called during a logic sweep, only the first-executed instance will update its timer. If a different instance is then executed, its timer value will remain unchanged.

In the case of multiple calls to a UDFB during a logic scan, only the first call will add elapsed time to its timer functions. This behavior matches the behavior of timers in a normal program block.

### Example

A UDFB is defined that uses a member variable for a timer function block. Two instances of the function block are created: timer\_A and timer\_B. During each logic scan, both timer\_A and timer\_B are executed. However, only the member variable in timer\_A is updated and the member variable in timer\_B always remains at 0.

## Off Delay Timer



The Off-Delay Timer (OFDT) increments while power flow is off, and the timer's Current Value (CV) resets to zero when power flow is on. OFDT passes power until the specified interval PV (Preset Value) has elapsed.

Mnemonics:  
 OFDT\_SEC  
 OFDT\_TENTHS  
 OFDT\_HUNDS  
 OFDT\_THOUS

Time may be counted in the following increments:

- Seconds
- Tenths (0.1) of a second
- Hundredths (0.01) of a second
- Thousandths (0.001) of a second

The range for PV is 0 to +32,767 time units. If PV is out of range, it has no effect on the timer's word 2. The state of this timer is retentive on power failure; no automatic initialization occurs at power-up.

When OFDT receives power flow, CV is set to zero and the timer passes power to the right. The output remains on as long as OFDT receives power flow.

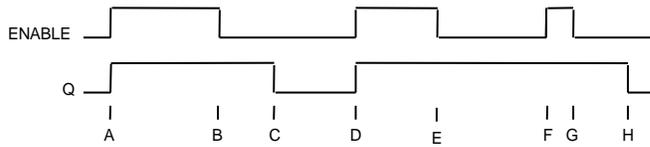
Each time the OFDT is invoked with its power flow input turned OFF, CV is updated to reflect the elapsed time since the timer was reset. OFDT continues passing power to the right until CV equals or exceeds PV. When this happens, OFDT stops passing power flow to the right and stops accumulating time. If PV is 0 or negative, the timer stops passing power flow to the right the first time that it is invoked with its power flow input OFF.

When the function receives power flow again, CV resets to zero.

### Notes:

- The best way to use an OFDT function is to invoke it with a particular reference address exactly one time each scan. Do not invoke an OFDT with the same reference address more than once per scan (inappropriate accumulation of time would result). When an OFDT appears in a program block, it accumulates time once per scan. Subsequent calls to that program block within the same scan will have no effect on its OFDTs.
- Do not program an OFDT function with the same reference address in two different blocks. You should not program a JUMP around a timer function. Also, if you use recursion (where a block calls itself either directly or indirectly), program the program block so that it invokes the timer before it makes any recursive calls to itself.
- For information on using timers inside parameterized blocks, refer to *Using OFDT, ONDTR and TMR in Parameterized Blocks*.
- An OFDT expires (turns OFF power flow to the right) the first scan that it does not receive power flow if the previous scan time was greater than PV.
- When OFDT is used in a program block that is not called every scan, the timer accumulates time between calls to the program block unless it is reset. This means that OFDT functions like a timer operating in a program with a much slower scan than the timer in the main program block. For program blocks that are inactive for a long time, OFDT should be programmed to allow for this catch-up feature. For example, if a timer in a program block is reset and the program block is not called (is inactive) for four minutes, when the program block is called, four minutes of time will already have accumulated. If the enable input is OFF, these four minutes are applied to the timer (that is, CV is set to 4 minutes).

Timing diagram



- A. ENABLE and Q both go high; timer is reset (CV = 0).
- B. ENABLE goes low; timer starts accumulating time.
- C. CV reaches PV; Q goes low and timer stops accumulating time.
- D. ENABLE goes high; timer is reset (CV = 0).
- E. ENABLE goes low; timer starts accumulating time.
- F. ENABLE goes high; timer is reset (CV = 0) before CV had a chance to reach PV. (The diagram is not to scale.)
- G. ENABLE goes low; timer begins accumulating time.
- H. CV reaches PV; Q goes low and timer stops accumulating time.

Operands for OFDT



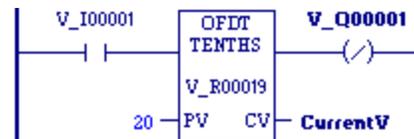
Warning

**Do not use the Address, Address+1, or Address+2 addresses with other instructions. Overlapping references cause erratic timer operation.**

Parameter	Description	Allowed Operands	Optional
Address (????)	The beginning address of a three-word WORD array: Word 1: Current value (CV) Word 2: Preset value (PV) Word 3: Control word	R, W, P, L, symbolic	No
PV	The Preset Value, used when the timer is enabled or reset. $0 \leq PV \leq +32,767$ . If PV is out of range, it has no effect on Word 2.	All except S, SA, SB, SC	Optional
CV	The current value of the timer.	All except S, SA, SB, SC, constant	Optional

Example for OFDT

The output action is reversed by the use of a negated output coil. In this circuit, the OFDT timer turns off negated output coil %Q0001 whenever contact %I0001 is closed. After %I0001 opens, %Q0001 stays off for 2 seconds then turns on.



## On Delay Stopwatch Timer



The retentive On-Delay Stopwatch Timer (ONDTR) increments while it receives power flow and holds its value when power flow stops.

Time may be counted in the following increments:

- Seconds
- Tenths (0.1) of a second
- Hundredths (0.01) of a second
- Thousandths (0.001) of a second

Mnemonics:

ONDTR\_SEC  
ONDTR\_TENTHS  
ONDTR\_HUNDS  
ONDTR\_THOUS

The range is 0 to +32,767 time units. The state of this timer is retentive on power failure; no automatic initialization occurs at power-up.

When ONDTR first receives power flow, it starts accumulating time (Current Value (CV)). When the CV equals or exceeds Preset Value (PV), output Q is energized, regardless of the state of the power flow input.

As long as the timer continues to receive power flow, it continues accumulating until CV equals the maximum value (+32,767 time units). Once the maximum value is reached, it is retained and Q remains energized regardless of the state of the enable input.

When power flow to the timer stops, CV stops incrementing and is retained. Output Q, if energized, will remain energized. When ONDTR receives power flow again, CV again increments, beginning at the retained value.

When reset (R) receives power flow and PV is not equal to zero, CV is set back to zero and output Q is de-energized.

**Note:** If PV equals zero, the time is disabled and the reset is activated, and the output of the time becomes high. Subsequent removal of the reset or activation of input will have no effect on the timer output; the output of the time remains high.

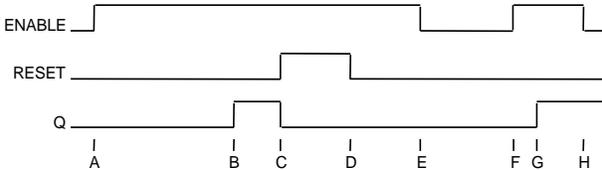
ONDTR passes power flow to the right when CV is greater than or equal to PV. Since no automatic initialization to the outgoing power flow state occurs at power-up, the power flow state is retentive across power failure.

### Notes:

- The best way to use an ONDTR function is to invoke it with a particular reference address exactly one time each scan. Do not invoke an ONDTR with the same reference address more than once per scan (inappropriate accumulation of time would result). When an ONDTR appears in a program block, it will only accumulate time once per scan. Subsequent calls to that same program block within the same scan will have no effect on its ONDTRs. Do not program an ONDTR function with the same reference address in two different blocks. You should not program a JUMPN around a timer function. Also, if you use recursion (that is, having a block call itself either directly or indirectly), program the program block so that it invokes the timer before it makes any recursive calls to itself.
- For information on using timers inside parameterized blocks, refer to *Using OFDT, ONDTR and TMR in Parameterized Blocks*.
- An ONDTR expires (passes power flow to the right) the first scan that is enabled and not reset if the previous scan time was greater than PV.

- When ONDTR is used in a program block that is not called every scan, it accumulates time between calls to the program block unless it is reset. This means that ONDTR functions like a timer operating in a program with a much slower scan than the timer in the main program block. For program blocks that are inactive for a long time, ONDTR should be programmed to allow for this catch-up feature. For example, if a timer in a program block is reset and the program block is not called (is inactive) for four minutes, when the program block is called, four minutes of time will already have accumulated. If the enable input is ON and the reset input is OFF, these four minutes are applied to the timer (that is, CV is set to 4 minutes).

**Timing diagram**



- ENABLE goes high; timer starts accumulating.
- Current value (CV) reaches preset value (PV); Q goes high. Timer continues to accumulate time until ENABLE goes low, RESET goes high or current value becomes equal to the maximum time.
- RESET goes high; Q goes low, accumulated time is reset (CV=0).
- RESET goes low; timer then starts accumulating again, as ENABLE is high.
- ENABLE goes low; timer stops accumulating. Accumulated time stays the same.
- ENABLE goes high again; timer continues accumulating time.
- CV becomes equal to PV; Q goes high. Timer continues to accumulate time until ENABLE goes low, RESET goes high or CV becomes equal to the maximum time.
- ENABLE goes low; timer stops accumulating time.

**Operands for On Delay Stopwatch Timer**



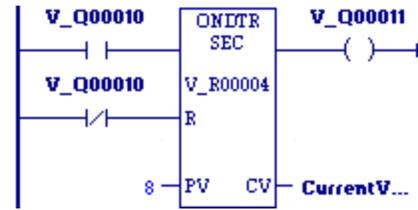
**Warning**

**Do not use the Address, Address+1, or Address+2 addresses with other instructions. Overlapping references cause erratic timer operation.**

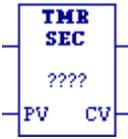
Parameter	Description	Allowed Operands	Optional
Address (????)	Beginning address of a three-word WORD array: Word 1: Current value (CV) Word 2: Preset value (PV) Word 3: Control word	R, W, P, L, symbolic	No
R	When R is ON, it resets the Current Value (Word 1) to zero.	Power flow	Optional
PV	The Preset Value, used when the timer is enabled or reset. $0 \leq PV \leq +32,767$ . If PV is out of range, it has no effect on Word 2.	All except S, SA, SB, SC	Optional
CV	Current Value of the timer	All except S, SA, SB, SC and constant	Optional

**Example for On Delay Stopwatch Timer**

A retentive on-delay timer is used to create a signal (%Q0011) that turns on 8.0 seconds after %Q0010 turns on, and turns off when %Q0010 turns off.



## On Delay Timer



The On-Delay Timer (TMR) increments while it receives power flow and resets to zero when power flow stops. The timer passes power after the specified interval PV (Preset Value) has elapsed, as long as power is received.

Mnemonics:  
 TMR\_SEC  
 TMR\_TENTHS  
 TMR\_HUNDS  
 TMR\_THOUS

The range for PV is 0 to +32,767 time units. If PV is out of range, it has no effect on the timer's word 2. The state of this timer is retentive on power failure; no automatic initialization occurs at power-up.

Time may be counted in the following increments:

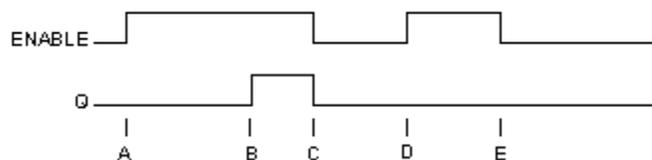
- Seconds
- Tenths (0.1) of a second
- Hundredths (0.01) of a second
- Thousandths (0.001) of a second

When TMR is invoked with its power flow input turned OFF, its Current Value (CV) is reset to zero, and the timer does not pass power flow to the right. Each time the TMR is invoked with its power flow input turned ON, CV is updated to reflect the elapsed time since the timer was reset. When CV reaches PV, the timer function passes power flow to the right.

### Notes:

- The best way to use a TMR function is to invoke it with a particular reference address exactly one time each scan. Do not invoke a TMR with the same reference address more than once per scan (inappropriate accumulation of time would result). When a TMR appears in a program block, it will only accumulate time once per scan. Subsequent calls to that same program block within the same scan will have no effect on its TMRs. Do not program a TMR function with the same reference address in two different blocks. You should not program a JUMP around a timer function. Also, if you use recursion (that is, having a block call itself either directly or indirectly), program the program block so that it invokes the timer before it makes any recursive calls to itself.
- For information on using timers inside parameterized blocks, refer to *Using OFDT, ONDTR and TMR in Parameterized Blocks*.
- A TMR timer expires (passes power flow to the right) the first scan that it is enabled if the previous scan time was greater than PV.
- When TMR is used in a program block that is not called every scan, TMR accumulates time between calls to the program block unless it is reset. This means that it functions like a timer operating in a program with a much slower sweep than the timer in the main program block. For program blocks that are inactive for a long time, TMR should be programmed to allow for this catch-up feature. For example, if a timer in a program block is reset and the program block is not called (is inactive) for 4 minutes, when the program block is called, four minutes of time will already have accumulated. If the enable input is ON, these four minutes are applied to the timer (i.e. CV is set to 4 minutes).

### Timing Diagram



ENABLE goes high; timer begins accumulating time.

CV reaches PV; Q goes high and timer continues accumulating time.

ENABLE goes low; Q goes low; timer stops accumulating time and CV is cleared.

ENABLE goes high; timer starts accumulating time.

ENABLE goes low before current value reaches PV; Q remains low; timer stops accumulating time and is cleared to zero (CV=0).

### Operands for On Delay Timer



#### Warning

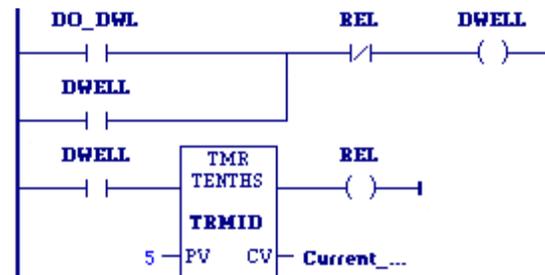
**Do not use the Address, Address+1, or Address+2 addresses with other instructions. Overlapping references cause erratic timer operation.**

Parameter	Description	Allowed Operands	Optional
????	The beginning address of a three-word WORD array: Word 1: Current value (CV) Word 2: Preset value (PV) Word 3: Control word	R, W, P, L, symbolic	No
PV	The Preset Value, used when the timer is enabled or reset. $0 \leq PV \leq +32,767$ . If PV is out of range, it has no effect on Word 2.	All except S, SA, SB, SC	Yes
CV	The current value of the timer.	All except S, SA, SB, SC and constant	Yes

#### Example for On Delay Timer

An on-delay timer with address TMRID is used to control the length of time that a coil is on. This coil has been assigned the variable DWELL. When the normally open (momentary) contact DO\_DWL is ON, coil DWELL is energized.

The contact of coil DWELL keeps coil DWELL energized (when contact DO\_DWL is released) and also starts the timer TMRID. When TMRID reaches its preset value of five tenths of a second, coil REL energizes, interrupting the latched-on condition of coil DWELL. The contact DWELL interrupts power flow to TMRID, resetting its current value and de-energizing coil REL. The circuit is then ready for another momentary activation of contact DO\_DWL.



### 4.13.3 Standard Timer Function Blocks

The standard timers are a pulse timer (TP), an on-delay timer (TON), and an off-delay timer (TOF). The pulse timer block can be used to generate output pulses of a given duration. The on-delay timer can be used to delay setting an output ON for a fixed period after an input is set ON. The off-delay timer can be used to delay setting an output OFF for a fixed period after an input goes OFF so that the output is held on for a given period longer than the input.

Notes:

- Any block type can contain calls to the standard timers. (See Chapter 2 for a discussion of the various block types.)
- Interrupt blocks can contain standard timers.
- An instance of a timer can be passed by reference to a parameterized block or UDFB.
- When the timer stops timing as a result of reaching its Preset Time (PT), the Elapsed Time (ET) contains the actual timer duration. For example, if the Preset Time was specified as 333ms, but the timer actually timed to 350ms, the 350ms value is saved in ET.

#### Data Required for Standard Timer Function Blocks

Each invocation of a timer has associated instance data that persists from one execution of the timer to the next. Instance variables are automatically located in symbolic memory. (You cannot specify an address.) You can specify a stored value for each element. The user logic cannot modify the values.

Each timer instance variable has the following structure. Elements of a timer structure cannot be published.

The instance data type for each timer must be the same as the timer type:

The TOF timer requires an instance variable of type TOF.

The TON timer requires an instance variable of type TON.

The TP timer requires an instance variable of type TP.

Element	Type	Description	Details
IN	BOOL	Timer input	Cannot be accessed in user logic.
PT	DINT	Preset time	Cannot be accessed in user logic.
ET	DINT	Elapsed time	Read only. Accessible in user logic.
Q	BOOL	Set ON when timer finishes timing	Read only. Accessible in user logic.
ENO	BOOL	Enable output	Read only. Accessible in user logic.
TI	BOOL	Set ON when the timer instance is timing (that is, ET is incrementing).	Read only. Accessible in user logic.

#### Resetting the Timer

The preset time (PT) may be changed while the timer is timing to affect the duration.

When the timer reaches PT, the timer stops timing and the elapsed time parameter (ET) contains the actual timer duration.

To reset the timer function block, set the PT input to 0. When the function block resets:

- ET is set to 0
- Q is set to off (0)
- The TI element is set to 0
- The IN parameter is ignored

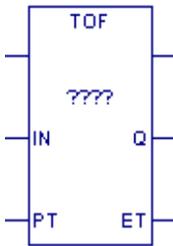
## Operands

TOF, TON and TP have the same input and output parameters, except for the instance variable, which must be the same type as the instruction.

**Note:** Writing or forcing values to the instance data elements IN, PT, Q, ET, ENO or TI may cause erratic operation of the timer function block.

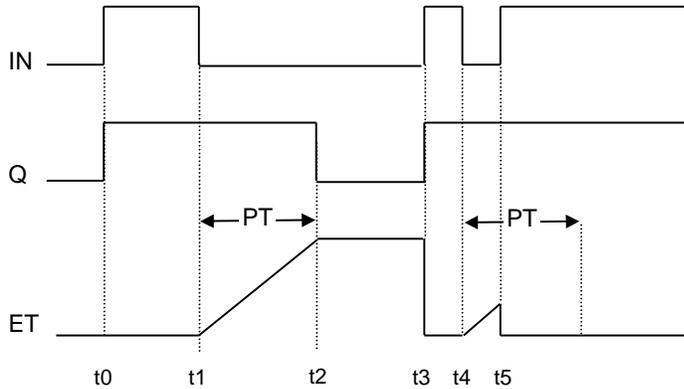
Parameter	Description	Allowed Types	Allowed Operands	Optional
????	Structure variable containing the internal data for the timer instance. (Refer to <i>Data Required for Standard Timer Function Blocks.</i> )	TOF, TON, or TP. Must be same type as the instruction.	NA	No
IN	Timer input. Controls when the timer will accumulate time.  TON and TP will begin to time when IN transitions from OFF to ON.  TOF will begin to time when IN transitions from ON to OFF.	Flow	NA	Yes
PT	Preset time (in ms). Indicates the amount of time the timer will time until turning Q either ON or OFF, depending on the timer type.  Setting PT to 0 resets the timer.	DINT	All except S, SA, SB, SC	Yes
Q	Timer output. Action depends on the timer type.  When TP is timing, Q is ON.  When TON is done timing, Q turns ON.  When TOF is done timing, Q turns OFF.	Flow	NA	Yes
ET	Elapsed time. Indicates the length of time, (in ms), that the timer has been measuring time.	DINT	All except S, SA, SB, SC and constants	Yes

### Timer Off Delay



When the input IN transitions from ON to OFF, the timer starts timing until a specified period of time (PT) has elapsed, then sets the output Q to OFF.

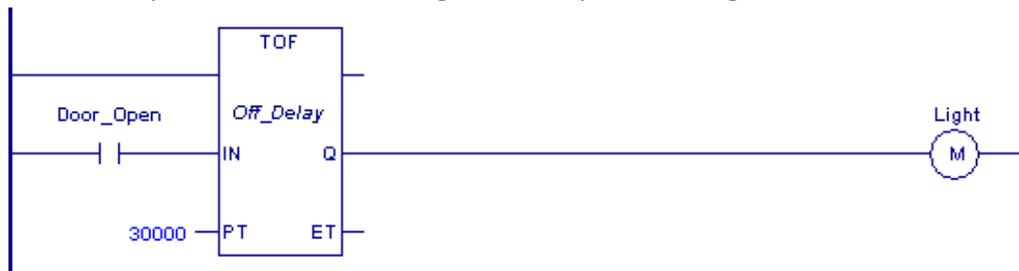
### Timing Diagram



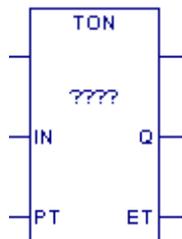
- t0 When input IN is set to ON, the output Q follows and remains ON. The elapsed time, ET, does not increment.
- t1 When IN goes OFF, the timer starts to measure time and ET increments. ET continues to increment until its value equals the preset time, PT.
- t2 When ET equals PT, Q is set to OFF and ET remains at the preset time, PT.
- t3 When input IN is set to ON, the output Q follows and remains ON. ET is set to 0.
- t4 When IN is set to OFF, ET, begins incrementing. When IN is OFF for a period shorter than that specified by PT, Q remains ON.
- t5 When IN is set to ON, ET is set to 0.

### Example

In the following sample rung, a TOF function block is used to keep Light ON for 30,000ms (30 seconds) after Door\_Open is set to OFF. As long as Door\_Open is ON, Light remains ON.

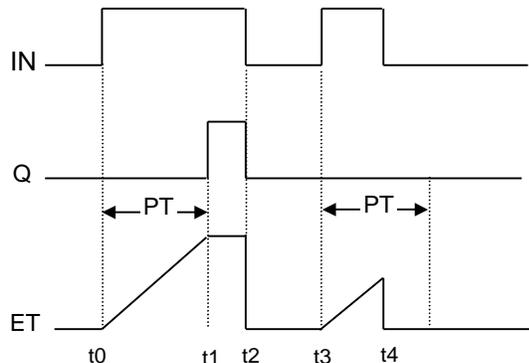


### Timer On Delay



When the input IN transitions from OFF to ON, the timer starts timing until a specified period of time (PT) has elapsed, then sets the output Q to ON.

### Timing Diagram



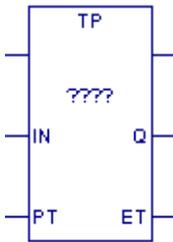
- t0 When input IN is set to ON, the timer starts to measure time and the elapsed time output ET starts to increment. The output Q remains OFF and ET continues to increment until its value equals the preset time, PT.
- t1 When ET equals PT, the output Q is goes ON, and ET remains at the preset time, PT. Q remains ON until IN goes OFF.
- t2 When IN is set to OFF, Q goes OFF and ET is set to 0.
- t3 When IN is set to ON, ET starts To increment.
- t4 If IN is ON for a shorter time than the delay specified in PT, the output Q remains OFF. ET is set to 0 when IN is set to OFF.

### Example

In the following sample rung, a TON function block is used to delay setting Start to ON for 1 minute (60,000ms) after Preheat is set to ON.

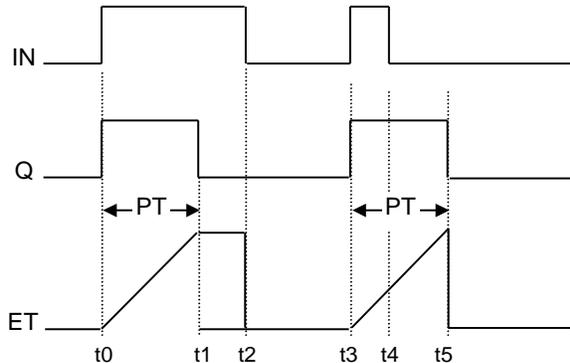


### Timer Pulse



When the input IN transitions from OFF to ON, the timer sets the output Q to ON for the specified time interval, PT

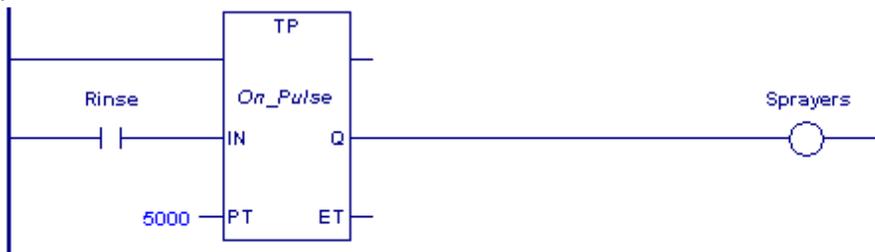
### Timing Diagram



- t0 When input IN is set to ON, the timer starts to measure time and the elapsed time output, ET, increments until its value equals that of the specified preset time, PT. Q is set to 0 on until ET equals PT.
- t1 When ET equals PT, Q is set to OFF. The value of ET is held until IN is set to OFF.
- t2 When IN is set to OFF, ET is set to 0.
- t3 When IN is set to ON, the timer starts to measure time and ET begins incrementing. Q is set to ON.
- t4 If the input is OFF for a period shorter than the input PT, the output Q remains on and ET continues incrementing.
- t5 When ET equals PT, Q is set to OFF and ET is set to 0.

### Example

In the following sample rung, a TP function block is used to set Sprayers to ON for a 5-second (5000ms) pulse.





## Chapter 5 Function Block Diagram (FBD)

---

Function Block Diagram (FBD) is an IEC 61131-3 graphical programming language that represents the behavior of functions, function blocks and programs as a set of interconnected graphical blocks.

The block types Block, Parameterized Block, and Function Block can be programmed in FBD. The `_MAIN` program block can also be programmed in FBD. For details on blocks, refer to *Program Data* in Chapter 3. For information on using the FBD editor in the programming software, refer to the online help.

For an overview of the types of operands that can be used with instructions, refer to *Operands for Instructions* in Chapter 3.

Most functions and function blocks implemented in FBD are the same as their LD counterparts. Instructions that are implemented differently are discussed in detail in this chapter. FBD has the following general differences compared to LD:

- In FBD, except for timers and counters, functions and function blocks do not have EN or ENO parameters.
- In FBD, all functions and function blocks display a solve order, which is calculated by the FBD editor.

The FBD implementation of the PACSystems instruction set includes the following categories:

- *Advanced Math Functions*
- *Bit Operation Functions*
- *Comments*
- *Comparison Functions*
- *Control Functions*
- *Counters*
- *Data Move Functions*
- *Math Functions*
- *Program Flow Functions*
- *Timers*
- *Type Conversion Functions*
- *PROFINET Communication*  
Consists of the `PNIO_DEV_COMM` function. For details, refer to the *PACSystems RX3i & RSTi-EP PROFINET I/O Controller Manual*, GFK-2571.

## 5.1 Note on Reentrancy

When a function block is created using the FBD language, the wires are created as global variables, not as members. This has two consequences. First, if there are multiple instances of that block in the program, the wires will show the values from the last instance executed during the sweep, not the values for the instance being viewed. This will give the appearance of incorrect operation while actually working properly.

The second consequence is that function blocks written in FBD are not reentrant. If you have multiple instances of a block, and one of them can be called by an interrupt, then it is possible for the interrupt to trigger while one instance of the block is in process, change the values of the wires, and then return control to the original block. This will result in improper operation.

There is a work-around for both of these symptoms, which is to create the wires as member variables rather than global variables. This must be done manually by creating member variables of the appropriate types. You can then right-click on each wire in the FBD diagram and use the *Replace Variable* command to change the wire from a global variable to a member variable.

---

### Caution

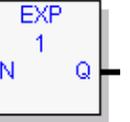
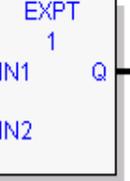
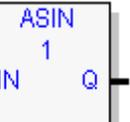
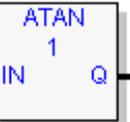
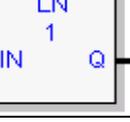
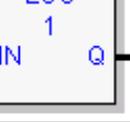
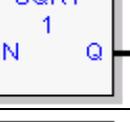
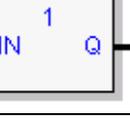


**Blocks written in the FBD language are not reentrant. Because of this, if the block is called directly, or indirectly, from an interrupt, the block must not be called anywhere else in the program, except when steps are taken to explicitly make it reentrant (see above). Doing so can lead to unexpected operation. This applies to basic blocks, parameterized blocks, and user-defined function blocks written in FBD.**

---

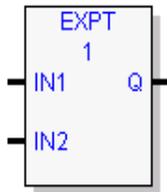
## 5.2 Advanced Math Functions

The Advanced Math functions perform logarithmic, exponential, square root, trigonometric, and inverse trigonometric operations.

Function	Description
	<p>Absolute value. Finds the absolute value of a double-precision integer (DINT), signed single-precision integer (INT), REAL or LREAL (floating-point) value. The mnemonic specifies the value's data type.</p> <p>For details, refer to <i>Absolute Value</i> in Chapter 4.</p>
	<p>Exponential. Raises <b>e</b> to the value specified in IN (<math>e^{IN}</math>). Calculates the inverse natural logarithm of the IN operand.</p> <p>For details, refer to <i>Exponential/Logarithmic Functions</i> in Chapter 4.</p>
	<p>Exponential. Calculates IN1 to the power of IN2 (<math>IN1^{IN2}</math>).</p> <p>For details, refer to <i>EXPT Function</i> below.</p>
	<p>Inverse trig. Calculates the inverse cosine of the IN operand and expresses the result in radians.</p> <p>For details, refer to <i>Inverse Trig – ASIN, ACOS, and ATAN</i> in Chapter 4.</p>
	<p>Inverse trig. Calculates the inverse sine of the IN operand and expresses the result in radians.</p> <p>For details, refer to <i>Inverse Trig – ASIN, ACOS, and ATAN</i> in Chapter 4.</p>
	<p>Inverse trig. Calculates the inverse tangent of the IN operand and expresses the result in radians.</p> <p>For details, refer to <i>Inverse Trig – ASIN, ACOS, and ATAN</i> in Chapter 4.</p>
	<p>Logarithmic. Calculates the natural logarithm of the operand IN.</p> <p>For details, refer to <i>Exponential/Logarithmic Functions</i> in Chapter 4.</p>
	<p>Logarithmic. Calculates the base 10 logarithm of the operand IN.</p> <p>For details, refer to <i>Exponential/Logarithmic Functions</i> in Chapter 4.</p>
	<p>Square root. Calculates the square root of the operand IN and stores the result in Q.</p> <p>For details, refer to <i>Square Root</i> in Chapter 4.</p>
	<p>Trig. Calculates the cosine of the operand IN, where IN is expressed in radians.</p> <p>For details, refer to <i>Trig Functions</i> in Chapter 4.</p>

<b>Function</b>	<b>Description</b>
 The diagram shows a rectangular function block labeled 'SIN' at the top. Below the label is the number '1'. On the left side, there is an input terminal labeled 'IN'. On the right side, there is an output terminal labeled 'Q'.	Calculates the sine of the operand IN, where IN is expressed in radians. For details, refer to <i>Trig Functions</i> in Chapter 4.
 The diagram shows a rectangular function block labeled 'TAN' at the top. Below the label is the number '1'. On the left side, there is an input terminal labeled 'IN'. On the right side, there is an output terminal labeled 'Q'.	Calculates the tangent of the operand IN, where IN is expressed in radians. For details, refer to <i>Trig Functions</i> in Chapter 4.

### 5.2.1 EXPT Function



The Power of X (EXPT) function raises the value of input IN1 to the power specified by the value IN2 and places the result in Q. The EXPT function operates on REAL or LREAL input value(s) and place the result in output Q. The instruction is not carried out if one of the following invalid conditions occurs:

- IN1 < 0, for EXPT
- IN1 or IN2 is a NaN (Not a Number)

Invalid operations (error cases) may yield results that are different from those in the LD implementation of this function.

#### Operands of the EXPT Function

<b>Parameter</b>	<b>Description</b>	<b>Allowed Types</b>	<b>Allowed Operands</b>	<b>Optional</b>
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN or IN1	For EXP, LOG, and LN, IN contains the REAL value to be operated on. The EXPT function has two inputs, IN1 and IN2. For EXPT, IN1 is the base value and IN2 is the exponent.	REAL, LREAL	All except variables located in %S–%SC	No
IN2 (EXPT)	The REAL exponent for EXPT.	REAL, LREAL	All except variables located in %S–%SC	No
Q	Contains the REAL logarithmic/exponential value of IN or of IN1 and IN2.	REAL, LREAL	All except constants and variables located in %S–%SC	No

### 5.3 Bit Operation Functions

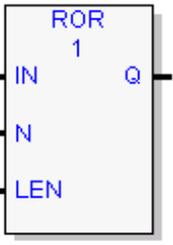
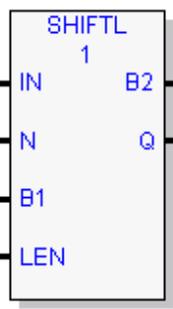
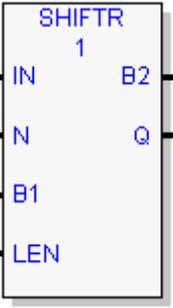
The Bit Operation functions perform comparison, logical, and move operations on bit strings. Bit Operation functions treat each WORD or DWORD data as a continuous string of bits, with bit 1 of the WORD or DWORD being the Least Significant Bit (LSB). The last bit of the WORD or DWORD is the Most Significant Bit (MSB).



#### Warning

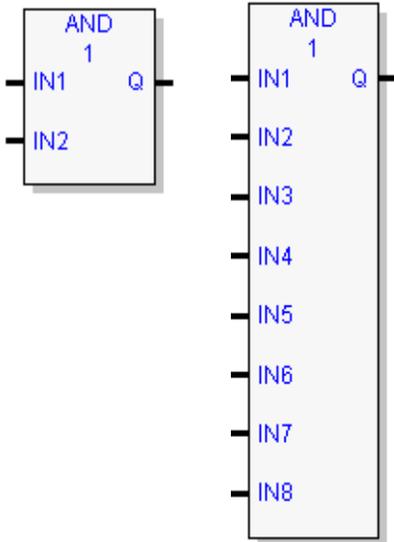
**Overlapping input and output reference address ranges in multiword functions is not recommended, as it can produce unexpected results.**

Function	Description
	<p>Logical AND. Compares the bit strings IN1 and IN2 bit by bit. When the corresponding bits are both 1, places a 1 in the corresponding location in output string Q; otherwise, places a 0 in the corresponding location in Q.</p> <p>If additional inputs (IN3 through IN8) are used, each additional bit string is compared to the string in Q and the result is placed in Q.</p> <p>For details, refer to <i>Logical AND</i>.</p>
	<p>Logical OR. Compares the bit strings IN1 and IN2 bit by bit. When a pair of corresponding bits are both 0, places a 0 in the corresponding location in output string Q; otherwise, places a 1 in the corresponding location in Q.</p> <p>If additional inputs (IN3 through IN8) are used, each additional bit string is compared to the string in Q and the result is placed in Q.</p> <p>For details, refer to <i>Logical OR</i>.</p>
	<p>Logical XOR. Compares the bit strings IN1 and IN2 bit by bit. When a pair of corresponding bits are different, places a 1 in the corresponding location in the output bit string Q; when a pair of corresponding bits are the same, places a 0 in Q.</p> <p>If additional inputs (IN3 through IN8) are used, each additional bit string is compared to the string in Q and the result is placed in Q.</p> <p>For details, refer to <i>Logical XOR</i>.</p>
	<p>Logical NOT. Sets the state of each bit in output bit string Q to the opposite state of the corresponding bit in bit string IN1.</p> <p>For details, refer to <i>Logical NOT</i>.</p>
	<p>Rotate Bits Left. Rotates all the bits in a string a specified number of places to the left.</p> <p>For details, refer to <i>Bit Operation Functions</i> in Chapter 4.</p>

Function	Description
	<p>Rotate Bits Right. Rotates all the bits in a string a specified number of places to the right. For details, refer to <i>Bit Operation Functions</i> in Chapter 4.</p>
	<p>Shift Bits Left. Shifts all the bits in a word or string of words to the left by a specified number of places. For details, refer to <i>Bit Operation Functions</i> in Chapter 4.</p>
	<p>Shift Bits Right. Shifts all the bits in a word or string of words to the right by a specified number of places. For details, refer to <i>Bit Operation Functions</i> in Chapter 4.</p>

### 5.3.1 Logical AND, Logical OR, and Logical XOR

The Logical functions examine each bit in bit string IN1 and the corresponding bit in bit string IN2, beginning with the least significant bit in each string, and places the result in Q. If additional inputs (IN3 up to IN8) are used, the function compares each bit in the input with the corresponding bit in Q and places the result in Q. The comparison is repeated for each input that is used. The input bit strings specified in IN1 ... IN8 may overlap.

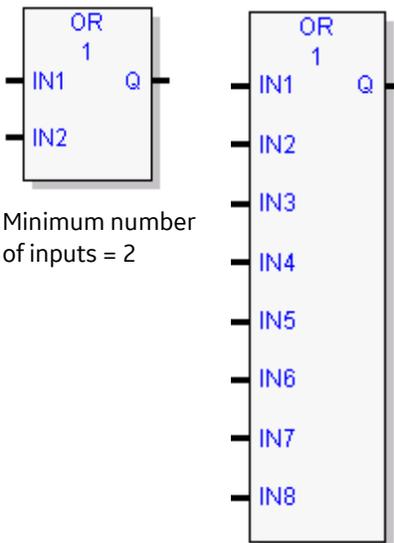


#### Logical AND

If both bits examined by the Logical AND function are 1, AND places a 1 in the corresponding location in output string Q. If either bit is 0 or both bits are 0, AND places a 0 in string Q in that location.

**Tip:** You can use the Logical AND function to build masks or screens, where only certain bits are passed (the bits opposite a 1 in the mask), and all other bits are set to 0.

Minimum number of inputs = 2  
Maximum number of inputs = 8



Minimum number of inputs = 2

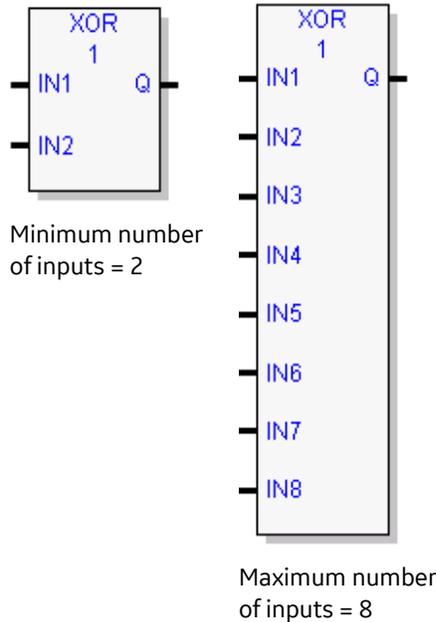
#### Logical OR

If either bit examined by the Logical OR function is 1, OR places a 1 in the corresponding location in output string Q. If both bits are 0, Logical OR places a 0 in string Q in that location.

**Tips:**

- You can use the Logical OR function to combine strings or to control many outputs with one simple logical structure. The Logical OR function is the equivalent of two relay contacts in parallel multiplied by the number of bits in the string.
- You can use the Logical OR function to drive indicator lamps directly from input states or to superimpose blinking conditions on status lights.

Maximum number of inputs = 8



### Logical XOR

If the bits in the strings examined by XOR are different, a 1 is placed in the corresponding position in the output bit string.

For each pair of bits examined, if only one bit is 1, XOR places a 1 in the corresponding location in string Q.

If both bits are 0, XOR places a 0 in the corresponding location in string Q.

#### Tips:

- If string IN2 and output string Q begin at the same reference, a 1 placed in string IN1 will cause the corresponding bit in string IN2 to alternate between 0 and 1, changing state with each scan as long as input is received.
- You can program longer cycles by pulsing the input to the function at twice the desired rate of flashing. The input pulse should be one scan long (one-shot type coil or self-resetting timer).
- You can use XOR to quickly compare two bit strings, or to blink a group of bits at the rate of one ON state per two scans.
- XOR is useful for transparency masks.

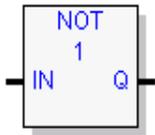
### Operands for AND, OR, and XOR

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN1	The value to operate on.	BOOL, WORD DWORD	All	No
IN2 (Must be the same data type as IN1.)	The value to operate on.	BOOL, WORD DWORD	All	No
IN3 ... IN8 (Must be the same data type as IN1.)	Values to operate on.	BOOL, WORD DWORD	All	Yes
Q (Must be the same data type as IN1 and IN2.)	The operation's result.	BOOL, WORD DWORD	All except constants and variables located in %S memory	No

### Properties for AND, OR, and XOR

Property	Valid Range
Number of Inputs	2 to 8

### 5.3.2 Logical NOT



The Logical Not or Logical Invert (NOT) function sets the state of each bit in the output bit string Q to the opposite of the state of the corresponding bit in bit string IN1.

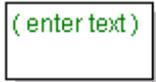
All bits are altered on each scan that input is received, making output string Q the logical complement of input string IN1.

#### Operands

<b>Parameter</b>	<b>Description</b>	<b>Allowed Types</b>	<b>Allowed Operands</b>	<b>Optional</b>
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN1	The input string to NOT.	WORD DWORD	All	No
Q (Must be the same data type as IN1)	The NOT's result.	WORD DWORD	All except constants and variables located in %S memory	No

## 5.4 Comments

### 5.4.1 Text Block



The Text block is used to place an explanation in the program. When you type in a comment, the first few words are displayed.



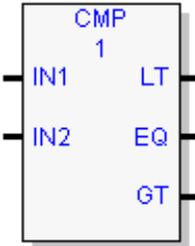
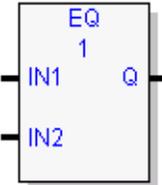
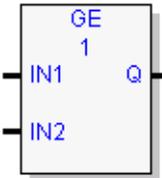
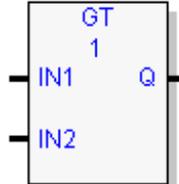
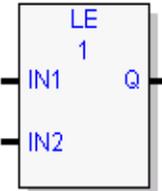
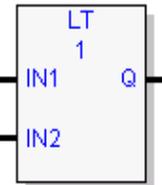
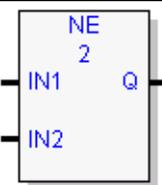
To increase the size of the text box and display more text, select the box and drag one of the handles.

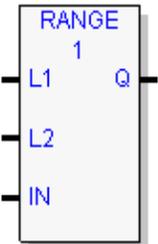
There are no operands for the Text block.

- Editing a comment makes the Programmer lose equality.
- Comment text is downloaded to the controller and retrieved upon Logic Upload.

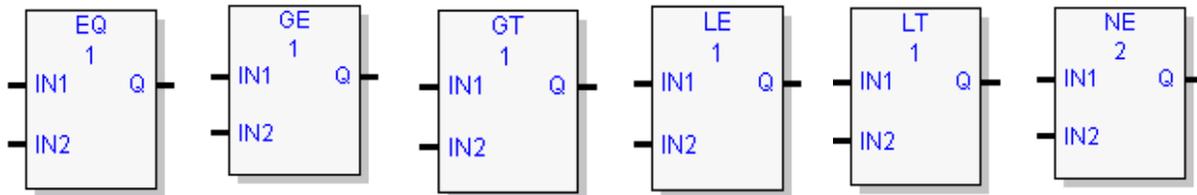
## 5.5 Comparison Functions

Comparison functions compare two values of the same data type or determine whether a number lies within a specified range. The original values are unaffected.

Function	Description
 <p>The diagram shows a rectangular function block labeled 'CMP 1'. It has two input terminals on the left labeled 'IN1' and 'IN2'. On the right side, there are three output terminals labeled 'LT', 'EQ', and 'GT'.</p>	<p>Compare. Compares two numbers, IN1 and IN2. For details, refer to <i>Relational Functions</i> in Chapter 4.</p>
 <p>The diagram shows a rectangular function block labeled 'EQ 1'. It has two input terminals on the left labeled 'IN1' and 'IN2'. On the right side, there is one output terminal labeled 'Q'.</p>	<p>Equal. Tests two numbers for equality. For details, refer to <i>Comparison Functions</i>.</p>
 <p>The diagram shows a rectangular function block labeled 'GE 1'. It has two input terminals on the left labeled 'IN1' and 'IN2'. On the right side, there is one output terminal labeled 'Q'.</p>	<p>Greater Than or Equal. Tests whether one number is greater than or equal to another. For details, refer to <i>Comparison Functions</i>.</p>
 <p>The diagram shows a rectangular function block labeled 'GT 1'. It has two input terminals on the left labeled 'IN1' and 'IN2'. On the right side, there is one output terminal labeled 'Q'.</p>	<p>Greater Than. Tests whether one number is greater than another. For details, refer to <i>Comparison Functions</i>.</p>
 <p>The diagram shows a rectangular function block labeled 'LE 1'. It has two input terminals on the left labeled 'IN1' and 'IN2'. On the right side, there is one output terminal labeled 'Q'.</p>	<p>Less Than or Equal. Tests whether one number is less than or equal to another. For details, refer to <i>Comparison Functions</i>.</p>
 <p>The diagram shows a rectangular function block labeled 'LT 1'. It has two input terminals on the left labeled 'IN1' and 'IN2'. On the right side, there is one output terminal labeled 'Q'.</p>	<p>Less Than. Tests whether one number is less than another. For details, refer to <i>Comparison Functions</i>.</p>
 <p>The diagram shows a rectangular function block labeled 'NE 2'. It has two input terminals on the left labeled 'IN1' and 'IN2'. On the right side, there is one output terminal labeled 'Q'.</p>	<p>Not Equal. Tests whether two numbers are not equal. For details, refer to <i>Comparison Functions</i>.</p>

<b>Function</b>	<b>Description</b>
	Range. Tests whether one number is within the range defined by two other supplied numbers. For details, refer to <i>Relational Functions</i> in Chapter 4.

### 5.5.1 Equal, Not Equal, Greater or Equal, Greater Than, Less or Equal, Less Than



The relational functions compare input IN1 to input IN2. These operands must be the same data type. If inputs IN1 and IN2 are equal, the function outputs the result to Q, unless IN1 and/or IN2 is NaN (Not a Number). The following relational functions can be used to compare two numbers:

Function	Definition	Relational Statement
EQ	Equal	IN1=IN2
NE	Not Equal	IN1≠IN2
GE	Greater Than or Equal	IN1≥IN2
GT	Greater Than	IN1>IN2
LE	Less Than or Equal	IN1≤IN2
LT	Less Than	IN1<IN2

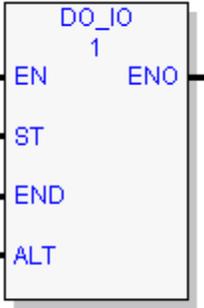
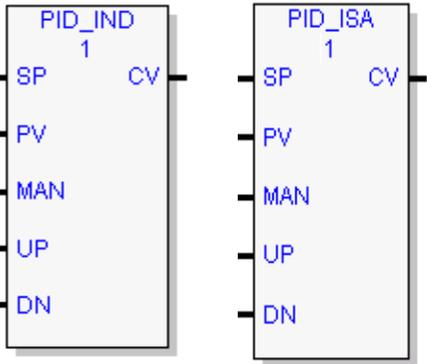
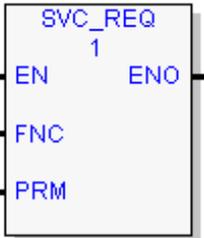
**Tip:** To compare values of different data types, first use conversion functions to make the types the same.

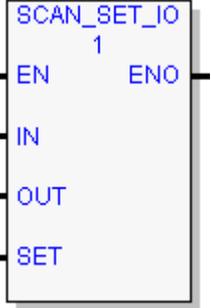
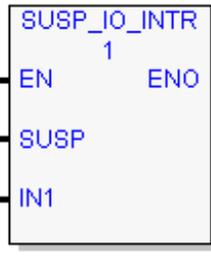
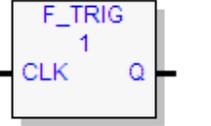
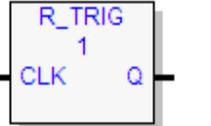
#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN1	The first value to be compared; the value on the left side of the relational statement.	BOOL (for EQ and NE functions only), BYTE, DINT, DWORD, INT, REAL, LREAL, UINT, WORD	All except S, SA, SB, SC	No
IN2	The second value to be compared; the value on the right side of the relational statement. IN2 must be the same data type as IN1.			No
Q	If the relational statement is true, Q=1.	BOOL Bit reference in a non-BOOL variable.	I, Q, G, M, T, SA, SB, SC All except constants.	No

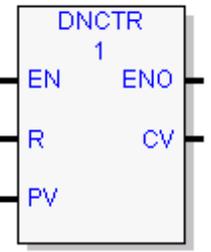
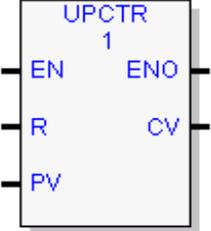
## 5.6 Control Functions

The control functions limit program execution and change the way the CPU executes the application program.

Function	Description
	<p>Do I/O Interrupt. For one scan, immediately services a specified range of inputs or outputs. (All inputs or outputs on a module are serviced if any reference locations on that module are included in the DO I/O function. Partial I/O module updates are not performed.) Optionally, a copy of the scanned I/O can be placed in internal memory, rather than at the real input points.</p> <p>For details, refer to <i>Control Functions</i> in Chapter 4.</p>
	<p>Mask I/O Interrupt. Mask or unmask an interrupt from an I/O board when using I/O variables. If not using I/O variables, use <i>SVC_REQ 17: Mask/Unmask I/O Interrupt</i>, described in Chapter 6.</p> <p>For details, refer to <i>Control Functions</i> in Chapter 4.</p>
	<p>Proportional Integral Derivative (PID) Control. Provides two PID closed-loop control algorithms:</p> <ul style="list-style-type: none"> <li>Standard ISA PID algorithm (PID_ISA)</li> <li>Independent term algorithm (PID_IND)</li> </ul> <p><b>Note:</b> For details, refer to Chapter 7.</p>
	<p>Service Request. Requests a special control system service.</p> <p><b>Note:</b> For details, refer to Chapter 6.</p>

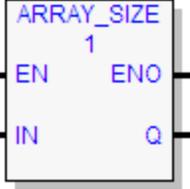
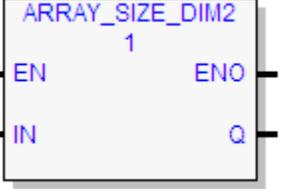
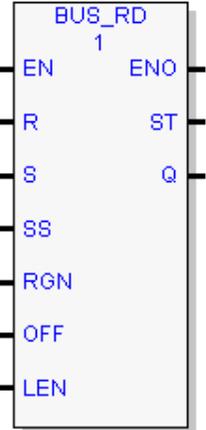
Function	Description
	<p>Scan Set I/O. Scans the IO of a specified scan set. For details, refer to <i>Control Functions</i> in Chapter 4.</p>
	<p>Suspend I/O. Suspends for one sweep all normal I/O updates, except those specified by DO I/O instructions. For details, refer to <i>Control Functions</i> in Chapter 4.</p>
	<p>Suspend I/O Interrupt. Suspend or resume an I/O interrupt when using I/O variables. If not using I/O variables, use <i>SVC_REQ 32: Suspend/Resume I/O Interrupt</i>, described in Chapter 6. For details, refer to <i>Control Functions</i> in Chapter 4.</p>
	<p>Falling Edge Trigger. Detects a high-to-low transition of a Boolean input. Produces a single output pulse when a falling edge is detected. For details, refer to <i>Control Functions</i> in Chapter 4.</p>
	<p>Rising Edge Trigger. Detects a low-to-high transition of a Boolean input. Produces a single output pulse when a rising edge is detected. For details, refer to <i>Control Functions</i> in Chapter 4.</p>

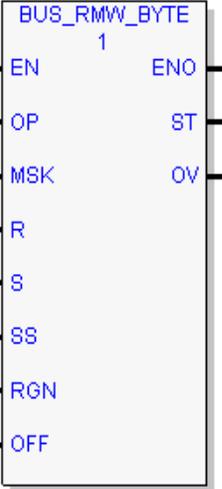
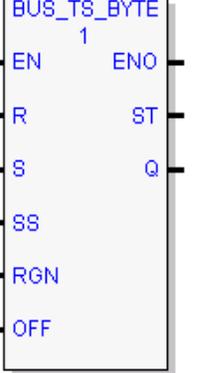
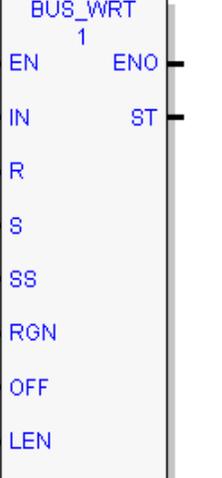
## 5.7 Counters

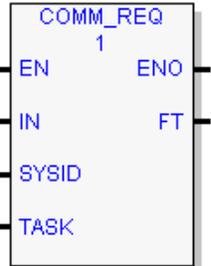
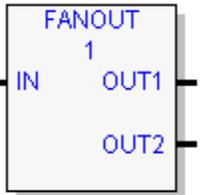
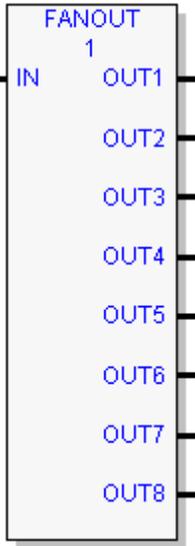
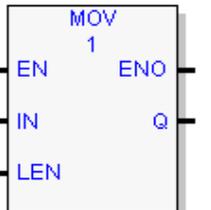
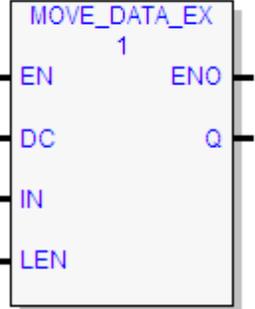
<b>Function</b>	<b>Description</b>
<p><b>control_parameter</b></p> 	<p>Down Counter. Counts down from a preset value. The output is ON whenever the Current Value is <math>\leq 0</math>.</p> <p>The parameter that appears above the function block is a one-dimensional, three-word array in %R, %W, %P, %L, or symbolic memory that the counter uses to store its current value, preset value and control word.</p> <p>For details, refer to <i>Counters</i> in Chapter 4.</p>
<p><b>control_parameter</b></p> 	<p>Up Counter. Counts up to a designated value. The output is ON whenever the Current Value is <math>\geq</math> the Preset Value.</p> <p>The parameter that appears above the function block is a one-dimensional, three-word array in %R, %W, %P, %L, or symbolic memory that the counter uses to store its current value, preset value and control word.</p> <p>For details, refer to <i>Counters</i> in Chapter 4.</p>

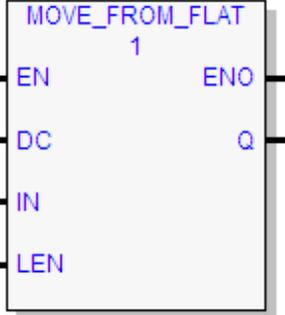
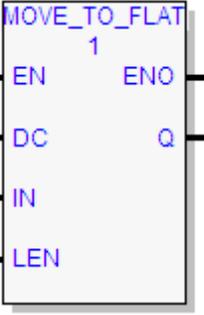
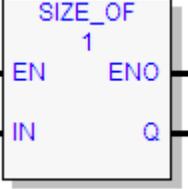
## 5.8 Data Move Functions

The Data Move functions provide basic data move capabilities.

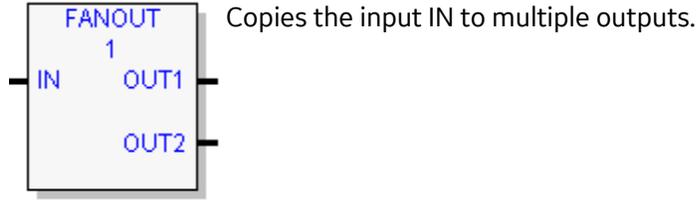
<b>Function</b>	<b>Description</b>
	<p>Array Size. Counts the number of elements in an array. For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>
	<p>Array Size Dim1. Returns the value of the Array Dimension 1 property of an array. For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>
	<p>Array Size Dim2. Returns the value of the Array Dimension 2 property of an array. For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>
	<p>Bus Read. Reads data from the bus. For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>

Function	Description
	<p>Bus Read Modify Write. Uses a read/modify/write cycle to update a data element in a module on the bus.</p> <p>Other BUS_RMW functions:          BUS_RMW_DWORD          BUS_RMW_WORD</p> <p>For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>
	<p>Bus Test and Set. Handles semaphores on the bus.</p> <p>Other BUS_TS function:          BUS_TS_WORD</p> <p>For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>
	<p>Bus Write. Writes data to a module on the bus.</p> <p>For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>

Function	Description	
 <p>COMM_REQ 1 EN ENO IN FT SYSID TASK</p>	<p>Communication Request. Allows the program to communicate with an intelligent module, such as a Genius Bus Controller or a High Speed Counter.</p> <p>For details, refer to <i>Communication Request</i> in Chapter 4.</p>	
 <p>FANOUT 1 IN OUT1 OUT2</p> <p>Minimum Outputs = 2</p>	 <p>FANOUT 1 IN OUT1 OUT2 OUT3 OUT4 OUT5 OUT6 OUT7 OUT8</p> <p>Maximum Outputs = 8</p>	<p><i>Fan Out</i>. Copies the input value to multiple outputs of the same data type as the input.</p> <p>For details, refer to <i>Fan Out</i> below.</p>
 <p>MOV 1 EN ENO IN Q LEN</p>	<p><i>Move Data</i>. Copies data as individual bits, so the new location does not have to be the same data type. Data can be moved into a different data type without prior conversion.</p> <p>For details, refer to <i>Move Data</i> below.</p>	
 <p>MOVE_DATA_EX 1 EN ENO DC Q IN LEN</p>	<p><i>Move Data Explicit</i>. Provides data coherency by locking symbolic memory being written to during the copy operation.</p> <p>For details, refer to <i>Data Move Functions</i> in Chapter 4.</p> <p><b>Note:</b> FBD and ST do not support the constant 0 as a value for the input IN.</p>	

Function	Description
	<p>Move From Flat. Copies reference memory data to a UDT variable or UDT array. Provides the option of locking the symbolic or I/O variable memory area being written to during the copy operation.</p> <p>For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>
	<p>Move to Flat. Copies data from symbolic or I/O variable memory to reference memory. Copies across mismatching data types.</p> <p>For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>
	<p>Size Of. Counts the number of bits used by a variable.</p> <p>For details, refer to <i>Data Move Functions</i> in Chapter 4.</p>

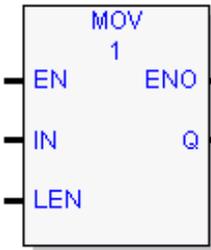
### 5.8.1 Fan Out



#### Operands

<b>Parameter</b>	<b>Description</b>	<b>Allowed Types</b>	<b>Allowed Operands</b>	<b>Optional</b>
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN	The input to copy to the outputs.	BOOL, DINT, DWORD, INT, REAL, UINT, or WORD variable or constant	All except SA, SB, SC.	No
OUT1 ...OUT8	Variables of the same data type as the IN operand. The outputs. Minimum: two outputs. Maximum: eight outputs.	Must be same type as IN.	All except S, SA, SB, SC and constant.	No

### 5.8.2 Move Data



When the input operand, EN, is set to ON, the MOVE instruction copies data as bits from one location in PACSystems controller memory to another. Because the data is copied as bits, the new location does not need to use the same type of memory area as the source. For example, you can copy data from an analog memory area into a discrete memory area, or vice versa.

MOV sets its output, ENO, whenever it receives data unless one of the following occurs:

- When the input, EN, is set to OFF, then the output, ENO, is set to OFF.
- When the input, EN is set to ON, and the input, IN, contains an indirect reference, and the memory of IN is out of range, then the output, ENO, is set to OFF.

The value to store at the destination Q is acquired from the IN parameter. If IN is a variable, the value to store in Q is the value stored at the IN address. If IN is a constant, the value to store in Q is that constant

The result of the MOVE depends on whether the data type for the Q operand is a bit reference or a non-bit reference:

- If Q is a non-bit reference, LEN (the length) indicates the number of memory locations in which the IN value should be repeated, starting at the location specified by Q.
- If Q is a bit reference, IN is treated as an array of bits. LEN therefore indicates the number of bits to acquire from the IN parameter to make up the stored value. If IN is a constant, bits are counted from the least-significant bit. If IN is a variable, LEN indicates the number of bits to acquire starting at the IN location. Regardless, only LEN bits are stored starting at address Q.

For example, if IN was the constant value 29 and LEN is 4, the results of a MOV operation are as follows:

- Q is a WORD reference: The value 29 is repeatedly stored in locations Q, Q+1, Q+2, and Q+3.
- Q is a BOOL reference: The binary representation of 29 is 11101. Since LEN is 4, only the four least-significant bits are used (1101). This value is stored at location Q in the same order, so 1 is stored in Q, 1 is stored in Q+1, 0 is stored in Q+2, and 1 is stored in Q+3.

If data is moved from one location in discrete memory to another, such as from %I memory to %T memory, the transition information associated with the discrete memory elements is updated to indicate whether or not the MOVE operation caused any discrete memory elements to change state.

**Note:** If an array of BOOL-type data specified in the Q operand does not include all the bits in a byte, the transition bits associated with that byte (which are not in the array) are cleared when the Move instruction receives data.

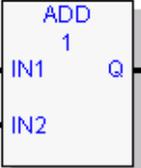
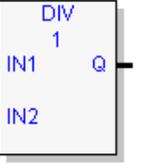
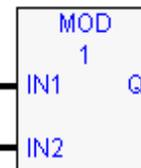
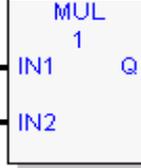
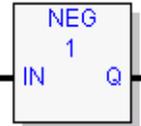
Data at the IN operand does not change unless there is an overlap in the source and destination—a situation that is to be avoided.

## MOV Operands

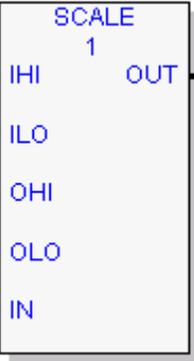
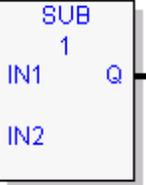
Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
EN	Enable	BOOL variable	data flow, I, Q, M, T, G, S, SA, SB, SC, discrete symbolic, I/O variable	No
		Bit reference in a non-BOOL variable	R, P, L, AI, AQ, W, non-discrete symbolic, I/O variable	
IN	<p>The source of the data to copy into the output Q. This can be either a constant or a variable whose reference address is the location of the first source data item to move.</p> <p>IN must have the same data type as the variable in the Q parameter.</p> <p>If IN is a BOOL variable or a bit reference, an %, %Q, %M, or %T reference address need not be byte-aligned, but 16 bits beginning with the reference address specified are displayed online.</p>	DINT, DWORD, INT, REAL, LREAL, UINT, WORD, or bit reference in a non-BOOL variable	All. S, SA, SB, SC allowed only for WORD, DWORD, BOOL types.	No
LEN	<p>The length of IN; the number of bits to move.</p> <p>If IN is a constant and Q is BOOL: 1 ≤ LEN ≤ 16;</p> <p>If IN is a constant and Q is <i>not</i> BOOL: 1 ≤ LEN ≤ 256.</p> <p>All other cases: 1 ≤ LEN ≤ 32,767</p> <p>LEN is also interpreted differently depending on the data type of the Q location. For details, see discussion under <i>Move Data</i>.</p>	Constant	Constant	No
ENO	<p>Indicates whether the operation was successfully completed.</p> <p>If ENO = ON (1), the operation was initiated. Results of the operation are indicated in the FT output.</p> <p>If ENO = OFF (0), the operation was not performed. If EN was ON, the FT output indicates an error condition. If EN was OFF, FT is not changed.</p>	BOOL variable	data flow, I, Q, M, T, G, discrete symbolic, I/O variable	Yes
		Bit reference in a non-BOOL variable	I, Q, M, T, G, R, P, L, AI, AQ, W, non-discrete symbolic, I/O variable	
Q	<p>The location of the first destination data item. Q must have the same data type as the variable in the IN parameter.</p> <p>If Q is a BOOL variable or a bit reference, an %, %Q, %M, or %T reference address does not need to be byte-aligned, but 16 bits beginning with the specified reference address are displayed online.</p>	DINT, DWORD, INT, REAL, LREAL, UINT, WORD, or bit reference in a non-BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, R, P, L, AI, AQ, W, symbolic, I/O variable	No

## 5.9 Math Functions

Your program may need to include logic to convert data to a different type before using a Math or Numerical function. The description of each function includes information about appropriate data types. The *Type Conversion Functions* section explains how to convert one data type into a different data type.

Function	Description
	<p>Addition. Adds two or up to eight numbers. For details, refer to <i>Add</i> below.</p>
	<p>Division.<sup>5</sup> Divides one number by another and outputs the quotient. <b>Note:</b> Take care to avoid overflow conditions when performing divisions. For details, refer to <i>Divide</i> below.</p>
	<p>Modulo Division. Divides one number by another and outputs the remainder. For details, refer to <i>Modulus</i> below.</p>
	<p>Multiplication.<sup>5</sup> Multiplies two or up to eight numbers. <b>Note:</b> Take care to avoid overflow conditions when performing multiplications. For details, refer to <i>Multiply</i> below.</p>
	<p>Negate. Multiplies a number by -1 and places the result in an output location. For details, refer to <i>Negate</i> below.</p>

<sup>5</sup> To avoid *Overflows* when multiplying or dividing 16-bit numbers, use the *Type Conversion Functions* to convert the numbers to a 32-bit data type.

Function	Description
	<p>Scales an input parameter and places the result in an output location. For details, refer to <i>Math Functions</i> in Chapter 4.</p>
	<p>Subtraction. Subtracts one or up to seven numbers from the input IN1 and places the result in an output location. For details, refer to <i>Subtract</i> below.</p>

The output is calculated when the instruction is performed without *Overflow*, unless an invalid operation occurs.

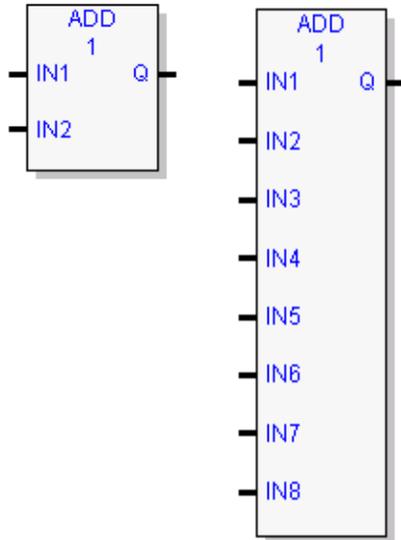
### 5.9.1 Overflow

If an operation on integer operands results in overflow, the output value wraps around.

Examples:

- If the ADD operation,  $32767 + 1$ , is performed on signed integer operands, the result is  $-32768$
- If the SUB operation,  $-32767 - 1$ , is performed on signed integer operands, the result is  $32767$
- If an ADD\_UINT operation is performed on  $65535 + 16$ , the result is  $15$ .

### 5.9.2 Add



Adds the operands IN1 and IN2 ... IN8 and stores the sum in Q. IN1 ... IN8 and Q must be of the same data type.

The result is output to Q when ADD is performed without *Overflow*, unless one of the following invalid conditions occurs:

- (+ ∞)
- IN1 and/or IN2 ... IN8 is NaN (Not a Number).

If an ADD operation results in *Overflow*, the result wraps around. For example:

- If an ADD\_DINT, ADD\_INT or ADD\_REAL operation is performed on 32767 + 1, Q will be set to -32768.
- If an ADD\_UINT operation is performed on 65535 + 16, Q will be set to 15.

Minimum number of inputs = 2  
Maximum number of inputs = 8.

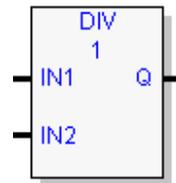
#### Operands of the ADD Function

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN1 ... IN8	The values to be added.	INT, DINT, REAL, LREAL, UINT Must be same data type as Q.	All except S, SA, SB, SC and data flow	No
Q	The sum of IN1 ... IN8. If an <i>Overflow</i> occurs, Q wraps around.	INT, DINT, REAL, LREAL, UINT variable Must be same data type as IN1 .... IN8.	All except S, SA, SB, SC, constant and data flow	No

#### Properties for ADD

Property	Valid Range
Number of Inputs	2 to 8

### 5.9.3 Divide



Divides the operand IN1 by the operand IN2 of the same data type as IN1 and stores the quotient in the output variable assigned to Q, also of the same data type as IN1 and IN2.

The result is output to Q when DIV is performed without *Overflow*, unless one of the following invalid conditions occurs:

- 0 divided by 0 (Results in an application fault.)
- IN1 and/or IN2 is NaN (Not a Number).

If an *Overflow* occurs, the result wraps around.

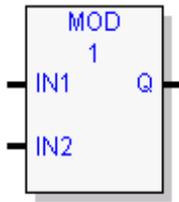
**Notes:**

- DIV rounds down; it does not round to the closest integer. For example, 24 DIV 5 = 4.
- Be careful to avoid overflows.

#### Operands for DIV\_UINT, DIV\_INT, DIV\_DINT, and DIV\_REAL

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN1	Dividend: the value to be divided; shown to the left of <i>DIV</i> in the equation IN1 DIV IN2=Q.	INT, DINT, UINT, REAL, LREAL	All except S, SA, SB, SC	No
IN2	Divisor: the value to divide into IN1; shown to the right of <i>DIV</i> in the equation IN1 DIV IN2=Q.	INT, DINT, UINT, REAL, LREAL	All except S, SA, SB, SC	No
Q	The quotient of IN1/IN2. If an <i>Overflow</i> occurs, the result is the largest value with the proper sign.	INT, DINT, UINT, REAL or LREAL variable	All except S, SA, SB, SC and constant	No

### 5.9.4 Modulus



Divides input IN1 by input IN2 and outputs the remainder of the division to Q.

All three operands must be of the same data type. The sign of the result is always the same as the sign of input parameter IN1. Output Q is calculated using the formula:

$$Q = IN1 - ((IN1 \text{ DIV } IN2) * IN2)$$

where DIV produces an integer number.

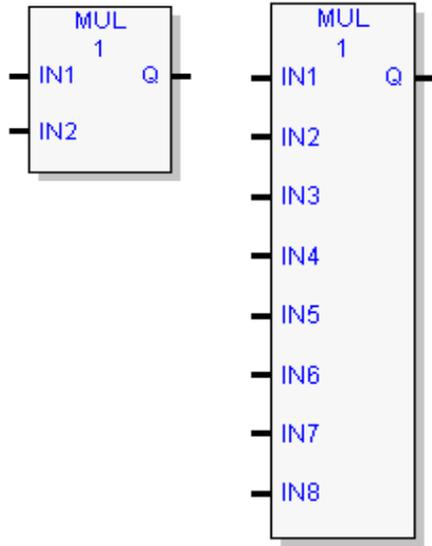
The result is output to Q unless one of the following invalid conditions occurs:

- 0 divided by 0 (Results in an application fault.)
- IN1 and/or IN2 is NaN (Not a Number)

#### Operands for Modulus Function

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN1	Dividend: the value to be divided into in order to obtain the remainder; shown to the left of <i>MOD</i> in the equation $IN1 \text{ MOD } IN2=Q$ .	INT, DINT, UINT	All except S, SA, SB, SC	No
IN2	Divisor: the value to divide into IN1; shown to the right of <i>MOD</i> in the equation $IN1 \text{ MOD } IN2=Q$ .	INT, DINT, UINT	All except S, SA, SB, SC	No
Q	The remainder of $IN1/IN2$ .	INT, DINT, UINT variable	All except S, SA, SB, SC and constant	No

### 5.9.5 Multiply



Multiplies two through eight operands (IN1 ... IN8) of the same data type and stores the result in the output variable assigned to Q, also of the same data type.

The output is calculated when the function is performed without *Overflow*, unless an invalid operation occurs.

If an *Overflow* occurs, the result wraps around.

Minimum number of inputs = 2      Maximum number of inputs = 8.

Mnemonic	Operation	Displays as
INT	$Q(16\text{-bit}) = IN1(16\text{-bit}) * IN2(16\text{-bit})$	base 10 number with sign, up to 5 digits long
DINT	$Q(32\text{-bit}) = IN1(32\text{-bit}) * IN2(32\text{-bit})$	base 10 number with sign, up to 10 digits long
REAL	$Q(32\text{-bit}) = IN1(32\text{-bit}) * IN2(32\text{-bit})$	base 10 number, sign and decimals, up to 8 digits long (excluding the decimals)
UINT	$Q(16\text{-bit}) = IN1(16\text{-bit}) * IN2(16\text{-bit})$	base 10 number, unsigned, up to 5 digits long

#### Operands for Multiply

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN1 ... IN8	The values to multiply. Must be the same data type as Q.	INT, DINT, UINT, REAL	All except S, SA, SB, SC	No
Q	The result of the multiplication.	INT, DINT, UINT, REAL variable	All except S, SA, SB, SC and constant	No

#### Properties for Multiply

Property	Valid Range
Number of Inputs	2 to 8

### 5.9.6 Negate

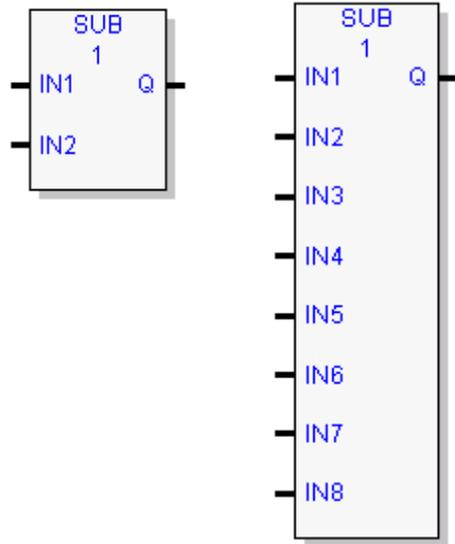


Multiplies a number by  $-1$  and places the result in the output location, Q.

#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN	The value to be negated.	INT, DINT, REAL	All except S, SA, SB, SC	No
Q	The result, $-1(IN)$	INT, DINT, REAL variable	All except S, SA, SB, SC and constant	No

### 5.9.7 Subtract



Subtracts the operands IN2 ...IN8 from the operand IN1 and stores the result in the output variable assigned to Q.

The calculation is carried out when SUB is performed without *Overflow*, unless an invalid operation occurs.

If a SUB operation results in *Overflow*, the result wraps around. For example:

- If a SUB\_DINT, SUB\_INT or SUB\_REAL operation is performed on 32768 - 1, Q will be set to -32767.

If a SUB\_UINT operation results in a negative number, Q wraps around. (For example, a result of -1 sets Q to 65535.)

Minimum number of inputs = 2      Maximum number of inputs = 8.

Mnemonic	Operation	Displays as
SUB_INT	$Q(16\text{-bit}) = IN1(16\text{-bit}) - IN2(16\text{-bit})$	base 10 number with sign, up to 5 digits long
SUB_DINT	$Q(32\text{-bit}) = IN1(32\text{-bit}) - IN2(32\text{-bit})$	base 10 number with sign, up to 10 digits long
SUB_REAL	$Q(32\text{-bit}) = IN1(32\text{-bit}) - IN2(32\text{-bit})$	base 10 number, sign and decimals, up to 8 digits long (excluding the decimals)
SUB_UINT	$Q(16\text{-bit}) = IN1(16\text{-bit}) - IN2(16\text{-bit})$	base 10 number, unsigned, up to 5 digits long

#### Operands for Subtract

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN1	The value to subtract from.	DINT, INT, REAL, UINT	All except S, SA, SB, SC	No
IN2 ... IN8	The value(s) to subtract from IN1. Must be the same data type as IN1.		All except S, SA, SB, SC	No
Q	The result of the subtraction. Must be the same data type as IN1.	DINT, INT, REAL, UINT variable	All except S, SA, SB, SC and constant	No

#### Properties for Subtract

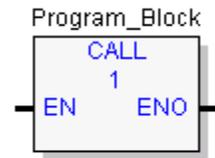
Property	Valid Range
Number of Inputs	2 to 8



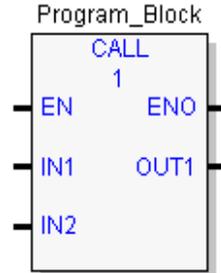
## 5.10 Program Flow Functions

The program flow functions limit program execution or change the way the CPU executes the application program.

### Function



Non-parameterized  
CALL

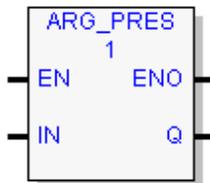


Parameterized CALL.  
May call a  
parameterized external  
block or a  
parameterized block.

### Description

The CALL function causes the logic execution to go immediately to the designated program block, external C block (parameterized or not), or parameterized block and execute it. After the block's execution is complete, control returns to the point in the logic immediately following the CALL instruction.

For details, refer to *Program Flow Functions* in Chapter 4.



The ARG\_PRES (Argument Present) function determines whether a parameter value was present when the function block instance of the parameter was invoked.

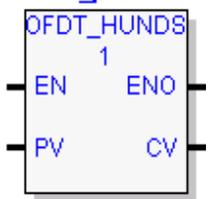
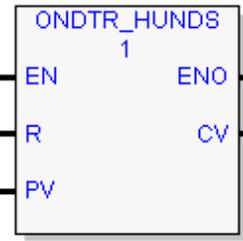
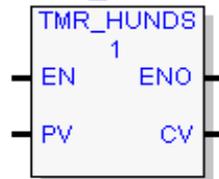
For details, refer to *Program Flow Functions* in Chapter 4.

## 5.11 Timers

This section describes the PACSystems timing functions that are implemented in the FBD language.

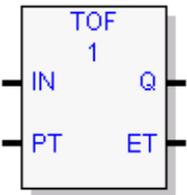
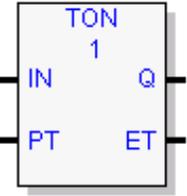
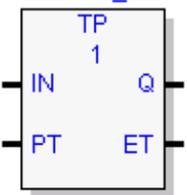
### 5.11.1 Built-in Timer Function Blocks

These function blocks use WORD Array instance data. The parameter that appears above the function block is a one-dimensional, three-word array in %R, %W, %P, %L, or symbolic memory that the timer uses to store its current value, preset value and control word.

Function	Description
<p><b>control_parameter</b></p> 	<p>Off Delay Timer. The timer's Current Value (CV) resets to zero when its enable parameter (EN) is set to ON.. CV increments while EN is OFF. When CV=PV (Preset Value), ENO is set to OFF until EN is set to ON again.</p> <p>Other OFDT functions:            OFDT_SEC            OFDT_TENTHS            OFDT_THOUS</p> <p>For details, refer to <i>Timers</i> in Chapter 4.</p>
<p><b>control_parameter</b></p> 	<p>On Delay Stopwatch Timer. Retentive on delay timer. Increments while EN is ON and holds its value when EN is OFF.</p> <p>ONDTR_SEC            ONDTR_TENTHS            ONDTR_THOUS</p> <p>For details, refer to <i>Timers</i> in Chapter 4.</p>
<p><b>control_parameter</b></p> 	<p>On Delay Timer. Simple on delay timer. Increments while EN is ON and resets to zero when EN is OFF.</p> <p>TMR_SEC            TMR_TENTHS            TMR_THOUS</p> <p>For details, refer to <i>Timers</i> in Chapter 4.</p>

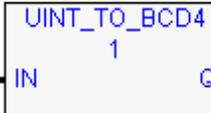
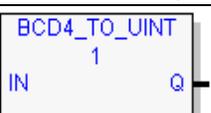
### 5.11.2 Standard Timer Function Blocks

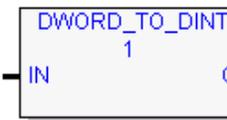
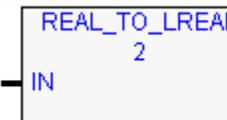
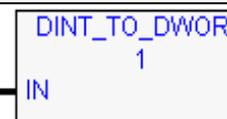
These functions blocks use Structure Variable instance data. Each invocation of a timer has associated instance data that persists from one execution of the timer to the next. Instance variables are automatically located in symbolic memory. (You cannot specify an address.) You can specify a stored value for each element. The user logic cannot modify the values.

Function	Description
<p><i>Instance_Var</i></p>  <p>The diagram shows a rectangular function block for TOF. At the top, it is labeled 'TOF' with an instance number '1' below it. On the left side, there are two input terminals: 'IN' and 'PT'. On the right side, there are two output terminals: 'Q' and 'ET'.</p>	<p>Timer Off Delay. When the input IN transitions from ON to OFF, the timer starts timing until a specified period of time has elapsed, then sets the output Q to OFF.</p> <p>For details, refer to <i>Timers</i> in Chapter 4.</p>
<p><i>Instance_Var</i></p>  <p>The diagram shows a rectangular function block for TON. At the top, it is labeled 'TON' with an instance number '1' below it. On the left side, there are two input terminals: 'IN' and 'PT'. On the right side, there are two output terminals: 'Q' and 'ET'.</p>	<p>Timer On Delay. When the input IN transitions from OFF to ON, the timer starts timing until a specified period of time has elapsed, then sets the output Q to ON.</p> <p>For details, refer to <i>Timers</i> in Chapter 4.</p>
<p><i>Instance_Var</i></p>  <p>The diagram shows a rectangular function block for TP. At the top, it is labeled 'TP' with an instance number '1' below it. On the left side, there are two input terminals: 'IN' and 'PT'. On the right side, there are two output terminals: 'Q' and 'ET'.</p>	<p>Timer Pulse. When the input IN transitions from OFF to ON, the timer sets the output Q to ON for a specified time interval.</p> <p>For details, refer to <i>Timers</i> in Chapter 4.</p>

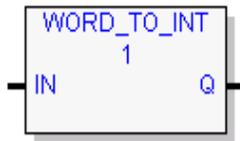
## 5.12 Type Conversion Functions

The Conversion functions change a data item from one number format (data type) to another. Many programming instructions, such as math functions, must be used with data of one type. As a result, data conversion is often required before using those instructions.

Function	Description
<b>Convert Angles</b>	
	DEG_TO_RAD: Converts degrees to radians. RAD_TO_DEG: Converts radians to degrees. For details, refer to <i>Conversion Functions</i> in Chapter 4.
<b>Convert to BCD4 (4-digit Binary-Coded-Decimal)</b>	
	UINT_TO_BCD4: Converts UINT (16-bit unsigned integer) to BCD4. INT_TO_BCD4: Converts INT (16-bit signed integer) to BCD4. For details, refer to <i>Conversion Functions</i> in Chapter 4.
<b>Convert to BCD8 (8-digit Binary-Coded-Decimal)</b>	
	DINT_TO_BD8: Converts DINT (32-bit signed integer) to BCD8. For details, refer to <i>Conversion Functions</i> in Chapter 4.
<b>Convert to INT (16-bit signed integer)</b>	
	BCD4_TO_INT: Converts BCD to INT. UINT_TO_INT: Converts UINT to INT DINT_TO_INT: Converts DINT to INT.. REAL_TO_INT: Converts REAL to INT. For details, refer to <i>Conversion Functions</i> in Chapter 4.
	Converts a 16-bit string (WORD) value to INT. For details, refer to <i>Convert WORD to INT</i> below.
<b>Convert to UINT (16-bit unsigned integer)</b>	
	BCD4_TO_UINT: Converts BCD4 to UINT. INT_TO_UINT: Converts INT to UINT. DINT_TO_UINT: Converts DINT to UINT. REAL_TO_UINT: Converts REAL to UINT. For details, refer to <i>Conversion Functions</i> in Chapter 4.
	WORD_TO_UINT: Converts a 16-bit string (WORD) value to UINT. For details, refer to <i>Convert DWORD to DINT</i> below.
<b>Convert to DINT (32-bit signed integer)</b>	
	BCD8_TO_DINT: Converts BCD8 to DINT. UINT_TO_DINT: Converts UINT to DINT. For details, refer to <i>Conversion Functions</i> in Chapter 4.

Function	Description
	<p>INT_TO_DINT: Converts INT to DINT.                      REAL_TO_DINT: Converts REAL (32-bit signed real or floating-point values) to DINT.                      For details, refer to <i>Conversion Functions</i> in Chapter 4.</p>
	<p>DWORD_TO_DINT: Converts a 32-bit bit string (DWORD) value to DINT.                      For details, refer to <i>Convert DWORD to DINT</i> below.</p>
<p><b>Convert to REAL (32-bit signed real or floating-point values)</b></p>	
	<p>BCD4_TO_REAL: Converts BCD4 to REAL.                      BCD8_TO_REAL: Converts BCD8 to REAL.                      UINT_TO_REAL: Converts UINT to REAL.                      INT_TO_REAL: Converts INT to REAL.                      DINT_TO_REAL: Converts DINT to REAL.                      LREAL_TO_REAL: Converts LREAL to REAL.                      For details, refer to <i>Conversion Functions</i> in Chapter 4.</p>
<p><b>Convert to LREAL(64-bit signed real or floating-point values)</b></p>	
	<p>Converts a REAL value to LREAL.                      For details, refer to <i>Conversion Functions</i> in Chapter 4.</p>
<p><b>Convert to WORD (16-bit string)</b></p>	
	<p>Converts an INT (16-bit signed integer) value to a WORD value.                      For details, refer to <i>Convert INT or UINT to WORD</i> below.</p>
	<p>Converts an unsigned single-precision integer (UINT) to WORD.                      For details, refer to <i>Convert INT or UINT to WORD</i> below.</p>
<p><b>Convert to DWORD (32-bit bit string)</b></p>	
	<p>Converts a double-precision signed integer (DINT) value to DWORD.                      For details, refer to <i>Convert DINT to DWORD</i> below.</p>
<p><b>Truncate</b></p>	
	<p>Rounds a REAL (32-bit signed real or floating-point) number down to a DINT number                      For details, refer to <i>Conversion Functions</i> in Chapter 4.</p>
	<p>Rounds a REAL (32-bit signed real or floating-point) number down to an INT number                      For details, refer to <i>Conversion Functions</i> in Chapter 4.</p>

### 5.12.1 Convert WORD to INT



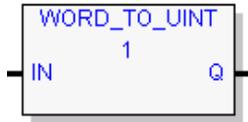
Converts the input data into the equivalent single-precision signed integer (INT) value, which it outputs to Q. This function does not change the original input data. The output data can be used directly as input for another program function, as in the examples.

The function passes data to Q, unless the data is out of range (0 through +65,535).

#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN	The value to convert to INT.	WORD	All except S, SA, SB, and SC	No
Q	The INT equivalent value of the original value in IN.	INT	All except S, SA, SB, SC and constant	No

### 5.12.2 Convert WORD to UINT



These functions convert the input data into the equivalent single-precision unsigned integer (UINT) value, which it outputs to Q.

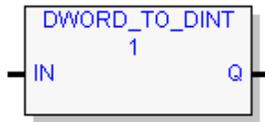
The conversion to UINT does not change the original data. The output data can be used directly as input for another program function, as in the example.

The function passes the converted data to Q, unless the resulting data is outside the range 0 to +65,535.

#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN	The value to convert to UINT.	WORD	All except S, SA, SB, and SC	No
Q	The UINT equivalent value of the original input value in IN.	UINT	All except S, SA, SB, SC and constant	No

### 5.12.3 Convert DWORD to DINT



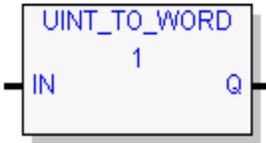
Converts DWORD data into the equivalent signed double-precision integer (DINT) value and stores the result in Q. The conversion to DINT does not change the original data.

The output data can be used directly as input for another program function. The function passes data to Q unless the data is out of range.

#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN	The value to convert to DINT.	DWORD	All except S, SA, SB, and SC	No
Q	The DINT equivalent value of the original input value in IN.	UINT	All except S, SA, SB, SC and constant	No

### 5.12.4 Convert INT or UINT to WORD



Converts an unsigned single-precision integer (UINT) operand IN to a 16-bit bit string (WORD) value and stores the result in the variable assigned to Q.



Converts a 16-bit signed integer (INT) operand IN to a 16-bit bit string (WORD) value and stores the result in the variable assigned to Q.

The output data can be used directly as input for another program function. The function passes data to Q unless the data is out of range.

#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN	The value to convert to WORD.	INT or UINT, depending on function	All except S, SA, SB, and SC	No
Q	The WORD equivalent value of the original value in IN. $0 \leq Q \leq 65,535$ .	WORD	All except S, SA, SB, SC and constant	No

### 5.12.5 Convert DINT to DWORD



When DINT\_TO\_DWORD receives data, it converts the input double-precision signed integer (DINT) data into the equivalent DWORD (32-bit bit string) value, which it outputs to Q. DINT\_TO\_DWORD does not change the original DINT data.

The output data can be used directly as input for another program function. The function passes data to Q unless the data is out of range.

#### Operands

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
IN	The value to convert to DWORD.	DINT	All except S, SA, SB, and SC	No
Q	The DWORD equivalent value of the original value in IN. $0 \leq Q \leq 4,294,967,295$ .	DWORD	All except S, SA, SB, SC and constant	No



## Chapter 6 Service Request Function

---

Use a Service Request function to request one of the following control system services:

- SVC\_REQ 1: Change/Read Constant Sweep Timer*
- SVC\_REQ 2: Read Window Modes and Time Values*
- SVC\_REQ 3: Change Controller Communications Window Mode*
- SVC\_REQ 4: Change Backplane Communications Window Mode and Timer Value*
- SVC\_REQ 5: Change Background Task Window Mode and Timer Value*
- SVC\_REQ 6: Change/Read Number of Words to Checksum*
- SVC\_REQ 7: Read or Change the Time-of-Day Clock*
- SVC\_REQ 8: Reset Watchdog Timer*
- SVC\_REQ 9: Read Sweep Time from Beginning of Sweep*
- SVC\_REQ 10: Read Target Name*
- SVC\_REQ 11: Read Controller ID*
- SVC\_REQ 12: Read Controller Run State*
- SVC\_REQ 13: Shut Down (STOP) CPU*
- SVC\_REQ 14: Clear Controller or I/O Fault Table*
- SVC\_REQ 15: Read Last-Logged Fault Table Entry*
- SVC\_REQ 16: Read Elapsed Time Clock*
- SVC\_REQ 17: Mask/Unmask I/O Interrupt*
- SVC\_REQ 18: Read I/O Forced Status*
- SVC\_REQ 19: Set Run Enable/Disable*
- SVC\_REQ 20: Read Fault Tables*
- SVC\_REQ 21: User-Defined Fault Logging*
- SVC\_REQ 22: Mask/Unmask Timed Interrupts*
- SVC\_REQ 23: Read Master Checksum*
- SVC\_REQ 24: Reset Module*
- SVC\_REQ 25: Disable/Enable EXE Block and Standalone C Program Checksums*
- SVC\_REQ 29: Read Elapsed Power Down Time*
- SVC\_REQ 32: Suspend/Resume I/O Interrupt*
- SVC\_REQ 45: Skip Next I/O Scan*
- SVC\_REQ 50: Read Elapsed Time Clock*
- SVC\_REQ 51: Read Sweep Time from Beginning of Sweep*
- SVC\_REQ 56: Logic Driven Read of Nonvolatile Storage*
- SVC\_REQ 57: Logic Driven Write to Nonvolatile Storage*

The following Service Requests are used in CPU HSB redundancy applications.

Refer to the *PACSystems Hot Standby CPU Redundancy User's Guide*, GFK-2308. For non-HSB applications, refer to *PACSystems RX7i, RX3i and RSTi-EP TCP/IP Ethernet Communications User Manual*, GFK-2224.

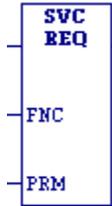
- SVC\_REQ 26* Role switch (redundancy)
- SVC\_REQ 27* Write to reverse transfer area (Hot Standby Redundancy)
- SVC\_REQ 28* Read from reverse transfer area (Hot Standby Redundancy)
- SVC\_REQ 43* Disable data transfer copy in backup unit (Hot Standby Redundancy)

SVC\_REQ 55 Set application redundancy mode (non-Hot Standby Redundancy)

## 6.1 Operation of SVC\_REQ Function

PACSystems supports the Service Request function in LD and FBD.

### 6.1.1 Ladder Diagram



When SVC\_REQ receives power flow, it requests the CPU to perform the special service identified by the FNC operand.

Parameters for SVC\_REQ are located in the parameter block, which begins at the reference identified by the PRM operand. The number of 16-bit references required depends on the type of special controller service being requested. The parameter block is used to store both the function's inputs and outputs.

SVC\_REQ passes power flow unless an incorrect function number, incorrect parameters, or out-of-range references are specified. Specific SVC\_REQ functions may have additional causes for failure. Because the service request continues to be invoked each time power flow is enabled to the function, additional enable/disable logic preceding the request may be necessary, depending upon the application. (For example, repeated calling of SVC\_REQ 24 would continually reset a module, probably not the intended behavior.) In many cases a transition contact or coil will be sufficient. Alternatively, you could use more complex logic, such as having the function contained within a block that is only called a single time.

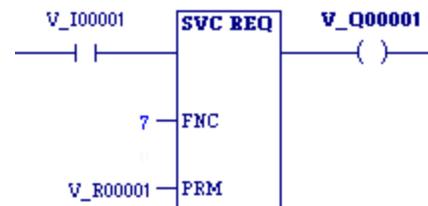
### Operands

**Note:** Indirect referencing is available for all register references (%R, %P, %L, %W, %AI, and %AQ).

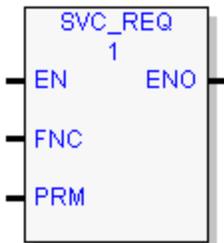
Operand	Data Type	Memory Area	Description
FNC	INT variable or constant	All except %S - %SC	Function number; Service Request number. The constant or reference that identifies the requested service.
PRM	WORD variable	All except flow, %S - %SC and constant	The first WORD in the parameter block for the requested service. Successive 16-bit locations store additional parameters.

### Example

When the enabling input %I0001 is ON, SVC\_REQ function number 7 is called, with the parameter block starting at %R0001. If the operation succeeds, output coil %Q0001 is set ON.



### 6.1.2 Function Block Diagram



The SVC\_REQ function requests the CPU to perform the special service identified by the FNC operand.

Parameters for SVC\_REQ are located in the parameter block, which begins at the reference identified by the PRM operand. The number of 16-bit references required depends on the type of special controller service being requested. The parameter block is used to store both the function's inputs and outputs.

#### Operands

**Note:** Indirect referencing is available for all register references (%R, %P, %L, %W, %AI, and %AQ).

Parameter	Description	Allowed Types	Allowed Operands	Optional
Solve Order	Calculated by the FBD editor.	NA	NA	No
EN	Enable input. When set to ON, the SVC_REQ executes	BOOL Bit reference in a non-BOOL variable	data flow, I, Q, M, T, G, S, SA, SB, SC, discrete symbolic, I/O variable I, Q, M, T, G, R, P, L, AI, AQ, W, non-discrete symbolic, I/O variable	No
FNC	Function number; Service Request number. The constant or variable that identifies the requested service.	INT, DINT, UINT, WORD, DWORD	All except %S - %SC You can use data flow only if the parameter block requires only one WORD If you use a symbolic variable or an I/O variable, ensure that its Array Dimension 1 property is set to a value large enough to contain the entire parameter block.	No
PRM	The first word in the parameter block for the requested service. Successive 16-bit locations store additional parameters.	INT, DINT, UINT, WORD, DWORD	All except flow, %S - %SC and constant	No
ENO	Set to ON unless an incorrect function number, incorrect parameters, or out-of-range references are specified. Specific SVC_REQ functions may have additional causes for failure.	BOOL Bit reference in a non-BOOL variable.	data flow, I, Q, M, T, G, non-discrete symbolic, I/O variable I, Q, M, T, G, R, P, L, AI, AQ, W, non-discrete symbolic, I/O variable	Yes

## 6.2 SVC\_REQ 1: Change/Read Constant Sweep Timer

Use SVC\_REQ function 1 to:

- Disable Constant Sweep mode
- Enable Constant Sweep mode and use the old Constant Sweep timer value
- Enable Constant Sweep mode and use a new Constant Sweep timer value
- Set a new Constant Sweep timer value only
- Read Constant Sweep mode state and timer value.

The parameter block has a length of two words used for both input and output.

SVC\_REQ executes successfully unless:

- A number other than 0, 1, 2, or 3 is entered as the requested operation:
- The scan time value is greater than 2550ms (2.55 seconds)
- Constant sweep time is enabled with no timer value programmed or with an old value of 0 for the timer.

### 6.2.1 To disable Constant Sweep mode:

Enter SVC\_REQ 1 with this parameter block:

<b>Address</b>	0
<b>Address + 1</b>	Ignored

### 6.2.2 To enable Constant Sweep mode and use the old timer value:

Enter SVC\_REQ 1 with this parameter block:

<b>Address</b>	1
<b>Address + 1</b>	0

If the timer value does not already exist, entering 0 causes the function to set the OK output to OFF.

### 6.2.3 To enable Constant Sweep mode and use a new timer value:

Enter SVC\_REQ 1 with this parameter block:

<b>Address</b>	1
<b>Address + 1</b>	New timer value <b>Note:</b> If the timer value does not already exist, entering 0 causes the function to set the OK output to OFF.

### 6.2.4 To change the timer value without changing the selection for sweep mode state:

Enter SVC\_REQ 1 with this parameter block:

<b>Address</b>	2
<b>Address + 1</b>	New timer value

### 6.2.5 To read the current timer state and value without changing either:

Enter SVC\_REQ 1 with this parameter block:

<b>Address</b>	3
<b>Address + 1</b>	ignored

#### Output

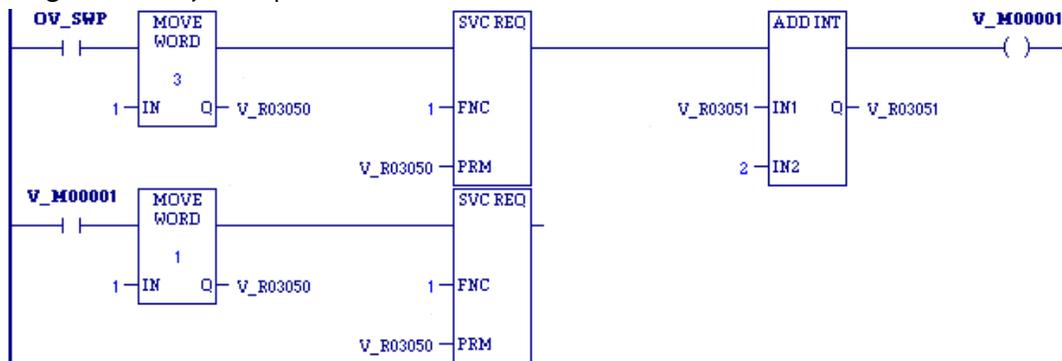
SVC\_REQ 1 returns the timer state and value in the same parameter block references:

<b>Address</b>	0 = Normal Sweep 1 = Constant Sweep
<b>Address + 1</b>	Current timer value

If the word address + 1 contains the hexadecimal value FFFF, no timer value has been programmed.

#### SVC\_REQ 1 Example

If contact OV\_SWP is set, the Constant Sweep Timer is read, the timer is increased by 2 ms, and the new timer value is sent back to the CPU. The parameter block is at location %R3050. The example logic uses discrete internal coil %M0001 as a temporary location to hold the successful result of the first rung line. On any sweep in which OV\_SWP is not set, %M0001 is turned off.



### 6.3 SVC\_REQ 2: Read Window Modes and Time Values

Use SVC\_REQ 2 to obtain the current window mode and time values for the controller communications window and the backplane communications and the background task window.

The parameter block has a length of three words. All parameters are output parameters. It is not necessary to enter values in the parameter block to program this function.

#### Output

Address	Window	High Byte	Low Byte
Address	Controller Communications Window	Mode	Value in ms
Address + 1	Backplane Communications Window	Mode	Value in ms
Address + 2	Background Window	Mode	Value in ms

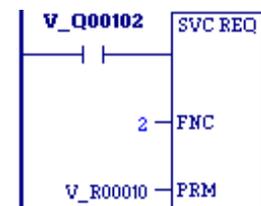
**Note:** A window is disabled when the time value is zero.

#### Mode Values

Mode Name	Value	Description
<b>Limited Mode</b>	0	The execution time of the window is limited to its respective default value or to a value defined using SVC_REQ 3 for the controller communications window or SVC_REQ 4 for the systems communications window. The window will terminate when it has no more tasks to complete.
<b>Constant Mode</b>	1	Each window will operate in a Run to Completion mode, and the CPU will alternate among the three windows for a time equal to the sum of each window's respective time value. If one window is placed in Constant mode, the remaining two windows are automatically placed in Constant mode. If the CPU is operating in Constant Window mode and a particular window's execution time is not defined using the associated SVC_REQ function, the default time for that window is used in the constant window time calculation.
<b>Run to Completion Mode</b>	2	Regardless of the window time associated with a particular window, whether default or defined using a service request function, the window will run until all tasks within that window are completed.

#### SVC\_REQ 2 Example

When %Q00102 is set, the CPU places the current time values of the windows in the parameter block starting at location %R0010.



## 6.4 SVC\_REQ 3: Change Controller Communications Window Mode

Use SVC\_REQ 3 to change the controller communications window mode and timer value. The change takes place during the next CPU sweep after the function is called.

The parameter block has a length of one word.

SVC\_REQ 3 executes unless a mode other than 0 (Limited) or 2 (Run to Completion) is selected.

### 6.4.1 To disable the controller communications window:

Enter SVC\_REQ 3 with this parameter block:

Address	High Byte	Low Byte
Address	0	0

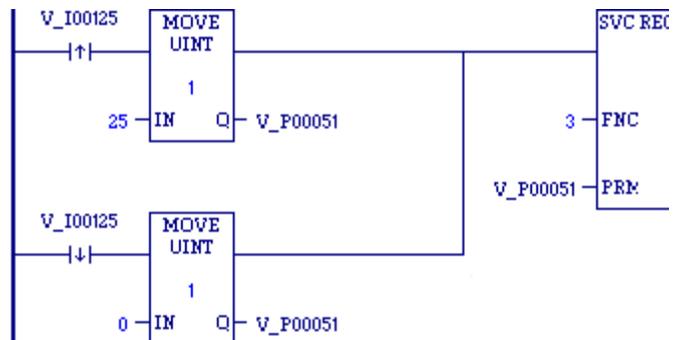
### 6.4.2 To re-enable or change the controller communications window mode:

Enter SVC\_REQ 3 with this parameter block:

Address	High Byte	Low Byte
Address	Mode: 0 = Limited 2 = Run to Completion	1ms ≤ value ≤ 255ms in 1ms increments

### SVC\_REQ 3 Example

When enabling input %I00125 transitions on, the controller communications window is enabled and assigned a value of 25ms. When the contact transitions off, the window is disabled. The parameter block is in global memory location %P00051.



## 6.5 SVC\_REQ 4: Change Backplane Communications Window Mode and Timer Value

Use SVC\_REQ 4 to change the Backplane Communications window mode and timer value. The change takes place during the next CPU sweep after the function is called.

SVC\_REQ 4 executes unless a mode other than 0 (Limited) or 2 (Run to Completion) is selected.

The parameter block has a length of one word.

### 6.5.1 To disable the Backplane Communications window:

Enter SVC\_REQ 4 with this parameter block:

Address	High Byte	Low Byte
Address	0	0

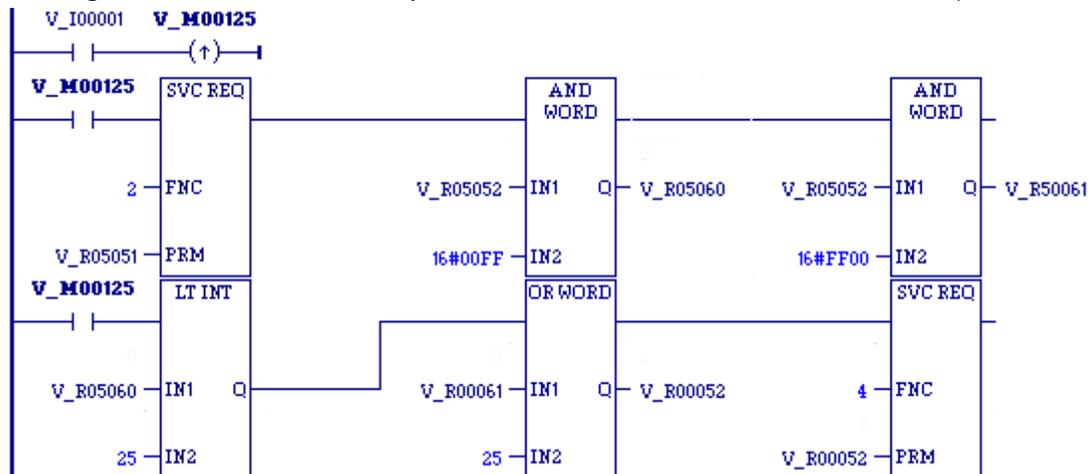
### 6.5.2 To enable the Backplane Communications window mode:

Enter SVC\_REQ 4 with this parameter block:

Address	High Byte	Low Byte
Address	Mode 0 = Limited 2 = Run to Completion	1ms ≤ value ≤ 255ms

### SVC\_REQ 4 Example

When enabling output %M0125 transitions on, the mode and timer value of the Backplane Communications window is read. If the timer value is greater than or equal to 25ms, the value is not changed. If it is less than 25ms, the value is changed to 25ms. In either case, when the rung completes execution the window is enabled. The parameter block for all three windows is at location %R5051. Since the mode and timer for the Backplane Communications window is the second value in the parameter block returned from the Read Window Values function (SVC\_REQ 2), the location of the existing window time for the Backplane Communications window is in the low byte of %R5052.



## 6.6 SVC\_REQ 5: Change Background Task Window Mode and Timer Value

Use SVC\_REQ 5 to change the Background Task window mode and timer value. The change takes place during the next CPU sweep after the function is called.

SVC\_REQ 5 executes unless a mode other than 0 (Limited) or 2 (Run-to-Completion) is selected.

The parameter block has a length of one word.

### 6.6.1 To disable the Background Task window:

Enter SVC\_REQ 5 with this parameter block:

Address	High Byte	Low Byte
Address	0	0

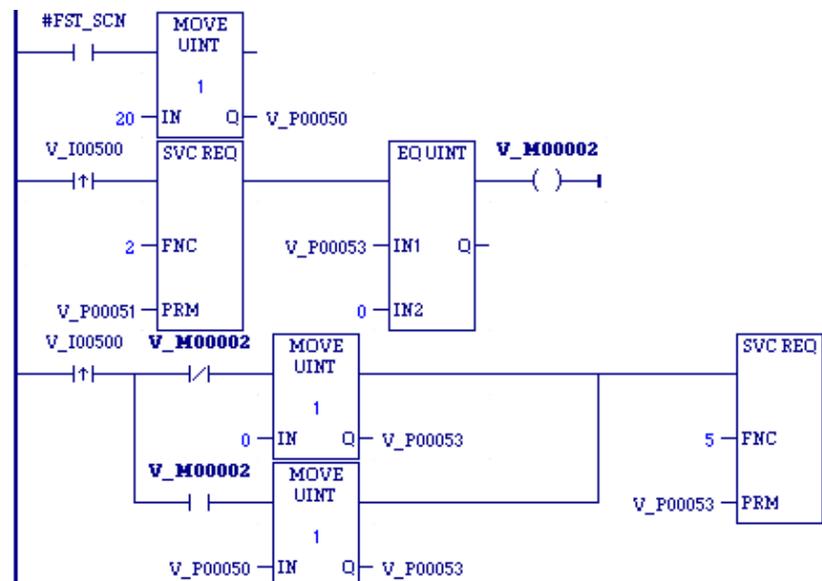
### 6.6.2 To enable the Background Task window mode:

Enter SVC\_REQ 5 with this parameter block:

Address	High Byte	Low Byte
Address	Mode 0 = Limited 2 = Run to Completion	1ms ≤ value ≤ 255ms

### SVC\_REQ 5 Example

When enabling contact #FST\_SCN is set in the first scan, the MOVE function establishes a value of 20ms for the Background task window, using a parameter block beginning at %P00050. Later in the program, when input %I00500 transitions on, the state of the Background task window toggles on and off. The parameter block for all three windows is at location %P00051. The time for the Background task window is the third value in the parameter block returned from the Read Window Values function (function #2); therefore, the location of the existing window time for the Background window is %P00053.



## 6.7 SVC\_REQ 6: Change/Read Number of Words to Checksum

Use SVC\_REQ 6 to read the current word count in the program to be check-summed or set a new word count. By default, 16 words are checked. The function is successful unless some number other than 0 or 1 is entered as the requested operation.

The parameter block has a length of 2 words.

### 6.7.1 To read the word count:

Enter a zero in the first word of the parameter block.

<b>Address</b>	0
<b>Address + 1</b>	Ignored

The function returns the current checksum (word count) in the second word of the parameter block. No range is specified for the read function; the value returned is the number of words currently being check-summed.

<b>Address</b>	0
<b>Address + 1</b>	Current word count

### 6.7.2 To set a new word count:

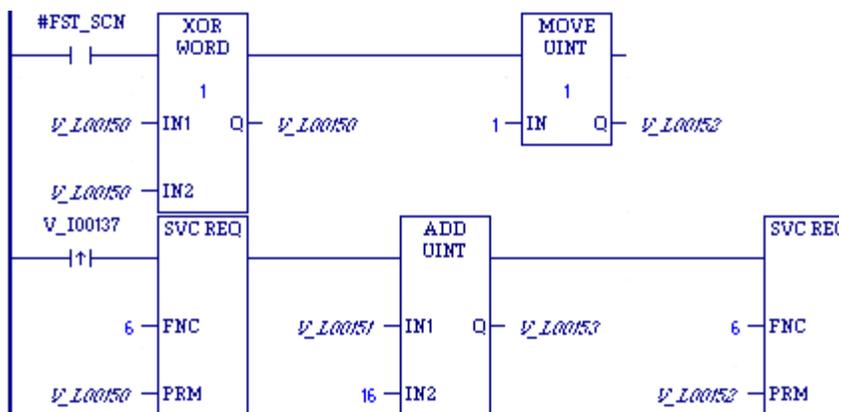
Enter a one in the first word of the parameter block and the new word count in the second word.

<b>Address</b>	1
<b>Address + 1</b>	New word count

The CPU changes the number of words to be check-summed to the value given in the second word of the parameter block, rounded up to the next multiple of 8. To disable check-summing, set the new word count to 0.

**SVC\_REQ 6 Example**

When enabling contact #FST\_SCN is set, the parameter blocks for the checksum task function are built. Later in the program, when input %I00137 transitions on, the number of words being check-summed is read from the CPU operating system. This number is increased by 16, with the results of the ADD\_UINT function being placed in the *hold new count for set* parameter. The second service request block requests the CPU to set the new word count.



The example parameter blocks are located at address %L00150. They have the following contents:

%L00150	0 = read current count
%L00151	hold current count
%L00152	1 = set current count
%L00153	hold new count for set

## 6.8 SVC\_REQ 7: Read or Change the Time-of-Day Clock

Use SVC\_REQ 7 to read or change the time of day clock in the CPU. The function is successful unless:

- An invalid number is entered for the requested operation.
- An invalid data format is specified.
- Data is provided in an unexpected format.

### 6.8.1 Parameter Block Formats

In the first two words of the parameter block, you specify whether to read or set the time and date, and which format to use.

<b>Address</b>	<b>2-Digit Year Format</b>	<b>4-Digit Year Format</b>
<b>Address (word 1)</b>	0 = read time and date	0 = read time and date
	1 = set time and date	1 = set time and date
<b>Address+1 (word 2)</b>	0 = numeric data format	80h = numeric data format
	1 = BCD format	81h = BCD format
	2 = unpacked BCD format	82h = unpacked BCD format
	3 = packed ASCII format (with embedded spaces and colons)	83h = packed ASCII format
	4 = POSIX format	n/a
<b>Address+2 (word 3) to the end</b>	Data	Data

Words 3 to the end of the parameter block contain output data returned by a read function, or new data being supplied by a change function. In both cases, format of these data words is the same. When reading the date and time, words (address + 2) to the end of the parameter block are ignored on input.

The format and length of the parameter block depends on the data format and number of digits required for the year:

<b>Data Format and N-digit Year</b>	<b>Length of parameter block (number of words)</b>
BCD, 2-digit year	6
BCD, 4-digit year	6
POSIX format	6
Unpacked BCD 2	9
Unpacked BCD 4	10
Numeric (2 and 4 digit years)	9
Packed ASCII, 2-digit year	12
Packed ASCII, 4-digit year	13

In any format:

- Hours are stored in 24-hour format.
- Day of the week is a numeric value ranging from 1 (Sunday) to 7 (Saturday).

Value	Day of the Week
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

### BCD, 2-Digit Year

In BCD format, each time and date item occupies one byte, so the parameter block has six words. The last byte of the sixth word is not used. When setting the date and time, this byte is ignored; when reading date and time, the function returns a null character (00).

Parameter Block Format	Address	Example (Sun., July 3, 2005, at 2:45:30 p.m. = 14:45:30 in 24-hour format)
1 = change or 0 = read	Address	0 (read)
1 (BCD format)	Address+1	1 (BCD format)

High Byte	Low Byte	Address	High Byte	Low Byte
month	year	Address+2	07 (July)	05 (year)
hours	day of month	Address+3	14 (hours)	03 (day)
seconds	minutes	Address+4	30 (seconds)	45 (minutes)
(null)	day of week	Address+5	00	01 (Sunday)

### BCD, 4-Digit Year

In this format, all bytes are used.

Parameter Block Format	Address	Example (Sun., July 3, 2005, at 2:45:30 p.m. = 14:45:30 in 24-hour format)
1 = change or 0 = read	Address	00 (read)
81h (BCD format, 4-digit)	Address+1	81h (BCD format, 4-digit)

High Byte	Low Byte	Address	High Byte	Low Byte
year	year	Address+2	20 (year)	05 (year)
day of month	month	Address+3	03 (day)	07 (July)
minutes	hours	Address+4	45 (minutes)	14 (hours)
day of week	seconds	Address+5	01 (Sunday)	30 (seconds)

## POSIX

The POSIX format of the Time-of-Day clock uses two signed 32-bit integers (two DINTs) to represent the number of seconds and nanoseconds since midnight January 1, 1970. Reading the clock in POSIX format might make it easier for your application to calculate time differences. This format can also be useful if your application communicates to other devices using the POSIX time format. To read and/or change the date and time using POSIX format, enter SVC\_REQ 7 with this parameter block:

<b>Parameter Block Format</b>	<b>Address</b>	<b>Example: December 1, 2000 at 12 noon</b>
1 = change or 0 = read	<b>Address</b>	0
4 (POSIX format)	<b>Address+1</b>	4
seconds (LSW)	<b>Address+2</b>	975,672,000
(MSW)	<b>Address+3</b>	
nanoseconds (LSW)	<b>Address+4</b>	0
(MSW)	<b>Address+5</b>	

The PACSystems CPU's maximum POSIX clock value is F48656FE (hexadecimal) seconds and 999,999,999 (decimal) nanoseconds, which corresponds to December 31st, 2099 at 11:59 pm. This is the maximum POSIX value that SVC\_REQ 7 will accept for changing the clock. This is also the maximum POSIX value SVC\_REQ 7 will return once the Time-Of-Day clock passes this date.

If SVC\_REQ 7 receives an invalid POSIX time to write to the clock, it does not change the Time-Of-Day clock and disables its power-flow output.

**Note:** When reading the PACSystems CPU clock in POSIX format, the data returned is not easily interpreted by a human viewer. If desired, it is up to the application logic to convert the POSIX time into year, month, day of month, hour, and seconds.

**Note:** At 03:14:08 UTC on 19 January 2038, 32-bit versions of the Unix time stamp will cease to work, as it will overflow the largest value that can be held in a signed 32-bit number (7FFFFFFF16 or 2,147,483,647). Before this moment, software using 32-bit time stamps will need to adopt a new convention for time stamps, and file formats using 32-bit time stamps will need to be changed to support larger time stamps or a different epoch.

### Unpacked BCD (2-Digit Year)

In Unpacked BCD format, each digit of the time and date items occupies the low-order four bits of a byte. The upper four bits of each byte are always zero. This format requires nine words. Values are hexadecimal.

<b>Parameter Block Format</b>	<b>Address</b>	<b>Example (Thurs., Dec. 8, 2002, at 9:34:57 a.m.)</b>
1 = change or 0 = read	<b>Address</b>	0h
2 (Unpacked BCD format)	<b>Address+1</b>	2h

<b>High Byte</b>	<b>Low Byte</b>		<b>High Byte</b>	<b>Low Byte</b>
	year	<b>Address+2</b>	00h	02h
	month	<b>Address+3</b>	01h	02h
	day of month	<b>Address+4</b>	02h	08h
	hours	<b>Address+5</b>	00h	09h
	minutes	<b>Address+6</b>	03h	04h
	seconds	<b>Address+7</b>	05h	07h
	day of week	<b>Address+8</b>	00h	05h

### Unpacked BCD (4-Digit Year)

In Unpacked BCD format, each digit of the time and date items occupies the low-order four bits of a byte. The upper four bits of each byte are always zero. This format requires nine words. Values are hexadecimal.

<b>Parameter Block Format</b>	<b>Address</b>	<b>Example (Thurs., Dec. 8, 2002, at 9:34:57 a.m.)</b>
1 = change or 0 = read	<b>Address</b>	0h
82h (Unpacked 4-digit BCD format)	<b>Address+1</b>	82h

<b>High Byte</b>	<b>Low Byte</b>		<b>High Byte</b>	<b>Low Byte</b>
	year	<b>Address+2</b>	00h	02h
	month	<b>Address+3</b>	01h	02h
	day of month	<b>Address+4</b>	00h	08h
	hours	<b>Address+5</b>	00h	09h
	minutes	<b>Address+6</b>	03h	04h
	seconds	<b>Address+7</b>	05h	07h
	day of week	<b>Address+8</b>	00h	05h

### Numeric, 2-Digit Year

In numeric format, the year, month, day of month, hours, minutes, seconds and day of week each occupy one unsigned integer. To read and/or change the date and time using the numeric format, enter SVC\_REQ function #7 with this parameter block:

<b>Parameter Block Format</b>	<b>Address</b>	<b>Example</b>
		<b>Wed., June 15, 2005, at 12:15:30 a.m.</b>
1 = change or 0 = read	<b>Address</b>	0
0 (Numeric format, 2-digit year)	<b>Address+1</b>	0

<b>High Byte</b>	<b>Low Byte</b>		<b>Value</b>
	year	<b>Address+2</b>	05
	month	<b>Address+3</b>	06
	day of month	<b>Address+4</b>	15
	hours	<b>Address+5</b>	12
	minutes	<b>Address+6</b>	15
	seconds	<b>Address+7</b>	30
	day of week	<b>Address+8</b>	04

### Numeric, 4-Digit Year

In numeric format, the year, month, day of month, hours, minutes, seconds and day of week each occupy one unsigned integer. To read and/or change the date and time using the numeric format, enter SVC\_REQ function #7 with this parameter block:

<b>Parameter Block Format</b>	<b>Address</b>	<b>Example: Wed., June 15, 2005, at 12:15:30 a.m.</b>
1 = change or 0 = read	<b>Address</b>	0
80h (Numeric format, 4 digit year)	<b>Address+1</b>	80h

<b>High Byte</b>	<b>Low Byte</b>		<b>Value</b>
	year	<b>Address+2</b>	2005
	month	<b>Address+3</b>	06
	day of month	<b>Address+4</b>	15
	hours	<b>Address+5</b>	12
	minutes	<b>Address+6</b>	15
	seconds	<b>Address+7</b>	30
	day of week	<b>Address+8</b>	04

### Packed ASCII, 2-Digit Year

In Packed ASCII format, each digit of the time and date items is an ASCII formatted byte. Spaces and colons are embedded into the data to format it for printing or display. ASCII format for a 2-digit year requires 12 words in the parameter block. Values are hexadecimal.

<b>Parameter Block Format</b>	<b>Address</b>	<b>Example</b> (Mon., Oct. 5, 2005, at 11:13:25 p.m. = 23:13:25 in 24-hour format)
1 = change or 0 = read	<b>Address</b>	0h (read)
3 (ASCII format)	<b>Address+1</b>	3h (ASCII format)

<b>High Byte</b>	<b>Low Byte</b>		<b>High Byte</b>	<b>Low Byte</b>
year	year	<b>Address+2</b>	35h (5)	30h (0)
month	(space)	<b>Address+3</b>	31h (1)	20h (space)
(space)	month	<b>Address+4</b>	20h (space)	30h (0)
day of month	day of month	<b>Address+5</b>	35h (5)	30h (leading 0)
hours	(space)	<b>Address+6</b>	32h (2)	20h (space)
:(colon)	hours	<b>Address+7</b>	3Ah (:)	33h (3)
minutes	minutes	<b>Address+8</b>	33h (3)	31h (1)
seconds	:(colon)	<b>Address+9</b>	32h (2)	3Ah (:)
(space)	seconds	<b>Address+10</b>	20h (space)	35h (5)
day of week	day of week	<b>Address+11</b>	32h (2 = Mon.)	30h (leading 0)

**Packed ASCII, 4-Digit Year**

ASCII format for a 4-digit year requires 13 words in the parameter block. Values are hexadecimal.

<b>Parameter Block Format</b>	<b>Address</b>	<b>Example</b> (Mon., Oct. 5, 2005, at 11:13:25 p.m. = 23:13:25 in 24-hour format)
1 = change or 0 = read	<b>Address</b>	0h (read)
83 (ASCII format)	<b>Address+1</b>	83h (ASCII format, 4-digit)

<b>High Byte</b>	<b>Low Byte</b>		<b>High Byte</b>	<b>Low Byte</b>
year (hundreds)	year (thousands)	<b>Address+2</b>	30h (0)	32h (2)
year (ones)	year (tens)	<b>Address+3</b>	35h (5)	30h (0)
month (tens)	(space)	<b>Address+4</b>	31h (1)	20h (space)
(space)	month (ones)	<b>Address+5</b>	20h (space)	30h (0)
day of month (ones)	day of month (tens)	<b>Address+6</b>	35h (5)	30h (leading 0)
hours (tens)	(space)	<b>Address+7</b>	32h (2)	20h (space)
: (colon)	hours (ones)	<b>Address+8</b>	3Ah (:)	33h (3)
minutes (ones)	minutes (tens)	<b>Address+9</b>	33h (3)	31h (1)
seconds (tens)	: (colon)	<b>Address+10</b>	32h (2)	3Ah (A)
(space)	seconds (ones)	<b>Address+11</b>	20 (space)	35 (5)
day of week (ones)	day of week (tens)	<b>Address+12</b>	32h (2 = Mon.)	30h (leading 0)

**SVC\_REQ 7 Example**

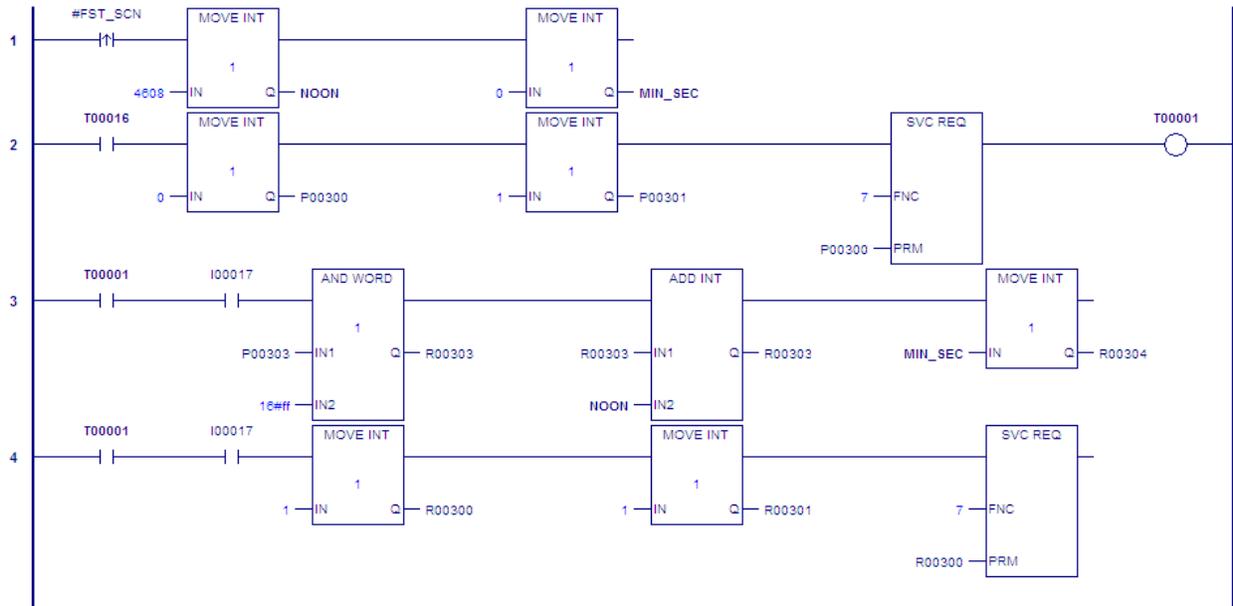
In this example, the time of day is set to 12:00 pm without changing the current year, BCD format requires six contiguous memory locations for the parameter block.

Rung 1 sets up the new time of day in two-digit year BCD format. It writes the value 4608 (equivalent to 12 00 BCD) to NOON and the value 0 to MIN\_SEC.

Rung 2 requests the current date and time using the parameter block located at %P00300.

Rung 3 moves the new time value into the parameter block starting at R00300. It uses AND and ADD operations to retrieve the current clock value from %P00303 and replace the hours, minutes and seconds portion of the value with the values in NOON and MIN\_SEC.

Rung 4 uses the parameter block beginning at %R00300 to set the new time.



## 6.9 SVC\_REQ 8: Reset Watchdog Timer

Use SVC\_REQ 8 to reset the watchdog timer during the scan.

Ordinarily, when the watchdog timer expires, the CPU goes to STOP-Halt mode without warning. SVC\_REQ 8 allows the timer to keep going during a time-consuming task (for example, while waiting for a response from a communications line).



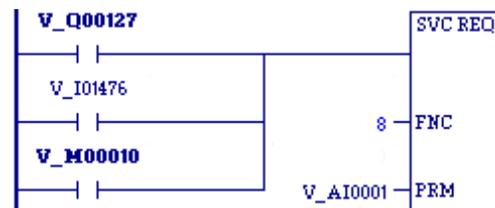
### Warning

**Be sure that resetting the watchdog timer does not adversely affect the controlled process.**

SVC\_REQ 8 has no associated parameter block; however, you must specify a dummy parameter, which SVC\_REQ 8 will not use.

### SVC\_REQ 8 Example

In the LD example at right, power flow through enabling output %Q0127 or input %I1476 or internal coil %M00010 causes the watchdog timer to be reset.



## 6.10 SVC\_REQ 9: Read Sweep Time from Beginning of Sweep

Use SVC\_REQ 9 to read the time in milliseconds since the start of the sweep. The data format is unsigned 16-bit integer.

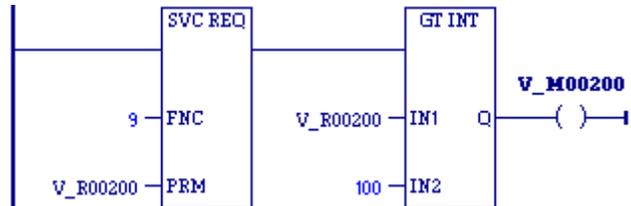
### Output

The parameter block is an output parameter block only; it has a length of one word.

<b>Address</b>	time since start of scan
----------------	--------------------------

### SVC\_REQ 9 Example

The elapsed time from the start of the scan is read into location %R00200. If it is greater than 100ms, internal coil %M0200 is turned on.



**Note:** Higher resolution (in nanoseconds) can be obtained by using *SVC\_REQ 51: Read Sweep Time from Beginning of Sweep*.

## 6.11 SVC\_REQ 10: Read Target Name

Use SVC\_REQ 10 to read the name of the currently executing target.

### Output

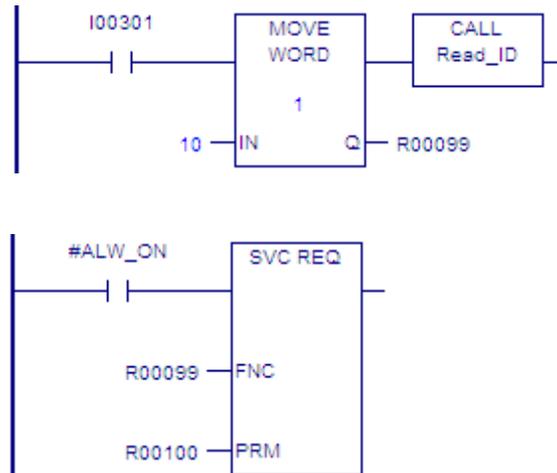
The output parameter block has a length of four words. It returns eight ASCII characters: the target name (from one to seven characters) followed by null characters (00h). The last character is always a null character. If the target name has fewer than seven characters, null characters are appended to the end.

Address	Low Byte	High Byte
Address	character 1	character 2
Address+1	character 3	character 4
Address+2	character 5	character 6
Address+3	character 7	00

### SVC\_REQ 10 Example

When enabling input %I0301 goes ON, register location %R0099 is loaded with the value 10, which is the function code for the Read Target Name function. The program block READ\_ID is then called to retrieve the target name. The parameter block is located at address %R0100.

Program block READ\_ID:



## 6.12 SVC\_REQ 11: Read Controller ID

Use SVC\_REQ 11 to read the name of the controller executing the program.

### Output

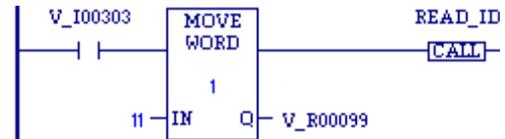
The output parameter block has a length of four words. It returns eight ASCII characters: the Controller ID (from one to seven characters) followed by null characters (00h). The last character is always a null character

If the Controller ID has fewer than seven characters, null characters are appended to the end.

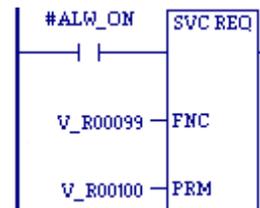
Address	Low Byte	High Byte
Address	character 1	character 2
Address+1	character 3	character 4
Address+2	character 5	character 6
Address+3	character 7	00

### SVC\_REQ 11 Example

When enabling input %I0303 is ON, register location %R0099 is loaded with the value 11, which is the function code for the Read Controller ID function. The program block READ\_ID is then called to retrieve the ID. The parameter block is located at address %R0100.



Program Block READ\_ID:



### 6.13 SVC\_REQ 12: Read Controller Run State

Use SVC\_REQ 12 to read the current RUN state of the CPU.

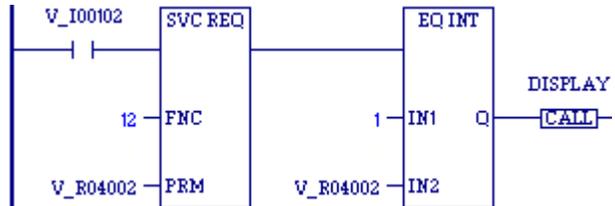
#### Output

The output parameter block has a length of one word.

<b>Address</b>	1 = run/disabled
	2 = run/enabled

#### SVC\_REQ 12 Example

When contact V\_I00102 is ON, the CPU run state is read into location %R4002. If the state is Run/Disabled, the CALL function calls program block DISPLAY.



## 6.14 SVC\_REQ 13: Shut Down (STOP) CPU

Use SVC\_REQ 13 to stop the CPU after the specified number of scans has been performed. All outputs go to their designated default states at the start of the next CPU scan. An informational *Shut Down Controller* fault is placed in the Controller Fault Table. The I/O scan continues as configured.

SVC\_REQ 13 has an input parameter block with a length of one word.

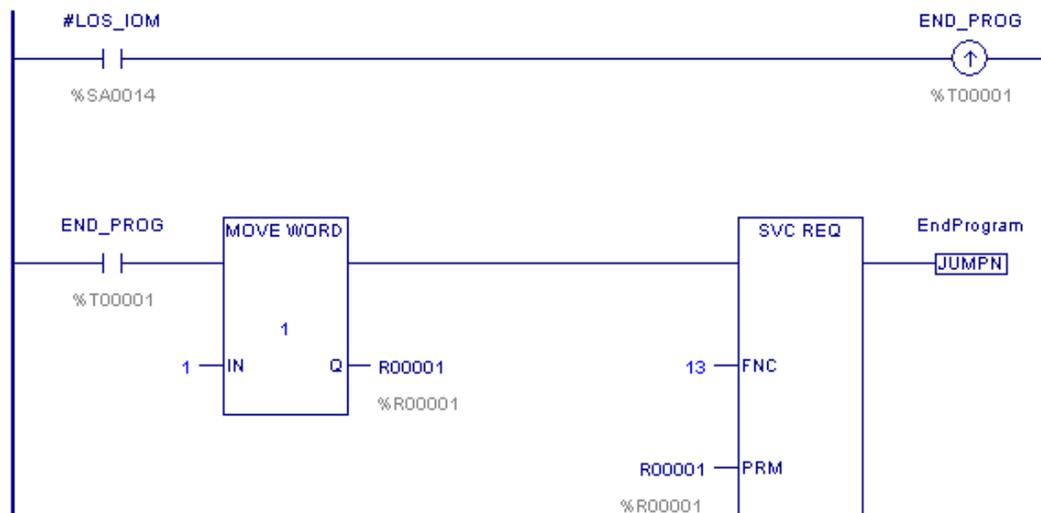
<b>Address</b>	Number of scans. Valid values: -1: The CPU uses the Number of Last Scans value configured in the Hardware Configuration Scan tab to determine when to transition to STOP Mode. For details on Hardware Configuration parameters, refer to <i>PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual</i> , GFK-2222. 1 through 5: The CPU finishes executing this scan, then executes this number of scans -1, and transitions to STOP Mode.
----------------	---

**Note:** For CPUs with firmware version earlier than 2.00, the value must be set to 0; otherwise the CPU does not stop.

### SVC\_REQ 13 Example

When a *Loss of I/O Module* fault occurs, the #LOS\_IOM contact turns ON and SVC\_REQ 13 executes.

In this example, if the *Shut Down CPU* function executes, the JUMPN to the end of the program prevents the logic that follows the JUMPN from executing in the current sweep.



The block's last instruction is a LABELN:



### 6.15 SVC\_REQ 14: Clear Controller or I/O Fault Table

Use SVC\_REQ 14 to clear either the Controller Fault Table or the I/O Fault Table. The SVC\_REQ output is set ON unless some number other than 0 or 1 is entered as the requested operation.

The parameter block has a length of 1 word. It is an input parameter block only. There is no output parameter block.

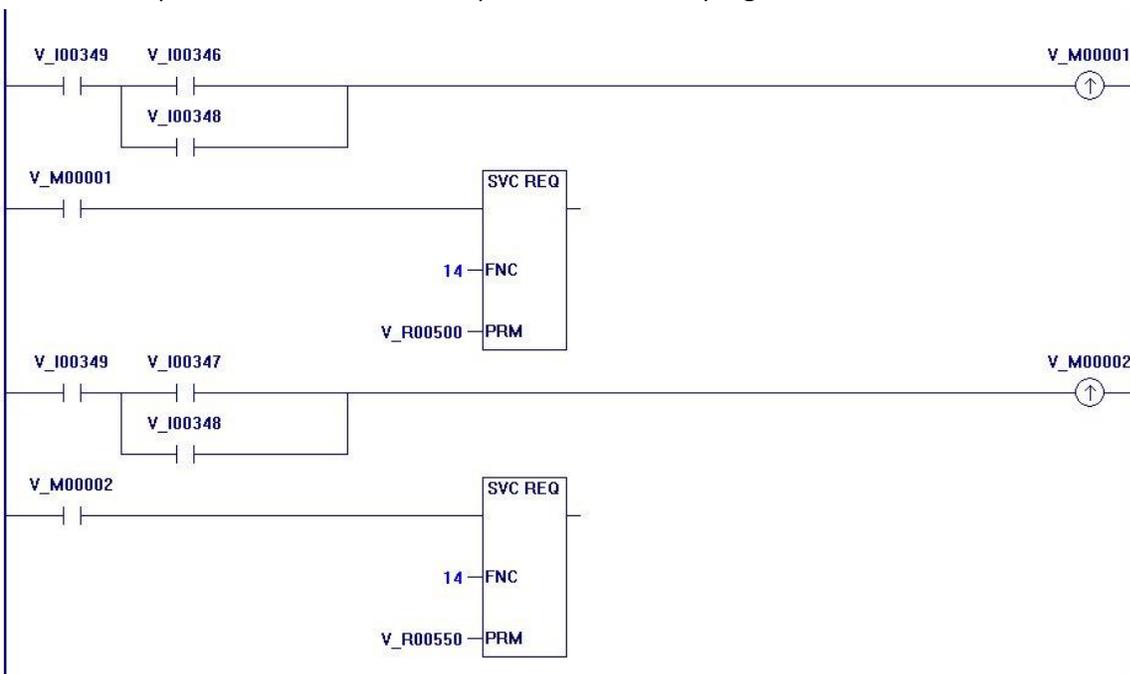
<b>Address</b>	0 = clear Controller Fault Table
	1 = clear I/O Fault Table

#### SVC\_REQ 14 Example

When inputs %I0346 and %I0349 are on, the Controller Fault Table is cleared. When inputs %I0347 and %I0349 are on, the I/O Fault Table is cleared. When input %I0348 is on and input %I0349 is on, both are cleared. Positive transition coils V\_M00001 and V\_M00002 are used to trigger these service requests to prevent the fault tables from being cleared multiple times.

The parameter block for the Controller Fault Table is located at %R0500; for the I/O Fault Table the parameter block is located at %R0550.

**Note:** Both parameter blocks are set up elsewhere in the program.



## 6.16 SVC\_REQ 15: Read Last-Logged Fault Table Entry

Use SVC\_REQ 15 to read the last entry logged in the Controller Fault Table or the I/O Fault Table. The SVC\_REQ output is set ON unless some invalid number is entered as the requested operation or the fault table is empty.

The non-extended parameter block has a length of 22 words and the extended parameter block has a length of 24 words.

### Input Parameter Block

Address	Format
Address+0	0 = Read Controller Fault Table
	1 = Read I/O Fault Table
	80h = Read extended Controller Fault Table
	81h = Read extended I/O Fault Table

### Output Parameter Block

The format of the output parameter block depends on whether SVC\_REQ 15 reads the Controller Fault Table, the extended Controller Fault Table, the I/O Fault Table or the extended I/O Fault Table.

Controller Fault Table Output Format		Address	I/O Fault Table Output Format	
High Byte	Low Byte		High Byte	Low Byte
	0	Address+0		1
unused	long/short (always 01)	Address+1	reference address memory type	long/short (always 03)
unused	unused	Address+2	reference address offset	
slot	rack	Address+3	slot	rack
	task	Address+4	block	bus
fault action	fault group	Address+5		point
error code		Address+6	fault action	fault group
fault extra data		Address+7	fault type	fault category
		Address+8 to Address+18	fault extra data	fault description
minutes	seconds	Address+19	minutes	seconds
day of month	hour	Address+20	day of month	hour
year	month	Address+21	year	month
milliseconds (extended format only)		Address+22	milliseconds (extended format only)	
not used (extended format only)		Address+23	not used (extended format only)	

### Long/Short Value

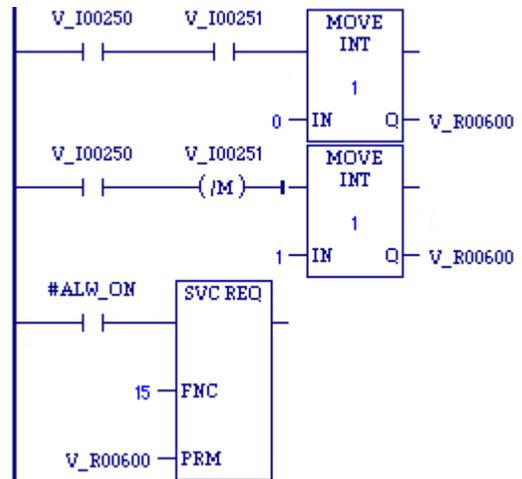
The first byte (low byte) of word *address +1* contains a number that indicates the length of the fault-specific data in the fault entry. Possible values are as follows:

Controller extended and non-extended fault tables	00 = 8 bytes (short)	01 = 24 bytes (long)
I/O extended and non-extended fault tables	02 = 5 bytes (short)	03 = 21 bytes (long)

**Note:** PACSystems CPUs always return the Long values for both extended and non-extended formats.

### SVC\_REQ 15 Example 1

When inputs %I0250 and %I0251 are both on, the first Move function places a zero (read Controller Fault Table) into the parameter block for SVC\_REQ 15. When input %I0250 is on and input %I0251 is off, the Move instruction instead places a one (read I/O Fault Table) in the SVC\_REQ parameter block. The parameter block is located at location %R0600.



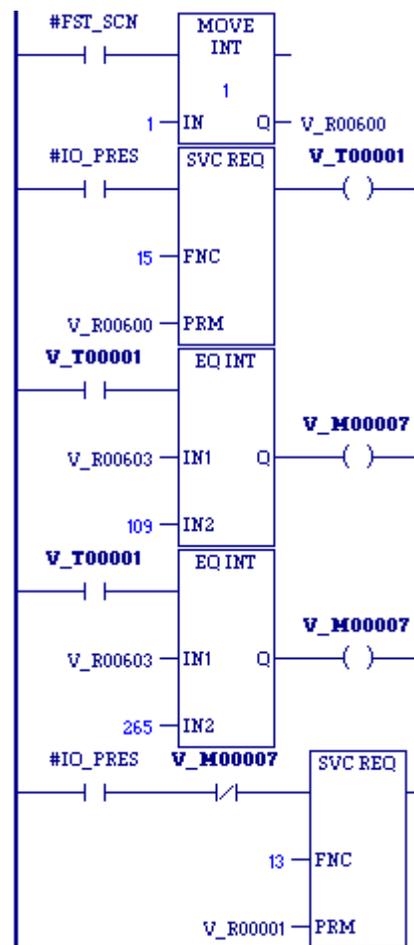
SVC\_REQ 15 Example 2

The CPU is shut down when any fault occurs on an I/O module except when the fault occurs on modules in rack 0, slot 9 and in rack 1, slot 9. If faults occur on these two modules, the system remains running. The parameter for *table type* is set up on the first scan. The contact IO\_PRES, when set, indicates that the I/O Fault Table contains an entry. The CPU sets the normally open contact in the scan after the fault logic places a fault in the table. If faults are placed in the table in two consecutive scans, the normally open contact is set for two consecutive scans.

The example uses a parameter block located at %R0600. After the SVC\_REQ function executes, the second, third, and fourth words of the parameter block identify the I/O module that faulted:

	High Byte	Low Byte
%R0600		1
%R0601	reference address memory type	long/short
%R0602	reference address offset	
%R0603	slot number	rack number
%R0604	block (bus address)	I/O bus no.
%R0605		point address
%R0606	fault data	

In the program, the EQ\_INT blocks compare the rack/slot address in the table to hexadecimal constants. The internal coil %M0007 is turned on when the rack/slot where the fault occurred meets the criteria specified above. If %M0007 is on, its normally closed contact is off, preventing the shutdown. Conversely, if %M0007 is off because the fault occurred on a different module, the normally closed contact is on and the shutdown occurs.



## 6.17 SVC\_REQ 16: Read Elapsed Time Clock

Use SVC\_REQ 16 to read the system's elapsed time clock. The elapsed time clock measures the time in seconds since the CPU was powered on. The parameter block has a length of three words used for output only.

### Output

<b>Address</b>	Seconds from power on (low order)
<b>Address+1</b>	Seconds from power on (high order)
<b>Address+2</b>	100 microsecond ( $\mu$ s) ticks

The first two words are the elapsed time in seconds. The last word is the number of 100  $\mu$ s ticks in the current second.

The resolution of the CPU's elapsed time clock is 100 microseconds ( $\mu$ s). The overall accuracy of the elapsed time clock is  $\pm 0.01\%$ . The accuracy of an individual sample of the elapsed time clock is approximately 105  $\mu$ s.



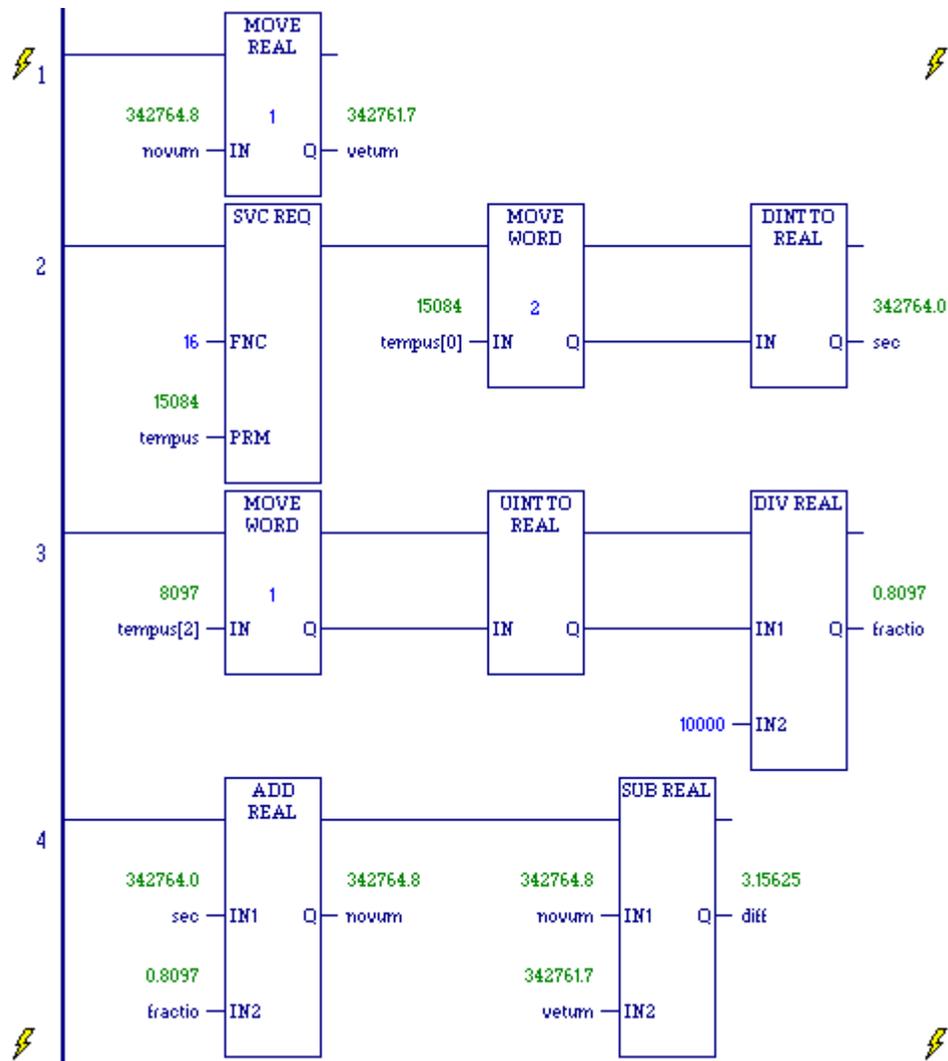
### Warning

**The SVC\_REQ instruction is not protected against operating system and user interrupts. The timing and length of these interrupts are unpredictable. The clock sample returned by SVC\_REQ 16 can sometimes be much more than 105  $\mu$ s old by the time execution is returned to the LD logic.**

### SVC\_REQ 16 Example

The following logic is used in a block that is called infrequently. The screen shot was taken between calls to the block. The logic displayed calculates the number of seconds that have elapsed since the last time the block was called. It performs the final operation on rung 4 by subtracting the time obtained by SVC\_REQ 16 the last time the block was called (vetum) from the time currently obtained by SVC\_REQ 16 (novum) and storing the calculated value in the variable named diff.

On rung 2, SVC\_REQ 16 returns three WORDs, stored in the 3-WORD array tempus. The first two WORDs (16-bit values) are moved to a DINT (a 32-bit value). This move amounts to a rough data type conversion that ignores the fact that the DINT type is actually a signed value. Despite that, the subsequent calculations are correct until the time since power-on reaches approximately 50 years. The DINT is converted to REAL to yield the number of whole seconds elapsed since power-on, stored in variable sec. On rung 3, the third word returned by SVC\_REQ 16, tempus[2], is converted to REAL. This is the number of 100  $\mu$ s ticks. To obtain a fraction of a second, stored in the variable fractio, the value is divided by 10,000. On rung 4, sec and fractio are added to express the exact number of seconds elapsed since power-on, and this value is stored in the variable novum. On rung 1, the previous value of novum was saved as vetum, the exact number of seconds elapsed since power-on the last time the block was called. The last instruction on the fourth rung subtracts vetum from novum to yield the number of seconds that have elapsed since the last time the block was called.



**Note:** Higher resolution (in nanoseconds) can be obtained by using `SVC_REQ 50: Read Elapsed Time Clock`.

## 6.18 SVC\_REQ 17: Mask/Unmask I/O Interrupt

Use SVC\_REQ 17 to mask or unmask an interrupt from an input/output board. When an interrupt is masked, the CPU does not execute the corresponding interrupt block when the input transitions and causes an interrupt.

The parameter block is an input parameter block only; it has a length of three words.

<b>Address</b>	0 = unmask input 1 = mask input
<b>Address+1</b>	memory type
<b>Address+2</b>	reference (offset)

*Memory type* is a decimal number that resides in the low byte of word *address + 1*. It corresponds to the memory type of the input:

<b>70</b>	%I memory in bit mode
<b>10</b>	%AI memory
<b>12</b>	%AQ memory

Successful execution occurs unless:

- Some number other than 0 or 1 is entered as the requested operation.
- The memory type of the input/output to be masked or unmasked is not %I, %AI or %AQ memory.
- The I/O board is not a supported input/output module.
- The reference address specified does not correspond to a valid interrupt trigger reference.
- The specified channel does not have its interrupt enabled in the configuration.

### 6.18.1 Masking/Unmasking Module Interrupts

During module configuration, interrupts from a module can be enabled or disabled. If a module's interrupt is disabled, it cannot be used to trigger logic execution in the application program and it cannot be unmasked. However, if an interrupt is enabled in the configuration, it can be dynamically masked or unmasked by the application program during system operation.

The application program can mask and unmask interrupts that are enabled using Service Request Function Block #17. To mask or unmask an interrupt from an open VME module, the application logic should pass VME\_INT\_ID (17 decimal, 11H) as the memory type and the VME interrupt id as the offset to SVC\_REQ 17.

When the interrupt is not masked, the CPU processes the interrupt and schedules the associated program logic for execution. When the interrupt is masked, the CPU processes the interrupt but does not schedule the associated program logic for execution.

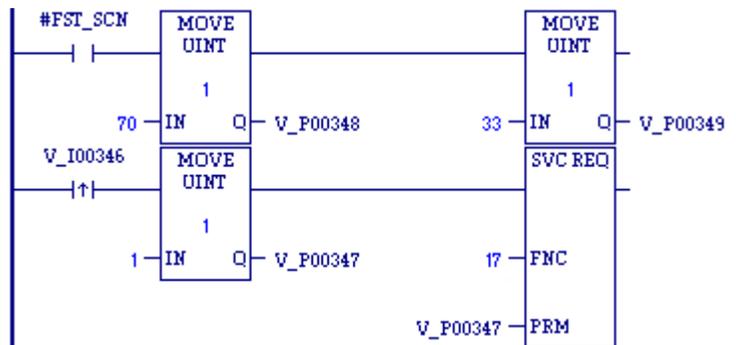
When the CPU transitions from STOP Mode to RUN Mode, the interrupt is unmasked.

For additional information on configuring and using VME module interrupts in a PACSystems RX7i control system, refer to *PACSystems RX7i User's Guide to Integration of VME Modules*, GFK-2235.

**SVC\_REQ 17 Example 1**

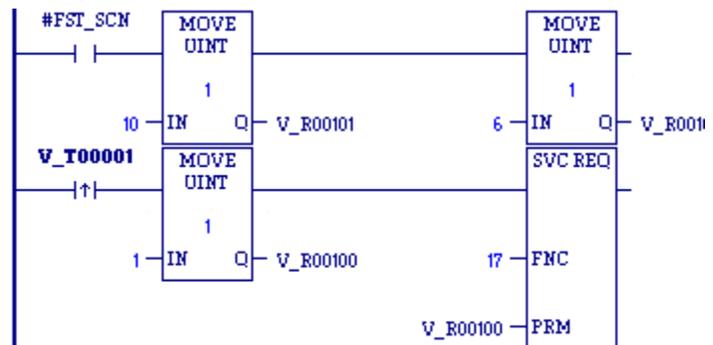
In this example, interrupts from input %I00033 are masked. The following values are moved into the parameter block, which starts at %P00347, on the first scan:

<b>Address</b>	%P00347	1	Interrupts from input are masked.
<b>Address + 1</b>	%P00348	70	Input type is %I.
<b>Address + 2</b>	%P00349	33	Offset is 33.



**SVC\_REQ 17 Example 2**

When %T00001 transitions on, alarm interrupts from input %AI0006 are masked. The parameter block at %R00100 is set up on the first scan.



## 6.19 SVC\_REQ 18: Read I/O Forced Status

Use SVC\_REQ 18 to read the current status of forced values in the CPU's %I and %Q memory areas.

**Note:** SVC\_REQ 18 does not detect overrides in %G or %M memory types. Use %S0011 (#OVR\_PRE) to detect overrides in %I, %Q, %G, %M, and symbolic memory types.

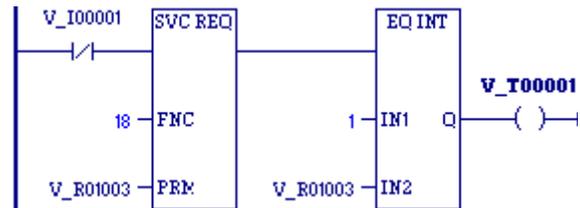
The parameter block has a length of one word used for output only.

### Output

<b>Address</b>	0 = No forced values are set
	1 = Forced values are set

### SVC\_REQ 18 Example

SVC\_REQ reads the status of I/O forced values into location %R1003. If the returned value in %R1003 is 1, there is a forced value, and EQ INT turns the %T0001 coil ON.



## 6.20 SVC\_REQ 19: Set Run Enable/Disable

Use SVC\_REQ 19 to permit the LD program to control the RUN mode of the CPU.

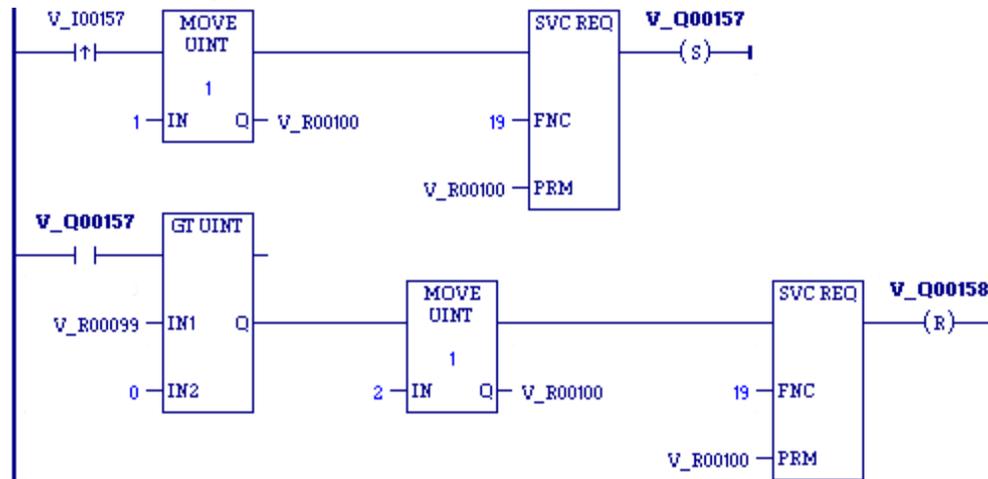
The parameter passed indicates which function to perform. The OK output is turned ON if the function executes successfully. It is set OFF if the requested operation is not SET RUN DISABLE mode (1) or SET RUN ENABLE mode (2).

The parameter block is an input parameter block only with this format:

<b>Address</b>	1 = SET RUN DISABLE mode
	2 = SET RUN ENABLE mode

### SVC\_REQ 19 Example

When input %I00157 transitions to on, the RUN DISABLE mode is set. When the SVC\_REQ function successfully executes, coil %Q00157 is turned on. When %Q00157 is on and register %R00099 is greater than zero, the mode is changed to RUN ENABLE mode. When the SVC\_REQ successfully executes, coil %Q00158 is turned off.



## 6.21 SVC\_REQ 20: Read Fault Tables

Use SVC\_REQ 20 to retrieve the entire Controller or I/O Fault Table and return it to the LD program in designated registers.

The first input parameter designates which table is to be read. A second input parameter (always zero for the standard Read Fault Tables) is used by the extended format to read a designated fault entry or to read a range of fault entries. The fault table data is placed in the parameter block following the input parameters.

The OK output is turned on if the function executes successfully. It is off if the requested operation is not Read Controller Fault Table (00h), Read I/O Fault Table (01h), Read Extended Controller Fault Table (80h), Read Extended I/O Fault Table (81h), Read I/O Fault Table with Remote Fault Record (41h), or Read Extended I/O Fault Table with Remote Fault Record (C1h). The OK output is also turned off if there is insufficient space in the specified memory reference to accommodate the requested fault data. If the specified fault table is empty, the function sets the OK output on, but returns only the fault table header information.

The parameter block is an input and output parameter block. The parameter block comes in two formats:

- Non-Extended: Read Controller Fault Table (00h), Read I/O Fault Table (01h) or Read I/O Fault Table with Remote Fault Record (41h)<sup>6</sup>
- Extended: Read Extended Controller Fault Table (80h), Read Extended I/O Fault Table (81h) or Read Extended I/O Fault Table with Remote Fault Record (C1h)<sup>6</sup>.

---

<sup>6</sup> I/O Fault Table with Remote Fault Record requires RX3i CPU firmware 9.40 or later.

### 6.2.1.1 Non-Extended Formats

#### Input Parameter Block Format

		Amount of Returned Data
<b>Address + 0</b>	00h = Read Controller Fault Table 01h = Read I/O Fault Table 41h = Read I/O Fault Table with Remote Fault Record	693 registers required for resulting output 693 registers required for resulting output 757 registers required for resulting output
<b>Address + 1</b>	Always 0	

#### Non-Extended Output Parameter Block Format

<b>Controller Fault Table Output Format</b>		<b>Address</b>	<b>I/O Fault Table Output Format</b>	
<b>High Byte</b>	<b>Low Byte</b>		<b>High Byte</b>	<b>Low Byte</b>
2018 Unused	00h = Controller Fault Table	<b>Address+0</b>	Unused	01h = I/O Fault Table
Unused	Always zero (0)	<b>Address+1</b>	Unused	Always zero (0)
Unused	Unused	<b>Address+2</b>	Unused	Unused
Unused	Unused	<b>Address+3— Address+14</b>	Unused	Unused
Minutes	Seconds	<b>Address+15— Address+17 (Time Since Last Clear, in BCD Format)</b>	Minutes	Seconds
Day of Month	Hour		Day of month	Hour
Year	Month		Year	Month
Number of faults since last clear		<b>Address+18</b>	Number of faults since last clear	
Number of faults in queue		<b>Address+19</b>	Number of faults in queue	
Number of faults read		<b>Address+20</b>	Number of faults read	
Start of fault data		<b>Address+21</b>	Start of fault data	

<b>Address</b>	<b>I/O Fault Table Output Format</b>	
	<b>High Byte</b>	<b>Low Byte</b>
<b>Address+0</b>	Unused	41h = I/O Fault Table with Remote Fault Record
<b>Address+1</b>	Starting index of faults to be read	
<b>Address+2</b>	Number of faults to be read	
<b>Address+3— Address+14</b>	Unused	Unused
<b>Address+15— Address+17 (Time Since Last Clear, in BCD Format)</b>	Minutes	Seconds
	Day of month	Hour
	Year	Month
<b>Address+18</b>	Number of faults since last clear	
<b>Address+19</b>	Number of faults in queue	

<b>Address+20</b>	Number of faults read
<b>Address+21</b>	Start of fault data

For the non-extended formats, the returned data for each fault consists of 21 words (42 bytes) for 00h and 01h and 23 words (46 bytes) for 41h. This request returns 16 Controller Fault Table entries or 32 I/O Fault Table entries, or the actual number of faults, if fewer. If the fault table read is empty, no data is returned.

The following tables show the return format of a Controller Fault Table entry and an I/O Fault Table entry.

### Format of Returned Data for Fault Table Entries

#### Format for Parameter Setting 00h or 01h

<b>Controller Fault Table (00h) Output Format</b>		<b>Address</b>	<b>I/O Fault Table (01h) Output Format</b>	
<b>High Byte</b>	<b>Low Byte</b>		<b>High Byte</b>	<b>Low Byte</b>
Unused	Long/short	<b>Address+21</b>	Memory type	Long/Short <sup>7</sup>
Unused	Unused	<b>Address+22</b>	Offset	
Slot	Rack	<b>Address+23</b>	Slot	Rack
Task		<b>Address+24</b>	Bus address	I/O Bus Number (block)
Fault action	Fault group	<b>Address+25</b>	Point	
Error code		<b>Address+26</b>	Fault action	Fault group
Fault extra data		<b>Address+27</b>	Fault type	Fault category
		<b>Address+28</b>	Fault extra data	Fault description
		<b>Address+29— Address+38</b>	Fault extra data	
Minutes	Seconds	<b>Address+39— Address+41 (Time-stamp, in BCD Format)</b>	Minutes	Seconds
Day of month	Hour		Day of month	Hour
Year	Month		Year	Month

Start of next fault output parameter block	<b>Address+42</b>	Start of next fault output parameter block
--	-------------------	--

<sup>7</sup> The Long/Short indicator in the low byte of *Address + 21* specifies the amount of fault data present in the fault entry:

<b>Fault Table</b>	<b>Long/Short Value</b>	<b>Fault Data Returned</b>
Controller	00	8 bytes of fault extra data present in the fault entry
	01	24 bytes of fault extra data
I/O	02	5 bytes of fault extra data
	03	21 bytes of fault extra data

Format for Parameter Setting 41h

Address	I/O Fault Table with Remote Fault Record (0x41) Output Format	
	High Byte	Low Byte
Address+21	Memory type	Long/Short <sup>7</sup>
Address+22	Offset	
Address+23	Slot	Rack
Address+24	Remote Slot	Remote Rack
Address+25	Remote Sub-Slot	Remote Device ID
Address+26	Bus address	I/O Bus Number (block)
Address+27	Point	
Address+28	Fault action	Fault group
Address+29	Fault type	Fault category
Address+30	Fault extra data	Fault description
Address+31– Address+40	Fault extra data	
Address+41– Address+43 (Time-stamp, in BCD Format)	Minutes	Seconds
	Day of month	Hour
	Year	Month
Address+44	Start of next fault output parameter block	

### 6.21.2 Extended Formats

Each extended format request can read a maximum of 64 faults, or the size of the fault table if it contains fewer than 64 faults.

For extended formats (Read Extended Controller Fault Table (80h), Read Extended I/O Fault Table (81h) or Read Extended I/O Fault Table with Remote Fault Record (C1h)), the controller calculates the number of entries being read. Be sure that sufficient register space is available to accommodate the number of fault entries requested. If the amount of data requested exceeds the register space available, the CPU returns a fault indicating that reference memory is out of range.

The total size of the fault table for the extended fault format is

Header Size + ((# fault entries) × (size of fault entry))

#### Input Parameter Block Format

		Amount of Returned Data
<b>Address+0</b>	80h = Read Extended Controller Fault Table 81h = Read Extended I/O Fault Table C1h = Read Extended I/O Fault Table with Remote Fault Record	23 words (46 bytes) for each fault entry 23 words (46 bytes) for each fault entry 25 words (50 bytes) for each fault entry
<b>Address+1</b>	Starting index of faults to be read	
<b>Address+2</b>	Number of faults to be read	

Extended Format Output Parameter Block Format

<b>Controller Fault Table Output Format</b>		<b>Address</b>	<b>I/O Fault Table Output Format</b>	
<b>High Byte</b>	<b>Low Byte</b>		<b>High Byte</b>	<b>Low Byte</b>
Unused	80h = Extended Controller Fault Table	<b>Address</b>	Unused	81h = Extended I/O Fault Table
Starting index of faults to be read		<b>Address+1</b>	Starting index of faults to be read	
Number of faults to be read		<b>Address+2</b>	Number of faults to be read	
Unused	Unused	<b>Address+3—Address+14</b>	Unused	Unused
Minutes	Seconds	<b>Address+15—Address+17 (Time Since Last Clear, in BCD Format)</b>	Minutes	Seconds
Day of Month	Hour		Day of month	Hour
Year	Month		Year	Month
Number of faults since last clear		<b>Address+18</b>	Number of faults since last clear	
Number of faults in queue		<b>Address+19</b>	Number of faults in queue	
Number of faults read		<b>Address+20</b>	Number of faults read	
Unused		<b>Address+21—Address+36</b>	Unused	
Start of fault data		<b>Address+37</b>	Start of fault data	

<b>Address</b>	<b>I/O Fault Table Output Format</b>	
	<b>High Byte</b>	<b>Low Byte</b>
<b>Address</b>	Unused	C1h = Extended I/O Fault Table with Remote Fault Record
<b>Address+1</b>	Starting index of faults to be read	
<b>Address+2</b>	Number of faults to be read	
<b>Address+3—Address+14</b>	Unused	Unused
<b>Address+15—Address+17 (Time Since Last Clear, in BCD Format)</b>	Minutes	Seconds
	Day of month	Hour
	Year	Month
<b>Address+18</b>	Number of faults since last clear	
<b>Address+19</b>	Number of faults in queue	
<b>Address+20</b>	Number of faults read	
<b>Address+21—Address+36</b>	Unused	
<b>Address+37</b>	Start of fault data	

## Format of Returned Data for Fault Table Entries

## Format for Parameter Setting 0x80h &amp; 0x81h

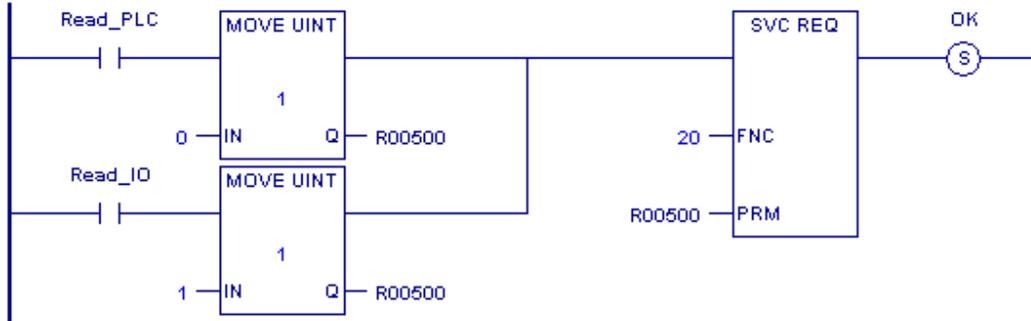
<b>Controller Fault Table (0x80) Output Format</b>		<b>Address</b>	<b>I/O Fault Table (0x81) Output Format</b>	
<b>High Byte</b>	<b>Low Byte</b>		<b>High Byte</b>	<b>Low Byte</b>
Unused	Long/Short	<b>Address+37</b>	Reference address memory type	Long/Short Value
Unused	Unused	<b>Address+38</b>	Reference address offset	
Slot	Rack	<b>Address+39</b>	Slot	Rack
Task		<b>Address+40</b>	Bus address	I/O bus number (block)
Fault action	Fault group	<b>Address+41</b>	point	
Error code		<b>Address+42</b>	Fault action	Fault group
Fault extra data		<b>Address+43</b>	Fault type	Fault category
		<b>Address+44</b>	Fault extra data	Fault description
		<b>Address+45— Address+54</b>	Fault extra data	
Minutes	Seconds	<b>Address+55—</b>	Minutes	Seconds
Day of month	Hour	<b>Address+58</b>	Day of month	Hour
Year	Month	<b>(Time-stamp in BCD Format)</b>	Year	Month
Milliseconds			Milliseconds	
Not used		<b>Address+59</b>	Not used	
Start of next fault output parameter block		<b>Address+60</b>	Start of next fault output parameter block	

Format for Parameter Setting 0xC1h

Address	<b>I/O Fault Table with Remote Fault Record (0xC1) Output Format</b>	
	<b>High Byte</b>	<b>Low Byte</b>
<b>Address+37</b>	Reference address memory type	Long/Short Value
<b>Address+38</b>	Reference address offset	
<b>Address+39</b>	Slot	Rack
<b>Address+40</b>	Remote Slot	Remote Rack
<b>Address+41</b>	Remote Sub-Slot	Remote Device ID
<b>Address+42</b>	Bus address	I/O bus number (block)
<b>Address+43</b>	point	
<b>Address+44</b>	Fault action	Fault group
<b>Address+45</b>	Fault type	Fault category
<b>Address+46</b>	Fault extra data	Fault description
<b>Address+47– Address+56</b>	Fault extra data	
<b>Address+57– Address+60 (Time-stamp in BCD Format)</b>	Minutes	Seconds
	Day of month	Hour
	Year	Month
	Milliseconds	
<b>Address+61</b>	Not used	
<b>Address+62</b>	Start of next fault output parameter block	

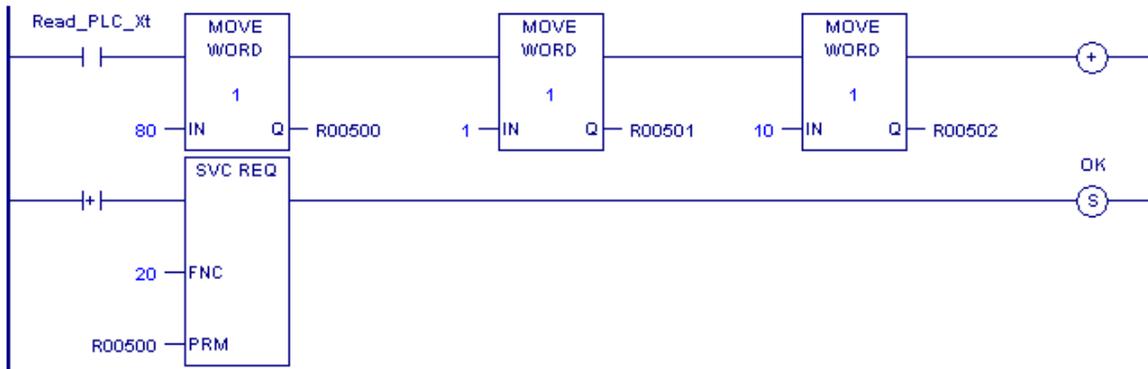
**SVC\_REQ 20 Example 1: Non-Extended Format**

When Read\_PLC transitions on, a value of 0 is moved to the parameter block, which is located at %R00500, and the Controller Fault Table is read. When Read\_IO transitions on, a value of 1 is moved to the parameter block and the I/O Fault Table is read. When the SVC\_REQ function successfully executes, coil OK is turned on.



**SVC\_REQ 20 Example 2: Extended Format**

When Read\_PLC\_Xt transitions on, the Extended Controller Fault Table is read. The parameter block begins at %R00500. %R00500 contains the fault table type (Controller Extended); %R00501 contains the starting fault to read, and %R00502 contains the number of faults to read starting with the fault number in %R00501. When the SVC\_REQ function successfully executes, coil OK is turned on.



## 6.22 SVC\_REQ 21: User-Defined Fault Logging

Use SVC\_REQ 21 to define a fault that can be displayed in the Controller Fault Table. The fault contains binary information or an ASCII message. The user-defined fault codes start at 0 hex.

The error code information for the fault must be within the range 0 to 2047 for an *Application Msg*: to be displayed. If the error code is in the range 81 to 112 decimal, the CPU sets a fault bit of the same number in %SA system memory. This allows up to 32 bits to be individually set.

<b>Error Code</b>	<b>Status Bit</b>
Errors 0—80	No bit set
Errors 81—112	Sets %SA
Errors 113—2047	No bit set
Errors 2048—32,767	Reserved

When EN is active, the fault data array referenced by IN is logged as a fault to the Controller Fault Table. If EN is not enabled, the ok bit is cleared. If the error code is out of range, the ok bit is cleared and the fault will not be logged as requested.

The parameter block is an input parameter block only with this format:

<b>Parameter address</b>	<b>Error code</b>	
	<b>MSB</b>	<b>LSB</b>
<b>Address+1</b>	Text2	Text1
<b>Address+2</b>	Text4	Text3
<b>Address+3</b>	Text6	Text5
<b>Address+4</b>	Text8	Text7
<b>Address+5</b>	Text10	Text9
<b>Address+6</b>	Text12	Text11
<b>Address+7</b>	Text14	Text13
<b>Address+8</b>	Text16	Text15
<b>Address+9</b>	Text18	Text17
<b>Address+10</b>	Text20	Text19
<b>Address+11</b>	Text22	Text21
<b>Address+12</b>	Text24	Text23

The input parameter data allows you to select an error code in the range 0 to 2047 and text information that will be placed in the fault extra data portion of a long controller fault. The controller fault address, fault group, and fault action are filled in by the function block.

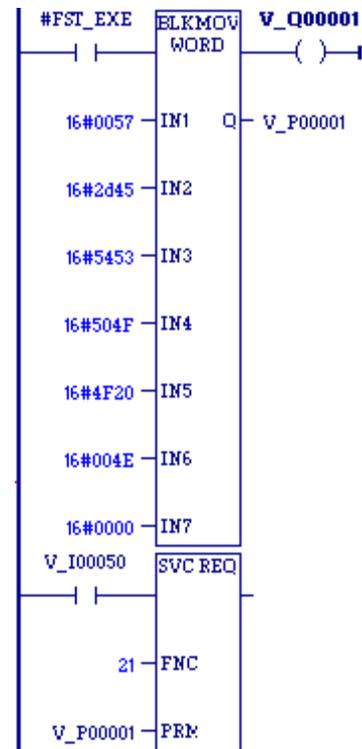
The fault text bytes 1 – 24 can be used to pass binary or ASCII data with the fault. If the first byte of the fault text data is non-zero, the data will be an ASCII message string. This message will then be displayed in the fault description area of the fault table. If the message is less than 24 characters, the ASCII string must be NULL byte-terminated. The programmer will display *Application Msg:* and the ASCII data will be displayed as a message immediately following *Application Msg:*. If the error code is between 1 and 2047, the error code number will be displayed immediately after *Msg:* in the *Application Msg:* string. (If the error code is greater than 2047, the function is ignored and its output is set to OFF.)

If the first byte of text is zero, then only *Application Msg:* will display in the fault description. The next 1-23 bytes will be considered binary data for user data logging. This data is displayed in the Controller Fault Table.

**Note:** When a user-defined fault is displayed in the Controller Fault Table, a value of -32768 (8000 hex) is added to the error code. For example, the error code 5 will be displayed as -32763.

**SVC\_REQ 21 Example**

The value passed to IN1 is the fault error code. The value passed in, 16x0057, represents an error code of 87 decimal and will appear as part of the fault message. The values of the next inputs give the ASCII codes for the text of the error message. For IN2, the input is 2D45. The low byte, 45, decodes to the letter **E** and the high byte, 2D, decodes to -. Continuing in this manner, the string continues with **S T O P O** and **N**. The final character, **00**, is the null character that terminates the string. In summary, the decoding yields the string message **E\_STOP ON**.



### 6.23 SVC\_REQ 22: Mask/Unmask Timed Interrupts

Use SVC\_REQ 22 to mask or unmask timed interrupts and to read the current mask. When the interrupts are masked, the CPU does not execute any timed interrupt block timed program that is associated with a timed interrupt. Timed interrupts are masked/unmasked as a group. They cannot be individually masked or unmasked.

Successful execution occurs unless some number other than 0 or 1 is entered as the requested operation or mask value.

The parameter block is an input and output parameter block.

To determine the current mask, use this format:

<b>Address</b>	0 = Read interrupt mask
----------------	-------------------------

The CPU returns this format:

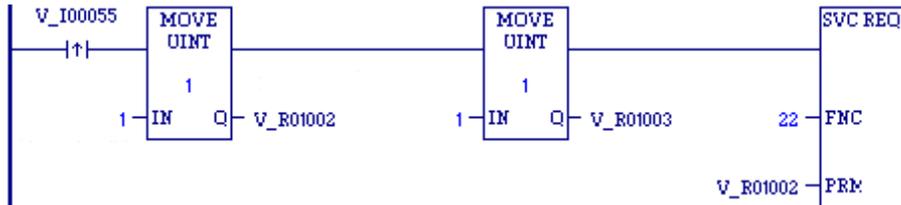
<b>Address</b>	0 = Read interrupt mask
<b>Address+1</b>	0 = Timed interrupts are unmasked 1 = Timed interrupts are masked

To change the current mask, use this format:

<b>Address</b>	1 = Mask/unmask interrupts
<b>Address+1</b>	0 = Unmask timed interrupts 1 = Mask timed interrupts

#### SVC\_REQ 22 Example

When input %I00055 transitions on, timed interrupts are masked.



## 6.24 SVC\_REQ 23: Read Master Checksum

Use SVC\_REQ 23 to read master checksums for the set of user program(s) and the configuration, and to read the checksum for the block from which the service request is made.

There is no input parameter block for this service request. The output parameter block requires 15 words of memory.

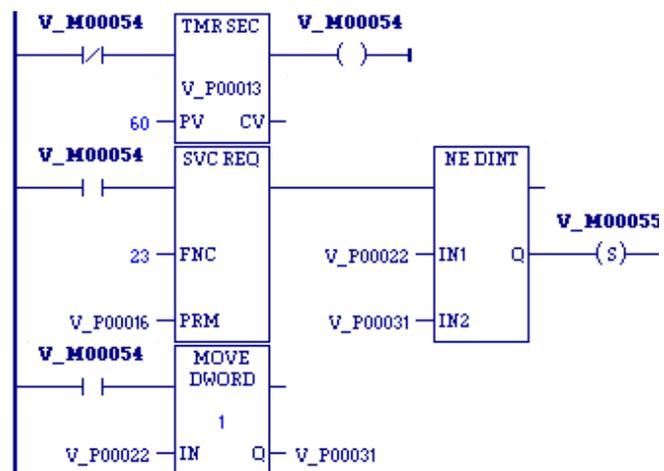
### Output

When a RUN Mode Store is active, the program checksums may not be valid until the store is complete. To determine when checksums are valid, three flags (one each for Program Block Checksum, Master Program Checksum, and Master Configuration Checksum) are provided at the beginning of the output parameter block.

Address	Description
Address	Program Checksum Valid (0 = not valid, 1 = valid)
Address + 1	Master Program Checksum Valid (0 = not valid, 1 = valid)
Address + 2	Master Configuration Checksum Valid (0 = not valid, 1 = valid)
Address + 3	Number of LD/SFC Blocks (including _MAIN)
Address + 4	Size of User Program in Bytes (DWORD data type)
Address + 6	Program Set Additive Checksum
Address + 7	Program CRC Checksum (DWORD data type)
Address + 9	Size of Configuration Data in Kbytes
Address + 10	Configuration Additive Checksum
Address + 11	Configuration CRC Checksum (DWORD data type)
Address + 13	high byte: always zero low byte: Currently Executing Block's Additive Checksum
Address + 14	Currently Executing Block's CRC Checksum

### SVC\_REQ 23 Example

When the timer using registers %P00013 through %P00015 expires, the checksum read is performed. The checksum data returns in registers %P00016 through %P00030. The master program checksum in registers %P00022 and %P00023 (the program checksum is a DWORD data type and occupies two adjacent registers) is compared with the last saved master program checksum. If these are different, coil %M00055 is latched on. The current master program checksum is then saved in registers %P00031 and %P00032.



## 6.25 SVC\_REQ 24: Reset Module

Use SVC\_REQ 24 to reset a daughterboard or some modules. Modules that support SVC\_REQ 24 include:

RX3i IC693BEM331, IC694BEM331, IC693APU300, IC694APU300, IC695ETM001, IC693ALG2222, IC694ALG2222, IC695PNC001

RX7i: Embedded Ethernet Interface module, IC697BEM731, IC698BEM731, IC697HSC700, IC697ALG230, IC698ETM001

The SVC\_REQ output is set ON unless one of the following conditions exists:

- An invalid number for rack and/or slot is entered.
- There is no module at the specified location.
- The module at the specified location does not support a runtime reset.
- The CPU was unable to reset the module at the specified location.

For this function, the parameter block has a length of 1 word. It is an input parameter block only.

<b>Address</b>	Module slot (low byte)
	Module rack (high byte)
	<i>Rack 0, Slot 1 indicates that a reset is to be sent to the daughterboard.</i>

**Note:** It is important to invoke SVC\_REQ #24 for a given module for only one sweep at a time. Each time this function executes, the target module will be reset regardless of whether it has finished starting up from a previous reset.

After sending a SVC\_REQ #24 to a module, you must wait a minimum of 5 seconds before sending another SVC\_REQ #24 to the same module. This ensures that the module has time to recover and complete its startup.

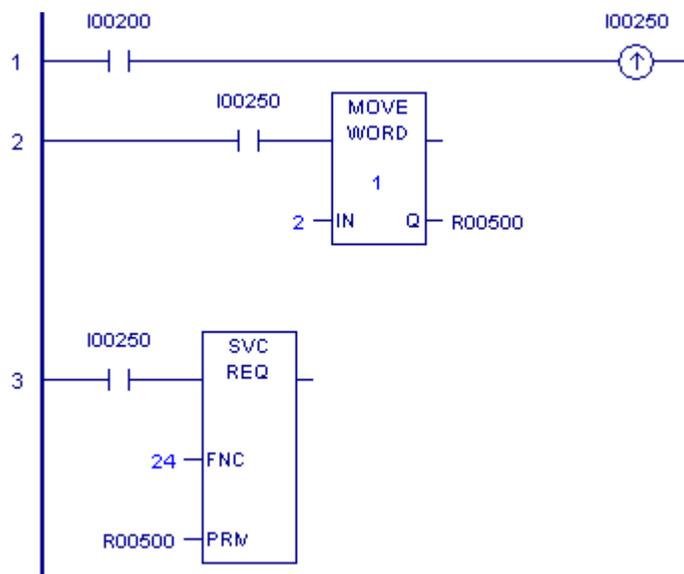
### SVC\_REQ 24 Example

This example resets the module in rack0/slot 2.

In rung 1, when contact %I00200 is closed, the positive transition coil sets %I00250 to ON for one sweep.

The MOVE\_WORD instruction in rung 2 receives power flow and moves the value 2 into %R00500.

The SVC\_REQ function in rung 3 then receives power flow and resets the module indicated by the rack/slot value in %R00500.



## 6.26 SVC\_REQ 25: Disable/Enable EXE Block and Standalone C Program Checksums

Use SVC\_REQ 25 to enable or disable the inclusion of EXE in the background checksum calculation. The default is to include the checksums.

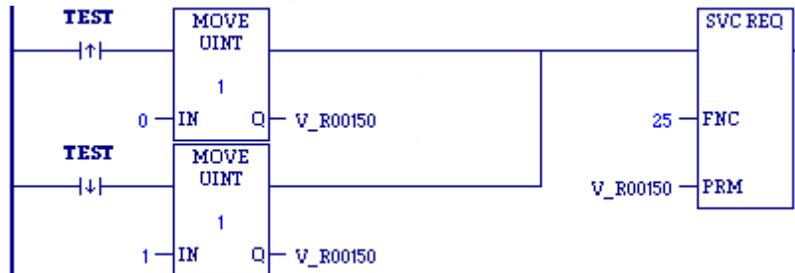
This service request uses only an input parameter block.

<b>Address</b>	0 = Disable C applications inclusion in checksum calculation
	1 = Enable C application inclusion in checksum calculation

The parameter block is unchanged after execution of the service request.

### SVC\_REQ 25 Example

When the coil TEST transitions from OFF to ON, SVC\_REQ 25 executes to disable the inclusion of EXE blocks in the background checksum calculation. When coil TEST transitions from ON to OFF, the SVC\_REQ executes to again include EXE blocks in the background checksum calculation.



## 6.27 SVC\_REQ 29: Read Elapsed Power Down Time

Use SVC\_REQ 29 to read the amount of time elapsed between the last power-down and the most recent power-up. If the watchdog timer expired before power-down, the CPU is not able to calculate the power down elapsed time, so the time is set to 0.

This service request cannot be accessed from a C block.

This function has an output parameter block only. The parameter block has a length of three words.

<b>Address</b>	Power-down elapsed seconds (low order)
<b>Address + 1</b>	Power-down elapsed seconds (high order)
<b>Address + 2</b>	100µS ticks

The first two words are the power-down elapsed time in seconds. The last word is the number of 100 µs ticks in the current second.

**Note:** Although this request responds with a resolution of 100 µs, the actual accuracy is 1 second. The battery-backed clock, which is used when the controller is powered down, is accurate to within 1 second.

### SVC\_REQ 29 Example

When input %I0251 is ON, the elapsed power-down time is placed into the parameter block that starts at %R0050. The output coil (%Q0001) is turned on.



### 6.28 SVC\_REQ 32: Suspend/Resume I/O Interrupt

Use SVC\_REQ 32 to suspend a set of I/O interrupts and cause occurrences of these interrupts to be queued until these interrupts are resumed. The number of I/O interrupts that can be queued depends on the I/O module's capabilities. The CPU informs the I/O module that its interrupts are to be suspended or resumed. The I/O module's default is resumed. The Suspend applies to all I/O interrupts associated with the I/O module. Interrupts are suspended and resumed within a single scan.

SVC\_REQ 32 uses only an input parameter block. Its length is three words.

<b>Address</b>	0 = resume interrupt 1 = suspend interrupt
<b>Address + 1</b>	memory type
<b>Address + 2</b>	reference (offset)

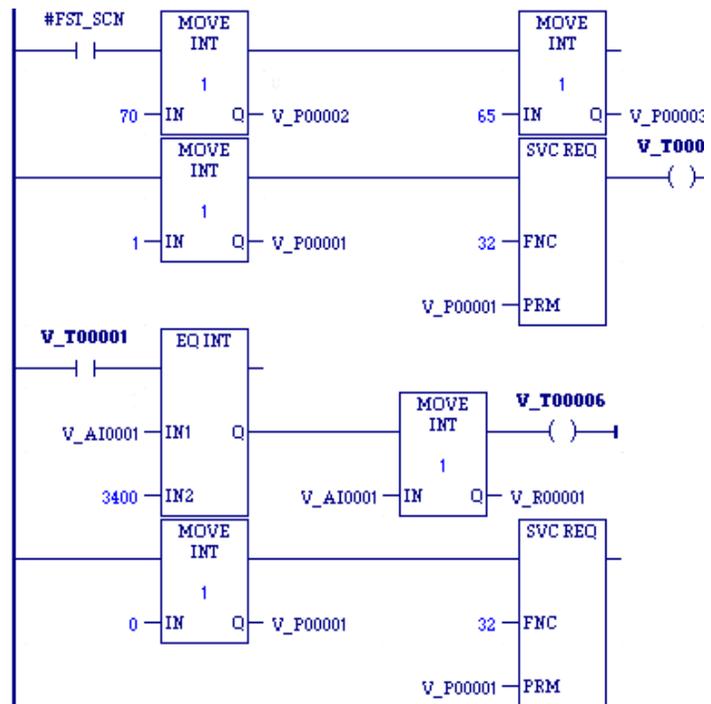
Successful execution occurs unless:

- Some number other than 0 or 1 is passed in as the first parameter.
- The memory type parameter is not 70 (%I memory).
- The I/O module associated with the specified address is not an appropriate module for this operation.
- The reference address specified is not the first %I reference for the High Speed Counter.
- Communication between the CPU and this I/O module has failed. (The board is not present, or it has experienced a fatal fault.)

**Note:** I/O interrupts, unless suspended or masked, can interrupt the execution of a function block. The most often used application of this Service Request is to prevent the effects of the interrupts for diagnostic or other purposes.

#### SVC\_REQ 32 Example

Interrupts from the high speed counter module whose starting point reference address is %I00065 will be suspended while the CPU solves the logic of the second rung. Without the Suspend, an interrupt from the HSC could occur during execution of the third rung and %T00006 could be set while %R00001 has a value other than 3,400. (%AI00001 is the first non-discrete input reference for the High Speed Counter.)



## 6.29 SVC\_REQ 45: Skip Next I/O Scan

Use the SVC\_REQ function #45 to skip the next output and input scans. Any changes to the output reference tables during the sweep in which the SVC\_REQ #45 was executed will not be reflected on the physical outputs of the corresponding modules. Any changes to the physical input data on the modules will not be reflected in the corresponding input references during the sweep after the one in which the SVC\_REQ #45 was executed.

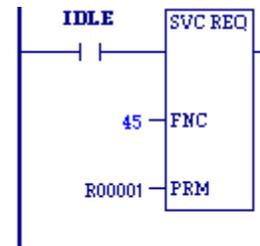
This function has no parameter block.

**Note:** This service request is provided for conversion of Series 90-30 applications. The Suspend I/O (SUS\_IO) function block, which is supported by all PACSystems firmware versions, should be used in new applications.

**Note:** The DOIO Function Block is not affected by the use of SVC\_REQ #45. It will still update the I/O when used in the same logic program as the SVC\_REQ #45.

### SVC\_REQ 45 Example

In the following LD example, when the *Idle* contact passes power flow, the next Output and Input Scan are skipped.



### 6.30 SVC\_REQ 50: Read Elapsed Time Clock

Use SVC\_REQ 50 to read the system's elapsed time clock. The elapsed time clock measures the time in seconds since the CPU was powered on. The parameter block has a length of four words used for output only.

#### Output

<b>Address</b>	Seconds from power on (low order)
<b>Address+1</b>	Seconds from power on (high order)
<b>Address+2</b>	nanosecond ticks (low order)
<b>Address+3</b>	nanosecond ticks (high order)

The first two words are the elapsed time in seconds. The second two words are the number of nanoseconds elapsed in the current second.

The resolution of the CPU's elapsed time clock is 100  $\mu$ s. The overall accuracy of the elapsed time clock is  $\pm 0.01\%$ . The accuracy of an individual sample of the elapsed time clock is approximately 105  $\mu$ s.

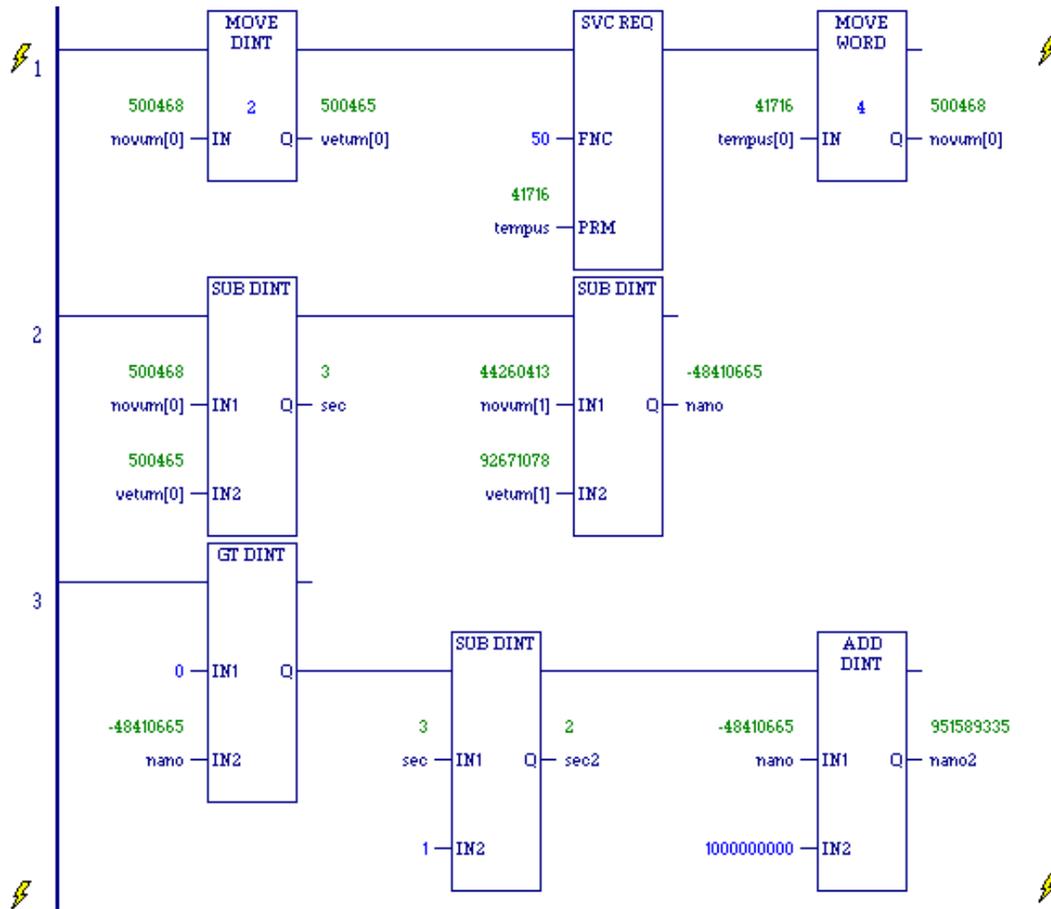


#### Warning

**The SVC\_REQ instruction is not protected against operating system and user interrupts. The timing and length of these interrupts are unpredictable. The clock sample returned by SVC\_REQ 50 can sometimes be much more than 105  $\mu$ s old by the time execution is returned to the LD logic.**

SVC\_REQ 50 Example

The following logic is used in a block that is called once in a while. The screen shot was taken between calls to the block. The second rung of logic calculates the number of seconds that have elapsed since the last time the block was called. The third rung calculates the number of nanoseconds to be added to, or subtracted from, the number of seconds. The first rung saves the previous value of novum[0] and novum[1] into vetum[0] and vetum[1] before the second rung of logic places the current time values in novum[0] and novum[1].



### 6.31 SVC\_REQ 51: Read Sweep Time from Beginning of Sweep

Use SVC\_REQ 51 to read the time in nanoseconds since the start of the sweep. The data is unsigned 32-bit integer.

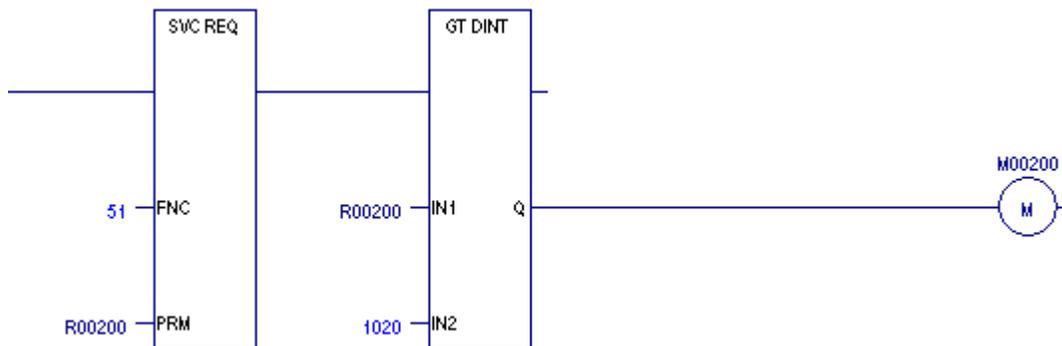
#### Output

The parameter block is an output parameter block only; it has a length of two words.

<b>Address</b>	time (nanoseconds) since start of scan – low order
<b>Address+1</b>	time (nanoseconds) since start of scan – high order

#### SVC\_REQ 51 Example

The elapsed time from the start of the scan is read into locations %R00200 and %R00201 if it is greater than 10,020ns, internal coil %M0200 is turned on.



## 6.32 SVC\_REQ 56: Logic Driven Read of Nonvolatile Storage

PACSystems controllers support a 64KB nonvolatile flash memory area, which can be accessed by the logic-driven read/write service requests. Values are stored in the nonvolatile storage area using *SVC\_REQ 57: Logic Driven Write to Nonvolatile Storage*. These values are applied to the controller user memory on power-up.

If you want only to write to nonvolatile storage and have the values restored on a power cycle, you may not need to use SVC\_REQ 56. However, a logic driven read from nonvolatile storage can be commanded as needed. For example, you can use #FST\_SCN with SVC\_REQ 56 calls to force a reload on each STOP Mode to RUN Mode transition.

SVC\_REQ 56 specifies a read operation from nonvolatile storage when the PACSystems is running. You can specify which reference address range to read and optionally a different destination memory location in CPU memory in which to place the read data. Using different memory locations enables you to set up a comparison between existing values in CPU memory with values in nonvolatile storage.

SVC\_REQ 56 execution time will vary depending on the number of values stored in nonvolatile storage, as it will find the most recent value for the requested reference address range.

You can read up to 32 words (64 bytes) inclusively per invocation of SVC\_REQ 56.

### 6.32.1 Discrete Memory

Discrete memory can be read as individual bits or as bytes. For more information, refer to *Memory Type Codes* below.

If a discrete memory destination is forced, the forced value remains intact in CPU memory even though the count in word 10 (address + 10) indicates that all the data was read and transferred.

If a memory location has an associated transition bit and SVC\_REQ 56 causes a transition on that value, the transition bit is set.

### 6.32.2 Storage Disabled Conditions

By default, the following write operations disable SVC\_REQ 56 until logic is written to nonvolatile storage:

- RUN Mode Store (RMS), even if a second RMS reverts everything to the original state.
- Test-Edit session, even when you cancel your edits.
- Word-for-word change.
- Downloading to RAM only of a stopped PACSystems CPU, even if the downloaded contents are equal to the contents already on the nonvolatile storage. Setting bit 0 of input word 8 (address + 7) to a value of 1 enables SVC\_REQ 56 despite the above conditions.

### 6.32.3 Maximum of One Active Instruction

When SVC\_REQ 56 is active, it does not support an interrupt that attempts to activate SVC\_REQ 57 or a second instance of SVC\_REQ 56. If an attempt fails, an error indicating that another instance is active will be returned.

### 6.32.4 ENO and Power Flow To The Right

If the status is Success or Partial Read (see address+9), on the SVC\_REQ instruction, ENO is set to True in FBD and ST, and power flow passes to the right in LD.

## 6.32.5 Parameter Block

<b>Address+0</b>	Memory type. Refer to <i>Memory Type Codes</i> below.								
<b>Address+1</b>	<p>The zero-based offset N to read from nonvolatile storage. Contains the complete offset for any memory area except %W, which also requires the use of address + 2 for offsets greater than 65,535.</p> <ul style="list-style-type: none"> <li>For %I, %Q, %M, %T, and %G memory in byte mode, <math>N = (Ra - 1) / 8</math>, where Ra = one-based reference address. For example, to read from the one-based bit reference address %T33, enter the byte offset 4: <math>(33 - 1) / 8 = 4</math>.</li> <li>For %W, %R, %AI, and %AQ memory, and for %I, %Q, %M, %T, and %G memory in bit mode, <math>N = Ra - 1</math>. For example, to read from the one-based reference address %R200, enter the zero-based reference offset 199; to read from %I73 in bit mode, enter offset 72. For memory in bit mode, the offset must be set on a byte boundary, that is, a number exactly divisible by 8: 0, 8, 16, 24, and so on.</li> </ul>								
<b>Address+2</b>									
<b>Address+3</b>	<p>Length. The number of items to read from nonvolatile storage beginning at the reference address calculated from the offset defined at [address + 1 and address + 2]. The length can be one of the following:</p> <table border="1"> <thead> <tr> <th>Description</th> <th>Valid range</th> </tr> </thead> <tbody> <tr> <td>The number of words (16-bit registers) to read from %W, %R, %AI, or %AQ nonvolatile storage</td> <td>1 through 32 words</td> </tr> <tr> <td>The number of bytes to read from %I, %Q, %M, %T, or %G in byte mode nonvolatile storage</td> <td>1 through 64 bytes</td> </tr> <tr> <td>The number of bits to read from %I, %Q, %M, %T, or %G in bit mode nonvolatile storage</td> <td>1 through 512 bits in increments of 8 bits</td> </tr> </tbody> </table> <p>The value must reside in the low byte of address + 3. The high byte must be set to zero.</p>	Description	Valid range	The number of words (16-bit registers) to read from %W, %R, %AI, or %AQ nonvolatile storage	1 through 32 words	The number of bytes to read from %I, %Q, %M, %T, or %G in byte mode nonvolatile storage	1 through 64 bytes	The number of bits to read from %I, %Q, %M, %T, or %G in bit mode nonvolatile storage	1 through 512 bits in increments of 8 bits
Description	Valid range								
The number of words (16-bit registers) to read from %W, %R, %AI, or %AQ nonvolatile storage	1 through 32 words								
The number of bytes to read from %I, %Q, %M, %T, or %G in byte mode nonvolatile storage	1 through 64 bytes								
The number of bits to read from %I, %Q, %M, %T, or %G in bit mode nonvolatile storage	1 through 512 bits in increments of 8 bits								
<b>Address + 4</b>	Destination memory. The CPU memory area to write the read data to. This does not need to be the same memory area as specified at [address]. Writing to a different memory area enables you to compare the values that were already in the CPU with the values read from nonvolatile storage.								
<b>Address+5</b>	<p>The zero-based offset N in CPU memory to start writing the read data to. Address + 5, the least significant word, contains the complete offset for any memory area except %W, which also requires the use of address + 6 for offsets greater than 65,535.</p> <ul style="list-style-type: none"> <li>For %I, %Q, %M, %T, and %G memory in byte mode, <math>N = (Ra - 1) / 8</math>, where Ra = one-based reference address. For example, to write to the one-based bit reference address %T33, enter the byte offset 4: <math>(33 - 1) / 8 = 4</math>.</li> <li>For %W, %R, %AI, and %AQ memory, and for %I, %Q, %M, %T, and %G memory in bit mode, <math>N = Ra - 1</math>. For example, to write to the one-based reference address %R200, enter the zero-based reference offset 199; to write to %I73 in bit mode, enter offset 72.</li> </ul>								
<b>Address+6</b>									
<b>Address+7</b>	<ul style="list-style-type: none"> <li>When bit 0 is set to 1, storage disabled conditions are ignored. A read is allowed even if the logic in RAM has changed since nonvolatile storage was read or written.</li> <li>Bits 1 through 15 must be set to zero; otherwise, the read fails.</li> </ul>								
<b>Address+8</b>	Reserved. Must be set to zero; otherwise, the read fails.								
<b>Address+9</b>	Response status. The status read from nonvolatile storage. The low byte contains the major error code; the high byte contains the minor error code. For definitions, refer to <i>Response Status Codes for SVC_REQ 56</i> .								
<b>Address+10</b>	Response Count. The number of words, bytes, or bits copied.								

## Memory Type Codes

<b>Type</b>	<b>Decimal Value</b>	<b>Type</b>	<b>Decimal Value</b>
%R	8	%G (byte mode)	56
%AI	10	%I (bit mode)	70
%AQ	12	%Q (bit mode)	72
%I (byte mode)	16	%T (bit mode)	74
%Q (byte mode)	18	%M (bit mode)	76
%T (byte mode)	20	%G (bit mode)	86
%M (byte mode)	22	%W	196

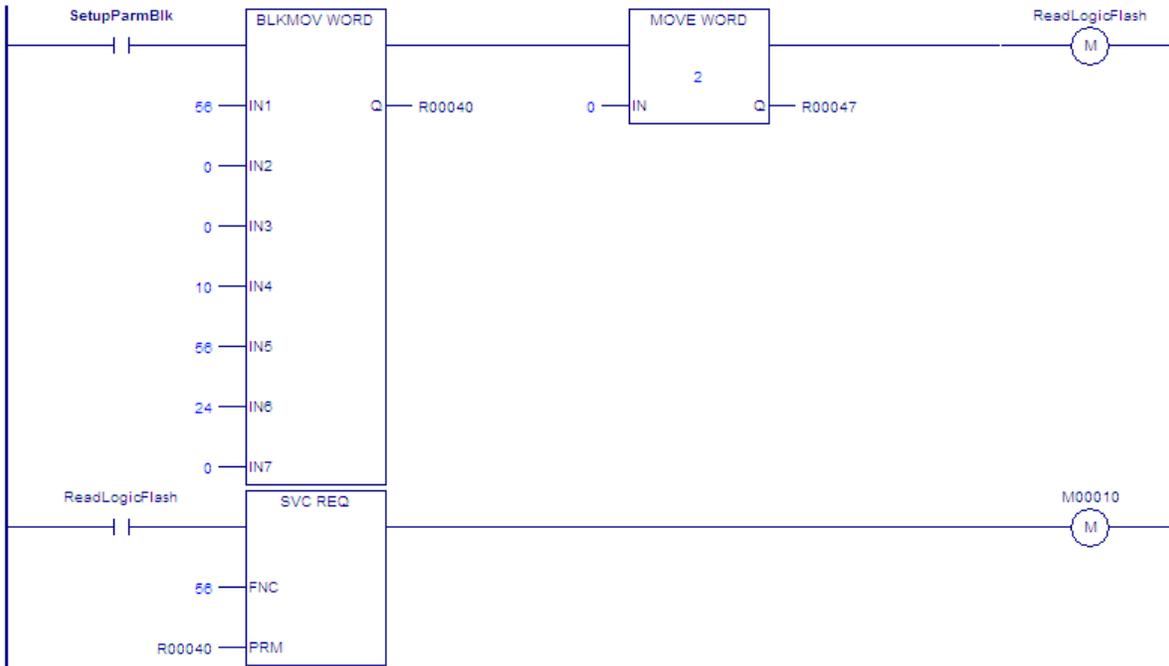
## Response Status Codes for SVC\_REQ 56

<b>Minor</b>	<b>Major</b>	<b>Description</b>
00	01	Success. All values requested were found and copied.
01	01	Partial Read. All values found were copied, but some or all values were not in storage.
01	02	Insufficient Destination Memory. The Destination memory location is not large enough to store the requested values.
02	02	Invalid Length. The length requested is larger than 64 bytes or less than 1 byte or the number of bits is not an exact multiple of 8.
03	02	Invalid storage or destination reference address. A specified memory area is not %, %Q, %T, %M, %G, %R, %AI, %AQ, or %W, or the offset is out of range, or the offset is not byte-aligned for discrete memory in bit mode.
04	02	Invalid request. Spare bits or spare words in parameter block are not set to zero.
01	03	Storage Busy. A SVC_REQ 57 or another SVC_REQ 56 instruction is active. For example, an interrupt block is attempting to execute SVC_REQ 56 when the block it interrupted was executing SVC_REQ 56.
01	04	Storage Disabled. The logic in RAM differs from the logic in nonvolatile storage. See Storage disabled conditions.
02	04	Storage Closed. Either the storage has not been created or a previous corruption error or unexpected read/write failure closed the storage.
01	05	Unexpected Read Failure. A command to the storage hardware failed unexpectedly.
02	05	Corrupted storage. A corrupted checksum or storage header caused a read to fail.

**SVC\_REQ 56 Example**

The following LD logic reads ten continuous bytes written to nonvolatile storage from %G1—%G80 into %G193—%G273. The value applied to IN1, 56, selects byte mode.

The parameter block starts at %R00040. The response words are returned to %R00049 and %R00050.



**Parameter Block for SVC\_REQ 56 Example**

Address + Offset	Address	Input Value	Definition
Address+0	%R00040	56	Data type = %G (byte mode)
Address+1	%R00041	0	Address written from, low word
Address+2	%R00042	0	Address written from, high word
Address+3	%R00043	10	Length = 10 bytes
Address+4	%R00044	56	Data type to write to = %G (byte mode)
Address+5	%R00045	24	Address to write to, low word
Address+6	%R00046	0	Address to write to, high word
Address+7	%R00047	0	Storage disabled conditions are enforced
Address+8	%R00048	0	Reserved, must be set to 0
Address+9	%R00049	NA	Response status.
Address+10	%R00050	NA	Response count.

## 6.33 SVC\_REQ 57: Logic Driven Write to Nonvolatile Storage

PACSystems controllers support a 65,500 byte nonvolatile flash memory area that can be accessed by the logic-driven read/write service requests. Values are stored in the nonvolatile storage area using SVC\_REQ 57. These values are applied to the controller user memory on power up.

SVC\_REQ 57 specifies a range of reference addresses to read from a running PACSystems CPU and write to nonvolatile storage. This feature is intended to retain a limited set of values, such as set points or tuning parameters that need to change when the PACSystems is running.

This feature uses 65,536 bytes of nonvolatile storage. But not all of this memory is available for the actual data being written by the service request. Some of the memory is used internally by the controller to maintain information about the data being stored.

**Note:** Nonvolatile storage is intended for storing values that do not change frequently. Once the nonvolatile storage area fills up, a power cycle or STOP Mode Store is required to store more values. The logic-driven write is not a replacement for battery backed RAM for values that change frequently or during every sweep. (Refer to *When nonvolatile storage is full* below.)

### 6.33.1 Length of Data Written

SVC\_REQ 57 scans the nonvolatile storage to find the most recent values stored for the specified range. If it finds no values for the range or the most recent stored values are different, the new values are written to nonvolatile storage.

SVC\_REQ 57 reports the length of data written in word 8 (*starting address + 7*) of the parameter block. The number of words written is calculated from the first word that changed to the end of the array. For example, if you specify 8 words to be written, but only the values of words 3 and 4 are changed, the SVC\_REQ identifies the first mismatch at word 3 and writes the values of words 3 through 8 (a length of 6 words).

You can write up to 32 words (64 bytes) inclusively per invocation of SVC\_REQ 57. Each invocation requires 4 words of command data (8 bytes). A 1-byte write requires 9 bytes whereas a 64-byte write requires 72 bytes. You can generally make the most efficient use of nonvolatile storage by transferring data in 56-byte increments, since this will actually write 64 bytes to the device. Given the bookkeeping overhead required by the Controller and possible fragmentation, at least 54,912 bytes and no more than 64,000 bytes will be available for the reference data and the 8 bytes of command data for each invocation. For additional information, refer to *Fragmentation* below.

### 6.33.2 Write Frequency

Multiple calls to SVC\_REQ 57 in a single sweep may cause CPU watchdog timeouts. The number of calls to SVC\_REQ 57 that can be made requires consideration of many variables: the software watchdog timeout value, how much data is being written, how long the sweep is, age of nonvolatile storage (flash), etc. If the application attempts to write to flash too frequently, the CPU could experience a watchdog timeout while waiting for a preceding write operation to complete.

The Logic Driven Read/Write to Flash service requests are not intended for high frequency use. We recommend limiting the number of calls to SVC\_REQ 57 to one call per sweep to avoid the potential of causing a watchdog timeout and the resulting transition to STOP-Halt mode.

### 6.33.3 Erase Cycles

The flash component on the PACSystems CPU is rated for 100K erase cycles. Erase cycles occur under the following conditions:

- Write to flash is commanded from the programmer.
- Clear flash operation.
- Flash compaction after a power cycle when flash memory allotted for SVC\_REQ 57 has become full.

### 6.33.4 Discrete Memory

Discrete memory can be written to as individual bits or as bytes. For more information, see Address. Force and transition information is not written to nonvolatile storage.

### 6.33.5 Retentiveness

Writing values to nonvolatile storage for non-retentive memory such as %T does not make the memory retentive. For example, all values stored to %T memory are set to zero on power-up or a STOP Mode to RUN Mode transition. You can, however, read such values from storage after power-up or STOP Mode to RUN Mode transition by using SVC\_REQ 56.

### 6.33.6 Maximum of One Active Instruction

When SVC\_REQ 57 is active, it does not support an interrupt that attempts to activate SVC\_REQ 56 or a second instance of SVC\_REQ 57.

### 6.33.7 Storage Disabled Conditions

By default, the following write operations disable SVC\_REQ 57 until logic is written to nonvolatile storage:

- RUN Mode Store (RMS), even if a second RMS reverts everything to the original state
- Test-Edit session, even when you cancel your edits
- Word-for-word change
- Downloading to RAM only of a stopped PACSystems CPU, even if the downloaded contents are equal to the contents already on the nonvolatile storage

Setting bit 0 of input word 4 (address + 4) to a value of 1 enables SVC\_REQ 57 despite the above conditions.

### 6.33.8 Error Checking

When writing to nonvolatile storage, error checking is provided to ensure that logic and the Hardware Configuration (HWC) in nonvolatile memory match the logic and HWC in PACSystems RAM.

### 6.33.9 Fragmentation

Due to the nature of the media in PACSystems CPUs, writes may produce fragmentation of the memory. That is, small portions of the memory may become unavailable, depending upon the sequence of the writes and the size of each one. Data is stored on the device in 128 512-byte sections. Each section uses 12 bytes of bookkeeping information, leaving a maximum of 64,000 bytes devoted to the reference data and command data for each invocation. However, the data for a single invocation cannot be split across sections. So, if there is insufficient space in the currently used section to contain the new data, the unused portion of that section becomes lost.

**Example:** Suppose that the current operation is writing 64 bytes of reference data and 8 bytes of command data (72 bytes total). If there are only 71 bytes remaining in the current section, the new data will be written to a new section and the unused 71 bytes in the old section become unavailable.

### 6.33.10 When nonvolatile storage is full

When logic driven user nonvolatile storage is full, a fault is logged. Before you can use SVC\_REQ 57 to write again, use one of the following solutions:

**To retain the most up-to-date data and continue writing with SVC\_REQ 57 to nonvolatile storage:**

1. Stop the PACSystems.
2. Power cycle the PACSystems.

A power cycle when nonvolatile storage is full triggers a compaction of existing data. During compaction, multiple writes of the same reference memory address are removed, which leaves only the most recent data, and contiguous reference memory addresses are combined into the fewest number of records necessary.

If compaction cannot take place, a second fault is logged and you need to use one of the following two solutions.

**To retain specific data from nonvolatile storage, clear nonvolatile storage, and then return the data to nonvolatile storage:**

1. While the controller is still running, use SVC\_REQ 56 to read the desired values into PACSystems memory.
2. Upload the current values from controller memory as initial values to your project.
3. Stop the controller.
4. Do one of the following:  
Clear the flash memory, or  
Write to flash. The flash is erased prior to writing, which frees up some space.
5. Download the initial values to the controller.
6. Start the controller.
7. Use SVC\_REQ 57 to write the desired values from controller memory to nonvolatile storage.

**To write to flash to erase everything:**

1. Stop the Controller.
2. Write to flash. The flash is erased prior to writing, which frees up some space.

### **6.33.11 Equality**

Because data in nonvolatile storage is not considered part of the project, writing to nonvolatile storage does not impact equality between the CPU and Logic Developer.

### **6.33.12 Redundancy**

Redundancy systems can benefit from the use of logic driven user nonvolatile storage as long as all of the references saved to nonvolatile storage are included in the transfer lists. Each redundancy CPU maintains its own separate logic driven user nonvolatile storage by means of SVC\_REQ 57 during its logic scan. If the values of reference addresses to be stored to user nonvolatile storage are synchronized, the logic driven user nonvolatile storage data in each CPU is identical. If the values to be stored are not synchronized, then each CPU's user nonvolatile storage may be different.

### **6.33.13 ENO and Power Flow to the Right**

If the status is Success or Partial Read, then on the SVC\_REQ instruction, ENO is set to True in FBD and ST, and power flow passes to the right in LD.

## 6.33.14 Parameter Block for SVC\_REQ 57

<b>Address+0</b>	Memory type. Refer to <i>Memory Type Codes</i> above.								
<b>Address+1</b>	<p>The zero-based offset N to write to nonvolatile storage. Contains the complete offset for any memory area except %W, which also requires the use of address + 2 for offsets greater than 65,535.</p> <ul style="list-style-type: none"> <li>▪ For %I, %Q, %M, %T, and %G memory in byte mode, <math>N = (Ra - 1) / 8</math>, where Ra = one-based reference address. For example, to read from the one-based bit reference address %T33, enter the byte offset 4: <math>(33 - 1) / 8 = 4</math>.</li> <li>▪ For %W, %R, %AI, and %AQ memory, and for %I, %Q, %M, %T, and %G memory in bit mode, <math>N = Ra - 1</math>. For example, to read from the one-based reference address %R200, enter the zero-based reference offset 199; to read from %I73 in bit mode, enter offset 72. For memory-in-bit mode, the offset must be set on a byte boundary, that is, a number exactly divisible by 8: 0, 8, 16, 24, and so on.</li> </ul>								
<b>Address+2</b>									
<b>Address+3</b>	<p>Length. The number of items to write to nonvolatile storage beginning at the reference address calculated from the offset defined at [address + 1 and address + 2]. The length can be one of the following:</p> <table border="1" style="margin-left: 40px;"> <thead> <tr> <th style="text-align: left;">Description</th> <th style="text-align: left;">Valid range</th> </tr> </thead> <tbody> <tr> <td>The number of words (16-bit registers) to read from %W, %R, %AI, or %AQ nonvolatile storage</td> <td>1 through 32 words</td> </tr> <tr> <td>The number of bytes to read from %I, %Q, %M, %T, or %G in byte mode nonvolatile storage</td> <td>1 through 64 bytes</td> </tr> <tr> <td>The number of bits to read from %I, %Q, %M, %T, or %G in bit mode nonvolatile storage</td> <td>1 through 512 bits in increments of 8 bits</td> </tr> </tbody> </table> <p>The value must reside in the low byte of address + 3. The high byte must be set to zero.</p>	Description	Valid range	The number of words (16-bit registers) to read from %W, %R, %AI, or %AQ nonvolatile storage	1 through 32 words	The number of bytes to read from %I, %Q, %M, %T, or %G in byte mode nonvolatile storage	1 through 64 bytes	The number of bits to read from %I, %Q, %M, %T, or %G in bit mode nonvolatile storage	1 through 512 bits in increments of 8 bits
Description	Valid range								
The number of words (16-bit registers) to read from %W, %R, %AI, or %AQ nonvolatile storage	1 through 32 words								
The number of bytes to read from %I, %Q, %M, %T, or %G in byte mode nonvolatile storage	1 through 64 bytes								
The number of bits to read from %I, %Q, %M, %T, or %G in bit mode nonvolatile storage	1 through 512 bits in increments of 8 bits								
<b>Address + 4</b>	<p>When bit 0 is set to 1, <i>Storage Disabled Conditions</i> are ignored. A write is allowed even if the logic in RAM has changed since nonvolatile storage was read or written.</p> <p>Bits 1 through 15 must be set to zero; otherwise, the write fails.</p>								
<b>Address+5</b>	Reserved. Value must be set to zero.								
<b>Address+6</b>	Response status. The low byte contains the major error code; the high byte contains the minor error code.								
<b>Address+7</b>	Count of items written: Words, bytes or bits. Calculated from the first word that changed to the end of the array.								
<b>Address+8</b>	The number of bytes available in nonvolatile storage.								
<b>Address+9</b>									
<b>Address+10</b>	Reserved.								
<b>Address+11</b>									

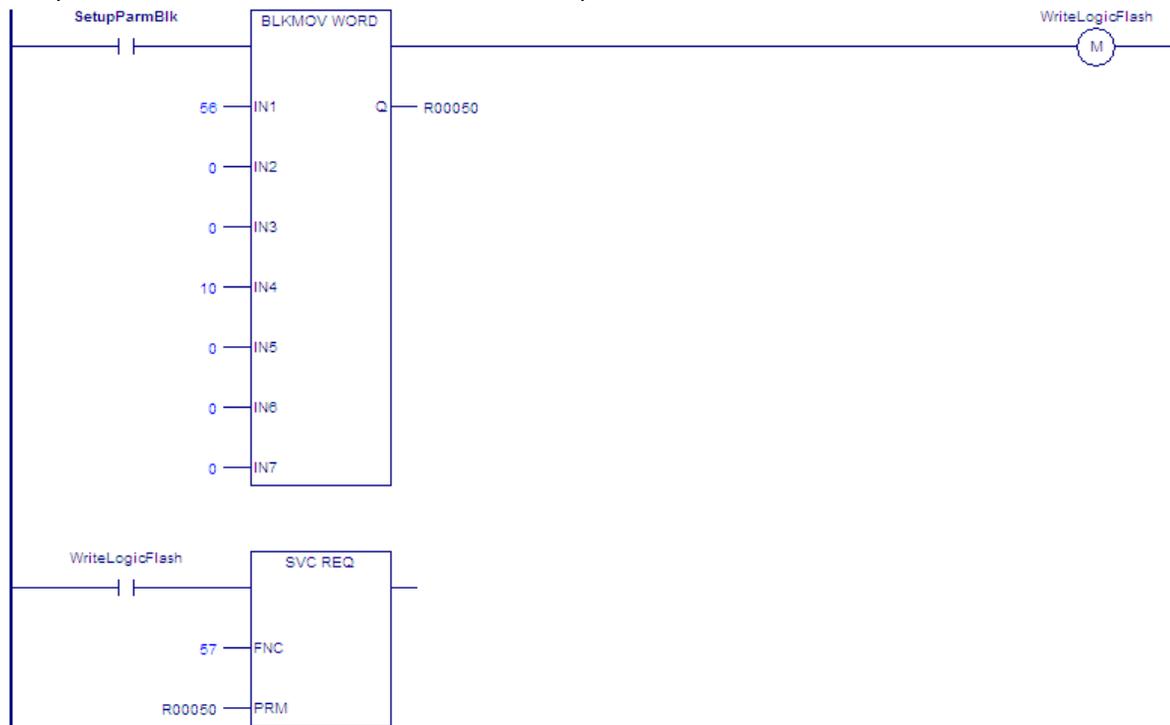
## Response Status Codes for SVC\_REQ 57

<b>Minor</b>	<b>Major</b>	<b>Description</b>
00	01	Success. All values requested were written.
01	01	Existing values found. All values requested are in storage, but one or more values were already stored.
01	02	Insufficient source memory. Counting from the offset, not enough reference addresses are left in the specified memory area.
02	02	Invalid length. The length requested was larger than 64 bytes or less than 1 byte or the number of bits is not divisible by 8.
03	02	Invalid source reference address. The memory area specified is not supported, the starting or ending offset is out of range, or the offset is not byte-aligned for discrete memory areas.
04	02	Invalid request. Spare bits or spare words in the parameter block are not set to zero.
01	03	Storage busy. A SVC_REQ 56 or another SVC_REQ 57 instruction is active. For example, an interrupt block is attempting to execute SVC_REQ 57 when the block it interrupted was executing SVC_REQ 57.
01	04	Storage disabled. The logic in RAM differs from the logic stored in nonvolatile storage. Refer to <i>Storage Disabled Conditions</i> above,
02	04	Storage closed. Either the storage has not been created or a previous corruption error or unexpected read/write failure closed the storage.
01	05	Unexpected write failure. The command to the storage hardware failed unexpectedly.
02	05	Corrupted storage. The write failed due to a bad checksum or corrupted storage header information.
01	06	Write failed. Storage is full.

**SVC\_REQ 57 Example**

The following LD logic writes ten continuous bytes to nonvolatile storage, ranging from %G1 through %G80. The value applied to IN1, 56, determines byte mode.

The parameter block starts at %R00050. The response words are returned to %R00056—%R00059.



**Parameter Block for SVC\_REQ 57 Example**

Address + Offset	Address	Input Value	Definition
Address+0	%R00050	56	Data type = %G (byte mode)
Address+1	%R00051	0	Address written from, low word
Address+2	%R00052	0	Address written from, high word
Address+3	%R00053	10	Length = 10 bytes
Address+4	%R00054	0	Storage disabled conditions are enforced
Address+5	%R00055	0	Reserved, must be set to 0
Address+6	%R00056	NA	Response status. The low byte contains the major error code; the high byte contains the minor error code.
Address+7	%R00057	NA	Count of items written: Words, bytes or bits.
Address+8	%R00058	NA	The number of bytes available in nonvolatile storage.
Address+9	%R00059	NA	
Address+10	%R00060	NA	Reserved
Address+11	%R00061	NA	Reserved





## Chapter 7 PID Built-In Function Block

This chapter describes the PID (Proportional plus Integral plus Derivative) built-in function block, which is used for closed-loop process control. The PID function compares feedback from a process variable (PV) with a desired process set point (SP) and updates a control variable (CV) based on the error.

The PID function uses PID loop gains and other parameters stored in a 40-word reference array of 16-bit integer words to solve the PID algorithm at the desired time interval.

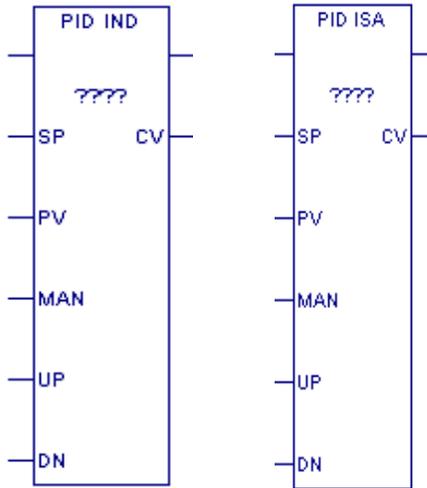


Figure 14: PID in Ladder Diagram

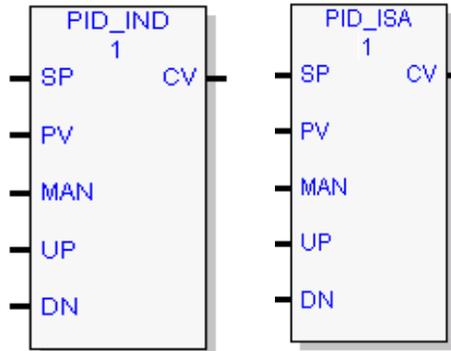
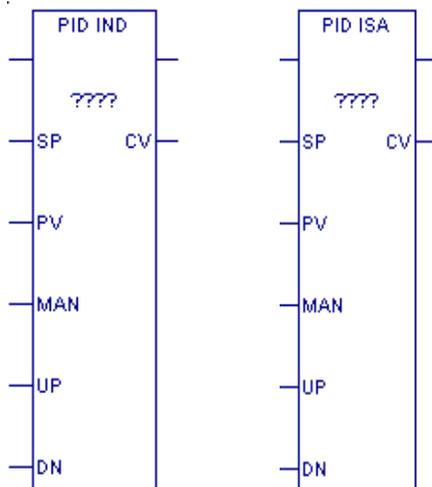


Figure 15: PID in Function Block Diagram

This chapter presents the following topics:

- *Operands of the PID Function*
- *Reference Array for the PID Function*
- *Operation of the PID Function*
- *PID Algorithm Selection (PIDISA or PIDIND) and Gain Calculations*
- *Determining the Process Characteristics*
- *Setting Tuning Loop Gains*
- *PID Example*

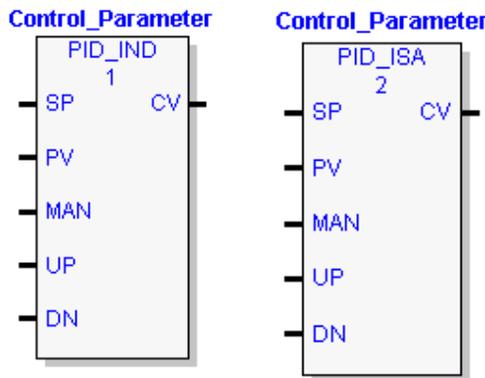
## 7.1 Operands of the PID Function



### 7.1.1 Operands for LD Version of PID Function Block

Parameter	Description	Allowed Types	Allowed Operands	Optional
(????)	Instance Variable name of the PID Parameter Block array, which contains user-configurable and internal parameters, described in <i>Reference Array for the PID Function</i> . Uses 40 words that cannot be shared.	WORD	R, L, P, W and symbolic	No
SP	The control loop or process set point. Set using process variable counts, the PID function adjusts the output control variable so that the process variable matches the set point (zero error).	INT, BOOL array of length 16 or more, Constant	All except S, SA, SB, and SC	No
PV	Process Variable input from the process being controlled. Often a %AI input.	INT, BOOL array of length 16 or more	All except S, SA, SB, and SC, and constant	No
MAN	While Power Flow is received, the PID function block is held in manual mode. If no Power Flow is received the PID function block is in Auto mode.	Power Flow	NA	No
UP	While Power Flow is received, the Manual Command is increased by 1 each user configured Sample Period.	Power Flow	NA	No
DN	While Power Flow is received, the Manual Command is decreased by 1 each user configured Sample Period.	Power Flow	NA	No
CV	The control variable output to the process. Often a %AQ output.	INT, BOOL array of length 16 or more	All except %S and constant	No

### 7.1.2 Operands for FBD Version of PID Function Block



Parameter	Description	Allowed Types	Allowed Operands	Optional
Control Structure Variable	Instance Variable name of the PID Parameter Block array, which contains user-configurable and internal parameters, described in <i>Reference Array for the PID Function</i> . Uses 40 words that cannot be shared.	WORD	R, L, P, W and symbolic	No
Function block solve order – FBD version	Calculated by the FBD editor. Can be changed by the user.	NA	NA	No
SP	The control loop or process set point. Set using process variable counts, the PID function adjusts the output control variable so that the process variable matches the set point (zero error).	INT, BOOL array of length 16 or more, Constant	All except S, SA, SB, and SC	No
PV	Process Variable input from the process being controlled. Often a %AI input.	INT, BOOL array of length 16 or more	All except S, SA, SB, SC and constant	No
MAN	When energized to 1 (through a contact), the PID function block is in <i>manual</i> mode. If this input is 0, the PID block is in <i>automatic</i> mode.	BOOL, Power Flow	All	No
UP	If energized along with MAN, increases the control variable by 1 CV count per solution of the PID function block.	BOOL, Power Flow	All	No
DN	If energized along with MAN, decreases the control variable by 1 CV count per solution of the PID function block.	BOOL, Power Flow	All	No
CV	The control variable output to the process. Often a %AQ output.	INT, BOOL array of length 16 or more	All except %S and constant	No

## 7.2 Reference Array for the PID Function

This parameter block for the PID function occupies 40 words of memory, located at the starting Instance Variable specified in the PID function block operands. Some of the words are configurable. Other words are used by the CPU for internal PID storage and are normally not changed.

Every PID function call must use a different 40-word memory area, even if all the configurable parameters are the same.

The configurable words of the reference array must be specified before executing the PID function. Zeros can be used for most default values. Once suitable PID values have been chosen, they can be defined as constants in BLKMOV functions so the program can set and change them as needed.

The LD version of the PID function does not pass power flow if there is an error in the configurable parameters. The function can be monitored using a temporary coil while modifying data.

### 7.2.1 Scaling Input and Outputs

All parameters of the PID function are 16-bit integer words for compatibility with 16-bit analog process variables. Some parameters must be defined in either PV counts or units or in CV counts or units.

The SP input must be scaled over the same range as the PV, because the PID function calculates error by subtracting these two inputs.

The process PV and control CV counts do not have to use the same scaling. Either may be -32,000 or 0 to 32,000 to match analog scaling, or from 0 to 10,000 to display variables as 0.00% to 100.00%. If the process PV and control CV do not use the same scaling, scale factors are included in the PID gains.

## 7.2.2 Reference Array Parameters

**Note:** Machine Edition software allows you to modify the configurable parameters for a PID instruction in real time in online programmer mode. To customize PID parameters, right click the PID function and select Tuning.

Words	Parameter/Description	Low Bit Units	Range
<b>1</b> <b>(Address+0)</b>	<b>Loop Number</b> Optional number of the PID block. It provides a common identification in the CPU with the loop number defined by an operator interface device.	Integer	0 to 255 (for user display only)
<b>2</b> <b>(Address+1)</b>	<b>Algorithm</b> 1 = ISA algorithm 2 = Independent algorithm	-	Set by the CPU
<b>3</b> <b>(Address+2)</b>	<b>Sample Period</b> The shortest time, in 10ms. Increments, between solutions of the PID algorithm. For example, use a 10 for a 100ms. Sample period. Minimum time of 10ms is enforced by the block if the sweep < 10ms)	10ms.	0 (every sweep) to 65535 (10.9 Min) At least 10ms.
<b>4,5</b> <b>(Address+3, Address+4)</b>	<b>Dead Band +</b> <b>Dead Band -</b> Integral values defining the upper (+) and lower (-) Dead Band limits. If no Dead Band is required, these values must be 0. If the PID Error (SP - PV) or (PV - SP) is above the (-) value and below the (+) value, the PID calculations are solved with an Error of 0. If non-zero, the (+) value must be greater than 0 and the (-) value less than 0 or the PID block will not function.  Leave these at 0 until the PID loop gains are set up or tuned. A Dead Band might be added to avoid small CV output changes due to variations in error.	PV Counts	Dead Band +: 0 to 32767 (never negative) Dead Band -: -32768 to 0 (never positive)
<b>6</b> <b>(Address+5)</b>	<b>PID_IND: Proportional Gain (Kp)</b> <b>PID_ISA: Controller gain (Kc = Kp)</b> PID_IND: Change in the control variable in CV Counts for a 100 PV Count change in the Error term. Entered as an integer representing a fixed-point decimal ratio with two decimal places. Displayed as a ratio of percentages with two decimal places.  For example, a Kp entered as 450 is displayed as 4.50 and results in a $K_p * \text{Error} / 100$ or $450 * \text{Error} / 100$ contribution to the PID Output. PID_ISA: Same as PID_IND. Kp is generally the first gain set when adjusting a PID loop.	%CV/%PV	0 to 327.67%

Words	Parameter/Description	Low Bit Units	Range
<b>7</b> <b>(Address+6)</b>	<p><b>PID_IND: Derivative Gain (Kd)</b>  <b>PID_ISA: Derivative Time (Td = Kd)</b></p> <p>PID_IND: Change in the control variable in CV Counts if the Error or PV changes 1 PV Count every 10ms. Entered as an integer representing a fixed-point decimal time in seconds with two decimal places. The least significant digit represents 0.01 second (10ms.) units. Displayed as seconds with two decimal places.</p> <p>For example, Kd entered as 120 is displayed as 1.20 Sec and results in a <math>Kd * \Delta Error / \text{delta time}</math> or <math>120 * 4 / 3</math> contribution to the PID Output if Error changes by 4 PV Counts every 30ms. Kd can be used to speed up a slow loop response, but is very sensitive to PV input noise. This noise sensitivity can be reduced by using the derivative filter, which is enabled by setting bit 5 of the <i>Config Word</i>.</p> <p>PID_ISA: The ISA derivative time in seconds, Td, is entered and displayed in the same way as Kd. Total derivative contribution to PID Output is <math>Kc * Td * \Delta Error / dt</math>.</p>	0.01 sec	0 to 327.67 sec
<b>8</b> <b>(Address+7)</b>	<p><b>PID_IND: Integral Rate (Ki)</b>  <b>PID_ISA: Integral Rate (1/Ti = Ki)</b></p> <p>PID_IND: Rate of change in the control variable in CV Counts per second when the Error is a constant 1 PV Count. Entered as an integer representing a fixed-point decimal rate with three decimal places. The least significant digit represents 0.001 counts per second, or 1 count per 0.001 second. Displayed as Repeats/Sec with three decimal places.</p> <p>For example, Ki entered as 1400 is displayed as 1.400 Repeats/Sec and results in a <math>Ki * Error * dt</math> or <math>1400 * 20 * 50/1000 = 1,400</math> contribution to PID Output for an Error of 20 PV Counts and a 50ms. CPU sweep time (Sample Period of 0).</p> <p>PID_ISA: The ISA Integral Time in <u>seconds</u>, Ti, must be inverted and entered, as integral rate, as described for PID_IND. Total integral contribution to PID Output is <math>Kc * Ki * Error * dt</math>.</p> <p>Ki is usually the second gain set after Kp.</p>	Repeats/0.001 Sec	0 to 32.767 repeats/sec
<b>9</b> <b>(Address+8)</b>	<p><b>CV Bias/Output Offset</b></p> <p>Number of CV Counts added to the PID Output before the rate and amplitude clamps. It can be used to set non-zero CV values when only Kp Proportional gains are used, or for feed-forward control of this PID loop output from another control loop.</p>	CV Counts	-32768 to 32767 (add to PID output)
<b>10, 11</b> <b>(Address+9, Address+10)</b>	<p><b>CV Upper Clamp</b>  <b>CV Lower Clamp</b></p> <p>Number of CV Counts that define the highest and lowest value that CV is allowed to take. These values are required. The Upper Clamp must have a more positive value than the Lower Clamp, or the PID block will not work. These are usually used to define limits based on physical limits for a CV output. They are also used to scale the Bar Graph display for CV. The PID block has anti-reset-windup, controlled by bit 4 of the Config Word, to modify the integral term value when a CV clamp is reached.</p>	CV Counts	-32,768 to 32,767 (Word 10 must be greater than word 11.)

Words	Parameter/Description	Low Bit Units	Range
<p><b>12</b> <b>(Address+11)</b></p>	<p><b>Minimum Slew Time</b></p> <p>Minimum number of seconds for the CV output to move from 0 to full travel of 100% or 32,000 CV Counts. It is an inverse rate limit on how fast the CV output can change.</p> <p>If positive, CV cannot change more than 32,000 CV Counts times the solution time interval (seconds) divided by Minimum Slew Time.</p> <p>For example, if the Sample Period is 2.5 seconds and the Minimum Slew Time is 500 seconds, CV cannot change more than <math>32,000 * 2.5 / 500</math> or 160 CV Counts per PID solution.</p> <p>The integral term value is adjusted if the CV rate limit is exceeded.</p> <p>When Minimum Slew Time is 0, there is no CV rate limit. Set Minimum Slew Time to 0 while tuning or adjusting PID loop gains.</p>	<p>Seconds / Full Travel</p>	<p>0 (none) to 32,000 sec to move full CV travel</p>

<p><b>13</b> <b>(Address+12)</b></p>	<p><b>Config Word</b></p> <p>The low 6 bits of this word are used to modify default PID settings. The other bits should be set to 0.</p> <p><b>Bit 0: Error Term Mode.</b></p> <p>When this bit has the default value of 0, the error term is SP - PV. If the Error=SP-PV is positive, the CV output will decrease. If the Error=SP-PV is negative, the CV output will increase. This type of operation is known as <i>reverse acting</i>. A good example is your home heating system.</p> <p>When this bit is 1, the error term is PV - SP. If the Error=PV-SP is positive, the CV output will increase. If the Error= PV-SP is negative, the CV output will decrease. This type of operation is known as <i>direct acting</i>. A good example is your home cooling system.</p> <p><b>Bit 1: Output Polarity.</b></p> <p>When this bit is 0, the CV output is the output of the PID calculation. When it is set to 1, the CV output is the negated output of the PID calculation. Setting this bit to 1 inverts the Output Polarity so that CV is the negative of the PID output rather than the normal positive value.</p> <p><b>Bit 2: Derivative Action on PV.</b></p> <p>When this bit is 0, the derivative action is applied to the error term. When it is set to 1, the derivative action is applied to PV only.</p> <p><b>Bit 3: Deadband action.</b></p> <p>When the Deadband action bit is 0, the actual error value is used for the PID calculation.</p> <p>When the Deadband action bit is 1, deadband action is chosen. If the error value is within the deadband limits, the error used for the PID calculation is forced to be zero. If, however, the error value is outside the deadband limits, the magnitude of the error used for the PID calculation is reduced by the deadband limit (<math> \text{error}  =  \text{error} - \text{deadband limit} </math>).</p> <p><b>Bit 4: Anti-reset windup action.</b></p> <p>When this bit is 0, the anti-reset-windup action uses a reset (integral term) back-calculation. When the output is clamped, the accumulated integral term is replaced with whatever value is necessary to produce the clamped output exactly.</p> <p>When the bit is 1, the accumulated integral term is replaced with the value of the integral term at the start of the calculation. In this way, the pre-clamp integral value is retained as long as the output is clamped. This option is not recommended for new applications. Refer to <i>CV Amplitude and Rate Limits</i> below.</p> <p><b>Bit 5: Enable derivative filtering.</b></p> <p>When this bit is set to 0, no filtering is applied to the derivative term. When set to 1, a first order filter is applied. This will limit the effects of higher frequency process disturbances, such as measurement noise, on the derivative term.</p>	<p>Low 6 bits used</p>	<p>Boolean</p>
--	--	------------------------	----------------

Words	Parameter/Description	Low Bit Units	Range																								
<b>14</b> (Address+13)	<b>Manual Command</b> Set to the current CV output while the PID block is in Automatic mode. When the block is switched to Manual mode, this value is used to set the CV output and the internal value of the integral term within the Upper and Lower Clamp and Slew Time limits.	CV Counts	Tracks CV in Auto or sets CV in Manual																								
<b>15</b> (Address+14)	<p><b>Control Word</b> If the Override bit (bit 0) is set to 1, the Control Word and the internal SP, PV and CV parameters must be used for remote operation of the PID block (see below). This allows a remote operator interface device, such as a computer, to take control away from the application program.</p> <hr/> <p style="text-align: center;"><b>Caution</b></p> <div style="display: flex; align-items: center;">  <p><b>If you do not want to allow remote operation of the PID block, make sure the Control Word is set to 0. If the low bit is 0, the next 4 bits can be read to track the status of the PID input contacts as long as the PID Enable contact has power.</b></p> </div> <hr/> <p>Control Word is a discrete data structure with the first five bit positions defined in the following format:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Bit</th> <th>Word Value</th> <th>Function</th> <th>Status or External Action if Override bit is set to 1:</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> <td>Override</td> <td>If 0, monitor block contacts below. If 1, set them externally.</td> </tr> <tr> <td>1</td> <td>2</td> <td>Manual /Auto</td> <td>If 1, block is in Manual mode. If other numbers, it is in Automatic mode.</td> </tr> <tr> <td>2</td> <td>4</td> <td>Enable</td> <td>Should normally be 1. Otherwise block is never called.</td> </tr> <tr> <td>3</td> <td>8</td> <td>UP /Raise</td> <td>If 1 and Manual (Bit 1) is 1, CV is incremented every solution.</td> </tr> <tr> <td>4</td> <td>16</td> <td>DN /Lower</td> <td>If 1 and Manual (Bit 1) is 1, CV is decremented every solution.</td> </tr> </tbody> </table>	Bit	Word Value	Function	Status or External Action if Override bit is set to 1:	0	1	Override	If 0, monitor block contacts below. If 1, set them externally.	1	2	Manual /Auto	If 1, block is in Manual mode. If other numbers, it is in Automatic mode.	2	4	Enable	Should normally be 1. Otherwise block is never called.	3	8	UP /Raise	If 1 and Manual (Bit 1) is 1, CV is incremented every solution.	4	16	DN /Lower	If 1 and Manual (Bit 1) is 1, CV is decremented every solution.	Maintained by the CPU, unless bit 0 (Override) is set to 1.	Boolean
Bit	Word Value	Function	Status or External Action if Override bit is set to 1:																								
0	1	Override	If 0, monitor block contacts below. If 1, set them externally.																								
1	2	Manual /Auto	If 1, block is in Manual mode. If other numbers, it is in Automatic mode.																								
2	4	Enable	Should normally be 1. Otherwise block is never called.																								
3	8	UP /Raise	If 1 and Manual (Bit 1) is 1, CV is incremented every solution.																								
4	16	DN /Lower	If 1 and Manual (Bit 1) is 1, CV is decremented every solution.																								
<b>16</b> (Address+15)	<b>Internal SP</b> Tracks the SP input. If Override = 1, must be set externally to solve the PID algorithm using an alternate SP value. The original SP value is maintained until overwritten.	Set and maintained by the CPU, unless bit 0 (Override) of Control Word is set to 1.	Non-configurable, unless bit 0 (Override) of Control Word is set to 1.																								
<b>17</b> (Address+16)	<b>Internal CV</b> Tracks CV output.	Set and maintained by the CPU.	Non-configurable.																								

<b>Words</b>	<b>Parameter/Description</b>	<b>Low Bit Units</b>	<b>Range</b>
<b>18</b> <b>(Address+17)</b>	<b>Internal PV</b> Tracks PV input. Must be set externally if Override bit is set to 1.	Set and maintained by the CPU, unless bit 0 (Override) of Control Word is set to 1.	Non-configurable, unless bit 0 (Override) of Control Word is set to 1.
<b>19</b> <b>(Address+18)</b>	<b>Output</b> A Signed word value representing the output of the function block before the optional inversion. If the output polarity bit in the Config Word is set to 0, this value equals the CV output. If the output polarity bit is set to 1, this value equals the negative of the CV output.	Set and maintained by the CPU.	Non-configurable.
<b>20</b> <b>(Address+19)</b>	<b>Derivative Term Storage</b> Used internally for storage of intermediate values. Do not write to this location.		
<b>21, 22</b> <b>(Address+20, Address+21)</b>	<b>Integral Term Storage</b> Used internally for storage of intermediate values. Do not write to these locations.		
<b>23</b> <b>(Address +22)</b>	<b>Slew Term Storage</b> Used internally for storage of intermediate values. Do not write to this location.		
<b>24 - 26</b> <b>(Address+23 - Address+25)</b>	<b>Previous Solution Time</b> Internal storage of time of last PID solution. Normally do not write to these locations. Some special circumstances may justify writing to these locations. <b>Note:</b> If you call the PID block in Automatic mode after a long delay, you might want to use SVC_REQ #16 or SVC_REQ #51 to load the current CPU elapsed time clock into Word 24 to update the last PID solution time to avoid a step change of the integral term.	Set and maintained by the CPU.	Non-configurable.
<b>27</b> <b>(Address+26)</b>	<b>Integral Remainder Storage</b> Holds remainder from integral term scaling.	Set and maintained by the CPU.	Non-configurable.
<b>28, 29</b> <b>(Address+27, Address+28)</b>	<b>SP, PV Lower Range</b> <b>SP, PV Upper Range</b> Optional integer values in PV Counts that define high and low display values for SP and PV. (Word 29 must be greater than word 28.)	PV Counts	-32768 to 32767
<b>30</b> <b>(Address+29)</b>	<b>Reserved</b> Word 30 is reserved. Do not use this location.	N/A	Non-configurable.
<b>31, 32</b> <b>(Address+30, Address+31)</b>	<b>Previous Derivative Term Storage</b> Used in calculations for derivative filter. Do not write to these locations.	Set and maintained by the CPU.	Non-configurable.

<b>Words</b>	<b>Parameter/Description</b>	<b>Low Bit Units</b>	<b>Range</b>
<b>33 - 40 (Address+32 - Address+39)</b>	<b>Reserved</b> Words 32-39 are reserved. Do not use these references.	N/A	Non-configurable

## 7.3 Operation of the PID Function

### 7.3.1 Automatic Operation

When the PID function block is called, it compares the current CPU time with the last PID solution time stored in the reference array. If the interval between the two times is equal to or greater than the Sample Period (word 3 of the reference array) and also equal to or greater than 10 ms, the PID algorithm is solved using this time interval. Both the last solution time and CV output are updated. In Automatic mode, the output CV is copied to the Manual Command parameter (word 14 of the reference array).

**Note:** If you call the PID block in Auto mode after a long delay, you may want to use SVC\_REQ 16 or SVC\_REQ 51 to load the current CPU time into the stored *Previous Solution Time* (word 24 of the reference array). This will update the last PID solution time and avoid a large step change of the integral term. Another method to prevent the step change is to copy the PV value to the SP before placing the loop into Auto.

### 7.3.2 Manual Operation

The PID function block is placed in Manual mode by providing power flow to both the Enable and Manual input contacts. The output CV is set from the Manual Command parameter. If either the UP or DN inputs have power flow, the Manual Command word is incremented (UP) or decremented (DN) by one CV count every PID Sample Period. For faster manual changes of the output CV, it is also possible to add or subtract any CV count value directly to/from the Manual Command word (word 14 of the reference array).

The PID function block uses the CV Upper Clamp and CV Lower Clamp parameters to limit the CV output. If a positive Minimum Slew Time (word 12 of the reference array) is defined, it is used to limit the rate of change of the CV output. If either CV Clamp or the rate of change limit is exceeded, the value of the integral (reset) term is adjusted so that CV is at the limit. The anti-reset-windup feature assures that when the error term tries to drive CV above (or below) the clamps for a long period of time, the CV output will move off the clamp immediately when the error term changes sufficiently.

This operation, with the Manual Command tracking CV in Automatic mode and setting CV in Manual mode, provides a bump-less transfer from Automatic to Manual mode. The CV Upper and Lower Clamps and the Minimum Slew Time always apply to the CV output in Manual mode and the integral term is always updated. This assures that when a user rapidly changes the Manual Command value in Manual mode, the CV output cannot change any faster than the slew rate limit set by the Minimum Slew Time, and the CV cannot go above the CV Upper Clamp limit or below the CV Lower Clamp limit.

In order to assure a bump-less transfer from Manual back to Automatic mode, the user program should copy the PV to the SP before switching back to Automatic mode. This allows the algorithm to update the last sample period time and prepare to re-calculate CV based upon the new Auto Mode SP commanded.

### 7.3.3 Time Interval for the PID Function

The start time of each CPU sweep is used as the current time when calculating the time interval between solutions of the PID function. The times and time intervals have a resolution of 100  $\mu$ s. When an application uses multiple PID functions, all of them use the same time value.

The PID algorithm is solved when the current time is equal to or greater than the time of the last PID solution plus the Sample Period or 10 ms; whichever is larger. If the Sample Period is set for execution on every sweep (value = 0), the PID function is restricted to a minimum of 10 ms between solutions. **If the sweep time is less than 10 ms, the PID function waits until enough sweeps have occurred to accumulate an elapsed time of at least 10 ms.** For example, if the sweep time is 9 ms, the PID function executes every other sweep, and the time interval between solutions is 18 ms. If a specific PID function is executed more than once per sweep (by referencing the same reference array location in multiple PID function blocks), the algorithm is solved only on the first call.

The longest possible interval between executions is 65,535 times 10 ms, or 10 minutes, 55.35 seconds.

## 7.4 PID Algorithm Selection (PIDISA or PIDIND) and Gain Calculations

The PID function supports both the Independent Term (PID\_IND) and ISA standard (PID\_ISA) forms of the PID algorithm. The Independent Term form takes its name from the fact that the coefficients for the proportional, integral and derivative terms act independently. The ISA algorithm is named for the Instrument Society of America (now the International Society for Measurement and Control), which standardized and promoted it.

The two algorithms differ in how words 6 through 8 of the reference array are used and in how the PID output (CV) is calculated.

The Independent term PID (PID\_IND) algorithm calculates the output as:

$$\text{PID Output} = K_p * \text{Error} + K_i * \text{Error} * dt + K_d * \text{Derivative} + \text{CV Bias}$$

where  $K_p$  is the proportional gain,  $K_i$  is the integral rate,  $K_d$  is the derivative time, and  $dt$  is the time interval since the last solution.

The ISA (PID\_ISA) algorithm has different coefficients for the terms:

$$\text{PID Output} = K_c * (\text{Error} + \text{Error} * dt/T_i + T_d * \text{Derivative}) + \text{CV Bias}$$

where  $K_c$  is the controller gain,  $T_i$  is the Integral time and  $T_d$  is the Derivative time. The advantage of PID\_ISA is that adjusting  $K_c$  changes the contribution for the integral and derivative terms as well as the proportional term, which can simplify loop tuning.

If you have the PID\_ISA  $K_c$ ,  $T_i$  and  $T_d$  values, use the following equations to convert them to use as PID\_IND parameters:

$$K_p = K_c, K_i = K_c/T_i, \text{ and } K_d = K_c * T_d$$

The following diagram shows how the PID\_IND algorithm works:

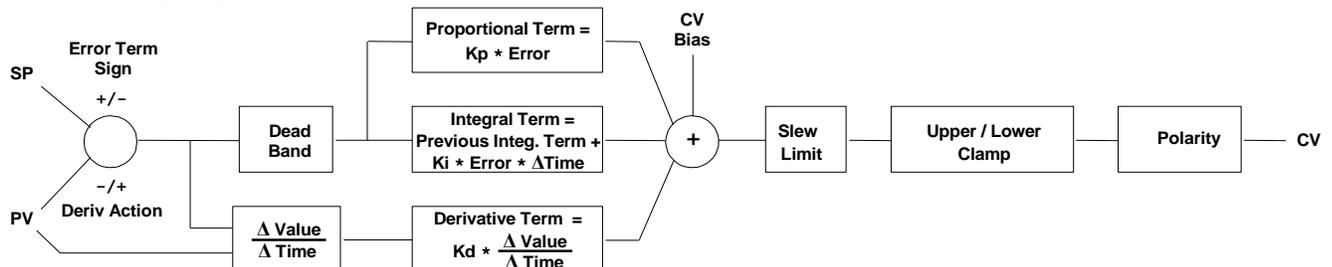


Figure 16: PID\_IND Diagram

The ISA Algorithm (PID\_ISA) is similar except that its  $K_c$  gain coefficient is applied after the three terms are summed, so that the integral gain is  $K_c / T_i$  and the derivative gain is  $K_c * T_d$ .

Bits 0, 1 and 2 in the Config Word set the Error sign, Output Polarity and Derivative Action, respectively.

### 7.4.1 Derivative Term

The Derivative Term is  $K_d$  (word 7 of the reference array) multiplied by the time rate of change of the Error term in the interval since the last PID solution.

$$\text{Derivative} = K_d * \Delta \text{Error} / dt = K_d * (\text{Error} - \text{previous Error}) / dt$$

where

$$dt = \text{Current controller time} - \text{controller time at previous PID solution.}$$

Two bits in the Config Word (word 13 of the reference array) affect the calculation of  $\Delta \text{Error}$ : Error Term Mode and Derivative Action. For additional information about the operation of these bits, refer to *Config Word* above.

### 7.4.2 Error Term Mode

The sign of the Error term is determined by the value of a mode bit in the reference array for the PID function.

In *reverse acting* mode, the change in the error term is:

$$\Delta \text{Error} = (\text{Error} - \text{previous Error}) = \Delta SP - \Delta PV$$

where

$$\Delta PV = (PV - \text{previous PV}), \text{ and } \Delta SP = (SP - \text{previous SP}).$$

However, in *direct acting* mode, the error term is  $(PV - SP)$ , the sign of the change in the error term is reversed:

$$\Delta \text{Error} = (\text{Error} - \text{previous Error}) = \Delta PV - \Delta SP.$$

### 7.4.3 Derivative Action on PV Bit

By default, the change in the error term depends on changes in both SP and PV. If SP is constant,  $\Delta SP = 0$ , and SP has no effect on the derivative term. When SP changes, however, it can cause large transient swings in the derivative term and hence the output. Loop stability can be improved by eliminating the effect of SP changes on the derivative term.

To calculate the Derivative based only on the change in PV, set bit 2 of the Config Word to 1. This modifies the equations above by assuming SP is constant ( $\Delta SP = 0$ ).

### 7.4.4 Combined Operation of Error Term and Derivative Action Modes

Bit 0 of Config Word		Bit 2 of Config Word		Error Term Value
Value	Error Term Mode	Value	Derivative Action	
0	Reverse Acting (default)	0	$\Delta SP$ included	$\Delta SP - \Delta PV$
1	Direct Acting	0	$\Delta SP$ included	$\Delta PV - \Delta SP$
0	Reverse Acting (default)	1	$\Delta SP$ ignored	$-\Delta PV$
1	Direct Acting	1	$\Delta SP$ ignored	$\Delta PV$

### 7.4.5 CV Bias Term

The CV Bias term (word 9 in the reference array) is an additive term separate from the PID inputs. It may be useful if you are using only Proportional gain (Kp) and you want the CV to be a non-zero value when the PV equals the SP and the Error is 0. In this case, set the CV Bias to the desired CV when the PV is at the SP. CV Bias can also be used for feed forward control where another PID loop or control algorithm is used to adjust the CV output of this PID loop.

If a non-zero Integral rate is used, the CV Bias will normally be 0 as the integral term acts as an automatic bias or *reset*. Just start up in Manual mode and use the Manual Command word (word 14 of the reference array) to set the desired CV, and then switch to Automatic mode. This will immediately calculate the required value for the integral term.

### 7.4.6 CV Amplitude and Rate Limits

The PID block does not send the calculated Output directly to CV. Both PID algorithms can impose amplitude and rate of change limits on the output Control Variable. If the Minimum Slew Time (word 12 of the reference array) is non-zero, the rate of change (slew rate) limit is determined by dividing the maximum CV value (32,000) by the Minimum Slew Time. For example, if the Minimum Slew Time is 100 seconds, the rate limit will be 320 CV counts per second. If the solution interval was 50 ms, the new CV output cannot change more than  $320 \times 50 / 1000$  or 16 CV counts from the previous CV output.

The CV output is then compared to the CV Upper Clamp and CV Lower Clamp values (words 10 and 11 of the reference array). If CV is outside either limit, the CV output is clamped to the appropriate limit value. When the CV output is modified to impose either slew rate or amplitude limits (or both) the stored integral term would normally accumulate a large value over time. This phenomenon is known as *reset windup*. Reset windup introduces errors in CV after the PID output no longer needs to be limited. For example, windup would prevent the CV output from moving off a clamp value immediately.

There are two optional methods for preventing reset windup. If the Anti-reset-windup Action bit (bit 4) of Config Word (word 13 of the reference array) is zero (the default), the integral term is adjusted at each PID solution to match the error input and limited CV output exactly. When PV changes while CV is clamped, or when CV is both rate and amplitude limited in a particular PID solution, this option assures that a smooth transition will always occur after CV is no longer limited.

If the Anti-reset-windup Action bit of Config Word is set, then the integral term stored on the previous PID solution is simply retained as long as CV is limited. This option was added to assure compatibility with existing PID applications when the default action described above was introduced. This option is not recommended for new applications.

Finally, the PID block checks the Output Polarity (bit 2 of the Config Word) and changes the sign of the output if the bit is 1.

CV = - (Clamped PID Output) if Output Polarity bit set, or

CV = (Clamped PID Output) if Output Polarity bit cleared.

If the block is in Automatic mode, the final CV is placed in the Manual Command (word 14 of the reference array). If the block is in Manual mode, the PID equation is skipped because CV is set by the Manual Command, but the slew rate and amplitude limits are still checked. This assures that the Manual Command cannot change the output above the CV Upper Clamp or below the CV Lower Clamp, and the output cannot change faster than allowed by the Minimum Slew Time.

### 7.4.7 Sample Period and PID Function Block Scheduling

The PID function block is a digital implementation of an analog control function, so the  $dt$  sample time in the PID Output equation is not the infinitesimally small sample time available with analog controls. The majority of processes being controlled can be approximated as a gain with a first or second order lag and (possibly) a pure time delay. The PID function block sets a CV output to the process and uses the process feedback PV to determine an Error to adjust the next CV output. A key process parameter is the total time constant, which is how fast the process can change PV when the CV is changed. As discussed in *Determining the Process Characteristics* below, the total time constant,  $T_p+T_c$ , for a first order system is the time required for PV to reach 63% of its final value when CV is stepped. The PID function block will not be able to control a process unless its Sample Period is well under half the total time constant. Larger Sample Periods will make it unstable.

The Sample Period should be no bigger than the total time constant divided by 10 (or down to 5 worst case). For example, if PV seems to reach about 2/3 of its final value in 2 seconds, the Sample Period should be less than 0.2 seconds, or 0.4 seconds worst case. On the other hand, the Sample Period should not be too small, such as less than the total time constant divided by 1000, or the  $K_i * \text{Error} * dt$  term for the PID integral term will round down to 0. For example, a very slow process that takes 10 hours or 36,000 seconds to reach the 63% level should have a Sample Period of 40 seconds or longer.

Variations of the time interval between PID function solutions can have short-term effects on the CV output. For example, if a step change to PV caused by measurement noise occurs between solutions, the value of the derivative term will be inversely proportional to the time interval. The performance of PID loops that are tuned for quick response may be improved when the solution interval is held constant by configuring the CPU for constant sweep mode. Depending on the CPU model and the application, constant sweep times of 10 ms, integer multiples of 10 ms, or exact divisors of 10 ms (1, 2 or 5 ms) will be possible. The Sample Period can then be set for a suitable multiple of 10 ms.

If many PID loops are used, allowing the application to solve all the loops on the same sweep may lead to wide variations in CPU sweep time. If the loops have a common Sample Period that is at least equal to the number of PID loops times the sweep time, a simple solution is to sequence one or more 1's through an array of zero's and use these bits to enable power flow to individual PID function blocks. The logic should assure that each PID function block is enabled no more often than its Sample Period.

## 7.5 Determining the Process Characteristics

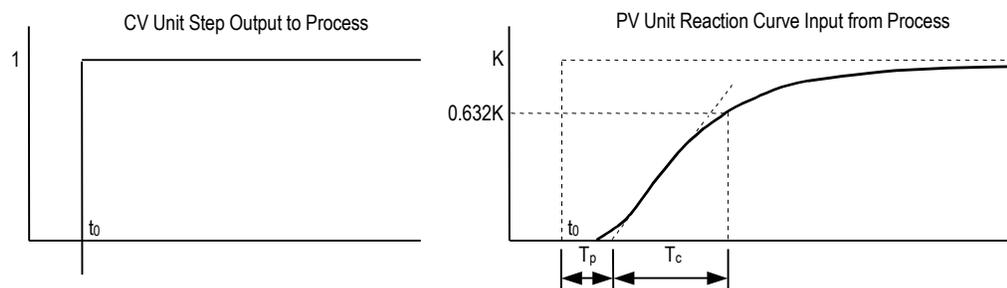
The PID loop gains,  $K_p$ ,  $K_i$  and  $K_d$ , are determined by the characteristics of the process being controlled. Two key questions when setting up a PID loop are:

1. How big is the change in PV when CV is changed by a fixed amount, or what is the open loop gain of the process?
2. How fast does the system respond, or how quickly does PV change after the CV output is stepped?

Many processes can be approximated by a process gain, first or second order lag and a pure time delay. In the frequency domain, the transfer function for a first order lag system with a pure time delay is:

$$\frac{PV(s)}{CV(s)} = G(s) = Ke^{-T_p/(1+T_c s)}$$

Plotting the response to a step input at time  $t_0$  in the time domain provides an open-loop unit reaction curve:



The following process model parameters can be determined from the PV unit reaction curve:

$K$	Process open loop gain = final change in PV/change in CV at time $t_0$ (Note no subscript on $K$ )
$T_p$	Process or pipeline time delay or dead time after $t_0$ before the process output PV starts moving
$T_c$	First order Process time constant, time required after $T_p$ for PV to reach 63.2% of the final PV

Usually the quickest way to measure these parameters is by putting the PID function block in Manual mode, making a small step change in the CV output by changing the Manual Command (word 14 of the reference array), and then plotting the PV response over time. For slow processes this can be done manually, but for faster processes a chart recorder or computer graphic data-logging package will help. The CV step size should be large enough to cause an observable change in PV, but not so large that it disrupts the process being measured. A good step size may be from 2 to 10% of the difference between the CV Upper and CV Lower Clamp values.

## 7.6 Setting Tuning Loop Gains

### 7.6.1 Basic Iterative Tuning Approach

Because PID parameters are dependent on the process being controlled, there are no predetermined values that will work. However, a simple iterative process can be used to find acceptable values for Kp, Ki, and Kd gains.

1. Set all the reference array parameters to 0, then set the CV Upper and CV Lower Clamps to the highest and lowest CV expected. Set the Sample Period to a value within the range  $T_c/10$  to  $T_c/100$ , where  $T_c$  is the estimated process time constant defined in *Determining the Process Characteristics*.
2. Put the PID function block in Manual mode and set the Manual Command (word 14 in the reference array) to different values to check if CV can be moved to Upper and Lower Clamp. Record the PV value at some CV point and load it into SP.
3. Set a small gain, such as  $100 * \text{Maximum CV}/\text{Maximum PV}$ , into Kp and turn off Manual mode. Step SP by 2% to 10% of the Maximum PV range and observe PV response. Increase Kp if PV step response is too slow or reduce Kp if PV overshoots and oscillates without reaching a steady value.
4. Once a Kp is found, start increasing Ki to get overshooting that dampens out to a steady value in two to three cycles. This may require reducing Kp. Also try different SP step sizes and CV operating points.
5. After suitable Kp and Ki gains are found, try adding Kd to get quicker responses to input changes, providing it doesn't cause oscillations. Kd is often not needed and will not work with noisy PV.
6. Check gains over different SP operating points and add Dead Band and Minimum Slew Time if needed. Some Reverse Acting processes may need setting of Config Word Error Term or Output Polarity bits.

### 7.6.2 Setting Loop Gains Using the Ziegler and Nichols Tuning Approach

This approach provides good response to system disturbances with gains producing an amplitude ratio of 1/4. The amplitude ratio is the ratio of the second peak over the first peak in the closed loop response.

1. Determine the three process model parameters,  $K$ ,  $T_p$  and  $T_c$  for use in estimating initial PID loop gains.

2. Calculate the Reaction rate:

$$R = K/T_c$$

3. For Proportional control only, calculate  $K_p$  as:

$$K_p = 1/(R * T_p) = T_c/(K * T_p)$$

For Proportional and Integral control, use:

$$K_p = 0.9/(R * T_p) = 0.9 * T_c/(K * T_p) \quad K_i = 0.3 * K_p/T_p$$

For Proportional, Integral and Derivative control, use:

$$K_p = G/(R * T_p) \quad \text{where } G \text{ is from } 1.2 \text{ to } 2.0$$

$$K_i = 0.5 * K_p/T_p$$

$$K_d = 0.5 * K_p * T_p$$

4. Check that the Sample Period is in the range

$$(T_p + T_c)/10 \text{ to } (T_p + T_c)/1000$$

### 7.6.3 Ideal Tuning Method

The *Ideal Tuning* procedure provides the best response to SP changes that are delayed only by the  $T_p$  process delay or dead time.

1. Determine the three process model parameters,  $K$ ,  $T_p$  and  $T_c$  for use in estimating initial PID loop gains.
2. Calculate  $K_p$ ,  $K_i$ , and  $K_d$  as follows:
$$K_p = 2 * T_c / (3 * K * T_p)$$
$$K_i = T_c$$
$$K_d = K_i / 4 \quad \text{if Derivative term is used}$$
3. Once initial gains are determined, convert them to integers.
4. Calculate the Process gain,  $K$ , as a change in input PV Counts divided by the resulting output step change in CV Counts. (Not in process PV or CV engineering units.) Specify all times in seconds.
5. Once  $K_p$ ,  $K_i$  and  $K_d$  are determined,  $K_p$  and  $K_d$  are multiplied by 100 while  $K_i$  is multiplied by 1000. The resulting values are entered into the corresponding reference array word locations.

## 7.7 PID Example

The following PID example has a sample period of 100ms, a Kp gain of 4.00 and a Ki gain of 1.500. The set point is stored in %R0001, the control variable is output in %AQ0002, and the process variable is returned in %AI0003. CV Upper and CV Lower Clamps must be set, in this case to 20000 and 4000, and an optional small Dead Band of +5 and -5 is included. The 40-word reference array starts in %R0100. Normally, user parameters are set in the reference array, but %M0006 can be set to re-initialize the 14 words starting at %R0102 (word 3) from constants stored in logic (a useful technique).

The block can be switched to Manual mode with %M1 so that the Manual Command, %R113, can be adjusted. Bits %M4 or %M5 can be used to increase or decrease %R113 and the PID CV by 1 every 100ms solution. For faster manual operation, bits %M2 and %M3 can be used to add or subtract the value in %R2 to/from %R113 every CPU sweep. The %T1 output is on when the PID is OK.

### 7.7.1 Reference Array Initialization using %M0006

For details on the contents of the reference array, refer to *Reference Array for the PID Function*.

Word	Function	Address	Value
3	Sample Period	%R102	10
4	+ Dead Band	%R103	5
5	- Dead Band	%R104	5
6	Kp	%R105	400
7	Kd	%R106	0
8	Ki	%R107	1500
9	CV Bias	%R108	0
10	CV Upper Clamp	%R109	2000
11	CV Lower Clamp	%R110	400
12	Minimum Slew Time	%R111	0
13	Config Word	%R112	0
14	Manual Command	%R113	0
15	Control Word	%R114	0
16	Internal SP	%R115	0

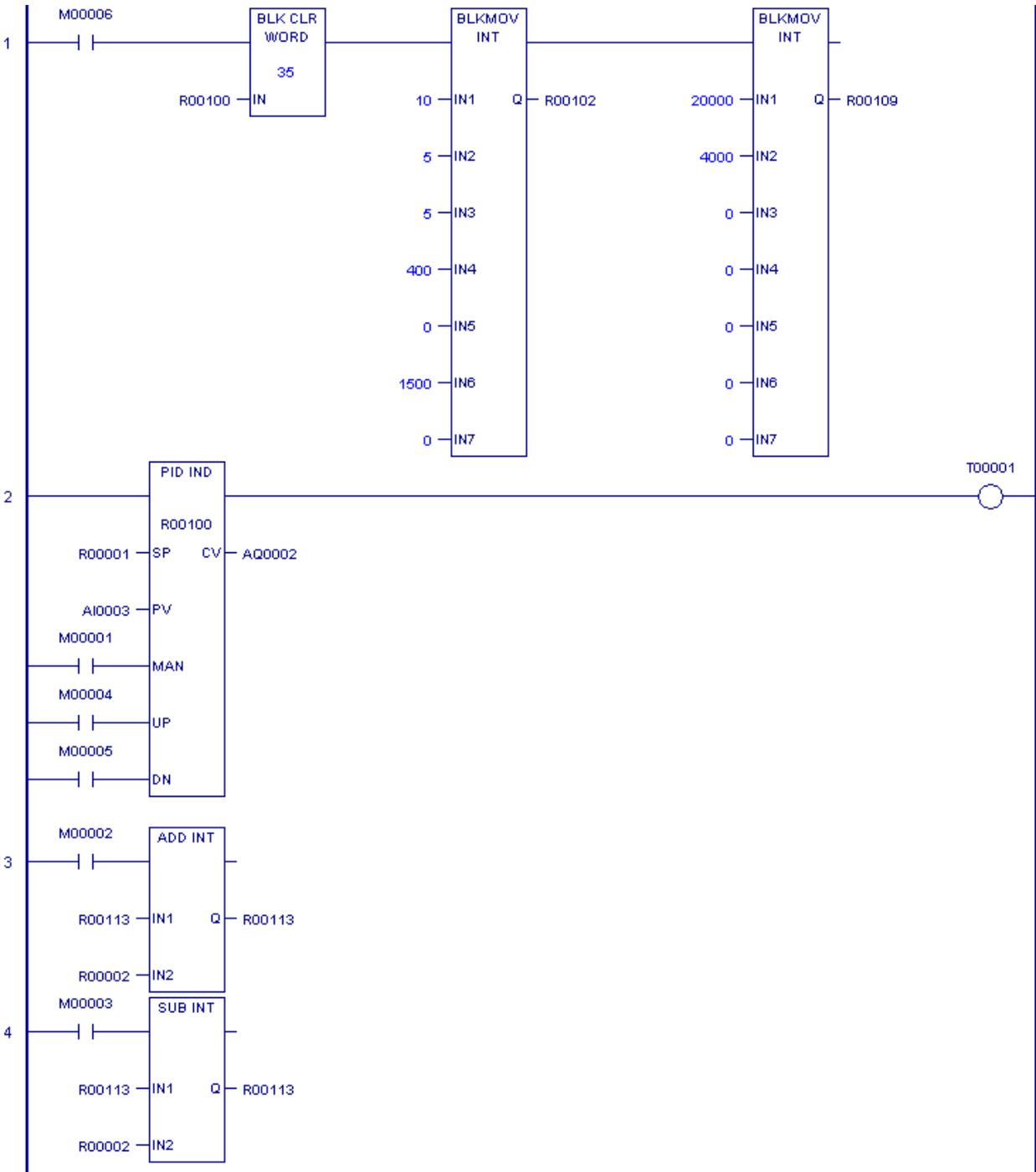


Figure 17: PID Example Logic



## Chapter 8 Structured Text (ST) Programming

---

The Structured Text (ST) programming language is an IEC 61131-3 textual programming language. This chapter describes how structured text is implemented in PACSystems. For information on using the structured text editor in the programming software, refer to the online help.

The block types Block, Parameterized Block, and Function Block (UDFB) can be programmed in ST. The \_MAIN program block can also be programmed in ST. For details on blocks, refer to *Program Organization* in Chapter 2.

### 8.1 Language Overview

#### 8.1.1 Statements

A structured text program consists of a series of statements, which are constructed from expressions and language keywords. A statement directs the PACSystems controller to perform a specified action. Statements provide variable assignments, conditional evaluations, iteration, and the ability to call built-in functions. PACSystems supports those statements described in *Statement Types*.

#### 8.1.2 Expressions

Expressions use operators to calculate values from variables and constants. An example of a simple expression is  $(x + 5)$ .

Composite expressions can be created by nesting simpler expressions, for example,

$(a + b) * (c + d) - 3.0 ** 4$ .

### 8.1.3 Operators

The table below lists the operators that you can use within an expression. They are listed according to their evaluation precedence, which determines the sequence in which they are executed within the expression. The operator with the highest precedence is applied first, followed by the operator with the next highest precedence. Operators of equal precedence are evaluated left to right. Operators in the same group, for example + and -, have the same precedence.

Any address operators used in LD can be used on ST operands. Address operators have precedence over the ST language operators. Address operators include indirect addressing (for example, @Var1), array indexing (for example, Var1[3]), bit within word addressing (for example, Var1.X[3]), and structure fields (for example, Var1.field1).

Precedence	Operator	Operand Types	Description
Group 1 (Highest)	(...)		Parenthesized expression
Group 2	- NOT	INT, DINT, REAL, LREAL BOOL, BYTE, WORD, DWORD	Negation Boolean complement
Group 3	**, <sup>^</sup>	INT, DINT, UINT, REAL, LREAL <sup>8</sup>	Exponentiation <sup>9,10</sup>
Group 4	* / MOD	INT, DINT, UINT, REAL, LREAL INT, DINT, UINT, REAL, LREAL INT, DINT, UINT	Multiplication <sup>9</sup> Division <sup>9,11</sup> Modulus operation <sup>11</sup>
Group 5	+ -	INT, DINT, UINT, REAL, LREAL INT, UINT, DINT, REAL, LREAL	Addition <sup>9</sup> Subtraction <sup>9</sup>
Group 6	<, >, <=, >=	INT, DINT, UINT, REAL, LREAL, BYTE, WORD, DWORD	Comparison
Group 7	= <>, !=	ANY <sup>12</sup> ANY <sup>12</sup>	Equality Inequality
Group 8	AND, &	BOOL, BYTE, WORD, DWORD	Boolean AND
Group 9	XOR	BOOL, BYTE, WORD, DWORD	Boolean exclusive OR
Group 10 (Lowest)	OR	BOOL, BYTE, WORD, DWORD	Boolean OR

Some comparison and math operators have corresponding built-in functions. For instance, the '+' operator is similar to the ADD\_INT function. You can use either the language operator or the built-in function. The built-in function has the advantage of returning an ENO status. For additional information refer to *Built-in Functions Supported for ST Calls*.

#### Operand Types

Type casting is not supported. To convert a type, use one of the built-in conversion functions. Use of built-in functions is described in *Function Call*.

For untyped operators (+, \*, ...), the types of the operands must match.

<sup>8</sup> The base must be type REAL or LREAL. If the base is REAL, the power can be type INT, DINT, UINT, or REAL and the result is type REAL. If the base is type LREAL, the power must be LREAL and the result will be LREAL

<sup>9</sup> Use of math operators can cause *Overflow* or *underflow*. *Overflow* results are truncated.

<sup>10</sup> If either operand is positive or negative infinity, the result is undefined.

<sup>11</sup> The CPU flags a "divide-by-0" error as an application fault.

<sup>12</sup> Operators that can take operands of type ANY can be used with any of the supported elementary data types. The supported data types are: BOOL, INT, DINT, UINT, BYTE, WORD, DWORD, LREAL and REAL. STRING and TIME data types are not supported

### 8.1.4 Structured Text Syntax

The syntax of the ST implementation for PACSystems follows the IEC 61131-3 standard.

- Structured Text statements must end in a semi-colon (;).
- Structured Text variables must be declared in the variable list for the target.

These symbols have the following functions.

**:=** assigns an expression to a variable

**;** required to designate the end of a statement

**[]** used for array indexing where the array index is an integer. For example, this sets the third element of an array to the value  $j+10$ : **intarray[3] = j + 10;**

**(\* \*)** designates a comment. These comments can span multiple lines. For example, **(\*This comment spans multiple lines.\*)**

**// or ‘** designates a single line comment. For example,

**c :=a+b; //This is a single line comment.**

**C :=a+b; ‘This is a single line comment.**

## 8.2 Statement Types

The Structured Text statements, which specify the actual program execution, consist of the following types, which are described in more detail on the following pages.

<b>Statement Type</b>	<b>Description</b>	<b>Example</b>
<b>Assignment</b>	Sets an object to a specified value.	A := 1; B := A; C := A + B;
<b>CASE</b>	Provides for the conditional execution of a set of statements.	CASE A OF 1,2 : C := 3; 3: C := 4; 4..5: C := 5; ELSE C := 0; END_CASE;
<b>COMMENT</b>	Places a text explanation in the program. Ignored by the ST compiler.	(* This is a block comment *) ' This is a line comment // This is a line comment //
<b>Function call</b>	Calls a function for execution.	FbInst(IN1 := 1, OUT1 => A);
<b>RETURN</b>	Causes the program to return from a subroutine. The return statement provides an early exit from a block.	RETURN;
<b>EXIT</b>	Terminates iterations before the terminal condition becomes TRUE (1).	EXIT;
<b>IF</b>	Specifies that one or more statements be executed conditionally.	IF (A < B) THEN C := 4; ELSIF (A = B) THEN C := 5; ELSE C := 6 END_IF;
<b>FOR ... DO</b>	Executes a statement sequence repeatedly based on the value of a control symbol.	FOR I := 1 TO 100 BY 2 DO IF (Var1 - I) = 40 THEN Key := I; EXIT; END_IF; END_FOR;
<b>WHILE</b>	Indicates that a statement sequence be executed repeatedly until a Boolean expression evaluates to FALSE (0).	WHILE J <= 100 DO J := J + 2; END_WHILE;
<b>REPEAT</b>	Indicates that a statement sequence be executed repeatedly until a Boolean expression evaluates to TRUE (1).	REPEAT J := J + 2; UNTIL J >= 100 END_REPEAT;
<b>ARG_PRESENT</b>	Determines whether a parameter value was present when the function block instance of the parameter was invoked. For example, a parameter can be optional (pass by value).	ARG_PRES (IN :=In1, Q:>Out1, ENO:>Out2);
<b>Empty Statement</b>		;

## 8.2.1 Assignment Statement

The assignment statement replaces the value of a variable with the result of evaluating an expression (of the same data type).

### Notes:

- Assignment statements can affect transition bits.
- Assignment statements take override bits into account.

### Format

#### **Variable := Expression;**

Where:

*Variable* is a simple variable, array element, etc.

*Expression* is a single value, expression, or complex expression.

### Examples

#### **Boolean assignment statements:**

```
VarBool1 := 1;
```

```
VarBool2 := (val <= 75);
```

#### **Array element assignment:**

```
Array_1[13] := (RealA /RealB)* PI;
```

## 8.2.2 Function Call

The structured text function call executes a predefined algorithm that performs a mathematical, bit string or other operation. The function call consists of the name of the function or block followed by required input or output parameters.

The structured text logic can call blocks or the PACSystems built-in functions listed in the table below. The call must be made in a single statement and cannot be part of a nested expression.

Calls to some functions, such as communications request (COMMREQ), require a command block or parameter block. For these functions, an array is declared, initialized in logic, and then passed as a parameter to the function.

### Built-in Functions Supported for ST Calls

**Note:** Only the functions listed in the following table are supported in the current PACSystems version. Other built-in functions are not supported.

Example: `cos(IN := inReal, Q => outReal, ENO => outBool);`

Category	Functions	More information
<b>Advanced Math</b>	ASIN, ATAN, ACOS, COS, SIN, TAN LOG, LN, EXP, EXPT, SQRT_INT, SQRT_DINT, SQRT_REAL	Chapter 4
<b>Math</b>	ABS_INT, ABS_DINT, ABS_REAL SCALE_DINT, SCALE_INT, SCALE_UINT	Chapter 4
<b>Communication</b>	PNIO_DEV_COMM	<i>PACSystems RX3i &amp; RSTi-EP PROFINET I/O Controller Manual, GFK-2571</i>
<b>Control</b>	DO_IO, MASK_IO_INTR, SCAN_SET_IO, SUS_IO, SUS_IO_INTR, SVC_REQ, SWITCH_POS, F_TRIG, R_TRIG	Chapter 4
<b>Data Conversion</b>	BCD4_TO_INT, BCD4_TO_UINT, BCD4_TO_REAL BCD8_TO_DINT, BCD8_TO_REAL DINT_TO_BCD8, DINT_TO_DWORD, DINT_TO_INT, DINT_TO_UINT, DINT_TO_REAL, DINT_TO_LREAL DWORD_TO_DINT INT_TO_BCD4, INT_TO_DINT, INT_TO_UINT, INT_TO_REAL, INT_TO_WORD UINT_TO_BCD4, UINT_TO_BCD8, UINT_TO_INT, UINT_TO_DINT, UINT_TO_REAL, UINT_TO_WORD REAL_TO_INT, REAL_TO_UINT, REAL_TO_DINT, REAL_TO_LREAL LREAL_TO_DINT, LREAL_TO_REAL TRUNC_INT, TRUNC_DINT DEG_TO_RAD, RAD_TO_DEG WORD_TO_INT, WORD_TO_UINT	Chapter 4
<b>Data Move</b>	ARRAY_SIZE, ARRAY_SIZE_DIM1, ARRAY_SIZE_DIM2, COMMREQ, MOVE_DATA_EX, SIZE_OF	Chapter 4
<b>PACMotion</b>	The RX3i CPUs support 56 PLCopen compliant motion functions and function blocks.	<i>PACMotion Multi-Axis Motion Controller User's Manual, GFK-2448</i>

## Calls to Standard Function Blocks

Standard function blocks are instructions that have instance data in the form of a structure variable. (For more information on function blocks and their instance data, refer to *Functions and Function Blocks* in Chapter 2.) Standard function blocks are called in the same way that a UDFB is called.

PACSystems controllers support three standard function blocks:

Pulse timer (TP)	Generates output pulses of a given duration	Refer to <i>Timer Pulse</i> in Chapter 4
On-delay timer (TON)	Delays setting an output ON for a fixed period after an input is set ON.	Refer to <i>On Delay Timer</i> in Chapter 4
Off-delay timer (TOF)	Delays setting an output OFF for a fixed period after an input goes OFF so that the output is held on for a given period longer than the input.	Refer to <i>Off Delay Timer</i> in Chapter 4

## Format of Calls to Standard Timer Function Blocks

**Notes:** TOF, TON and TP have the same input and output parameters, except for the instance variable, which must be the same type as the instruction.

Writing or forcing values to the instance data elements IN, PT, Q, ET, ENO or TI may cause erratic operation of the timer function block.

Instance data can be a variable or a parameter of the current UDFB or parameterized block.

### Formal Convention

```
myTOF_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO => outBoolSuccess);
```

```
myTON_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO => outBoolSuccess);
```

```
myTP_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO => outBoolSuccess);
```

**Note:** ENO is an optional BOOL output parameter. If ENO is used in a statement that uses the formal convention, the state of *outBoolSuccess* is set to 1 (call was successful) or 0 (call failed).

### Informal Convention

```
myTOF_Instance_Data(inBool, inDINT, outDINT, outBool);
```

```
myTON_Instance_Data(inBool, inDINT, outDINT, outBool);
```

```
myTP_Instance_Data(inBool, inDINT, outDINT, outBool);
```

**Note:** When using the informal convention, the operands must be assigned in the order shown above (that is, IN, PT, ET, Q and ENO).

## Block Types Supported for ST Calls

An ST block can call blocks of type Block, Parameterized Block, or user defined Function Block (UDFB) or External Block (C block). For more information on block types, refer to Chapter 2.

### Formal Calls vs. Informal Calls

PACSystems supports formal and informal calls in ST.

<b>Formal Calls</b>	<b>Informal Calls</b>
Input parameter assignments use the ':=' notation while output assignments use the '=>' notation.	Input and output parameters are listed in parentheses.
Optional parameters can be omitted.	Parameters cannot be omitted.
Parameters can be in any order.	Parameters must be in the correct order as follows: Inputs Instance location (if required) Length parameter (if required) Outputs, starting with the last output parameter.
The ENO parameter is specified in a formal function or block call.  All built-in functions and user-defined blocks have an optional ENO output parameter indicating the success of the function or block. Either ENO or YO can be used as this output parameter name.	The ENO parameter is not specified in an informal function or block call.

### Format of Formal Function Call

**FunctionName(IN1 := inparam1, IN2 := inparam2, OUT1 => outparam1, ENO => enoparam);**

### Format of Informal Function Call

**FunctionName(inparam1, inparam2, outparam1);**

### Example

This code fragment shows the TAN function call.

**TAN( AnyReal, Result );**

### 8.2.3 RETURN Statement

The return statement provides an early exit from a block. For example, in the following lines of code the third line will never execute. The variable **a** will have the value 4.

```
a := 4;  
RETURN;  
a := 5;
```

## 8.2.4 IF Statement

The IF construct offers conditional execution of a statement list. The condition is determined by result of a Boolean expression. The IF construct includes two optional parts, ELSE and ELSIF, that provide conditional execution of alternate statement list(s). One ELSE and any number of ELSIF sections are allowed per IF construct.

### Format

```

IF BooleanExpression1 THEN
    StatementList1;
[ELSIF BooleanExpression2 THEN (*Optional*)
    StatementList2;]
[ELSE (*Optional*)
    StatementList3;]
END_IF;

```

Where:

*BooleanExpression* Any expression that resolves to a Boolean value.

*StatementList* Any set of structured text statements.

**Note:** Either ELSIF or ELSEIF can be used for the else if clause in an IF statement.

### Operation

The following sequence of evaluation occurs if both optional parts are present:

- If BooleanExpression1 is TRUE (1), StatementList1 is executed. Program execution continues with the statement following the END\_IF keyword.
- If BooleanExpression1 is FALSE (0) and BooleanExpression2 is TRUE (1), StatementList2 is executed. Program execution continues with the statement following the END\_IF keyword.
- If both Boolean expressions are FALSE (0), StatementList3 is executed. Program execution continues with the statement following the END\_IF keyword.

If an optional part is not present, program execution continues with the statement following the END\_IF keyword.

### Example

The following code fragment puts text into the variable Status, depending on the value of I/O point input value.

```

IF Input01 < 10.0 THEN
    Status := Low_Limit_Warning;
ELSIF Input02 > 90.0 THEN
    Status := Upper_Limit_Warning;
ELSE
    Status := Limits_OK;
END_IF;

```

## 8.2.5 CASE Statement

The CASE ... OF construct offers conditional execution of statement lists. It uses the value of an ST integer expression to determine whether to execute a statement list. The statement list to be executed can be selected from multiple statement lists, depending on the value of the associated integer expression.

Conditions can be expressed as a single value, a list of values, or a range of values. The single-value, list of values, or range forms can be used by themselves or in combination. The optional ELSE keyword can be used to execute a statement list when the associated value does not meet any of the specified conditions.

You can have a maximum of 1024 cases in a single CASE ... OF construct. Additional cases can be handled by adding the ELSE keyword to the construct and specifying a nested CASE ... OF construct or an IF ... THEN construct after the ELSE.

The number of nested CASE ... OF constructs and the number of levels are limited by the memory in your computer.

The number of constants and constant ranges in a single conditional statement is limited by the memory in your computer.

### Format

```

CASE Integer_Expression OF
  Int1: StatementList_1; (*Single Value*)
  Int2,Int3,Int4: StatementList_2; (*List of Values*)
  Int5..Int6: StatementList_3; (*Range of Values*)
[ELSE StatementList_Else;] (*Optional*)
END_CASE;

```

Where:

<i>Integer_Expression</i>	An ST expression that resolves to an integer (INT, DINT or UINT) value.
<i>Int</i>	A constant integer value.
<i>StatementList_1 ... StatementList_n</i>	Structured Text statements.

### Operation

The Int values are compared to Integer\_Expression. The statement list following the first Int value that matches Integer\_Expression is executed. If the optional ELSE keyword is used and no Int value matches Integer\_Expression, the statement list following ELSE is executed. Otherwise, no statement list is executed.

### Requirements for Conditional Statements

- All constants must be of type INT, DINT or UINT.
- In range declarations, the beginning value must be less than the ending value (reading from left to right). For example, 10..3 and 5..5 are invalid.

- Overlapping values in different case conditions are not allowed. For example, 5..10 and 7 cannot be specified as conditions in the same CASE ... OF construct.

### Examples

The following code fragment assigns a value to the variable ColorVariable.

```
CASE ColorSelection OF  
  0: ColorVariable:= Red;  
  1: ColorVariable:= Yellow;  
  2,3,4: ColorVariable:= Green;  
  5..9: ColorVariable:= Blue;  
ELSE ColorVariable:= Violet;  
END_CASE;
```

The following code fragment uses a nested CASE...OF...END\_CASE construct.

```
CASE ColorSelection OF  
  0: ColorVariable:= Red;  
  1: ColorVariable:= Yellow;  
  2,3,4: ColorVariable:= Green;  
  5..9: ColorVariable:= Blue;  
ELSE  
  CASE ColorSelection OF  
    10: ColorVariable:= Violet;  
  ELSE ColorVariable:= Black;  
  END_CASE;  
  ColorError: 1;  
END_CASE;
```

## 8.2.6 FOR ... DO Statements

The FOR loop repeatedly executes a statement list contained within the FOR ... DO ... END\_FOR construct. It is useful when the number of iterations can be predicted in advance, for example to initialize an array. The number of iterations is determined by the value of a control variable which is incremented (or decremented) from an initial value to a final value by the FOR statement.

By default, each iteration of the FOR statement changes the value of the control variable by 1. The optional BY keyword can be used to specify an increment or decrement of the control variable by specifying a (non-zero) positive or negative integer or an expression that resolves to an integer.

FOR loops can be nested to a maximum of ten levels.

### Format

```
FOR Control_Variable := Start_Value TO End_Value [BY Step_Value] DO  
    Statement list;  
END_FOR;
```

Where:

<i>Control_Variable</i>	The control variable. Can be an INT, DINT or UINT variable or parameter.
<i>Start_Value</i>	The starting value of the control variable. Must be an expression, variable, or constant of the same data type as Int_Variable.
<i>End_Value</i>	The ending value of the control variable. Must be an expression, variable, or constant of the same data type as Int_Variable.
<i>Step_Value</i>	(Optional) The increment or decrement value for each iteration of the loop. Must be an expression, variable, or constant of the same data type as Int_Variable. If Step_Value is not specified, the control variable is incremented by 1.
<i>Statement list</i>	Any list of Structured Text statements.

### Operation

The values of Start\_Value, End\_Value and Step\_Value are calculated at the beginning of the FOR loop. On the first iteration, Control\_Variable is set to Start\_Value.

At the beginning of each iteration, the termination condition is tested. If it is satisfied, execution of the loop is complete and the statements after the loop will proceed. If the termination condition is not satisfied, the statements within the FOR...END\_FOR construct are executed. At the end of each iteration, the value of Control\_Variable is incremented by Step\_Value (or 1 if Step\_Value is not specified).

The termination condition of a FOR loop depends on the sign of the step value.

#### **Step Value    Termination Condition**

> 0	Control_Variable > End_Value
< 0	Control Variable < End Value
0	None. A termination condition is never reached and the loop will repeat infinitely.

As with the other iterative statements (WHILE and REPEAT), loop execution can be prematurely halted by an EXIT statement.

To avoid infinitely repeating or unpredictable loops, the following precautions are recommended:

- Do not allow the statement list logic within the FOR loop to modify the control variable.

- Do not use the control variable in logic outside the FOR loop.

### Examples

The following code fragment initializes an array of 100 elements starting at %R1000 (given that R1000 is at %R1000) by assigning a value of 10 to all array elements.

```
FOR R1000 := 1 TO 100 DO  
    @R1000 := 10;  
END_FOR;
```

The following code fragment assigns the values of an I/O point to array elements over ten I/O scans. The last entry is put in the array element with the smallest index.

```
FOR R1000 := 10 TO 1 BY -1 DO  
    @R1000 := Input01;  
END_FOR;
```

## 8.2.7 WHILE Statement

The WHILE loop repeatedly executes (iterates) a statement list contained within the WHILE...END\_WHILE construct as long as a specified condition is TRUE (1). It checks the condition first, then conditionally executes the statement list. This looping construct is useful when the statement list does not necessarily need to be executed.

### Format

```
WHILE <BooleanExpression> DO  
    <StatementList>;  
END_WHILE;
```

Where:

*BooleanExpression* Any expression that resolves to a Boolean value.

*StatementList* Any set of Structured Text statements.

### Operation

If BooleanExpression is FALSE (0), the loop is immediately exited; otherwise, if the BooleanExpression is TRUE (1), the StatementList is executed and the loop repeated. The statement list may never execute, since the Boolean expression is evaluated at the beginning of the loop.

**Note:** It is possible to create an infinite loop that will cause the watchdog timer to expire. Avoid infinite loops.

### Example

The following code fragment increments J by a value of 2 as long as J is less than or equal to 100.

```
WHILE J <= 100 DO  
    J := J + 2;  
END_WHILE;
```

### 8.2.8 REPEAT Statement

The REPEAT loop repeatedly executes (iterates) a statement list contained within the REPEAT...END\_REPEAT construct until an exit condition is satisfied. It executes the statement list first, then checks for the exit condition. This looping construct is useful when the statement list needs to be executed at least once.

#### Format

```
REPEAT  
    StatementList;  
UNTIL BooleanExpression END_REPEAT;
```

Where:

*BooleanExpression* Any expression that resolves to a Boolean value.  
*StatementList* Any set of Structured Text statements.

#### Operation

The StatementList is executed. If the BooleanExpression is FALSE (0), then the loop is repeated; otherwise, if the BooleanExpression is TRUE (1), the loop is exited. The statement list executes at least once, since the BooleanExpression is evaluated at the end of the loop.

**Note:** It is possible to create an infinite loop that will cause the watchdog timer to expire. Avoid infinite loops.

#### Example

The following code fragment reads values from an array until a value greater than 5 is found (or the upper bound of the array is reached). Since at least one array value must be read, the REPEAT loop is used. All variables in this example are of type DINT, UINT, or INT.

```
Index := 1;  
REPEAT  
    Value := @Index;  
    Index := Index + 1;  
UNTIL Value > 5 OR Index >= UpperBound END_REPEAT;
```

## 8.2.9 ARG\_PRES Statement

The ARG\_PRES function determines whether an input parameter value was present when the function block instance of the parameter was invoked. This may be necessary if the parameter is optional (pass by value).

This function must be called from a function block instance or a parameterized block.

### Format

**ARG\_PRES (IN :=In1, Q:>Out1, ENO:>Out2);**

Where:

*In1* Must be an input parameter of the function block that contains the ARG\_PRES instruction. Cannot be an array element or structure element. An alias to a parameter should resolve only to the parameter name.

Can be a BOOL, DINT, DWORD, INT, REAL, UINT, WORD variable, variable array head name or variable array head name element [000]. Input or output parameter value of a function block instance or a parameterized block

*Out2* A BOOL variable. True if the parameter is present, otherwise false.

**Note:** ENO is an optional BOOL output parameter. If ENO is used in a statement that uses the formal convention, the state of *Out2* is set to 1 (call was successful) or 0 (call failed).

### Example

The parameter TempVal is an input to the function block CheckTemp. In the following code fragment, ARG\_PRES is used to determine whether a value existed for the parameter TempVal when an instance of CheckTemp was invoked. If TempVal had a value, the BOOL output Temp\_Pres is set to 1.

**ARG\_PRES (TempVal, Temp\_Pres);**

### 8.2.10 Exit Statement

The EXIT statement is used to terminate and exit from a loop (FOR, WHILE, REPEAT) before it would otherwise terminate. Program execution resumes with the statement following the loop terminator (END\_FOR, END\_WHILE, END\_REPEAT). An EXIT statement is typically used within an IF statement.

#### Format

**EXIT;**

Where:

*ConditionForExiting*     An expression that determines whether to terminate early.

#### Example

The following code fragment shows the operation of the EXIT statement. When the variable number equals 10, the WHILE loop is exited and execution continues with the statement immediately following END\_WHILE.

```
while (1) do  
  a := a + 1;  
  IF (a = 10) THEN  
    EXIT;  
  END_IF;  
END_WHILE;
```

## Chapter 9 *Diagnostics*

---

This chapter explains the PACSystems fault handling system, provides definitions of fault extra data, and suggests corrective actions for faults.

Faults occur in the control system when certain failures or conditions happen that affect the operation and performance of the system. Some conditions, such as the loss of an I/O module or rack, may impair the ability of the PACSystems controller to control a machine or process. Other conditions, such as when a new module comes online and becomes available for use, may be displayed to inform or alert the user.

Any detected fault is recorded in the Controller Fault Table or the I/O Fault Table, as applicable.

Information in this chapter is organized as follows:

- *Fault Handling Overview*
- *Using the Fault Tables*
- *System Handling of Faults*
- *Controller Fault Descriptions and Corrective Actions*
- *I/O Fault Descriptions and Corrective Actions*
- *Diagnostic Logic Blocks (DLBs)*

## 9.1 Fault Handling Overview

The PACSystems CPU detects three classes of faults:

<b>Fault Class</b>	<b>Examples</b>
Internal Failures (Hardware)	Non-responding modules Failed battery Failed Energy Pack (CPE302/CPE305/CPE310/CPE330 models) Memory checksum errors
External I/O Failures (Hardware)	Loss of rack or module Addition of rack or module Loss of Genius I/O block
Operational Failures	Communication failures Configuration failures Password access failures

### 9.1.1 System Response to Faults

Hardware failures require that either the system be shut down or the failure be tolerated. I/O failures may be tolerated by the control system, but they may be intolerable by the application or the process being controlled. Operational failures are normally tolerated.

Faults have three attributes:

Fault Table Affected	I/O Fault Table Controller Fault Table
Fault Action	Fatal Diagnostic Informational
Configurability	Configurable Non-configurable

### 9.1.2 Fault Tables

The PACSystems CPU maintains two fault tables, the Controller Fault Table for internal CPU faults and the I/O Fault Table for faults generated by I/O devices (including I/O controllers). For more information, refer to *Using the Fault Tables* below.

### 9.1.3 Fault Actions and Fault Action Configuration

Fatal faults cause the fault to be recorded in the appropriate table, diagnostic variables to be set, and the system to be stopped. Only fatal faults cause the system to stop.

Diagnostic faults are recorded in the appropriate table, and any diagnostic variables are set. Informational faults are only recorded in the appropriate table.

<b>Fault Action</b>	<b>Response by CPU</b>
Fatal	Log fault in fault table. Set fault references. Go to STOP/Fault Mode.
Diagnostic	Log fault in fault table. Set fault references.
Informational	Log fault in fault table.

The hardware configuration can be used to specify the fault action of some fault groups. For these groups, the fault action can be configured as either fatal or diagnostic. When a fatal or diagnostic fault within a configurable group occurs, the CPU executes the configured fault action instead of the action specified within the fault.

**Note:** The fault action displayed in the expanded fault details indicates the fault action specified by the fault that was logged, but not necessarily the executed fault action. To determine what action was executed for a particular fault in a configurable fault group, you must refer to the hardware configuration settings.

#### Faults that are part of configurable fault groups:

<b>Fault Action Displayed in Fault Table</b>	Informational	Diagnostic	Fatal
<b>Fault Action Executed</b>	Informational	Diagnostic or Fatal. Determined by action selected in Hardware Configuration.	Diagnostic or Fatal. Determined by action selected in Hardware Configuration.

#### Faults that are part of non-configurable fault groups:

<b>Fault Action Displayed in Fault Table</b>	Informational	Diagnostic	Fatal
<b>Fault Action Executed</b>	Informational	Diagnostic	Fatal

## 9.2 Using the Fault Tables

To display the fault tables in Logic Developer software,

1. Go online with the PACSystems.
2. Select the Project tab in the Navigator, right click the Target node and choose Diagnostics. The Fault Table Viewer appears.

The Controller Fault Table and the I/O Fault Table display the following information:

<b>Controller Time/Date</b>	The current date and time of the CPU.
<b>Last Cleared</b>	The date and time faults were last cleared from the fault table. This information is maintained by the PACSystems controller.
<b>Status</b>	Displays <i>Updating</i> while the programmer is reading the fault table. Status is <i>Online</i> when update is complete.
<b>Total Faults</b>	The total number of faults since the table was last cleared.
<b>Entries Overflowed</b>	The number of entries lost because the fault table has overflowed since it was cleared. Each fault table can contain up to 64 faults.

### 9.2.1 Controller Fault Table

The Controller Fault Table displays CPU faults such as password violations, configuration mismatches, parity errors, and communications errors.

Figure 18: Controller Fault Table Display

The Controller Fault Table provides the following information for each fault:

- Location** Identifies the location of the fault by rack.slot.
- Description** Corresponds to a fault *group*, which is identified in the fault Details.
- Date/Time** The date and time the fault occurred based on the CPU clock.
- Details** To view detailed information, click the fault entry. Refer to *Viewing Controller Fault Details* for more information.

## Viewing Controller Fault Details

**Note:** The fault action displayed in the expanded fault details indicates the fault action specified by the fault that was logged, but not necessarily the executed fault action. To determine what action was executed for a particular fault in a configurable fault group, you must refer to the hardware configuration settings.

To see controller fault details, click the fault entry. The detailed information box for the fault appears. (To close this box, click the fault.)

0.1	Failed battery signal			01-02-2000 19:06:59
...	Error Code	Group	Action	Task Num
	0	18	2:Diagnostic	0
	Fault Extra Data: 02 00			

**Figure 19: Detail Information for Controller Fault Entry**

The detailed information for controller faults includes the following:

- Error Code** Further identifies the fault. Each fault group has its own set of error codes.
- Group** Group is the highest classification of a fault and identifies the general category of the fault. The fault description text displayed by your programming software is based on the fault group and the error codes.
- Action** Fatal, Diagnostic, or Informational. For definitions of these actions, refer to *Fault Actions and Fault Action Configuration*.
- Task Number** Not used for most faults. When used, provides additional information for Technical Support representatives.
- Fault Extra Data** Provides additional information for diagnostics by Technical Support engineers. Explanations of this information are provided as appropriate for specific faults in *Controller Fault Descriptions and Corrective Actions* below.

## User-Defined Faults

User-defined faults can be logged in the Controller Fault Table. When a user-defined fault occurs, it is displayed in the appropriate fault table as *Application Msg (error\_code)*: and may be followed by a descriptive message up to 24 characters. The user can define all characters in the descriptive message. Although the message must end with the null character, e.g., zero (0), the null character does not count as one of the 24 characters. If the message contains more than 24 characters, only the first 24 characters are displayed.

Certain user-defined faults can be used to set a system status reference (%SA0081–%SA0112).

User-defined faults are created using SVC\_REQ 21: User-Defined Fault Logging, which is described in Chapter 6.

**Note:** When a user-defined fault is displayed in the Controller Fault table, a value of -32768 (8000 hex) is added to the error code. For example, the error code 5 will be displayed as -32763.

## 9.2.2 I/O Fault Table

The I/O Fault Table displays I/O faults such as circuit faults, address conflicts, forced circuits, I/O module addition/loss faults and I/O bus faults.

The fault table displays a maximum of 64 faults. When the fault table is full, it displays the earliest 32 faults (33–64) and the last 32 faults (1–32). When another fault is received, fault 32 is shoved out of the table. In this way, the first 32 faults are preserved for the user to view.

Fault Table Viewer							Status
Choose Fault Table <input type="radio"/> PLC <input checked="" type="radio"/> I/O		PLC Date/Time: 09-22-2005 12:41:56 Last Cleared: 09-13-2005 12:06:57				Online	
I/O Fault Table (Displaying 27 of 27 faults, 0 Overflowed)							
Loc	CIRC No.	Variable Name	Ref. Address	Fault Category	Fault Type	Date/Time	
0.3	n/a			Loss of I/O Module		09-22-2005 03:27:38	
0.5	n/a	ai1		Loss of I/O Module		09-22-2005 03:27:38	
0.6	n/a	in1		Loss of I/O Module		09-22-2005 03:27:38	
0.3	n/a			Loss of I/O Module		09-22-2005 03:24:51	
0.5	n/a	ai1		Loss of I/O Module		09-22-2005 03:24:51	
0.6	n/a	in1		Loss of I/O Module		09-22-2005 03:24:51	

Figure 20: I/O Fault Table Display

The I/O Fault Table provides the following information for each fault:

- Location** Identifies the location of the fault by rack.slot location, and sometimes bus and buss address.
- CIRC No.** When applicable, identifies the specific I/O point on the module.
- Variable Name** If the fault is on a point that is mapped to an I/O variable, and the variable is set to publish (either internal or external), the I/O Fault Table displays the variable name. Unpublished I/O variables will not be displayed in this field.
- Ref. Address** If the fault is on a point that is mapped to a reference address, this field identifies the I/O memory type and location (offset) that corresponds to the point experiencing the fault. When a Genius device fault or local analog module fault occurs, the reference address refers to the first point on the block where the fault occurred.
- Note:** The Reference Address field displays 16 bits and %W memory has a 32-bit range. Addresses in %W are displayed correctly for offsets in the 16-bit range ( $\leq 65,535$ ). For %W offsets greater than 16 bits, the I/O Fault Table displays a blank reference address.
- Fault Category** Specifies a general classification of the fault.
- Fault Type** Consists of subcategories under certain fault categories. Set to zero when not applicable to the category.
- Date/Time** The date and time the fault occurred based on the CPU clock.

**Details** To view detailed information, click the fault entry. Refer to *Viewing I/O Fault Details* for more information.

**Viewing I/O Fault Details**

To see I/O fault details, click the fault entry. The detailed information box for the fault appears. (To close this box, click the fault.)

0.3	1	%AQ 00001	Circuit Fault	Analog Fault	01-01-2000 00:02:27		
.....	I/O Bus	Bus Address	Point Address	Group	Action	Category	Fault Type
	n/a	n/a	1	10	2:Diagnostic	1	2
	Fault Extra Data	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00					
	Fault Description	Input Open Wire					

**Figure 21: I/O Fault Table Fault Entry Detail Display**

The detailed information for I/O faults includes:

- I/O Bus** When the module in the slot is a Genius Bus Controller (GBC), this number is always one.
- Bus Address** The serial bus address of the Genius device that reported or has the fault.
- Point Address** Identifies the point on the I/O device that has the fault when the fault is a point-type fault.
- Group** Fault group is the highest classification of a fault. It identifies the general category of the fault.
- Action** Fatal, Diagnostic, or Informational. For definitions of these actions, refer to *Fault Actions and Fault Action Configuration*.
- Category** Identifies the category of the fault.
- Fault Type** Identifies the fault type by number. Set to zero when not applicable to the category.
- Fault Extra Data** Provides additional information for diagnostics by Technical Support engineers. Explanations of this information are provided as appropriate for specific faults in *I/O Fault Descriptions and Corrective Actions*.
- Fault Description** Provides a specific fault code when the I/O fault category is a circuit fault (discrete circuit fault, analog circuit fault, low-level analog fault) or module fault. It is set to zero for other fault categories.

### 9.3 System Handling of Faults

The system fault references listed below can be used to identify the specific type of fault that has occurred. (A complete list of *System Status References* is provided in Chapter 3.)

<b>System Fault Reference</b>	<b>Address</b>	<b>Description</b>
#ANY_FLT	%SC0009	Any new fault in either table since the last power-up or clearing of the fault tables
#SY_FLT	%SC0010	Any new system fault in the Controller Fault Table since the last power-up or clearing of the fault tables
#IO_FLT	%SC0011	Any new fault in the I/O Fault Table since the last power-up or clearing of the fault tables
#SY_PRES	%SC0012	Indicates that there is at least one entry in the Controller Fault Table
#IO_PRES	%SC0013	Indicates that there is at least one entry in the I/O Fault Table
#HRD_FLT	%SC0014	Any hardware fault
#SFT_FLT	%SC0015	Any software fault

On power-up, the system fault references are cleared. If a fault occurs, the positive contact transition of any affected reference is turned on the sweep after the fault occurs. The system fault references remain on until both fault tables are cleared or All Memory in the CPU is cleared.

### 9.3.1 System Fault References

When a system fault reference is set, additional fault references are also set. These other types of faults are listed in *Fault References for Configurable Faults* below and *Fault References for Non-Configurable Faults* in the section which follows.

#### Fault References for Configurable Faults

<b>Fault</b> (Default Action)	<b>Address</b>	<b>Description</b>	<b>May Also Be Set</b>
#SBUS_ER (diagnostic)	%SA0032	System bus error. All system bus error faults are logged as informational.	#HRD_FLT, #SY_PREP, #SY_FLT
#SFT_IOC <sup>13</sup> (diagnostic)	%SA0029	Non-recoverable software error in an I/O Controller (IOC).	#IO_FLT, #IO_PREP, #SFT_FLT
#LOS_RCK <sup>14</sup> (diagnostic)	%SA0012	Loss of rack (BRM failure, loss of power) or missing a configured rack.	#SY_FLT, #SY_PREP, #IO_FLT, #IO_PREP
#LOS_IOC <sup>15</sup> (diagnostic)	%SA0013	Loss of I/O Controller or missing a configured Bus Controller.	#IO_FLT, #IO_PREP
#LOS_IOM (diagnostic)	%SA0014	Loss of I/O module (does not respond), or missing a configured I/O module.	#IO_FLT, #IO_PREP
#LOS_SIO (diagnostic)	%SA0015	Loss of intelligent module (does not respond), or missing a configured module.	#SY_FLT, #SY_PREP
#IOC_FLT (diagnostic)	%SA0022	Non-fatal bus or I/O Controller error, more than 10 bus errors in 10 seconds. (Error rate is configurable.)	#IO_FLT, #IO_PREP
#CFG_MM (fatal)	%SA0009	Configuration mismatch. Wrong module type detected. The CPU does not check the configuration parameter settings for individual modules such as Genius I/O blocks.	#SY_FLT, #SY_PREP
#OVR_TMP (diagnostic)	%SA0008	CPU temperature has exceeded its normal operating temperature.	#SY_FLT, #SY_PREP

**Note:** If the fault action for a fault logged to the fault table is informational, the configured action is not used. For example, if the logged fault action for an SBUS\_ERR is informational, but you configure it as fatal, the action is still informational.

<sup>13</sup> The #SFT\_IOC software fault will have the same action as what you set for #LOS\_IOC.

<sup>14</sup> When a Loss of Rack or Addition of Rack fault is logged, individual loss or add faults for each module in that rack are usually not generated.

<sup>15</sup> Even if the #LOS\_IOC fault is configured as Fatal, the CPU will not go to STOP/FAULT unless both GBCs of an internal redundant pair fail.

## Fault References for Non-Configurable Faults

<b>Fault</b>	<b>Address</b>	<b>Description</b>	<b>Result</b>
#PS_FLT	%SA0005	Power supply fault	Sets #SY_FLT, #SY_PRE
#HRD_CPU (fatal)	%SA0010	CPU hardware fault (such as failed memory device or failed serial port).	Sets #SY_FLT, #SY_PRE, #HRD_FLT
#HRD_SIO (diagnostic)	%SA0027	Non-fatal hardware fault on any module in the system, such as failure of a serial port on a LAN interface module.	Sets #SY_FLT, #SY_PRE, #HRD_FLT
#PNIO_ ALARM	%SA0030	A diagnostic PROFINET alarm has been received and an I/O fault has been logged in group 28.	Sets #ANY_FLT, #IO_FLT, #IO_PRE
#SFT_SIO (diagnostic)	%SA0031	Non-recoverable software error in a LAN interface module.	Sets #SY_FLT, #SY_PRE, #SFT_FLT
#PB_SUM (fatal)	%SA0001	Program or block checksum failure during power-up or in RUN Mode.	Sets #SY_FLT, #SY_PRE
#LOW_BAT (diagnostic)	%SA0011	The low battery indication is not supported for all CPU versions. For details, refer to <i>Battery Status (Group 18)</i> .	Sets #SY_FLT, #SY_PRE
#OV_SWP (diagnostic)	%SA0002	Constant sweep time exceeded.	Sets #SY_FLT, #SY_PRE
#SY_FULL #IO_FULL (diagnostic)	%SA0022	Controller fault table full (64 entries). I/O Fault Table full (64 entries).	Sets #SY_FLT, #SY_PRE, #IO_FLT, #IO_PRE
#APL_FLT (diagnostic)	%SA0003	Application fault.	Sets #SY_FLT, #SY_PRE
#ADD_RCK <sup>14</sup> (diagnostic)	%SA0017	New rack added, extra rack, or previously faulted rack has returned.	Sets #SY_FLT, #SY_PRE
#ADD_IOC (diagnostic)	%SA0018	Extra IOC, previously faulted I/O Controller is no longer faulted.	Sets #IO_FLT, #IO_PRE
#ADD_IOM (diagnostic)	%SA0019	Extra IO module, or previously faulted I/O module is no longer faulted.	Sets #IO_FLT, #IO_PRE
#ADD_SIO (diagnostic)	%SA0020	New intelligent module is added, or previously faulted module no longer faulted.	Sets #SY_FLT, #SY_PRE
#IOM_FLT (diagnostic)	%SA0023	Point or channel on an I/O module; a partial failure of the module.	Sets #IO_FLT, #IO_PRE
#NO_PROG (information)	%SB0009	No application program is present at power-up. Should only occur the first time the PACSystems controller is powered up or if the user memory containing the program fails.	CPU will not go to RUN Mode; it continues executing STOP Mode sweep until a valid program is loaded. This can be a <i>null</i> program that does nothing. Sets #SY_FLT and #SY_PRE.

<b>Fault</b>	<b>Address</b>	<b>Description</b>	<b>Result</b>
#BAD_RAM (fatal)	%SB0010	Corrupted program memory at power-up. Program could not be read and/or did not pass checksum tests.	Sets #SY_FLT and #SY_PRES.
#WIND_ER (information)	%SB0001	Window completion error. Servicing of Controller Communications or Logic Window was skipped. Occurs in Constant Sweep mode.	Sets #SY_FLT and #SY_PRES.
#BAD_PWD (information)	%SB0011	Change of privilege level request to a protection level was denied; bad password.	Sets #SY_FLT and #SY_PRES.
#NUL_CFG (fatal)	%SB0012	No configuration present upon transition to RUN Mode. Running without a configuration is equivalent to suspending the I/O scans.	Sets #SY_FLT and #SY_PRES.
#SFT_CPU (fatal)	%SB0013	CPU software fault. A non-recoverable error has been detected in the CPU. May be caused by Watchdog Timer expiring.	CPU immediately transitions to STOP/Halt Mode. The only activity permitted is communication with the programmer. To be cleared, controller power must be cycled. Sets SY_FLT, SY_PRES, and SFT_FLT.
#STOR_ER (fatal)	%SB0014	Download of data to CPU from the programmer failed; some data in CPU may be corrupted.	CPU will not transition to RUN Mode. This fault is not cleared at power-up, intervention is required to correct it. Sets SY_FLT and SY_PRES.

### 9.3.2 Using Fault Contacts

Fault (-[F]-) and no-fault (-[NF]-) contacts can be used to detect the presence of I/O faults in the system. These contacts cannot be overridden. The following table shows the state of fault and no-fault contacts.

<b>Condition</b>	<b>[F]</b>	<b>[NF]</b>
Fault Present	ON	OFF
Fault Absent	OFF	ON

An NF contact will be ON (F contact will be OFF) when the referenced I/O point is not faulted, **or** the referenced I/O point does not exist in the hardware configuration.

#### Fault Locating References (Rack, Slot, Bus, Module)

The PACSystems CPU supports reserved fault names for each rack, slot, bus, and module. By programming these names on the FAULT and NOFLT contact instructions, logic can be executed in response to faults associated with configured racks and modules.

#### Fault Locating Reference Name Format

These fault names can only be programmed on the FAULT and NOFLT contacts. The reserved fault names are always available. It is not necessary to enable a special option, such as point faults.

<b>Fault Reference Type</b>	<b>Reserved Name</b>	<b>Comment</b>
Rack	#RACK_000r	Where r is rack number 0 to 7.
Slot	#SLOT_0rss	Where r is rack number 0 to 7 and ss is slot number 0 to 31.
Bus	#BUS_0rssb (Genius only)	Where r is rack number 0 to 7, ss is slot number 0 to 31, and b is the bus number (1 or 2).
Module	#M_rssbmmm (Genius only)	Where r is rack number 0 to 7, ss is slot number 0 to 31, b is the bus number (1 or 2), and mmm is the Bus Address number 000 to 255.

These fault names do not correspond to %SA, %SB, %SC, or to any other reference type. They are mapped to a memory area that is not user-accessible. Only the name is displayed.

**Fault Reference Name Examples:**

#RACK\_0001 represents rack 1.

#SLOT\_0105 represents rack 1, slot 5.

#BUS\_02041 represents rack 2, slot 4, bus 1.

#M\_2061028 represents rack 2, slot 6, bus 1, Genius module 28.

**Note:** When a slot level failure fault is reported to the fault tables, all bus and module fault locating references associated with that slot are set (the FAULT contact passes power flow, and the NOFLT contact does not pass power flow), regardless of what type of module it is. Conversely, when a slot level reset fault is reported to the fault tables, all bus and module fault locating references are cleared (the FAULT contact does not pass power flow, and the NOFLT contact passes power flow).

**Behavior of Fault Locating References**

At power-up, all fault locating references are cleared in the CPU. When a fault is logged, the CPU transitions the state of the affected reference(s). The state of the fault reference remains in the fault state until one of the following actions occurs:

- Both the Controller and the I/O Fault Tables are cleared through your programming software either by clearing each table individually or clearing the entire CPU memory.
- The associated device (rack, I/O module, or Genius device) is added back into the system. Whenever an *Addition of . . .* fault is logged, the CPU initializes all fault references associated with the device to the NoFlt state. These references remain in the NoFlt state until another fault associated with the device is reported. (This could take several seconds for distributed I/O faults, especially if the bus controller has been reset.)

**Note:** These fault references are set for informational purposes only. They should not be used to qualify I/O data. The Alarm Contacts (described in *Using Alarm Contacts*) may be used to qualify I/O data. The CPU does not halt execution as a result of setting a fault locating reference to the Fault state.

The fault references have a cascading effect. If there is a problem in the module located at rack 5, slot 6, bus 1, module 29, the following fault references are set: RACK\_05, SLOT\_0506, BUS\_05061, and M\_5061029. There will only be one entry in the fault table to describe the problem with the module. The fault table does not show separate entries pertaining to the rack, slot, and bus in this case.

If an analog base module (IC697ALG230) is lost, the fault locating reference for that module is set. The fault locating references for its expander modules (IC697ALG440 and ALG441) are not set as a result of the loss. Therefore, any fault locating references to an expander module should also reference the base module to verify that the module or its base have not been lost.

### 9.3.3 Using Point Faults

Point faults pertain to external I/O faults, although they are also set due to the failure of associated higher-level internal hardware (for example, IOC failure or loss of a rack). To use point faults, they must be enabled in Hardware Configuration on the Memory parameters tab of the CPU.

When enabled, a bit for each discrete I/O point and a byte for each analog I/O channel are allocated in CPU memory. The CPU memory used for point faults is included in the total reference table memory size. The FAULT and NOFLT contacts, described in *Using Alarm Contacts*, provide access to the point faults.

The full support of point fault contacts depends on the capability of the I/O module. Some Series 90-30 modules do not support point fault contacts. The point fault contacts for these modules remain all off, unless a Loss of I/O Module occurs, in which case the RX3i CPU turns on all point fault contacts associated with the lost module.

### 9.3.4 Using Alarm Contacts

High (-[HA]-) and low (-[LA]-) alarm contacts are used to represent the state of the analog input module comparator function. To use alarm contacts, point faults must first be enabled in Hardware Configuration on the Memory parameters tab of the CPU.

The following example logic uses both high and low alarm contacts.



**Note:** HA and LA contacts do not create an entry in a fault table.

## 9.4 Controller Fault Descriptions and Corrective Actions

Each fault explanation contains a fault description and instructions to correct the fault. Many fault descriptions have multiple causes. In these cases, the error code and additional fault information are used to distinguish among fault conditions sharing the same fault description.

### 9.4.1 Controller Fault Groups

<b>Group</b>	<b>Name</b>	<b>Default Fault Action<sup>16</sup></b>	<b>Configurable</b>
1	Loss of or Missing Rack	Diagnostic	Yes
4	Loss of or Missing Option Module	Diagnostic	Yes
5	Addition of, or Extra Rack	N/A	No
8	Reset of, Addition of, or Extra Option Module	N/A	No
11	System Configuration Mismatch	Fatal <sup>17</sup>	Yes
12	System Bus Error	Fatal	Yes
13	CPU Hardware Failure	N/A	No
14	Module Hardware Failure	N/A	No
16	Option Module Software Failure	N/A	No
17	Program or Block Checksum Failure Group	N/A	No
18	Battery Status Group	N/A	No
19	Constant Sweep Time Exceeded	N/A	No
20	System Fault Table Full	N/A	No
21	I/O Fault Table Full	N/A	No
22	User Application Fault	N/A	No
24	CPU Over Temperature	Diagnostic	Yes
128	System Bus Failure	N/A	No
129	No User Program on Power-up	N/A	No
130	Corrupted User Program on Power-up	N/A	No
131	Window Completion Failure	N/A	No
132	Password Access Failure	N/A	No
134	Null System Configuration for RUN Mode	N/A	No
135	CPU System Software Failure	N/A	No
137	Communications Failure During Store	N/A	No
140	Non-critical CPU Software Event	N/A	No

<sup>16</sup> The fault action indicated is not applicable if the fault is displayed as informational. Faults displayed as informational, always behave as informational.

<sup>17</sup> If a system configuration mismatch occurs when the CPU is in RUN Mode, the fault action will be Diagnostic regardless of the fault configuration. For additional information, refer to *Fault Parameters* in *PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual*, GFK-2222.

## 9.4.2 Loss of or Missing Rack (Group 1)

The fault group Loss of or Missing Rack occurs when the system cannot communicate with an expansion rack because the BTM (Bus Transmitter Module) in the main rack failed, the BRM (Bus Receiver Module) in the expansion rack failed, power failed in the expansion rack, or the expansion rack was configured in the configuration file but did not respond during power-up.

Default action: Diagnostic. Configurable.

### 1, Rack Lost

The CPU generates this error when the main rack can no longer communicate with an expansion rack. The error is generated for each expansion rack that exists in the system.

#### Correction

- 1) Power off the system. Verify that both the BTM and the BRM are seated properly in their respective racks and that all cables are properly connected and seated.
- 2) Replace the cables.
- 3) Replace the BRM.
- 4) Replace the BTM.

### 2, Rack Not Responding

The CPU generates this error when the configuration file stored from the programmer indicates that a particular expansion rack should be in the system but none responds for that rack number.

#### Correction

- 1) Check rack number jumper behind power supply—first on missing rack and then on all other racks—for duplicated rack numbers.
- 2) Update the configuration file if a rack should not be present.
- 3) Add the rack to the hardware configuration if a rack should be present and one is not.
- 4) Power off the system. Verify that both the BTM and the BRM are seated properly in their respective racks and that all cables are properly connected and seated.
- 5) Replace the cables.
- 6) Replace the BRM.
- 7) Replace the BTM.
- 8) Check for Termination Plug on last BRM.

### 9.4.3 Loss of or Missing Option Module (Group 4)

The fault group Loss of or Missing Option Module occurs when a LAN interface module, BTM, or BRM fails to respond. The failure may occur at power-up or store of configuration if the module is missing or during operation if the module fails to respond. This may also occur due to hot removal of an option module.

Default action: Diagnostic. Configurable

#### 3C hex/60 decimal, Module in Firmware Update Mode

The CPU generates this error when it finds a module in Firmware Update mode. Modules in this mode will not communicate with the CPU.

##### Correction

- 1) Run the firmware update utility for the module.
- 2) Reset the module with the push-button.
- 3) Power-cycle the entire system.
- 4) Power-cycle the rack containing the module.

#### 63 hex/99 decimal, Module Hot Removed

The CPU logs this fault when it detects hot removal of an option module such as the LAN interface module. No correction necessary

#### All Others, Module Failure During Configuration

The CPU generates this error when a module fails during power-up or configuration store.

##### Correction

- 1) Power off the system. Replace the module located in that rack and slot.
- 2) If the board is located in an expansion rack, verify BTM/BRM cable connections are tight and the modules are seated properly; verify the addressing of the expansion rack.
- 3) Replace the BTM.
- 4) Replace the BRM.
- 5) Replace the rack.

### 9.4.4 Addition of, or Extra Rack (Group 5)

This fault group occurs when a configured expansion rack with which the CPU could not communicate comes online or is powered on, or an unconfigured rack is found.

Action: Non-configurable.

#### 1, Extra Rack

##### Correction

- 1) Check rack jumper behind power supply for correct setting.
- 2) Update the configuration file to include the expansion rack.

**Note:** No correction necessary if rack was just powered on.

### 9.4.5 Reset of, Addition of, or Extra Option Module (Group 8)

The fault group Reset of, Addition of, or Extra Option Module occurs when an option module (LAN interface module, BTM, etc.) comes online, is reset, is hot inserted or a module is found in the rack but is not configured.

Action: Non-configurable.

#### 3, LAN Interface Restart Complete, Running Utility

The LAN Interface module has restarted and is running a utility program.

#### Correction

Refer to the LAN Interface manual, GFK-0868 or GFK-0869 (previously GFK-0533).

#### 7, Extra Option Module

**Note:** This fault is logged for an RX3i CPE310 that is configured as a CPU310, or a CPE330 configured as a CPU320, because the RX3i system detects the embedded Ethernet module as an unconfigured module.

#### Correction

- 1) Update the configuration file to include the module.
- 2) Remove the module from the system.

#### E Hex/14 Decimal, Option Module Hot inserted

The CPU logs this fault when it detects hot insertion of an option module such as the LAN interface module. No correction necessary

**Note:** When configuration is cleared or stored, a reset fault is generated for every intelligent option module physically present in the system.

### 9.4.6 System Configuration Mismatch (Group 11)

The fault group Configuration Mismatch occurs when the module occupying a slot is different from that specified in the configuration file. When the GBC generates the mismatch because of a Genius block, the second byte in the Fault Extra Data field contains the bus address of the mismatched block.

Default action: Fatal. Configurable.

**Note:** If a system configuration mismatch occurs when the CPU is in RUN Mode, the fault action will be Diagnostic regardless of the fault configuration. See *Fault Parameters* in *PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual*, GFK-2222.

#### 2, Genius I/O Block Model Number Mismatch

The CPU generates this fault when the configured and physical Genius I/O blocks have different model numbers.

#### Correction

- 1) Replace the Genius I/O block with one corresponding to the configured module.
- 2) Update the configuration file.

#### Fault Extra Data for Genius I/O Block Model Number Mismatch

Byte	Value
[0]	FF (flag byte)
[1]	Serial Bus address
[2]	Installed module type (refer to <i>Installed/Configured Module Types (Bytes 2 and 3 of Fault Extra Data)</i> below).
[3]	Configured module type (refer to <i>Installed/Configured Module Types (Bytes 2 and 3 of Fault Extra Data)</i> below).

#### Installed/Configured Module Types (Bytes 2 and 3 of Fault Extra Data)

Number		Description
Decimal	Hexadecimal	
4	4	Genius Network Interface (GENI)
5	5	Phase B Hand Held Monitor
6	6	Phase B Series Six GBC with Diagnostics
7	7	Phase B Series Six GBC without Diagnostics
8	8	PLCM/Series Six
9	9	PLCM/Series 90-70
10	A	Series 90-70 Single Channel Bus Controller
11	B	Series 90-70 Dual Channel Bus Controller
12	C	Series 90-10 Genius Communications Module
13	D	Series 90-30 Genius Communications Module
32	20	High Speed Counter
69	45	Phase B 115Vac 8-point (2 amp) Grouped Block
70	46	Phase B 115Vac/125Vdc 8-point Isolated Block
70	46	Phase B 115Vac/125Vdc 8-point Isolated Block without Failed Switch
71	47	Phase B 220Vac 8-point Grouped Block

<b>Number</b>		<b>Description</b>
<b>Decimal</b>	<b>Hexadecimal</b>	
72	48	Phase B 24-48Vdc 16-point Proximity Sink Block
72	48	Phase B 24Vdc 16-point Proximity Sink Block
73	49	Phase B 24-48Vdc 16-point Source Block
73	49	Phase B 24Vdc 16-point Proximity Source Block
74	4A	Phase B 12-24Vdc 32-point Sink Block
75	4B	Phase B 12-24Vdc 32-point Source Block
76	4C	Phase B 12-24Vdc 32-point 5V Logic Block
77	4D	Phase B 115Vac 16-point Quad State Input Block
78	4E	Phase B 12-24Vdc 16-point Quad State Input Block
79	4F	Phase B 115/230Vac 16-point Normally Open Relay Block
80	50	Phase B 115/230Vac 16-point Normally Closed Relay Block
81	51	Phase B 115Vac 16-point AC Input Block
82	52	Phase B 115Vac 8-point Low-Leakage Grouped Block
127	7F	Genius Network Adapter (GENA). Refer to <i>GENA Application ID Numbers</i> below.
131	83	Phase B 115Vac 4-input, 2-output Analog Block
132	84	Phase B 24Vdc 4-input, 2-output Analog Block
133	85	Phase B 220Vac 4-input, 2-output Analog Block
134	86	Phase B 115Vac Thermocouple Input Block
135	87	Phase B 24Vdc Thermocouple Input Block
136	88	Phase B 115Vac RTD Input Block
137	89	Phase B 24/48Vdc RTD Input Block
138	8A	Phase B 115Vac Strain Gauge/mV Analog Input Block
139	8B	Phase B 24Vdc Strain Gauge/mV Analog Input Block
140	8C	Phase B 115Vac 4-input, 2-output Current Source Analog Block
141	8D	Phase B 24Vdc 4-input, 2-output Current Source Analog Block

### GENA Application ID Numbers

If the model number is 7F hex (Genius Network Adapter), the block may be one of the following. (The GENA Application ID is shown for reference.)

<b>Number</b>		<b>Description</b>
<b>Decimal</b>	<b>Hexadecimal</b>	
131	83	115Vac/230Vac/125Vdc Power Monitor Module
132	84	24/48Vdc Power Monitor Module
160	A0	Genius Remote 90-70 Rack Controller

#### 4, I/O Type Mismatch

The CPU generates this fault when the physical and configured I/O types of Genius grouped blocks are different.

##### Correction

- 1) Remove the indicated Genius module and install the module indicated in the configuration file.
- 2) Update the Genius module descriptions in the configuration file to agree with what is physically installed.

##### Fault Extra Data for I/O Type Mismatch

Byte	Value
[0]	FF
[1]	Bus address
[2]	Installed module's I/O type
[3]	Configured module's I/O type

##### Genius Installed Module I/O Types (Byte 2 of Fault Extra Data)

Value	Description
01	Input only
02	Output only
03	Combination

##### Genius Configured Module I/O Types (Byte 3 of Fault Extra Data)

Decimal	Value		Description
	Decimal	Hexadecimal	
0	0	0	Discrete input
1	1	1	Discrete output
2	2	2	Analog input
3	3	3	Analog output
4	4	4	Discrete grouped
5	5	5	Analog grouped
20	14	14	Analog in, discrete in
21	15	15	Analog in, discrete out
24	18	18	Analog in, discrete grouped
30	1E	1E	Analog out, discrete in
31	1F	1F	Analog out, discrete out
34	22	22	Analog out, discrete grouped
50	32	32	Analog grouped, discrete in
51	33	33	Analog grouped, discrete out
54	36	36	Analog grouped, discrete grouped

### 8, Analog Expander Mismatch

The CPU generates this error when the configured and physical Analog Expander modules have different model numbers.

#### Correction

- 1) Replace the Analog Expander module with one corresponding to configured module.
- 2) Update the configuration file.

### 9, Genius I/O Block Size Mismatch

The CPU generates this error when block configuration size does not match the configured size.

#### Correction

Reconfigure the block.

#### Fault Extra Data for Genius I/O Block Size Mismatch

Byte	Value
[0]	FF
[1]	Bus address
[2]	Module's broadcast data length
[3]	Configured module's broadcast data length

### A hex/10 decimal, Unsupported Feature

Configured feature not supported by this revision of the module.

#### Correction

- 1) Update the module to a revision that supports the feature.
- 2) Change the module configuration.

#### Fault Extra Data for Unsupported Feature

Byte	Value
[8]	Contains a reason code indicating what feature is not supported. 0x5 - GBC revision too old 0x6 - Only supported in main rack

### E hex/14 decimal, LAN Duplicate MAC Address

This LAN Interface module has the same MAC address as another device on the LAN. The module is off the network.

#### Correction

- 1) Change the module's MAC address.
- 2) Change the other device's MAC address.

### F hex/15 decimal, LAN Duplicate MAC Address Resolved

Previous duplicate MAC address has been resolved. The module is back on the network. This is an informational message. No correction required.

**10 hex/16 decimal, LAN MAC Address Mismatch**

MAC address programmed by softswitch utility does not match configuration stored from software.

**Correction**

Change MAC address on softswitch utility or in software.

**11 hex/17 decimal, LAN Softswitch/Modem mismatch**

Configuration of LAN module does not match modem type or configuration programmed by softswitch utility.

**Correction**

- 1) Correct configuration of modem type.
- 2) Consult LAN Interface manual for configuration setup.

**13 hex/19 decimal, DCD Length Mismatch**

Directed control data lengths do not match.

**Correction**

See Fault Extra Data.

**Fault Extra Data for DCD Length Mismatch**

<b>Byte</b>	<b>Value</b>
[0]	FF
[1]	Bus address
[2]	Module's directed data length
[3]	Configured module's directed data length

**25 hex/37 decimal, Controller Reference Out-of-Range**

A reference on either the trigger, disable, or I/O specification is out of the configured limits.

**Correction**

Modify the incorrect reference to be within range, or increase the configured size of the reference data.

**26 hex/38 decimal, Bad Program Specification**

The I/O specification of a program is corrupted.

**Correction**

Contact Technical Support.

**27 hex/39 decimal, Unresolved or Disabled Interrupt Reference**

The CPU generates this error when an interrupt trigger reference is either out of range or disabled in the I/O module's configuration.

**Correction**

- 1) Remove or correct the interrupt trigger reference.
- 2) Update the configuration file to enable this particular interrupt.

### **43 hex/67 decimal, Module Configuration Failure**

Module configuration was not successfully accepted by the module.

#### **Correction**

Check fault table for other module-specific faults for possible reasons why the module did not accept the configuration. Check that the configuration for the module is correct and valid.

### **4B hex/75 decimal, ECC jumper is disabled, but should be enabled**

If the CPU redundancy feature is supported and required, the ECC jumper must be in the enabled position.

#### **Correction**

Set the ECC jumper to the enabled position. (See the instructions provided with the Redundancy CPU firmware upgrade kit).

### **4C hex/76 decimal, ECC jumper is enabled, but should be disabled**

If the CPU firmware does not support redundancy, the ECC jumper must be in the disabled position.

#### **Correction**

Set the ECC jumper to the disabled position (jumper on one pin or removed entirely).

### **All Others, Module and Configuration do not Match**

The CPU generates this fault when the module occupying a slot is not of the same type that the configuration file indicates.

#### **Correction**

- 1) Replace the module in the slot with the type indicated in the configuration file.
- 2) Update the configuration file.

### **9.4.7 System Bus Error (Group 12)**

The fault group System Bus Error occurs when the CPU encounters a bus error.

Default action: Diagnostic. Configurable.

#### **4, Unrecognized VME Interrupt Source**

The CPU generates this error when a module generates an interrupt not expected by the CPU (unconfigured or unrecognized).

#### **Correction**

Ensure that all modules configured for interrupts have corresponding interrupt declarations in the program logic.

### 9.4.8 CPU Hardware Failure (Group 13)

The fault group CPU Hardware occurs when the CPU detects a hardware failure, such as a RAM failure or a communications port failure.

When a CPU Hardware failure occurs, the OK LED will flash on and off to indicate that the failure was not serious enough to prevent Controller Communications to retrieve the fault information.

Action: Non-configurable.

#### **6E hex/110 decimal, Time-of-Day Clock not Battery-Backed**

The battery-backed value of the time-of-day clock has been lost.

##### **Correction**

- 1) Replace the battery. Do not remove power from the main rack until replacement is complete. Reset the time-of-day clock using your programming software.
- 2) Replace the module.

#### **0A8 hex/168 decimal, Critical Over-Temperature Failure**

CPU's critical operating temperature exceeded.

##### **All Others**

##### **Correction**

Replace the module.

##### **Fault Extra Data for CPU Hardware Failure**

For a RAM failure in the CPU (one of the faults reported as a CPU hardware failure), the address of the failure is stored in the first four bytes of the field.

### 9.4.9 Module Hardware Failure (Group 14)

The fault group Module Hardware Failure occurs when the CPU detects a non-fatal hardware failure on any module in the system, for example, a serial port failure on a LAN interface module. The fault action for this group is Diagnostic.

Action: Non-configurable.

#### **1A0 hex/416 decimal, Missing 12 Volt Power Supply**

A power supply that supplies 12 volts is required to operate the LAN Interface module.

#### **Correction**

- 1) Install/replace a 100 watt power supply.
- 2) Connect an external VME power supply that supplies 12 volts.

#### **1C2 - 1C6 hex (450 - 454 decimal), LAN Interface Hardware Failure**

Refer to the LAN Interface manual, GFK-0868 or GFK-0869 (previously GFK-0533), for a description of these errors.

#### **All Others, Module Hardware Failure**

A module hardware failure has been detected.

#### **Correction**

Replace the affected module.

### 9.4.10 Option Module Software Failure (Group 16)

The fault group Option Module Software Failure occurs when:

- A non-recoverable software failure occurs on an intelligent option module.
- The module type is not a supported type.
- The Ethernet Interface logs an event in its Ethernet exception log.

Action: Non-configurable.

#### 1, Unsupported Board Type

The board is not one of the supported types.

##### Correction

- 1) Upload the configuration file and verify that the software recognizes the board type in the file. If there is an error, correct it, download the corrected configuration file, and retry.
- 2) Display the Controller Fault Table on the programmer. Contact Technical Support, giving them all the information contained in the fault entry.

#### 2, 3, COMMREQ Frequency Too High

COMMREQs are being sent to a module faster than it can process them.

##### Correction

Change the application program to send COMMREQs to the module at a slower rate or check the completion status of each COMMREQ before sending the next.

#### 4, More Than One BTM in a Rack

There is more than one BTM present in the rack.

##### Correction

Remove one of the BTMs from the rack; there can only be one in a CPU rack.

#### >4, Option Module Software Failure

Software failure detected on an option module.

##### Correction

- 1) Reload software into the indicated module.
- 2) Replace the module.

#### >400, LAN System Software Fault

The Ethernet interface software has detected an unusual condition and recorded an event in its exception log. The Fault Extra Data contains the corresponding event in the Ethernet exception log, which can be viewed by the Ethernet Interface's Station Manager function. The first two digits of Fault Extra Data contain the Event type; the remaining data correspond to the four-digit values for Entry 2 through Entry 6. Some exceptions may also contain optional multi-byte SCode and other data.

##### Correction

For information on interpreting the fault extra data, refer to the *PACSystems TCP/IP Ethernet Communications Station Manager User Manual*, GFK-2225, Appendix B.

### **9.4.11 Program or Block Checksum Failure (Group 17)**

The fault group Program or Block Checksum Failure occurs when the CPU detects error conditions in program or blocks. It also occurs during RUN Mode background checking. In all cases, the Fault Extra Data field of the Controller Fault Table record contains the name of the program or block in which the error occurred.

Action: Non-configurable.

#### **All Error Codes, Program or Block Checksum Failure**

The CPU generates this error when a program or block is corrupted.

#### **Correction**

- 1) Clear CPU memory and retry the store.
- 2) Examine C application for errors.
- 3) Display the Controller Fault Table on the programmer. Contact Technical Support, giving them all the information contained in the fault entry.

#### **Fault Extra Data for Program or Block Checksum Failure**

The name of the offending program block is contained in the first eight bytes of the Fault Extra Data field.

### 9.4.12 Battery Status (Group 18)

Faults in this group occur when the CPU detects a failed battery (or Energy Pack).

Action: Non-configurable.

#### 0, Failed Battery

##### **CPUs with battery-backed RAM, including RX7i CPUs, and RX3i CPU310, CPU315, CPU/CRU320 and NIU001**

The battery in the CPU module has failed or is disconnected.

If the battery is disconnected, this fault is logged for all CPU types and all supported battery types.

Should a Smart Battery fail during operation, this fault is logged for all CPU types. When used with a legacy (non-smart) battery, this indication is not reliable.

##### **CPE302, CPE305 and CPE310**

The Energy Pack has failed or is disconnected.

#### **Correction**

Replace the battery or Energy Pack. For instructions on replacing the battery, refer to the *PACSystems Battery and Energy Pack Manual*, GFK-2741.

#### **1, Low Battery – CPUs with Battery-Backed RAM**

This fault is supported only by the CPU versions listed in the *PACSystems Battery and Energy Pack Manual*, GFK-2741.

The CPU detects the low battery condition only while the CPU is powered up.

If a low battery condition occurs while the CPU is powered down, the CPU logs a Low Battery fault upon power-up as soon as it detects the signal from the smart battery.

While the CPU is powered up, it is unlikely that a Low Battery fault will be detected because the current drain on the battery is negligible. The exception is when a good battery is replaced with a low battery while the CPU has power. In this case, a Low Battery fault would indicate that a good battery has been accidentally replaced with a depleted battery.

The Controller fault table indicates the battery status. For details of LED operation of specific CPUs, refer to *PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual*, GFK-2222.

When a Failed Battery fault is logged, this fault is also logged.

#### **Correction**

Replace the battery. For instructions on replacing the battery, refer to the *PACSystems Battery and Energy Pack Manual*, GFK-2741.

#### **1, Low Battery – CPE302/CPE305/CPE310/CPE330 CPUs with Energy Pack**

The Status LED and the Controller fault table indicate the Energy Pack status.

<b>PLC_BAT (%S0014)</b>	<b>LOW_BAT (%SA0011)</b>	<b>Energy Pack Status</b>
0	0	Energy Pack connected and operational (may be charging)
1	1	Energy Pack not connected or has failed
0	1	Energy Pack is nearing its end-of-life and should be replaced.

### 9.4.13 Constant Sweep Time Exceeded (Group 19)

The fault group Constant Sweep Exceeded occurs when the CPU operates in Constant Sweep mode and detects that the sweep has exceeded the constant sweep timer. In the fault extra data, the DWORD at byte offset 8 contains the amount of time that the sweep went beyond the constant sweep time (in microsecond units). Stored in Big Endian format.

Action: Non-configurable.

#### 0, Constant Sweep

##### Correction

If Constant Sweep (0):

- 1) Increase constant sweep time.
- 2) Remove logic from application program.

**Note:** Error code 1 is not used.

### 9.4.14 System Fault Table Full (Group 20)

The fault group System Fault Table Full occurs when the *Controller Fault Table* reaches its limit.

Action: Non-configurable.

#### 0, System Fault Table Full

##### Correction

Clear the Controller Fault Table.

### 9.4.15 I/O Fault Table Full (Group 21)

The fault group I/O Fault Table Full occurs when the *I/O Fault Table* reaches its maximum configured limit. To avoid loss of additional faults, clear the earliest entry from the table.

Action: Non-configurable.

#### 0, I/O Fault Table Full

##### Correction

Clear the I/O Fault Table.

### 9.4.16 User Application Fault<sup>18</sup> (Group 22)

The fault group Application Fault occurs when the CPU detects a fault in the user program.

Action: Non-configurable.

#### 2, Software Watchdog Timer Expired

The CPU generates this error when the watchdog timer expires. The CPU stops executing the user program and enters STOP/Halt Mode. To recover, cycle power to the CPU with battery disconnected. Causes of timer expiration include: Looping, via jump, very long program, etc.

##### Correction

- 1) Determine what caused the expiration (logic execution, external event, etc.) and correct.
- 2) Use the system service function block to restart the watchdog timer.

#### 7, Application Stack Overflow

Block call depth has exceeded the CPU capability.

##### Correction

Increase the program's stack size or adjust application program to reduce nesting.

#### 11 hex/17 decimal, Program Run Time Error

A run-time error occurred during execution of a program.

##### Correction

Correct the specific problem in the application.

#### 22 hex/34 decimal, Unsupported Protocol

Hardware does not support configured protocol.

#### 33 hex/51 decimal, Flash Read Failed

Possible causes:

- 1) Files not in flash. (May be caused by power cycle during flash write.)
- 2) Could not read from flash because OEM protection is enabled.

#### 34 hex/52 decimal, Memory Reference Out of Range

A user logic memory reference, computed during logic execution, is out of range. Includes indirect references, array element references, and potentially other types of references.

##### Correction

Correct logic or adjust memory size in hardware configuration.

#### 35 hex/53 decimal, Divide by zero attempted in user logic.

User logic contained a divide by zero operation. (Applies to ST and FBD logic.)

##### Correction

Correct logic.

---

<sup>18</sup> Error Codes 1, 4, 5, 6, 8-15, 28, 29 and 49 are not used by PACs.

**36 hex/54 decimal, Operand is not byte aligned.**

A variable in user logic is not properly byte-aligned for the requested operation.

**Correction**

Correct logic or adjust memory size in hardware configuration.

**39 hex/57 decimal, DLB heartbeat not received, All DLBs stopped and deleted**

The controller has not received a heartbeat signal from the programmer within the time specified by the DLB Heartbeat setting in the Target properties.

**Correction**

Increase the DLB Heartbeat setting. For additional information, refer to *Executing DLBs*.

**3B hex /59 decimal, PSB called by a block whose %L or %P memory is not large enough to accommodate this reference.**

Parameterized blocks do not have their own %L data, but instead inherit the %L data of their calling blocks. If %L references are used within a parameterized block and the block is called by \_MAIN, %L references are inherited from the %P references wherever encountered in the parameterized block (for example, %L0005 = %P0005). For a discussion of the use of local data with parameterized blocks, refer to *Parameterized Blocks and Local Data* in Chapter 2.

**Correction**

Determine which block called the parameterized subroutine block and increase the size of %L or %P memory allocated to the calling block. (To do this, change the Extra Local Words setting in the block's Properties.)

The maximum size of %L or %P is 8192 words per block. If your application needs more space, consider changing some %P or %L references to %R, %W, %AI, or %AQ. These changes require a recompilation of the program block and a STOP Mode Store to the CPU.

It is possible, by using Online Editing in the programming software to cause a parameterized block to use %L higher than allowed because of the way it inherits data. To correct this condition, delete the %L variables from the logic and then remove the unused variables from the variable list. These changes require a recompilation of the program block and a STOP Mode Store to the CPU.

### 9.4.17 CPU Over-Temperature (Group 24)

Default action: Diagnostic. Configurable.

#### 1, Over-Temperature failure.

CPU's normal operating temperature exceeded.

##### Correction

Turn off CPU to allow heat to disperse and install a fan kit to regulate temperature.

### 9.4.18 Power Supply Fault (Group 25)

Action: Non-configurable.

#### 1, Power supply failure.

Unknown power supply failure.

##### Correction

Replace power supply module.

#### 2, Power supply overloaded

The load on the power supply has reached its rated maximum

##### Correction

Replace power supply with a higher capacity model or reconfigure system to reduce load on power supply.

#### 3, Power supply switched off

The switch on the power supply was moved to the OFF position.

#### 4, Power-supply has exceeded normal operating temperature

The temperature of the power supply is a just a few degrees from causing it to turn off.

##### Correction

Turn off system to allow heat to disperse. Install a fan kit to regulate temperature.

### 9.4.19 No User Program on Power-Up (Group 129)

The fault group No User Program on Power-Up occurs when the CPU powers up with its memory preserved but no user program exists in the CPU. The CPU detects the absence of a user program on power-up; the controller stays in STOP Mode.

Action: Non-configurable.

##### Correction

Download an application program before attempting to go to RUN Mode.

### 9.4.20 Corrupted User Program on Power-Up (Group 130)

The fault group Corrupted User Program on Power-Up occurs when the CPU detects corrupted user RAM. The CPU will remain in STOP Mode.

Action: Non-configurable.

#### 1, Corrupted user RAM on power-up

The CPU generates this error when it detects corrupted user RAM on power-up.

#### Recommended Corrections, Listed in Order

- 1) Cycle power without battery or Energy Pack.
- 2) Examine any C applications for errors.
- 3) Replace the volatile memory backup battery on the CPU.
- 4) Replace the CPU.

#### 7, User memory not preserved over power cycle

The CPU generates this error when it detects a battery failure that occurred while the controller was powered down.

If this fault occurs on a power cycle when the battery was not detached or replaced, the battery has failed and should be replaced.

#### Correction

Replace the battery on the CPU. For instructions on replacing the battery, refer to the *PACSystems Battery and Energy Pack Manual*, GFK-2741.

### 9.4.21 Window Completion Failure (Group 131)

The fault group Window Completion Failure is generated by the pre-logic and end-of-sweep processing software in the CPU. The fault extra data contains the name of the task that was executing when the error occurred.

Action: Non-configurable.

#### 0, Window Completion Failure

The CPU generates this error when it is operating in Constant Sweep mode and the constant sweep time was exceeded before the programmer window had a chance to begin executing.

#### Correction

Increase the constant sweep timer value.

#### 1, Logic Window Skipped

The logic window was skipped due to lack of time to execute.

#### Correction

- 1) Increase base cycle time.
- 2) Reduce Communications Window time.

### 9.4.22 Password Access Failure (Group 132)

The fault group Password Access Failure occurs when the CPU receives a request to change to a new privilege level and the password included with the request is not valid for that level.

Action: Non-configurable.

#### **0, Password Access Failure**

##### **Correction**

Retry the request with the correct password.

### 9.4.23 Null System Configuration for RUN Mode (Group 134)

The fault group Null System Configuration for RUN Mode occurs when the CPU transitions from STOP Mode to one of the RUN Modes and a configuration file is not present. The transition to Run is permitted, but no I/O scans occur.

Action: Informational. Non-configurable.

#### **0, Null System Configuration for RUN Mode**

##### **Correction**

Download a configuration file.

### 9.4.24 CPU System Software Failure (Group 135)

Faults in this group are generated by the operating software of the CPU. They occur at many different points of system operation. When a fatal fault occurs, the CPU immediately transitions to STOP/Halt. The only activity permitted when the CPU is in this mode is communications with the programmer. The only method of clearing this condition is to cycle power on the controller with the battery disconnected.

Action: Non-configurable.

#### 5A hex/90 decimal, User Shut Down Requested

The CPU generates this informational alarm when SVC\_REQ #13 (User Shut Down) executes in the application program.

##### Correction

None required. Information-only alarm.

#### 94 hex/148 decimal, Units Contain Mismatched Firmware, Update Recommended

This fault is logged each time the redundancy state changes and the redundant CPUs contain incompatible firmware.

##### Correction

Ensure that redundant CPUs have compatible firmware.

#### D8 hex/216 decimal, Processor Exception Trap

The processor has detected an error condition while executing an instruction. The CPU was placed into STOP/Halt mode.

##### Correction

Disconnect the battery from the CPU and cycle power to clear the STOP/Halt condition.

#### DA hex/218 decimal, Critical Over-Temperature Failure

Critical operating temperature of CPU exceeded.

##### Correction

Turn off CPU to allow heat to disperse and install a fan kit to regulate temperature.

### All Others, CPU Internal System Error

An internal system error has occurred that should **not** occur in a production system.

#### Correction

Display the Controller Fault Table on the programmer. Contact Technical Support and give them all the information contained in the fault entry.

<b>Error</b>	<b>Fault Extra Data Value (First Byte)</b>	<b>Description</b>
DEVICE_NOT_AVAILABLE	CF	Specific device is not available in the system.
BAD_DEVICE_DATA	CC	Data stored on device has been corrupted and is no longer reliable. Or, Flash Memory has not been initialized.
DEVICE_RW_ERROR	CB	Error occurred during a read/write of the Flash Memory device.
FLASH_INCOMPAT_ERROR	8E	Data in Flash Memory is incompatible with the CPU firmware release due to the CPU firmware revision numbers, the instruction groups supported, or the CPU model number.
ITEM_NOT_FOUND_ERROR	8D	One or more specified items were not found in Flash Memory.

### 9.4.25 Communications Failure During Store (Group 137)

This fault group occurs during the store of programs or blocks and other data to the CPU. The stream of commands and data for storing programs or blocks and data starts with a special start-of-sequence command and terminates with an end-of-sequence command. This fault is logged if communications with the programming device performing the store is interrupted or any other failure that terminates the store occurs. As long as this fault is present in the system, the controller will not transition to RUN Mode. This fault is *not* automatically cleared on power-up; you must specifically clear the condition.

Action: Non-configurable.

#### 0, Communications Failure During Store

##### Correction

Clear the fault and retry the download of the program or configuration file.

#### 1, Communications Lost During RUN Mode Store

Communications or power was lost during a RUN Mode Store. The new program or block was not activated and was deleted.

##### Correction

Perform the RUN Mode Store again. This fault is diagnostic.

#### 2, Communications Lost During Cleanup for RUN Mode Store

Communications was lost, or power was lost during the cleanup of old programs or blocks during a RUN Mode Store. The new program or block is installed, and the remaining programs and blocks were cleaned up.

##### Correction

None required. This fault is informational.

#### 3, Power Lost During a RUN Mode Store

Power was lost in the middle of a RUN Mode Store.

##### Correction

Delete and restore the program. This error is fatal.

### 9.4.26 Non-Critical CPU Software Event (Group 140)

This group is used for recording conditions in the system that may provide valuable information to Technical Support.

Default action: Non-configurable.

<b>Error Code</b>	<b>Description</b>	<b>Correction</b>
1-30	Events during power-up	No corrective action is required unless this fault occurs with other specific faults. The fault may contain useful information for Technical Support if other problems are encountered.
31-50	Events on the serial port or in a serial protocol	
51, 52	Miscellaneous internal system events	
53	Access control fault	See details below.
54 and greater	Miscellaneous internal system events	No corrective action is required unless this fault occurs with other specific faults. The fault may contain useful information for Technical Support if other problems are encountered.

#### Error code 53, Access Control Fault

If data access is prevented because of the Enhanced Security settings, the Controller logs a fault into the fault table. This fault can be used to help diagnose access problems. To prevent overflowing the fault table, only one fault is logged until the fault table is cleared.

#### Fault example

Location: 0.8 Date/Time: 07-07-2013 17:06:55.087

Group: 140 INFO\_CPU\_SOFTWR - CPU software event

Error Code: 53 Action:1 Task Num:3

Extra Data: 00 fa 02 a5 00 00 00 00 01 1e 06 00 00 00 00 00 00 00 01 00 00 00 00 00

#### Meaning of this example fault

A 1-bit READ request beginning at %S7 was rejected due to an access violation.

**Interpreting the Fault Extra Data**

Bytes 1 - 8: Ignored when decoding a security-related fault.

Byte 9: The operation during which the fault occurred.

- 01 (as in the example): Read
- 02: Write

Byte 10: The hexadecimal value (HV) that specifies a CPU memory area.

<b>Hexadecimal Value (HV)</b>	<b>Memory area</b>
08	%R (Register memory)
0A	%AI (Analog input memory)
0C	%AQ (Analog output memory)
10	%I (Discrete input memory)
12	%Q (Discrete output memory)
14	%T (Discrete temporary status memory)
16	%M (Discrete momentary internal memory)
18	%SA (Discrete system memory A)
1A	%SB (Discrete system memory B)
1C	%SC (Discrete system memory C)
1E	%S (Discrete system memory)
38	%G (Genius global memory)
C4	%W (Bulk Memory)

Bytes 11–18: 0-based bit offset of the memory area being accessed. The 8-byte value is encoded in little endian format, meaning that the byte values are reversed. In the example, the value is 0x0000000000000006, which is equal to 1-based bit offset 7.

Bytes 19–22: The length in bits of data requested. In the example, 1 bit was requested.

Bytes 23–24: Ignored when decoding a security-related fault.

## 9.5 I/O Fault Descriptions and Corrective Actions

The I/O fault table reports the following data about faults:

- Fault Group
- Fault Action
- Fault category
- Fault type
- Fault description

All faults have a fault category, but a fault type and fault group may not be listed for every fault. To view the detailed information pertaining to a fault, click the fault entry in the I/O Fault Table.

**Note:** The model number mismatch and I/O type mismatch faults are reported in the controller fault table under the System Configuration Mismatch group. They are not reported in the I/O fault table.

### 9.5.1 Fault Extra Data

An I/O fault table entry contains up to 21 bytes of I/O fault extra data that contains additional information related to the fault. Not all entries contain I/O fault extra data.

### 9.5.2 I/O Fault Groups

Group Number	Group Name	Default Fault Action <sup>16</sup>	Configurable
2	Loss of or Missing IOC	Diagnostic	Yes
3	Loss of or Missing I/O module or network Device	Diagnostic	Yes
6	Addition or Reset of, or Extra IOC	N/A	No
7	Addition of or Extra I/O module or network Device	N/A	No
9	IOC or I/O Bus Fault	Diagnostic	Yes
10	I/O Module Fault	N/A	No
15	IOC Software Failure	Same As Group 2 <sup>19</sup>	Yes
16	Module Software Failure	N/A	No
28	PROFINET Alarms	Diagnostic	No
133	Genius Block Address Mismatch	N/A	No

<sup>19</sup> The fault action for the IOC Software Failure group 15 always matches the action used by the Loss of or Missing IOC group 2. If the Loss of or Missing IOC group is configured, the IOC Software Failure group is also configured to take the same fault action.

### 9.5.3 I/O Fault Categories

<b>Category</b>	<b>Fault Type</b>	<b>Fault Description</b>	<b>Fault Extra Data</b>
Circuit Fault (1)	Discrete Fault (1)	Loss of User Side Power (01 hex)	Circuit Configuration
		Short Circuit in User Wiring (02 hex)	Circuit Configuration
		Sustained Overcurrent (04 hex)	Circuit Configuration
		Low or No Current Flow (08 hex)	Circuit Configuration
		Switch Temperature Too High (10 hex)	Circuit Configuration
		Switch Failure (20 hex)	Circuit Configuration
		Point Fault (83 hex)	Circuit Configuration
		Output Fuse Blown (84 hex)	Circuit Configuration
	Analog Fault (2)	Input Channel Low Alarm (01 hex)	Circuit Configuration
		Input Channel High Alarm (02 hex)	Circuit Configuration
		Input Channel Under Range (04 hex)	Circuit Configuration
		Input Channel Over Range (08 hex)	Circuit Configuration
		Input Channel Open Wire (10 hex)	Circuit Configuration
		Over Range or Open Wire (18 hex)	Circuit Configuration
		Output Channel Under Range (20 hex)	Circuit Configuration
		Output Channel Over Range (40 hex)	Circuit Configuration
		Expansion Channel Not Responding (80 hex)	Circuit Configuration
		Invalid Data (81 hex)	Circuit Configuration

<b>Category</b>	<b>Fault Type</b>	<b>Fault Description</b>	<b>Fault Extra Data</b>
	Low-Level Analog Fault (4)	Input Channel Low Alarm (01 hex)	Circuit Configuration
		Input Channel High Alarm (02 hex)	Circuit Configuration
		Input Channel Under Range (04 hex)	Circuit Configuration
		Input Channel Over Range (08 hex)	Circuit Configuration
		Input Channel Open Wire (10 hex)	Circuit Configuration
		Wiring Error (20 hex)	Circuit Configuration
		Internal Fault (40 hex)	Circuit Configuration
		Input Channel Shorted (80 hex)	Circuit Configuration
		Invalid Data (81 hex)	Circuit Configuration
		GENA (Genius Network Adapter) Fault (3)	GENA Circuit Fault (80 hex)
	Remote I/O Scanner Fault	Remote I/O Scanner Circuit Fault	Byte 1: Circuit Type Byte 2: I/O Type
Loss of Block (2)	Not Specified (0) A/D Communications Lost (1)	NA	Block Configuration Number of Input Circuits Number of Output Circuits
Addition of Block (3)	NA	NA	Block Configuration Number of Input Circuits Number of Output Circuits
I/O Bus Fault (6)	Bus Fault (1) Bus Outputs Disabled (2) SBA Conflict (3)	NA	NA
Genius Module Fault (8)	Headend Fault (0) A to D Comm. Fault (1) User Scaling Error (5) Store Fail (6)	Configuration Memory Failure (08 hex) Calibration Memory Failure (20 hex) Shared RAM Failure (40 hex) Internal Circuit Fault (80 hex) Watchdog Timeout (81 hex) Output Fuse Blown (84 hex)	NA
Addition of IOC (9)	NA	Extra Module (01 hex) Reset Request (02 hex)	NA

<b>Category</b>	<b>Fault Type</b>	<b>Fault Description</b>	<b>Fault Extra Data</b>
Loss of IOC (10)	NA	NA	Timeout Unexpected State Unexpected Mail Status VME Bus Error
IOC Software Fault (11)	NA	NA	NA
Forced Circuit (12)	NA	NA	Block Configuration Discrete/Analog Indication*
Unforced Circuit (13)	NA	NA	Block Configuration Discrete/Analog Indication*
Loss of I/O Module (14)	NA	NA	NA
Addition of I/O Module (15)	NA	VME Module Reset Requested (30 hex)	NA
Extra I/O Module (16)	NA	NA	NA
Extra Block (17)	NA	NA	NA
IOC Hardware Failure (18)	NA	NA	NA
GBC stopped reporting faults because too many faults have occurred (19)	GBC detected high error count on Genius Bus and dropped off the bus for at least 1.5 seconds. (1)	NA	NA
GBC Software Exception (21)	Datagram queue full (1) R/W request queue full (2) Low priority mail rejected (3) Background message received before CPU completed initialization (4) Genius software version too old (5) Excessive use of internal GBC memory (6)	NA	
Block Switch (22) – redundant Genius block switched bus	NA	NA	Block Configuration Number of Input Circuits Number of Output Circuits Rack/Slot address of GBC from which block was removed.
Block not active on redundant bus (23)	NA	NA	NA
Reset of IOC (27)	NA	NA	NA

<b>Category</b>	<b>Fault Type</b>	<b>Fault Description</b>	<b>Fault Extra Data</b>
PROFINET network faults (33 and higher)	NA	Refer to PROFINET controller documentation.	NA

### 9.5.4 Circuit Faults (Category 1)

Circuit faults apply to Genius I/O modules and the IC697VRD008 RTD/Strain Bridge modules. Fault extra data is available for all faults in this category. More than one condition may be present in a particular reporting of the fault.

Action: Diagnostic.

#### Fault Extra Data for Circuit Faults

##### Genius Bus Controller

Circuit fault entries use one or two bytes of the fault extra data area. If the GBC reports the fault, the first byte is generated by the GBC and the second byte contains the circuit configuration and is encoded as shown in the following table.

<b>Value (Byte 2)</b>	<b>Description</b>
1	Circuit is an input.
2	Circuit is an input.
3	Circuit is an output.

If the fault type is a GENA fault, the second byte contains the data that was reported from the GENA module in Fault Byte 2 of its *Report Fault* message.

##### VRD001 RTD/Strain Bridge

Circuit fault entries; 13 bytes of the fault extra data area. The fault extra data is encoded as shown in the following table.

<b>Bytes</b>	<b>Description</b>
1-10	Used by technical support.
11	Line number
12	Module number
13	Used by technical support.

## Fault Descriptions for Discrete Faults

### 1, Loss of User Side Power

The GBC generates this error when there is a power loss on the field wiring side of a Genius I/O block.

#### Correction

- 1) (Only valid for Isolated I/O blocks.) Initiate *Pulse Test* COMREQ #1. Pulse test may be enabled or disabled at I/O block.
- 2) Correct the power failure.

### 2, Short Circuit in User Wiring

The GBC generates this error when it detects a short circuit in the user wiring of a Genius block. A short circuit is defined as a current level greater than 20 amps.

#### Correction

Fix the cause of the short circuit.

### 4, Sustained Overcurrent

The GBC generates this error when it detects a sustained current level greater than 2 amps in the user wiring.

#### Correction

Fix the cause of the over current.

### 8, Low or No Current Flow

The GBC generates this error when there is very low or no current flow in the user circuit.

#### Correction

Fix the cause of the condition.

### 10 hex, Switch Temperature Too High

The GBC generates this error when the Genius block reports a high temperature in the Genius Smart Switch.

#### Correction

- 1) Ensure that the block is installed to provide adequate circulation.
- 2) Decrease the ambient temperature surrounding the block.
- 3) Install RC Snubbers on inductive loads.

### 20 hex, Switch Failure

The GBC generates this error when the Genius block reports a failure in the Genius Smart Switch.

#### Correction

- 1) Check for shunts across Genius output (pushbuttons).
- 2) Replace the Genius I/O block.

### 83 hex, Point Fault

The CPU generates this error when it detects a failure of a single I/O point on a Genius I/O module.

#### Correction

Replace the Genius I/O block.

### **84 hex, Output Fuse Blown**

The CPU generates this error when it detects a blown fuse on a Genius I/O output block.

#### **Correction**

- 1) Determine and repair the cause of the fuse blowing; replace the fuse.
- 2) Replace the block.

## **Fault Descriptions for Analog Faults**

### **1, Input Channel Low Alarm**

The GBC generates this error when the Genius Analog module reports a low alarm on an input channel.

#### **Correction**

Correct the condition causing the low alarm.

### **2, Input Channel High Alarm**

The GBC generates this error when the Genius Analog module reports a high alarm on an input channel.

#### **Correction**

Correct the condition causing the high alarm.

### **4, Input Channel Under Range**

The GBC generates this error when the Genius Analog module reports an under-range condition on an input channel.

#### **Correction**

Correct the problem causing the condition.

### **8, Input Channel Over Range**

The GBC generates this error when the Genius Analog module reports an over-range condition on an input channel.

#### **Correction**

Correct the problem causing the condition.

### **10 hex/16 decimal, Input Channel Open Wire**

The GBC generates this error when a Genius Analog module detects an open wire condition on an input channel.

#### **Correction**

Correct the problem causing the condition.

### **18 hex/24 decimal, Over Range or Open Wire**

Inputs open or inputs off-scale.

#### **Correction**

Correct the problem causing the condition.

### **20 hex/32 decimal, Output Channel Under Range**

The GBC generates this error when the Genius Analog module reports an under-range condition on an output channel.

#### **Correction**

Correct the problem causing the condition.

### **40 hex/64 decimal, Output Channel Over Range**

The GBC generates this error when the Genius Analog module reports an over-range condition on an output channel.

#### **Correction**

Correct the problem causing the condition.

### **80 hex/128 decimal, Expansion Channel Not Responding**

The CPU generates this error when data from an expansion channel on a multiplexed analog input board is not responding.

#### **Correction**

- 1) Check wiring to the module.
- 2) Replace the module.

### **81 hex/129 decimal, Invalid Data**

The GBC generates this error when it detects invalid data from a Genius Analog input block.

#### **Correction**

Correct the problem causing the condition.

## **Low-Level Analog Faults**

### **1, Input Channel Low Alarm**

The GBC generates this error when the Genius Analog module reports a low alarm on an input channel.

#### **Correction**

Correct the condition causing the low alarm.

### **2, Input Channel High Alarm**

The GBC generates this error when the Genius Analog module reports a high alarm on an input channel.

#### **Correction**

Correct the condition causing the high alarm.

### **4, Input Channel Under Range**

The GBC generates this error when the Genius Analog module reports an under-range condition on an input channel.

#### **Correction**

Correct the problem causing the condition.

### 8, Input Channel Over Range

The GBC generates this error when the Genius Analog module reports an over-range condition on an input channel.

#### **Correction**

Correct the problem causing the condition.

### 10 hex, Input Channel Open Wire

The GBC generates this error when the Genius Analog module detects an open wire condition on an input channel.

#### **Correction**

Correct the problem causing the condition.

### 20 hex/32 decimal, Wiring Error

The GBC generates this error when the Genius Analog module detects an improper RTD connection or thermocouple reverse junction fault.

#### **Correction**

Correct the problem causing the condition.

### 40 hex/64 decimal, Internal Fault

The GBC generates this error when the Genius Analog module reports a cold junction sensor fault on a thermocouple block or an internal error in an RTD block.

#### **Correction**

Correct the problem causing the condition.

### 80 hex/128 decimal, Input Channel Shorted

The GBC generates this error when it detects an input channel shorted on a Genius RTD or Strain Gauge Block.

#### **Correction**

Correct the problem causing the condition.

### 81 hex/129 decimal, Invalid Data

The GBC generates this error when it detects invalid data from a Genius Analog input block.

#### **Correction**

Correct the problem causing the condition.

### GENA Fault

The GENA Fault has no fault descriptions associated with it. GENA Fault Byte 2 is the first byte of the fault extra data.

### 80 hex/128 decimal

The Genius I/O operating software generates this error when it detects a failure in a GENA block attached to the Genius I/O bus.

#### **Correction**

Replace the GENA block.

### 9.5.5 Loss of Block (Category 2)

The fault category Loss of Block applies to Genius devices.

Action: Diagnostic.

#### Loss of Block

The GBC generates this error when it is unable to communicate to the Genius device.

#### Correction

- 1) Verify power and wiring to the block.
- 2) Replace the block.

#### Loss of Block - A/D Communications Fault

The GBC generates this error when it detects a failure of A/D communications on a Genius device.

#### Correction

- 1) Verify power and serial bus wiring to the block.
- 2) Replace the block.

#### Fault Extra Data for Loss of Block

The Loss of Block fault provides four bytes of fault extra data. The second byte contains the block configuration and is encoded as shown in the following table. The third byte specifies the number of input circuits possibly used, and the fourth byte specifies the number of output circuits possibly used.

#### Block Configuration (Byte 2)

<b>Value</b>	<b>Description</b>
1	Block is configured for inputs only.
2	Block is configured for outputs only.
3	Block is configured for inputs and outputs (grouped block).

### 9.5.6 Addition of Block (Category 3)

The fault category Addition of Block applies only to Genius devices. There are no fault types or fault descriptions associated with this category.

The Genius operating software generates this error when it detects that a Genius block that stopped communicating with the controller starts communicating again.

Action: Diagnostic.

#### Correction

Informational only. None required.

#### Fault Extra Data for Addition of Block

The Addition of Block fault provides four bytes of fault extra data. The second byte contains the block configuration and is encoded as shown in the following table. The third byte specifies the number of input circuits possibly used, and the fourth byte specifies the number of output circuits possibly used.

#### Block Configuration (Byte 2)

<b>Value</b>	<b>Description</b>
1	Block is configured for inputs only.
2	Block is configured for outputs only.
3	Block is configured for inputs and outputs (grouped block).

### 9.5.7 I/O Bus Fault (Category 6)

The fault category I/O Bus Faults has three fault types associated with it.

Default action: Diagnostic. Configurable.

#### Bus Fault

The GBC operating software generates this error when it detects a failure with a Genius I/O bus. (Generated when Error Rate in the GBC configuration is exceeded—the default Error Rate is 10 errors in a 10 second period).

#### Correction

- 1) Determine the reason for the bus failure and correct it.
- 2) Replace the GBC.

The Error Rate can be set higher than the default value if needed, but the bus should be examined electrically—use an oscilloscope for waveform check.

#### Bus Outputs Disabled

The GBC operating software generates this error when it times out waiting for the CPU to perform an output scan.

#### Correction

- 1) Reduce time between GBC output scans by assigning them to scan set 1.
- 2) Increase CPU software watchdog timer setting
- 3) Replace the CPU.
- 4) Display the controller fault table on the programmer. Contact Technical Support, giving them all the information contained in the fault entry.

#### SBA Conflict

The GBC detected a conflict between its serial bus address and that of another device on the bus.

#### Correction

Adjust one of the conflicting serial bus addresses.

### 9.5.8 Module Fault (Category 8)

The fault category Module Fault has one fault type, headend fault, and eight fault descriptions. This fault category does not provide fault extra data. The default fault action for this category is Diagnostic.

#### 08 hex, Configuration Memory Failure

The GBC generates this error when it detects a failure in a Genius block's EEPROM or NVRAM.

##### Correction

Replace the Genius block's electronics module.

#### 20 hex/32 decimal, Calibration Memory Failure

The GBC generates this error when it detects a failure in a Genius block's calibration memory.

##### Correction

Replace the Genius block's electronics module.

#### 40 hex/64 decimal, Shared RAM Fault

The GBC generates this error when it detects an error in a Genius block's shared RAM.

##### Correction

Replace the Genius block's electronics module.

#### 80 hex/128 decimal, Module Fault

An internal failure has been detected in a module.

##### Correction

Replace the affected module.

#### 81 hex/129 decimal, Watchdog Timeout

The CPU generates this error when it detects that an input module watchdog timer has expired.

##### Correction

Replace the input module.

#### 84 hex/132 decimal, Output Fuse Blown

The CPU generates this error when it detects a blown fuse on an output module.

##### Correction

- 1) Determine and repair the cause of the fuse blowing, and replace the fuse.
- 2) Replace the module.

### 9.5.9 Addition of IOC (Category 9)

The fault category Addition of I/O Controller has no fault types or fault descriptions associated with it. The default fault action for this category is Diagnostic.

#### Addition of IOC

The CPU generates this error when an IOC that has been faulted returns to operation or when an IOC is found in the system and the configuration file indicates that no IOC is to be in that slot or when an IOC is hot inserted.

#### Correction

- 1) No action is necessary if the faulted module is in a remote rack and is returning due to a remote rack power cycle.
- 2) Update the configuration file or remove the module.

#### 01 hex, Extra Module

Module present, but not configured.

#### Correction

Update the configuration file or remove the module.

#### 02 hex, Reset Request

Module added back after reset request. No corrective action is necessary.

### 9.5.10 Loss of or Missing IO Controller (Category 10)

The fault category Loss of IOC has no fault types or fault descriptions associated with it.

Default action: Diagnostic. Configurable.

**Note:** This fault is always displayed as Fatal in the I/O Fault Table, regardless of its configured action.

The CPU generates this error when it cannot communicate with an I/O Controller and an entry for the IOC exists in the configuration file.

This fault is also logged when an IOC is hot removed (No corrective action necessary in this case).

#### Correction

- 1) Verify that the module in the slot/bus address is the correct module.
- 2) Review the configuration file and verify that it is correct.
- 3) Replace the module.
- 4) If fault is not resolved, display the controller fault table on the programmer. Contact Technical Support, giving them all the information contained in the fault entry.

#### Fault Extra Data for Loss of or Missing IOC

Fault extra data for Loss of or Missing IOC provides additional information for diagnostics by Technical Support.

### 9.5.11 IOC (I/O Controller) Software Fault (Category 11)

The fault category IOC Software Fault applies to any type of I/O Controller.

Action: Fatal.

#### Datagram Queue Full, Read/Write Queue Full

Too many datagrams or read/write requests have been sent to the GBC.

#### Correction

Adjust the system to reduce the request rate to the GBC.

#### Response Lost

The GBC is unable to respond to a received datagram or read/write request.

#### Correction

Adjust the system to reduce the request rate to the GBC.

### 9.5.12 Forced and Unforced Circuit (Categories 12 and 13)

The fault categories Forced Circuit and Unforced Circuit report point conditions and therefore are not technically faults. They have no fault types or fault descriptions. These reports occur when a Genius I/O point was forced or unforced with the Hand-Held Monitor.

Action: Informational.

#### Fault Extra Data for Forced/Unforced Circuit

Three bytes of fault extra data are present when a circuit force is added or removed

<b>Byte Number</b>	<b>Description</b>	<b>Value</b>	<b>Description</b>
1	Circuit Configuration	1	Circuit is an input.
		2	Circuit is an input.
		3	Circuit is an output.
2	Analog/Discrete Information	1	Block is a discrete block.
		2	Block is an analog block.
		3	Block has both discrete and analog.

### 9.5.13 Loss of or Missing I/O Module (Category 14)

The fault category Loss of I/O Module applies to discrete and analog I/O modules. There are no fault types or fault descriptions associated with this category.

Default action: Diagnostic. Configurable.

The CPU generates this error when it detects that an I/O module is no longer responding to commands from the CPU, or when the configuration file indicates an I/O module is to occupy a slot and no module exists in the slot. This fault is also logged when an I/O module is hot removed (No corrective action necessary in this case).

#### Correction

- 1) Replace the module.
- 2) Correct the configuration file.
- 3) Display the I/O fault table on the programmer. Contact Technical Support, giving them all the information contained in the fault entry.

### 9.5.14 Addition of I/O Module (Category 15)

The fault category Addition of I/O Module applies to discrete and analog I/O modules. There are no fault types or fault descriptions associated with this category.

Action: Diagnostic.

#### Addition of I/O Module

The CPU generates this error when an I/O module that had been faulted returns to operation or is hot inserted.

#### Correction

- 1) No action necessary if module was removed or replaced or if the remote rack was power cycled.
- 2) Update the configuration file or remove the module.

### 30 hex/48 decimal, VME Reset on Request

Reset of VME module was requested. No corrective action necessary.

### 9.5.15 Extra I/O Module (Category 16)

The fault category Extra I/O Module applies to discrete and analog I/O modules. There are no fault types or fault descriptions associated with this category.

Action: Diagnostic.

The CPU generates this error when it detects an I/O module in a slot that the configuration file indicates should be empty.

#### Correction

- 1) Remove the module. (It may be in the wrong slot.)
- 2) Update and restore the configuration file to include the extra module.

### 9.5.16 Extra Block (Category 17)

The fault category Extra Block applies only to Genius I/O devices. There are no fault types or fault descriptions associated with this category.

Action: Diagnostic.

The GBC generates this error when it detects a Genius device on the bus at a serial bus address where the configuration file does not have a block.

#### Correction

- 1) Remove or reconfigure the block. (It may be at the wrong serial bus address.)
- 2) Update and restore the configuration file to include the extra block.

### 9.5.17 IOC Hardware Failure (Category 18)

The fault category IOC Hardware Failure has no fault types or fault descriptions.

Action: Diagnostic.

The Genius operating software generates this error when it detects a hardware failure in the bus communication hardware or a baud rate mismatch.

#### Correction

- 1) Verify that the baud rate set in the configuration file for the GBC agrees with the baud rate programmed in every block on the bus.
- 2) Change the configuration file and restore it, if necessary.
- 3) Replace the GBC.
- 4) Selectively remove each block from the bus until the offending block is isolated then replace it.

### 9.5.18 GBC Stopped Reporting Faults (Category 19)

GBC detected a high error count on the Genius I/O bus and dropped off the bus for at least 1.5 seconds.

#### Correction

Check for incorrect wiring, interference from other equipment, a loose connection, or a failed device on the Genius bus.

### 9.5.19 GBC Software Exception (Category 21)

#### 1, Incoming datagram queue full

Too many datagrams or read/write requests have been sent to the GBC.

##### **Correction**

Adjust the system to reduce the request rate to the GBC.

#### 2, Read/write request queue full

The queue for Read/Write requests in the GBC is full. The requests may be from the Genius Bus or from COMMREQs.

##### **Correction**

Adjust the system to reduce the request rate to the GBC.

#### 3, Low priority mail queue from GBC to CPU full

The response to the CPU was lost.

#### 4, Genius background message requiring CPU action received before CPU completed

##### **initialization**

Message was ignored.

#### 5, GBC software version too old

##### **Correction**

Update GBC firmware.

#### 6, Excessive use of internal GBC memory

##### **Correction**

Verify COMMREQ usage.

### 9.5.20 Block Switch (Category 22)

The Block Switch fault category has no fault types or fault descriptions.

Action: Diagnostic.

The GBC generates this error when a Genius block on redundant Genius buses switches from one bus to another.

#### Correction

- 1) No action is required to keep the block operating.
- 2) The bus that the block switched from may need to be repaired.
  - a) Verify the bus wiring.
  - b) Replace the I/O controller.
  - c) Replace the Bus Switching Module (BSM).

#### Fault Extra Data for Block Switch

Byte Number	Description	Value	Description
1	Circuit configuration	1	Circuit is an input.
		2	Circuit is an input.
		3	Circuit is an output.
2	Block configuration	1	Block is configured for inputs only.
		2	Block is configured for outputs only.
		3	Block is configured for inputs and outputs (grouped block).
3	Number of input circuits used		
4	Number of output circuits used		

### 9.5.21 Reset of IOC (Category 27)

The fault category Reset of I/O Controller has no fault types or fault descriptions associated with it. The default fault action for this category is Diagnostic.

The CPU generates this message when an I/O Controller is reset. No corrective action necessary.

## 9.6 Diagnostic Logic Blocks (DLBs)

A Diagnostic Logic Block (DLB) is a block of Ladder Diagram logic that can be downloaded to the controller for independent execution. These blocks are useful tools for interacting with an application that is running in the PACSystems controller. DLBs may be used to:

- Collect information from a running application to analyze and diagnose problems
- Test modifications and corrections to a running application before actually incorporating them into the application.
- Test the devices that will be controlled by the application.

DLBs are intended to accomplish a specific task that is temporary in nature, such as diagnosing the source of a problem or testing tuning parameters. When you have finished using a DLB, it should be removed from the host controller. At this point the application logic and its variable allocation return to what it was before the DLB was downloaded.

You can also remove the DLBs from the Logic Developer target, at which point the target's logic and variable allocation will be identical to what they were before the DLBs were introduced.

Note that, although the DLB is removed from the controller, any changes the DLB made to the system are **not** removed. For example, if the DLB logic changes a hardware parameter, the parameter does not return to its previous value when the DLB is removed.

DLB logic can be executed with the controller in STOP IO Enabled Mode, which allows debugging the application without the main application program running.

---

### Caution



**Do not use a DLB as a permanent part of a production application, because a DLB is stopped and deleted from memory when Logic Developer loses its Programmer-mode connection with the host controller. This could happen if the programmer's communications cable is disconnected or if a second programmer connects serially to the same RX3i and establishes a Programmer-mode session.**

---

**Note:** Redundancy CPUs do not support DLBs.

### 9.6.1 DLB Operation

DLBs are created as components of a specific Target and are separate from the application logic block components associated with a target.

They are written in LD programming language and support many of the same features, such as View Lock, Edit Lock, etc. as other block types.

A target can have a maximum of 128 DLBs in a given PME target. Each DLB can have associated published variable table (PVT) and cam profile (used with Motion applications) files. Each DLB can use up to 128K bytes of memory.

A DLB can be copied and pasted like other blocks. Regardless of where a DLB is pasted, normal conflict handling is applied.

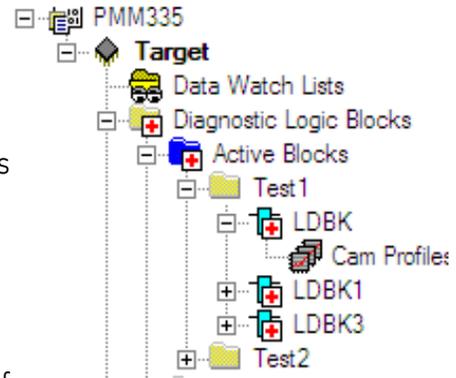


Figure 22: Diagnostic Logic Blocks (DLBs) assigned to Target in MPE

An active DLB can be dragged to the Toolchest, to folders under the **Active Blocks** node, or to folders under the **Program Blocks** node. Note that only active blocks can be dragged. Downloading, executing, or modifying a DLB does not affect the equality of the main logic program.

#### Suspend I/O Function and DLBs

The Suspend I/O (SUS\_IO) function operates the same in a DLB as it does in application logic. Both application logic and DLB logic execute in the CPU Sweep Logic window. Therefore, when a SUSPEND\_IO is executed by either the application or the DLB, outputs are held current during the output scan that occurs immediately after the Logic window finishes its execution, and input references will not be updated from inputs during the input scan that occurs immediately before the Logic window is executed in the next CPU sweep.

Note that a SUSPEND\_IO only affects normal I/O scans. It does not affect I/O scanning that is done as the result of DO\_IO or SCAN\_SET\_IO functions that execute in application or DLB logic. SUS\_IO has the same effect whether it is executed once in a sweep or multiple times in a sweep.

### Restrictions on DLB Operation

Because DLBs are intended only for temporary use, there are more restrictions on their operation compared to application logic blocks. All built-in functions and function blocks other than those listed below can be used in DLB logic.

- DLB logic may not call any logic block or be called by any logic block.
- You cannot define parameters or scheduling for a DLB.
- A DLB has no parameters other than the standard ENO output parameter. Since DLBs cannot be called from other blocks, you can access its ENO parameter only by reading or writing it in the DLB's logic.
- You cannot use variables that have %L or %P addresses. As a consequence, the following features that require %L or %P memory cannot be used in a DLB:
  - a. #FST\_EXE system variable
  - b. The built-in timer function blocks, ONDTR, OFDT, and TMR
  - c. %L or %P variables.
- Locally scoped variables must be symbolic. For additional information, refer to DLB Variables.
- DLBs or their associated files cannot be loaded from the RX3i.
- DLBs and their associated files cannot be downloaded to flash memory.
- You cannot give an LD DLB the name \_MAIN.
- You cannot modify an active LD DLB while it is executing on the Controller.
- You cannot perform a Test Edit (Online Edit Mode and Online Test Mode).
- You cannot perform word-for-word changes on an active DLB.

### DLB Variables

A DLB can have its own variables, which are local to the DLB and not accessible by any other block. All DLB local variables are symbolic, retentive, and published.

Local variables should be used within DLBs whenever possible. If the system is already running and you create new global variables in the DLB, the programming software will not download the DLB because the programmer's memory map will no longer match the RX3i controller's memory map.

DLB logic can read and write the global variables of the application that resides in the same target as it does. These variables may be mapped or symbolic.

To use functions that require the use of located variables, a DLB must use the global located variables of the application that resides in the same target as the DLB. These functions include:

- a. COMMREQ (location of the Status variable)
- b. DO\_IO
- c. Some SVC\_REQ functions

A DLB can create aliases to global located application variables or arrays of variables that were specifically created and documented to serve as *scratchpad* memory for DLBs that need to use located variables.

## 9.6.2 Executing DLBs

### DLB Properties

The properties for an active DLB include *Execution Mode*, which has the following possible values:

- **Sweep** (Default) - The DLB executes at a fixed point in the normal Controller sweep, until explicitly stopped.
- **Update Rate** - Uses the *Update Rate* defined for the Target. The actual rate varies from a minimum value equal to the *Update Rate* to a maximum value of *Update Rate + 1 sweep*. If the sweep takes more time than the update rate, the DLB is executed as soon as the user logic program execution completes in the current sweep.
- **Scan Once** - The DLB executes exactly one time when the user requests for DLB execution to start. It then stops executing until it is manually instructed to run again.

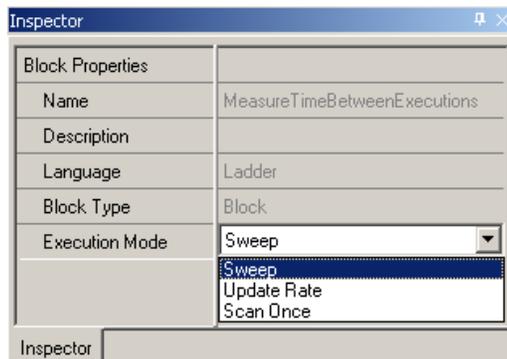


Figure 23: Properties of Diagnostic Logic Block (DLB)

### Target Properties

The Target properties include *DLB Heartbeat*, which specifies, in milliseconds, the maximum time the controller waits for a heartbeat signal from the programmer. If a heartbeat timeout occurs, the DLB will be stopped and removed from the controller. This insures that DLB execution is stopped in the event of a communications failure between the programmer and the controller.

With larger applications or a slower PC, some operations such as opening the Controller File Explorer may cause the DLB Heartbeat to time out. If this happens, you may need to increase the DLB Heartbeat interval.

The DLB Heartbeat must always be greater than the *Update Rate* setting for the Target.

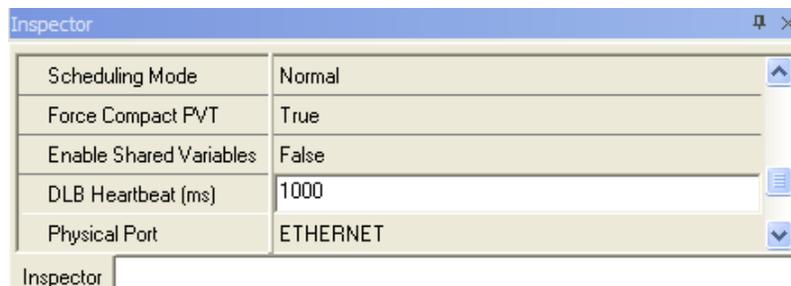


Figure 24: DLB Heartbeat Setting

**Right-click Online Operations for an Active DLB**

<b>Menu</b>	<b>Enable Rules</b>	<b>Description</b>
<b>Download</b>	Disabled if block is already running on controller, target not in programmer mode, Config+Logic is not equal, or Access Level prevents write.	Downloads block to controller, removing any other DLB that was already there.
<b>Start</b>	Disabled if block is already running, target not in programmer mode, another block is executing on controller, HWC+Logic is not equal, or Access Level prevents write	Downloads block to controller, removing any other DLB that was already there, and then starts executing block.
<b>Stop</b>	Disabled if block is not executing	Stops execution of block.
<b>Remove</b>	Disabled if block is not on controller, block is executing, or not in programmer mode	Stops block, then removes it from controller.

**DLB Online Operations**

Only a single DLB can be downloaded and executed on the controller at a time. To download an Active DLB to the controller, you must have:

- Program logic and HWC equal to the controller (Logic EQ)
- Target in programmer mode
- Sufficient privilege to write to the controller

<b>Operation</b>	<b>Minimum PACSystems RX3i Privilege Level Required</b>
Storing DLBs in STOP Mode	3
Storing DLBs in RUN Mode	4

When a DLB is downloaded, you are given the option of storing initial values or clearing memory for local variables. If another DLB is already downloaded on the controller it will be removed before the selected DLB is downloaded.

When a DLB is downloaded to the controller, all variables locally scoped to the DLB are published from the controller so that HMI's or other devices can view the data.

While a DLB is running, the active target is read-only; no changes are allowed to DLB or the application logic. If the DLB has been downloaded to the controller but is not executing, changes are allowed but the first change will remove the DLB from the controller. You will be prompted to confirm the change before the DLB is removed. Uploading of the DLB is not supported.

Once a DLB is downloaded to the controller, it can be started if the main program is running on the controller in STOP with I/O Enabled or RUN with I/O Enabled Mode.

## Removing a DLB from the Controller

The following actions will cause the DLB to be removed from the controller. If the DLB is executing, it will be stopped before being removed.

- Removing the DLB from the controller through the Online Operations menu.
- Programmer connection to controller is lost by going offline or a communication failure that causes a DLB Heartbeat timeout
- Switching from programmer mode to monitor mode
- Downloading to controller (Config, Logic, Stored Values, etc.)
- Clearing the controller, other than fault tables and controller supplemental files
- Performing any Flash operation, other than Verify
- Uploading from controller (Config, Logic, Stored Values, etc.)
- Changing the DLB that is on the controller

If there is an executing DLB, and you transition from RUN Mode to STOP Mode, the executing DLB will be stopped as well. The DLB will not be removed from the controller in this case.

If you initiate an upload, and there is a DLB on the controller, you will be prompted for confirmation and notified that the DLB will be removed and that all active DLBs will be made inactive. If there are no DLBs on the controller but there is at least one active DLB, you will be prompted for confirmation and notified that all active DLBs will be made inactive. If you choose to abort the upload, no changes are made. If you proceed, all DLBs are deactivated. If DLBs are de-activated, you will have to reactivate them manually.

When a DLB is removed from the controller, any PMM data logger (DLOG) and event queue (ELOG) files that were created by the DLB are also removed.

## Basic Steps for Using a DLB in the Controller

- 1) Create an LD Block under the Active Blocks DLB Node in the Navigator.  
You can accomplish this in several ways, such as by creating a new block under the Active Blocks node, dragging a block from the Toolchest, or copying and pasting a block from another project.
- 2) Select DLB block properties, for example, Execution Mode, as desired.
- 3) If necessary, change the Target property, DLB Heartbeat. For larger projects, you may need to increase DLB Heartbeat from its default value of 1000ms to avoid timing out while performing some operations, such as opening the Controller File Explorer.
- 4) Go online to the Controller and go into Programmer Mode, Logic Equal.
- 5) Right click the DLB and select the Online Operations menu to download the DLB to the controller and start its execution. (To download and start the DLB in one operation, select Online Operations > Start.)
- 6) Monitor DLB execution.

### Monitoring DLB Execution

There are several tools to monitor the execution of the DLB in the controller:

- DLB Local Symbolic variables monitored in Data Watch, LD Editor, or Data Monitor.
- DLB Icon shows the DLB state in the Navigator: Downloaded  to controller or Executing .
- A Proficiency View application can monitor the execution of the DLB by using its Local Symbolic Variables in Panels and Scripts.

The DLB block icon in the Navigator indicates its current state, as shown below:

**Inactive DLB -**

 (block displayed in gray)

**Active DLB Downloaded to Controller -**

 (block displayed in blue)

**Executing DLB -**

 (block displayed in green)

### 9.6.3 Diagnostic Logic Block (DLB) Example

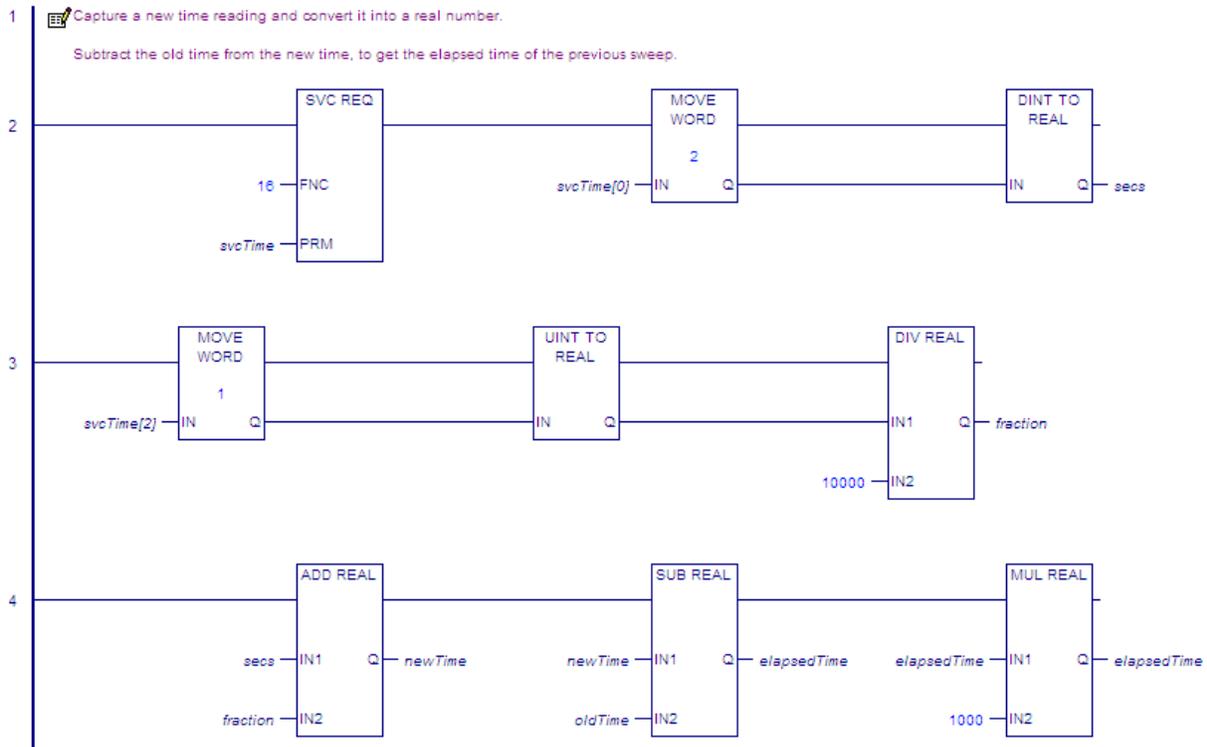
In this example, a block of LD logic is downloaded to the controller and executed.

The basic steps for using a sample DLB in the controller are as follows:

- 1) Create an LD block named *MonitorScan* and place it in the Toolchest. For information on working with the Toolchest, refer to the online help.

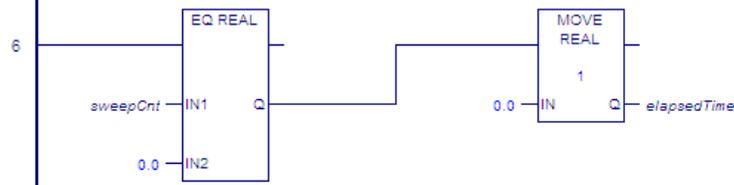
The logic in the DLB block measures Controller scan time. It calculates the Minimum (minTime), Maximum (maxTime), and Average (avgTime) time between DLB block executions. When the DLB is set to Sweep Mode, these values should be close to the Controller Sweep time.

#### Logic for the MonitorScan Block

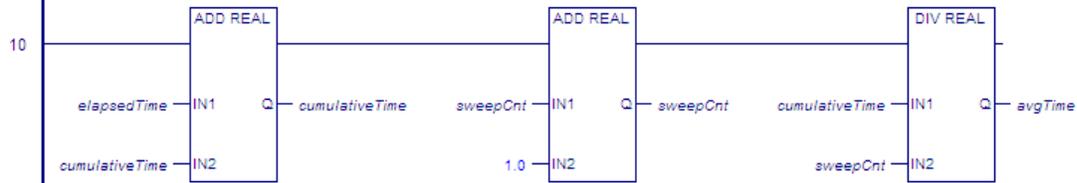
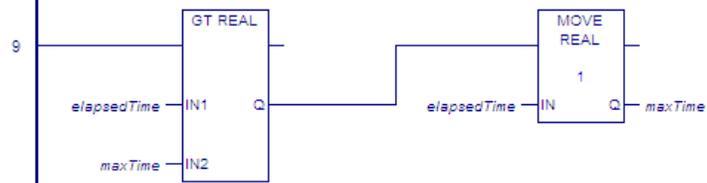
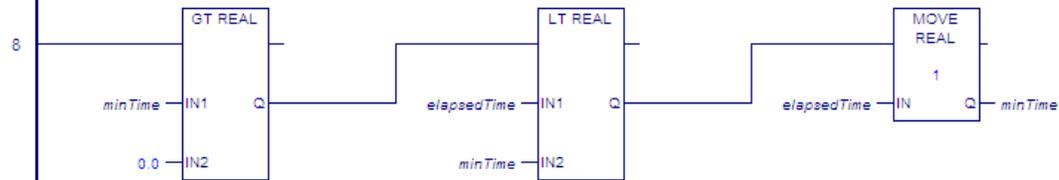


## Chapter 9. Diagnostics

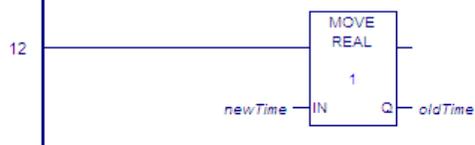
5  On the "first" sweep, force elapsed sweep time to 0.



7  Update the min, max, and average sweep times



11  Initialize old time to previous new time



- 2) Drag and drop the DLB Block from the Toolchest to the Active Blocks node in the Navigator.

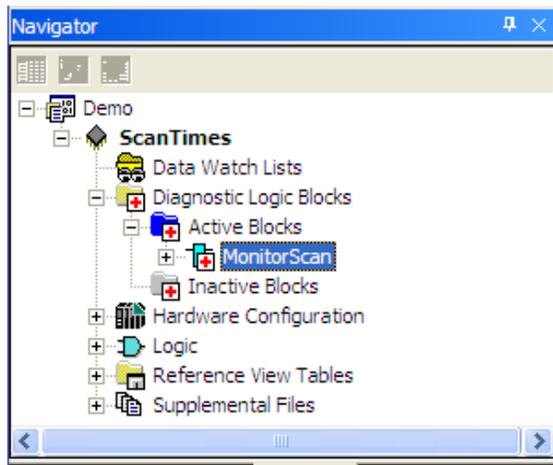


Figure 25: Drag DLB from Toolchest and Drop in Active Blocks Node

- 3) In the DLB block properties, set the Execution Mode to Sweep.

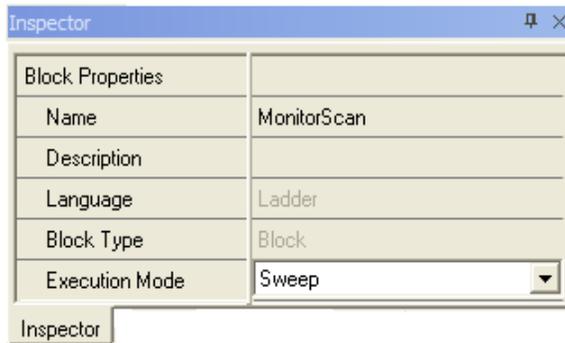


Figure 26: Set DLB Execution Mode to Sweep (Properties Tab)

- 4) Go online to the Controller, and select Programmer Mode. Put the Controller in RUN Mode or STOP Enabled Mode.
- 5) Select the DLB Online Operations > Start menu to download the DLB to the controller and start its execution.

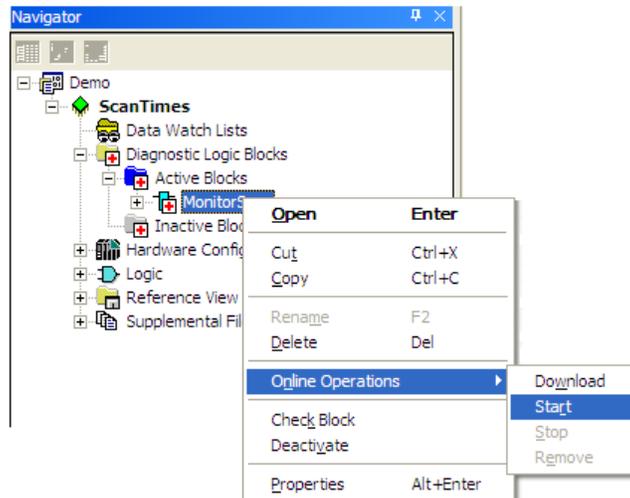


Figure 27: Start DLB Execution

- 6) In the Initialize Symbolic Variables dialog box, select how new local symbolic variables will be initialized and click OK.

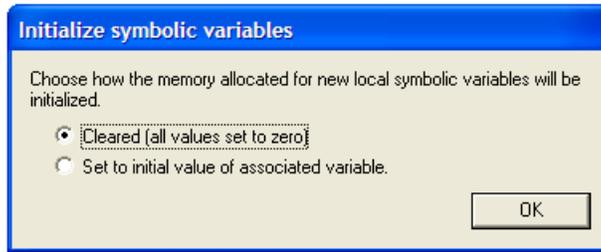


Figure 28: Initialize Local Symbolic Variables

- 7) Notice the change in the DLB Icon and the DLB status in the Status bar.

**DLB Block Icon/Status Bar Once Started.**

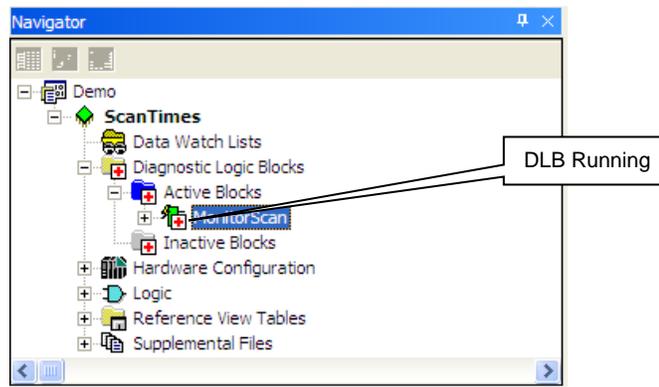


Figure 29: DLB Icon and Status Bar after Execution has Commenced

- 8) Open the DLB block and place the DLB variables in the Data Watch window to observe their operation.

Variable Name	Address	Value
GEF MonitorScan.avgTime		11.06866
GEF MonitorScan.minTime		0.0
GEF MonitorScan.maxTime		27109.38
GEF MonitorScan.elapsedTime		7.8125

Figure 30: Data Watch for DLB Variables





GE Automation & Controls  
Information Centers

**Headquarters:**

1-800-433-2682 or 1-434-978-5100

Global regional phone numbers  
are available on our web site

[www.geautomation-ip.com](http://www.geautomation-ip.com)

Copyright ©2014-2018

General Electric Company. All Rights Reserved.

\*Trademark of General Electric Company.

All other brands or names are property of their  
respective holders.

**Additional Resources**

For more information, please  
visit the GE's Automation &  
Controls web site:

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