Modicon
Ladder Logic Block Library User Guide

840 USE 101 00 Version 3.0

August 2001

Schneider Electric
One High Street
North Andover , MA 01845

Preface

The data and illustrations found in this book are not binding. We reserve the right to modify our products in line with our policy of continuous product development. The information in this document is subject to change without notice and should not be construed as a commitment by Schneider Electric.

Schneider Electric assumes no responsibility for any errors that may appear in this document. If you have any suggestions for improvements or amendments or have found errors in this publication, please notify us by using the form on the last page of this publication.

No part of this document may be reproduced in any form or by any means, electronic or mechanical, including photocopying, without express written permission of the Publisher, Schneider Electric.

Caution: All pertinent state, regional, and local safety regulations must be observed when installing and using this product. For reasons of safety and to assure compliance with documented system data, repairs to components should be performed only by the manufacturer.

MODSOFT® is a registered trademark of Schneider Electric.
The following are trademarks of Schneider Electric.

Modicon	Quantum Automation Series
Modbus Plus	Modbus
Modbus II	984 PLC
Compact 984 PLC	Modicon Micro PLC

DIGITAL® and DEC® are registered trademarks of Digital Equipment Corporation.

IBM® and IBM AT® are registered trademarks of International Business Machines Corporation.

Microsoft® and MS DOS® are registered trademarks of Microsoft Corporation.

© Copyright 2001, Schneider Electric
Printed in U.S.A.

Contents

Chapter 1 Ladder Logic Overview 1

1.1 Segments and Networks in Ladder Logic 2
1.1.1 A Ladder Logic Network 2
1.1.2 Coil Placement in a Network 3
1.1.3 Ladder Logic Segments 4
1.2 How a PLC Solves Ladder Logic 5
1.3 Ladder Logic Elements and Instructions 7

Chapter 2 Memory Allocation in a PLC 11

2.1 User Memory 12
2.1.1 User Logic 12
2.1.2 User Memory 12
2.1.3 System Overhead 13
2.1.4 Memory Backup 13
2.2 State RAM Values 14
2.2.1 A Referencing System for Inputs and Outputs 14
2.2.2 Storing Discrete and Register Data in State RAM 15
2.3 State RAM Structure 16
2.3.1 Minimum Required State RAM Values 17
2.3.2 History and Disable Bits for Discrete References 17
2.4 The Configuration Table 18
2.4.1 Assigning a Battery Coil 18
2.4.2 Assigning a Timer Register 18
2.4.3 The Time of Day Clock 19
2.4.4 Configuration Overview 20
2.5 The I/O Map Table 22
2.5.1 Determining the Size of the I/O Map Table 22
2.5.2 Writing Data to the I/O Map Table 22

Chapter 3 Ladder Logic Opcodes 23

3.1 Translating Ladder Logic Elements in System Memory 24
3.1.1 Translating Logic Elements and Non-DX Functions 24
3.2 Translating DX Instructions in the System Memory Database 26
3.2.1 How the x and z Bits Are Used in 16-bit Nodes 26
3.2.2 How the x and z Bits Are Used in 24-bit Nodes 27
3.2.3 Opcodes for Standard DX Instructions 28
3.2.4 How the y Bits are Utilized for DX Functions 28
3.3 Opcode Defaults for Loadables 29
3.3.1 How to Handle Opcode Conflicts 29

Chapter 4 Ladder Logic Elements 31

4.1 Contacts 32
4.1.1 Normally Open Contacts 32
4.1.2 Normally Closed Contacts 33
4.1.3 Positive Transitional Contacts 33
4.1.4 Negative Transitional Contacts 34
4.2 Coils 36
4.2.1 Normal Coils 36
4.2.2 Latched or Memory-retentive Coils 36
4.2.3 A Simple Contact-Coil logic Example 37
4.2.4 Coil Usage in a Logic Network 37
4.2.5 General Coil Usage Guidelines 38
4.3 Shorts 39
4.3.1 Horizontal Shorts 39
4.3.2 Vertical Shorts 39
4.4 Using Logic Elements to Create Control Circuits 40
4.4.1 A Logical AND Circuit 40
4.4.2 A Logical OR Circuit 40
4.4.3 A Logical XOR Circuit 41
4.4.4 Building a Seal Circuit 41
4.5 Storing Contacts and Coils in Registers 43
4.6 NOBT 45
4.7 NCBT 47
4.8 NBIT 49
4.9 SBIT 51
4.10 RBIT 53

4.11 Example: Implementing a Motor Starter Circuit 55

Chapter 5 Counters and Timers 59

5.1 UCTR 60
5.1.1 Characteristics 60
5.1.2 Representation in Ladder Logic 60
5.1.3 Up-Counter Example 61
5.2 DCTR 62
5.3 T1.0 Timer 64
5.3.1 Characteristics 64
5.3.2 Representation in Ladder Logic 64
5.3.3 A One-second Timer Example 66
5.4 T0.1 Timer 67
5.5 T.01 Timer 69
5.6 T1MS Timer 71
5.6.1 Characteristics 71
5.6.2 Representation in Ladder Logic 71
5.6.3 A Millisecond Timer Example 72

Chapter 6 Integer and 16-bit Math Instructions 75

6.1 ADD 76
6.2 SUB 78
6.3 MUL 80
6.4 DIV 82
6.5 AD16 84
6.6 SU16 86
6.7 TEST 88
6.8 MU16 90
6.9 DV16 93
6.10 ITOF 96
6.11 FTOI 98
6.12 BCD 100
6.13 A Fahrenheit-to-Centigrade Conversion Example 102

Chapter 7 Enhanced Math Capabilities 103

7.1 Capabilities of the EMTH Instruction 104

7.2 Double Precision EMTH Functions 107
7.2.1 Double Precision Addition 107
7.2.2 Double Precision Subtraction 108
7.2.3 Double Precision Multiplication 109
7.2.4 Double Precision Division 110
7.3 Integer EMTH Functions 111
7.3.1 Square Root 111
7.3.2 Process Square Root 112
7.3.3 Base 10 Logarithm 114
7.3.4 Base 10 Antilogarithm 115
7.4 Floating Point EMTH Functions 116
7.4.1 The IEEE Floating Point Standard 116
7.4.2 Dealing with Negative Floating Point Numbers 116
7.4.3 Integer-to-Floating Point Conversion 117
7.4.4 Integer + Floating Point Addition 118
7.4.5 Integer   Floating Point Subtraction 118
7.4.6 Integer x Floating Point Multiplication 119
7.4.7 Integer Divided by Floating Point 119
7.4.8 Floating Point   Integer Subtraction 120
7.4.9 Floating Point Divided by Integer 120
7.4.10 Integer-Floating Point Comparison 121
7.4.11 Floating Point-to-Integer Conversion 122
7.4.12 Floating Point Addition 123
7.4.13 Floating Point Subtraction 123
7.4.14 Floating Point Multiplication 124
7.4.15 Floating Point Division 124
7.4.16 Floating Point Comparison 125
7.4.17 Floating Point Square Root 126
7.4.18 Changing the Sign of a Floating Point Number 126
7.4.19 Load the Floating Point Value of p 127
7.4.20 Floating Point Sine of an Angle (in Radians) 128
7.4.21 Floating Point Cosine of an Angle (in Radians) 129
7.4.22 Floating Point Tangent of an Angle (in Radians) 130
7.4.23 Floating Point Arcsine of an Angle (in Radians) 130
7.4.24 Floating Point Arc Cosine of an Angle (in Radians) 131
7.4.25 Floating Point Arc Tangent of an Angle (in Radians) 132
7.4.26 Floating Point Conversion of Radians to Degrees 133
7.4.27 Floating Point Conversion of Degrees to Radians 134

7.4.28 Raising a Floating Point Number to an Integer Power . . 134
7.4.29 Floating Point Exponential Function 135
7.4.30 Floating Point Natural Logarithm 136
7.4.31 Floating Point Common Logarithm 137
7.4.32 Floating Point Error Report Log 138
7.5 MATH 139
7.5.1 Characteristics 139
7.5.2 Decimal Square Root 139
7.5.3 Process Square Root 140
7.5.4 Base 10 Logarithm 141
7.5.5 Base 10 Antilogarithm 142
7.6 DMTH 143
7.6.1 Characteristics 143
7.6.2 Double Precision Addition 143
7.6.3 Double Precision Subtraction 144
7.6.4 Double Precision Multiplication 145
7.6.5 Double Precision Division 146

Chapter 8 Equation Networks 149

8.1 Equation Network Structure 150
8.2 Data Types 154
8.2.1 Variable Data 154
8.2.2 Constant Data 155
8.3 Algebraic Operators 157
8.3.1 How an Equation Network Resolves an Equation 157
8.3.2 Operator Precedence 158
8.3.3 Using Parentheses in an Expression 159
8.4 Functions 161
8.4.1 Entering Functions in an Equation Network 161
8.4.2 Limits on the Argument to a Function 162
8.5 Data Conversions in an Equation Network 163
8.6 Roundoff Differences in PLCs without a Math Coprocessor 166
8.7 Benchmark Performance 167

Chapter 9 DX Move Instructions 169

9.1 DX Move Operations 170
9.1.1 DX Tables 170

9.1.2 Specifying Discrete References in a DX Table 170
9.1.3 Pointers in a DX Instruction Node 170
9.2 R→T 171
9.2.1 Characteristics 171
9.2.2 Representation 171
9.2.3 An R→T Example 172
9.3 T→R Move 174
9.3.1 Characteristics 174
9.3.2 Representation 174
9.3.3 A T→R Example 175
9.4 T→T Move 177
9.4.1 Characteristics 177
9.4.2 Representation 177
9.4.3 A T→T Example 179
9.5 FIN 181
9.6 FOUT 184
9.7 SRCH 187
9.7.1 Characteristics 187
9.7.2 Representation 187
9.7.3 A SRCH Example 188
9.8 BLKM 189
9.8.1 Characteristics 189
9.8.2 Representation 189
9.8.3 A Recipe Storage Example 190
9.9 BLKT 192
9.9.1 Characteristics 192
9.9.2 Representation 192
9.9.3 A BLKT Example 194
9.10 TBLK 195
9.10.1 Characteristics 195
9.10.2 Representation 195
9.10.3 A TBLK Example 197
9.11 IBKR 198
9.11.1 Characteristics 198
9.11.2 Representation 198
9.11.3 An IBKR Example 199
9.12 IBKW 201
9.12.1 Characteristics 201

9.12.2 Representation 201
9.12.3 An IBKW Example 202

Chapter 10 DX Matrix Instructions 205

10.1 DX Matrix Operations 206
10.2 AND 207
10.2.1 Characteristics 207
10.2.2 Representation 208
10.2.3 An AND Example 209
10.3 OR 210
10.3.1 Characteristics 210
10.3.2 Representation 210
10.3.3 An OR Example 212
10.4 XOR 213
10.4.1 Characteristics 213
10.4.2 Representation 214
10.4.3 An XOR Example 215
10.5 COMP 216
10.5.1 Characteristics 216
10.5.2 Representation 216
10.5.3 A COMP Example 217
10.6 CMPR 218
10.6.1 Characteristics 218
10.6.2 Representation 218
10.6.3 A CMPR Example 220
10.7 SENS 221
10.7.1 Characteristics 221
10.7.2 Representation 221
10.7.3 A SENS Example: Reporting Status Information 222
10.8 MBIT 224
10.9 BROT 226
10.10 A Simple Table Averaging Example 228
10.11 Setting Step Flags and Monitoring Steps in Modsoft SFC 229

Chapter 11 Monitoring Remote I/O System Status 231

11.1 STAT 232
11.2 The S901 Status Table 234

11.2.1 S901 Controller Status Words 235
11.2.2 S901 I/O Module Health Status Words 238
11.2.3 S901 RIO Communication Status Words 239
11.3 The S908 Status Table 240
11.3.1 S908 PLC Status Words 241
11.3.2 S908 I/O Module Health Status Words 244
11.3.3 S908 I/O Communication Status Words 245
11.4 The Compact PLC Status Table 250
11.4.1 Compact PLC Status Words 251
11.4.2 Compact I/O Module Health Status Words 253
11.4.3 Compact I/O Communication Status Words 253
11.5 Micro PLC Status Table 255
11.5.1 Micro PLC Status Words 256
11.5.2 Micro I/O Expansion Health 258
11.5.3 Start-up Error Codes 260
11.5.4 Micro PLC Global Communications Status 261
11.6 HLTH 264
11.6.1 Learn Mode 264
11.6.2 Monitor Mode 264
11.6.3 Characteristics 265
11.6.4 Representation 265
11.6.5 HLTH Example 272

Chapter 12 Monitoring Distributed I/O System Status 275

12.1 The DIO Health Status Table 276
12.2 DIOH 278

Chapter 13 Bypassing Networks with SKP 281

13.1 SKP 282
13.1.1 Characteristics 282
13.1.2 Representation in Ladder Logic 282
13.1.3 A Simple SKP Example 283
13.2 Off-line Instructions for Skipping Steps in Modsoft SFC 284

Chapter 14 Extended Memory Capabilities 287

14.1 Extended Memory File Structure 288

14.2 How Extended Memory Is Stored in User Memory 289
14.3 XMWT 290
14.4 XMRD 292

Chapter 15 ASCII Communication Instructions 295

15.1 READ 296
15.2 WRIT 300
15.3 Formatting Messages for ASCII READ/WRIT Operations 303
15.3.1 Format Specifiers 303
15.4 COMM 306
15.4.1 Characteristics 306
15.4.2 Representation 306
15.4.3 Message Formats for the COMM Instruction 308
15.4.4 Set-up Considerations for Control/Monitor Signals 311
15.5 ASCII Character Set 312

Chapter 16 Sequential Control Instructions 315

16.1 The Tenor Drum Model 316
16.1.1 A Mechanical Tenor Drum 316
16.1.2 Drum and ICMP Operations 317
16.2 DRUM 318
16.3 ICMP 321
16.3.1 Characteristics 321
16.3.2 Representation 321
16.3.3 Cascaded DRUM/ICMP Blocks 323
16.4 SCIF 324
16.5 A Sequence Control Example Using the SCIF Instruction 327

Chapter 17 The Checksum Instruction 329

17.1 CKSM 330

Chapter 18 The Modbus Plus Master Instruction 333

18.1 MSTR Overview 334
18.2 MSTR Function Error Codes 339
18.2.1 Modbus Plus and SY/MAX EtherNet Error Codes 339
18.2.2 SY/MAX specific Error Codes 340

18.2.3 TCP/IP EtherNet Error Codes 342
18.2.4 CTE Error Codes for SY/MAX and TCP/IP EtherNet 344
18.3 Read and Write MSTR Operations 345
18.3.1 Network Implementation 345
18.3.2 Control Block Utilization 345
18.4 Get Local Statistics MSTR Operation 347
18.4.1 Network Implementation 347
18.4.2 Control Block Utilization 347
18.5 Clear Local Statistics MSTR Operation 349
18.5.1 Network Implementation 349
18.5.2 Control Block Utilization 349
18.6 Write Global Data MSTR Operation 351
18.6.1 Network Implementation 351
18.6.2 Control Block Utilization 351
18.7 Read Global Data MSTR Operation 352
18.7.1 Network Implementation 352
18.7.2 Control Block Utilization 352
18.8 Get Remote Statistics MSTR Operation 353
18.8.1 Network Implementation 353
18.8.2 Control Block Utilization 353
18.9 Clear Remote Statistics MSTR Operation 355
18.9.1 Network Implementation 355
18.9.2 Control Block Utilization 355
18.10 Peer Cop Health MSTR Operation 357
18.10.1 Network Implementation 357
18.10.2 Control Block Utilization 357
18.10.3 Peer Cop Communications Health Status Information . . 357
18.11 Reset Option Module MSTR Operation 360
18.11.1 Network Implementation 360
18.11.2 Control Block Utilization 360
18.12 Read CTE (Config Extension Table) MSTR Operation 361
18.12.1 Network Implementation 361
18.12.2 Control Block Utilization 361
18.12.3 CTE Display Implementation 362
18.13 Write CTE (Config Extension Table) MSTR Operation 363
18.13.1 Network Implementation 363
18.13.2 Control Block Utilization 363
18.13.3 CTE Display Implementation 364

18.14 Modbus Plus Network Statistics 365
18.15 TCP/IP Ethernet Statistics 370

Chapter 19 Ladder Logic Subroutines 371

19.1 Subroutine Overview 372
19.1.1 The Value of Subroutines 372
19.1.2 Where to Store Subroutines in Ladder Logic 372
19.2 JSR 373
19.3 LAB 375
19.4 RET 377
19.5 A Subroutine Example 378
19.6 CTIF 380
19.7 Some Cautionary Notes About Subroutines 384

Chapter 20 Ladder Logic Interrupt Handling for Quantum PLCs 385

20.1 Overview 386
20.1.1 Interrupt-related Performance 386
20.1.2 Instructions Not Used in an Interrupt Handler 386
20.2 Interval Timer Interrupt (ITMR) Instruction 388
20.3 Interrupt Mask/Unmask Instructions 391
20.3.1 ID Characteristics 392
20.3.2 IE Characteristics 393
20.3.3 BMDI Characteristics 394
20.4 Immediate I/O (IMIO) Instruction 396

Chapter 21 Closed Loop Control Instructions 399

21.1 A Closed Loop Control System 400
21.1.1 Set Point and Process Variable 400
21.2 PID2 401
21.2.1 Characteristics 403
21.2.2 Representation 404
21.2.3 A PID2 Level Control Example 409
21.3 PCFL 413
21.3.1 Characteristics 413
21.3.2 Representation 414
21.3.3 Input and Output Flags 415

21.4 PCFL Advanced Calculations . . . 417
21.4.1 AVER . . . 417
21.4.2 CALC . . . 418
21.4.3 EQN . . . 420
21.5 PCFL Signal Processing Functions . . . 422
21.5.1 ALARM . . . 422
21.5.2 AIN . . . 424
21.5.3 AOUT . . . 426
21.5.4 DELAY . . . 427
21.5.5 LKUP . . . 429
21.5.6 INTEG . . . 430
21.5.7 LLAG . . . 431
21.5.8 LIMIT . . . 432
21.5.9 LIMV . . . 433
21.5.10 MODE . . . 434
21.5.11 RAMP . . . 436
21.5.12 RMPLN . . . 437
21.5.13 RATE . . . 439
21.5.14 SEL . . . 440
21.6 PCFL Regulatory Functions . . . 442
21.6.1 General Equations . . . 442
21.6.2 KPID . . . 444
21.6.3 ONOFF . . . 446
21.6.4 PID . . . 448
21.6.5 A PID Example . . . 450
21.6.6 PI . . . 453
21.6.7 RATIO . . . 455
21.6.8 TOTAL . . . 456
22.1 Loadable Software Packages . . . 460
22.1.1 Loadable Support for Controller Option Modules . . . 460
22.1.2 Other Loadable Functions . . . 460
22.2 HSBY . . . 462
22.2.1 Characteristics . . . 462
22.2.2 Representation . . . 462
22.2.3 An HSBY Reverse Transfer Example . . . 464
22.3 CHS . . . 466
22.3.1 How to Configure a Quantum Hot Standby System . . . 466
22.3.2 CHS Instruction Characteristics . . . 467

22.3.3 Representation 468
22.4 CALL 471
22.5 ESI 474
22.5.1 ESI-Driven Command Sequences 474
22.5.2 Characteristics 475
22.5.3 Representation 475
22.5.4 Error Checking 477
22.5.5 The Read ASCII Message Command 478
22.5.6 Write ASCII Message 481
22.5.7 Get Data 482
22.5.8 Put Data (Subfunction 4) 483
22.5.9 Abort (Middle Input ON) 487
22.5.10 Module Status Word 487
22.6 MBUS 489
22.6.1 Characteristics 489
22.6.2 Representation 490
22.6.3 The MBUS Get Statistics Function 491
22.7 PEER 494
22.8 Custom Loadables 496
22.8.1 Programming Environment 496
22.8.2 Characteristics 497
22.8.3 Representation 497
22.9 The EARS Loadable 499
22.9.1 PLC Functions in an Event/Alarm Recording System ... 499
22.9.2 HostePLC Interaction 499
22.9.3 The EARS Block 500
22.10 EUCA 503
22.10.1 Characteristics 503
22.10.2 Representation 504
22.10.3 A EUCA Example 505
22.10.4 Example 2 507
22.10.5 Example 3 509

Appendix Optimizing Performance via the Segment Scheduler 511

Index 527

Chapter 1
Ladder Logic Overview

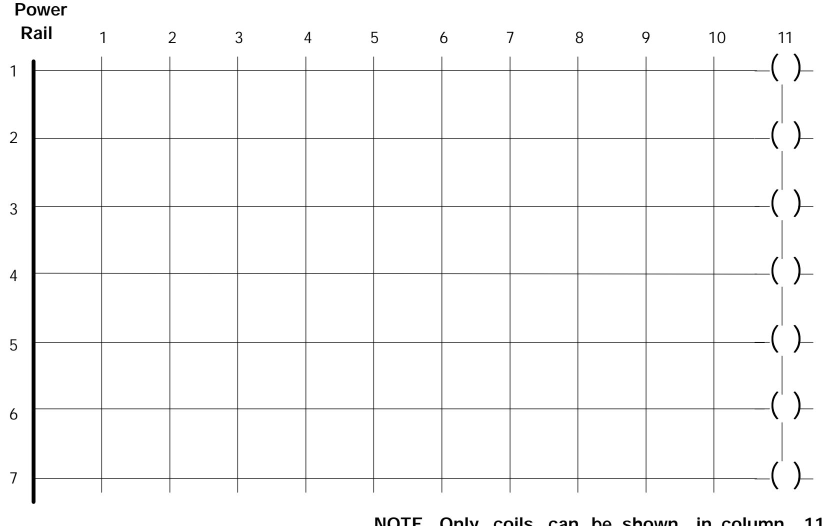
- ☐ Segments and Networks in Ladder Logic
- ☐ How a PLC Solves Ladder Logic
- ☐ Ladder Logic Elements and Instructions

1.1 Segments and Networks in Ladder Logic

Ladder logic is an easy-to-use graphical programming language that implements relay-equivalent symbology. Its major components are single-node elements and multi-node instructions. These components are programmed into networks, which are ladder logic constructs of a preset size and shape. A ladder logic program comprises a sequence of networks collected together in one or more segments.

1.1.1 A Ladder Logic Network

A network is a ladder diagram bounded on the left and right by power rails. By convention, the rail on the left is shown and the one on the right is not. Seven rungs (or rows) run from left to right between the two power rails. Each rung is eleven columns wide.



The 77 regions formed by the intersections of rungs and columns are called *nodes* . Logic elements and instructions can be programmed into these nodes. All 77 nodes in a network may be used to store ladder logic elements and instructions, which are the fundamental building blocks of the logic program. Some rules of placement apply, particularly with respect to coil placement.

1.1.2 Coil Placement in a Network

When a coil is inserted on a rung of a network, no other logic elements or instructions can be placed to the right of it on that rung. The seven nodes in the 11th column are reserved for displaying coils. Many software panels allow you to select the way you display coils in a network, either in their logic-solve positions or expanded to column 11 where they can all be viewed in parallel.

The two examples below show the same logic structure with the coils displayed differently according to user preference. The first example shows the coils displayed in their logic-solve positions and the second example shows the coils displayed in expanded positions.

Coils Displayed in Logic-solve Positions

Coils Displayed in Expanded Positions

Although the coil expansion display shows the coils in the 11th column, they are solved in their real logic-solve position. Coil 00103 is solved immediately after contact 10034 and coil 00102 is solved immediately after contact 10033 in both examples above. Coil 00101 is always the last coil solved in the network.

1.1.3 Ladder Logic Segments

Because the structure of a network is fixed, the logic program generally overlaps into multiple networks. A group of contiguous networks performing a task or subtask in the application program is called a *segment*. There is no prescribed limit on the number of networks that can be placed in a segment—size is limited only by the amount of User Memory available and by the maximum amount of PLC scan time (250 ms).

For small ladder logic applications, a single segment may be sufficient to store the whole program. For larger applications, such as multi-drop remote I/O applications, several segments may be programmed. As a rule in RIO configurations, the number of segments in the program equals the number of I/O drops; you may want to use more segments than drops, but never fewer segments than drops.

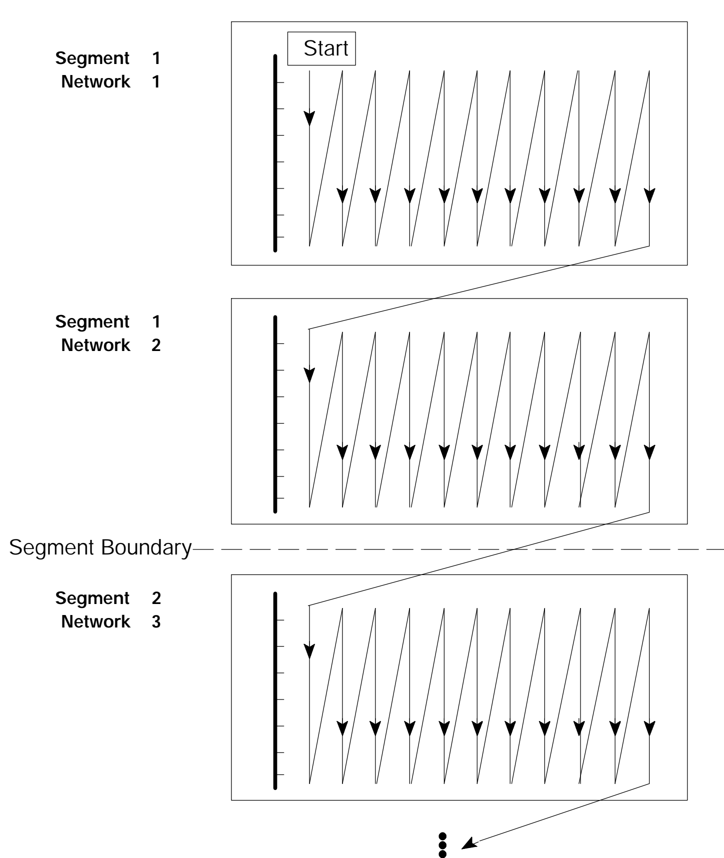
Segments are numbered 1 ... *n*, up to a maximum of 32, in the order they are created by the programmer. You may modify the order in which segments are solved with the *segment scheduler*, an editor available with your panel software that allows you to adjust the order-of-solve table in system memory. Refer to Appendix A for a description of how to improve system performance via the segment scheduler.

With some PLCs, you may also create an unscheduled segment that contains one or more ladder logic subroutines, which can be called from the scheduled segments via the JSR function.

1.2 How a PLC Solves Ladder Logic

The PLC scans the ladder logic program sequentially in the following order:

- Segments are scanned according to the way they are scheduled in an order-of-solve table known as the *segment scheduler*. The segment scheduler can be customized during system configuration, or it can default to a standard scanning sequence (segment 1 followed by segment 2 followed by segment 3, etc.)
- Networks in each segment are scanned contiguously
- Nodes within each network are scanned top to bottom, left to right



The PLC begins solving logic in the network at the top of the leftmost column and proceeds down, then moves to the top of the next column and proceeds down, as shown in the illustration. Each node is solved in the order it is encountered in the logic scan. Power flow within the network is down each column from left to right, never from bottom to top and never from right to left.

1.3 Ladder Logic Elements and Instructions

There is a core set of ladder logic elements (contacts, coils, vertical and horizontal shorts) and instructions built into all PLC firmware packages. Additional instructions are available for specific PLC types as either built-in or loadable instructions. This section provides a brief list of the available instructions and their functions; a detailed description of all instruction, including the PLC models they are available on, is provided in later chapters of this book.

Standard Ladder Logic Elements

<table>
<tr><th>Symbol</th><th>Meaning</th><th>Nodes Consumed</th></tr>
<tr><td>-| |-</td><td>A normally open (N.O.) contact</td><td>1</td></tr>
<tr><td>-|/|-</td><td>A normally closed (N.C.) contact</td><td>1</td></tr>
<tr><td>-|↑|-</td><td>A positive transitional (P.T.) contact</td><td>1</td></tr>
<tr><td>-|↓|-</td><td>A negative transitional (N.T.) contact</td><td>1</td></tr>
<tr><td>-( )-</td><td>A normal coil</td><td>1</td></tr>
<tr><td>-(L)- -(M)-</td><td>A memory-retentive or latched coil; the two symbols mean the same thing, and the user may select the preferred version for on-line display</td><td>1</td></tr>
<tr><td>—</td><td>A horizontal short</td><td>1</td></tr>
<tr><td>|</td><td>A vertical short</td><td>0</td></tr>
</table>

Standard Ladder Logic Instructions for All PLCs

Instruction	Meaning	Nodes Consumed
Counter and Timer Instructions		
UCTR	Counts up from 0 to a preset value	2
DCTR	Counts down from a preset value to 0	2
T1.0	Timer that increments in seconds	2
T0.1	Timer that increments in tenths of a second	2
T.01	Timer that increments in hundredths of a second	2
Integer Math Instructions		
ADD	Adds top node value to middle node value	3
SUB	Subtracts middle node value from top node value	3
MUL	Multiplies top node value by middle node value	3
DIV	Divides top node value by middle node value	3

DX Move Instructions		
R→T	Moves register values to a table	3
T→R	Moves specified table values to a register	3
T→T	Moves a specified set of values from one table to another table	3
BLKM	Moves a specified block of data	3
FIN	Specifies first-entry in a FIFO queue	3
FOUT	Specifies first-entry out of a FIFO queue	3
SRCH	Performs a table search	3
STAT	Displays status registers from status table in system memory	1
DX Matrix Instructions		
AND	Logically ANDs two matrices	3
OR	Does logical inclusive OR of two matrices	3
XOR	Does logical exclusive OR of two matrices	3
COMP	Performs logical complement of values in a matrix	3
CMPR	Logically compares values in two matrices	3
MBIT	Logical bit modify	3
SENS	Logical bit sense	3
BROT	Logical bit rotate	3
Skip-Node Instruction		
SKP	Skips a specified number of networks in a ladder logic program	1

Some ladder logic instructions are standard (built in) to some PLCs but unavailable in others. For example, PLCs with the Modbus Plus communication capability built in it are shipped with an MSTR instruction in the firmware while PLCs that cannot operate on Modbus Plus do not support this instruction. Here is a list of these select built-in instructions:

Built-in Ladder Logic Instructions for Select PLCs

Instruction	Meaning	Nodes Cons umed
Bit Manipulation Instructions		
NOBT	Uses a register to represent 16 bits as N.O. contacts	2
NCBT	Uses a register to represent 16 bits as N.C. contacts	2
NBIT	Uses an output register to represent 16 bits as normal coils	2
SBIT	Latches a bit in an output register to remain ON	2
RBIT	Clears a bit that has been set via the SBIT instruction	2

Other Math Instructions		
AD16	Signed/unsigned 16-bit addition	3
SU16	Signed/unsigned 16-bit subtraction	3
TEST	Compares the magnitudes of the values in the top and middle nodes	3
MU16	Signed/unsigned 16-bit multiplication	3
DV16	Signed/unsigned 16-bit division	3
ITOF	Signed/unsigned integer-to-floating point conversion	3
FTOI	Floating point-to-signed/unsigned integer conversion	3
EMTH	Performs 38 math operations, including floating point math operations and extra integer math operations such as square root	3
BCD	Converts binary values to BCD values and BCD values to binary values	3
Equation Network ■	Uses an entire ladder logic network as an editing environment where a user can enter equations in a standard syntax	77
Interrupt Instructions		
ITMR ■	Defines an interval timer that generates interrupts into the normal logic scan and initiates an interrupt handling subroutine	2
ID ■	Interrupt disable	1
IE ■	Interrupt enable	1
BMDI ■	Masks timer-generated and local I/O-generated interrupts, performs a block data move, then unmasks the interrupts	3
IMIO ■	Permits immediate access of specified I/O modules from within ladder logic	2
ASCII Messaging Instructions		
READ	Reads data entered at an ASCII device into the PLC via its RIO link	3
WRIT	Sends a message from the PLC to an ASCII device via its RIO link	3
COMM	Combines both ASCII READ and WRITE capabilities for simple (canned) messages in the Micro PLCs	3
Ladder Logic Subroutine Instructions		
JSR	Jumps from scheduled logic scan to a ladder logic subroutine	2
LAB	Labels the entry point of a ladder logic subroutine	1
RET	Returns from the subroutine to scheduled logic	1
CTIF	Used to set up high-speed input terminals on a Micro PLC for scheduled-logic interrupts and/or counter/timer operations	3
Other Special-purpose Instructions		
CKSM	Calculates any of four types of checksum operations (CRC-16, LRC, straight CKSM, and binary add)	3
MSTR	Specifies a function from a menu of networking operations	3
PID2	Performs proportional-integral-derivative calculations for closed-loop control	3
PCFL	Accesses advanced functions from a process control library	3
TBLK	Moves a block of data from a table to another specified block area	3
BLKT	Moves a block of registers to specified locations in a table	3
SCIF	Provides tenor drum sequencer functionality and the ability to do input comparisons within the application program	3
T1MS	A timer that increments in milliseconds	3

IBKR	Performs an indirect block read operation—i.e., copies specified registers to a working block of holding registers	3
IBKW	Performs an indirect block write operation—i.e., copies registers from a working block to individual register locations	3

Other instructions are available for specific PLCs as loabable functions. Loadables support optional software development products that can be purchased for special applications. The loadable instructions may be used only with specific PLC models. Loadable instructions include:

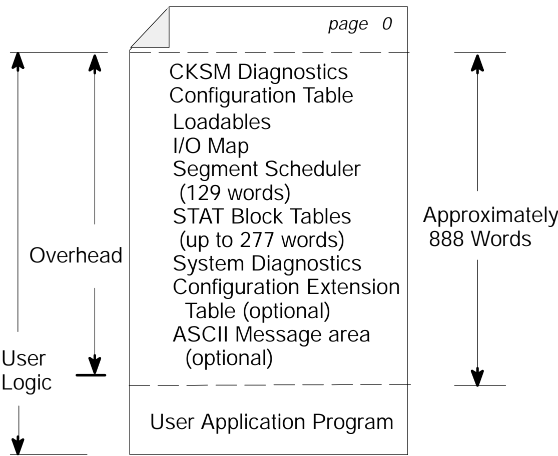
Instruction	Meaning	Nodes Consumed
HSBY	Sets up a 984 hot standby back-up PLC that takes control of the application if the primary PLC goes down	3
CHS	Optional method for setting up a Quantum hot standby back-up PLC	3
CALL	Supports 984 Coprocessor option module applications	3
MBUS PEER	For initiating message transactions on a Modbus II network	3
ESI	Optional instruction in Quantum PLCs that supports the 140 ESI 062 10 Quantum ASCII module	3
FN*xx*	A three-node template for creating custom loadable instructions via Assembly or C source code	3
DRUM ICMP	Supports sequence control application logic in some PLC models that do not have the built-in SCIF instruction	3
MATH DMTH	Support some square root, logarithm, and double-precision math functions in PLCs that cannot support the Enhanced Math library	3
EARS	Supports an event/alarm recording system by tracking events/alarms and reporting time-stamped messages	3
EUCA	Performs an engineering unit conversion algorithm	3
HLTH	Detects changes in the I/O system and reports problems on an exception-only basis	3

Chapter 2
Memory Allocation in a PLC

- ☐ User Memory
- ☐ State RAM Values
- ☐ State RAM Structure
- ☐ The Configuration Table
- ☐ The I/O Map Table

2.1 User Memory

User memory is the space provided in the PLC for the logic program and for system overhead. User memory sizes vary from 1K ... 64K words, depending on PLC type and model. Each word in user memory is stored on page 0 in the PLC's memory structure; words may be either 16 or 24 bits long, depending on the CPU size.

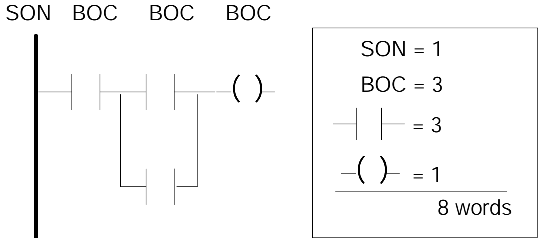


2.1.1 User Logic

The amount of space available for application logic is calculated by subtracting the amount of space consumed by system overhead from the total amount of user logic. System overhead in a relatively conservative system configuration can be expected to consume around 1000 words; system configurations with moderate or large I/O maps will require more overhead.

2.1.2 User Memory

Ladder logic requires one word of either 16-bit or 24-bit memory to uniquely identify each node in an application program. Contacts and coils each occupy one node, and therefore one word. Instructions, which usually comprise two or three nodes, require two or three words, respectively. Other elements that control program scanning—start of a network (SON), beginning of a column (BOC), and horizontal shorts—use one word of user logic memory as well.



Note: Vertical shorts do not consume any words of user memory.

2.1.3 System Overhead

System overhead refers to the contents of a set of tables where the system's size, structure, and status are defined. Some overhead tables have a predetermined amount of memory allocated to them. The configuration table, for example, contains 128 words, and the order-of-solve table (the segment scheduler) contains 129 words. Other tables, such as the I/O map (aka traffic cop), can consume a large amount of memory, but its size is not predetermined.

Optional pieces of system overhead—e.g., the loadable table, the ASCII message area, the configuration extension table—may or may not consume memory depending on the requirements of your application.

2.1.4 Memory Backup

User memory is stored in CMOS RAM. In the event that power is lost, CMOS RAM is backed up by a long-life (typically 12-month) battery. In many PLC models, the battery is a standard part of the hardware package; in smaller-scale PLCs—e.g., the Micro PLCs—a battery is available as an option.

In the case of the Micro PLCs, where the battery is an option, an area in its Flash memory is available for backing up user logic. (Flash is a standard feature on the Micros.)

2.2 State RAM Values

As part of your PLC's configuration process, you specify a certain number of discrete outputs or coils, discrete inputs, input registers, and output holding registers available for application control. These inputs and outputs are placed in a table of 16-bit words in an area of system memory called *state RAM*.

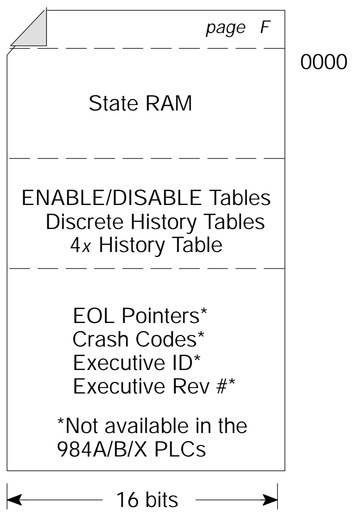
2.2.1 A Referencing System for Inputs and Outputs

The system uses a reference numbering system to identify the various types of inputs and outputs. Each reference number has a leading digit that identifies its data type (discrete input, discrete output, register input, register output) followed by a string of digits indicating its unique location in state RAM:

Reference Indicator	Reference Type	Meaning
0*x*	discrete output or coil	Can be used to drive a real output through an output module or to set one or more internal coils in state RAM. The state of a coil can be used to drive multiple contacts.
1*x*	discrete input	Can be used to drive contacts in the logic program. Its ON/OFF state is controlled by an input module.
3*x*	input register	Holds numerical inputs from an external source—for example, a thumbwheel entry, an analog signal, data from a high speed counter. A 3*x* register can also be used to store 16 contiguous discrete signals, which may be entered into the register in either binary or binary coded decimal (BCD) format.
4*x*	output holding register	Can be used to store numerical (decimal or binary) information in state RAM or to send the information to an output module.
6*x*	extended memory register	Stores binary information in extended memory area; available only in PLCs with 24-bit CPUs that support extended memory—the 984B, the E984-785, and the Quantum Automation Series PLCs

2.2.2 Storing Discrete and Register Data in State RAM

State RAM data is stored in 16-bit words on page F in System Memory. The state RAM table is followed by a discrete history table that stores the state of the bits at the end of the previous scan, and by a table of the current ENABLE/DISABLE status of all the discrete (0*x* and 1*x*) values in state RAM.

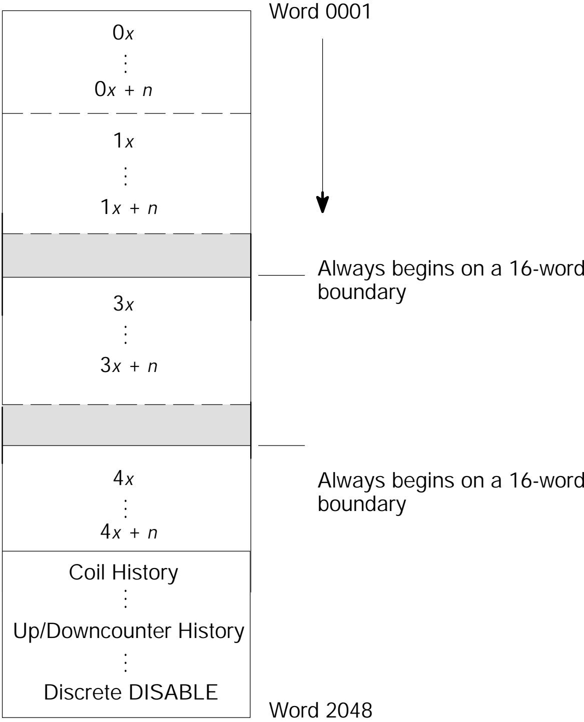


Each 0*x* or 1*x* value implemented in user logic is represented by one bit in a word in state RAM, by a bit in a word in the history table, and by a bit in a word in the DISABLE table. In other words, for every discrete word in the state RAM table there is one corresponding word in the history table and one corresponding word in the DISABLE table.

Counter input states for the previous scan are represented on page F in an up-counter/down-counter history table. Each counter register is represented by a single bit in a word in the table; a value of 1 indicates that the top input was ON in the last scan, and a value of 0 indicates that the top input was OFF in the last scan.

2.3 State RAM Structure

Words are entered into the state RAM table from the top down in the following order:



Discrete references come before registers, the 0*x* words first followed by the 1*x* words. The discrete references are stored in words containing 16 contiguous discrete references.

The register values follow the discrete words. Blocks of 3*x* and 4*x* register values must each begin at a word that is a multiple of 16. For example, if you allocate five words for eighty 0*x* references and five words for eighty 1*x* references, you have used words 0001 ... 0010 in state RAM. Words 0011 ... 0016 are then left empty so that the first 3*x* reference begins at word 0017.

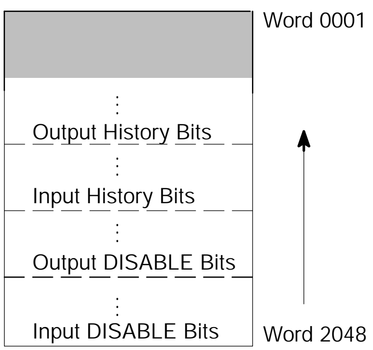
2.3.1 Minimum Required State RAM Values

A minimum configuration consists of the following allocations in state RAM:

Reference Type	Minimum Words		Minimum Bits (Discretes)	
	for Modsoft	for P190	for Modsoft	for P190
Discrete out (0*x*)	3	1	48	16
Discrete in (1*x*)	1	1	16	16
Register in (3*x*)	1	1		
Register out (4*x*)	1	1		

2.3.2 History and Disable Bits for Discrete References

For each word allocated to discrete references, two additional words are allocated in the history/disable tables. These tables follow the state RAM table on page F in system memory. They are generated from the bottom up in the following manner:



2.4 The Configuration Table

The configuration table is one of the key pieces of overhead contained in system memory. It comprises 128 consecutive words and provides a means of accessing information defining your control system capabilities and your user logic program.

With your programming panel software, you can access the configurator editor, which allows you to specify the configuration parameters—such as those shown on the following page—for your control system.

When a PLC's memory is empty—in a state called DIM AWARENESS—you are not able to write a I/O map or a user logic program. Therefore, the first programming task you must undertake with a new PLC is to write a valid configuration table using your configurator editor.

2.4.1 Assigning a Battery Coil

A 0*x* coil can be set aside in the configuration to reflect the current status of the PLC's battery backup system. If this coil has been set and is queried, it displays a discrete value of either 0, indicating that the battery system is healthy, or 1, indicating that the battery system is not healthy.

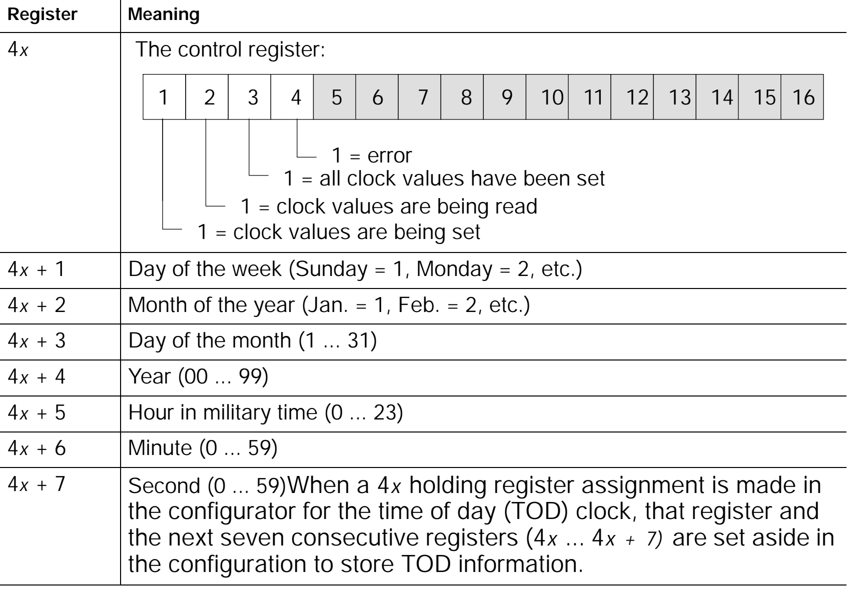
2.4.2 Assigning a Timer Register

A 4*x* register can be set aside in the configuration as a synchronization timer. It stores a count of clock cycles in 10 ms increments. If this register is set and queried, it displays a free-running value that ranges from 0000 to FFFF hex with wrap-around to 0000.

☞ **Note:** *If you are doing explicit address routing in bridge mode on a Modbus Plus network* , the location of the explicit address table in the configuration is dependent on the timer register address—i.e., a timer register must be assigned in order to create the explicit address table. The explicit address table can consist of from 0 ... 10 blocks, each block containing five consecutive 4*x* registers. The address of the first block in the explicit address table begins with the 4*x* register immediately following the address assigned to the timer register. Therefore, when you assign the timer register, you must choose a 4*x* register address that has the next 5 ... 50 registers free for this kind of application.

2.4.3 The Time of Day Clock

When a 4*x* holding register assignment is made in the configurator for the time of day (TOD) clock, that register and the next seven consecutive registers (4*x* ... 4*x + 7)* are set aside in the configuration to store TOD information. The block of registers is implemented as follows:



Register	Meaning
4*x*	The control register: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 = error 1 = all clock values have been set 1 = clock values are being read 1 = clock values are being set
4*x* + 1	Day of the week (Sunday = 1, Monday = 2, etc.)
4*x* + 2	Month of the year (Jan. = 1, Feb. = 2, etc.)
4*x* + 3	Day of the month (1 ... 31)
4*x* + 4	Year (00 ... 99)
4*x* + 5	Hour in military time (0 ... 23)
4*x* + 6	Minute (0 ... 59)
4*x* + 7	Second (0 ... 59)When a 4*x* holding register assignment is made in the configurator for the time of day (TOD) clock, that register and the next seven consecutive registers (4*x* ... 4*x + 7)* are set aside in the configuration to store TOD information.

The block of registers is implemented as follows. For example, if you configured register 40500 for your TOD clock, set the bits appropriately as shown above, then read the clock values at 9:25:30 on Tuesday, July 16, 1991, the register values displayed in decimal format would read:

Register	Meaning
400500	0110000000000000
400501	3 (decimal)
400502	7 (decimal)
400503	16 (decimal)
400504	91 (decimal)
400505	9 (decimal)
400506	25 (decimal)
400507	30 (decimal)

2.4.4 Configuration Overview

Data Type	Format	Default Setting	Notes and Exceptions
Configuration Size			
# of coils	Even multiple of 16	16	
# of discrete inputs	Even multiple of 16	16	
# of register outputs		01	
# of register inputs		01	
# of I/O drops	Up to 32, depending on PLC type	01	Used only when I/O is configured in drops.
# of I/O modules	Up to 1024, depending on PLC type	00	Not displayed by editor; used by system to calculate I/O map words.
# of logic segments	Generally equal to # of drops	00	Add one additional segment for subroutines.
# of I/O channels	Even number from 02 ... 32	02	Used only when I/O is configured in channels.
Memory size	PLC-dependent	PLC-dependent	
Modbus (RS-232) Port Parameters			
Communication mode	ASCII or RTU	RTU	
Baud rate	50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200	9600	
Parity	ON/OFF; EVEN/ODD	ON/EVEN	
Stop bit(s)	1 or 2	2	
Device address	001 ... 247	001	
Delay time (in ms)	01 ... 20 (representing 10 ... 200 ms)	01 (10 ms)	Modbus port delay times are implemented only in the 984A/B/X PLCs.

ASCII Message Table			
# of messages	Up to 9999	00	If your PLC doesn't support remote I/O, it cannot support ASCII devices. (exception: The Micros)
Size of message area	Decimal > 0 < difference between memory size (32K or 64K) and sys- tem. overhead	00	
# of ASCII ports	Two per drop, up to 32	00	
ASCII port parameters	Baud	1200	
	Parity	ON/EVEN	
	# of stop bits	01	
	# of data bits per char- acter	08	
	Presence of a keyboard	NONE	
Simple ASCII input	A 4*x* value representing the first of 32 registers for simple ASCII input	NONE	Only a 984B PLC supports simple ASCII input.
Simple ASCII output	A 4*x* value representing the first of 32 registers for simple ASCII output	NONE	Only 984A and 984B PLCs support simple ASCII output.
Special Functions			
SKIP functions allowed	YES/NO	NO	
Timer register	A 4*x* register set aside to hold a number of 10 ms clock cycles	NONE	
TOD clock	A 4*x* register, the first of eight reserved for time of day values	NONE	
Battery coil	A 0*x* reference reflect- ing the status of battery backup system	00000	Once a battery coil is placed in a Configura- tion Table, it cannot be removed.
Loadable Instructions			
Install loadable	PROCEED or CANCEL		Various controllers sup- port different kinds of loadable instruction sets. Make sure that your loadables and con- troller are compatible.
Delete loadable(s	DELETE ALL, DELETE ONE, or CANCEL		
Writing Configurator Data to System Memory			
Write data as specified	PROCEED or CANCEL	NONE	PROCEED will over- write any previous Table data.

2.5 The I/O Map Table

Just as a PLC needs to be physically linked to I/O modules in order to become a working control system, the references in user logic need to be linked in the system architecture to the signals received from the input modules and sent to the output modules. The I/O map table provides that link.

2.5.1 Determining the Size of the I/O Map Table

The I/O map directs data flow between the input/output signals and the user logic program; it tells the PLC how to implement inputs in user logic and provides a pathway down which to send signals to the output modules. The I/O map table, which is stored on page 0 in system memory, consumes a large but not predetermined amount of system overhead.

Its length is a function of the number of discrete and register I/O points your system has implemented and is defined by the type of I/O modules you specify in the configuration table.

The minimum allowable size of the I/O map table is nine words.

2.5.2 Writing Data to the I/O Map Table

With your programming panel software, you can access a I/O map editor that allows you to define:

- The number of drops in the remote I/O system
- The number of discretes/registers that may be used for input and output
- The number, type, and slot location of the I/O modules in the drop
- The reference numbers that link the discretes/registers to the I/O modules
- Drop hold-up time for each I/O drop
- ASCII messaging port addresses (if used) for any drop

Chapter 3
Ladder Logic Opcodes

- Translating Ladder Logic Elements in the System Memory Database
- Translating DX Functions in the System Memory Database
- Opcode Defaults for Loadables

3.1 Translating Ladder Logic Elements in the System Memory Database

A PLC automatically translates symbolic ladder elements and function blocks into database nodes that are stored on page 0 in system memory. A node in ladder logic is a 16- or 24-bit word—an element such as a contact translates into one database node, while an instruction such as an ADD block translates into three database nodes. The database format differs for 16-bit and 24-bit nodes:

16-bit Node Format

x	x	x	x	x	y	y	y	y	y	y	y	y	z	z	z

24-bit Node Format

x	x	x	x	x	x	x	x	y	y	y	y	y	y	y	y	y	y	y	y	y	z	z	z

The five most significant bits in a 16-bit node and the eight most significant bits in a 24-bit node—the *x* bits—are reserved for *opcodes* . An opcode defines the type of functional element associated with the node—for example, the code 01000 specifies that the node is a normally open contact, and the code 11010 specifies that the node is the third of three nodes in a multiplication function block.

3.1.1 Translating Logic Elements and Non-DX Functions

When the system is translating standard ladder logic elements and non-DX function blocks, it uses the remaining (*y* and *z*) bits as *pointers* to register or bit locations in State RAM associated with the discretes or registers used in your ladder logic program.

With a 16-bit node, 11 bits are available as state RAM pointers, giving you a total addressing capability of 2048 words. The maximum number of configurable registers in most 16 bit machines is 1920, with the balance occupied by up to 128 words (2048 bits) of discrete reference, disable, and history bits. An exception is the 984-680/-685 PLCs, which have an extended registers option that supports 4096 registers in state RAM.

With a 24-bit node, 16 bits are available as state RAM pointers. The maximum number of configurable registers in a 24-bit machine is 9999.

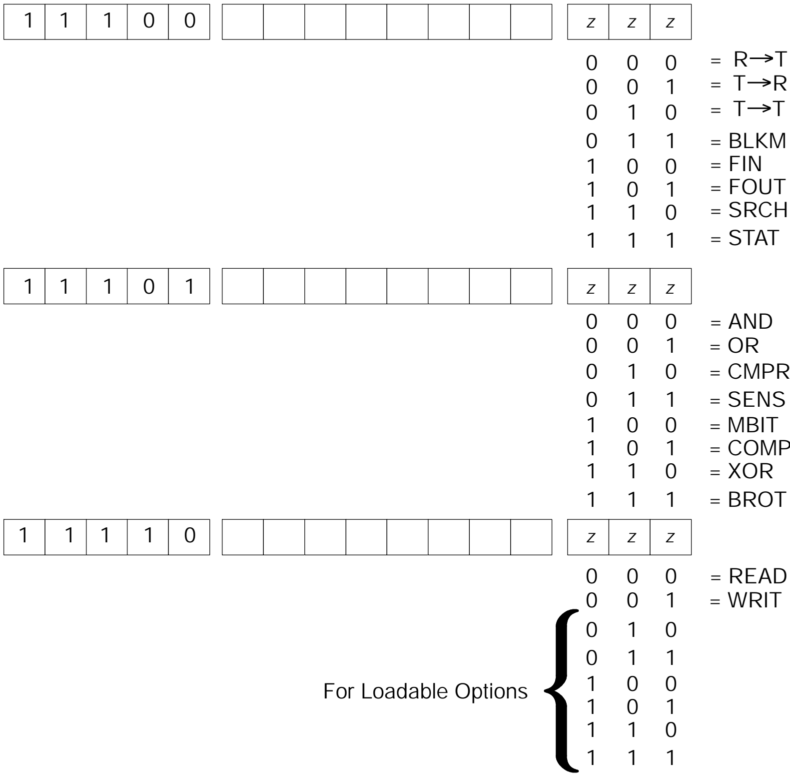
Opcodes are generally expressed by their hex values:

00	Beginning of a column in a network
01	Beginning of a column in a network
02	Beginning of a column in a network
03	Beginning of a column in a network
04	Start of a network
05	I/O exchange/End-of-Logic
06	Null Element
07	Horizontal short
08	N.O. contact
09	N.C. contact
0A	P.T. contact
0B	N.T. contact
0C	Normal coil
0D	Memory-retentive (latched) coil
0E	Constant quantity skip function
0F	Register quantity skip function
10	Constant value storage
11	Register reference
12	Discrete group reference
13	DCTR instruction
14	UCTR instruction
15	T1.0 instruction
16	T0.1 instruction
17	T.01 instruction
18	ADD instruction
19	SUB instruction
1A	MULT instruction
1B	DIV instruction
31	AD16 instruction
32	SU16 instruction
33	MU16 instruction
34	DV16 instruction
35	TEST instruction
36	ITOF instruction
37	FTOI instruction
5E	PID2 instruction
7F	EMTH instruction
9F	BLKT instruction
BE	LAB instruction
BF	CKSM or MSTR instruction
DE	DMTH or JSR instruction
DF	TBLK instruction
FE	RET instruction

3.2 Translating DX Instructions in the System Memory Database

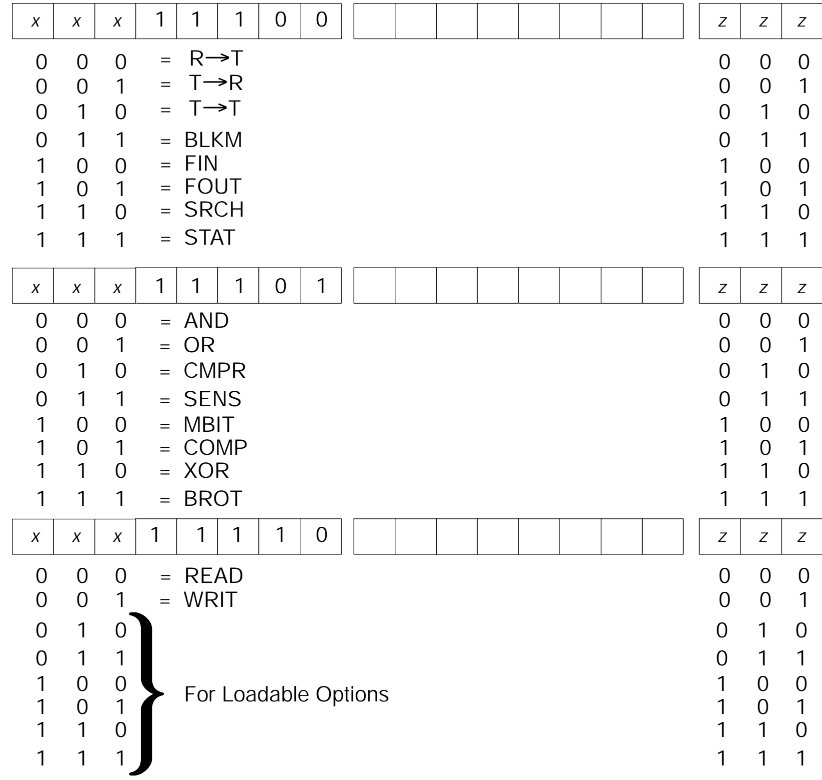
3.2.1 How the x and z Bits Are Used in 16-bit Nodes

When you are using a 16-bit CPU, you are left with only four more x-bit combinations—11100, 11101, 11110, and 11111—with which to express opcodes for the DX instructions. To gain the necessary bit values, the system uses the three least significant (z) bits along with the x bits to express the opcodes:



3.2.2 How the x and z Bits Are Used in 24-bit Nodes

In the 24-bit CPUs, the three most significant x bits are used to indicate the type of DX function:



The z bits, which simply echo the three most significant x bits, may be ignored in the 24-bit nodes.

3.2.3 Opcodes for Standard DX Instructions

1C	R→T instruction
3C	T→R instruction
5C	T→T instruction
7C	BLKM instruction
9C	FIN instruction
BC	FOUT instruction
DC	SRCH instruction
FC	STAT instruction
20	DIOH instruction
1D	AND instruction
3D	OR instruction
5D	CMPR instruction
7D	SENS instruction
9D	MBIT instruction
BD	COMP instruction
DD	XOR instruction
FD	BROT instruction
1E	READ instruction
3E	WRIT instruction
7E	XMWT instruction
9E	XMRD instruction
51	IBKR
52	IBKW

Note: These opcodes are hard-coded in the appropriate system firmware, and they cannot be altered.

3.2.4 How the y Bits are Utilized for DX Functions

The *y* bits in a database node holding DX function data contain a binary number that expresses the number of registers being transferred in the function. A 16-bit database node has 8 y bits. A 16-bit CPU is, therefore, machine limited to no more than 255 transfer registers per DX operation.

A 24-bit database node has 13 y bits. A 24-bit CPU is, therefore, capable of reaching a theoretical machine limit of 8191 transfer registers per DX operation; practically, however, the greatest number of transfer registers allowed in a 24-bit DX operation is 999.

3.3 Opcode Defaults for Loadables

Various ladder logic instructions are available only in loadable software packages. When instructions are loaded to a controller, they are stored in RAM on page 0 in system memory. They are not resident on the EPROM. The loadable functions have the following opcodes:

FF	HSBY instruction
5F	CALL, FN*xx* , or EARS instruction
1F	MBUS instruction
3F	PEER instruction
DE	DMTH instruction
BE	MATH or EARS instruction
FE	DRUM instruction
7F	ICMP instruction

3.3.1 How to Handle Opcode Conflicts

Note: No two instructions with the same opcode can coexist on a PLC.

The easiest way to stay out of trouble is to never employ two loadables with conflicting opcodes in your user logic. If you are using MODSOFT panel software, it allows you to change the opcodes for loadable instructions. The *lodutil* utility in the Modicon Custom Loadable Software package (SW-AP98-GDA) also allows you to change loadable opcodes.

Warning! If you modify any loadables so that their opcodes are different from the ones shown in this chapter , you must use caution when porting user logic to or from your controller . The opcode conflicts that can result may hang up the target controller or cause the wrong function blocks to be executed in ladder logic.

Chapter 4
Ladder Logic Elements

- Contacts
- Coils
- Shorts
- Using Logic Elements to Create Control Circuits
- Storing Contacts and Coils in Registers
- NCBT
- NOBT
- NBIT
- SBIT
- RBIT
- Example: Implementing a Motor Starter Circuit

4.1 Contacts

Contacts are used to pass or inhibit power flow in a ladder logic program. They are discrete—i.e., each consumes one I/O point in ladder logic. A single contact can be tied to a 0*x* or 1*x* reference number in the PLC's state RAM, in which case each contact consumes one node in a ladder network.

Four kinds of contacts are available:

- Normally open (N.O.) contacts
- Normally closed (N.C.) contacts
- Positive transitional (P.T.) contacts
- Negative transitional (N.T.) contacts

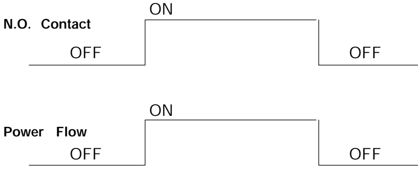
4.1.1 Normally Open Contacts

Size
One node high

Symbol

–| |–

Function
Passes power when its referenced coil or input is ON:



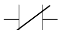
PLC Compatibility
Standard in all PLC types

Opcode
08 hex

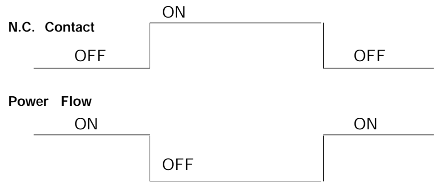
4.1.2 Normally Closed Contacts

Size
One node high

Symbol



Function
Passes power when its referenced coil or input is OFF:



PLC Compatibility
Standard in all PLC types.

Opcode
09 hex

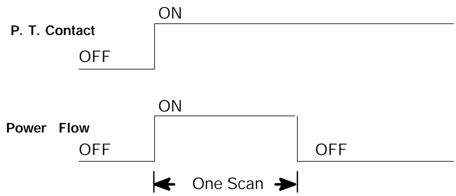
4.1.3 Positive Transitional Contacts

Size
One node high

Symbol

Function

Passes power for only one scan as the contact or coil transitions from OFF to ON:



PLC Compatibility

Standard in all PLC types

Opcode

0A hex

4.1.4 Negative Transitional Contacts

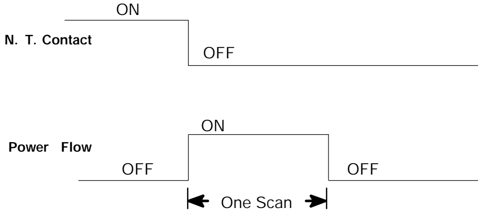
Size

One node high

Symbol

Function

Passes power for only one scan as the contact or coil transitions from ON to OFF:



PLC Compatibility

Standard in all PLC types

Opcode

0B hex

Note: A transitional contact will pass power continuously if the referenced coil is skipped by a SKP instruction (see Chapter 13) or by the segment scheduler (see Appendix A). A transitional contact may not pass power if it is referenced to an input that has been scheduled to read from the I/O drop more than once per scan via the segment scheduler.

4.2 Coils

A coil is a discrete output that is turned ON and OFF by power flow in the logic program. A single coil is tied to a 0*x* reference in the PLC's state RAM. Because output values are updated in state RAM by the PLC, a coil may be used internally in the logic program or externally via the I/O map to a discrete output unit in the control system. When a coil is ON, it either passes power to a discrete output circuit or changes the state of an internal relay contact in state RAM.

There are two types of coils:

- A normal coil
- A memory-retentive, or latched, coil

4.2.1 Normal Coils

Size
One node high

Symbol

–( )–

Function
When power is removed from a PLC, a normal coil will be turned OFF. Once power is restored , the coil will always be in the OFF state on the first logic scan.

PLC Compatibility
Standard in all PLC types

Opcode
0C hex

4.2.2 Latched or Memory-retentive Coils

Size
One node high

Symbol

—(M)— or —(L)— User-selectable choice of symbolic display

Function

If a memory-retentive (or latched) coil is ON at the time a PLC loses power, the coil will come back up in an ON state when power is restored. The coil will maintain that ON state for the first logic scan, and then the logic program will take control.

PLC Compatibility

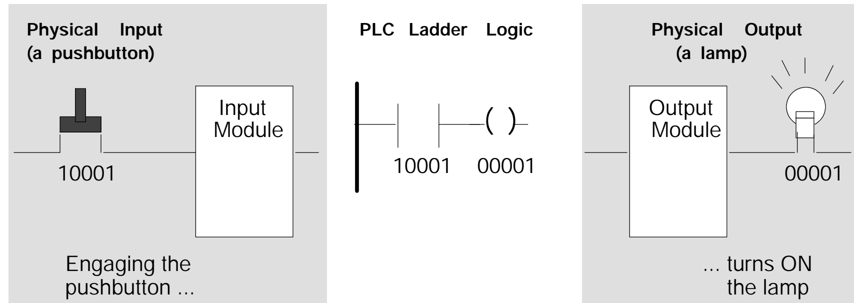
Standard in all PLC types

Opcodes

0D hex

4.2.3 A Simple Contact-Coil logic Example

Here is a simple example of how an N.O. contact senses the state of a physical input—in this case, a pushbutton—and how a normal coil controls the state of a physical output—in this case, a lamp. The pushbutton is used to close a normally open contact and thereby pass power to the lamp (the normal coil).



4.2.4 Coil Usage in a Logic Network

A ladder logic network can contain a maximum of seven coils, up to one per rung. See page 3 for a discussion of how coils can and cannot be arranged in a network.

4.2.5 General Coil Usage Guidelines

Once a 0x reference number has been assigned to a coil, it cannot be assigned to any other coils in the logic program. It can be referenced to any number of relay contacts, which can then be controlled via the state of the coil with the same reference number. Most panel software packages have a feature called tracing with which you can locate the positions in ladder logic of the contacts controlled by a coil. Refer to your software user manual for more details.

Enable/Disable Capabilities for Discrete Values

Via panel software, you may disable a logic coil or a discrete input in your logic program. A disable condition will cause the input field device to have no control over its assigned 1*x* logic and the logic to have no control over the disabled 0*x* value.

Memory protection in the PLC must be OFF before you disable (or enable) a coil or a discrete input.

Caution: There is an important exception you need to be aware of when disabling coils: data transfer functions that allow coils in their *destination* nodes recognize the current ON/OFF state of all coils, whether they are disabled or not, and cause the logic to respond accordingly. If you are expecting a disabled coil to remain disabled in the DX function, your application may experience unexpected and undesirable effects.

Forcing Discretes ON and OFF

Most panel softwares also provide FORCE ON and FORCE OFF capabilities. When a coil or discrete input is disabled, you can change its state from OFF to ON with FORCE ON, and from ON to OFF with FORCE OFF.

When a coil or discrete input is enabled, it cannot be forced ON or OFF.

4.3 Shorts

Shorts are simply straight-line connections between contacts and/or instructions in a ladder logic network. Shorts may be inserted horizontally or vertically in a network.

4.3.1 Horizontal Shorts

Size

One node high

Symbol

———

Function

Expands logic horizontally along a rung in a ladder logic network

PLC Compatibility

Standard in all PLC types

Opcode

07 hex

4.3.2 Vertical Shorts

Size

Unique among logic elements in that it does not use any nodes in a logic network

Function

Connects contacts or instructions vertically in a network column, or node inputs and outputs to create either/or conditions. When two contacts are connected by vertical shorts, power is passed when one or both contacts receive power.

PLC Compatibility

Standard in all PLC types

4.4 Using Logic Elements to Create Control Circuits

Horizontal and vertical shorts can be combined with contacts to create logic circuits that control the flow of power to coils.

4.4.1 A Logical AND Circuit

A string of contacts placed in series along a rung in a network creates a logical AND condition. For example, in order to enable coil 00001 below, N.C. contact 10001 must be turned OFF by its associated field device and N.O. contacts 10002 and 10003 must be turned ON by their associated field devices. If one or more of these conditions is not true, the coil is disabled.

10001 10002 10003 00001

4.4.2 A Logical OR Circuit

Contacts in parallel on different rungs of the ladder can create various logical OR conditions. First, let's look at an OR condition. The top rung of ladder below contains two N.O. contacts (10001 and 10002), and the lower rung contains a single contact (10003) followed by a horizontal short. A vertical short connects the two rungs in the third column of the network. Power can pass through the network to energize coil 00001 when either contacts 10001 and 10002 are energized or when contact 10003 is energized.

10001 10002 00001

10003

4.4.3 A Logical XOR Circuit

Contacts and vertical shorts can also be used to create Exclusive-OR (XOR) circuits. Below is a circuit that prevents coil 00001 from energizing when contacts 10001 and 10002 pass power simultaneously.

10001 10002 00001
10001 10002

Coil 00001 can be enabled only when one but not both of the following conditions is true:

- The field device sensed by contact 10001 is ON and the field device sensed by contact 10002 is OFF
- The field device sensed by contact 10001 is OFF and the field device sensed by contact 10002 is ON

4.4.4 Building a Seal Circuit

A simple seal circuit can be built by placing two contacts in parallel with one of the contacts referenced to a coil in the circuit. In the top two rungs of the ladder below is a circuit with two contacts (10001 and 10002) that monitor two pushbuttons (STOP and START), and a coil (00001) that controls a motor starter:

Start P/B Stop P/B
Motor Starter
10002 10001 00001
00001
Running Lamp (Red)
00001 00002
Stopped Lamp (Green)
00001 00003

The Stop pushbutton is sensed by N.C. contact 10001 and makes sure that no power is being passed through the circuit while it is ON. The Start pushbutton is sensed by N.O. contact 10002. This contact passes power only when input 10001 is OFF, causing N.C. contact 10001 to pass power.

Contact 00001, which is placed in parallel with the Start pushbutton contact, is pulled ON when the Motor Starter coil (00001) is turned ON. It latches the Start condition open once contact 10002 has been opened. Once coil 00001 has been enabled, latched contact 00001 keeps it ON even after the Start pushbutton has been disengaged; the only way to turn OFF the coil is by engaging the Stop pushbutton—i.e., turning ON N.C. contact 10001.

The logic on the bottom two rungs of the network turns ON one of two colored lamps that indicate the current state of the motor starter. When the Motor Starter coil is ON, it pulls both of the contacts in the bottom rungs ON. When these two contacts are ON, N.O. contact 00001 enables coil 00002, which turns ON a red Motor Starter Running lamp, and N.C. contact 00001 disables coil 00003, which turns OFF a green Motor Starter Stopped lamp.

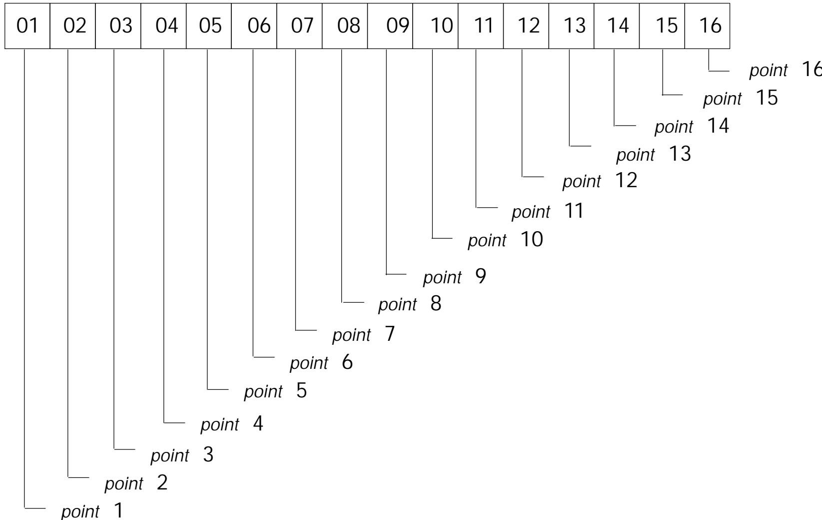
When these two contacts are OFF, N.O. contact 00001 disables coil 00002, which turns OFF the Motor Starter Running lamp, and N.C. contact 00001 enables coil 00003, which turns ON the Motor Starter Stopped lamp.

4.5 Storing Contacts and Coils in Registers

Five two-high instructions, standard in the Quantum Series PLCs. allow you to store groups of up to 16 coils or contacts in 3*x* or 4*x* registers:

- NOBT, for sensing one of up to 16 bits in a 3*x* or 4*x* register and representing it as if it were an N.O. contact
- NCBT, for sensing one of up to 16 bits in a 3*x* or 4*x* register and representing it as if it were an N.C. contact
- NBIT, for setting and clearing a specified bit in a 4*x* register
- SBIT, for latching a specified bit in a 4*x* register
- RBIT, for resetting a specified bit that has been set in a 4*x* register

These instructions handle each bit in the register like a discrete point as follows:



Advantages

These instructions provide greater flexibility in developing applications. They dramatically increase the number of discrete points available for an application. The top output from the instructions automatically indicates the state of the specified element—i.e., you do not need to use the SENS or MBIT instructions to check or modify the current logic state of the bit.

4.6 NOBT

The normally open bit (NOBT) instruction lets you sense the logic state of a bit in a register by specifying its associated bit number in the bottom node. The bit is representative of an N.O contact.

4.6.1 Characteristics

Size
Two nodes high

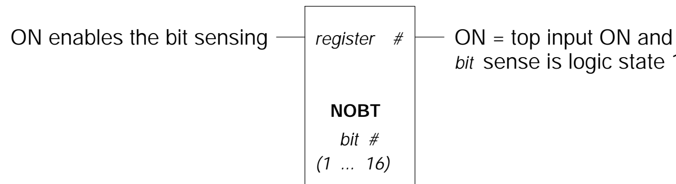
PLC Compatibility

- Standard in the Quantum Automation Series PLCs
- Not available in other PLC types

Opcode
40 hex

4.6.2 Representation in Ladder Logic

Block Structure



Input
NOBT has one control input to the top node, which enables the operation when it is ON.

Output
NOBT produces one output from the top node. It passes power when the top input is ON and when the specified bit is ON—i.e., its logic state is 1.

Top Node Content

The *register number* entered in the top node is the 3*x* input register or a 4*x* holding register whose bit pattern is being used to represent N.O. contacts.

Bottom Node Content

The *bit #* entered in the bottom node indicates which one of the 16 bits is being sensed.

4.7 NCBT

The normally closed bit (NCBT) instruction lets you sense the logic state of a bit in a register by specifying its associated bit number in the bottom node. The bit is representative of an N.C contact. It passes power from the top output when the specified bit is OFF and the top input is ON.

4.7.1 Characteristics

Size
Two nodes high

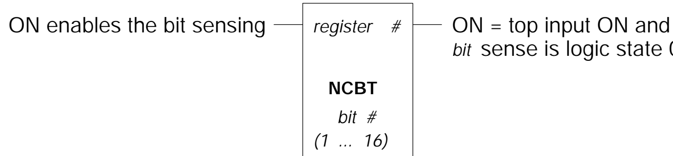
PLC Compatibility

- Standard in the Quantum Automation Series PLCs
- Not available in other PLC types

Opcode
41 hex

4.7.2 Representation in Ladder Logic

Block Structure



Input
NCBT has one control input to the top node, which enables the operation when it is ON.

Output
NCBT produces one output from the top node. It passes power when the top input is ON and when the specified bit is OFF—i.e., its logic state is zero.

Top Node Content

The *register number* entered in the top node is the 3*x* input register or a 4*x* holding register whose bit pattern is being used to represent N.C. contacts.

Bottom Node Content

The *bit #* entered in the bottom node indicates which one of the 16 bits is being sensed.

4.8 NBIT

The normal bit (NBIT) instruction lets you control the state of a bit from a register by specifying its associated bit number in the bottom node. The bits being controlled are similar to coils—when a bit is turned ON, it stays ON until a control signal turns it OFF.

☞ **Note:** The NBIT instruction does not follow the same rules of network placement as 0*x*-referenced coils do. An NBIT instruction cannot be placed in column 11 of a network and it can be placed to the left of other logic nodes on the same rungs of the ladder.

4.8.1 Characteristics

Size
Two nodes high

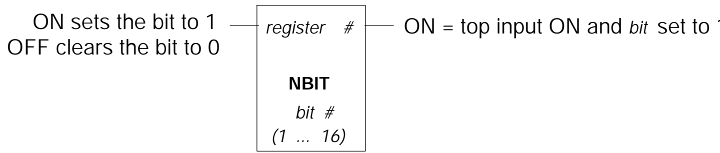
PLC Compatibility

- Standard in the Quantum Automation Series PLCs
- Not available in other PLC types

Opcode
42 hex

4.8.2 Representation in Ladder Logic

Block Structure



Input
NBIT has one control input to the top node, which sets the specified bit to 1 when it is ON and clears the specified bit to 0 when it is OFF.

Output
NBIT produces one output from the top node. It echos the state of the top input, thereby indicating the OFF/ON state of the specified bit.

Top Node Contents

The 4*x register number* entered in the top node is the holding register whose bit pattern is being controlled.

Bottom Node Contents

The *bit #* entered in the bottom node indicates which one of the 16 bits is being controlled.

4.9 SBIT

The set bit (SBIT) instruction lets you set the state of the specified bit to ON (1) by powering the top input.

Note: The SBIT instruction does not follow the same rules of network placement as 0*x*-referenced coils do. An SBIT instruction cannot be placed in column 11 of a network and it can be placed to the left of other logic nodes on the same rungs of the ladder.

4.9.1 Characteristics

Size
Two nodes high

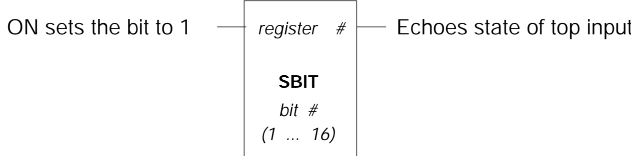
PLC Compatibility

- Standard in the Quantum Automation Series PLCs
- Not available in other PLC types

Opcode
43 hex

4.9.2 Representation in Ladder Logic

Block Structure



Input
SBIT has one control input to the top node, which sets the specified bit to 1 when it is ON. The bit remains set after power is removed from the input.

Output
SBIT produces one output from the top node, which echoes the state of the top input.

Top Node Content

The 4*x register number* entered in the top node is the holding register whose bit pattern is being controlled.

Bottom Node Content

The *bit #* entered in the bottom node indicates which one of the 16 bits is being set.

4.10 RBIT

The reset bit (RBIT) instruction lets you clear a latched-ON bit by powering the top input. The bit remains cleared after power is removed from the input. This instruction is designed to clear a bit set by the SBIT instruction.

Note: The RBIT instruction does not follow the same rules of network placement as 0*x*-referenced coils do. An RBIT instruction cannot be placed in column 11 of a network and it can be placed to the left of other logic nodes on the same rungs of the ladder.

4.10.1 Characteristics

Size
Two nodes high

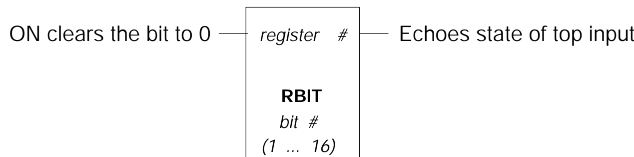
PLC Compatibility

- Standard in the Quantum Automation Series PLCs
- Not available in other PLC types

Opcode
44 hex

4.10.2 Representation in Ladder Logic

Block Structure



Input
RBIT has one control input to the top node, which clears the specified bit to 0 when it is ON.

Output
RBIT produces one output from its top node, which echoes the state of the top input.

Top Node Content

The 4*x register number* entered in the top node is the holding register whose bit pattern is being controlled.

Bottom Node Content

The *bit #* entered in the bottom node indicates which one of the 16 bits is being cleared.

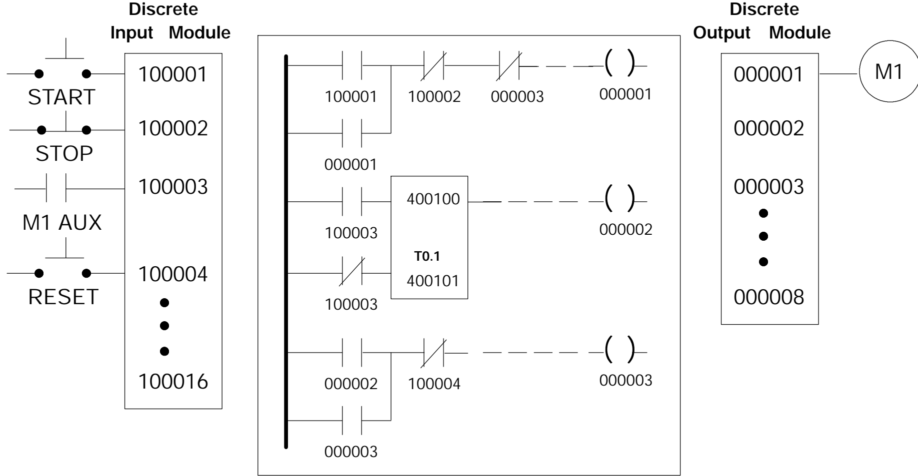
4.1 1 Example: Implementing a Motor Starter Circuit

Below are three ladder logic schemes, each designed to control the same simple motor starter circuit. The first example is a conventional contact/coil relay logic implementation. The second example is an imitation of the first example, this time using bits within a register instead of discretes to control the circuit. The third example shows how the register implementation can be optimized in ladder logic.

The motor (M1) is is turned ON with a START button mapped to input 1 and turned OFF via a STOP button mapped to input 2. An auxiliary contact (M1 AUX) is mapped to input 3. This contact will trigger a timer if for some reason it stays open for a preprogrammed amount of time after the START button is engaged. The contact will turn M1 OFF if it has not been closed by the time the timer expires. The only way to restart M1 after it has been shut OFF by the timer is via a RESET button, which is mapped to input 4.

A Discrete Implementation

Below is an illustration of conventional ladder logic used to implement the motor starter. A 16-point discrete input module has been I/O mapped to 16 contiguous 1*x* references (100001 ... 100016) in logic, and an 8-point discrete output module has been I/O mapped to eight contiguous 0*x* references (000001 ... 000008).



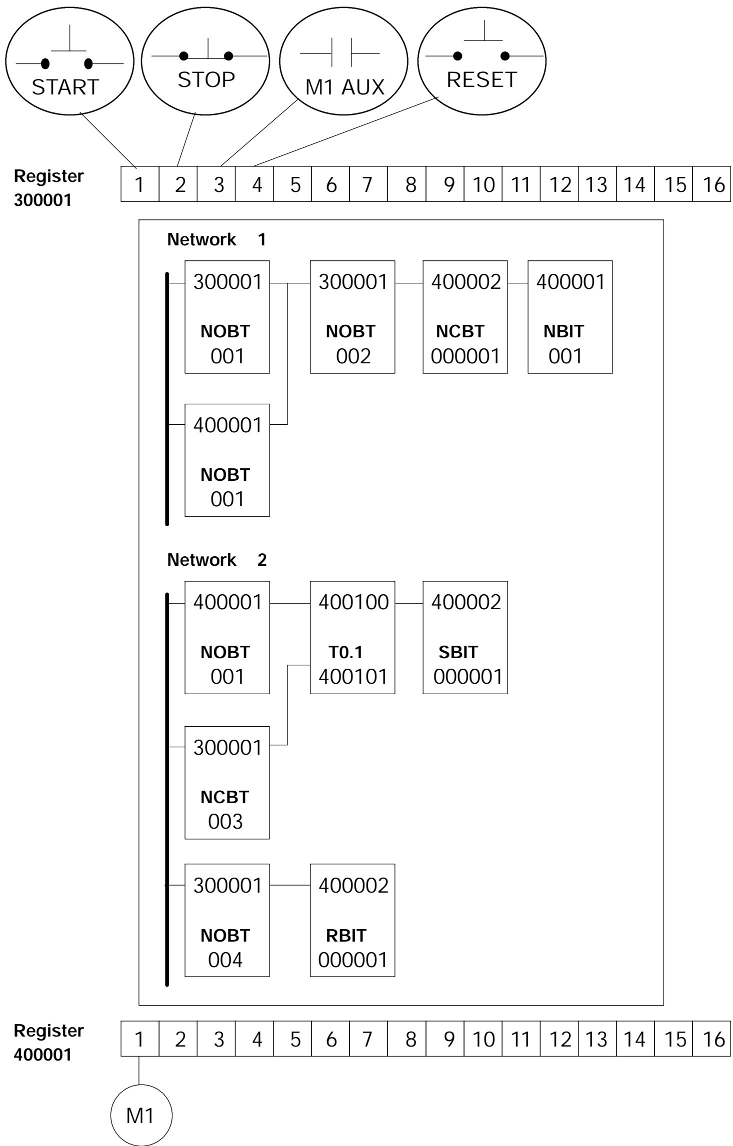
The top two rows of logic implement the START and STOP buttons, passing power to coil 000001 when contact 100001 is made and removing power when contact 100002 is disengaged.

The middle two rows of logic implement the auxiliary contact (M1 AUX), starting a timer if contact 100001 is made but contact 100003 remains open. If contact 100003 is not made by the time the *timer preset* is reached (see page 67), coil 000001 will be turned OFF.

The bottom two rows of logic control the RESET button (100004). If M1 AUX has turned OFF coil 000001, the only way to restart M1 is by pushing the RESET button then the START button. If the M1 AUX contact (100003) still remains open after M1 has been restarted, the timer will be restarted and M1 AUX will again turn OFF M1 when the *timer preset* is reached.

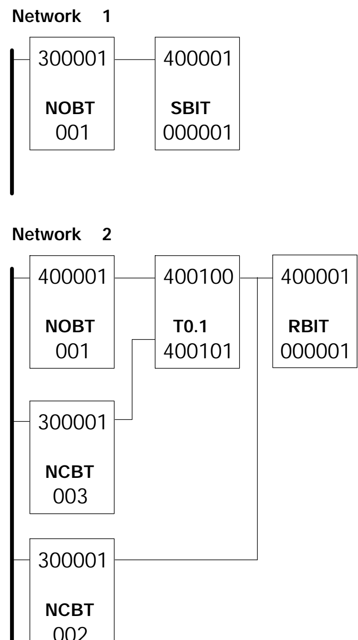
A Register Implementation

Here is another ladder logic scheme that performs the same control functions. This time, the discrete input module is I/O mapped to bits in a 3*x* register and the discrete output module is I/O mapped to bits in a 4*x* register.



An Optimized Register Implementation

Here is an optimized ladder logic scheme that uses registers to control the motor starter circuit. With this logic, you do not need to separately program the RESET control.



Chapter 5
Counters and Timers

- □ UCTR
- □ DCTR
- □ T1.0 Timer
- □ T0.1 Timer
- □ T.01 Timer
- □ T1MS Timer

5.1 UCTR

The UCTR instruction counts control input transitions from OFF to ON up from zero to a *counter preset* value.

5.1.1 Characteristics

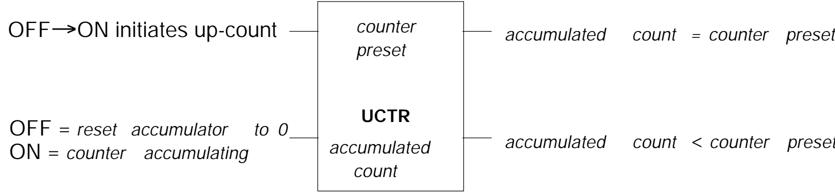
Size
Two nodes high

PLC Compatibility
Standard in all PLC types

Opcode
14 hex

5.1.2 Representation in Ladder Logic

Block Structure



Inputs
UCTR has two input controls. The input to the top node initiates the counter operation. The input to the bottom node is ON while the counter is accumulating. If it goes OFF, the *accumulated count* is reset to zero.

Outputs
UCTR can produce one of two possible outputs. The output from the top node passes power when the *accumulated count* reaches the specified *counter preset*. The output from the bottom node passes power if the *accumulated count* value falls below the *counter preset* value.

Top Node Content

The *counter preset* stored in the top node of each instruction can be

- Displayed explicitly as an integer
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

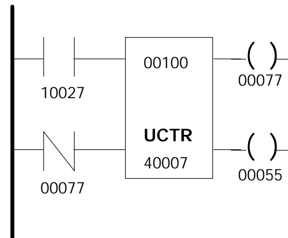
If the *counter preset* is entered as an integer, the range of allowable values is a function of the type of PLC in use:

PLC Type	Allowable Integer Range
Any PLC with a 16 Bit CPU	1 ... 999
E685/785 PLCs, L785 PLCs, and Quantum Series PLCs	1 ... 65, 535
Other PLCs with 24-bit CPUs	1 ... 9,999

Bottom Node Content

The 4*x* register entered in the bottom node contains the *accumulated count* , which increments by one on each transition from OFF to ON of the top input until it reaches the specified *counter preset* value.

5.1.3 Up-Counter Example



When contact 00077 is receiving power, UCTR is enabled. Each time contact 10027 transitions from OFF to ON, the *accumulated count* value increments by 1. When the value reaches 100 (when contact 10027 has transitioned 100 times), the top output passes power. Coil 00077 is energized, and coil 00055 is de-energized. Contact 00077 loses power when coil 00077 is energized, and the *accumulated count* value is reset to 0 on the next scan. On the next scan, coil 00077 is de-energized. Contact 00077 is then re-energized and the UCTR function is enabled.

5.2 DCTR

The DCTR instruction counts control input transitions from OFF to ON down from a *counter preset* value to zero.

5.2.1 Characteristics

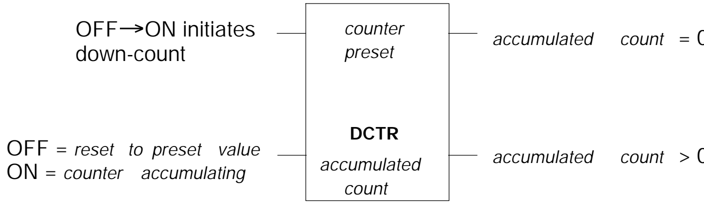
Size
Two nodes high

PLC Compatibility
Standard in all PLC types

Opcode
13 hex

5.2.2 Representation in Ladder Logic

Block Structure



Inputs
DCTR has two input controls. The input to the top node initiates the counter operation. The input to the bottom node is ON while the counter is accumulating. If it goes OFF, the *accumulated count* is reset to the *counter preset* value.

Outputs
DCTR can produce one of two possible outputs. The output from the top node passes power when the *accumulated count* decrements to zero. The output from the bottom node passes power if the *accumulated count* value is greater than the zero.

Top Node Content

The *counter preset* stored in the top node of each instruction can be

- Displayed explicitly as an integer
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

If the *counter preset* is entered as an integer, the range of allowable values is a function of the type of PLC in use:

PLC Type	Allowable Integer Range
Any PLC with a 16 Bit CPU	1 ... 999
E685/785 PLCs, L785 PLCs, and Quantum Series PLCs	1 ... 65, 535
Other PLCs with 24-bit CPUs	1 ... 9,999

Bottom Node Content

The 4*x* register entered in the bottom node contains the *accumulated count* , which decrements by one on each transition from OFF to ON of the top input until it reaches zero.

5.3 T1.0 Timer

The T1.0 timer instruction measures time in one-second increments. It can be used for timing an event or creating a delay.

5.3.1 Characteristics

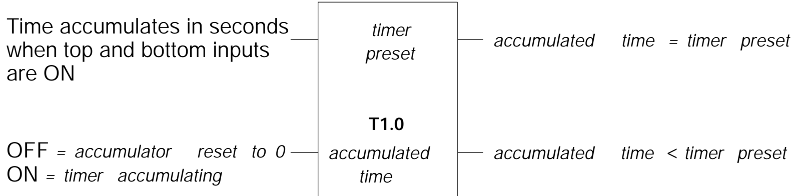
Size
Two nodes high

PLC Compatibility
Standard in all PLC types

Opcode
15 hex

5.3.2 Representation in Ladder Logic

Block Structure



Inputs
T1.0 has two input controls. The input to the top node initiates the timer operation. The input to the bottom node is ON while the timer is accumulating. If it goes OFF, the *accumulated time* is reset to zero.

Outputs
T1.0 can produce one of two possible outputs. The output from the top node passes power when the *accumulated time* reaches the specified *timer preset* value. The output from the bottom node passes power if the *accumulated time* value drops below the *timer preset* value.

Top Node Content

The *timer preset* in the top node is a value that specifies how many one-second increments the timer can accumulate. This value can be:

- Displayed explicitly as an integer
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

If the *timer preset* is entered as an integer, the range of allowable values is a function of the type of PLC in use:

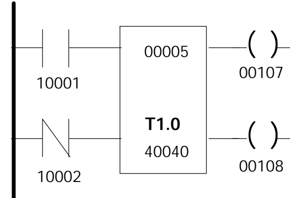
PLC Type	Allowable Integer Range
Any PLC with a 16 Bit CPU	1 ... 999
E685/785 PLCs, L785 PLCs, and Quantum Series PLCs	1 ... 65, 535
Other PLCs with 24-bit CPUs	1 ... 9,999

Bottom Node Content

The 4*x* register entered in the bottom node stores the *accumulated time* count in one-second increments.

Caution: If you cascade T1.0 timers with *presets* of 1, the timers will time-out together; to avoid this problem, change the *presets* to 10 and substitute a T0.1 timer .

5.3.3 A One-second Timer Example



The example above assumes that 10002 is closed (timer enabled) and that the value contained in register 40040 is 0. Because 40040 does not equal the *timer preset* (5), coil 00107 is OFF and coil 00108 is ON. When 10001 is closed, 40040 begins to accumulate counts at 1 s intervals until it reaches 5. At that point, 00107 is ON and 00108 is OFF.

When 10002 is opened, 40040 resets to 0, coil 00107 goes OFF, and 00108 goes ON.

☞ **Note:** If the *accumulated time* value is less than the *timer preset* value, the bottom output will pass power even though no inputs to the block are present.

5.4 T0.1 Timer

A T0.1 instruction measures time in in tenth-of-a-second increments. It can be used for timing an event or creating a delay. T0.1 has two control inputs and can produce one of two possible outputs. An output passing power indicates a timer error.

5.4.1 Characteristics

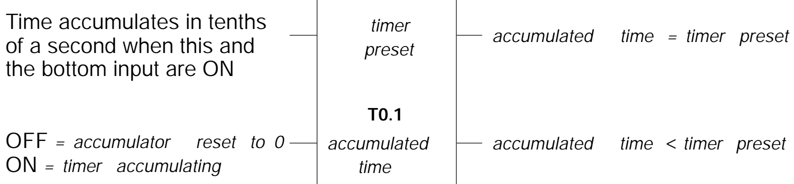
Size
Two nodes high

PLC Compatibility
Standard in all PLC types

Opcode
16 hex

5.4.2 Representation in Ladder Logic

Block Structure



Inputs
T0.1 has two input controls. The input to the top node initiates the timer operation. The input to the bottom node is ON while the timer is accumulating. If it goes OFF, the *accumulated time* is reset to zero.

Outputs
T0.1 can produce one of two possible outputs. The output from the top node passes power when the *accumulated time* reaches the specified *timer preset* value. The output from the bottom node passes power if the *accumulated time* value drops below the *timer preset* value.

Top Node Content

The *timer preset* in the top node is a value that specifies how many tenth-of-a-second increments the timer can accumulate. This value can be:

- Displayed explicitly as an integer
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

If the *timer preset* is entered as an integer, the range of allowable values is a function of the type of PLC in use:

PLC Type	Allowable Integer Range
Any PLC with a 16 Bit CPU	1 ... 999
E685/785 PLCs, L785 PLCs, and Quantum Series PLCs	1 ... 65, 535
Other PLCs with 24-bit CPUs	1 ... 9,999

Bottom Node Content

The 4*x* register entered in the bottom stores the *accumulated time* count in tenth-of-a-second increments.

Caution: If you cascade T0.1 timers with *presets* of 1, the timers will time-out together; to avoid this problem, change the *presets* to 10 and substitute a T.01 timer .

5.5 T.01 Timer

The T.01 instruction measures time in hundredth-of-a-second intervals. It can be used for timing an event or creating a delay. T.01 has two control inputs and can produce one of two possible outputs. An output passing power indicates a timer error.

5.5.1 Characteristics

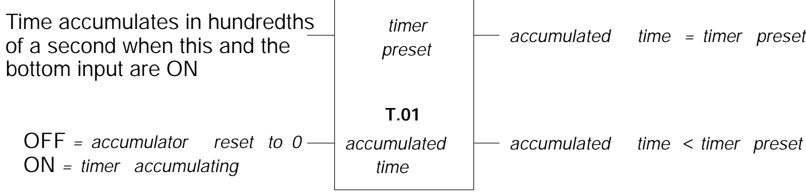
Size
Two nodes high

PLC Compatibility
Standard in all PLC types

Opcode
17 hex

5.5.2 Representation in Ladder Logic

Block Structure



Inputs
T.01 has two input controls. The input to the top node initiates the timer operation. The input to the bottom node is ON while the timer is accumulating. If it goes OFF, the *accumulated time* is reset to zero.

Outputs
T.01 can produce one of two possible outputs. The output from the top node passes power when the *accumulated time* reaches the specified *timer preset* value. The output from the bottom node passes power if the *accumulated time* value drops below the *timer preset* value.

Top Node Content

The *timer preset* in the top node is a value that specifies how many hundredth-of-a-second increments the timer can accumulate. This value can be:

- Displayed explicitly as an integer
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

If the *timer preset* is entered as an integer, the range of allowable values is a function of the type of PLC in use:

PLC Type	Allowable Integer Range
Any PLC with a 16 Bit CPU	1 ... 999
E685/785 PLCs, L785 PLCs, and Quantum Series PLCs	1 ... 65, 535
Other PLCs with 24-bit CPUs	1 ... 9,999

Bottom Node Content

The 4*x* register entered in the bottom node stores the *accumulated time* count in hundredth-of-a-second increments.

5.6 T1MS Timer

The T1MS instruction measures time in ms intervals. It can be used for timing an event or creating a delay. T1MS has two control inputs (to the top and middle nodes) and can produce one of two possible outputs. An output passing power from the top or middle node indicates a timer error.

5.6.1 Characteristics

Size
Three nodes high

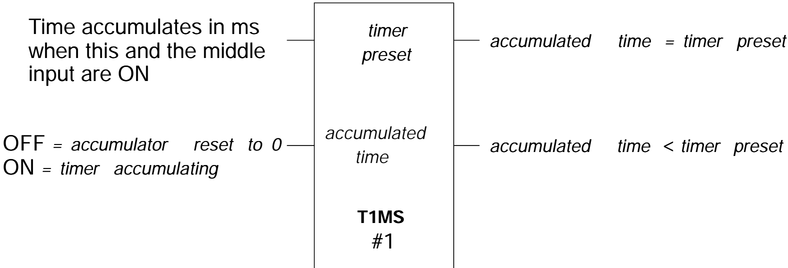
PLC Compatibility

- Standard in Micro PLC models and the Quantum CPU 424 02 PLC
- Not available in all other PLC types

Opcode
1E hex

5.6.2 Representation in Ladder Logic

Block Structure



Inputs
T1MS has two input controls. The input to the top node initiates the timer operation. The input to the middle node is ON while the timer is accumulating. If it goes OFF, the *accumulated time* is reset to zero.

Outputs
T1MS can produce one of two possible outputs. The output from the top node passes power when the *accumulated time* reaches the specified *timer preset* value. The output from the middle node passes power if the *accumulated time* value drops below the *timer preset* value.

Top Node Content
The *timer preset* stored in the top node is a value that specifies how many increments the timer can accumulate. It can be:

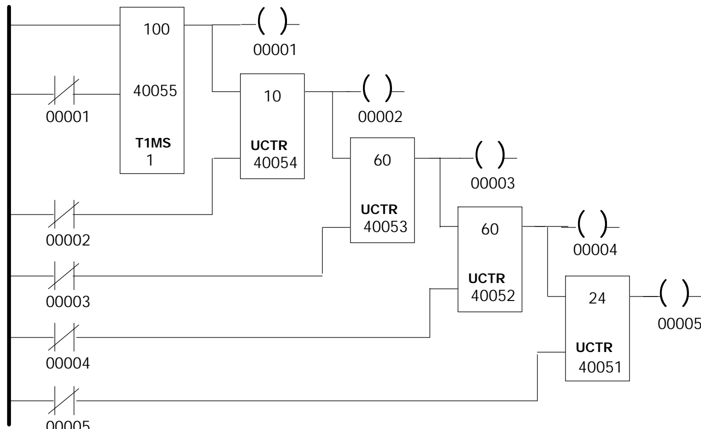
- ☐ Displayed explicitly as an integer in the range 1 ... 999
- ☐ Stored in a 3*x* input register
- ☐ Stored in a 4*x* holding register

Middle Node Content
The 4*x* register entered in the middle node stores the *accumulated time* in ms increments.

Bottom Node Content
The bottom node always contains a constant value of #1.

5.6.3 A Millisecond Timer Example

Here is the ladder logic for a real-time clock with millisecond accuracy. This example can be programmed only for a Micro PLC:



The T1MS instruction is programmed to pass power at 100 ms intervals; it is followed by a cascade of four up-counters that store the the time respectively in hundredth-of-a-second units, tenth-of-a-second units, one- second units, one-minute units, and one-hour units.

When logic solving begins, the accumulated time value begins incrementing in register 40055 of the T1MS block. After 100 one-ms increments, the top output passes power and energizes coil 00001. At this point, the value in register 40055 in the timer is reset to 0. The accumulated count value in register 40054 in the first UCTR block increments by 1, indicating that 100 ms have passed. Because the accumulated time count in T1MS no longer equals the timer preset, the timer begins to re-accumulate time in ms.

When the accumulated count in register 40054 of the first UCTR instruction increments to 10, the top output from that instruction block passes power and energizes coil 00002. The value in register 40054 then resets to 0, and the accumulated count in register 40053 of the second UCTR block increments by 1.

As the times accumulate in each counter, the time of day can be read in five holding registers as follows:

Register	Unit of Time	Valid Range
40055	thousandths-of-a-second	0 ... 100
40054	tenths-of-a-second	0 ... 10
40053	seconds	0 ... 60
40052	minutes	0 ... 60
40051	hours	0 ... 24

Chapter 6
Integer and 16-bit Math Instructions

Two groups of instructions that support basic math operations are described in this chapter. The first group comprises four integer-based instructions—ADD, SUB, MUL, and DIV—which come standard in all PLCs and support straightforward unsigned addition, subtraction, multiplication, and division of register or constant values.

The second group, which is available only in certain PLCs, contains five comparable instructions—AD16, SU16, TEST, MU16, and DV16—that support signed and unsigned 16-bit math calculations and comparisons.

Three additional instructions—ITOF, FTOI, and BCD—are provided to convert the formats of numerical values (from integer to floating point, floating point to integer, binary to BCD, and BCD to binary). Conversion operations are useful in expanded math (see Chapter 7) and process control instructions (see Chapter 21).

6.1 ADD

The ADD instruction adds unsigned *value 1* (its top node) to unsigned *value 2* (its middle node) and stores the *sum* in a holding register in the bottom node.

6.1.1 Characteristics

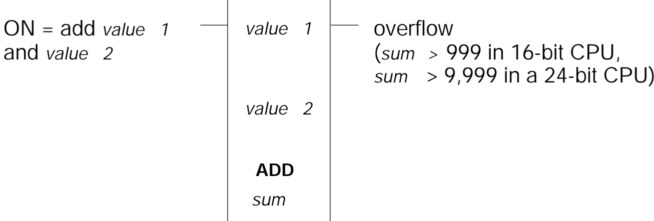
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
18 hex

6.1.2 Representation in Ladder Logic

Block Structure



Input
ADD has one control input (to the top node), which initiates the operation when it is ON.

Output
ADD can produce one possible output. The output passing power from the top node indicates an overflow in the value of the *sum* .

Top and Middle Node Content

The top and middle nodes contain *value 1* and *value 2*, respectively. The value in each node may be:

- Displayed explicitly as an integer in the range 1 ... 999 in a 16 bit CPU or 1 ... 9,999 in a 24 bit CPU
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

Bottom Node Content

The 4*x* register entered in the bottom node stores the *sum* of the ADD operation.

6.2 SUB

The SUB instruction performs an absolute subtraction of *value 1* *value 2* (top node middle node) and stores the *difference* in a holding register in the bottom node.

6.2.1 Characteristics

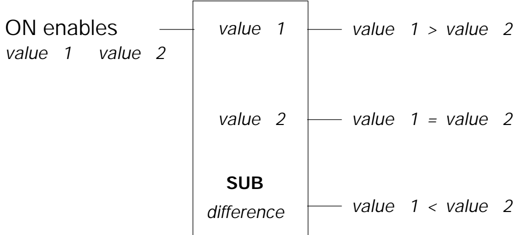
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
19 hex

6.2.2 Representation in Ladder Logic

Block Structure



Input
SUB has one control input (to the top node), which initiates the operation when it is ON.

Outputs
SUB produces one of three possible outputs. The state of the outputs indicates the result of a magnitude comparison between *value 1* and *value 2*. SUB is often used as a comparator where the state of the outputs identifies whether *value 1* is greater than, equal to, or less than *value 2*.

Top and Middle Node Content

The top and middle nodes contain *value 1* and *value 2*, respectively. The value in each node may be:

- ☐ Displayed explicitly as an integer in the range 1 ... 999 in a 16 bit CPU or 1 ... 9,999 in a 24 bit CPU
- ☐ Stored in a 3*x* input register
- ☐ Stored in a 4*x* holding register

Bottom Node Content

The 4*x* register entered in the bottom node stores the absolute (unsigned) *difference* between *value 1* and *value 2*.

6.3 MUL

The MUL instruction multiplies unsigned *value 1* (its top node) by unsigned *value 2* (its middle node) and stores the *product* in two contiguous holding registers in the bottom node.

6.3.1 Characteristics

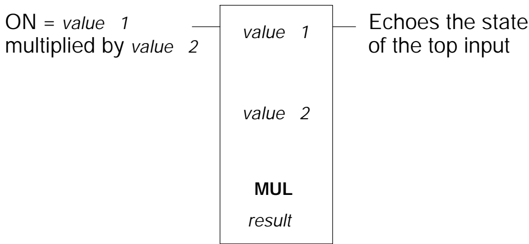
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
1A hex

6.3.2 Representation in Ladder Logic

Block Structure



Input
MUL has one control input (to the top node), which initiates the operation when it is ON.

Output
MUL produces an output from the top node, which echoes the state of the top input.

Top and Middle Node Content

The top and middle nodes contain *value 1* and *value 2*, respectively. The value in each node may be:

- Displayed explicitly as an integer in the range 1 ... 999 in a 16 bit CPU or 1 ... 9,999 in a 24 bit CPU
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

Bottom Node Content

The 4*x* register entered in the bottom node is the first of two contiguous holding registers where the *product* is stored. The high-order digits in the *product* are stored in the displayed register, and the low-order digits are stored in the implied register.

For example, if *value 1* = 8,000 and *value 2* = 2, the product is 16,000. The displayed register contains the value 0001 (the high-order half of the *product*), and implied register contains the value 6,000 (the low-order half of the *product*).

6.4 DIV

The DIV instruction divides unsigned *value 1* (its top node) by unsigned *value 2* (its middle node) and posts the *quotient* and *remainder* in two contiguous holding registers in the bottom node.

6.4.1 Characteristics

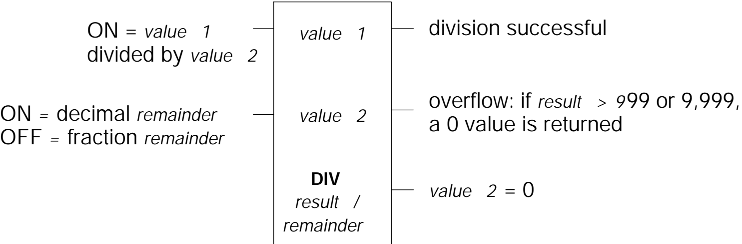
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
1B hex

6.4.2 Representation in Ladder Logic

Block Structure



Inputs
DIV has two control inputs (to the top and middle nodes). The top input initiates the operation when it is ON.

The state of the input to the middle node indicates whether the *remainder* will be expressed as a decimal or as a fraction. For example, if *value 1* = 8 and *value 2* = 3, the decimal *remainder* (middle input ON) is 6666; the fractional *remainder* (middle input OFF) is 2.

Outputs

DIV can produce one of three possible outputs. Power passed at the top output indicates the successful completion of a DIV operation. Power passed from the middle or bottom output indicates an error in the operation.

Top Node Content

The top node contains *value 1*, which can be:

- ☐ Displayed explicitly as an integer in the range 1 ... 999 in a 16-bit CPU or 1 ... 9,999 in a 24-bit CPU
- ☐ Stored in two contiguous input registers, 3*x* for the high-order half of the value and 3*x* + 1 for the low-order half
- ☐ Stored in two contiguous holding registers, 4*x* for the high-order half of the value and 4*x* + 1 for the low-order half

Middle Node Content

The middle node contains *value 2*, which can be:

- ☐ Displayed explicitly as an integer in the range 1 ... 999 in a 16-bit CPU or 1 ... 9,999 in a 24-bit CPU
- ☐ Stored in one 3*x* input register
- ☐ Stored in one 4*x* holding register

Bottom Node Content

The 4*x* register entered in the bottom node is the first of two contiguous holding registers. The *result* of the division is posted in the displayed register. The *remainder* is posted in the implied register as either a decimal or a fraction (depending on the state of the middle input).

6.5 AD16

The AD16 instruction performs signed or unsigned 16-bit addition on *value 1* (its top node) and *value 2* (its middle node), then posts the *sum* in a 4*x* holding register in the bottom node.

6.5.1 Characteristics

Size
Three nodes high

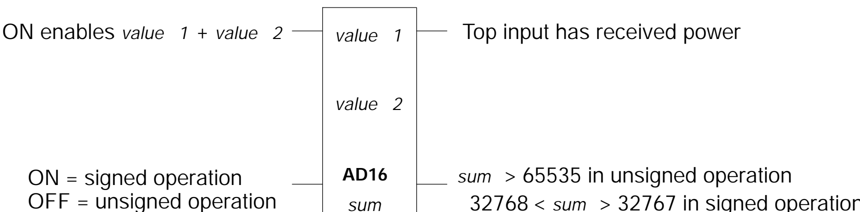
PLC Compatibility

- Standard in E984-685 and E984-785 PLCs and in the Quantum Automation Series PLCs
- Not available in all other PLC types

Opcode
31 hex

6.5.2 Representation in Ladder Logic

Block Structure



Inputs
AD16 has two control inputs (to the top and bottom nodes). The top input initiates the operation when it is ON. The state of the bottom input indicates whether the addition will be a signed or unsigned operation.

Outputs
AD16 can produce one of two possible outputs. Power passed at the top output indicates the successful completion of a AD16 operation. Power passed from the bottom output indicates an overflow in the *sum* .

Top and Middle Node Content

The top and middle nodes contain *value 1* and *value 2*, respectively. The value in each node may be:

- Displayed explicitly as an integer in the range 1 ... 65,535
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

Note: In order to support constant values of up to 65,535 in the top node using Modsoft, you must set the **/6** switch. This switch setting enables 6-bit references.

Bottom Node Content

The 4*x* register entered in the bottom node stores the *sum* of the 16-bit addition.

6.6 SU16

The SU16 instruction performs a signed or unsigned 16-bit subtraction (*value 1* *value 2*) on the top and middle node values, then posts the signed or unsigned *difference* in a 4*x* holding register in the bottom node.

6.6.1 Characteristics

Size
Three nodes high

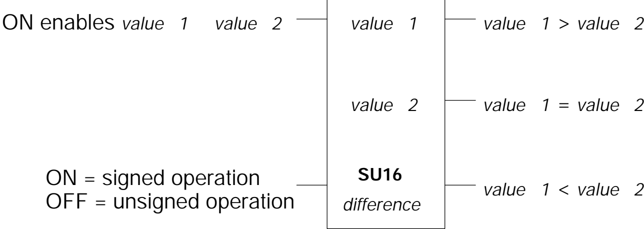
PLC Compatibility

- Standard in E984-685 and E984-785 PLCs and in the Quantum Automation Series PLCs
- Not available in all other PLC types

Opcode
32 hex

6.6.2 Representation in Ladder Logic

Block Structure



Inputs
SU16 has two control inputs (to the top and bottom nodes). The top input initiates the operation when it is ON. The state of the bottom input indicates whether the addition will be a signed or unsigned operation.

Outputs
SU16 produces one of three possible outputs. The state of the outputs indicates the relationship between *value 1* and *value 2*.

Top and Middle Node Content

The top and middle nodes contain *value 1* and *value 2*, respectively. The value in each node may be:

- Displayed explicitly as an integer in the range 1 ... 65,535
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

Note: In order to support constant values of up to 65,535 in the top node using Modsoft, you must set the **/6** switch. This switch setting enables 6-bit references.

Bottom Node Content

The 4*x* register entered in the bottom node contains the signed or unsigned *difference* between *value 1* and *value 2*.

6.7 TEST

The TEST instruction compares the signed or unsigned size of the 16-bit values in the top and middle nodes and describes the relationship via the states of the block outputs.

6.7.1 Characteristics

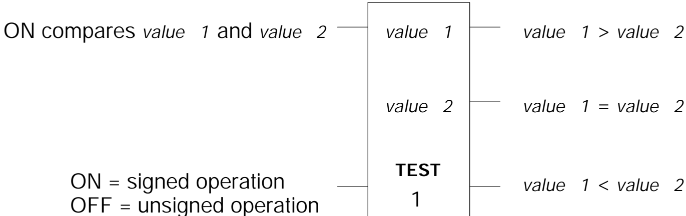
Size
Three nodes high

PLC Compatibility
- ☐ Standard in E984-685 and E984-785 PLCs and in the Quantum Automation Series PLCs
- ☐ Not available in all other PLC types

Opcode
35 hex

6.7.2 Representation in Ladder Logic

Block Structure



Inputs
TEST has two control inputs (to the top and bottom nodes). The top input initiates the operation when it is ON. The state of the bottom input indicates whether the comparison will be a signed or unsigned operation.

Outputs
TEST produces one of three possible outputs. The state of the outputs indicates the relationship between *value 1* and *value 2*.

Top and Middle Node Content

The top and middle nodes contain *value 1* and *value 2*, respectively. The value in each node may be:

- Displayed explicitly as an integer in the range 1 ... 65,535
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

☞ **Note:** In order to support constant values of up to 65,535 in the top node using Modsoft, you must set the **/6** switch. This switch setting enables 6-bit references.

Bottom Node Content

The bottom node contains constant value of 1.

6.8 MU16

The MU16 instruction performs signed or unsigned multiplication on the 16-bit values in the top and middle nodes, then posts the *product* in two contiguous holding registers in the bottom node.

6.8.1 Characteristics

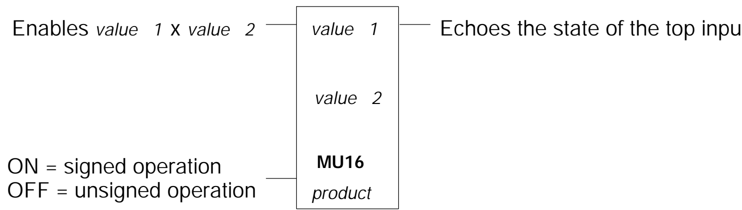
Size
Three nodes high

PLC Compatibility
- ☐ Standard in E984-685 and E984-785 PLCs and in the Quantum Automation Series PLCs
- ☐ Not available in all other PLC types

Opcode
33 hex

6.8.2 Representation in Ladder Logic

Block Structure



Inputs
MU16 has two control inputs (to the top and bottom nodes). The top input initiates the operation when it is ON. The state of the bottom input indicates whether the multiplication will be a signed or unsigned operation.

Output
MU16 produces one output from the top node, which echoes the state of the top input.

Top and Middle Node Content

The top and middle nodes contain *value 1* and *value 2*, respectively. The value in each node may be:

- Displayed explicitly as an integer in the range 1 ... 65,535
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

Note: To enter a constant value in the top node, start with a # symbol followed immediately by the constant—e.g., #65535.

Bottom Node Content

The bottom node contains the first of two contiguous 4*x* holding registers, where the *product* is stored. The displayed (4*x*) register contains half of the product and the implied (4*x* + 1) register contains the other half. The product can have an unsigned value in the range 1 ... 4,294,967,295.

If the product is a value > 65,535, it can be displayed only in long decimal format. If you are using panel software that does not support long decimal format, the value of the product will not be seen. (Modsoft does support long decimal format.)

In early versions of the E984-685 and E984-785 System Executives, the high-order half of the *product* was stored in the displayed register in the bottom node. In later Exec versions, the low-order half of the *product* is stored in the displayed register. This new format makes the instruction work together with the host interface, MSL, PCFL, and custom loadable functions without conversion; the new format follows Intel double-word conventions.

Caution: Before you upgrade your PLC Executive in Flash, you need to know what revision level of the Executive was used to create your ladder logic program. If you have created the program with one of the early versions of the Executive described above and then load a later version, the logic scan will read the product registers in a different order, and, depending on the way logic is used, it may misinterpret the product value.

Note: For E984-685 PLCs, Executive firmware revisions 2.10 and lower use the displayed register in the bottom node to store the high-order half of the *product* and the implied register to store the low-order half. Firmware revisions 2.11 and later store the low-order half of the *product* in the displayed register and the high-order half in the implied register.

For E984-785 PLCs, Executive firmware revisions 1.10 and lower use the displayed register in the bottom node to store the high-order half of the *product* and the implied register to store the low-order half. Firmware revisions 1.11 and later store the low-order half of the *product* in the displayed register and the high-order half in the implied register.

6.9 DV16

The DV16 instruction performs a signed or unsigned division on the 16-bit values in the top and middle nodes (*value 1* / *value 2*), then posts the quotient and remainder in two contiguous 4*x* holding registers in the bottom node.

6.9.1 Characteristics

Size
Three nodes high

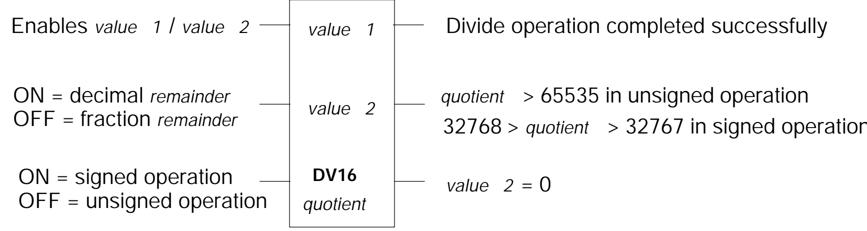
PLC Compatibility

- Standard in E984-685 and E984-785 PLCs and in the Quantum Automation Series PLCs
- Not available in all other PLC types

Opcode
34 hex

6.9.2 Representation in Ladder Logic

Block Structure



Inputs
DV16 has three control inputs. The top input initiates the operation when it is ON. The state of the input to the middle node indicates whether the *remainder* will be expressed as a decimal or as a fraction. For example, if *value 1* = 8 and *value 2* = 3, the decimal *remainder* (middle input ON) is 6666; the fractional *remainder* (middle input OFF) is 2.

The state of the bottom input indicates whether the addition will be a signed or unsigned operation.

Outputs

DV16 can produce one of three possible outputs. Power passed at the top output indicates the successful completion of a DIV operation; power passed from at the middle or bottom output indicates an error in the operation.

Top Node Content

The top node contains *value 1*, which may be:

- Displayed explicitly as an integer in the range 1 ... 65,535
- Stored in two contiguous 3*x* input registers
- Stored in two contiguous 4*x* holding registers

When the values in the top and middle nodes are displayed via registers, they can have unsigned values in the range 1 ... 4,294,967,295. If a value > 65,535, it can be displayed only in long decimal format. If you are using panel software that does not support long decimal format, the value of the product will not be seen. (Modsoft does support long decimal format.)

In some versions of the E984-685 and E984-785 System Executives, the high-order half of *value 1* is stored in the displayed register in the top node; in other Exec versions, the low-order half of *value 1* is stored in the displayed register. This new format makes the instruction work together with the host interface, MSL, PCFL, and custom loadable functions without conversion; the new format follows Intel double-word conventions.

Caution: Before you upgrade your PLC Executive in Flash, you need to know what revision level of the Executive was used to create your ladder logic program. If you have created the program with one of the early versions of the Executive described above and then load a later version, the logic scan will read the *value 1* registers in a different order, and, depending on the way the logic is used, it may misinterpret the dividend to be used in the divide operation.

Note: For E984-685 PLCs, Executive firmware revisions 2.10 and lower store the high-order half of *value 1* in the displayed register in the top node and the low-order half of *value 1* in the implied register. Firmware revisions 2.11 and later store the low-order half of *value 1* in the displayed register and the high-order half of in the implied register.

For E984-785 PLCs, Executive firmware revisions 1.10 and lower store the high-order half of *value 1* in the displayed register in the top node and the low-order half of in the implied register. Firmware revisions 1.11 and later store the low-order half of *value 1* in the displayed register and the high-order half of in the implied register.

Middle Node Content

The middle node contains *value 2*, which may be:

- Displayed explicitly as an integer in the range 1 ... 65,535
- Stored in one 3*x* input register
- Stored in one 4*x* holding register

Note: To enter an integer value in the middle node, start with a # symbol followed immediately by the integer—e.g., **#65535**.

Bottom Node Content

The 4*x* register entered in the bottom node is the first of two contiguous holding registers. The *quotient* of the DV16 operation is posted in the displayed register. The *remainder* of the divide operation is posted in the implied register as either a decimal or a fraction (depending on the state of the middle input).

6.10 IT OF

The ITOF instruction performs the conversion of a signed or unsigned *integer value* (its top node) to a floating point (FP) value, and stores the *FP value* in two contiguous 4*x* registers in the middle node.

6.10.1 Characteristics

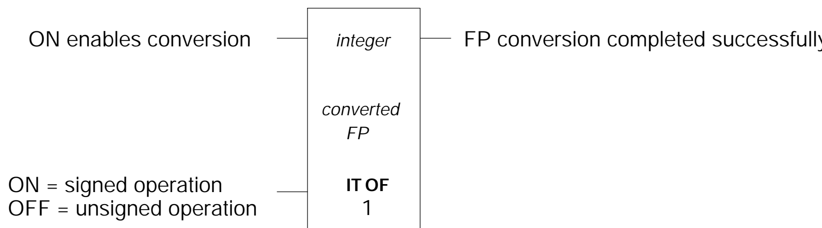
Size
Three nodes high

PLC Compatibility
- Standard in E984-685 and E984-785 PLCs and in the Quantum Automation Series PLCs
- Not available in all other PLC types

Opcode
36 hex

6.10.2 Representation in Ladder Logic

Block Structure



Inputs
ITOF has two control inputs (to its top and bottom nodes). The top input initiates the operation when it is ON. The state of the bottom input indicates whether the conversion will be a signed or unsigned operation.

Output
ITOF produces one output from the top node upon successful completion of the conversion.

Top Node Content

The top node contains the *integer value*. It may be:

- Displayed explicitly as an integer in the range 1 ... 65,535
- Stored in a 3x input register
- Stored in a 4*x* holding register

Note: In order to support constant values of up to 65,535 in the top node using Modsoft, you must set the **/6** switch. This switch setting enables 6-digit references.

Middle Node Content

The 4*x* register entered in the middle node is the first of two contiguous holding registers where the converted *FP value* is stored.

Bottom Node Content

The bottom node contains a constant value of 1.

6.1 1 FT OI

The FTOI instruction performs the conversion of a *floating value* to a signed or unsigned integer (stored in two contiguous registers in the top node), then stores the *converted integer value* in a 4*x* register in the middle node.

6.1 1.1 Characteristics

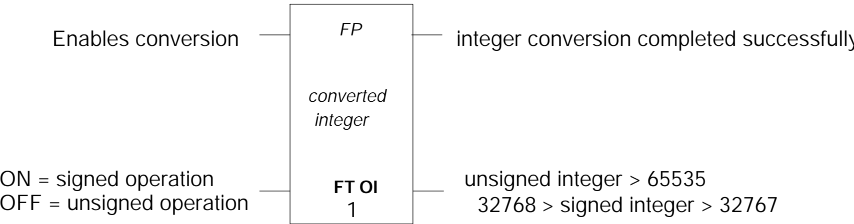
Size
Three nodes high

PLC Compatibility
- ☐ Standard in E984-685 and E984-785 PLCs and in the Quantum Automation Series PLCs
- ☐ Not available in all other PLC types

Opcode
37 hex

6.1 1.2 Representation in Ladder Logic

Block Structure



Inputs
FTOI has two control inputs (to its top and bottom nodes). The top input initiates the operation when it is ON. The state of the bottom input indicates whether the conversion will be a signed or unsigned operation.

Outputs
FTOI produces two possible outputs. The output from the top node goes ON upon successful completion of the conversion. If the output from the

bottom node passes power, the value of the converted integer value is out of range.

Top Node Content

The 4*x* register entered in the top node is the first of two contiguous holding registers where the *floating point* value is stored.

Middle Node Content

The 4*x* register entered in the middle node is where the *converted integer* value is posted.

Bottom Node Content

The bottom node contains a constant value of 1.

6.12 BCD

The BCD instruction can be used to convert a binary value to a binary coded decimal (BCD) value or a BCD value to a binary value. The type of conversion to be performed is controlled by the state of the bottom input.

6.12.1 Characteristics

Size
Three nodes high

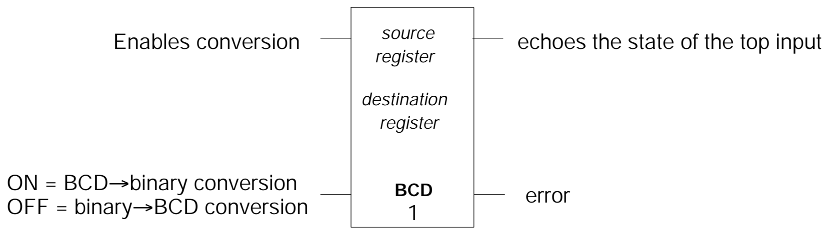
PLC Compatibility

- Standard in the Quantum Automation Series PLCs
- Not available in all other PLC types

Opcode
53 hex

6.12.2 Representation in Ladder Logic

Block Structure



Inputs
BCD has two control inputs (to its top and bottom nodes). The top input initiates the operation when it is ON. The state of the bottom input indicates the type of conversion to be performed—when ON, a BCD-to-binary format conversion is performed, and when OFF, a binary-to-BCD format conversion is performed.

Outputs
FTOI produces two possible outputs. The output from the top node echoes the state of the top input. The output from the bottom node will pass power if an error has been detected in the conversion operation.

Top Node Content

The 3*x* or 4*x* register entered in the top node is the *source* register where the numerical value to be converted is stored.

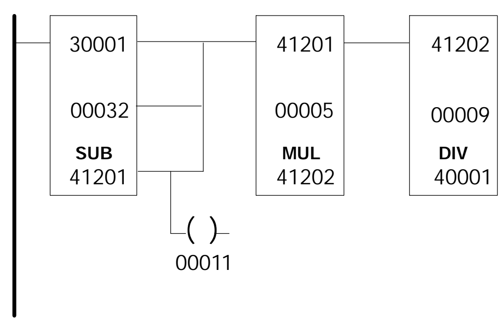
Middle Node Content

The 4*x* register entered in the middle node is there *destination register* where the converted numerical value is posted.

Bottom Node Content

The bottom node contains a constant value of 1.

6.13 A Fahrenheit-to-Centigrade Conversion Example



Note: The vertical short to coil 00011 must be to the left of the vertical shorts linking the three outputs from the SUB block.

We want to implement the formula

$$^{\circ}C = (^{\circ}F \quad 32) \text{ x } {}^{5}/_{9}$$

When the top input of the SUB instruction receives power, the number 32 is subtracted from the value in register 30001, which represents some number of degrees Fahrenheit. The *result* is placed in register 41201.

The top input to the MUL instruction then receives power, whether the SUB *result* is positive, negative, or 0. If the SUB *result* is negative, coil 00011 is energized to indicate a negative value.

The value in register 41201 is then multiplied by 5, and the *result* is placed in registers 41202 and 41203. The top input of the DIV instruction is then energized, and the value in registers 41202 and 41203 is divided by 9. The *result*, which is the temperature conversion in degrees Centigrade, is placed in register 40001.

Chapter 7
Enhanced Math Capabilities

Many Modicon PLCs have an Enhanced Executive that supports the EMTH (extended math) instruction. This instruction accesses a library of double-precision math, square root and logarithm calculations, and floating point (FP) arithmetic functions.

Some PLCs—such as the 984A, 984B, and 984X Chassis Mount PLCs—which do not support Enhanced Executives, have two loadable options available for extended math instructions. The MATH and DMTH instructions provide double precision math, square root, process square root, log, and antilog functions comparable to those available to other PLCs in the EMTH library.

7.1 Capabilities of the EMTH Instruction

The EMTH instruction allows you to select from a library of 38 extended math functions. Each of the functions has an alphabetical indicator which can be selected from a pulldown menu in your panel software and which appears in the bottom node. EMTH control inputs and outputs are function-dependent. See the chart on pages 105 ... 106 for an overview; for detail, see the various functional description given throughout this chapter.

7.1.1 Characteristics

Size
Three nodes high

PLC Compatibility

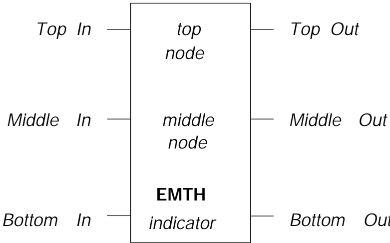
- Standard in all PLC models except the 984A, 984B, or 984X Chassis Mount PLCs and the 110CPU311 and 110CPU411 Micro PLCs

- Not available in the above-listed PLC models

Opcode
7F hex

7.1.2 Representation in Ladder Logic

Generic Block Structure



Top Node Content
The top node requires two consecutive registers, usually 4*x* holding registers but, in the integer math cases, either 4*x* or 3*x* registers.

Middle Node Content

The middle node requires either two, four, or six consecutive registers, depending on the function you are implementing. Use $4x$ holding registers.

Bottom Node Content

An alphabetical indicator appears in the bottom node, identifying the EMTH function you have chosen from the library.

Inputs and Outputs

The implementation of inputs to and outputs from the block depends on the EMTH function you select:

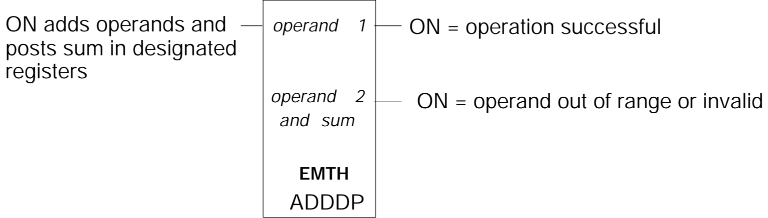
EMTH Function	Indicator	Active Inputs	Active Outputs
Double Precision Math			
Addition	ADDDP	Top only	Top, Middle
Subtraction	SUBDP	Top only	Top, Middle, Bottom
Multiplication	MULDP	Top only	Top, Middle
Division	DIVDP	Top, Middle	Top, Middle, Bottom
Integer Math			
Square root	SQRT	Top only	Top, Middle
Process square root	SQRTP	Top only	Top, Middle
Logarithm	LOG	Top only	Top, Middle
Antilogarithm	ANLOG	Top only	Top, Middle
Floating Point Math			
Integer-to-FP conversion	CNVIF	Top only	Top only
Integer + FP	ADDIF	Top only	Top only
Integer FP	SUBIF	Top only	Top only
Integer x FP	MULIF	Top only	Top only
Integer ÷ FP	DIVIF	Top only	Top only
FP Integer	SUBFI	Top only	Top only
FP ÷ Integer	DIVFI	Top only	Top only
Integer-FP comparison	CMPIF	Top only	Top only
FP-to-Integer conversion	CNVFI	Top only	Top, Bottom
Addition	ADDFP	Top only	Top only
Subtraction	SUBFP	Top only	Top only
Multiplication	MULFP	Top only	Top only
Division	DIVFP	Top only	Top only
Comparison	CMPFP	Top only	Top, Middle, Bottom
Square root	SQRFP	Top only	Top only
Change sign	CHSIN	Top only	Top only
Load Value of π	PI	Top only	Top only
Sine in radians	SINE	Top only	Top only

Cosine in radians	COS	Top only	Top only
Tangent in radians	TAN	Top only	Top only
Arcsine in radians	ARSIN	Top only	Top only
Arccosine in radians	ARCOS	Top only	Top only
Arctangent in radians	ARTAN	Top only	Top only
Radians to degrees	CNVRD	Top only	Top only
Degrees to radians	CNVDR	Top only	Top only
FP to an integer power	POW	Top only	Top only
Exponential function	EXP	Top only	Top only
Natural log	LNFP	Top only	Top only
Common log	LOGFP	Top only	Top only
Report errors	ERLOG	Top only	Top, Middle

7.2 Double Precision EMTH Functions

7.2.1 Double Precision Addition

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second 4*x* register is implied. *Operand 1* is stored here.

Each register holds a value in the range 0000 ... 9999, for a combined double precision value in the range 0 ... 99,999,999. The high-order half of *operand 1* is stored in the displayed register, and the low-order half is stored in the implied register.

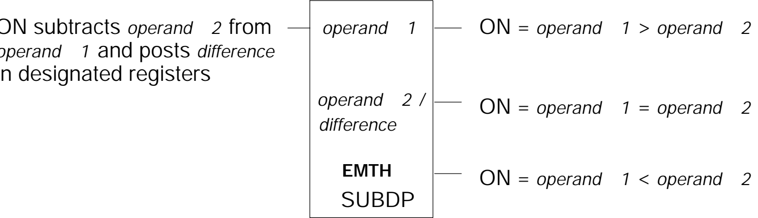
Middle Node Content

The first of six contiguous 4*x* registers is entered in the middle node. The remaining five registers are implied:

- The displayed register and the first implied register store the high-order and low-order halves of *operand 2*, respectively, for a combined double precision value in the range 0 ... 99,999,999
- The value stored in the second implied register indicates whether an overflow condition exists (a value of 1 = overflow)
- The third and fourth implied registers store the high-order and low-order halves of the double precision sum, respectively
- The fifth implied register is not used in the calculation but must exist in state RAM

7.2.2 Double Precision Subtraction

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second 4*x* register is implied. *Operand 1* is stored here.

Each register holds a value in the range 0000 ... 9999, for a combined double precision value in the range 0 ... 99,999,999. The low-order half of *operand 1* is stored in the displayed register, and the high-order half is stored in the implied register.

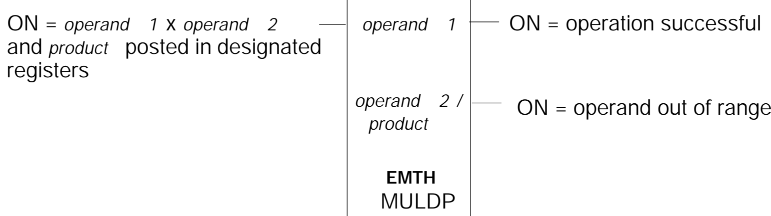
Middle Node Content

The first of six contiguous 4*x* registers is entered in the middle node. The remaining five registers are implied:

- The displayed register and the first implied register store the high-order and low-order halves of *operand 2*, respectively, for a combined double precision value in the range 0 ... 99,999,999
- The second and third implied registers store the high-order and low-order halves, respectively, of the absolute *difference* in double precision format
- The value stored in the fourth implied register indicates whether or not the operands are in the valid range (1 = out of range and 0 = in range)
- The fifth implied register is not used in this calculation but must exist in state RAM

7.2.3 Double Precision Multiplication

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second 4*x* register is implied. *Operand 1* is stored here.

The second 4*x* register is implied. Each register holds a value in the range 0000 ... 9999, for a combined double precision value in the range 0 ... 99,999,999. The high-order half of *operand 1* is stored in the displayed register, and the low-order half is stored in the implied register.

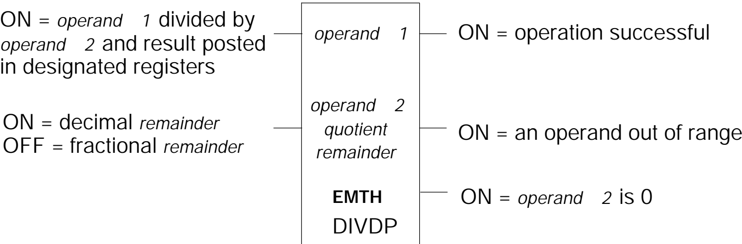
Middle Node Content

The first of six contiguous 4*x* registers is entered in the middle node. The remaining five registers are implied:

- The displayed register and the first implied register store the high-order and low-order halves of *operand 2*, respectively, for a combined double precision value in the range 0 ... 99,999,999
- The last four implied registers store the double precision *product* in the range 0 ... 9,999,999,999,999,999

7.2.4 Double Precision Division

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. *Operand 1* is stored here.

Each register holds a value in the range 0000 ... 9999, for a combined double precision value in the range 0 ... 99,999,999. The high-order half of *operand 1* is stored in the displayed register, and the low-order half is stored in the implied register.

Middle Node Content

The first of six contiguous 4*x* registers is entered in the middle node. The remaining five registers are implied:

- The displayed register and the first implied register store the high-order and low-order halves of *operand 2*, respectively, for a combined double precision value in the range 0 ... 99,999,999

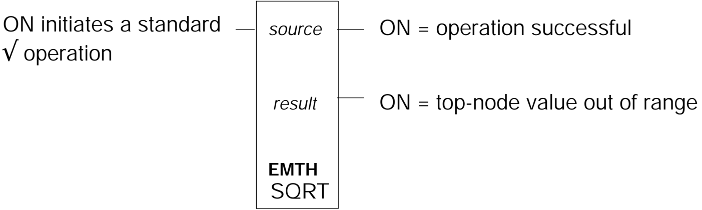
Note: Since division by 0 is illegal, a 0 value causes an error—an error trapping routine sets the remaining middle-node registers to 0000 and turns the bottom output ON.

- The second and third implied registers store an eight-digit *quotient*
- The fourth and fifth implied registers store the *remainder* —if the *remainder* is expressed as a fraction, it is eight digits long and both registers are used; if the remainder is expressed as a decimal, it is four digits long and only the fourth implied register is used

7.3 Integer EMTH Functions

7.3.1 Square Root

Block Structure



Top Node Content

The first of two contiguous 3*x* or 4*x* registers is entered in the top node. The second register is implied. The *source* value—i.e., the value for which the square root will be derived—is stored here.

If you specify a 4*x* register, the *source* value may be in the range 0 ... 99,999,99. The low-order half of the value is stored in the implied register, and the high-order half is stored in the displayed register.

If you specify a 3*x* register, the *source* value may be in the range 0 ... 9,999. The square root calculation is done on only the value in the displayed register; the implied register is required but not used.

Middle Node Content

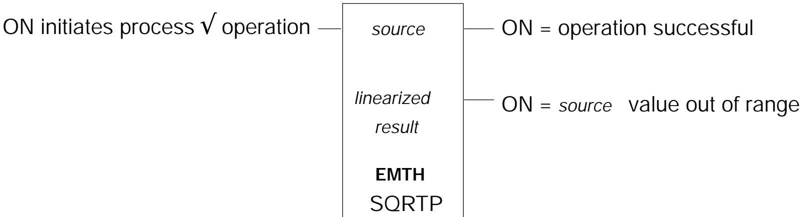
Enter the first of two contiguous 4*x* registers in the middle node. The second register is implied. The *result* of the standard square root operation is stored here.

The *result* is stored in the fixed-decimal format: **1234. 5600.** where the displayed register stores the four-digit value to the left of the first decimal point and the implied register stores the four-digit value to the right of the first decimal point. Numbers after the second decimal point are truncated; no round-off calculations are performed.

7.3.2 Process Square Root

The process square root function tailors the standard square root function for closed loop analog control applications. It takes the result of the standard square root result, multiplies it by 63.9922—the square root of 4095—and stores that *linearized result* in the middle-node registers.

Block Structure



7.3.2.1 Top Node Content

The first of two contiguous 3*x* or 4*x* registers is entered in the top node. The second register is implied. The *source* value—i.e., the value for which the square root will be derived—is stored in these two registers. In order to generate values that have meaning, the *source* value must not exceed 4095. In a 4*x* register group the *source* value will therefore be stored in the implied register, and in a 3*x* register group the *source* value will be stored in the displayed register.

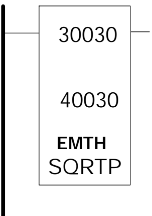
7.3.2.2 Middle Node Content

The first of two contiguous 4*x* registers is entered in the middle node. The second register is implied. The *linearized result* of the process square root operation is stored here.

The *result* is stored in the fixed-decimal format: **1234. 5600.** where the displayed register stores the four-digit value to the left of the first decimal point and the implied register stores the four-digit value to the right of the first decimal point. Numbers after the second decimal point are truncated; no round-off calculations are performed.

How the Process Square Root Function Works

Look at the instruction example below for a quick overview of how the process square root is calculated.



Suppose a *source* value of 2000 is stored in register 30030 of EMTH function 6. First, a standard square root operation is performed:

$$\sqrt{2000} = 0044.72$$

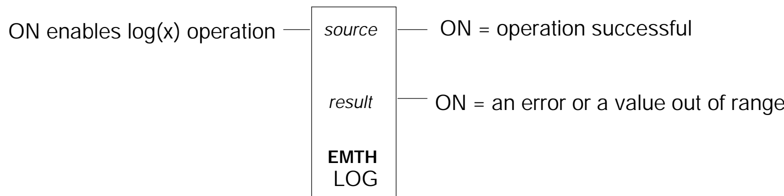
which is then multiplied by 63.9922, yielding a *linearized result* of 2861.63. The *linearized result* is placed in registers 40030 and 40031 in the middle node of the EMTH instruction:

- Register 40030 stores the high-order half (2861)
- Register 40031 stores the low-order half (6300)

The process square root is often used to *linearize* signals from differential pressure flow transmitters so that they may be used as inputs in closed loop control operations.

7.3.3 Base 10 Logarithm

Block Structure



Top Node Content

The first of two contiguous 3*x* or 4*x* registers is entered in the top node. The second register is implied. The *source* value upon which the log calculation will be performed is stored in these registers.

If you specify a 4*x* register, the *source* value may be in the range 0 ... 99,999,99. The low-order half of the value is stored in the implied register, and the high-order half is stored in the displayed register.

If you specify a 3*x* register, the *source* value may be in the range 0 ... 9,999. The log calculation is done on only the value in the displayed register; the implied register is required but not used.

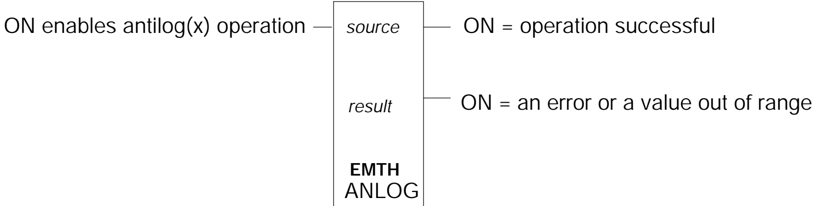
Middle Node Content

The middle node contains a single 4*x* holding register where the *result* of the base 10 log calculation is posted. The *result* is expressed in the fixed decimal format **1. 234** , and is truncated after the third decimal position.

The largest *result* that can be calculated is 7.999, which would be posted in the middle register as 7999.

7.3.4 Base 10 Antilogarithm

Block Structure



Top Node Content

The top node is a single 4*x* holding register or 3*x* input register. The *source* value—i.e., the value on which the antilog calculation will be performed—is stored here in the fixed decimal format **1. 234** . It must be in the range 0 ... 7999, representing a *source* value up to a maximum of 7.999.

Middle Node Content

The first of two contiguous 4*x* registers is entered in the middle node. The second register is implied. The *result* of the antilog calculation is posted here in the fixed decimal format **12345678** .

The most significant bits are posted in the displayed register, and the least significant bits are posted in the implied register. The largest antilog value that can be calculated is 99770006 (9977 posted in the displayed register and 0006 posted in the implied register).

7.4 Floating Point EMTH Functions

To make use of the floating point (FP) capability, the four-digit integer values used in standard math instructions (see Chapter 6) must be converted to the IEEE floating point format. All calculations are then performed in FP format, and the results must be converted back to integer format.

7.4.1 The IEEE Floating Point Standard

EMTH floating point functions require values in 32-bit IEEE floating point format. Each value has two registers assigned to it—the eight most significant bits representing the exponent and the other 23 bits (plus one assumed bit) representing the mantissa and the sign of the value.

☞ **Note:** Floating point calculations have a mantissa precision of 24 bits, which guarantees the accuracy of the seven most significant digits. The accuracy of the eighth digit in an FP calculation can be inexact.

It is virtually impossible to recognize an FP representation on the programming panel. Therefore, all numbers should be converted back to integer format before you attempt to read them.

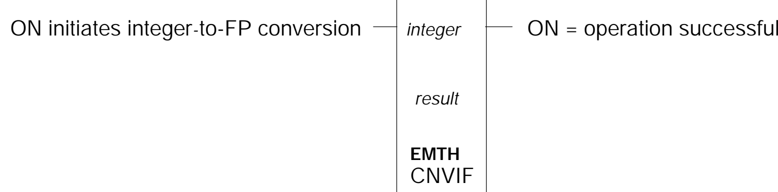
7.4.2 Dealing with Negative Floating Point Numbers

Standard integer math calculations do not handle negative numbers explicitly. The only way to identify negative values is by noting that the SUB function block has turned the bottom output ON.

If such a negative number is being converted to floating point, perform the Integer-to-FP conversion (EMTH function #9), then use the Change Sign function (EMTH function #24) to make it negative prior to any other FP calculations.

7.4.3 Integer-to-Floating Point Conversion

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The double precision *integer* value to be converted to 32-bit FP format is stored here.

Note: If an invalid *integer* value (> 9999) is entered in either of the two top-node registers, the FP conversion will be performed but an error will be reported and logged in the EMTH ERLOG function (see page 138). The *result* of the conversion may not be correct.

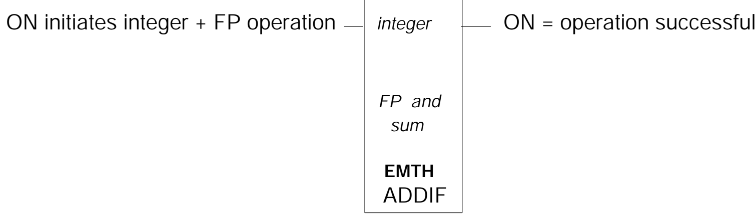
Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The FP *result* of the conversion is posted in the second and third implied registers. The displayed register and the first implied register are not used in the function but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.4 Integer + Floating Point Addition

Block Structure



Top Node Content

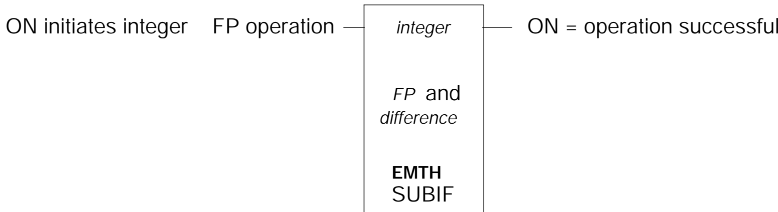
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The double precision *integer* value to be added to the *FP* value is stored here.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The displayed register and the first implied register store the *FP* value to be added in the operation, and the *sum* is posted in the second and third implied registers. The *sum* is posted in FP format.

7.4.5 Integer Floating Point Subtraction

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The double precision *integer* value from which the *FP* value is subtracted is stored here.

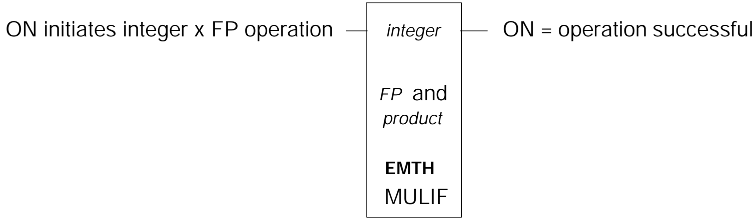
Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The displayed register and

the first implied register store the *FP* value to be subtracted from the *integer* value, and the *difference* is posted in the second and third implied registers. The *difference* is posted in FP format.

7.4.6 Integer x Floating Point Multiplication

Block Structure



Top Node Content

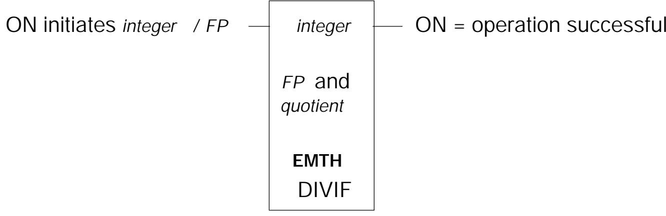
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The double precision *integer* value to be multiplied by the *FP* value is stored here.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The displayed register and the first implied register store the *FP* value to be multiplied in the operation, and the *product* is posted in the second and third implied registers. The *product* is posted in FP format.

7.4.7 Integer Divided by Floating Point

Block Structure



Top Node Content

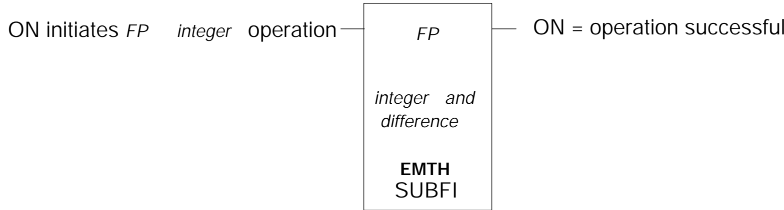
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The double precision *integer* value to be divided by the *FP* value is stored here.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The displayed register and the first implied register store the *FP* value to be divided in the operation, and the *quotient* is posted in the second and third implied registers. The *quotient* is posted in FP format.

7.4.8 Floating Point Integer Subtraction

Block Structure



Top Node Content

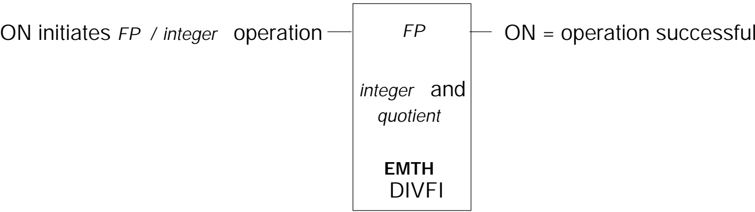
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The *FP* value from which the *integer* value is subtracted is stored here.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The displayed register and the first implied register store the double precision *integer* value to be subtracted from the *FP* value, and the *difference* is posted in the second and third implied registers. The *difference* is posted in FP format.

7.4.9 Floating Point Divided by Integer

Block Structure



Top Node Content
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The FP value to be divided by the *integer* value is stored here.

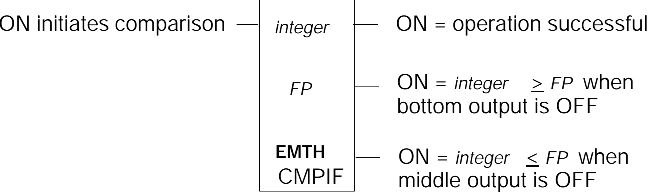
Middle Node Content
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The double precision *integer* value that divides the *FP* value is posted in the displayed register and the first implied register, and the quotient is posted in the second and third implied registers. The *quotient* is posted in FP format.

7.4.10 Integer-Floating Point Comparison

When EMTH function 16 compares its *integer* and *FP* values, the combined states of the middle and bottom outputs indicate their relationship:

Middle Output	Bottom Output	Relationship
ON	OFF	*integer* > *FP*
OFF	ON	*integer* < *FP*
ON	ON	*integer* = *FP*

Block Structure

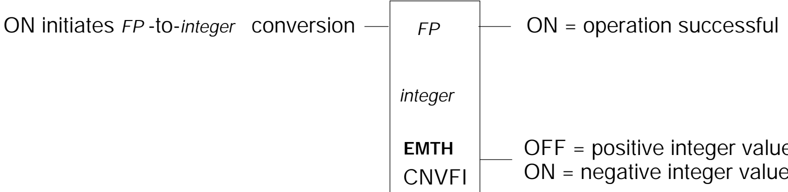


Top Node Content
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The double precision *integer* value to be compared is stored here.

Middle Node Content
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The *FP* value to be compared is entered in the displayed register and the first implied register; the second and third implied registers are not used in the comparison but their allocation in state RAM is required.

7.4.1 1 Floating Point-to-Integer Conversion

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The *FP* value to be converted is stored here.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The double precision *integer* result of the conversion is stored in the second and third implied registers. This value should be the largest *integer value* possible that is ≤ the *FP value*. For example, the *FP* value 3.5 is converted to the *integer* value 3, while the *FP* value 3.5 is converted to the *integer* value 4.

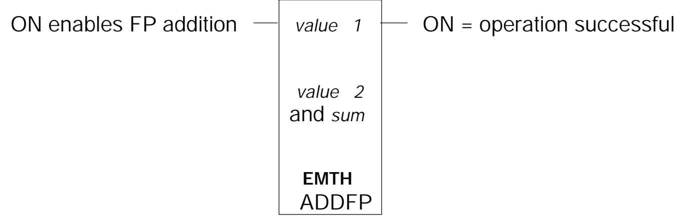
Note: If the resultant integer is too large for 984 double precision integer format (> 99,999,999), the conversion still occurs but an error is logged in the EMTH ERLOG function (see page 138).

The displayed register and the first implied register in the middle node are not used in the conversion but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.12 Floating Point Addition

Block Structure



Top Node Content

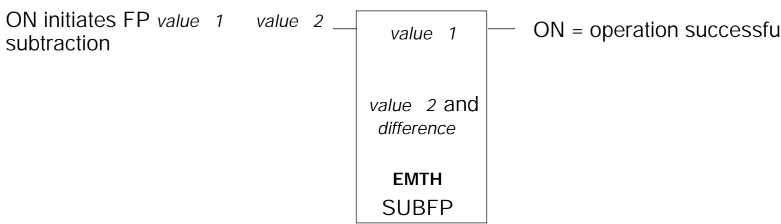
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. FP *value 1* in the addition is stored here.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. FP *value 2* is stored in the displayed register and the first implied register. The *sum* of the addition is stored in FP format in the second and third implied registers.

7.4.13 Floating Point Subtraction

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. FP *value 1*—the value from which *value 2* will be subtracted—is stored here.

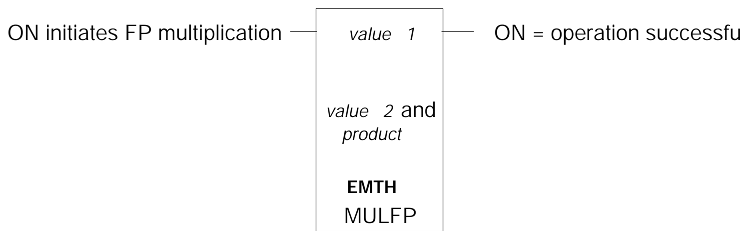
Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. FP *value 2*—the value to be subtracted from *value 1*—is stored in the displayed register and the

first implied register. The *difference* of the subtraction is stored in FP format in the second and third implied registers.

7.4.14 Floating Point Multiplication

Block Structure



Top Node Content

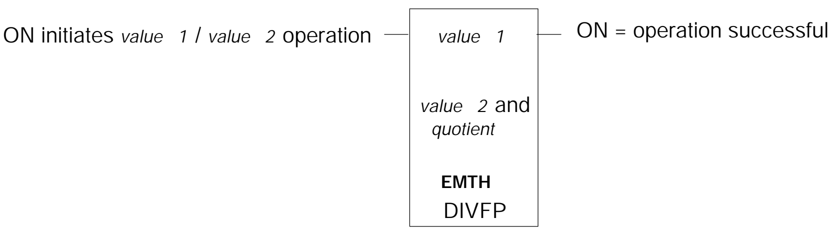
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. FP *value 1* in the multiplication operation is stored here.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. FP *value 2* in the multiplication operation is stored in the displayed register and the first implied register. The *product* of the multiplication is stored in FP format in the second and third implied registers.

7.4.15 Floating Point Division

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. FP *value 1*, which will be divided by the *value 2*, is stored here.

Middle Node Content

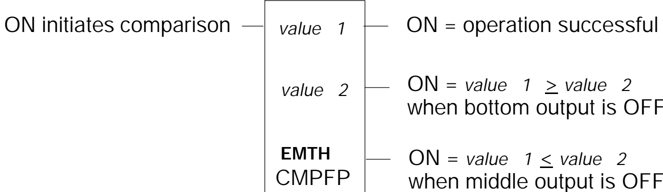
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. FP *value 2*, the value by which *value 1* is divided, is stored in the displayed register and the first implied register. The *quotient* is posted in FP format in the second and third implied registers.

7.4.16 Floating Point Comparison

When EMTH function 22 compares its two *FP* values, the combined states of the middle and bottom outs indicate their relationship:

Middle Output	Bottom Output	Relationship
ON	OFF	*value 1* > *value 2*
OFF	ON	*value 1* < *value 2*
ON	ON	*value 1* = *value 2*

Block Structure



Top Node Content

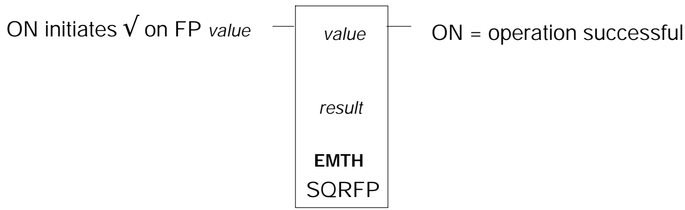
The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The first FP value (*value 1*) to be compared is stored here.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The second FP value (*value 2*) to be compared is entered in the displayed register and the first implied register; the second and third implied registers are not used in the comparison but their allocation in state RAM is required.

7.4.17 Floating Point Square Root

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The FP *value* on which the square root operation is performed is stored here.

Middle Node Content

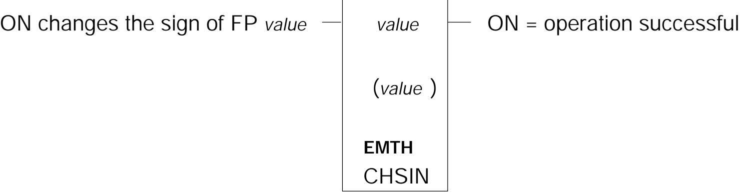
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The *result* of the square root operation is posted in FP format in the second and third implied registers. The displayed register and the first implied register in the middle node are not used in the operation but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.18 Changing the Sign of a Floating Point Number

Block Structure



Top Node Content
The first of two contiguous 4x registers is entered in the top node. The second register is implied. The FP *value* whose sign will be changed is stored here.

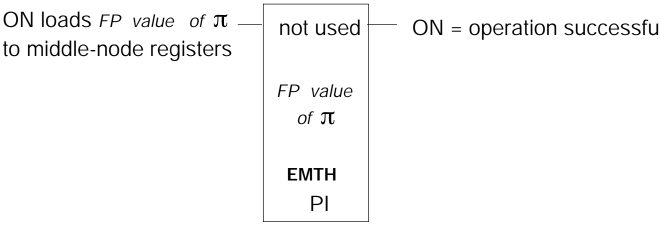
Middle Node Content
The first of four contiguous 4x registers is entered in the middle node. The remaining three registers are implied.

The top node FP *value* in the top node is posted in the second and third implied registers. The displayed register and the first implied register in the middle node are not used in the operation but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4x reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.19 Load the Floating Point Value of π

Block Structure



Top Node Content
The first of two contiguous 4x registers is entered in the top node. The second register is implied. These registers are not used but their allocation in state RAM is required.

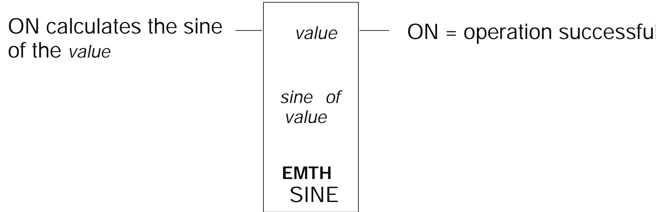
Middle Node Content
The first of four contiguous 4x registers is entered in the middle node. The remaining three registers are implied.

The *FP value of* π is posted in the second and third implied registers. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.20 Floating Point Sine of an Angle (in Radians)

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. An FP *value* indicating the value of an angle in radians is stored here. The magnitude of this value must be < 65536.0; if not:

- The sine is not computed
- An invalid result is returned
- An error is flagged in the EMTH ERLOG function (see page 138)

Middle Node Content

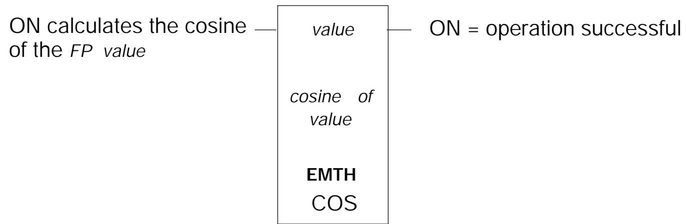
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The sine of the *value* in the top node is posted in the second and third implied registers in FP format. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.21 Floating Point Cosine of an Angle (in Radians)

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. An *FP value* indicating the value of an angle in radians is stored here. The magnitude of this value must be < 65536.0; if not:

- The cosine is not computed
- An invalid result is returned
- An error is flagged in the EMTH ERLOG function (see page 138)

Middle Node Content

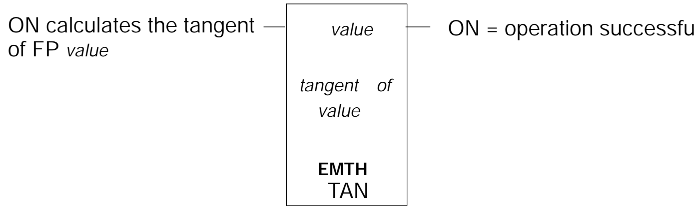
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The cosine of the *value* in the top node is posted in the second and third implied registers in FP format. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.22 Floating Point Tangent of an Angle (in Radians)

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. A *value* in FP format indicating the value of an angle in radians is stored here. The magnitude of this value must be < 65536.0; if not:

Middle Node Content

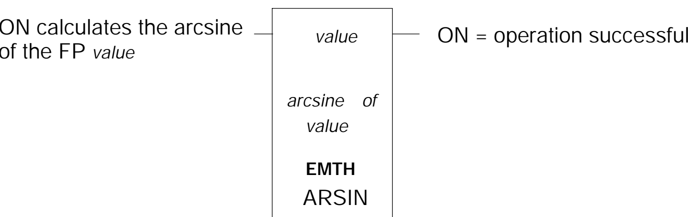
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The tangent of the *value* in the top node is posted in the second and third implied registers in FP format. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.23 Floating Point Arcsine of an Angle (in Radians)

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. An *FP value* indicating the sine of an angle between $\pi/2$ ... $\pi/2$ radians is stored here. This *value* —the sine of an angle—must be in the range of 1.0 ... +1.0; if not:

- The arcsine is not computed
- An invalid result is returned
- An error is flagged in the EMTH ERLOG function (see page 138)

Middle Node Content

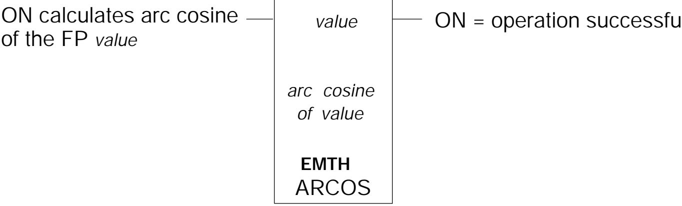
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The arcsine in radians of the *value* in the top node is posted in the second and third implied registers in FP format. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.24 Floating Point Arc Cosine of an Angle (in Radians)

Block Structure



Top Node Content

The first of two contiguous 4x registers is entered in the top node. The second register is implied. An FP value indicating the cosine of an angle between 0 ... p radians is stored here. This value must be in the range of 1.0 ... +1.0; if not:

- ☐ The arc cosine is not computed
- ☐ An invalid result is returned
- ☐ An error is flagged in the EMTH ERLOG function (see page 138)

Middle Node Content

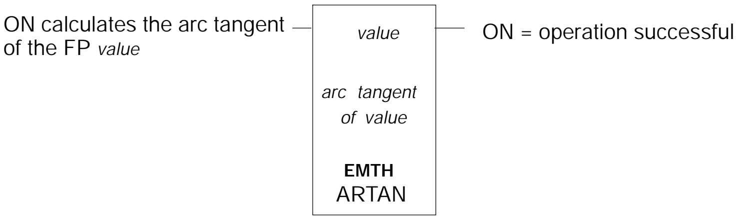
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The arc cosine in radians of the FP *value* in the top node is posted in the second and third implied registers. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.25 Floating Point Arc Tangent of an Angle (in Radians)

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. An FP *value* indicating the tangent of an angle between π/2 ... π/2 radians is stored here. Any valid FP value is allowed.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

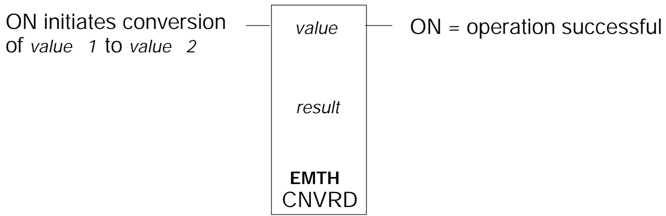
The arc tangent in radians of the FP *value* in the top node is posted in the second and third implied registers. The displayed register and the

first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.26 Floating Point Conversion of Radians to Degrees

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The *value* in FP format of an angle in radians is stored here.

Middle Node Content

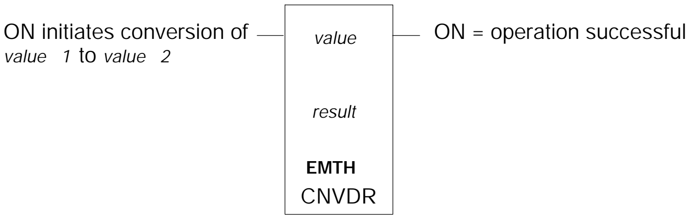
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The converted *result* in FP format of the top-node *value* (in degrees) is posted in the second and third implied registers. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.27 Floating Point Conversion of Degrees to Radians

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The *value* in FP format of an angle in degrees is stored here.

Middle Node Content

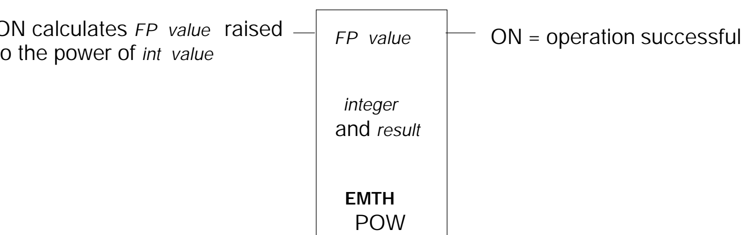
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The converted *result* in FP format of the top-node *value* (in radians) is posted in the second and third implied registers. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.28 Raising a Floating Point Number to an Integer Power

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. The *FP value* to be raised to the integer power is stored here.

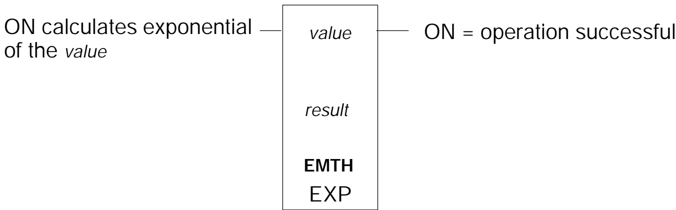
Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The bit values in the displayed register must all be cleared to zero. An *integer* value representing the power to which the top-node *value* will be raised is stored in the first implied register. The *result* of the *FP value* being raised to the power of the *integer* value is stored in the second and third implied registers.

7.4.29 Floating Point Exponential Function

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. A *value* in FP format in the range 87.34 ... +88.72 is stored here.

If the *value* is out of range, the *result* will either be 0 or the maximum value. No error will be flagged.

Middle Node Content

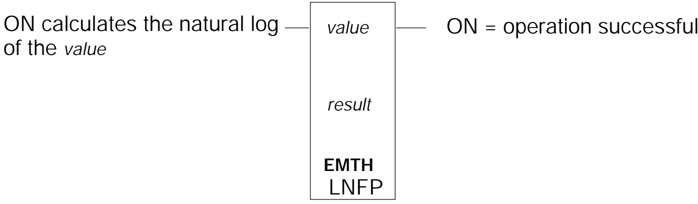
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The exponential of the *value* in the top node is posted in FP format in the second and third implied registers. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.30 Floating Point Natural Logarithm

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. A *value* > 0 is stored here in FP format.

If the *value* ≤ 0, an invalid result will be returned in the middle node and an error will be logged in the EMTH ERLOG function (see page 138).

Middle Node Content

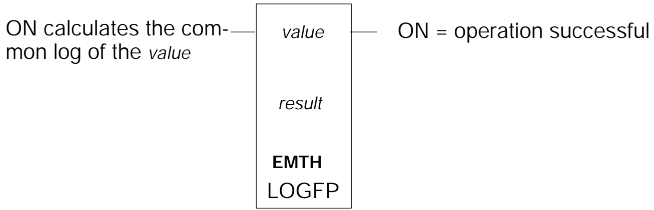
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The natural logarithm of the *value* in the top node is posted in FP format in the second and third implied registers. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.31 Floating Point Common Logarithm

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. A *value* > 0 is stored here in FP format.

If the *value* ≤ 0, an invalid result will be returned in the middle node and an error will be logged in the EMTH ERLOG function (see page 138).

Middle Node Content

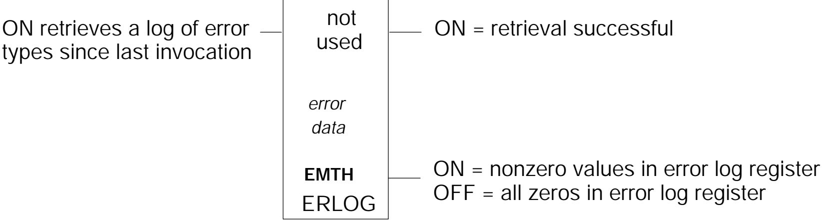
The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied.

The common logarithm of the *value* in the top node is posted in FP format in the second and third implied registers. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since the first two middle-node registers are not used.

7.4.32 Floating Point Error Report Log

Block Structure

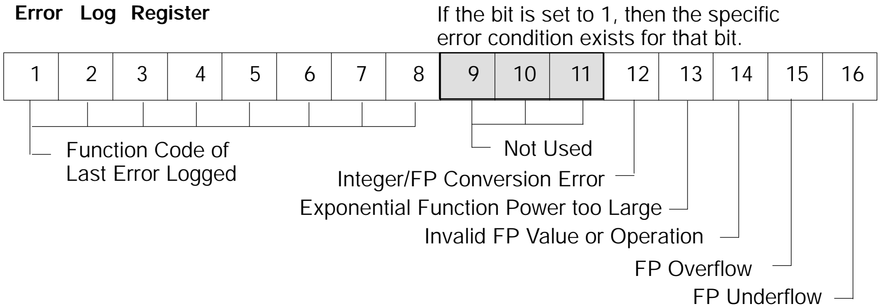


Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. These two registers are not used in the operation but their allocation in state RAM is required.

Middle Node Content

The first of four contiguous 4*x* registers is entered in the middle node. The remaining three registers are implied. The second implied register is used as the error log register:



The third implied register has all its bits cleared to zero. The displayed register and the first implied register are not used but their allocation in state RAM is required.

Tip: To preserve registers, you can make the 4*x* reference numbers assigned to the displayed register and the first implied register in the middle node equal to the register references in the top node, since these registers must be allocated but none are used.

7.5 MA TH

The MATH instruction performs any one of four integer math operations, which is called by entering a function code in the range 1 ... 4 in the bottom node:

Code	MA TH Function
1	Decimal square root
2	Process square root
3	Logarithm (base 10)
4	Antilogarithm (base 10)

7.5.1 Characteristics

Size
Three nodes high

PLC Compatibility

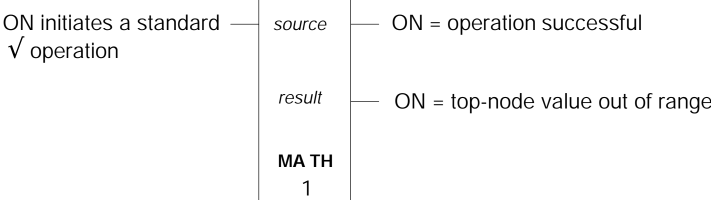
☐ Available as a loadable for the 984A, 984B, and 984X Chassis Mount PLCs (SW-AP9*x*-D*x*A loadable library)

☐ Not available in all other PLC types

Opcode
BE hex (default)

7.5.2 Decimal Square Root

Block Structure



Top Node Content
The first of two contiguous 3*x* or 4*x* registers is entered in the top node. The second register is implied. The *source* value—i.e., the value for which the square root will be derived—is stored here.

If you specify a 4*x* register, the *source* value may be in the range 0 ... 99,999,99. The low-order half of the value is stored in the implied register, and the high-order half is stored in the displayed register.

If you specify a 3*x* register, the *source* value may be in the range 0 ... 9,999. The square root calculation is done on only the value in the displayed register; the implied register is required but not used.

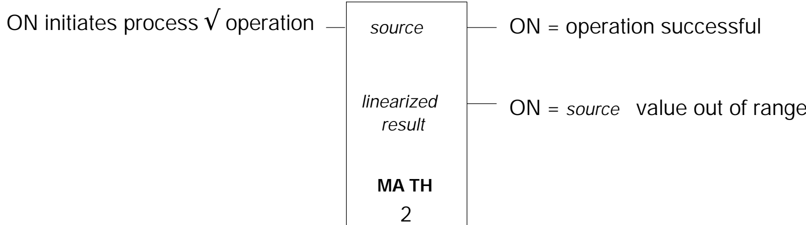
Middle Node Content
Enter the first of two contiguous 4*x* registers in the middle node. The second register is implied. The *result* of the standard square root operation is stored here.

The *result* is stored in the fixed-decimal format: **1234. 5600.** where the displayed register stores the four-digit value to the left of the first decimal point and the implied register stores the four-digit value to the right of the first decimal point. Numbers after the second decimal point are truncated; no round-off calculations are performed.

7.5.3 Process Square Root

The process square root function tailors the standard square root function for closed loop analog control applications. It takes the result of the standard square root result, multiplies it by 63.9922—the square root of 4095—and stores that *linearized result* in the middle-node registers.

Block Structure



Top Node Content
The first of two contiguous 3*x* or 4*x* registers is entered in the top node. The second register is implied. The *source* value—i.e., the value for which the square root will be derived—is stored in these two registers.

In order to generate values that have meaning, the *source* value must not exceed 4095. In a 4*x* register group the *source* value will therefore

be stored in the implied register, and in a 3*x* register group the *source* value will be stored in the displayed register.

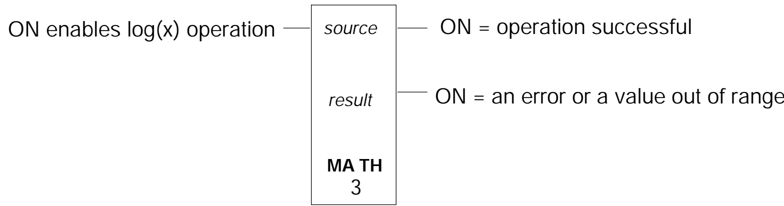
Middle Node Content

The first of two contiguous 4*x* registers is entered in the middle node. The second register is implied. The *linearized result* of the process square root operation is stored here.

The *result* is stored in the fixed-decimal format: **1234. 5600.** where the displayed register stores the four-digit value to the left of the first decimal point and the implied register stores the four-digit value to the right of the first decimal point. Numbers after the second decimal point are truncated; no round-off calculations are performed.

7.5.4 Base 10 Logarithm

Block Structure



Top Node Content

The first of two contiguous 3*x* or 4*x* registers is entered in the top node. The second register is implied. The *source* value upon which the log calculation will be performed is stored in these registers.

If you specify a 4*x* register, the *source* value may be in the range 0 ... 99,999,99. The low-order half of the value is stored in the implied register, and the high-order half is stored in the displayed register.

If you specify a 3*x* register, the *source* value may be in the range 0 ... 9,999. The log calculation is done on only the value in the displayed register; the implied register is required but not used.

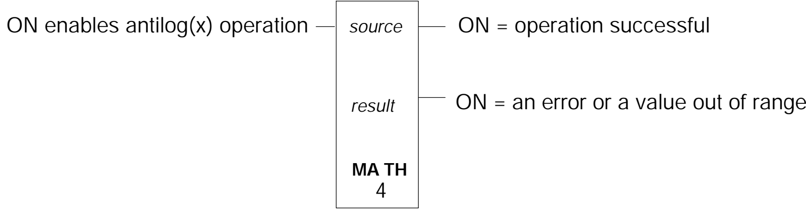
Middle Node Content

The middle node contains a single 4*x* holding register where the *result* of the base 10 log calculation is posted. The *result* is expressed in the fixed decimal format **1. 234** , and is truncated after the third decimal position.

The largest *result* that can be calculated is 7.999, which would be posted in the middle register as 7999.

7.5.5 Base 10 Antilogarithm

Block Structure



Top Node Content

The top node is a single 4*x* holding register or 3*x* input register. The *source* value—i.e., the value on which the antilog calculation will be performed—is stored here in the fixed decimal format **1. 234** . It must be in the range 0 ... 7999, representing a *source* value up to a maximum of 7.999.

Middle Node Content

The first of two contiguous 4*x* registers is entered in the middle node. The second register is implied. The *result* of the antilog calculation is posted here in the fixed decimal format **12345678** .

The most significant bits are posted in the displayed register, and the least significant bits are posted in the implied register. The largest antilog value that can be calculated is 99770006 (9977 posted in the displayed register and 0006 posted in the implied register).

7.6 DMTH

The DMTH function performs any one of four possible double precision math operations, which is called by entering a function code in the range 1 ... 4 in the bottom node:

Code	DMTH Function
1	Double precision addition
2	Double precision subtraction
3	Double precision multiplication
4	Double precision division

7.6.1 Characteristics

Size
Three nodes high

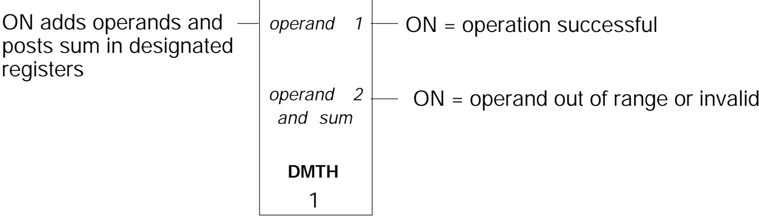
PLC Compatibility

- Available as a loadable for the 984A, 984B, and 984X Chassis Mount PLCs (SW-AP9*x*-D*x*A loadable library)
- Not available in all other PLC types

Opcode
DE hex (default)

7.6.2 Double Precision Addition

Block Structure



Top Node Content
The first of two contiguous 4*x* registers is entered in the top node. The second 4*x* register is implied. *Operand 1* is stored here.

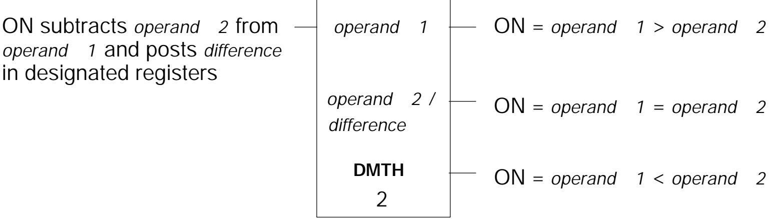
Each register holds a value in the range 0000 ... 9999, for a combined double precision value in the range 0 ... 99,999,999. The high-order half of *operand 1* is stored in the displayed register, and the low-order half is stored in the implied register.

Middle Node Content
The first of six contiguous 4*x* registers is entered in the middle node. The remaining five registers are implied:

- The displayed register and the first implied register store the high-order and low-order halves of *operand 2*, respectively, for a combined double precision value in the range 0 ... 99,999,999
- The value stored in the second implied register indicates whether an overflow condition exists (a value of 1 = overflow)
- The third and fourth implied registers store the high-order and low-order halves of the double precision sum, respectively
- the fifth implied register is not used in the calculation but must exist in state RAM

7.6.3 Double Precision Subtraction

Block Structure



Top Node Content
The first of two contiguous 4*x* registers is entered in the top node. The second 4*x* register is implied. *Operand 1* is stored here.

Each register holds a value in the range 0000 ... 9999, for a combined double precision value in the range 0 ... 99,999,999. The high-order half of *operand 1* is stored in the displayed register, and the low-order half is stored in the implied register.

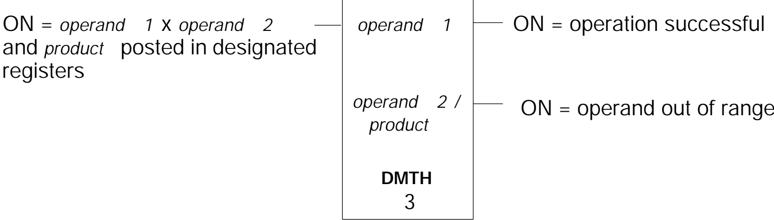
Middle Node Content

The first of six contiguous 4*x* registers is entered in the middle node. The remaining five registers are implied:

- The displayed register and the first implied register store the high-order and low-order halves of *operand 2*, respectively, for a combined double precision value in the range 0 ... 99,999,999
- The second and third implied registers store the high-order and low-order halves, respectively, of the absolute *difference* in double precision format
- The value stored in the fourth implied register indicates whether or not the operands are in the valid range (1 = out of range and 0 = in range)
- The fifth implied register is not used in this calculation but must exist in state RAM

7.6.4 Double Precision Multiplication

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second 4*x* register is implied. *Operand 1* is stored here.

The second 4*x* register is implied. Each register holds a value in the range 0000 ... 9999, for a combined double precision value in the range 0 ... 99,999,999. The high-order half of *operand 1* is stored in the displayed register, and the low-order half is stored in the implied register.

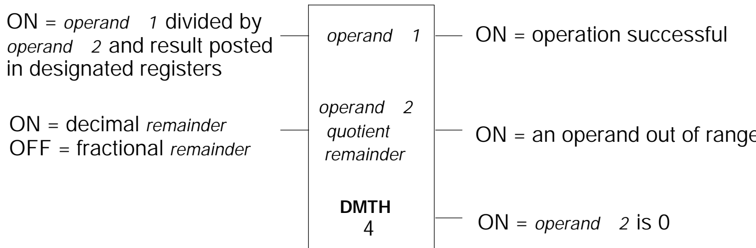
Middle Node Content

The first of six contiguous 4*x* registers is entered in the middle node. The remaining five registers are implied:

- ☐ The displayed register and the first implied register store the high-order and low-order halves of *operand 2*, respectively, for a combined double precision value in the range 0 ... 99,999,999
- ☐ The last four implied registers store the double precision *product* in the range 0 ... 9,999,999,999,999,999

7.6.5 Double Precision Division

Block Structure



Top Node Content

The first of two contiguous 4*x* registers is entered in the top node. The second register is implied. *Operand 1* is stored here.

Each register holds a value in the range 0000 ... 9999, for a combined double precision value in the range 0 ... 99,999,999. The high-order half of *operand 1* is stored in the displayed register, and the low-order half is stored in the implied register.

Middle Node Content

The first of six contiguous 4*x* registers is entered in the middle node. The remaining five registers are implied:

- ☐ The displayed register and the first implied register store the high-order and low-order halves of *operand 2*, respectively, for a combined double precision value in the range 0 ... 99,999,999

Note: Since division by 0 is illegal, a 0 value causes an error—an error trapping routine sets the remaining middle-node registers to 0000 and turns the bottom output ON.

- ☐ The second and third implied registers store an eight-digit *quotient*
- ☐ The fourth and fifth implied registers store the *remainder* —if the *remainder* is expressed as a fraction, it is eight digits long and both registers are used; if the remainder is expressed as a decimal, it is four digits long and only the fourth implied register is used

Chapter 8
Equation Networks

Equation Network is a departure from standard ladder logic. Instead of using a two- or three-high function block configuration, this instruction takes a ladder logic network and uses it as an editor where you can compose a complex equation using algebraic notation. It allows you to use standard math operators such as +, , *, /, as well as conditional and logical expressions. It also lets you specify variables and constants as necessary, and to group expressions in nested layers of parentheses.

The power of an Equation Network is its ability to deal with complexity in a clear and efficient way. An equation composed in a single Equation Network might require many networks of standard ladder logic to produce the same result. An Equation Network can also be read and understood by other users without the need for detailed annotation, as is often required when standard ladder logic is used for complex calculations.

8.1 Equation Network Structure

Equation Network is a special type of ladder logic network that allows you to specify the value of a result register in algebraic notation. If your PLC has a floating point processor, Equation Network takes advantage of this feature for faster processing. It uses a full ladder logic network to compose the equation, with a contact or horizontal short as the enabling input and up to five output coils to describe the state of the result.

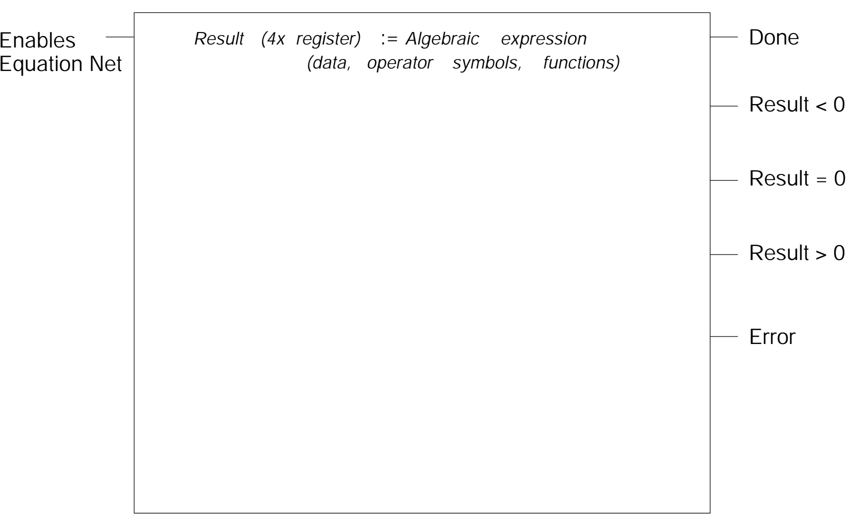
8.1.1 Characteristics

Size
One full ladder logic network

PLC Compatibility
Standard in Quantum PLCs with Executive 2.0 or greater; unavailable in earlier Quantums and for other PLC types

8.1.2 Representation in Ladder Logic

Block Structure



Input
Equation Network has one control input (to the top row), which is used to enable/disable the equation. The input may be a normally open

(N.O.) contact, a normally closed (N.C.) contact, a horizontal short, or an open.

- When an N.O. contact is used, the equation is solved when the contact's referenced coil or input is ON
- When an N.C. contact is used, the equation is solved when the contact's referenced coil or input is OFF
- A horizontal short causes the equation to be solved on every scan
- When an open is used, the Equation Network is not solved

Outputs

Equation Network can produce five possible outputs from the top five rows of the network to describe the result of the equation. You choose the outputs you want to use by assigning 0*x* reference numbers to them.

The outputs are displayed as coils in the last column of the Equation Network. The row in which the output coils are placed determines their meanings:

- When the equation passes power to the output from the top row, the equation has completed successfully without an error
- When the equation passes power to the output from the second row, the equation has completed successfully and the result is less than zero
- When the equation passes power to the output from the third row, the equation has completed successfully and the result is equal to zero
- When the equation passes power to the output from the fourth row, the equation has completed successfully and the result is greater than zero
- When the equation passes power to the output from the fifth row, the data in the equation has caused a calculation error

If the fifth output goes ON, it indicates an error condition. One of the following messages will appear at the bottom of the Equation Network screen:

Error Message	Meaning
`Invalid Op.`	An internal error generated by the math coprocessor
`Overflow`	A value is too large to be represented in its specified data type
`Underflow`	(For floating point data only) A number is too small to be represented in FP format
`Divide by 0`	The variable, constant, or result of a function directly to the right of a / operator has a value of zero
`Invalid operation with Boolean Data`	Occurs when a Boolean value is entered in an argument to a function

Equation Content

The content of the Equation Network is in the form:

result := *algebraic expression*

where

- The *result* is a variable contained in one or two 4*x* registers—it may be a signed or unsigned 16-bit short integer, a signed or unsigned 32-bit long integer, or a floating point number
- The *algebraic expression* is a syntactically correct construction of variable and/or constant data, standard algebraic operators, and/or functions; parentheses can be used to define the order in which the expression is evaluated and to indicate arguments to functions within the expression

Size of the Equation Network

An Equation Network can contain up to a maximum of 81 words. Words are used according to the following rules:

- The enabling input, if it is an N.O. or N.C. contact, consumes one word; an open consumes one word; a horizontal short used as the input does not consume a word
- Each output coil you implement consumes one word
- Each 16-bit register and/or discrete reference in the Equation Network consumes one word
- Each operator in the Equation window consumes one word
- Each function in the Equation window consumes one word

- Each short integer constant consumes one word
- Each floating point or long constant consumes two words
- Each open/closed parenthetical pair consumes two words

8.2 Data Types

Six data types are allowed in an Equation Network. Each variable and constant used in the Equation Network is of one of these data types. Data types can be mixed in an Equation Network.

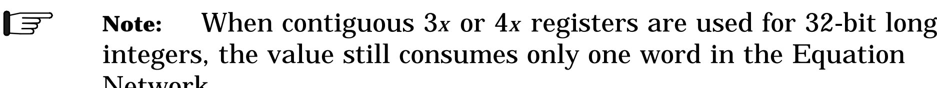
A data type is specified by appending a suffix to a variable or constant. Data type suffixes are:

Data Type	Suffix	Applies to
Boolean (binary)	B	Constants, 1*x*, or 0*x*
Unsigned short integer	U	Constants, 3*x*, or 4*x*
Signed short integer	S	Constants, 3*x*, or 4*x*
Signed long integer	L	Constants, 3*x*, or 4*x*
Unsigned long integer	UL	Constants, 3*x*, or 4*x*
Floating point number	F	Constants, 3*x*, or 4*x*

8.2.1 Variable Data

Variable data within an Equation Network can be in 0*x* and 1*x* discrete references and in 3*x* and 4*x* registers.

Data Type	Variable Type	Words Consumed	Registers Consumed
Boolean	0*x* or 1*x*	One	N/A
Unsigned 16-bit variable	3*x* or 4*x*	One	One
Signed 16-bit variable	3*x* or 4*x*	One	One
Unsigned long (32-bit) variable	3*x* or 4*x*	One	Two
Signed long (32-bit) variable	3*x* or 4*x*	One	Two
Floating point variable	3*x* or 4*x*	One	Two



Note: When contiguous 3*x* or 4*x* registers are used for 32-bit long integers, the value still consumes only one word in the Equation Network.

Note: When 3*x* or 4*x* registers are used for a floating point number, the value requires one word for complete definition.

Entering Variable Data in an Equation Network

When entering a 0*x* or 1*x* reference as a discrete variable in an Equation Network, the reference is assumed to be Boolean and you do not need append the suffix B to the reference. Thus, the entries **000010** and **000010B** are equivalent.

No other suffixes are legal with a 0*x* or 1*x* reference.

When entering a 3*x* or 4*x* register in an Equation Network, the following rules apply:

- If you enter a register without a suffix appended to it, it is assumed to represent a signed 16-bit integer variable and you do not need append the suffix S to the reference; thus the entries **400023** and **40023S** are equivalent
- If you enter a register with the suffix U appended to it—e.g., **300004U**—you indicate that a single register containing an unsigned 16-bit integer variable is used
- If you enter a register with the suffix L appended to it, you indicate that two contiguous registers containing a signed 32-bit long integer variable are used—e.g., **400012L** implies that register 400013 is also used
- If you enter a register with the suffix UL appended to it, you indicate that two contiguous registers containing an unsigned 32-bit long integer variable are used—e.g., **300006UL** implies that register 300007 is also used
- If you enter a register with the suffix F appended to it, you indicate that two contiguous registers containing a floating point variable are used—e.g., 400101F implies that register 400102 is also used
- The suffix B cannot be appended to a 3*x* or 4*x* register entry

8.2.2 Constant Data

Constants can also be used to specify data in an Equation Network. Long (32-bit) constants and floating point constants always require two words. The least significant byte (LSB) is always in the first of the two words; both words must have the same data type.

Data Type	words Consumed	Valid Range of Values
Boolean	One	0, 1
Signed 16-bit constant	One	32,768 ... +32,767
Unsigned 16-bit constant	One	0 ... 65,535
Signed long (32-bit) constant	Two	2 x 10^{9} ... +2 x 10^{9}
Unsigned long (32-bit) constant	Two	0 ... 4,294,967,295
Floating Point constant	Two	8.43 x 10^{37} ≤ |x| ≤ 3.402 x 10^{38}

Entering Constant Data in an Equation Network

A constant is prefaced with a # sign and appended with a data-type suffix (see page 154). All constant values are in decimal format—hexadecimal values are not allowed in Modsoft.

If a constant is entered in an Equation Network without a suffix appended to it, it is assumed to be a signed short integer. For example, the entries # **3574** and # **3574S** are equivalent.

A Boolean constant must have the suffix B appended to it. The only two valid Boolean constants are **#0B** and **#1B**; no other values are legal Boolean constants.

8.3 Algebraic Operators

Below is the list of operator symbols supported by Equation Network. They are grouped by precedence from highest to lowest, where unary operations are evaluated before exponentiation operations in an expression, multiply operations are evaluated before add operations, etc.

<table>
<tr><th>Operation Groupings</th><th>Operator Symbol</th><th>Description</th></tr>
<tr><td>Unary</td><td></td><td>Negation</td></tr>
<tr><td></td><td>~</td><td>One's complement</td></tr>
<tr><td>Exponentiation</td><td>**</td><td>Exponent</td></tr>
<tr><td>Multiply</td><td>*</td><td>Multiplication</td></tr>
<tr><td></td><td>/</td><td>Division</td></tr>
<tr><td>Add</td><td>+</td><td>Addition</td></tr>
<tr><td></td><td></td><td>Subtraction</td></tr>
<tr><td>Logical bitwise</td><td>&</td><td>AND</td></tr>
<tr><td></td><td>|</td><td>OR</td></tr>
<tr><td></td><td><<</td><td>Left shift</td></tr>
<tr><td></td><td>>></td><td>Right shift</td></tr>
<tr><td></td><td>^</td><td>XOR</td></tr>
<tr><td>Relational</td><td><</td><td>Less than</td></tr>
<tr><td></td><td><=</td><td>Less than or equal to</td></tr>
<tr><td></td><td>=</td><td>Equal</td></tr>
<tr><td></td><td>< ></td><td>Not equal</td></tr>
<tr><td></td><td>>=</td><td>Greater than or equal to</td></tr>
<tr><td></td><td>></td><td>Greater than</td></tr>
<tr><td>Conditional</td><td>? x : y</td><td>Then x Else y (required after a relational (If) argument)</td></tr>
<tr><td>Assignment</td><td>:=</td><td>Placed between the result and the expression in an Equation Network; indicates that the value of the expression is copied into the result variable</td></tr>
</table>

8.3.1 How an Equation Network Resolves an Equation

A Equation Network will calculate its *result* in one of two ways, depending on the types of operators used in the *expression* :

method 1 Evaluate a single *expression* and execute it by copying the derived value to the *result* register

method 2 Evaluate the validity of the first of three arguments in a conditional expression and execute by copying the value from either the second or third argument in the conditional expression to the *result* register

If the expression being evaluated contains only some combination of unary, exponentiation, mathematical , and/or logical bitwise operators, it is treated as a single argument and is solved via method 1. For example, in the equation:

```
400001 := (#16 ** #2    #5) * #7
```

the square of 16 (256) minus 5 (251) is multiplied by 7, and the result (1,757) is copied to register 400001.

If you use one or more of the six relational operators shown in the previous table, you are creating the first of three arguments that comprise a conditional expression. The conditional operators must be used to create Then/Else arguments in the *expression* , and method 2 is used to execute the *result* . For example, in the equation:

```
400001 := 400002 >= #100 ? 300001 : 300002
```

the value in register 400002 is evaluated to see if it is greater than or equal to 100—this is the first argument in the conditional expression. If the value is greater than or equal to 100, the second argument is executed and the value in register 300001 is copied to register 400001. If it is less than 100, the third argument is executed and the value in register 300002 is copied to register 400001.

8.3.2 Operator Precedence

In a string of data types and operators, the order of precedence in the expression determines the order in which operations will be evaluated. For example, in the equation:

```
400001 := 300001F ** 300002F * 300003 + 300004 & 300005          (1)
> 300006 ? 300007 : 300008
```

the operations in the first argument of the conditional expression are evaluated from left to right in the order they appear. First, the value in register 300001 is raised to the power of the value in register 300002, then multiplied by the value in register 300003. That result is added to the value in register 300004, then logically ANDed with the value in register 300005, and compared with the value in register 300006.

If the > comparison is true, the second argument in the conditional expression is executed, and the value in register 300007 is copied to register 400001. If the > comparison is false, the third argument in the conditional expression is executed, and the value in register 300008 is copied to register 400001.

Operator precedence forces the opposite effect on the first argument of the conditional expression in equation (2) below:

400001 := 300002U > 300003U & 300004U + 300005F * 300006F (2)
** 300007U ? 300008 : 300009

Here the first operation to be evaluated is the exponentiation of the value in register 300006 by the value in register 300007, followed by multiplication by the value in register 300005, then addition with the value in register 300004, then logically ANDing the result with the value in register 300003, and finally comparing the result with the value in register 300002.

If the > comparison is true, the second argument in the conditional expression is executed, and the value in register 300008 is copied to register 400001. If the > comparison is false, the third argument in the conditional expression is executed, and the value in register 300009 is copied to register 400001.

When operators of equal precedence appear in an expression, they are generally evaluated in the order from left to right and top to bottom in the Equation Network.

8.3.3 Using Parentheses in an Equation Network Expression

You can alter the the order in which an expression is evaluated by enclosing portions of the expression in parentheses. Parenthetical portions of the expressions are evaluated before portions outside the parentheses. Notice the difference between in the way the following expressions are evaluated with and without parentheses:

400001 := 300001U < 300002U | 300004U & 300001U (3a)
+ 300003U ? 300004 : 300005

and

400001 := 300001U < (300002U | 300004U & 300001U) (3b)
+ 300003U ? 300004 : 300005

The expression in equation (3a) is evaluated by precedence as:

```
300001U < ( ( 300002U | 300004U ) & ( 300001U + 300003U ) )
? 300005 : 300006
```

where the sum of the values in registers 300001 and 300003 is ANDed with the logical OR of the values in registers 300002 and 300004.

On the other hand, expression (3b) is evaluated by ORing the values in registers 300002 and 300004, then ANDing the result with the value in register 300001, and finally adding the value in register 300003.

Nested Parentheses

When multiple levels of parenthetical data are nested in an expression, the most deeply nested parenthetical data is evaluated first. An Equation Network permits up to 10 nested levels of parentheses in an expression.

For example, the order in which the expression in equation (2) is evaluated can be seen more clearly when parentheses are used:

```
300002U > ( 300003U & ( 300004U + ( 300005U * ( 300006F **
300007F ) ) ) ) ? 300008 : 300009
```

Entering Parentheses in an Equation Network

Equation Network will echo back to you the expression as you enter it. It does not prevent you from entering additional levels of parentheses even when they may not be necessary to make the expression syntactically correct. For example, in the expression:

```
( ( ( ( 300004U + 300005U ) ) ) ) / 300006U
```

Equation Network maintains the four nested level of parentheses in the expression even when only one set of parentheses may be needed.

Note: The expression must have an equal and balanced number of open and closed parentheses in order to compile properly. If it does not, a compiler error will be generated and the Equation Network will not function.

Each pair of open and closed parentheses consumes two words in the Equation Network.

8.4 Functions

The following functions are recognized in an Equation Network:

Function Name	Meaning
ABS	Absolute value
ARCCOS	Arc cosine
ARCSIN	Arc sine
ARCTAN	Arc tangent
COS	Cosine
COSD	Cosine of degrees
EXP	Exponent function (power of *e*) Does not need to be a whole number
FIX	Convert floating point to integer (presumes an FP argument)
FLOAT	Converts integer to floating point (presumes an integer argument)
LN	Natural logarithm (base *e*)
LOG	Common logarithm (base 10)
SIN	Sine of radians
SIND	Sine of degrees
SQRT	Square root
TAN	Tangent of radians
TAND	Tangent of degrees

Each function used in an Equation Network consumes one word.

8.4.1 Entering Functions in an Equation Network

A function must be entered with its argument in the following form in the Equation Network expression:

function name (*argument*)

where the *function name* is one of those listed in the table above and the *argument* is entered in parentheses immediately after the *function name*. The *argument* may be entered as:

- One or more unary operations
- One or more exponential operations
- One or more multiplication/division operations
- One or more addition/subtraction operations

- One or more logical operations
- One or more relational operations
- Any legal combinations of the above operations

For example. if you want to calculate the absolute value of the sine of the number in FP register 400025 and place the result in FP register 400015, enter the following in the Equation Network:

```
400015F := ABS (SIN (400025F))
```

See section 8.3 for more details about these operations.

8.4.2 Limits on the Argument to a Function

The argument to a function in an Equation Network is resolved to a floating point (FP) number. The FP value must be in the following range, depending on the type of function:

Function	Argument	Range
ABS	FP value	3.402823×10^{38} ... $+3.402823 \times 10^{38}$
ARCCOS	FP value	1.00000 ... +1.00000
ARCSIN	FP value	1.00000 ... +1.00000
ARCTAN	FP value	3.402823×10^{38} ... $+3.402823 \times 10^{38}$
COS	FP value	3.402823×10^{38} ... $+3.402823 \times 10^{38}$
COSD	FP value	3.224671×10^{4} ... $+3.224671 \times 10^{4}$
EXP	FP value	87.33655 ... +88.72284
FIX	FP value	2.147484×10^{9} ... $+2.147484 \times 10^{9}$
FLOAT	FP value	3.402823×10^{38} ... $+3.402823 \times 10^{38}$
LN	FP value	0 ... 3.402823×10^{38}
LOG	FP value	0 ... 3.402823×10^{38}
SIN	FP value	3.402823×10^{38} ... $+3.402823 \times 10^{38}$
SIND	FP value	1.724705×10^{4} ... $+1.724705 \times 10^{4}$
SQRT	FP value	0 ... 3.402823×10^{38}
TAN	FP value	3.402823×10^{38} ... $+3.402823 \times 10^{38}$, not $^{p}/_{2} \times n$ (where *n* is an integer value)
TAND	FP value	1.351511×10^{4} ... $+1.351511 \times 10^{4}$, not $90 \times n$ (where *n* is an integer value)

8.5 Data Conversions in an Equation Network

In an Equation Network, some combinations of operators will convert the value of an operand from one data type to another. The following set of rules applies to mixed data types in an Equation Network:

- All 16 bit signed and unsigned numbers are automatically promoted to 32 bits before an operation.
- In an operation between signed and unsigned numbers, the unsigned number is assumed to be signed without checking for overflow.
- An operation involving a Boolean and any other data type uses the other data type and assigns a value of 1 or 0 to the Boolean.
- An operation between floating point numbers and signed or unsigned numbers automatically promotes the long integer to floating point and assumes assigned number without checking for overflow.
- An operation involving a bitwise logical AND, OR, or XOR does not check data types and automatically assumes unsigned numbers.
- A bitwise logical AND, OR, or XOR operation with a Boolean argument results in a 0 (false) or a 0*x*FFFFFFFF (true).
- The unary Not One's complement operation does not operate on floating point numbers and treats signed numbers as if they were unsigned.
- In a shift forward or shift back operation, the number by which the argument is being shifted is always treated as a positive integer between 0 ... 32. If the value of the *by* number > 32, it is automatically ANDed with 0*x*1f to make it < 32.
- Signed numbers are shifted arithmetically, and unsigned numbers are shifted logically.
- A floating point number that is shifted becomes useless, since its data type remains unchanged.
- Attempting to shift a Boolean argument produces an error.

- The unary negation of an unsigned number produces that number's two's complement.
- The unary negation of a signed or floating point number changes the sign of the number.
- The unary negation of a Boolean operator results in a change of true/false state of the Boolean.
- An absolute value operation does not change the data type of the result.
- Attempting to find the absolute value of a Boolean argument produces an error.
- A floating point result is always produced by an EXP, LN, LOG, SQRT, SIN, COS, TAN, SIND, COSD, TAND, ARCSIN, ARCCOS, or ARCTAN function. If the original argument was not a floating point number, it will be promoted to one, assuming a signed number without checking for overflow. The exception is an original Boolean argument, which will produce an error with any of these functions.
- A Boolean + Boolean operation is an OR operation.
- A Boolean   Boolean operation is an XOR operation.
- Boolean * Boolean, Boolean / Boolean, and Boolean ** Boolean operations are AND operations.
- A Boolean assignment (=) to a signed or unsigned number produces a signed or unsigned 0 or 1.
- A Boolean assignment (=) to a floating point number produces a floating point 0.0 or 1.0.
- A long/short signed/unsigned number assignment (=) to a short unsigned number produces a result in the range 0 ... 65,535. Overflow is set if the result is > 65,535.
- A long/short signed/unsigned number assignment (=) to a short signed number produces a result in the range 32,768 ... 32,767. Overflow is set if the result is > 32,767 or < 32,768.
- A floating point number assignment (=) to a long/short signed/unsigned number will be truncated.

- A floating point number assignment (=) to a short unsigned number produces a result in the range 0 ... 65,535. Overflow is set if the result is > 65,535.

- A floating point number assignment (=) to a short signed number produces a result in the range 32,768 ... 32,767. Overflow is set if the result is > 32,767 or < 32,768.

- A floating point number assignment (=) to a long unsigned number produces a result in the range 0 ... 4,294,967,295. Overflow is set if the result is >4,294,967,295.

- A floating point number assignment (=) to a long signed number produces a result in the range 2,147,483,648 ... 2,147,483,647. Overflow is set if the result is >2,147,483,647 or < 2,147,483,648.

8.6 Roundoff Differences in PLCs without a Math Coprocessor

Equation Networks can be executed by Quantum PLCs like the 140 CPU 424 02 and 140 CPU 213 04, which have on board math coprocessors, as well as by the 140 CPU 113 02 and 03 PLCs, which do not have math coprocessors. Quantum PLCs without math coprocessors use a 32-bit processing mechanism within the PLC itself to handle floating point calculations, and they can produce results that are less accurate than those produced by the 80-bit math coprocessor.

Differences in accuracy can be noticed starting in the sixth position to the right of the decimal point. For example, the 140 CPU 424 02 and 213 04 will calculate the equation

401010F = SIN(#45)

and produce the result 0.8509035, whereas the 140 CPU 113 02/03 will handle the same equation and produce the result 0.8509022.

For applications that require accuracy beyond the fifth decimal position, a Quantum PLC with a math coprocessor is recommended. Generally, if your application does not require this kind of accuracy, a PLC without a math coprocessor may be acceptable.

Another potential consideration is the effect of less accurate calculation on a truncated result. For example, a PLC with a math coprocessor will calculate the tangent of 225 degrees

401015F = TAND(#225)

as 1, whereas a PLC without a math coprocessor will produce the result 0.999991. If we were to assign the TAND operation to a non-floating point register, Equation Network will truncate the result so that

401040 = TAND(#225)

will produce a result of 1 when the math coprocessor is used but a result of 0 when the coprocessor is not used.

8.7 Benchmark Performance

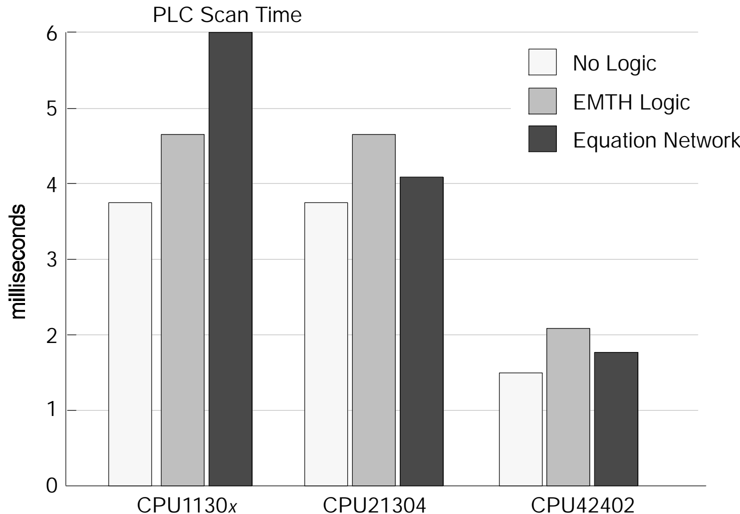
Benchmark tests were performed on three Quantum PLCs—the CPU113, CPU213, and CPU 424—solving the same equation with an Equation Network operation and EMTH ladder logic operations. The equation was:

A = ((B*C) + D   E / SINE F)

where A, B, C, D,E, and F are either constants or registers.

Note: This equation was the only logic loaded to the Quantum PLCs for the benchmark tests.

The graph below shows the scan times for the three PLCs. Notice that EMTH performance on the CPU113 and CPU213 is identical; this is because EMTH does not utilize the math coprocessor available on the CPU213. Equation Network performance, which does use a math coprocessor when it is available, improves by 15% in the CPU213 over the CPU113.



Note: The Equation Network approach provides a more accurate result than the interpolated math implemented in EMTH operations.

Note: Equation Network operations yield even better performance versus EMTH operations as the equations become more complex.

Chapter 9
DX Move Instructions

- DX Move Operations
- R→T
- T→R
- T→T
- FIN
- FOUT
- SRCH
- BLKM
- BLKT
- TBLK
- IBKW
- IBKR

9.1 DX Move Operations

DX MOVE instructions copy registers or 16-bit words of data from one memory area in the PLC to another. The copied data can then be operated on, and the original data remain intact.

9.1.1 DX Tables

A group of contiguous 16-bit registers is called a *table* . The minimum table length is 1—i.e., one register. The maximum table length depends on the instruction and on the kind of CPU (16- or 24-bit) the PLC is using.

9.1.2 Specifying Discrete References in a DX Table

Groups of contiguous discretes can also be placed a DX table. Each word contains 16 contiguous 0*x* or 1*x* discrete references.

When implementing discretes in a DX table, specify the first 0*x* or 1*x* reference number of the sequence in the appropriate node of the DX instruction. The specified—i.e., displayed—reference number must be of the *first of 16* type—00001, 10001, 00017, 10017, 00033, 10033, ... , etc. When you specify the length of the table, the remaining discrete references will be automatically implied as part of the table. The references will be ordered sequentially in groups of 16 per word.

For example, if you specify 10001 as the displayed reference in a source or destination table register and you specify a length of 10 words as the length of that table, the PLC will place discrete input references 10001 ... 10160 in ten contiguous words in that table. Word 1 would contain references 10001 ... 10016, word 2 would contain references 10017 ... 10032, etc., up to word 10, which would contain references 10145 ... 10160.

9.1.3 Pointers in a DX Instruction Node

Some DX move functions use a register to indicate which table position the relevant data has been copied from or moved to. This register is called a *pointer* . The pointer value must never exceed the table length. Zero is a valid pointer value, typically indicating that the next operation of the function block will be to copy data from or read data to the first table position.

9.2 R→T

The R→T instruction copies the bit pattern of a register or of a string of contiguous discretes stored in a word into a specific register located in a table. It can accommodate the transfer of one register/word per scan.

9.2.1 Characteristics

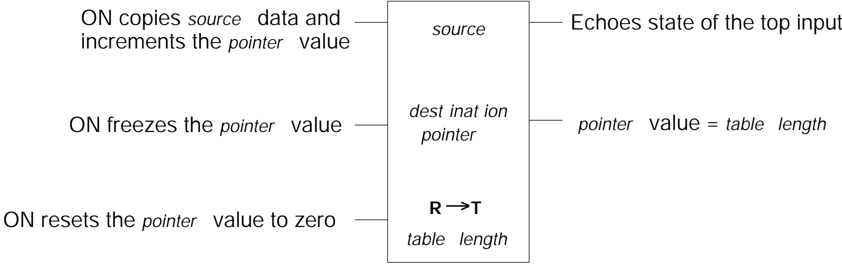
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
1C hex

9.2.2 Representation

Block Structure



Inputs
R→T has three control inputs. The input to the top node initiates the DX move operation. When the input to the middle node goes ON, the current value stored in the *destination pointer* register is frozen while the DX operation continues. This causes new data being copied to the *destination* to overwrite the data copied on the previous scan.

When the input to the bottom node goes ON, the value in the *destination pointer* register is reset to zero. This causes the next DX move operation to copy *source* data into the first register in the *destination* table.

Outputs

R→T can produce two possible outputs, from the top and middle nodes. The state of the output from the top node echoes the state of the top input. The output from the middle node goes ON when the value in the *destination pointer* register equals the specified *table length*. At this point, the instruction cannot increment any further.

Top Node Content

The *source* data to be copied in the current scan is specified in the top node. It can be:

- ☐ The first 0*x* in a string of 16 contiguous coils or discrete outputs
- ☐ The first 1*x* in a string of 16 contiguous discrete inputs
- ☐ The first 3*x* in a block of contiguous input registers
- ☐ The first 4*x* in a block of contiguous holding registers

Middle Node Content

The 4*x* register entered in the middle node is a *pointer* to the *destination* table where *source* data will be copied in the scan. The first register in the *destination* table is the next contiguous 4*x* register following the *pointer*—i.e., if the *pointer* register is 40027, then the *destination* table begins at register 40028.

The value posted in the *pointer* register indicates the register in the *destination* table where the *source* data will be copied. A value of zero indicates that the *source* data will be copied to the first register in the destination table; a value of 1 indicates that the *source* data be copied to the second register in the destination table; etc.

Bottom Node Content

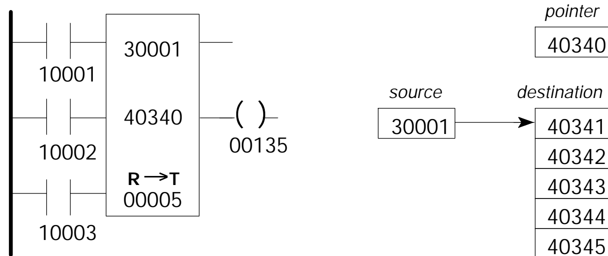
The integer value entered in the bottom node specifies *table length*—i.e., the number of registers in the *destination* table. The *table length* value can range from 1 ... 255 in 16-bit CPUs and from 1 ... 999 in 24-bit CPUs.

The value posted in the *destination pointer* register cannot be larger than the *table length* integer specified in this node.

9.2.3 An R→T Example

In the ladder logic example below, suppose initially that contact 10001 (the control input to the top node) is passing power on each scan while

contacts 10002 and 10003 (the control inputs to the middle and bottom nodes) are de-energized.



At the beginning of the first scan, the value in the *pointer* register (40340) is zero, indicating that the bit pattern in the *source* register will be copied to the first register in the destination table. On the first scan with contact 10001 energized, the bit pattern in *source* register 30001 is copied to register 40341 and the value in the *pointer register* is incremented to 1. On the second scan with 10001 energized, the contents of *source* register 30001 are copied to register 40342 (the second register in the destination table) and the value in the *pointer register* is incremented to 2.

This DX operation continues through five scans of the energized contact. At the fifth scan, which copies the contents of 30001 to register 40345 and increments the *pointer* value to the *table length*, the middle output passes power, energizing coil 00135.

Note: No further R→T operations are possible while the two values are equal, and the middle output continues to pass power regardless of the state of the input.

Now let's consider what happens when the control input to the middle or bottom node passes power. If, after the second scan, contact 10002 were to be energized, the *pointer* value would be frozen at 2. In this case, all subsequent scans of 10001 would cause the contents of *source* register 30001 to be copied to destination register 40343.

If contact 10003 were to be energized at any time, the value in the *pointer register* would be reset to zero and the contents of *source* register 30001 would be copied to destination register 40341 in the subsequent scan with contact 10001 energized.

9.3 T→R Move

The T→R instruction copies the bit pattern of a register or 16 contiguous discretes in a table to a specific holding register. It can accommodate the transfer of one register per scan. It has three control inputs and produces two possible outputs.

9.3.1 Characteristics

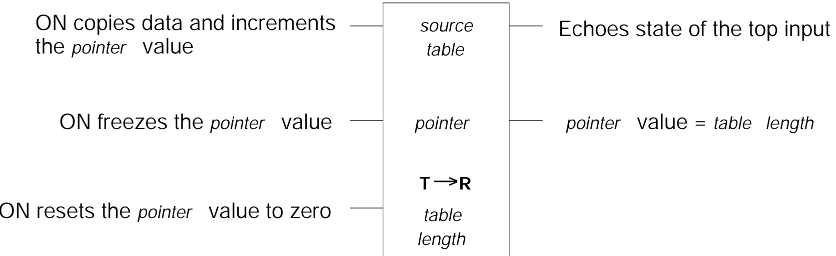
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
3C hex

9.3.2 Representation

Block Structure



Inputs
T→R has three control inputs. The input to the top node initiates the DX move operation.

When the input to the middle node goes ON, the current value stored in the *pointer* register is frozen while the DX operation continues. This causes the same table data to be written to the *destination* register on each scan.

When the input to the bottom node goes ON, the value in the *pointer* is reset to zero. This causes the next DX move operation to copy the first *destination* register in the table.

Outputs
T→R can produce two possible outputs, from the top and middle nodes. The state of the output from the top node echoes the state of the top input. The output from the middle node goes ON when the value in the *pointer* register equals the specified *table length*. At this point, the instruction cannot increment any further.

Top Node Content
The top node references the first register or discrete reference in the *source table*. A register or string of contiguous discretes from this table will be copied in a scan in a table-to-register operation. The displayed reference in this node can be:

- The first 0*x* reference in a table of coils or discrete outputs
- The first 1*x* reference in a table of discrete inputs
- The first 3*x* register in a table of input registers
- The first 4*x* register in a table of holding registers

Middle Node Content
The 4*x* register entered in the middle node is a *pointer* to the *destination* where the *source* data will be copied. The *destination* register is the next contiguous 4*x* register after the *pointer*. For example, if the middle node displays a *pointer* of 40100, then the *destination* register for the T→R copy is 40101.

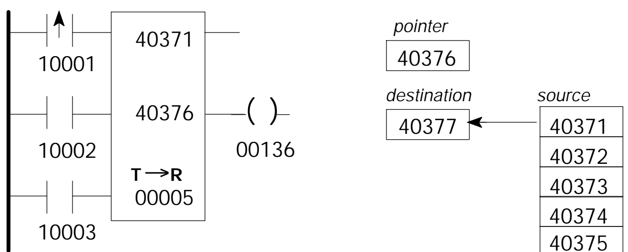
The value stored in the *pointer* register indicates which register in the *source table* will be copied to the destination register in the current scan. A value of 0 in the *pointer* indicates that the bit pattern in the first register of the *source table* will be copied to the destination; a value of 1 in the *pointer* register indicates that the bit pattern in the second register of the *source table* will be copied to the destination register; etc.

Bottom Node Content
The integer value entered in the bottom node specifies the *length* of the *source table*—i.e., the number of registers that may be copied. It is in the range 1 ... 255 in 16-bit CPUs and 1 ... 999 in 24-bit CPUs.

9.3.3 A T→R Example

In the ladder logic example below, suppose initially that contact 10001 (the control input to the top node) is passing power on each scan while

contacts 10002 and 10003 (the control inputs to the middle and bottom nodes) are de-energized.



At the beginning of the first scan, the value in the *pointer* register (40376) is zero, indicating that the bit pattern in the *source table* will be copied to the destination register. The first transition of P.T. contact 10001 copies the contents of *source register* 40371 to *destination* register 40377 and increments the value in the *pointer* to 1. The second transition of contact 10001 copies the contents of *source register* 40372 to *destination* register 40377 and increments the value in the *pointer register* to 2. This continues for five scans.

When the fifth transition of contact 10001 copies the contents of register 40375 to *destination* register 40377, the *pointer* value to increments to 5. Because the *pointer* value now equals the *table length* , the middle output passes power, energizing coil 00136.

☞ **Note:** No further T→R operations are possible while the two values are equal, and the middle output continues to pass power regardless of the state of the input.

Now let's consider what happens when the control input to the middle or bottom node passes power. If, after the second transition of contact 10001, contact 10002 were to be energized, the *pointer* value would be frozen at 2. In this case, all subsequent transitions of 10001 would cause the contents of *source* register 40373 to be copied to destination register 40377.

If contact 10003 were to be energized at any time, the value in the *pointer* would be reset to zero, and the next transition of contact 10001 would copy the contents of *source register* 40371 to destination register 40377.

9.4 T→T Move

The T→T instruction copies the bit pattern of a register or of 16 discretes from a position within one table to an equivalent position in another table of registers. It can accommodate the transfer of one register per scan. It has three control inputs and produces two possible outputs.

9.4.1 Characteristics

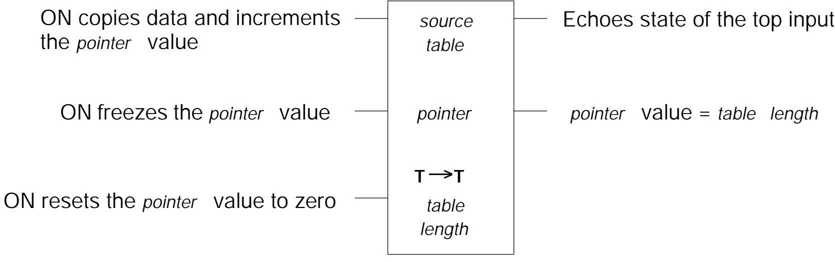
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
5C hex

9.4.2 Representation

Block Structure



Inputs
T→T has three control inputs. The input to the top node initiates the DX move operation.

When the input to the middle node goes ON, the current value stored in the *pointer* register is frozen while the DX operation continues. This causes new data being copied to the *destination* to overwrite the data copied on the previous scan.

When the input to the bottom node goes ON, the value in the *pointer* register is reset to zero. This causes the next DX move operation to copy *source* data into the first register in the *destination* table.

Outputs

T→T can produce two possible outputs, from the top and middle nodes. The state of the output from the top node echoes the state of the top input. The output from the middle node goes ON when the value in the *pointer* register equals the specified *table length*. At this point, the instruction cannot increment any further.

Top Node Content

The top node references the first register or discrete reference in the *source table*. A register or string of contiguous discretes from this table will be copied in a scan in a table-to-register operation. The displayed reference in this node can be:

- The first 0*x* in a *source table* of coils or discrete outputs
- The first 1*x* in a *source table* of discrete inputs
- The first 3*x* in a *source table* of input registers
- The first 4*x* in a *source table* of holding registers

Middle Node Content

The 4*x* register entered in the middle node is a *pointer* into both the *source* and *destination* tables, indicating where the data will be copied from and to in the current scan. The first register in the *destination* table is the next contiguous 4*x* register following the *pointer*. For example, if the middle node displays a a *pointer* reference of 40100, then the first register in the *destination* table is 40101.

The value stored in the *pointer* register indicates which register in the *source table* will be copied to which register in the *destination* table. Since the *length* of the two tables is equal and T→T copy is to the equivalent register in the *destination* table, the current value in the *pointer* register also indicates which register in the *destination* table the *source* data will be copied to.

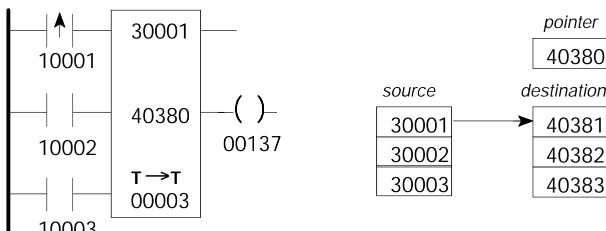
A value of 0 in the *pointer* register indicates that the bit pattern in the first register of the *source table* will be copied to the first register of the *destination* table; a value of 1 in the *pointer* register indicates that the bit pattern in the second register of the *source table* will be copied to the second register of the *destination* register; etc.

Bottom Node Content

The integer value in the bottom node specifies the *table length* of both the *source* and destination tables, since the two tables must be equal in length. *Table length* may range from 1 ... 255 in 16-bit CPUs and 1 ... 999 in 24-bit CPUs.

9.4.3 A T→T Example

In the ladder logic example below, suppose initially that contact 10001 (the control input to the top node) is passing power on each scan while contacts 10002 and 10003 (the control inputs to the middle and bottom nodes) are de-energized.



At the beginning of the first scan, the value in the *pointer* register (40380) is zero, indicating that the bit pattern in the first register in the *source table* will be copied to the first register in the destination table. The first transition of P.T. contact 10001 copies the bit pattern in *source* register 30001 to destination register 40381, then increments the value in the *pointer register* to 1. The second transition of 10001 copies the contents of *source* register 30002 to destination register 40382 and increments the value in the *pointer register* to 2. The third transition of contact 10001 copies the contents of 30003 to register 40383 and increments the *pointer* value to 3 (the *table length*). At this point, the middle output passes power and energizes coil 00137.

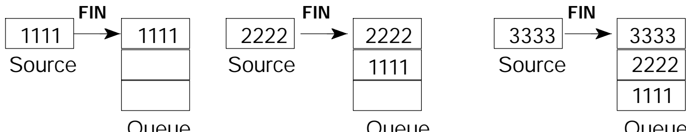
Note: No further T→T operations are possible while the two values are equal, and the middle output continues to pass power regardless of the state of the input.

Now let's consider what happens when the control input to the middle or bottom node passes power. If, after the second transition of contact 10001, contact 10002 were to be energized, the value in the *pointer register* would be frozen at 2, and all subsequent transitions of contact 10001 would cause the value in *source* register 30003 to be copied to destination register 40383.

If contact 10003 were to be energized at any time, the value in the *pointer register* would be reset to zero, and the next transition of contact 10001 would copy the contents of *source register* 30001 to destination register 40381.

9.5 FIN

The FIN instruction is used to produce a first-in queue. It copies the *source data* from the top node to the first register in a queue of holding registers. The *source* data is always copied to the register at the top of the queue. When a queue has been filled, no further *source* data can be copied to it.



An FOUT instruction (see page 184) needs to be used to clear the register at the bottom of the queue.

An FIN instruction has one control input and can produce three possible outputs.

9.5.1 Characteristics

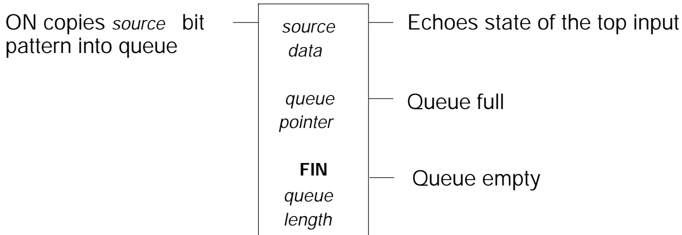
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
9C hex

9.5.2 Representation

Block Structure



Input

FIN has one control input, to the top node. When this input passes power, it initiates the FIN operation.

Outputs

FIN can produce three possible outputs. The output from the top node echoes the state of the top input.

The output from the middle node goes ON when the queue is full. No more *source* data can be copied to the queue when this output is ON.

The output from the bottom node is ON whenever the queue is empty—i.e., the value posted in the *queue pointer* register is zero.

Top Node Content

The *source data* indicated in the top node will be copied to the top of the destination queue in the current logic scan. The *source data* may be referenced by:

- The first 0*x* reference in a string of 16 contiguous coils or discrete outputs
- The first 1*x* reference in a string of 16 contiguous discrete inputs
- A 3*x* input register
- A 4*x* holding register

Middle Node Content

The 4*x* register entered in the middle node is a *queue pointer*. The first register in the queue is the next contiguous 4*x* register following the *pointer*. For example, if the middle node displays a a *pointer* reference of 40100, then the first register in the queue is 40101.

The value posted in the *queue pointer* equals the number of registers in the queue that are currently filled with *source data*. The value of the *pointer* cannot exceed the integer maximum *queue length* value specified in the bottom node.

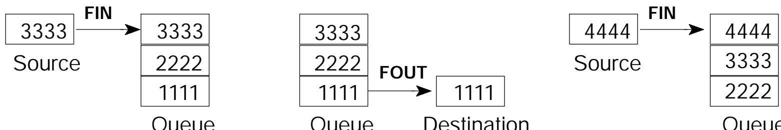
If the value in the *queue pointer* equals the integer specified in the bottom node, the middle output passes power and no further *source data* can be written to the queue until an FOUT instruction clears the register at the bottom of the queue.

Bottom Node Content

The integer value entered in the bottom node specifies the *queue length*—i.e., the number of 4*x* registers in the destination queue. The *length* can range from 1 ... 100.

9.6 FOUT

The FOUT instruction works together with the FIN instruction to produce a first in-first out (FIFO) queue. It moves the bit pattern of the holding register at the bottom of a full queue to a *destination* register or to word that stores 16 discrete outputs.



Tip The FOUT instruction should be placed before the FIN instruction (see page 181) in the ladder logic FIFO to ensure removal of the oldest data from a full queue before the newest data is entered. If the FIN block were to appear first, any attempts to enter the new data into a full queue would be ignored.

An FOUT instruction has one control input and can produce three possible outputs.

Warning! FOUT will override any disabled coils within a *destination* register without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can change as a result of the FOUT operation.

9.6.1 Characteristics

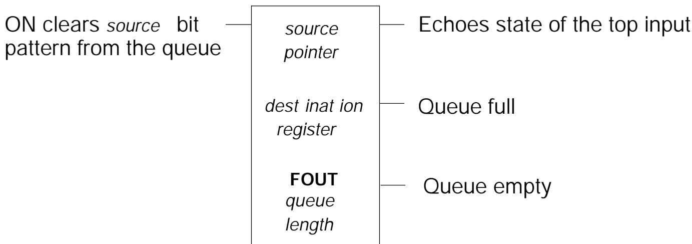
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
BC hex

9.6.2 Representation

Block Structure



Input

FOUT has one control input, to the top node. When this input passes power, it initiates the FOUT operation.

Outputs

FOUT can produce three possible outputs. The output from the top node echoes the state of the top input. The output from the middle node goes ON when the queue is full; no more *source* data can be copied to the queue when this output is ON. The output from the bottom node is ON when the queue is empty—i.e., when the value in the *queue pointer* register is zero.

Top Node Content

In the FOUT instruction, the *source data* comes from the 4*x* register at the bottom of a full queue. The next contiguous 4*x* register following the *source pointer* register in the top node is the first register in the the queue. For example, if the top node displays *pointer* register 40100, then the first register in the queue is 40101.

The value posted in the *source pointer* equals the number of registers in the queue that are currently filled. The value of the *pointer* cannot exceed the integer maximum *queue length* value specified in the bottom node. If the value in the *source pointer* equals the integer specified in the bottom node, the middle output passes power and no further FIN data can be written to the queue until the FOUT instruction clears the register at the bottom of the queue to the *destination register* .

Middle Node Content

The destination specified in the middle node can be a 0*x* reference or 4*x* register. When the queue has data and the top control input to the FOUT passes power, the *source data* is cleared from the bottom register in the queue and is written to the *destination register*.

Bottom Node Content

The integer in the bottom node specifies the *queue length*—i.e., the number of 4*x* registers in the queue. The *length* can range from 1 ... 100.

9.7 SRCH

The SRCH instruction searches the registers in a *source table* for a specific bit pattern. The function will search the entire *source table* in a single scan until either a match is found or the end-of-table is reached.

9.7.1 Characteristics

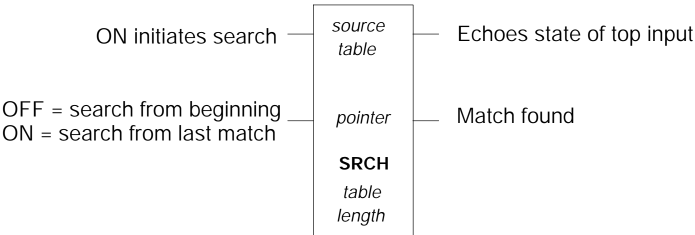
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
DC hex

9.7.2 Representation

Block Structure



Inputs
SRCH has two control inputs (to the top and middle nodes). The input to the top node initiates the SRCH operation. The state of the input to the middle node indicates where the SRCH operation will originate.

Outputs
SRCH can produce up to two outputs. The state of the output from the top node echoes the state of the top input. Power passed from the middle node indicates that the bit pattern being searched for has been found in the *source table* .

Top Node Content
The top node specifies the *source table* to be searched. The node may reference:

- The first 3*x* reference in a table of input registers
- The first 4*x* reference in a table of holding registers

Middle Node Content

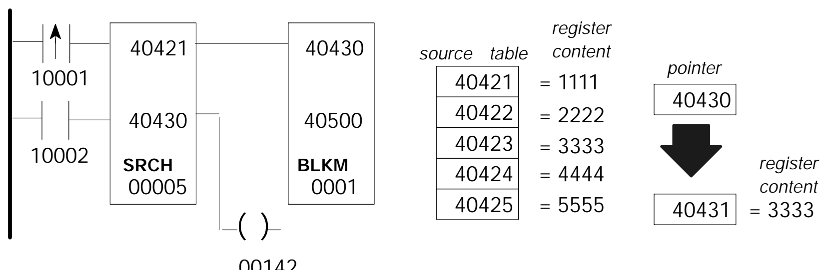
The 4*x* register entered in the middle node is the *pointer* into the *source table*. It points to the *source* register that contains the same value as the value stored in the next contiguous register after the pointer—e.g, if the *pointer* register is 40015, then register 40016 contains a value that the SRCH instruction will attempt to match in *source table*.

Bottom Node Content

The integer value entered in the bottom node specifies the *table length* —i.e., the number of registers in the *source table.* The *length* can range from 1 ... 100.

9.7.3 A SRCH Example

In the following example, we search a *source table* that contains five registers (40421 ... 40425) for a specific bit pattern. The pointer register (40430) indicates that the desired bit pattern is stored in register 40431 and we see that that register contains a bit value of 3333.



In each scan where P.T. contact 10001 transitions from OFF to ON, the *source* table is searched for a bit pattern equivalent to the value 3333. When the match is found, the middle output passes power to coil 00142.

If N.O. contact 10002 is OFF when the match is found at register 40423, the SRCH instruction energizes coil 00142 for one scan, then starts the search again in the next scan at the top of the *source table* (register 40421). If contact 10002 is ON, the SRCH instruction energizes coil 00142 for one scan, then starts the search in register 40424.

Because the top input is a P.T. contact, on any scan where power is not applied to the top input the pointer value is cleared. We use a BLKM instruction here to save the pointer value to register 40500.

9.8 BLKM

The BLKM (block move) instruction copies the entire contents of a *source table* to a *destination table* in one scan.

Warning! BLKM will override any disabled coils within a *destination table* without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can change as a result of the BLKM instruction.

9.8.1 Characteristics

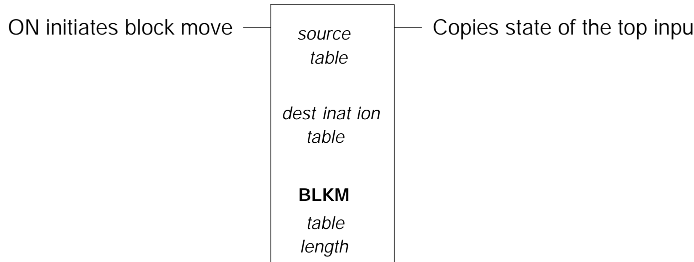
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
7C hex

9.8.2 Representation

Block Structure



Input
BLKM has one control input (to the top node). This input initiates the DX move operation.

Output
BLKM produces one output (from the top node), which echoes the state of the top input.

Top Node Content

The top node specifies the *source table* that will have its contents copied in the block move. The node may reference:

- The first 0*x* reference in a table of contiguous coils or discrete outputs
- The first 1*x* reference in a table of contiguous discrete inputs
- The first 3*x* reference in a table of contiguous input registers
- The first 4*x* reference in a table of contiguous holding registers

Middle Node Content

The middle node specifies the *destination table* where the contents of the *source table* will be copied in the block move. The node may reference:

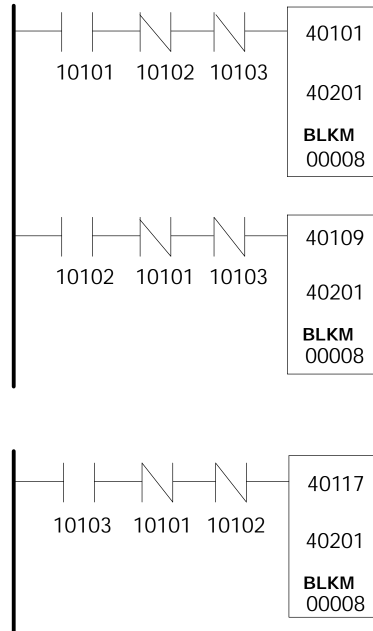
- The first 0*x* reference in a table of contiguous coils or discrete outputs
- The first 4*x* reference in a table of contiguous holding registers

Bottom Node Content

The integer value entered in the bottom node specifies the *table size* —i.e., the number of registers or 16-bit words—for both the *source* and *destination* tables; they are of equal length. The *table length* can range from 1 ... 100.

9.8.3 A Recipe Storage Example

You can use ladder logic to write specific process programs (or *recipes),* store each in a unique table, then write a general process program and store it in another working table. The recipe tables must be structured with similar information in corresponding registers—if a heating temperature is in the third register in one recipe table, it should be in the third register in all recipe tables. Recipes can be pulled into the generic process program with BLKM instructions:



The process is controlled with three input switches—10101, 10102, and 10103. To run process A, turn on 10101, and leave 10102 and 10103 off. When input 10101 is energized, it passes power through normally closed contacts 10102 and 10103. A BLKM instruction moves the recipe for process A from registers 40101 ... 40108 to registers 40201 ... 40208. This table of registers is a working table, with each register controlling a part of the general process. By using one working table, you can control the output for three separate processes with only one program.

9.9 BLKT

The BLKT (block-to-table) instruction combines the functions of R→T and BLKM in a single instruction. In one scan, it can copy data from a *source block* to a *destination* block in a table. The *source block* is of a fixed *length*. The block within the table is of the same *length*, but the overall length of the table is limited only by the number of registers in your system configuration.

9.9.1 Characteristics

Size
Three nodes high

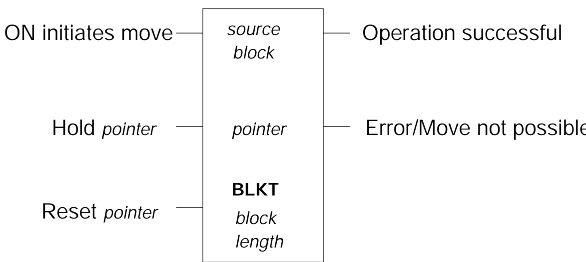
PLC Compatibility

- Standard in 110CPU512 and 110CPU612 Micro PLCs, in all Quantum Automation Series PLCs, and in all Slot Mount and Compact PLC models
- Available as a loadable for 984A, 984B, and 984X Chassis Mount PLCs
- Not available in other PLC models

Opcode
9F hex

9.9.2 Representation

Block Structure



Inputs
BLKT has three control inputs. The input to the top node initiates the DX move operation. The inputs to the middle and bottom node can be

used to control the *pointer* so that *source* data is not copied into registers that are needed for other purposes in the logic program.

Warning! BLKT is a powerful instruction that can corrupt all the 4*x* registers in your PLC with data copied from the *source block*.. You should use external logic in conjunction with the middle or bottom input to confine the value in the *pointer* to a safe range.

When the input to the middle node is ON, the value in the *pointer* register is frozen while the BLKT operation continues. This causes new data being copied to the *destination* to overwrite the block data copied on the previous scan.

When the input to the bottom node is ON, the value in the *pointer* register is reset to zero. This causes the BLKT operation to copy *source* data into the first block of registers in the *destination* table.

Outputs

BLKT can produce one of two possible outputs. When the move is successful, power is passed to the output from the top node. If an error occurs in the operation, power is passed to the output from the middle node.

Top Node Content

The 4*x* register entered in the top node is the first holding register in the *source block*—i.e, the block of contiguous registers whose content will be copied to a block of registers in the *destination* table.

Middle Node Content

The 4*x* register entered in the middle node is the *pointer* to the *destination* table. The first register in the *destination* table is the next contiguous register after the *pointer*— e.g., if the *pointer* register is 40107, then the first register in the *destination* table is 40108.

Note: The *destination* table is segmented into a series of register blocks, each of which is the same length as the *source block*. Therefore, the size of the *destination* table is a multiple of the *length* of the *source block*, but its overall size is not specifically defined in the instruction. If left uncontrolled, the *destination* table could consume all the 4*x* registers available in the PLC configuration.

The value stored in the *pointer* register indicates where in the *destination* table the *source* data will begin to be copied. This value specifies the block number within the *destination* table.

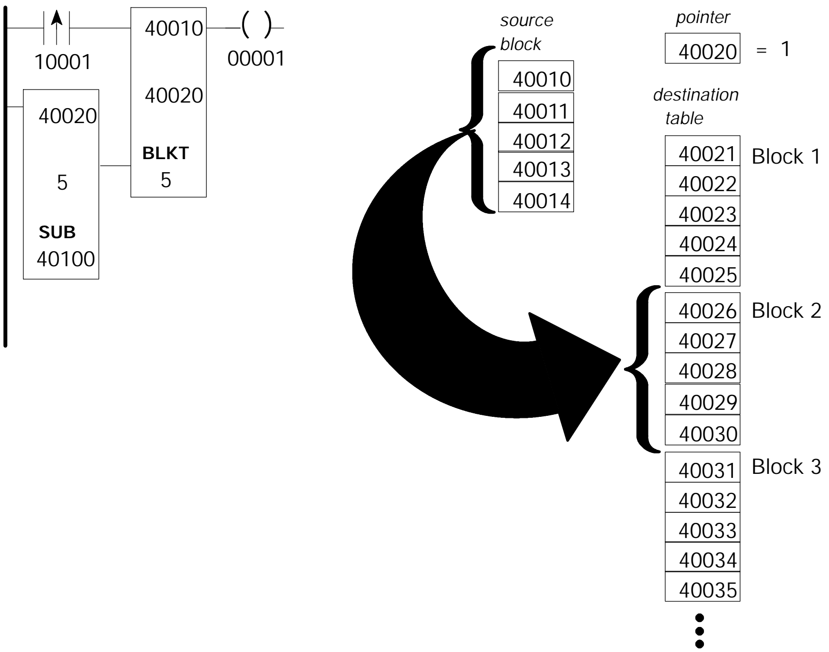
Bottom Node Content

The integer value entered in the bottom node specifies *block length* —i.e., the number of 4*x* registers—of the *source* block (and of the *destination* block). The valid range is from 1 ... 100.

9.9.3 A BLKT Example

Below is an example of a BLKT operation. The *source block* is five registers long (40010 ... 40014). The *destination* table starts at register 40021 and is segmented into a string of five-register blocks (40021 ... 40025, 40026 ... 40030, etc.).

In the illustration below, we see the what happens on the second transition of P.T. contact 10001. The value inside the *pointer* (register 40020) increments to 1, and the data contained in the *source block* registers is copied into the second block in the *destination* table (registers 40026 ... 40030). Coil 00001 goes ON when the BLKT move is complete.



The SUB instruction in the ladder logic is used to control the use of registers in the *destination* table. Here we restrict the table to 25 registers by clearing the value in the *pointer* register to zero after five BLKT transfers.

9.10 TBLK

The TBLK (table-to-block) instruction combines the functions of T→R and the BLKM in a single instruction. In one scan, it can copy up to 100 contiguous 4*x* registers from a table to a *destination block*. The *destination block* is of a fixed *length*. The block of registers being copied from the *source table* is of the same *length*, but the overall length of the *source table* is limited only by the number of registers in your system configuration.

9.10.1 Characteristics

Size

Three nodes high

PLC Compatibility

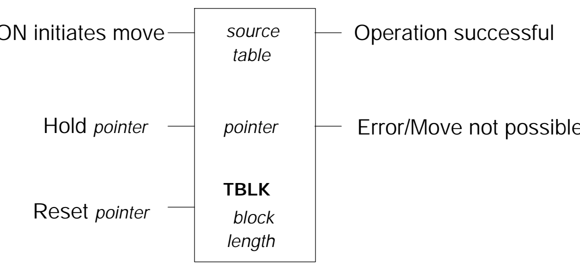
- Standard in 110CPU512 and 110CPU612 Micro PLCs, all Quantum Automation Series PLCs, and all Slot Mount and Compact PLC models
- Available as a loadable for 984A, 984B, and 984X Chassis Mount PLCs
- Not available in other PLC models

Opcode

DF hex

9.10.2 Representation

Block Structure



Inputs

TBLK has three control inputs. The input to the top node initiates the DX move operation. The inputs to the middle and bottom node can be used to control the value in the *pointer* so that size of the *source table* can be controlled.

Warning! You should use external logic in conjunction with the middle or bottom input to confine the value in the *destination pointer* to a safe range.

When the input to the middle node is ON, the value in the *pointer* register is frozen while the TBLK operation continues. This causes the same *source* data block to be copied to the *destination table* on each scan.

When the input to the bottom node is ON, the *pointer* value is reset to zero. This causes the TBLK operation to copy data from the first block of registers in the *source table.*

Outputs

TBLK can produce one of two possible outputs. When the move is successful, power is passed to the output from the top node. If an error occurs in the operation, power is passed to the output from the middle node.

Top Node Content

The 4*x* register entered in the top node is the first holding register in the *source table.*

Note: The *source table* is segmented into a series of register blocks, each of which is the same length as the *destination* block. Therefore, the size of the *source table* is a multiple of the *length* of the *destination* block, but its overall size is not specifically defined in the instruction. If left uncontrolled, the *source table* could consume all the 4*x* registers available in the PLC configuration.

Middle Node Content

The 4*x* register entered in the middle node is the *pointer* to the *source* block. The first register in the *destination* block is the next contiguous register after the *pointer* . For example, if the *pointer* is register 40107, then the first register in the *destination* block is 40108.

The value stored in the *pointer* indicates which block of data from the *source table* will be copied to the *destination* block. This value specifies a block number within the *source table* .

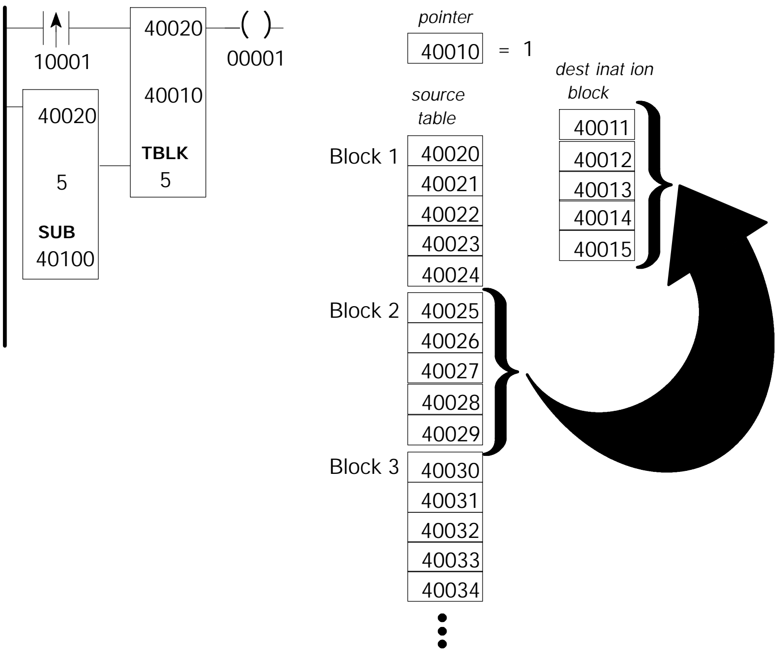
Bottom Node Content

The integer value entered in the bottom node specifies *block length* —i.e., the number of 4*x* registers—of the *destination* block (and of the blocks within the *source table*). The valid range is from 1 ... 100.

9.10.3 A TBLK Example

Below is an example of a TBLK operation. The *destination block* is five registers long (40011 ... 40015). The *source table* starts at register 40020 and is segmented into a series of five-register *source* blocks (40020 ... 40024, 40025 ... 40029, etc.).

In the illustration below, we see the what happens on the second transition of P.T. contact 10001. The value inside the *pointer* register increments to 1, and the data contained in the second *source* block (registers 40025 ... 40029) is copied into the five-register *destination* block (40011 ... 40015). Coil 00001 goes ON when the TBLK move is complete.



The SUB instruction in the ladder logic is used to control the use of registers in the *source table* . Here we restrict the table to 25 registers by clearing the value in the *pointer* register to zero after five TBLK transfers.

9.1 1 IBKR

The IBKR (indirect block read) instruction lets you access non-contiguous registers dispersed throughout your application and copy the contents into a *destination* block of contiguous registers. This instruction can be used with subroutines or for streamlining data access by host computers or other PLCs.

9.1 1.1 Characteristics

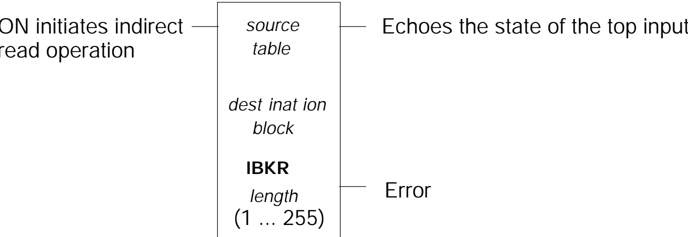
Size
Three nodes high

PLC Compatibility
- Standard in all Quantum Automation Series PLCs
- Not available in other PLC types

Opcode
51 hex

9.1 1.2 Representation

Block Structure



Input
IBKR has one control input (to the top node), which initiates the operation.

Outputs
IBKR produces two possible outputs (from the top and bottom nodes). The output from the top node echoes the state of the top input. Power is passed to the output from the bottom node if there is an error in the *source table* —e.g., if the *source* register does not exist.

Top Node Content

The 4*x* register entered in the top node is the first holding register in a *source table*. The registers in this table contain values that are pointers to the non-contiguous registers you want to collect in the operation.

Middle Node Content

The 4*x* register entered in the middle node is the first in a block of contiguous *destination* registers—i.e., the block to which the *source* data will be copied.

Bottom Node Content

The integer value entered in the bottom node indicates the *length* —i.e., the number of registers—in the *source table* (and the *destination block*).

9.1 1.3 An IBKR Example

Say you want to collect the data stored in the following five registers dispersed throughout the logic program and read the data into a contiguous block where it can be read by a host computer in a single instruction:

Register		Content
400014	=	200
400199	=	600
400337	=	400
400841	=	1000
401061	=	800

You can create a *source table* with five holding registers by specifying the first register (400100) in the top node and specifying a *length* of 5 in the bottom node:

Enter a value in each register in the table that points to the registers above:

Source Register		Content (pointer)
400100	=	14
400101	=	199
400102	=	337
400103	=	841
400104	=	1061

The register entered in the middle node (400001) is the first register in the *destination block*. The IBKR instruction loads the *destination block* as follows:

Destination Register		Content
400001	=	200
400002	=	600
400003	=	400
400004	=	1000
400005	=	800

9.12 IBKW

The IBKW (indirect block write) instruction lets you copy the data from a table of contiguous registers into several non-contiguous registers dispersed throughout your application.

9.12.1 Characteristics

Size
Three nodes high

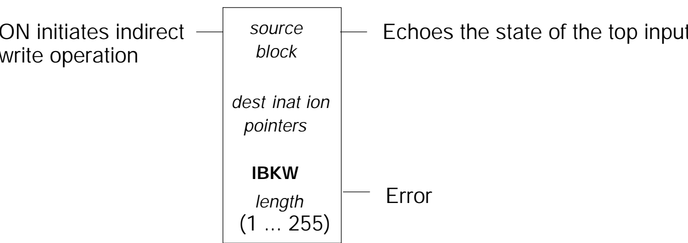
PLC Compatibility

- Standard in all Quantum Automation Series PLCs
- Not available in other PLC types

Opcode
52 hex

9.12.2 Representation

Block Structure



Input
IBKW has one control input (to the top node), which initiates the operation.

Outputs
IBKW produces two possible outputs (from the top and bottom nodes). The output from the top node echoes the state of the top input. Power is passed to the output from the bottom node if there is an error in the *destination table* .

Top Node Content
The 4*x* register entered in the top node is the first in a block of *source* registers. The registers in this block contain values that will be copied to non-contiguous registers dispersed throughout the logic program.

Middle Node Content
The 4*x* register entered in the middle node is the first in a block of contiguous *destination pointer* registers. Each of these registers contains a value that points to the address of a register where the *source* data will be copied.

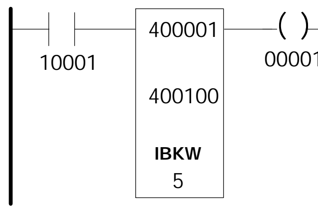
Bottom Node Content
The integer value entered in the bottom node indicates the *length* —i.e., the number of registers—in the *source block* (and the *destination pointer* block).

9.12.3 An IBKW Example

Say you have a block of five contiguous registers (400001 ... 400005) that contain *source* data:

Destination Register		Content
400001	=	200
400002	=	400
400003	=	600
400004	=	800
400005	=	1000

To disperse this *source* data to non-contiguous registers in your logic program, establish a *destination pointer* block starting at register 400100 in the middle node:



Assume that the contents of the *source* registers will be copied to non-contiguous destination registers as follows:

Source Register		Destination Register
400100	=	400014
400101	=	400037
400102	=	400019
400103	=	400061
400104	=	400041

To accomplish this indirect read operation, the values inside the *destination pointer* block need to be set up as follows:

Source Register		Content (pointer)
400100	=	14
400101	=	37
400102	=	19
400103	=	61
400104	=	41

The IBKW instruction loads the *destination pointers* as follows:

Destination Pointer		Content
400014	=	200
400037	=	400
400019	=	600
400061	=	800
400041	=	1,000

Chapter 10
DX Matrix Instructions

- DX Matrix Operations
- AND
- OR
- XOR
- COMP
- CMPR
- Sensing and Modifying Bits in a Matrix
- Rotating a Bit Pattern
- How to Report Status Information
- A Simple Table Averaging Example
- Setting Step Flags and Monitoring Steps in Modsoft SFC

10.1 DX Matrix Operations

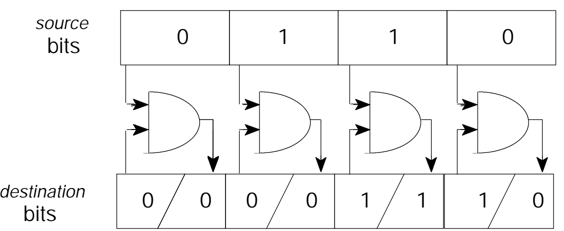
A DX matrix is a sequence of data bits formed by consecutive 16-bit words or registers derived from tables. DX matrix functions operate on bit patterns within tables.

Just as with DX move instructions, the minimum table length is 1 and the maximum table length depends on the type of DX instruction you use and on the size of the CPU 16- or 24-bit) in your PLC.

Groups of 16 discretes can also be placed in tables. The reference number used is the first discrete in the group, and the other 15 are implied. The number of the first discrete must be of the *first of 16* type—00001, 10001, 00017, 10017, 00033, 10033, ... , etc.

10.2 AND

The AND instruction performs a Boolean AND operation on the bit patterns in the *source* and *destination matrices*. The ANDed bit pattern is then posted in the *destination matrix*, overwriting its previous contents:



Warning! AND will override any disabled coils within the *destination matrix* without enabling them. This can cause personal injury if a coil has disabled an operation for maintenance or repair because the coil's state can be changed by the AND operation.

10.2.1 Characteristics

Size

Three nodes high

PLC Compatibility

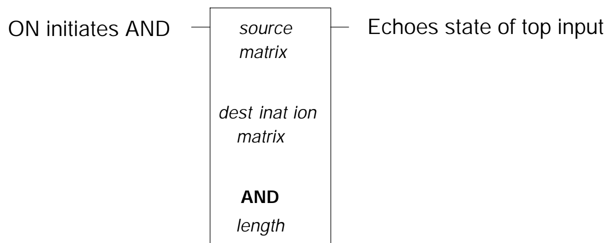
Standard in all PLC types

Opcode

1D hex

10.2.2 Representation

Block Structure



Input

AND has one control input (to its top node), which initiates the logical operation.

Output

AND produces one output (from its top node), which echoes the state of the top input.

Top Node Content

The entry in the top node is the first reference in the *source matrix*. It may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 1*x* reference in a matrix of contiguous discrete inputs
- The first 3*x* reference in a matrix of contiguous input registers
- The first 4*x* reference in a matrix of contiguous holding registers

Middle Node Content

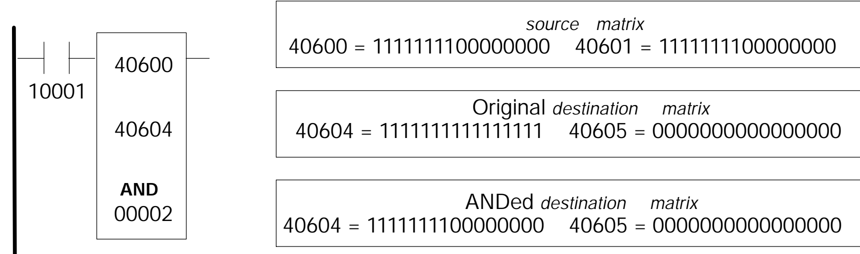
The entry in the middle node is the first reference in the *destination matrix*. It may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 4*x* reference in a matrix of contiguous holding registers

Bottom Node Content

The integer entered in the bottom node specifies the *matrix length* —i.e., the number of registers or 16-bit words in the two matrixes. The *matrix length* can be in the range 1 ... 100. A length of 2 indicates that 32 bits in each matrix will be ANDed.

10.2.3 An AND Example

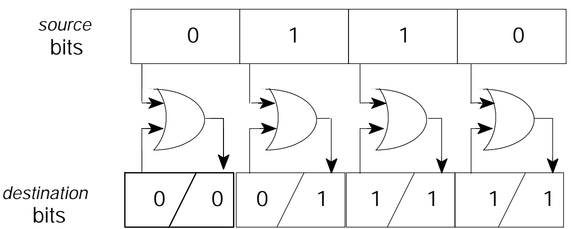


When contact 10001 passes power, the *source matrix* formed by the bit pattern in registers 40600 and 40601 is ANDed with the *destination matrix* formed by the bit pattern in registers 40604 and 40605. The ANDed bits are then copied into registers 40604 and 40605, overwriting the previous bit pattern in the *destination matrix*.

Tip: If you want to retain the original destination bit pattern of registers 40604 and 40605, copy the information into another table using a BLKM before performing the AND operation.

10.3 OR

The OR instruction performs a Boolean OR operation on the bit patterns in the *source* and *destination matrices*. The ORed bit pattern is then posted in the *destination matrix*, overwriting its previous contents:



Warning! OR will override any disabled coils within the *destination matrix* without enabling them. This can cause personal injury if a coil has disabled an operation for maintenance or repair because the coil's state can be changed by the OR operation.

10.3.1 Characteristics

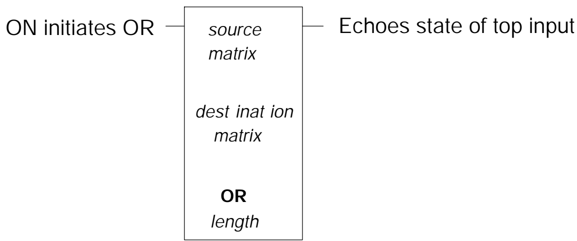
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
3D hex

10.3.2 Representation

Block Structure



Input

OR has one control input (to its top node), which initiates the logical operation.

Output

OR produces one output (from its top node), which echoes the state of the top input.

Top Node Content

The entry in the top node is the first reference in the *source matrix*. It may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 1*x* reference in a matrix of contiguous discrete inputs
- The first 3*x* reference in a matrix of contiguous input registers
- The first 4*x* reference in a matrix of contiguous holding registers

Middle Node Content

The entry in the middle node is the first reference in the *destination matrix*. It may be:

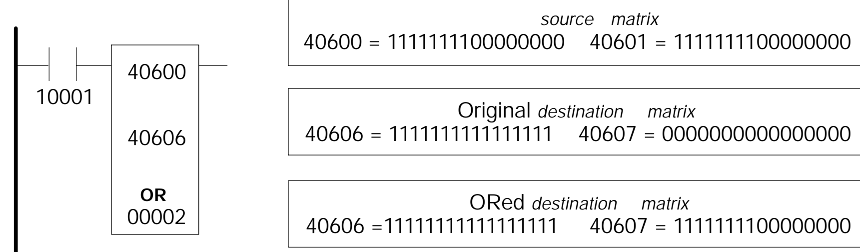
- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 4*x* reference in a matrix of contiguous holding registers

Bottom Node Content

The integer entered in the bottom node specifies the *matrix length*—i.e., the number of registers or 16-bit words in the two

matrixes. The *matrix length* can be in the range 1 ... 100. A length of 2 indicates that 32 bits in each matrix will be ORed.

10.3.3 An OR Example



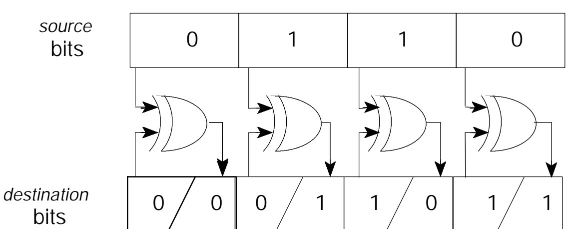
Whenever contact 10001 passes power, the *source matrix* formed by the bit pattern in registers 40600 and 40601 is ORed with the *destination matrix* formed by the bit pattern in registers 40606 and 40607. The ORed bit pattern is then copied into registers 40606 and 40607, overwriting the original destination bit pattern.

Caution: Outputs and coils cannot be turned OFF with the OR instruction.

Tip: If you want to retain the original destination bit pattern of registers 40606 and 40607, copy the information into another table using a BLKM before performing the OR operation.

10.4 XOR

The **XOR** instruction performs a Boolean Exclusive OR operation on the bit patterns in the *source* and *destination matrices*. The XORed bit pattern is then posted in the *destination matrix*, overwriting its previous contents:



Warning! XOR will override any disabled coils within the *destination matrix* without enabling them. This can cause personal injury if a coil has disabled an operation for maintenance or repair because the coil's state can be changed by the XOR operation.

10.4.1 Characteristics

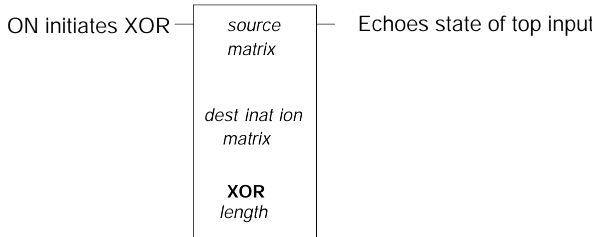
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
DD hex

10.4.2 Representation

Block Structure



Input

XOR has one control input (to its top node), which initiates the logical operation.

Output

XOR produces one output (from its top node), which echoes the state of the top input.

Top Node Content

The entry in the top node is the first reference in the *source matrix* . It may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 1*x* reference in a matrix of contiguous discrete inputs
- The first 3*x* reference in a matrix of contiguous input registers
- The first 4*x* reference in a matrix of contiguous holding registers

Middle Node Content

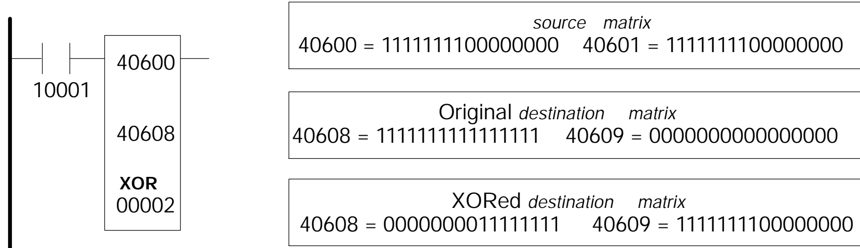
The entry in the middle node is the first reference in the *destination matrix* . It may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 4*x* reference in a matrix of contiguous holding registers

Bottom Node Content

The integer entered in the bottom node specifies the *matrix length* —i.e., the number of registers or 16-bit words in the two matrices. The *matrix length* can be in the range 1 ... 100. A length of 2 indicates that 32 bits in each matrix will be XORed.

10.4.3 An XOR Example



When contact 10001 passes power, the *source matrix* formed by the bit pattern in registers 40600 and 40601 is XORed with the *destination matrix* formed by the bit pattern in registers 40608 and 40609. The XORed bit pattern is then copied into registers 40608 and 40609, overwriting the original destination bit pattern.

Tip: If you want to retain the original destination bit pattern of registers 40608 and 40609, copy the information into another table using a BLKM before performing the XOR operation.

10.5 COMP

The COMP instruction complements the bit pattern—i.e., changes all 0's to 1's and all 1's to 0's—of a *source matrix*, then copies the complemented bit pattern into a *destination matrix*. The entire COMP operation is accomplished in one scan.

Warning! COMP will override any disabled coils in the *destination matrix* without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can be changed by the COMP operation.

10.5.1 Characteristics

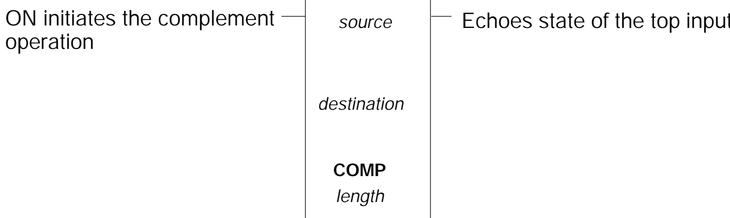
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
BD hex

10.5.2 Representation

Block Structure



Input
COMP has one control input (to its top node), which initiates the complementing operation.

Output
COMP produces one output (from its top node), which echoes the state of the top input.

Top Node Content

The entry in the top node is the first reference in the *source matrix*, which contains the original bit pattern before the complement operation. The entry may be:

- ☐ The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- ☐ The first 1*x* reference in a matrix of contiguous discrete inputs
- ☐ The first 3*x* reference in a matrix of contiguous input registers
- ☐ The first 4*x* reference in a matrix of contiguous holding registers

Middle Node Content

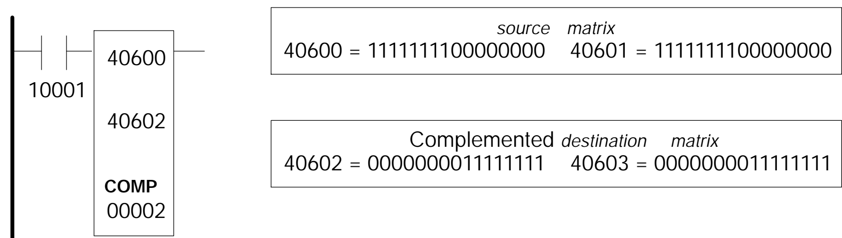
The entry in the middle node is the first reference in the *destination matrix* where the complemented bit pattern will be posted. It may be:

- ☐ The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- ☐ The first 4*x* reference in a matrix of contiguous holding registers

Bottom Node Content

The integer value entered in the bottom node specifies a *matrix length* —i.e., the number of registers or 16-bit words in the matrices. *Matrix length* can range from 1 ... 100. A length of 2 indicates that 32 bits in each matrix will be complemented.

10.5.3 A COMP Example



When contact 10001 passes power, the bit pattern in the *source matrix* (registers 40600 and 40601) is complemented, then the complemented bit pattern is posted in the *destination* matrix (registers 40602 and 40603). The original bit pattern is maintained in the *source matrix*.

10.6 CMPR

The CMPR instruction compares the bit pattern in *matrix a* against the bit pattern *matrix b* for miscompares. In a single scan, the two matrices are compared bit position by bit position until a miscompare is found or the end of the matrices is reached (without miscompares).

10.6.1 Characteristics

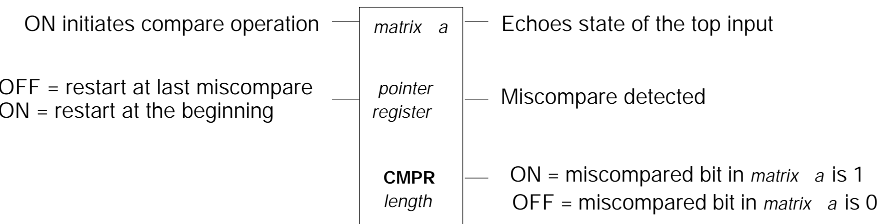
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
5D hex

10.6.2 Representation

Block Structure



Inputs
CMPR has two control inputs (to the top and middle nodes). The input to the top node initiates the comparison. The state of the input to the middle node determines the location in the logic program where the next comparison will start.

Outputs
CMPR produces three possible outputs. The output from the top node echoes the state of the top input. Power is passed to the output from the middle node when a miscompare is found. The state of the output from the bottom node indicates whether the miscompared bit in *matrix a* is a 1 or a 0.

Top Node Content

The entry in the top node is the first reference in *matrix a*, one of the two matrices to be compared. The entry may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 1*x* reference in a matrix of contiguous discrete inputs
- The first 3*x* reference in a matrix of contiguous input registers
- The first 4*x* reference in a matrix of contiguous holding registers

Middle Node Content

The *pointer register* entered middle node must be a 4*x* holding register. It is the *pointer* to *matrix b*, the other matrix to be compared. The first register in *matrix b* is the next contiguous 4*x* register following the *pointer register*.

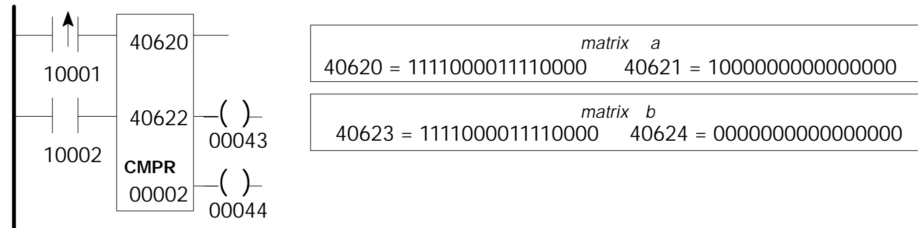
The value stored inside the *pointer register* increments with each bit position in the two matrices that is being compared. As bit position 1 in *matrix a* and *matrix b* is compared, the *pointer register* contains a value of 1; as bit position 2 in the matrices are compared, the *pointer* value increments to 2; etc.

When the outputs signal a miscompare, you can check the accumulated count in the *pointer register* to determine the bit position in the matrices of the miscompare.

Bottom Node Content

The integer value entered in the bottom node specifies a *length* of the two matrices—i.e., the number of registers or 16-bit words in each matrix. (*Matrix a* and *matrix b* have the same length.) The *matrix length* can range from 1 ... 100 —i.e., a *length* of 2 indicates that *matrix a* and *matrix b* contain 32 bits.

10.6.3 A CMPR Example



If contact 10002 is energized, the bit pattern in *matrix a* (registers 40620 and 40621) is compared against the bit pattern in *matrix b* (registers 40623 and 40624) on every scan that 10001 receives power. The comparison is done bit by bit in one scan.

The comparison proceeds without miscompares until bit 17 is reached. (Bit 17 is the leftmost bit in registers 40621 in matrix a and 40624 in matrix b.) Bit 17 in *matrix a* = 1 and in *matrix b* = 0.

At this point, pointer register 40622 has incremented to 17, where it stops and energizes coils 00043 and 00044 for one scan. An energized coil 00043 indicates that a miscompare has been found, and an energized coil 00044 indicates that the miscompared bit is a 1 in *matrix a* (and therefore a 0 in *matrix b*). By checking the value in the pointer register, we know that the miscompare is at bit position 17.

On the second transition of contact 10001, the comparison starts again at bit 1 and stops again when the value in the *pointer* increments to 17.

If contact 10002 is de-energized, the first transition of contact 10001 stops the function at 40622 = 17; 00043 and 00044 will energize for one scan. On the second transition of 10001, the function will stop at 40622 = 00; 00143 and 00044 will energize for one scan.

10.7 SENS

The SENS instruction examines and reports the sense—1 or 0—of a specific *bit location* in a *data matrix*. One *bit location* is sensed per scan.

10.7.1 Characteristics

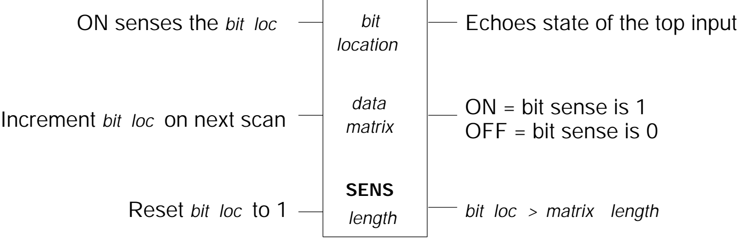
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
7D hex

10.7.2 Representation

Block Structure



Inputs
SENS has three possible control inputs. The input to the top node initiates the bit sense operation. An input to the middle node causes the *bit location* specified in the top node to increment by one on the next scan. An input to the bottom node causes the *bit location* to be reset to 1.

Outputs
SENS can produce three possible outputs. The state of the output from the top node echoes the state of the top input. The state of the output from the middle node indicates the sense of the current *bit location*. Power is passed to the output from the bottom node if an invalid *bit location* is entered in the top node.

Top Node Content

The entry top node is the specific *bit location* that you want to sense in the *data matrix*. It may be:

- Entered explicitly as an integer value in the range 1 ... 999 in a 16-bit CPU or 1 ... 9600 in a 24-bit CPU
- Stored in a 3*x* input register as a value in the range 1 ... 4080 in a 16-bit CPU or 1 ... 9600 in a 24-bit CPU
- Stored in a 4*x* holding register as a value in the range 1 ... 4080 in a 16-bit CPU or 1 ... 9600 in a 24-bit CPU

Note If the *bit location* is entered as an integer or in a 3*x* register, the instruction will ignore the state of the middle and bottom inputs.

Middle Node Content

The middle node is the first word or register in the *data matrix*. It may be:

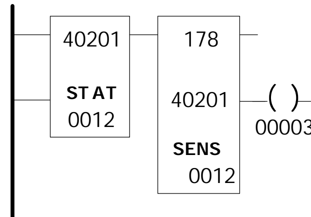
- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 4*x* reference in a matrix of contiguous holding registers

Bottom Node Content

The integer value entered in the bottom node specifies a *matrix length*—i.e., the number of 16-bit words or registers in the *data matrix*. The *length* can range from 1 ... 255 in a 16-bit CPU and from 1 ... 600 in a 24-bit CPU—e.g., a *matrix length* of 200 indicates 3200 *bit locations*.

10.7.3 A SENS Example: Reporting Status Information

By using a SENS instruction together with a STAT instruction (see Chapter 11), you can report system status information in ladder logic. In the example below, the STAT block calls 12 registers from the system status table (registers 40201 ... 40212). These 12 registers comprise the *data matrix* scanned by the SENS instruction.



The top input to the STAT block, which passes power on every scan, posts current status information from the first 12 words in the status table in registers 40201 ... 40212.

Suppose we want to check the health of the I/O module in slot 2 of drop 1, rack 1 in the I/O network. The status bit of interest happens to be the second bit in the 12th register (40212) in the status table, which will have a value of 1 if the module is healthy.

Since each bit's state represents a different piece of status information, you can use a SENS block to report the sense of the desired incoming bit. The SENS instruction views the 12 registers as a 12-by-16 matrix of bit values. In this case, the *bit location* of interest in the *data matrix* is 178 (bit 2 in register 40212).

By connecting the top output from the STAT instruction to the top input to the SENS instruction, you check the sense of bit 178 on every scan. If the SENS block passes power to coil 00003, it indicates a bit value of 1 and therefore a healthy module in slot 2 of the drop. If coil 00003 stays OFF, it indicates that the module in that slot is unhealthy.

10.8 MBIT

The MBIT instruction modifies *bit locations* within a *data matrix* —i.e., it sets the bit(s) to 1 or clears the bit(s) to 0. One *bit location* may be modified per scan.

Warning! MBIT will override any disabled coils within a *destination* group without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can change as a result of the MBIT instruction.

10.8.1 Characteristics

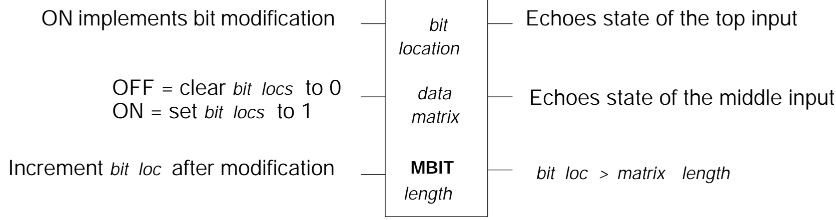
Size
Three nodes high

PLC Compatibility
Standard in all PLC types

Opcode
9D hex

10.8.2 Representation

Block Structure



Inputs
MBIT has three possible control inputs. The input to the top node initiates the bit modification. The state of the input to the middle node indicates whether MBIT will be used to set or to clear the *bit locations* in the *matrix* . An input to the bottom node causes the *bit location* specified in the top node to increment by one on the next scan.

Outputs

MBIT can produce three possible outputs. The state of the output from the top node echoes the state of the top input, and the state of the output from the middle node echoes the state of the middle input.

Power passing to the output from the bottom node indicates an error condition.

Top Node Content

The entry in the top node is the specific *bit location* that you want to set or clear in the *data matrix*. It may be:

- Entered explicitly as an integer value in the range 1 ... 999 in a 16-bit CPU or 1 ... 9600 in a 24-bit CPU
- Stored in a 3*x* input register as a value in the range 1 ... 4080 in a 16-bit CPU or 1 ... 9600 in a 24-bit CPU
- Stored in a 4*x* holding register as a value in the range 1 ... 4080 in a 16-bit CPU or 1 ... 9600 in a 24-bit CPU

Note If the *bit location* is entered as an integer or in a 3*x* register, the instruction will ignore the state of the bottom input.

Middle Node Content

The middle node is the first word or register in the *data matrix*. It may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 4*x* reference in a matrix of contiguous holding registers

Bottom Node Content

The integer value entered in the bottom node specifies a *matrix length*—i.e., the number of 16-bit words or registers in the *data matrix*. The *length* can range from 1 ... 255 in a 16-bit CPU and from 1 ... 600 in a 24-bit CPU—e.g., a *matrix length* of 200 indicates 3200 *bit locations*.

10.9 BROT

The BROT (bit rotate) instruction shifts the bit pattern in a *source matrix*, then posts the shifted bit pattern in a *destination matrix*. The bit pattern shifts left or right by one position per scan.

Warning! BROT will override any disabled coils within a *destination matrix* without enabling them. This can cause injury if a coil has been disabled for repair or maintenance if BROT unexpectedly changes the coil's state.

10.9.1 Characteristics

Size

Three nodes high

PLC Compatibility

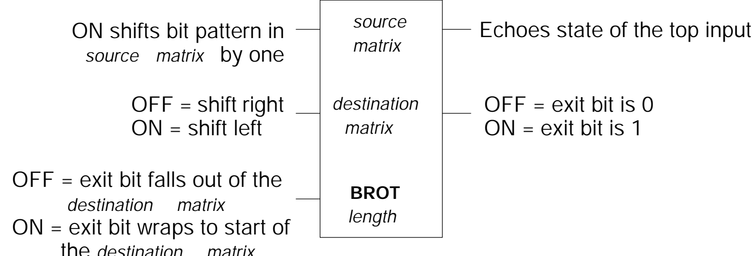
Standard in all PLC types

Opcode

FD hex

10.9.2 Representation

Block Structure



Inputs

BROT has three control inputs. The inputs determine the way the bits will be shifted.

Outputs

BROT can produce two possible outputs (from the top and middle nodes). The output from the top node echoes the state of the top input.

The output from the middle node indicates the sense of the bit that exits the *source matrix* (the leftmost or rightmost bit) as a result of the shift.

Top Node Content

The entry in the top node is the first reference in the *source matrix* —i.e., in the *matrix* that will have its bit pattern shifted. The entry may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 1*x* reference in a matrix of contiguous discrete inputs
- The first 3*x* reference in a matrix of contiguous input registers
- The first 4*x* reference in a matrix of contiguous holding registers

Middle Node Content

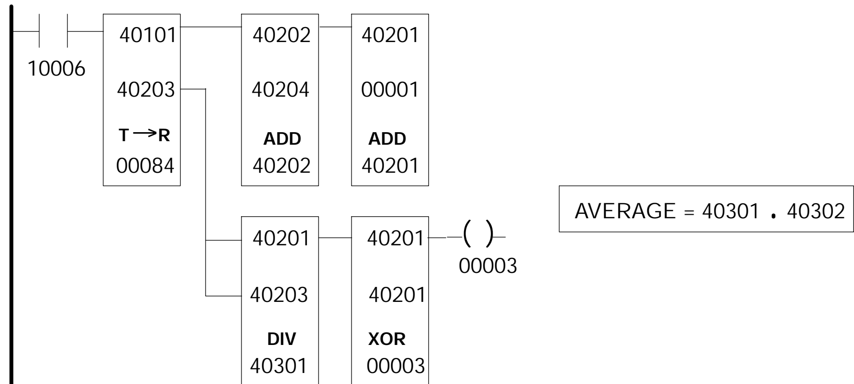
The entry in the middle node is the first reference in the *destination matrix* —i.e., in the *matrix* that shows the shifted bit pattern. The entry may be:

- The first 0*x* reference in a matrix of contiguous coils or discrete outputs
- The first 4*x* reference in a matrix of contiguous holding registers

Bottom Node Content

The integer value entered in the bottom node specifies the *matrix length* —i.e., the number of registers or 16-bit words in each of the two matrices. (The *source matrix* and *destination matrix* have the same *length*.) The *matrix length* can range from 1 ... 100—e.g., a *matrix length* of 100 indicates 1600 *bit locations*.

10.10 A Simple Table Averaging Example



When contact 10006 passes power to the top node of the T→R instruction, the value in the first register of the table (register 40101) is copied into the middle node (40204) of the first ADD instruction. The middle node of the DIV instruction (40203) holds the pointer value. Because the top output of the T→R block is passing power, the first ADD block receives power, causing the value copied to register 40204 to be added to the value in register 40202. The initial value stored in register 40202 is 0.

This routine continues until the value in the pointer register of the T→R instruction (40203) increments to the *table length* —84. The middle output in the T→R block then passes power to the DIV instruction. The values in registers 40201 and 40202 are divided by 84. The *quotient* is posted in register 40301, and the *remainder* is posted in register 40302.

The top output from the DIV instruction passes power to the XOR instruction. By using the XOR to exclusively OR the values in matrix 40201 ... 40203 with themselves, you clear the matrix to 0. The top output of the XOR instruction passes power to coil 00003, indicating that the current table averaging operation is complete and that a new one should start.

10.1 1 Setting Step Flags and Monitoring Steps in Modsoft SFC

Modsoft panel software provides two additional off-line instructions that can be used when you are programming a sequential function chart (SFC) application:

- TC sets the step flag, allowing the SFC program to pass through the transition to the next step; it would appear on-line as an MBIT instruction
- RStF monitors the state of the current step and uses an output to signal whether it is active or not; it would appear on-line as a SENS instruction

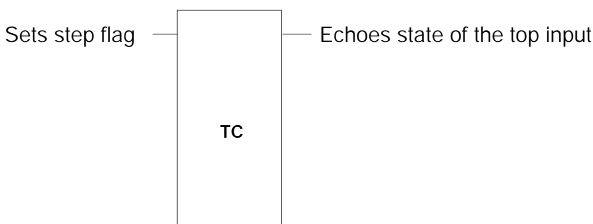
10.1 1.1 Characteristics

Size
Three nodes high

PLC Compatibility
Not PLC-based instructions; reside in Modsoft panel software and are executed as SKP instructions by the PLC

10.1 1.2 Off-line Representations

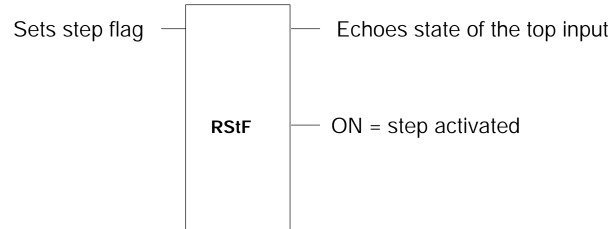
TC



A TC block is normally placed in the lower right corner of the transition network. The input should be a coil reference signifying the end of the process contained in the previous step. Once TC has been executed, the PLC will make one more scan of the previous step.

To invoke this instruction off-line, push <Alt> **F** and type MBIT.

RStF



The RStF block is usually placed in a step. The middle output passes power as long as the step is active. Because SFC allows one extra scan of an inactive step, the middle output can be used to shut down outputs. When a step becomes inactive, its associated network logic is skipped, leaving the outputs (ON or OFF) from the last scan.

To invoke this instruction off-line, push <Alt> **F** and type SENS.

Chapter 11
Monitoring Remote I/O System Status

- STAT
- The S901 Status Table
- The S908 Status Table
- The Compact PLC Status Table
- The Micro PLC Status Table
- HLTH

11.1 STAT

The STAT instruction accesses a specified number of words in a status table in the PLC's system memory. Here vital diagnostic information regarding the health of the PLC and its remote I/O drops is posted. This information includes:

- PLC status
- Possible error conditions in the I/O modules
- Input-to-PLC-to-output communication status

The full length—i.e., number of words—in the status table will vary depending on the type of PLC you are using and on the I/O communications protocol. With the STAT instruction, you can copy some or all of the status words into a block of registers or a block of contiguous discrete references.

Caution: We recommend that you do not use discretes in the STAT *destination* node because of the excessive number required to contain status information.

The copy to the STAT block always begins with the first word in the table up to the last word of interest to you. For example, if the status table is 277 words long and you are interested only in the statistics provided in word 11, you need to copy only words 1 ... 11 by specifying a *length* of 11 in the STAT instruction.

11.1.1 Characteristics

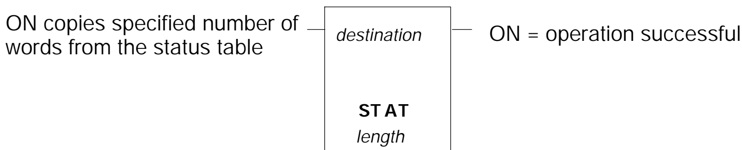
Size
Two nodes high

PLC Compatibility
Standard in all PLC types (but maximum status table length varies according to PLC type and I/O communications protocol in use)

Opcode
FC hex

11.1.2 Representation

Block Structure



Top Node Content

The reference number entered in the top node is the first position in the *destination* block—i.e., the block where the current words of interest from the status table will be copied. The reference may be

- The first 0*x* reference in a block of contiguous discrete outputs
- The first 4*x* reference in a block of contiguous holding registers

The number of holding registers or 16-bit words in the *destination* block is specified in the bottom node.

Bottom Node Content

The integer value entered in the bottom node specifies the number of registers or 16-bit words in the *destination* block where the current status information will be written. The maximum allowable *length* will differ according to the type of PLC in use and the type of I/O communications protocol employed:

- For a 984A, 984B, or 984X Chassis Mount PLC using the *S901* RIO protocol, the available range of the system status table is 1 ... 75 words
- For PLCs with 16-bit CPUs using the *S908* RIO protocol—e.g. the 38*x*, 48*x*, and 68*x* Slot Mount PLCs—the available range of the system status table is 1 ... 255
- For PLCs with 24-bit CPUs using the *S908* RIO protocol—e.g., the 78*x* Slot Mount PLCs, the Quantum PLCs—the available range of the system status table is 1 ... 277
- For Compact-984 PLCs, the available range of the system status table is 1 ... 184
- For Modicon Micro PLCs, the available range of the system status table is 1 ... 56

11.2 The S901 Status Table

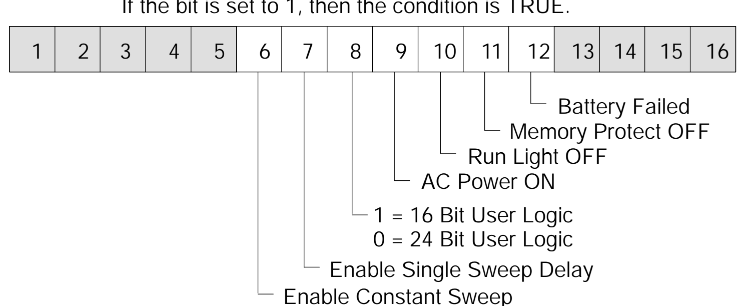
The 75 words in the S901 status table are divided into three sections—the first 11 words for controller status information, the next 32 words for I/O module health information, and the last 32 words for I/O communications information:

Decimal Word	Word Content		Hex Word
1	Controller Status		01
2			02
3	Controller Status		03
4	S901 Status		04
5	Controller Stop State		05
6	Number of Segments in User Logic		06
7	Address of End-0f-Logic Pointer		07
8	RIO Redundancy and Timeout		08
9	ASCII Message Status		09
10	Run Load Debug Status		0A
11	Address of Status Word Pointer Table		0A
	High Byte	**Low Byte**	
12	Channel 1 Input	Channel 2 Input	0C
13	Channel 3 Input	Channel 4 Input	0D
. . .	. . .	. . .	. . .
28	Channel 31 Input	Channel 32 Input	1C
29	Channel 1 Output	Channel 2 Output	1D
30	Channel 3 Output	Channel 4 Output	1E
. . .	. . .	. . .	. . .
43	Channel 31 Output	Channel 32 Output	2B
44	Remote I/O Channels 5 and 6 (First word)		2C
45	Remote I/O Channels 5 and 6 (Second word)		2D
46	Remote I/O Channels 7 and 8 (First word)		2E
47	Remote I/O Channels 7 and 8 (Second word)		2F
. . .	. . . . . .		. . .
70	Remote I/O Channels 31 and 32 (First word)		46
71	Remote I/O Channels 31 and 32 (Second word)		47
72	Remote I/O Channels 1 and 2 (First word)		48
73	Remote I/O Channels 1 and 2 (Second word)		49
74	Remote I/O Channels 3 and 4 (First word)		4A
75	Remote I/O Channels 3 and 4 (Second word)		4B

11.2.1 S901 Controller Status Words

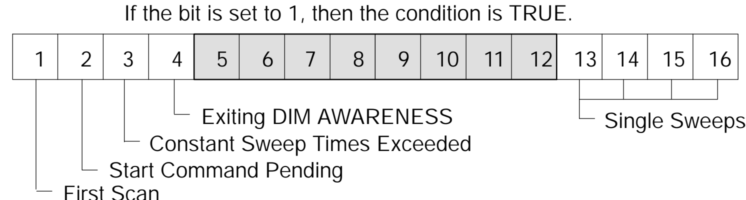
Words 1 ... 11 display the controller status words:

Word 1 displays the following aspects of the controller's status:

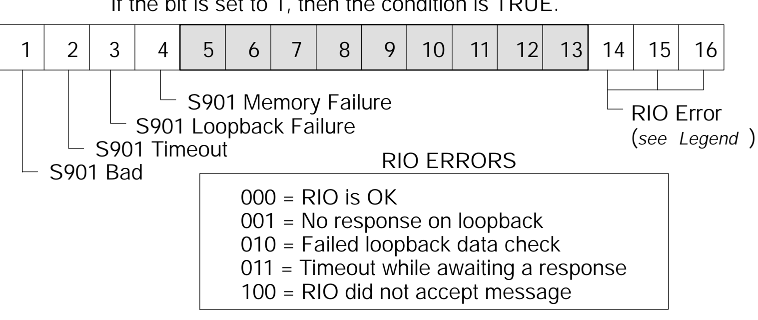


Word 2 is not used. All its bits are cleared to zero.

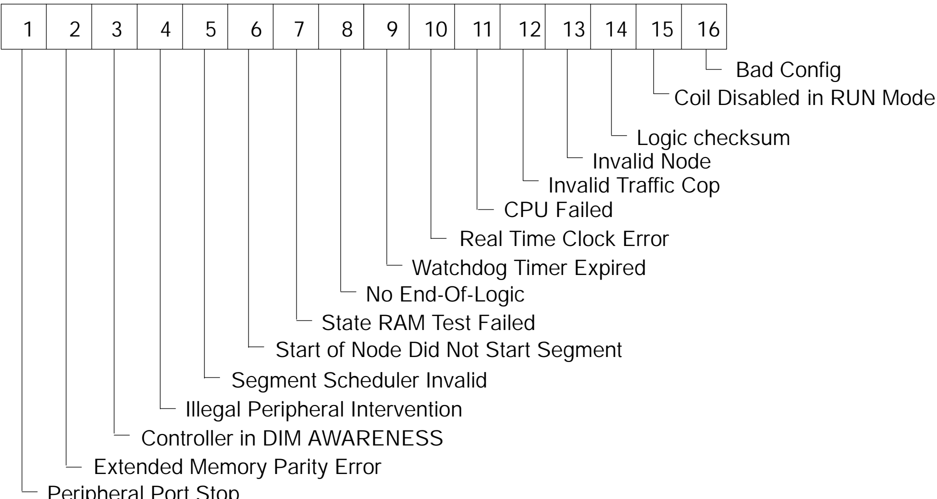
Word 3 displays the following aspects of the controller's status:



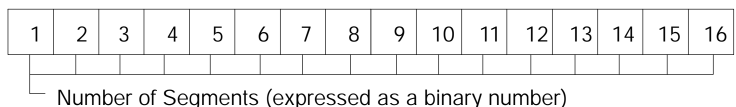
Word 4 displays the status of the S901 Remote I/O Processor:



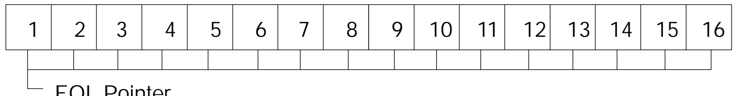
Word 5 displays the controller's stop state conditions:



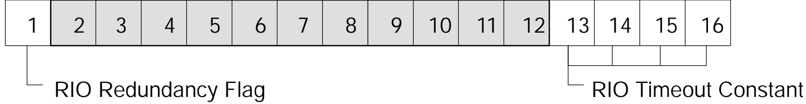
Word 6 displays the number of logic segments:



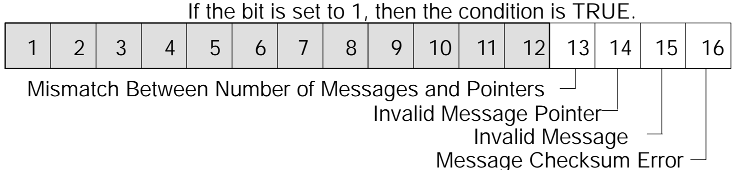
Word 7 displays the end-of-logic (EOL) pointer:



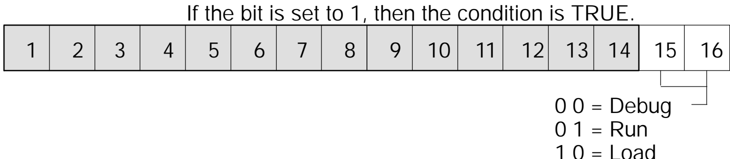
Word 8 holds a RIO redundancy flag and displays an RIO timeout constant:



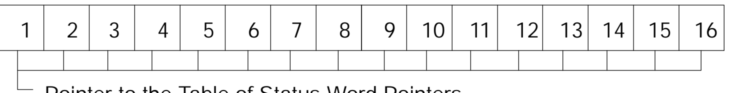
Word 9 displays the ASCII message status:



Word 10 uses its two most significant bits to display the RUN load debug status:

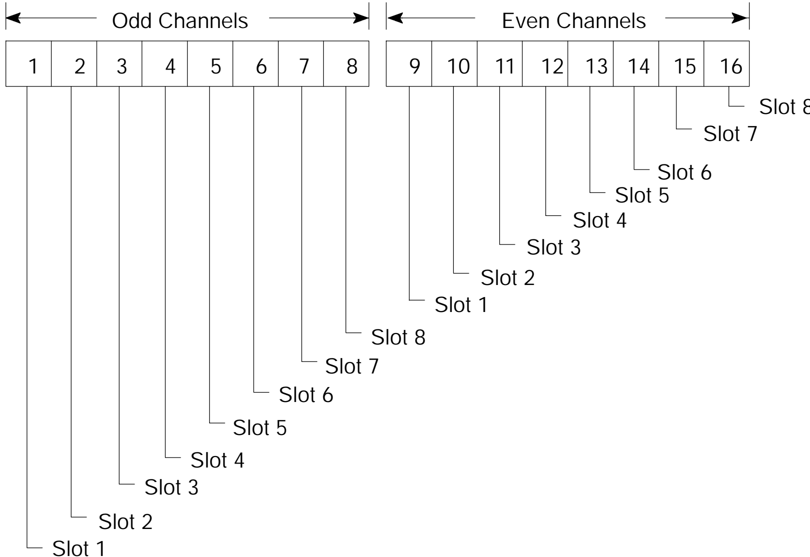


Word 11 displays the address of the table of status word pointers:



11.2.2 S901 I/O Module Health Status Words

Words 12 ... 43 use the high and low bytes to display the health of the I/O modules in the odd and even channels. Each of these 32 status words is organized as follows:



If a specified slot is inhibited in the traffic cop, the bit is 0. If the slot contains an input module or an input/output module, the bit is 1. If the slot contains an output module and the module's **COMM ACTIVE** LED is ON, the bit is 0; if slot contains an output module and the module's **COMM ACTIVE** LED is OFF, the bit is 1.

Note: These indicators are valid only when scan time > 30 ms.

11.2.3 S901 RIO Communication Status Words

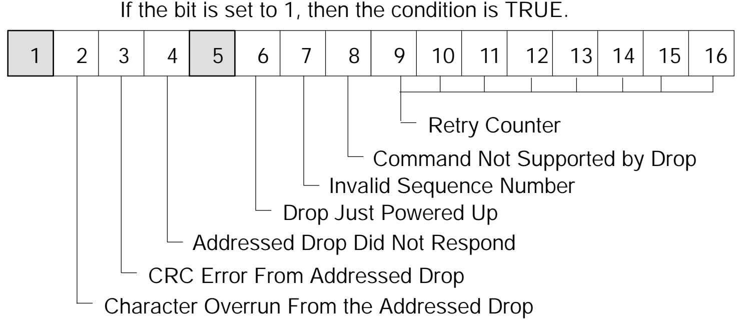
RIO system communication status is given in words 44 ... 75. Two words are used to describe each of up to 16 drops. The format of the first word is:

If the bit is set to 1, then the condition is TRUE.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Busy 1
Send Sequence
Cable B
Receive Sequence
Busy 0
Current Message Not Supported
Byte Count Underrun
Sequence Number Invalid
Function Scheduled:
0 0 0 = Normal I/O
0 0 1 = Restart (Comm Reset)
0 1 0 = Restart (Application Reset)
1 0 0 = Inhibit

The format of the second word is:



11.3 The S908 Status Table

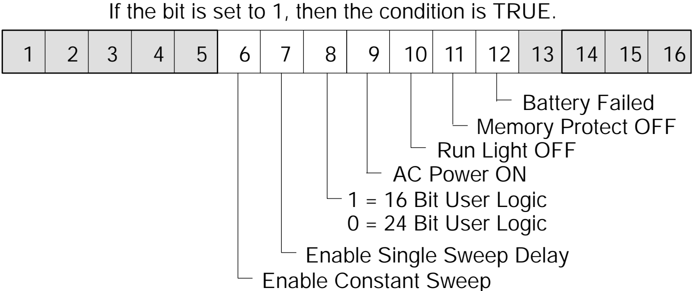
The 277 words in the S908 status table are organized in three sections— controller status, I/O module health, and I/O communication health:

Decimal Word	Word Content	Hex Word
1	Controller Status	01
2	Hot Standby Status	02
3	Controller Status	03
4	RIO Status	04
5	Controller Stop State	05
6	Number of Ladder Logic Segments	06
7	End-of-logic (EOL) Pointer	07
8	RIO Redundancy and Timeout	08
9	ASCII Message Status	09
..10	RUN/LOAD/DEBUG Status	0A
11		0B
12	Drop 1, Rack 1	0C
..13	Drop 1, Rack 2	0D
. . .	. . . . . .	. . .
16	Drop 1, Rack 5	0F
17	Drop 2, Rack 1	10
18	Drop 2, Rack 2	11
. . .	. . . . . .	. . .
171	Drop 32, Rack 5	AB
172	S908 Startup Error Code	AC
173	Cable A Errors	AD
174	Cable A Errors	AE
175	Cable A Errors	AF
176	Cable B Errors	B0
177	Cable B Errors	B1
178	Cable B Errors	B2
179	Global Communication Errors	B3
180	Global Communication Errors	B4
181	Global Communication Errors	B5
182	Drop 1 Errors/Health Status and Retry Counters (in the Compact 984 Controllers) (First word)	B6
183	Drop 1 Errors/Health Status and Retry Counters (in the Compact 984 Controllers) (Second word)	B7
184	Drop 1 Errors/Health Status and Retry Counters (in the Compact 984 Controllers) (Third word)	B8

185	Drop 2 Errors/Health Status and Retry Counters (in the Compact 984 Controllers) (First word)	B9
. . .	. . . . . .	. . .
275	Drop 32 Errors/Health Status and Retry Counters (in the Compact 984 Controllers) (First word)	113
276	Drop 32 Errors/Health Status and Retry Counters (in the Compact 984 Controllers) (Second word)	114
277	Drop 32 Errors/Health Status and Retry Counters (in the Compact 984 Controllers) (Third word)	115

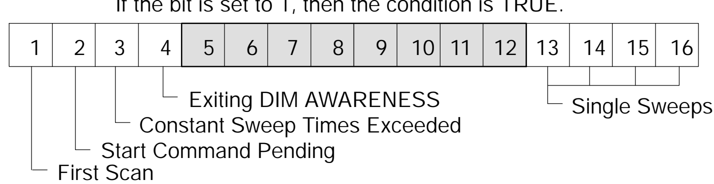
11.3.1 S908 PLC Status Words

Word 1 displays the following aspects of the PLC's status:

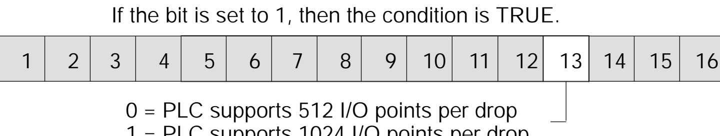


Word 2 is reserved for internal use. It describes the health and operating state of an S911 Hot Standby system; the bits could potentially be misinterpreted based on the moment they are observed relative to the machine's state. If you want to view the status of a Hot Standby system, use the HSBY status register, which is part of the HSBY loadable instruction (page 464).

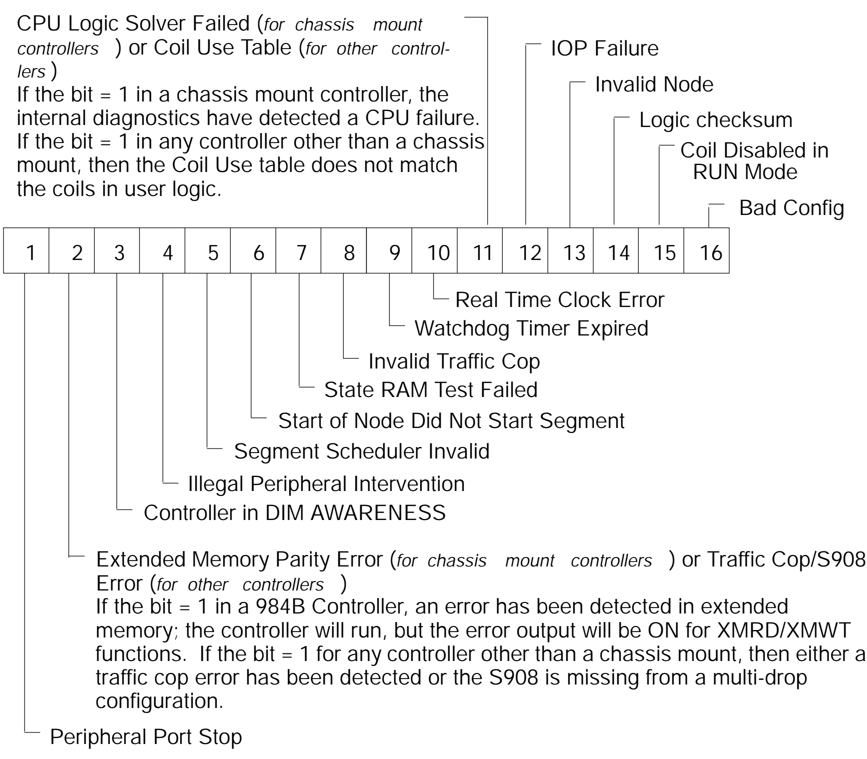
Word 3 displays more aspects of the controller status:



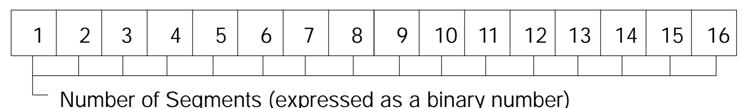
Word 4 is not used with S908 in the 984A/B/X controllers; in other controllers, bit 13 is used as follows:



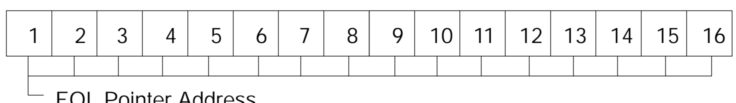
Word 5 displays the PLC's stop state conditions:



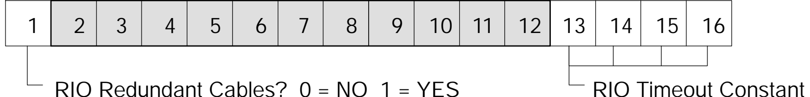
Word 6 displays the number of segments in ladder logic; a binary number is shown:



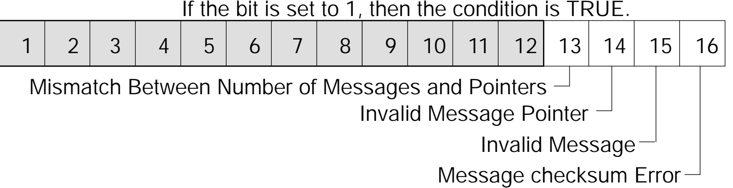
Word 7 displays the address of the end-of-logic (EOL) pointer:



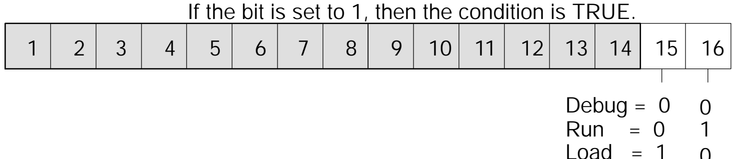
Word 8 uses its most significant bit to display whether or not redundant coaxial cables are run to the remote I/O drops, and it uses its four least significant bits to display the remote I/O timeout constant:



Word 9 uses its four least significant bits to display ASCII message status:



Word 10 uses its two least significant bits to display RUN/LOAD/DEBUG status:



Word 11 is not used.

11.3.2 S908 I/O Module Health Status Words

Status words 12 ... 171 display I/O module health status.

Five words are reserved for each of up to 32 drops, one word for each of up to five possible racks (I/O housings) in each drop. Bits 1 ... 16 in each word represent the health of the associated I/O module in each rack.

If the bit is set to 1, then the condition is TRUE.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7	Slot 8	Slot 9	Slot 10	Slot 11	Slot 12	Slot 13	Slot 14	Slot 15	Slot 16

Four conditions must be met before an I/O module can indicate good health:

- The slot must be traffic copped
- The slot must contain a module with the correct personality
- Valid communications must exist between the module and the RIO interface at remote drops
- Valid communications must exist between the RIO interface at each remote drop and the I/O processor in the controller

Converting from Word # to Drop and Rack

$$\frac{\textit{Word \#} - 12}{5} = \textit{Quotient} + \textit{Remainder}$$

where

Drop # = Quotient + 1
Rack # = Remainder + 1

Converting from Drop and Rack to Word #

Word # = (*Drop #* x 5) + *Rack #* + 6

Status Words for the MMI Operator Panels

The status of the 32 Element Pushbutton Panels and PanelMate units on an RIO network can also be monitored with an I/O health status word. The Pushbutton Panels occupy slot 4 in an I/O rack and can be monitored at bit 4 of the appropriate status word. A PanelMate on RIO occupies slot 1 in rack 1 of the drop and can be monitored at bit 1 of the first status word for the drop.

Tip The ASCII Keypad's communication status can be monitored with the error codes in the ASCII READ/WRIT blocks (see Chapter 15).

11.3.3 S908 I/O Communication Status Words

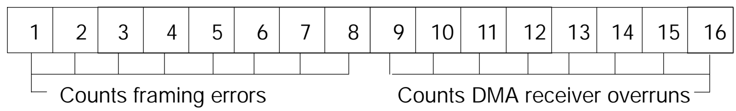
Status words 172 ... 277 contain the I/O system communication status. Words 172 ... 181 are global status words. Among the remaining 96 words, three words are dedicated to each of up to 32 drops, depending on the type of PLC.

Word 172 stores the *S908 Startup Error Code*. This word is always 0 when the system is running. If an error occurs, the controller does not start—it generates a stop state code of 10 (word 5):

S908 Start-up Error Codes

Code	Error	Meaning (Where the error has occurred)
01	BADTCLEN	Traffic Cop length
02	BADLNKNUM	Remote I/O link number
03	BADNUMDPS	Number of drops in Traffic Cop
04	BADTCSUM	Traffic Cop checksum
10	BADDDLEN	Drop descriptor length
11	BADDRPNUM	I/O drop number
12	BADHUPTIM	Drop holdup time
13	BADASCNUM	ASCII port number
14	BADNUMODS	Number of modules in drop
15	PRECONDRP	Drop already configured
16	PRECONPRT	Port already configured
17	TOOMNYOUT	More than 1024 output points
18	TOOMNYINS	More than 1024 input points
20	BADSLTNUM	Module slot address
21	BADRCKNUM	Module rack address
22	BADOUTBC	Number of output bytes
23	BADINBC	Number of input bytes
25	BADRF1MAP	First reference number
26	BADRF2MAP	Second reference number
27	NOBYTES	No input or output bytes
28	BADDISMAP	Discrete not on 16-bit boundary
30	BADODDOUT	Unpaired odd output module
31	BADODDIN	Unpaired odd input module
32	BADODDREF	Unmatched odd module reference
33	BAD3X1XRF	1*x* reference after 3*x* register
34	BADDMYMOD	Dummy module reference already used
35	NOT3XDMY	3*x* module not a dummy
36	NOT4XDMY	4*x* module not a dummy
40	DMYREAL1X	Dummy, then real 1*x* module
41	REALDMY1X	Real, then dummy 1*x* module
42	DMYREAL3X	Dummy, then real 3*x* module
43	REALDMY3X	Real, then dummy 3*x* module

Words 173 ... 175 are Cable A error words:

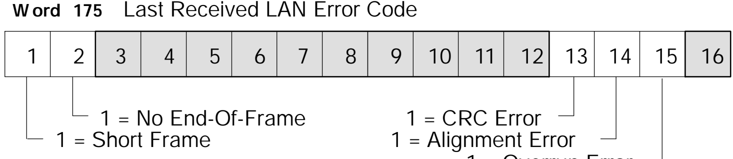


Word 174

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

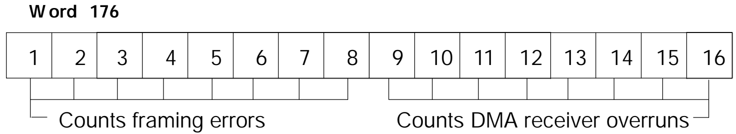
Counts receiver errors — Counts bad drop receptions

Word 175 Last Received LAN Error Code

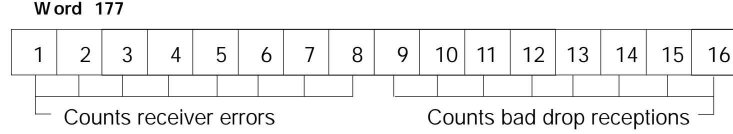


Words 176 ... 178 are Cable B error words:

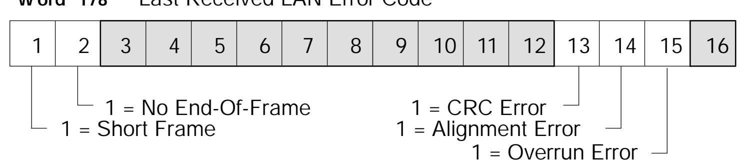
Word 176



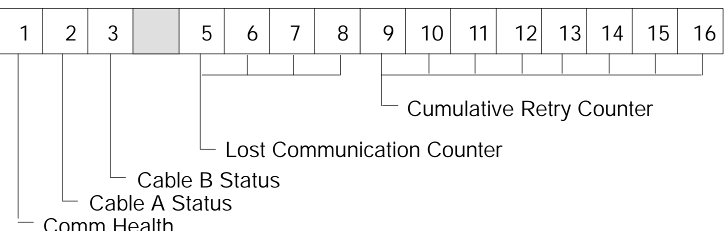
Word 177



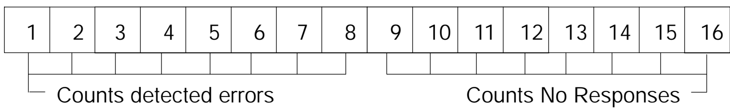
Word 178 Last Received LAN Error Code



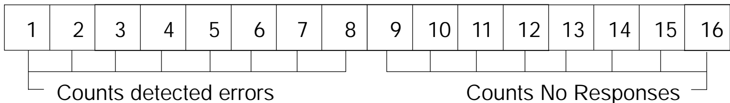
Word 179 displays global communication status:



Word 180 is the global cumulative error counter for Cable A:

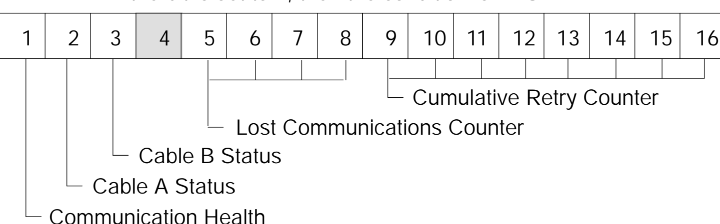


Word 181 is the global cumulative error counter for Cable B:

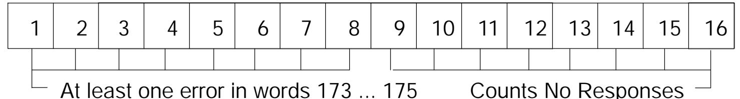


Words 182 ... 277 are used to describe remote I/O drop status; three status words are used for each drop.

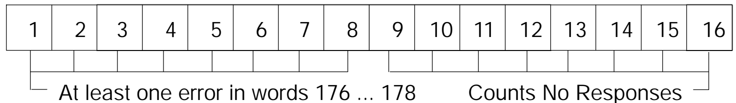
The first word in each group of three displays communication status for the appropriate drop:



The second word in each group of three is the drop cumulative error counter on Cable A for the appropriate drop:

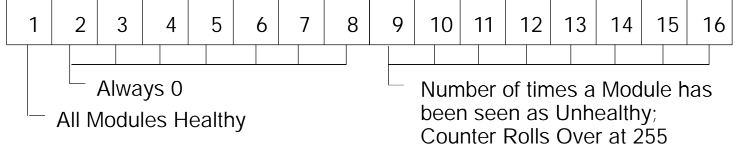


The third word in each group of three is the drop cumulative error counter on Cable B for the appropriate drop:



Note: For PLCs where drop 1 is reserved for local I/O, status words 182 ... 184 are used as follows:

Word 182 displays local drop status:



Word 183 is used as a 16-bit I/O bus error counter.

Word 184 is used as a 16-bit I/O bus retry counter.

Converting a Word # to a Drop # or Word

$$\frac{word\ \# - 182}{3} = quotient \text{ and } remainder$$

quotient + 1 = *drop #*
remainder + 1 = *word*

Converting a Drop # or Word to a Word #

(*drop #* x 3) + *word* + 178 = *word #*

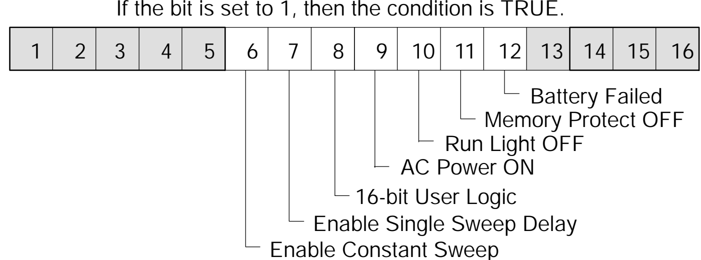
11.4 The Compact PLC Status Table

The Compact PLC status table in memory is organized in three groups— current PLC status information, the current health of the A120 I/O modules in the system, and current global health statistics. Words 16 ... 181 are reserved in the table but they are not used to post I/O or system health statistics.

Decimal Word	Word Content	Hex Word
1	Controller Status	01
2		02
3	Controller Status	03
4		04
5	Controller Stop State	05
6	Number of Ladder Logic Segments	06
7	End-of-logic (EOL) Pointer	07
8	Memory Sizing Word for Panel (in the 984-145 Compact Controller)	08
9		09
10	RUN/LOAD/DEBUG Status	0A
11		0B
12	Rack 1	0C
13	Rack 2	0D
14	Rack 3	0E
15	Rack 4	0F
16		10
. . .	. . . . . .	. . .
182	Systemwide I/O health status	B9
183	I/O Error Count	BA
184	PAB Bus Retry Count	BB

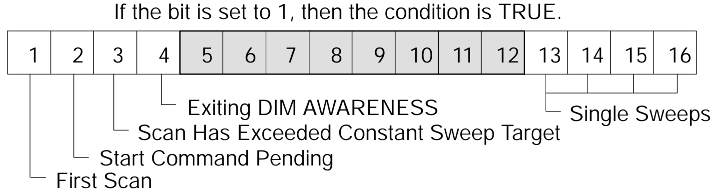
11.4.1 Compact PLC Status Words

Word 1 displays the following aspects of the PLC's status:



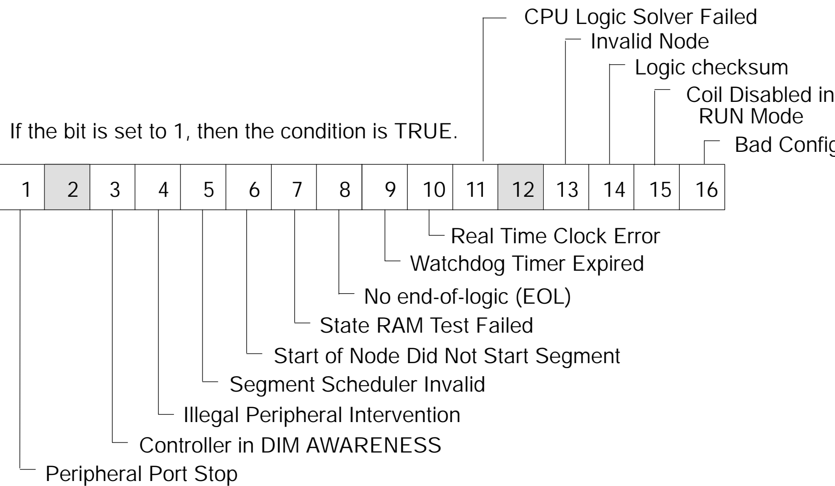
Word 2 is not used.

Word 3 displays more aspects of the PLC status:

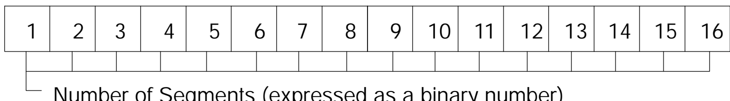


Word 4 is not used.

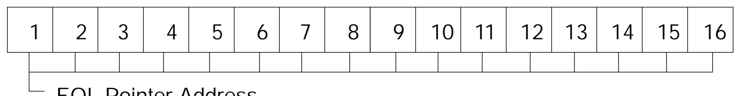
Word 5 displays the PLC's stop state conditions:



Word 6 displays the number of segments in ladder logic; a binary number is shown:



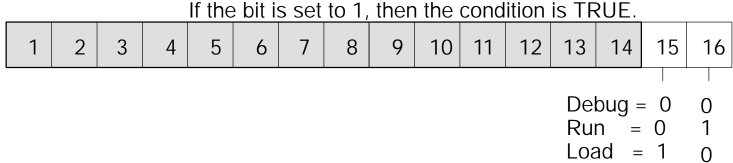
Word 7 displays the address of the end-of-logic (EOL) pointer:



Word 8 is used only with the Compact 984-145. It provides memory sizing information to the programming panel.

Word 9 is not used.

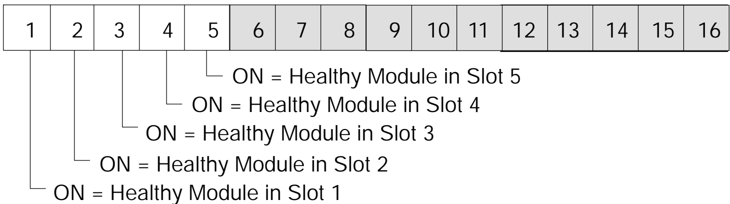
Word 10 uses its two least significant bits to display RUN/LOAD/DEBUG status:



Word 11 is not used.

11.4.2 Compact I/O Module Health Status Words

Status words 12 ... 15 are used to display the health status of the A120 I/O modules in each of the four racks. The most significant bit in each of these four words represents the I/O module in slot 1 of its associated rack:



Three conditions must be met before an I/O module can indicate good health:

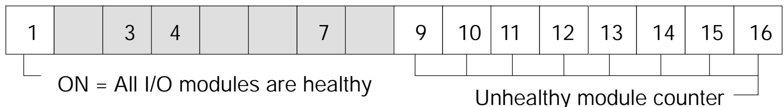
- □ The slot must be traffic copped
- □ The slot must contain a module with the correct personality
- □ Valid communications must exist between the module and the RIO interface at remote drops

Words 16 ... 181 are not used.

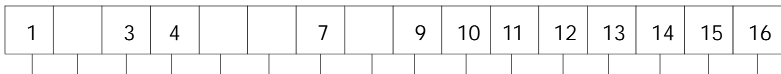
11.4.3 Compact I/O Communication Status Words

The last three words in the Compact PLC status table describe the health of the communications on the installed A120 I/O modules.

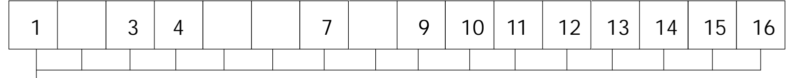
- □ Word 182 describes systemwide I/O health status. Bits 9 ... 16 form a counter that increments each time an unhealthy module is found. The counter rolls over when the count surpasses 255:



- Word 183 accumulates the I/O error count. It increments once on every logic scan where an unhealthy I/O module is found:



- Word 184 keeps a count of the retries on the PAB bus. Bits 1 ... 16 accumulate a count that increments once each time a comm retry occurs. If after one try and four retries a bus error is still detected, the PLC stops and displays error code 10 on the programming panel. Normally, all bits in this word should be 0s.



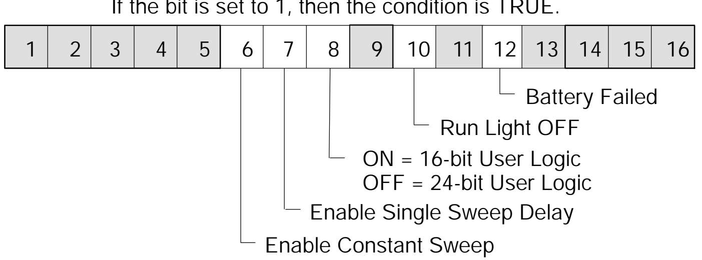
11.5 Micro PLC Status Table

The Modicon Micro PLC status table is organized in three grups—current PLC status information, the current health of the I/O locations in the system, and global, health and communication status for the remote drops.

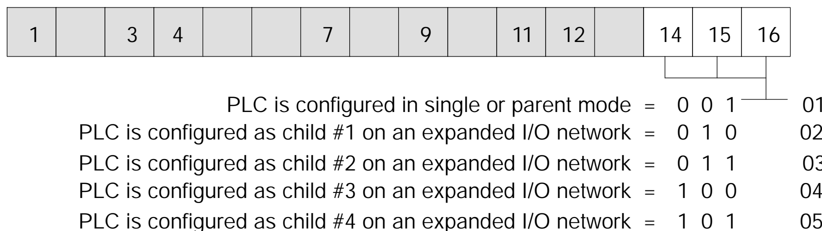
Decimal Word	Word Content	Hex Word
1	Controller Status	01
2	PLC Drop Address	02
3	Controller Status	03
4	Maximum number of I/O drops	04
5	Controller Stop State	05
6	Number of Ladder Logic Segments	06
7	End-of-logic (EOL) Pointer	07
8		08
9		09
10	RUN/LOAD/DEBUG Status	0A
11		0B
12	Drop 1, Rack 1	0C
13	Drop 1, Rack 2	0D
14	Drop 1, Rack 3	0E
15	Drop 1, Rack 4	0F
16	Drop 1, Rack 5	10
17	Drop 2, Rack 1	11
. . .	. . . . . .	. . .
31	Drop 5, Rack 4	1F
32	Start-up Error Code Log	20
33	Global Communications Status (word 1)	21
. . .	. . . . . .	. . .
36	Global Communications Status (word 4)	24
37	Rack 1 I/O Health Status	25
38	Rack 1 I/O Error Detection Counter	26
39	Rack 1 I/O Retry Counter	27
40		28
41	Communication health on an I/O expansion network (parent PLC only)	29
. . .	. . . . . .	. . .
56	Communication health on an I/O expansion network (parent PLC only)	38

11.5.1 Micro PLC Status Words

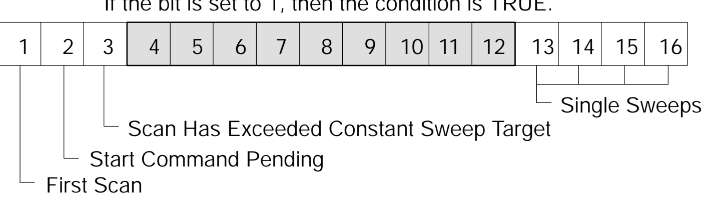
Word 1 displays the following aspects of the PLC's status:



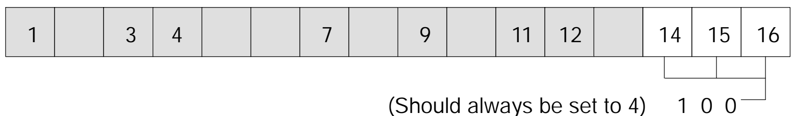
Word 2 displays the PLC drop address:



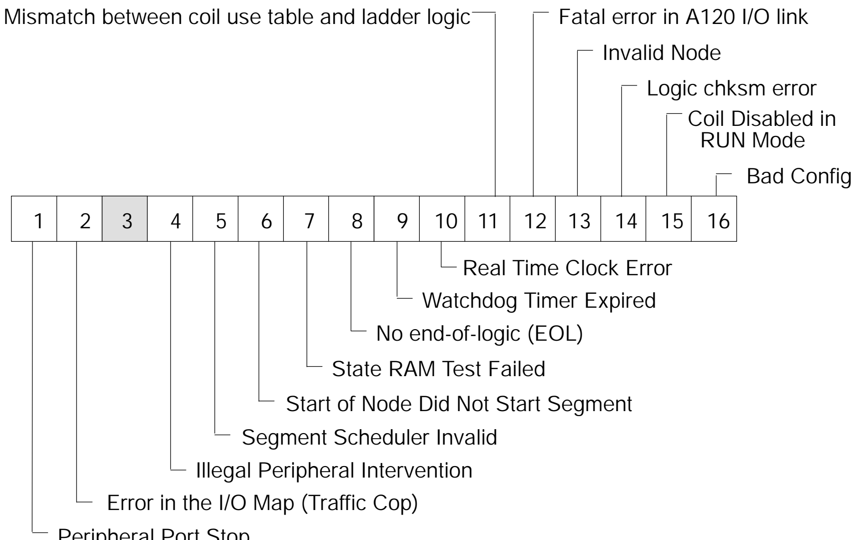
Word 3 displays more aspects of the PLC status:



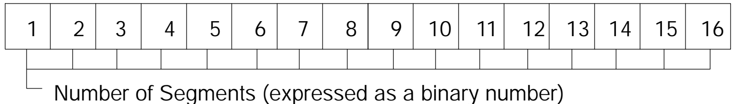
Word 4 displays the maximum number of drops allowed in the I/O network:



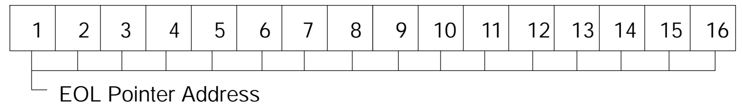
Word 5 displays the PLC's stop state conditions:



Word 6 displays the number of segments in ladder logic; a binary number is shown:



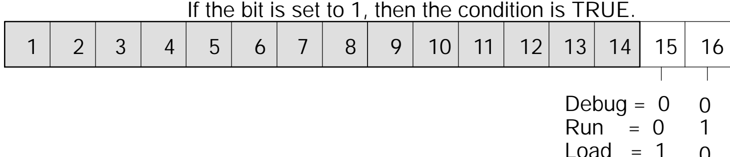
Word 7 displays the address of the end-of-logic (EOL) pointer:



Word 8 is reserved.

Word 9 is reserved.

Word 10 uses its two least significant bits to display RUN/LOAD/DEBUG status:



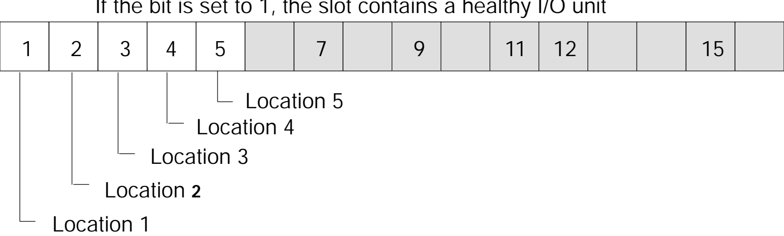
Word 11 is reserved.

11.5.2 Micro I/O Expansion Health

Words 12 ... 31 are used to display the health status of an I/O expansion system on a Micro control network. Four contiguous words are used for each of up to five Modicon Micro PLCs on an I/O expansion network; one word in each group of four is used for each possible I/O rack, assuming A120 I/O expansion.

Rack 1 is always a Modicon Micro PLC. Racks 2 ... 4 are A120 I/O racks connected to rack 1 via an A120 I/O expansion port.

Each word contains five representative bits that show the health of the associated I/O unit in each rack—i.e., each rack can support a maximum of five I/O locations:



With respect to A120 I/O modules, a location is the physical slot position of the module in its DTA housing. With respect to a Micro PLC, the location relates to the following fixed components on the unit:

- Location 1 represents the fixed discrete inputs and outputs on the unit
- Location 2 represents the dedicated interrupt component status on the unit
- Location 3 represents the user-selectable counter/timer count on the unit
- Location 4 represents any fixed analog inputs and outputs on the unit
- Location 5 represents the data transfer component on the unit for serial I/O expansion

An I/O location is healthy when it is configured and I/O mapped correctly. Its personality is correct, and valid communications exist between it and the CPU that controls it.

Converting from Word # to PLC and Rack

$$\frac{\textit{word}\ \#\quad 12}{4} = \textit{quotient} + \textit{remainder}$$

where

- $\textit{quotient} + 1 = \textit{drop}\ \#$
- $\textit{remainder} + 1 = \textit{rack}\ \#$

11.5.3 Start-up Error Codes

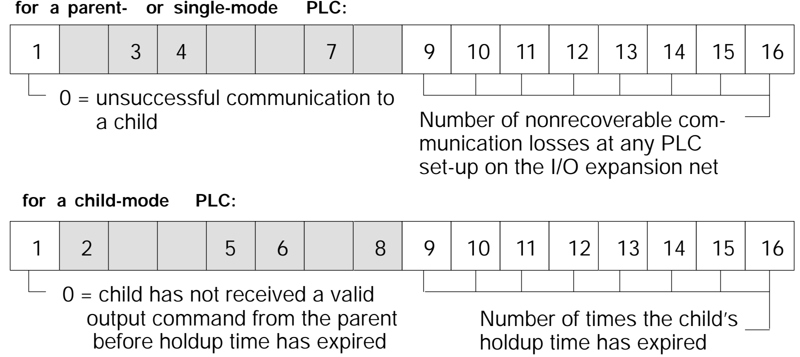
Word 32 is a log of the start-up error codes:

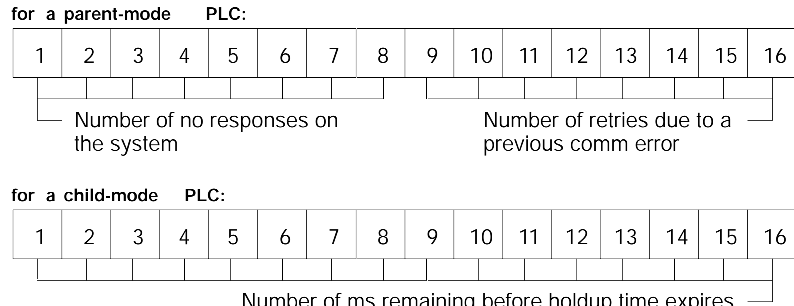
1		3	4			7		9		11	12	13	14	15	16

Error	11	12	13	14	15	16
Bad I/O map length	0	0	0	0	0	1
Bad link number for child PLCs on the network	0	0	0	0	1	0
Wrong number of child PLCs in I/O map	0	0	0	0	1	1
Bad I/O map checksum	0	0	0	1	0	0
Bad child PLC descriptor length	0	0	1	0	1	0
Bad child PLC number	0	0	1	0	1	1
Bad holdup time for child PLC on the network	0	0	1	1	0	0
Bad ASCII port number	0	0	1	1	0	1
Bad number of slots in a child PLC	0	0	1	1	1	0
Child PLC has already been set up	0	0	1	1	1	1
comm port has already been set up	0	1	0	0	0	0
More than 1024 output points	0	1	0	0	0	1
More than 1024 input points	0	1	0	0	1	0
Bad slot address	0	1	0	1	0	0
Bad rack address	0	1	0	1	0	1
Bad number of output bytes	0	1	0	1	1	0
Bad number of input bytes	0	1	0	1	1	1
Bad first reference number	0	1	1	0	0	1
Bad second reference number	0	1	1	0	1	0
No input or output bytes	0	1	1	0	1	1
Discrete not on a 16-bit boundary	0	1	1	1	0	0
Unpaired odd output unit	0	1	1	1	1	0
Unpaired odd input unit	0	1	1	1	1	1
Unmatched odd input/output unit reference	1	0	0	0	0	0
1*x* reference after 3*x* register	1	0	0	0	0	1
Dummy unit reference already used	1	0	0	0	1	0
3*x* reference not a dummy	1	0	0	0	1	1
4*x* reference not a dummy	1	0	0	1	0	0
Dummy, then real 1*x* reference	1	0	1	0	0	0
Real, then dummy 1*x* reference	1	0	1	0	0	1
Dummy, then real 3*x* reference	1	0	1	0	1	0
Real, then dummy 3*x* reference	1	0	1	0	1	1
Too many I/O points in a drop	1	0	1	1	0	0
Bad unit descriptor rack	1	1	0	0	1	0
Bad unit descriptor slot	1	1	0	0	1	1
Bad unit descriptor input byte count	1	1	0	1	0	0
Bad unit descriptor output byte count	1	1	0	1	0	1
I/O driver has not been loaded	1	1	0	1	1	0
Unit can be used only in rack 1	1	1	0	1	1	1

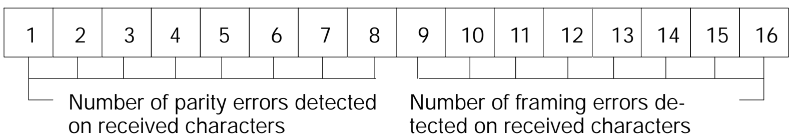
11.5.4 Micro PLC Global Communications Status

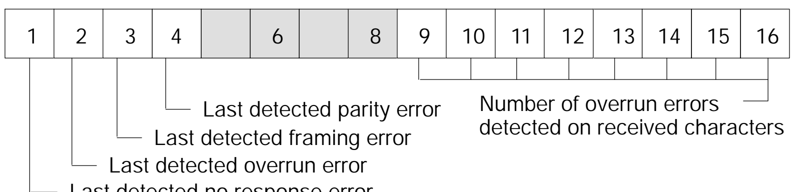
Words 33 and 34 in the Micro status table use their bit values differently depending on whether they are in a parent or a child PLC on the I/O expansion net:



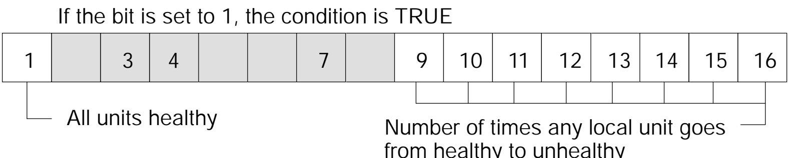


Words 35 and 36 are used only when the PLC is a parent on the I/O expansion net:

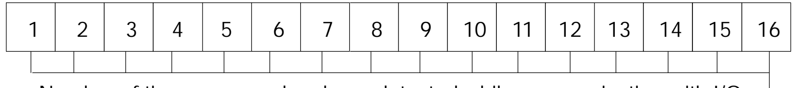




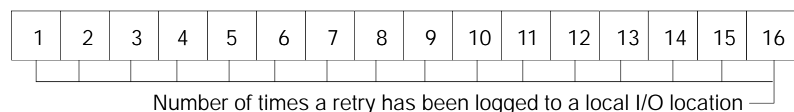
Words 37 ... 39 are used for Micro PLCs that implement A120 expansion. Word 37 displays the healthy of communications in rack 1 of the I/O expansion network:



Words 38 displays I/O error detection in rack 1:



Words 39 is used as an I/O retry counter for rack 1:

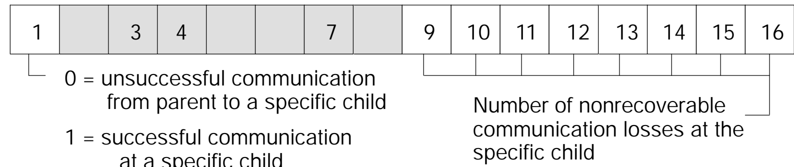


Words 41 ... 56 are for communications on the I/O expansion network—they have meaning only in parent units. Each potential child PLC on the network is described by a group of four contiguous words:

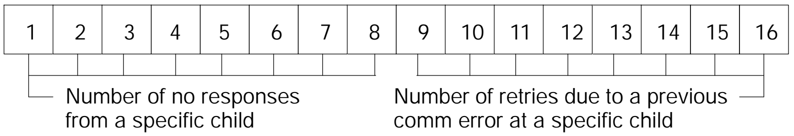
- Words 41 ... 44 apply to child #1
- Words 45 ... 48 apply to child #2

- Words 49 ... 52 apply to child #3
- Words 53 ... 56 apply to child #4

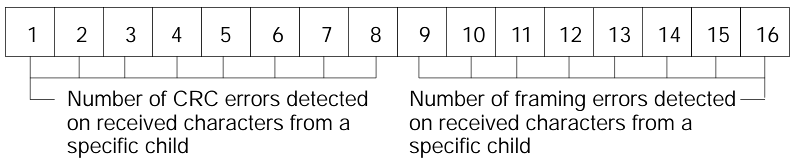
Words 41, 45, 49, and 53 have the following common format:



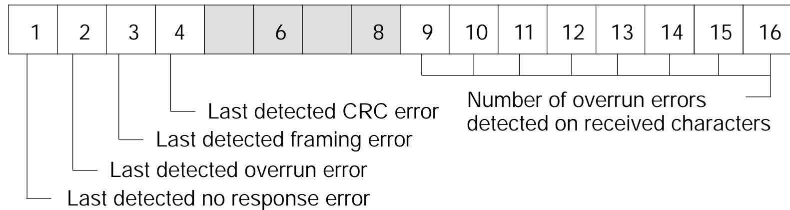
Words 42, 46, 50, and 54 have the following common format:



Words 43, 47, 51, and 55 have the following common format:



Words 44, 48, 52, and 56 have the following common format:



11.6 HL TH

The HLTH instruction creates history and status matrices from internal memory registers that may be used in ladder logic to detect changes in PLC status and communication capabilities with the I/O. It can also be used to alert the user to changes in a PLC System. HLTH has two modes of operation, *learn* and *monitor* .

11.6.1 Learn Mode

HLTH can be initialized to learn the configuration in which it is implemented and save the information as a point- in-time reference called *history* matrix. This matrix contains:

- A user-designated drop number for communications status monitoring
- User logic checksum
- Disabled I/O indicator
- S911 Health
- Choice of single or dual cable system
- Traffic Cop display

11.6.2 Monitor Mode

Monitor mode enables an operation that checks PLC system conditions. Detected changes are recorded in a status matrix. The status matrix monitors the most recent system conditions and sets bit patterns to indicate detected changes. The status matrix contains:

- Communication status of the drop designated in the history matrix
- A flag to indicate when there is any disabled I/O
- Flags to indicate the “on/off’ status of constant sweep and the Memory protect key switch
- Flags to indicate a battery-low condition and if Hot Standby is functional

- Failed module position data
- Changed user logic checksum flag
- RIO lost-communication flag

The HLTH instruction block has three control inputs and can produce three possible outputs. The combined states of the inputs to the middle and bottom nodes control the operating mode:

Middle Input	Bottom Input	Operation
ON	OFF	Learn Mode as Dual Cable System
ON	ON	Learn Mode as Single Cable System
OFF	ON	Monitor Mode
OFF	OFF	Monitor Mode Update Logic Checksum

11.6.3 Characteristics

Size
Three nodes high

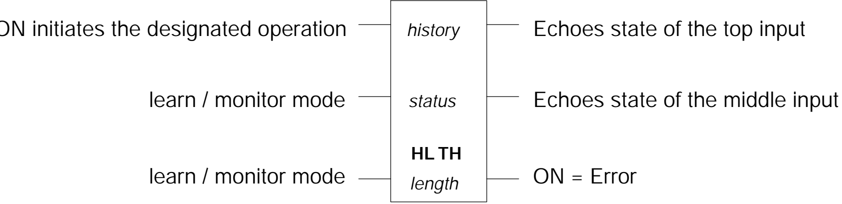
PLC Compatibility
Available as a loadable in all PLC types except:

- 984-120, 984-130, 984-131, and 984-141 Compact PLCs
- All 984A, 984B, and 984X Chassis Mount PLCs

Opcode
03F hex (default)

11.6.4 Representation

Block Structure

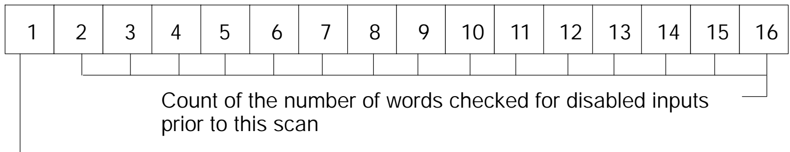


Top Node Content—The History Matrix

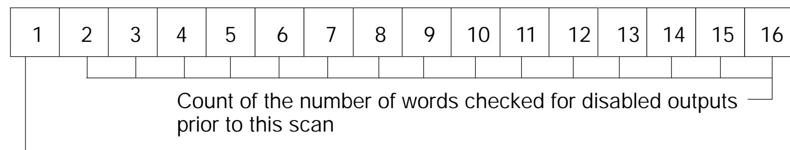
The $4x$ register entered in the top node is the first in a block of contiguous registers that comprise the *history* matrix. The data for the *history* matrix is gathered by the instruction during a learn mode operation and is set in the matrix when the mode changes to monitor.

The *history* matrix can range from 6 ... 135 registers in length. Below is a description of the words in the *history* matrix. The information from word 1 is contained in the displayed register in the top node and the information from words 2 ... 135 is stored in the implied registers.

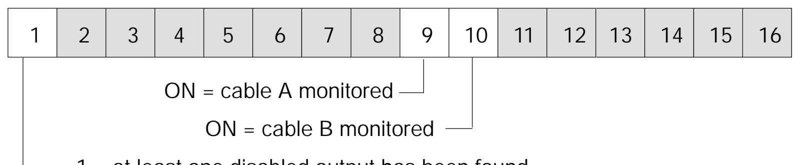
- Word 1: Enter drop number (range 0 ... 32) to be monitored for retries
- Word 2: High word of learned checksum
- Word 3: Low word of learned checksum
- Word 4: The status and a counter for multiplexing the inputs. HLTH processes 16 words of input (256 inputs) per scan. This word holds the last word location of the last scan. The register is overwritten on every scan. The value in the counter portion of the word increases to the maximum number of inputs, then restarts at 0:



- Word 5: Status and a counter for multiplexing outputs to detect if one is disabled. HLTH looks at 16 words (256 outputs) per scan to find one that is disabled. It holds the last word location of the last scan. The block is overwritten on every scan. The value in the counter portion increases to maximum outputs then restarts at 0:

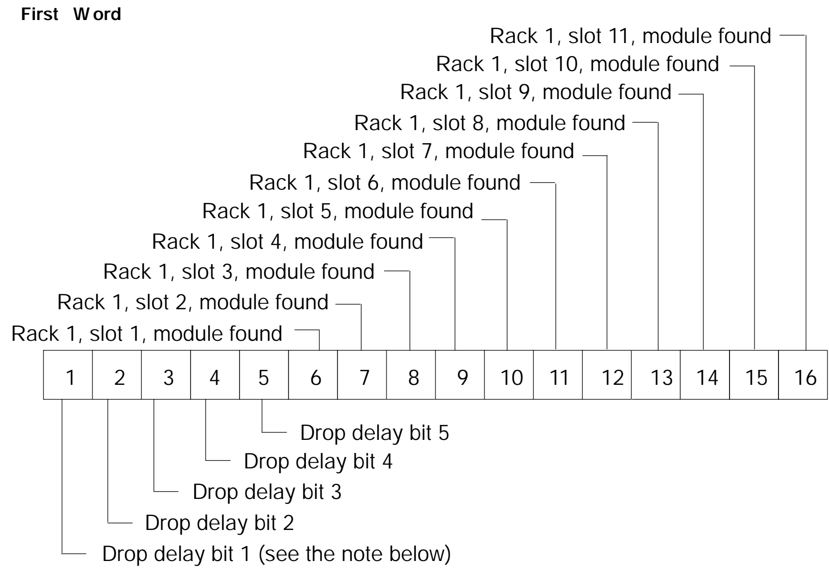


- Word 6: Hot Standby cable learned data
- Words 7 ... 10: Four words that define the learned condition of drop 1
- Words 11 ... 14: Four words that define the learned condition of drop 2



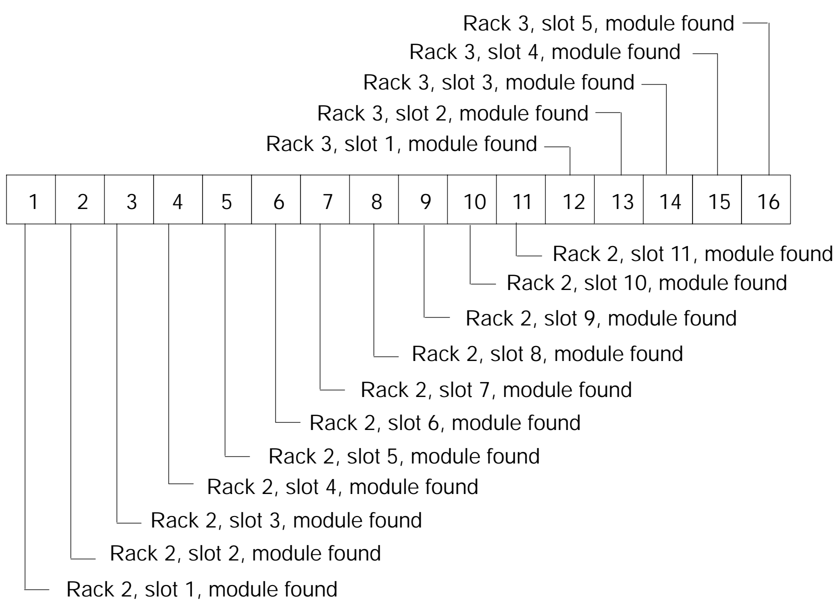
- Words 132 ... 135: Four words that define the learned condition of drop 32

The structure of the four words allocated to each drop are as follows:

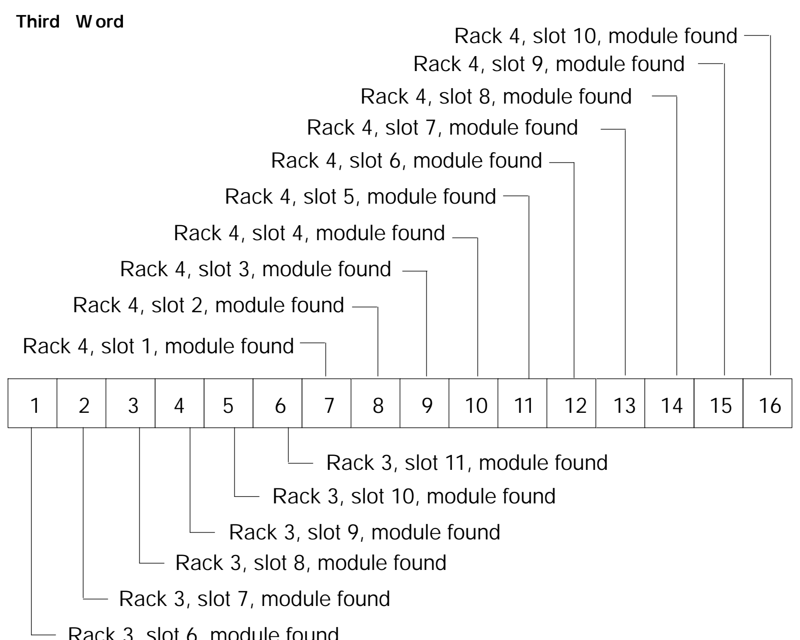


Note: Drop delay bits are used by the software to delay the monitoring of the drop for four scans after reestablishing communications with a drop. The delay value is for internal use only and needs no user intervention.

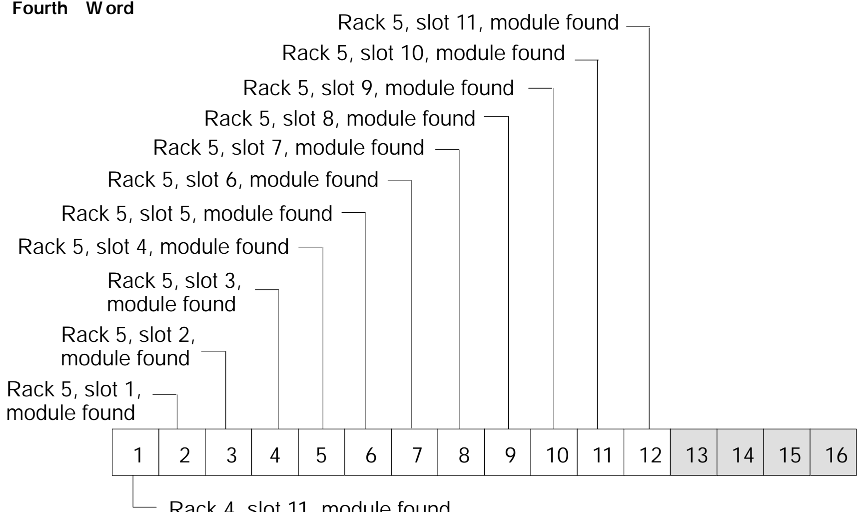
Second Word



Third Word



Fourth Word

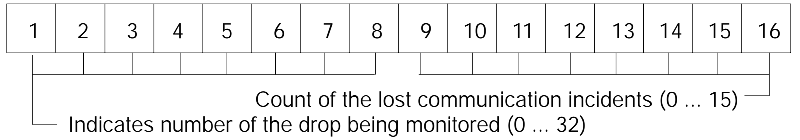


Middle Node Content—The Status Matrix

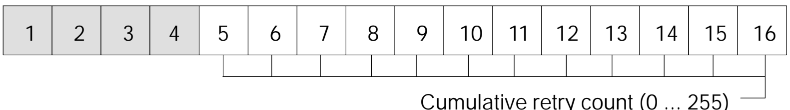
The 4*x* register entered in the middle node is the first in a block of contiguous holding registers that will comprise the *status* matrix. The *status* matrix is updated by the HLTH instruction during monitor mode (the top input is ON and the middle input is OFF).

The *status* matrix can range from 3 ... 132 registers in length. Below is a description of the words in the *status* matrix. The information from word 1 is contained in the displayed register in the middle node and the information from words 2 ... 132 is stored in the implied registers.

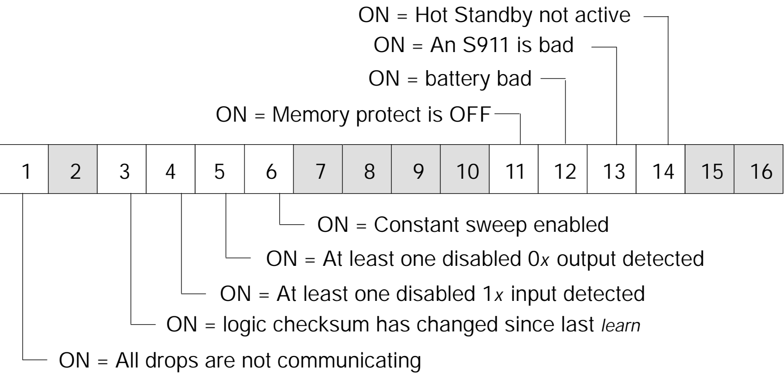
Word 1 is a counter for lost-communications at the drop being monitored:



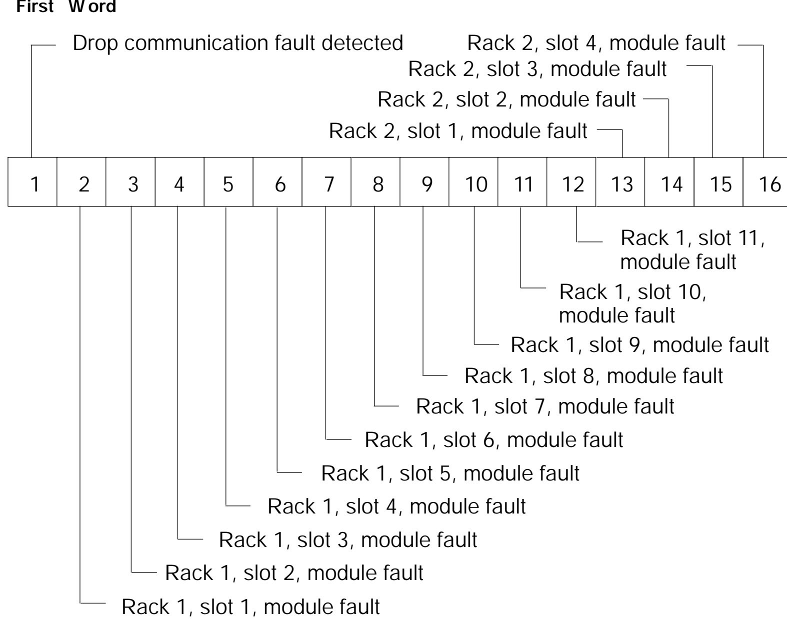
Word 2 is the cumulative retry counter for the drop being monitored (the drop number is indicated in the high byte of word 1):



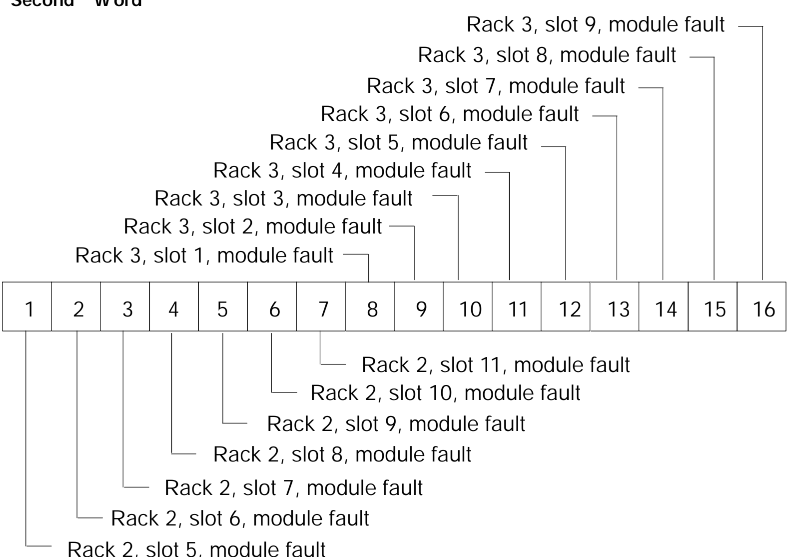
Word 3 updates PLC status (including Hot Standby health) on every scan:



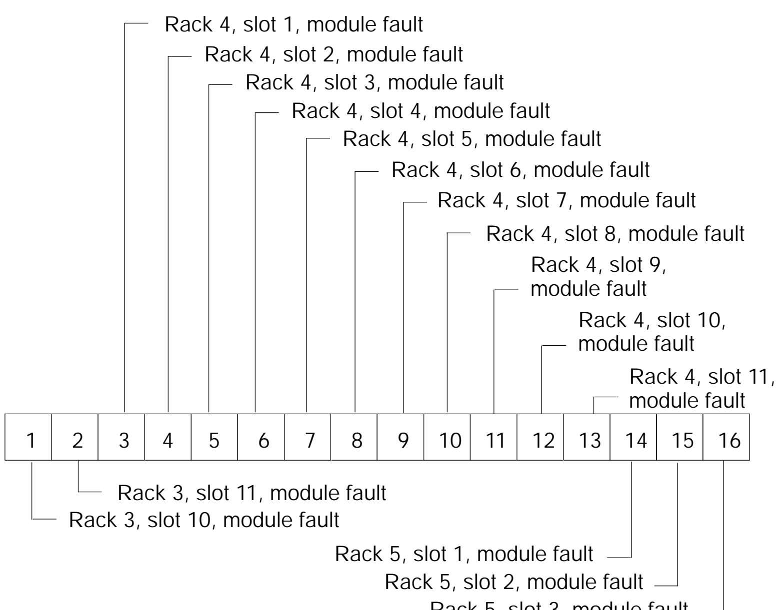
Words 4 ... 7 indicate drop 1 status; words 8 ... 11 indicate drop 2 status; etc., through words 129 ... 132, which indicate drop 32 status. The structure of the four words allocated to each drop is as follows:



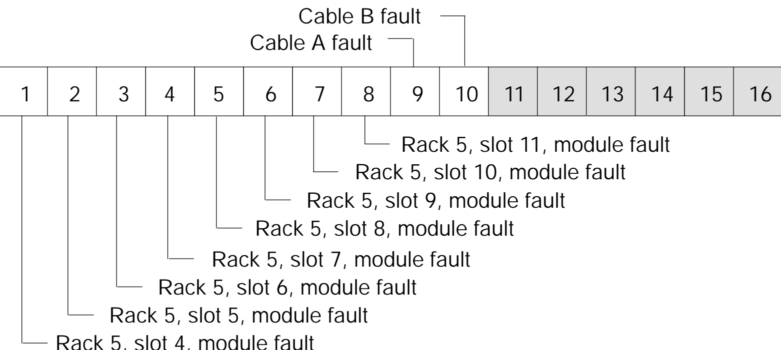
Second Word



Third Word



Fourth Word



Bottom Node Content

The decimal value entered in the bottom node is a function of how many I/O drops you want to monitor. Each drop requires four registers/matrix. The *length* value is calculated using the following formula:

length = (# of I/O drops x 4) + 3

This value gives you the number of registers in the *status* matrix. You only need to enter this one value as the *length* because the length of the history matrix is automatically increased by 3 registers—i.e., the size of the history matrix is *length* + 3.

11.6.5 HLTH Example

Suppose the HLTH instruction is going to be programmed to monitor the status of two remote I/O drops on a PLC network. The logic to *learn* the traffic cop is programmed for the first logic scan, then to *monitor* status in the subsequent scan. In this example, the RIO network is a single-cable system.

The *length* of the *status* matrix is determined by the formula:

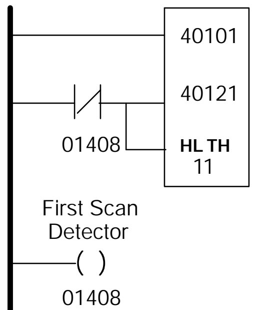
length = (*number of RIO drops* x 4) + 3

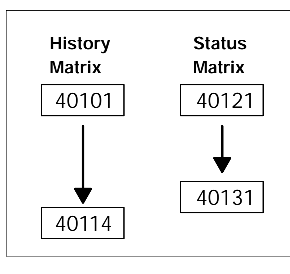
or

(2 x 4) + 3 = 11

The software will automatically add 3 to the *length* to establish a *history* matrix with 14 registers.

The two holding registers entered in the top and middle nodes of the instruction become the first registers in each matrix:





On the first scan, coil 1408 is OFF, and power is applied to all three inputs. Thus, the instruction executes a *learn* of the present configuration and sets the appropriate bits in the *history* matrix. The learn is for only a single-cable system.

On subsequent scans coil 1408 is ON and power is removed from the second and third inputs. This causes the instruction to monitor the status of the PLC and its two remote I/O drops. The appropriate health bits are set in the *status* matrix.

The third word of the *status* matrix is for the PLC. Words 4 ... 7 represent the status of drop 1, and words 8 .. 11 represent the status of drop 2. These status bits are updated each scan.

If all the I/O modules that have been mapped in the Traffic Cop are communicating, all the bits in the *status* matrix related to module health are OFF. If a module stops communicating, it's assigned bit will turn ON.

To see the cumulative retries for drop one, enter the value 5 in register 40101 (the first register in the *history* matrix). Do this in monitor mode. The HLTH instruction moves the cumulative retries for the drop into the second register of the *status* matrix (40122) The value can range from 0 ... 255; it rolls over to zero after reaching 255.

Chapter 12
Monitoring Distributed I/O System Status

- The DIO Status Table
- DIOH

12.1 The DIO Health Status Table

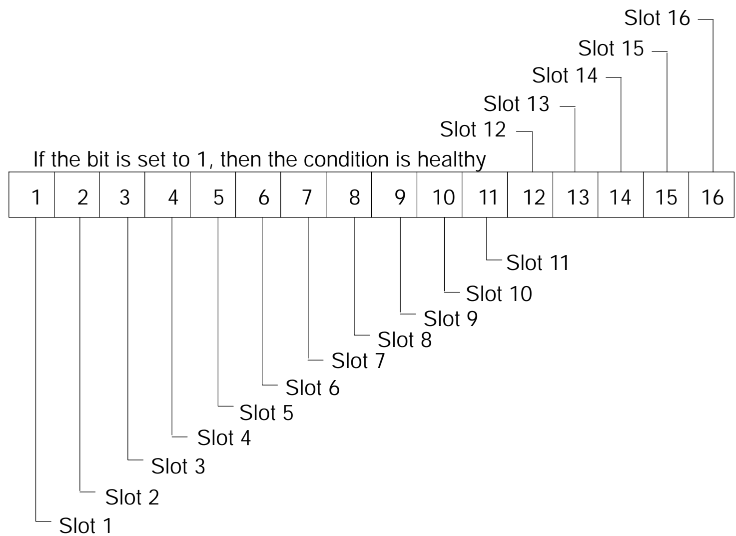
The distributed I/O (DIO) health tables allocates one 16-bit word for each configured drop in a DIO system. Up to 189 distributed drops are configurable on three networks, up to 63 drops per network. The Modbus Plus port on the PLC is used as the head processor for network 1, and two additional DIO option modules may be used in the local rack to support networks 2 and 3.

Note: The assignment of network 2 or network 3 to a particular DIO processor is handled automatically by the PLC. If two processors are installed when the system is powered up, the PLC selects the leftmost of the two as the processor for network 2 and the rightmost as the processor for network 3. If only one DIO processor is installed in the rack when the system is powered up, the PLC always selects it as the processor for network 2. If a second DIO processor is added to the rack after power-up, the PLC selects it as the processor for network 3 regardless of its position in the rack relative to the other processor.

The DIO table is divided into three sections, with 64 words reserved for each of the three possible networks:

W ord	Content
1	DIO drop 1 (first drop on network 1)
2	DIO drop 2
. . .	. . .
64	DIO drop 64 (last drop on network 1)
65	DIO drop 65 (first drop on network 2)
66	DIO drop 66
. . .	. . .
128	DIO drop 128 (last drop on network 2)
129	DIO drop 129 (first drop on network 3)
130	DIO drop 130
. . .	. . .
.192	DIO drop 192 (last drop on network 3)

Each drop can support one rack of I/O, with as many as 16 slots available (depending on the type of rack used at the drop. A bit in each word indicates the health of the module in the associated slot. The format of registers 1 ... 192 is:



Four conditions must be met before a module can indicate good health:

- The slot must be traffic copped
- The slot must contain a module with the correct personality
- Valid communications must exist between the I/O module and the DIO interface at the drop
- Valid communications must exist between the DIO interface at each drop and the head DIO processor in rack with the PLC

For a complete understanding of how to set up a distributed I/O network, refer to *Modicon Modbus Plus System Planning and Installation Guide*.

12.2 DIOH

The DIOH instruction lets you retrieve health data from a specified group of drops on the distributed I/O network. It accesses the DIO health status table, where health data for modules in up to 189 distributed drops is stored.

12.2.1 Characteristics

Size

Three nodes high

PLC Compatibility

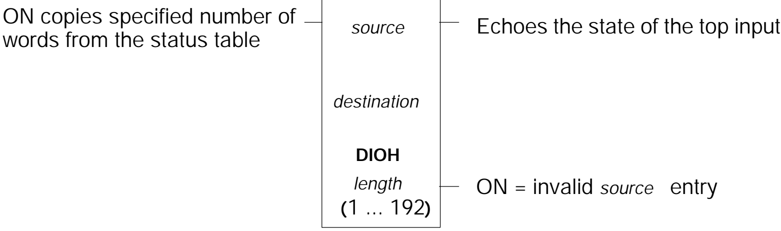
- Standard in the CPU 113 02, CPU 113 03, and CPU 213 04 Quantum Automation Series PLC
- Not available in other PLC types

Opcode

20 hex

12.2.2 Representation

Block Structure



Input

The DIOH instruction has one control input to the top node, initiates the retrieval of the specified status words from the DIO health table into the *destination* table.

Outputs

DIOH produces two possible outputs. The output from the top node echoes the state of the top input. The output from the bottom node goes ON if an invalid *source* constant is entered in the top node.

Top Node Content

The *source* value entered in the top node is a four-digit constant in the form *xxyy*, where:

- *xx* is a decimal value in the range 00 ... 16, indicating the slot number in which the relevant DIO processor resides. The value 00 can always be used to indicate the Modbus Plus ports on the PLC, regardless of the slot in which it resides.

- *yy* is a decimal value in the range 1 ... 64, indicating the drop number on the appropriate token ring.

For example, if you are interested in retrieving drop status starting at distributed drop #1 on a network being handled by a DIO processor in slot 3, enter 0301 in the top node.

Middle Node Content

The 4*x* register entered in the middle node is the first holding register in the *destination* table—i.e., in a block of contiguous registers where the retrieved health status information is stored.

Bottom Node Content

The integer value entered in the bottom node specifies the *length* —i.e., the number of 4*x* registers—in the *destination* table. The *length* is in the range 1 ... 64.

Note: If you specify a *length* that exceeds the number of registers available, the instruction will return status information only for the registers available. For example, if you specify the 63rd word in the DIOH health status table in the middle node register and then request a *length* of 5, the instruction will give you only two registers (the 63rd and 64th status words) in the destination table.

Chapter 13
Bypassing Networks with SKP

The SKP instruction is a standard instruction in all PLCs. It should be used with caution.

Warning! SKP is a dangerous instruction that should be used carefully . If inputs and outputs that normally effect control are unintentionally skipped (or not skipped), the result can create hazardous conditions for personnel and application equipment.

Three special Modsoft off-line functions—SKIP, SKPC, and SKPR—are also provided in the panel software for sequential function chart (SFC) applications. These three off-line functions are executed by the PLC as SKP instructions in on-line mode.

13.1 SKP

When a SKP instruction is implemented, skipped networks in the ladder logic program are not solved. SKP instructions can be used to reduce scan time and, in effect, establish subroutines within the scheduled logic.

A SKP operation cannot pass the boundary of a segment. No matter how many extra networks you specify to be skipped, the instruction will stop if it reaches the end of a segment.

☞ **Note:** A SKP instruction can be activated only if you specify in the configurator editor that skips are allowed.

13.1.1 Characteristics

Size
One node high

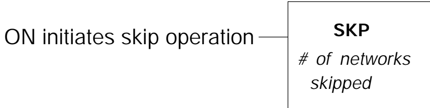
PLC Compatibility
Standard instruction in all PLC types

Opcode

- 0E hex if the number of networks to be skipped is specified as a constant
- 0F hex if the number of networks to be skipped is specified in a register

13.1.2 Representation in Ladder Logic

Block Structure



Input
SKP has one control input that initiates a skip network operation when it passes power. A SKP operation is performed on every scan while the input is ON.

Node Content

The value entered in the node specifies the number of networks to be skipped. The value can be:

- Specified explicitly as an integer constant in the range 1 ... 999
- Stored in a 3*x* input register
- Stored in a 4*x* holding register

The node value includes the network that contains the SKP instruction. The nodal regions in the network where the SKP resides that have not already been scanned will be skipped; this counts as one of the networks specified to be skipped. The CPU continues to skip networks until the total number of networks skipped equals the value specified.

13.1.3 A Simple SKP Example

Below is an illustration showing two contiguous networks of ladder logic. The first network contains a SKP instruction that specifies that two networks will be skipped when contact 10001 passes power.

When N.O. contact 10001 is closed, the remainder of the top network and all of the bottom network are skipped. The power flow display for these two networks becomes invalid, and your system displays an information message to that effect.

Coil 00193 is still controlled by contact 10003 because the solution of coil 00193 occurs before the SKP instruction. Coil 00116 will remain in whatever state it was in when the bottom network was skipped.

13.2 Off-line Instructions for Skipping Steps in Modsoft SFC

Modsoft panel software provides three additional off-line skip instructions that can be used when you are programming a sequential function chart (SFC) application:

- SKIP, which skips to a specific network in a permanent 0 (P0) or SFC step
- SKPC (skip constant), which lets you reduce scan time in an SFC or macro application by explicitly specifying a number of networks to be skipped
- SKPR (skip register), which lets you reduce scan time in an SFC or macro application using a value stored in a 3*x* or 4*x* register to specify the number of networks to be skipped

None of these skipping operations can pass the boundary of a segment. No matter how many extra networks you specify to be skipped, the instruction will stop if it reaches the end of a segment.

Warning! Because these three off-line functions are executed as SKP instructions in ladder logic, their use can be potentially dangerous. If inputs and outputs that normally effect control are unintentionally skipped (or not skipped), the result can create hazardous conditions for personnel and application equipment.

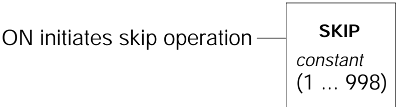
13.2.1 Characteristics

Size
One node high

PLC Compatibility
Not PLC-based instructions; reside in Modsoft panel software and are executed as SKP instructions by the PLC

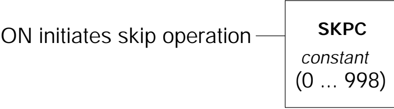
13.2.2 Off-line Representations

SKIP



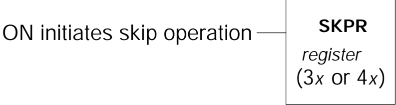
To invoke this instruction off-line, push <Alt> **F** and type **SKIP**

SKPC



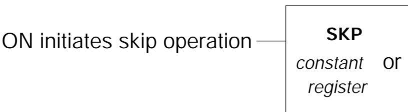
To invoke this instruction off-line, push <Alt> **F** and type **SKPC**

SKPR



To invoke this instruction off-line, push <Alt> **F** and type **SKPR**

13.2.3 On-line Representation



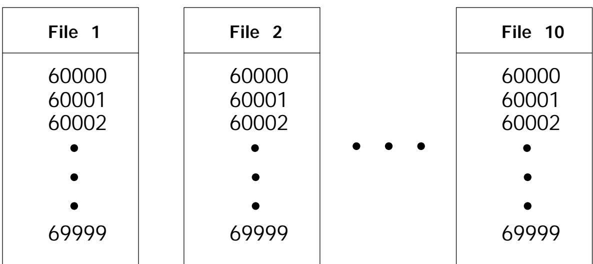
Chapter 14
Extended Memory Capabilities

- ☐ Extended Memory File Structure
- ☐ How Extended Memory Is Stored in System Memory
- ☐ XMWT
- ☐ XMRD

14.1 Extended Memory File Structure

Several of the 24-bit PLCs provide an optional capability for supporting *extended memory*. Extended memory is used for massive data storage in a group of files made up of storage registers. These extended memory storage registers use 6*x* reference numbers.

Extended memory provides up to ten files, and each file can contain as many as 10,000 registers ranging from 60000 ... 69999:



Optional sizes of extended memory are available for the various PLC models that support it:

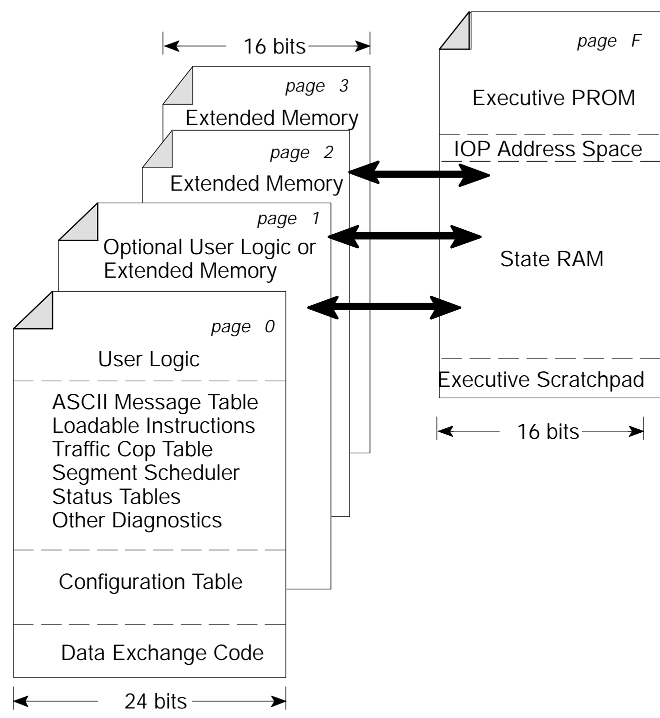
984B	E984-785	L984-785	Quantum Series
32K words	24K words	0K words	0K words
64K words	96K words	24K words	80K words
96K words		72K words	
		96K words	

The total memory available may be up to 128K words, with either 32K words or 64K words allocated for user logic memory so that:

- A 984B with 32K words of memory has no extended memory
- A 984B with 64K words of memory may use all 64K for user logic or 32K of user logic and 32K words of extended memory
- A 984B with 96K words of memory may use 32K for user logic and 64K for extended memory or 64K for user logic and 32K for extended memory
- A 984B with 128K words of memory may use 32K for user logic and 96K for extended memory or 64K for user logic and 64K for extended memory

14.2 How Extended Memory Is Stored in User Memory

Extended Memory consists of a bank of memory registers located on pages 1 ... 3 in system memory; these registers may be used as mass storage area for 984 holding registers or as a buffer for input registers. You can store additional state RAM data not being used in a particular application here.



Note: Pages 2 and 3 of Extended Memory contain 16 bit words, as do all pages except pages 0 and 1 in a 24 bit machine.

Pages 0 and 1 each contain 32K 24 bit words. If you choose 32K for extended memory, only page 0 is used, and page 1 is available for optional user logic.

14.3 XMWT

The XMWT instruction is used to write data from a block of input registers or holding registers in state RAM to a block of 6*x* registers in an extended memory file.

14.3.1 Characteristics

Size

Three nodes high

PLC Compatibility

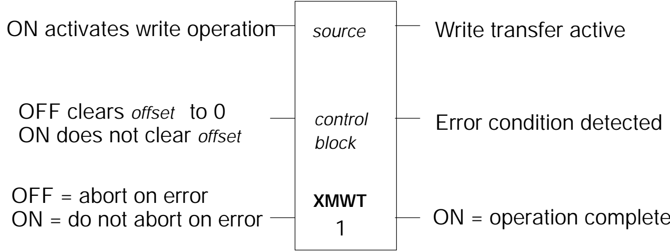
- Standard in 984B Chassis Mount PLCs E984-785 and L984-785 Slot Mount PLCs, and all Quantum Automation Series PLCs
- Unavailable in all other PLC types

Opcode

7E hex

14.3.2 Representation

Block Structure



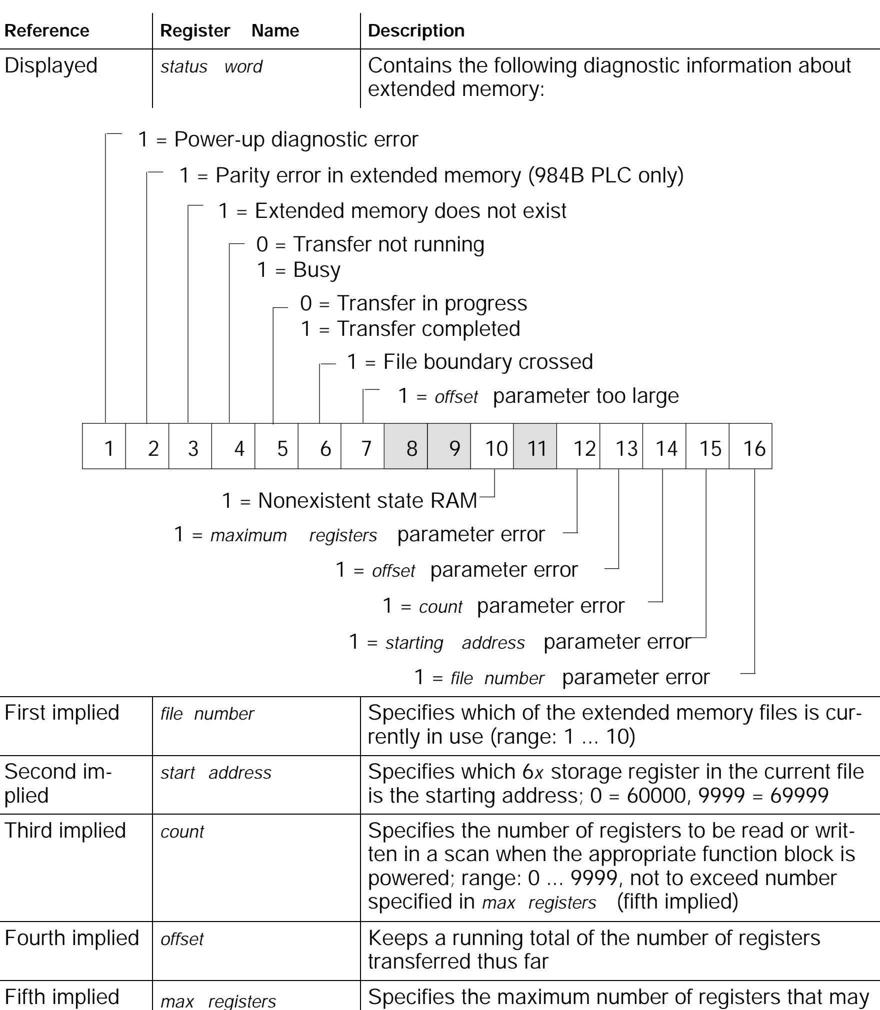
Top Node Content

The 3*x* or 4*x* register entered in the top node is the first in a block of contiguous *source* registers—i.e., input or holding registers whose contents will be written to 6*x* extended memory registers.

Middle Node Content

The 4*x* register entered in the middle node is the first of six contiguous holding registers in the extended memory *control block* .

Reference	Register Name	Description
Displayed	*status word*	Contains the following diagnostic information about extended memory:



Reference	Register Name	Description
First implied	*file number*	Specifies which of the extended memory files is currently in use (range: 1 ... 10)
Second implied	*start address*	Specifies which 6*x* storage register in the current file is the starting address; 0 = 60000, 9999 = 69999
Third implied	*count*	Specifies the number of registers to be read or written in a scan when the appropriate function block is powered; range: 0 ... 9999, not to exceed number specified in *max registers* (fifth implied)
Fourth implied	*offset*	Keeps a running total of the number of registers transferred thus far
Fifth implied	*max registers*	Specifies the maximum number of registers that may be transferred when the function block is powered (range: 0 ... 9999)

If you are in multi-scan mode, these six registers should be reserved for use only by this instruction.

Bottom Node Content

The bottom node contains the constant value 1, which cannot be changed.

14.4 XMRD

The XMRD instruction is used to copy a table of 6*x* extended memory registers to a table of 4*x* holding registers in state RAM.

14.4.1 Characteristics

Size

Three nodes high

PLC Compatibility

- Standard in 984B Chassis Mount PLCs E984-785 and L984-785 Slot Mount PLCs, and all Quantum Automation Series PLCs

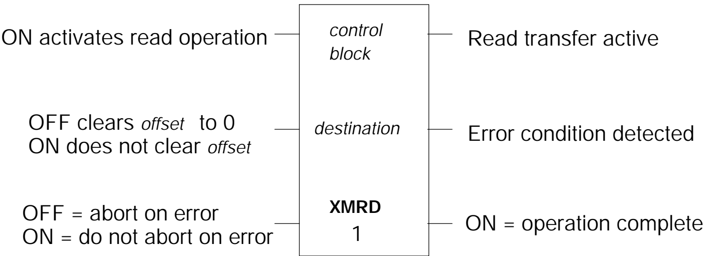
- Unavailable in all other PLC types

Opcode

9E hex

14.4.2 Representation

Block Structure



Top Node Content

The 4*x* register entered in the top node is the first of six contiguous holding registers in the extended memory *control block* .

Reference	Register Name	Description
Displayed	*status word*	Contains the following diagnostic information about extended memory:

1 = Power-up diagnostic error
1 = Parity error in extended memory (984B PLC only)
1 = Extended memory does not exist
0 = Transfer not running
1 = Busy
0 = Transfer in progress
1 = Transfer completed
1 = File boundary crossed
1 = *offset* parameter too large

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1 = Nonexistent state RAM
1 = *maximum registers* parameter error
1 = *offset* parameter error
1 = *count* parameter error
1 = *starting address* parameter error
1 = *file number* parameter error

Reference	Register Name	Description
First implied	*file number*	Specifies which of the extended memory files is currently in use (range: 1 ... 10)
Second implied	*start address*	Specifies which 6*x* storage register in the current file is the starting address; 0 = 60000, 9999 = 69999
Third implied	*count*	Specifies the number of registers to be read or written in a scan when the appropriate function block is powered; range: 0 ... 9999, not to exceed number specified in *max registers* (fifth implied)
Fourth implied	*offset*	Keeps a running total of the number of registers transferred thus far
Fifth implied	*max registers*	Specifies the maximum number of registers that may be transferred when the function block is powered (range: 0 ... 9999)

If you are in multi-scan mode, these six registers should be unique to this function block.

Middle Node Content

The middle node is the first 4*x* holding register in a table of registers that receive the transferred data from the 6*x* extended memory storage registers.

Bottom Node Content

The bottom node contains the constant value 1, which cannot be changed.

Chapter 15
ASCII Communication Instructions

Most PLCs that support ASCII messaging use instructions called READ and WRIT to handle the sending of messages to display devices and the receiving of messages from input devices. These instructions provide the routines necessary for communication between an ASCII message table inside the PLC's system memory and an interface module at the Remote I/O drops.

An exception is the Micro PLCs, where messaging between the PLC and the I/O is handled by a single instruction called COMM. COMM provides both read and write functions. It gives you the ability to use one of the local ports on the Micro PLC as the messaging port or, if the Micro is a parent, one of the ports on a child PLC as the messaging port.

In the Quantum PLCs, ASCII message handling may be handled differently in that the message table can be actually programmed in an ESI 062 ASCII module. In this case, the messaging operations between the PLC and the Quantum I/O module can be managed with a loadable instruction called ESI, described in section 22.5.

15.1 READ

The READ instruction provides the ability to read data from an ASCII input device (keyboard, bar code reader, etc.) into the PLC's memory via its RIO network. The connection to the ASCII device is made at an RIO interface.

In the process of handling the messaging operation, READ performs the following functions:

- Verifies the correctness of the ASCII communication parameters—e.g., the port number, the message number
- Verifies the lengths of variable data fields
- Performs error detection and recording
- Reports RIO interface status

READ requires two tables of registers—a *destination* table where retrieved variable data (the message) is stored, and a *control block* where comm port and message parameters are identified.

15.1.1 Characteristics

Size
Three nodes high

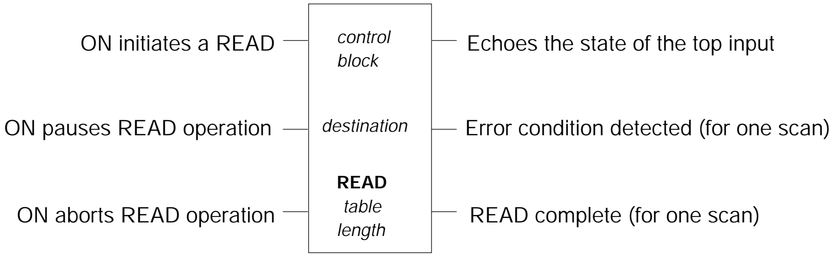
PLC Compatibility

- Standard in all PLC types that support S901 or S908 remote I/O communications
- Unavailable in all other PLC types

Opcode
1E hex

15.1.2 Representation

Block Structure



Inputs

READ has three control inputs that can start, pause, and abort the READ operation.

Outputs

READ can produce three possible outputs. The output from the middle node goes ON to if an error has been detected in the communication or if the operation has timed out. The output from the bottom node goes ON when the READ operation is completed.

Top Node Content

The 4*x* register entered in the top node is the first of seven contiguous holding register in the *control block* .

Register	Definition
Displayed	Port number and error code:

PLC error code

0 0 0 1 Error in the input to RIOP from ASCII device
0 0 1 0 Exception response from RIOP—bad data
0 0 1 1 Sequenced number from RIOP differs from expected value
0 1 0 0 User register checksum error—often caused by altering READ registers while the block is active
0 1 0 1 Invalid port or message number detected
0 1 1 0 User-initiated abort—bottom input energized
0 1 1 1 No response from drop—communication error
1 0 0 0 Node aborted because of SKP instruction
1 0 0 1 Message area scrambled—reload memory
1 0 1 0 Port not configured in the traffic cop
1 0 1 1 Illegal ASCII request
1 1 0 0 Unknown response from ASCII port
1 1 0 1 Illegal ASCII element detected in user logic
1 1 1 1 RIOP in the PLC is down

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Input from the ASCII device not compatible with format
Input buffer overrun—data received too quickly at RIOP
USART error—bad byte received at RIOP
Illegal format—not received properly by RIOP
ASCII device off-line—check cabling
ASCII message terminated early (in keyboard mode)
Comm port # (1 ... 32)

Register	Definition
First implied	Message number
Second implied	Number of registers required to satisfy format
Third implied	Count of the number of registers transmitted thus far
Fourth implied	Status of the solve
Fifth implied	
Sixth implied	Checksum of registers 0 ... 5

Middle Node Content

The middle node contains the first 4*x* register in a *destination* table. Variable data in a READ message are written into this table. The *length* of the table is defined in the bottom node.

Consider this READ message:

```
please enter password: AAAAAAAAAA
```

(Embedded Text) (Variable Data)

Note: An ASCII READ message may contain the embedded text—placed inside quotation marks—as well as the variable data in the format statement—i.e., the ASCII message.

The 10-character ASCII field **AAAAAAAAAA** is the variable data field; variable data must be entered via an ASCII input device.

Bottom Node Content

The integer value entered in the bottom node specifies the *length* of the *destination* table—i.e., the number of registers where the message data will be stored. The *length* can range from 1 ... 255 in a 16-bit CPU and from 1 ... 999 in a 24-bit CPU.

15.2 WRIT

The WRIT instruction sends a message from the PLC over the RIO communications link to an ASCII display (screen, printer, etc.).

In the process of sending the messaging operation, WRIT performs the following functions:

- Verifies the correctness of the ASCII communication parameters—e.g., the port number, the message number
- Verifies the lengths of variable data fields
- Performs error detection and recording
- Reports RIO interface status

WRIT requires two tables of registers—a *source* table where variable data (the message) is copied, and a *control block* where comm port and message parameters are identified.

15.2.1 Characteristics

Size
Three nodes high

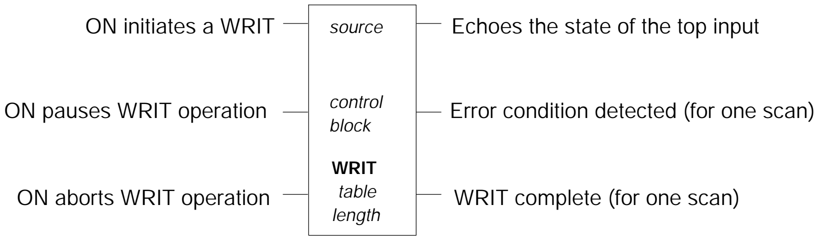
PLC Compatibility

- Standard in all PLC types that support S901 or S908 remote I/O communications
- Unavailable in all other PLC types

Opcode
3E hex

15.2.2 Representation

Block Structure



Inputs

WRIT has three control inputs that can start, pause, and abort the WRIT operation.

Outputs

WRIT can produce three possible outputs. The output from the middle node goes ON to if an error has been detected in the communication or if the operation has timed out. The output from the bottom node goes ON when the WRIT operation is completed.

Top Node Content

The top node contains the first 3*x* or 4*x* register in a *source* table whose length is specified in the bottom node. This table contains the data required to fill the variable field in a message. Consider the following WRIT message

```
vessel #1 temperature is: III
```

The 3-character ASCII field `III` is the variable data field; variable data are loaded, typically via DX moves, into a table of variable field data.

Middle Node Content

The 4*x* register entered in the middle node is the first of seven contiguous holding registers in the *ASCII control block* :

Register	Definition
Displayed	Port number and error code: *PLC error code* (bits 1–4) 0 0 0 1 Error in the input to RIOP from ASCII device 0 0 1 0 Exception response from RIOP—bad data 0 0 1 1 Sequenced number from RIOP differs from expected value 0 1 0 0 User register checksum error—often caused by altering READ registers while the block is active 0 1 0 1 Invalid port or message number detected 0 1 1 0 User-initiated abort—bottom input energized 0 1 1 1 No response from drop—communication error 1 0 0 0 Node aborted because of SKP instruction 1 0 0 1 Message area scrambled—reload memory 1 0 1 0 Port not configured in the traffic cop 1 0 1 1 Illegal ASCII request 1 1 0 0 Unknown response from ASCII port 1 1 0 1 Illegal ASCII element detected in user logic 1 1 1 1 RIOP in the PLC is down 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 6: Input from the ASCII device not compatible with format 7: Input buffer overrun—data received too quickly at RIOP 8: USART error—bad byte received at RIOP 9: Illegal format—not received properly by RIOP 10: ASCII device off-line—check cabling 11: ASCII message terminated early (in keyboard mode) 12–16: Comm port # (1 ... 32)
First implied	Message number
Second implied	Number of registers required to satisfy format
Third implied	Count of the number of registers transmitted thus far
Fourth implied	Status of the solve
Fifth implied	
Sixth implied	Checksum of registers 0 ... 5

Bottom Node Content

The integer value entered in the bottom node specifies the *length* of the *source* table—i.e., the number of registers where the message data will be stored. The *length* can range from 1 ... 255 in a 16-bit CPU and from 1 ... 999 in a 24-bit CPU.

15.3 Formatting Messages for ASCII READ/WRIT Operations

The ASCII messages used in the READ and WRIT instructions can be created via your panel software using the format specifiers described below. Format specifiers are character symbols that indicate:

- The ASCII characters used in the message
- Register content displayed in ASCII character format
- Register content displayed in hexadecimal format
- Register content displayed in integer format
- Subroutine calls to execute other message formats

15.3.1 Format Specifiers

Specifier		
/	Meaning	ASCII return (CR) and linefeed (LF)
	Field width	None (defaults to 1)
	Prefix	None (defaults to 1)
	Input format	Outputs CR, LF; no ASCII characters accepted
	Output format	Outputs CR, LF

Specifier		
“ ”	Meaning	Enclosure for octal control code
	Field width	Three digits enclosed in double quotes
	Prefix	None
	Input format	Accepts three octal control characters
	Output format	Outputs three octal control characters

Specifier		
‘ ’	Meaning	Enclosure for ASCII text characters
	Field width	1 ... 128 characters
	Prefix	None (defaults to 1)
	Input format	Inputs number of upper and/or lower case printable characters specified by the field width
	Output format	Outputs number of upper and/or lower case printable characters specified by the field width

Specifier	Attribute	Description
X	Meaning	Space indicator—e.g., **14X** indicates 14 spaces left open from the point where the specifier occurs
	Field width	1 ... 255 spaces
	Prefix	None (defaults to 1)
	Input format	Inputs specified number of spaces
	Output format	Outputs specified number of spaces
( )	Meaning	Repeat contents of the parentheses—e.g., **2 (4X, I5)** says repeat **4X, I5** two times
	Field width	None
	Prefix	1 ... 255
	Input format	Repeat format specifiers in parentheses the number of times specified by the prefix
	Output format	Repeat format specifiers in parentheses the number of times specified by the prefix
I	Meaning	Integer—e.g., **I5** specifies five integer characters
	Field width	1 ... 8 characters
	Prefix	1 ... 99
	Input format	Accepts ASCII characters 0 ... 9. If the field width is not satisfied, the most significant characters in the field are padded with zeros
	Output format	Outputs ASCII characters 0 ... 9. If the field width is not satisfied, the most significant characters in the field are padded with zeros. The overflow field consists of asterisks.
L	Meaning	Leading zeros—e.g., **L5** specifies five leading zeros
	Field width	1 ... 8 characters
	Prefix	1 ... 99
	Input format	Accepts ASCII characters 0 ... 9. If the field width is not satisfied, the most significant characters in the field are padded with zeros
	Output format	Outputs ASCII characters 0 ... 9. If the field width is not satisfied, the most significant characters in the field are padded with zeros. The overflow field consists of asterisks.
A	Meaning	Alphanumeric—e.g., **A27** specifies 27 alphanumeric characters, no suffix allowed
	Field width	1 ... 99
	Prefix	None (defaults to 1)
	Input format	Accepts any 8-bit character except reserved delimiters such as CR, LF, ESC, BKSPC, DEL.
	Output format	Outputs any 8-bit character

O	Meaning	Octal—e.g., **O2** specifies two octal characters
	Field width	1 ... 6 characters
	Prefix	1 ... 99
	Input format	Accepts ASCII characters 0 ... 7. If the field width is not satisfied, the most significant characters are padded with zeros.
	Output format	Outputs ASCII characters 0 ... 7. If the field width is not satisfied, the most significant characters are padded with zeros. No overflow indicators.

B	Meaning	Binary—e.g., **B4** specifies four binary characters
	Field width	1 ... 16 characters
	Prefix	1 ... 99
	Input format	Accepts ASCII characters 0 and 1. If the field width is not satisfied, the most significant characters are padded with zeros.
	Output format	Outputs ASCII characters 0 and 1. If the field width is not satisfied, the most significant characters are padded with zeros. No overflow indicators.

H	Meaning	Hexadecimal—e.g., **H2** specifies two hex characters
	Field width	1 ... 4 characters
	Prefix	1 ... 99
	Input format	Accepts ASCII characters 0 ... 9 and A ... F. If the field width is not satisfied, the most significant characters are padded with zeros.
	Output format	Outputs ASCII characters 0 ... 9 and A ... F. If the field width is not satisfied, the most significant characters are padded with zeros. No overflow indicators.

15.4 COMM

The COMM instruction gives you the ability to read and write canned messages to/from ASCII character input/output devices via one of the built-in communication ports on a Micro PLC or, if the PLC is a parent, via a comm port on one of the child PLCs on the expansion link.

15.4.1 Characteristics

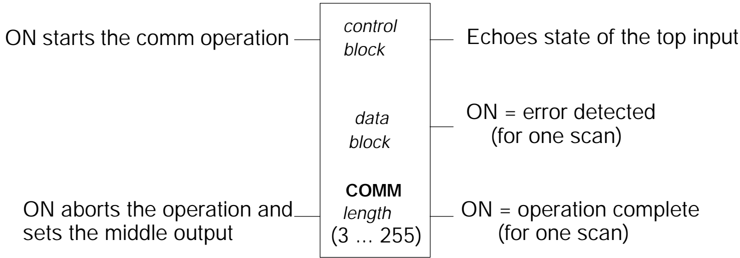
Size
Three nodes high

PLC Compatibility
Available only in the Micro PLCs

Opcode
hex

15.4.2 Representation

Block Structure



Top Node Content
The 4*x* register entered in the top node is the first of 10 contiguous holding registers in the *control block* :

Register	Content
Displayed	Message format (see page 310)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Error (bits 13–16)	Code
No error	0 0 0 0
Unconfigured child selected in fifth implied register	0 0 0 1
COMM instruction active longer than the time specified in ninth implied register	0 0 1 0
Invalid operation type (format) selected in displayed register	0 0 1 1
Number of data fields specified in second implied register bigger than the constant in the bottom node of the COMM instruction	0 1 0 0
Receiver buffer error detected	0 1 0 1
Bad integer value detected in incoming or outgoing data	0 1 1 0
Bad hex value detected in incoming or outgoing data	0 1 1 1
Number of bytes to be transmitted exceeds transmit buffer size— 256 bytes for the local ASCII port, 64 bytes for each child	1 0 0 0
No local port configured for ASCII	1 0 0 1
Port in use by parent/child	1 0 1 0
Child is unhealthy	1 0 1 1
DSR line is active	1 1 0 0

Register	Content
First implied	COMM error status
Second implied	Number of data fields provided/expected
Third implied	Number of data fields processed (updated by the instruction)
Fourth implied	
Fifth implied	Port number (1 for a port on the local PLC, 2 ... 5 if local PLC is a parent using a port on a child)
Sixth implied	
Seventh implied	
Eighth implied	
Ninth implied	Active status timer

Middle Node Content

The middle node contains the first 4*x* register of the *data block* —a table where variable message data is placed. In a read operation, the *data block* is a destination table; in a write operation, it is a *source* table.

Bottom Node Content

The integer value entered in the bottom node specifies the *length* —i.e., the number of registers—in the *data block* . The *length* can range from 3 ... 255.

15.4.3 Message Formats for the COMM Instruction

The ASCII communications capability in the Micros is for simple canned message formats. The table below shows the formats available for the COMM instruction:

Canned Message Format	Decimal Format Indicator
Flush input buffer	1001
Flush input byte, no CR/LF	1001
Control/Monitor signals*	1002
Read ASCII character, no CR/LF	1010
Write ASCII character, no CR/LF	1110
Read ASCII character, CR/LF	1020
Write ASCII character, CR/LF	1120
Read integer (1 ... 4), no CR/LF	1031 ... 1034
Write integer (1 ... 4), no CR/LF	1131 ... 1134
Read integer (1 ... 4), CR/LF	1041 ... 1044
Write integer (1 ... 4), CR/LF	1141 ... 1144
Read hex (1 ... 4), no CR/LF	1051 ... 1054
Write hex (1 ... 4), no CR/LF	1151 ... 1154
Read hex (1 ... 4), CR/LF	1061 ... 1064
Write hex (1 ... 4), CR/LF	1161 ... 1164

* Some special requirements in the middle and bottom node are implemented when *Control/Monitor signals* format is used (see page 313).

Note: The difference between CR/LF and no CR/LF formats is the way in which they handle carriages and linefeeds:

- For a write operation with CR/LF, the COMM instruction automatically sends a carriage return/linefeed after the selected number of items is sent. For a write operation with no CR/LF, the COMM instruction does not automatically send any carriage returns or linefeeds.
- For a read operation with CR/LF, the format is satisfied when either the selected number of items is input—i.e., taken out of the output buffer—or when you input a carriage return or linefeed; in the second case, the CR/LF is not put into any register. For a read operation with no CR/LF, inputting the selected number of items is the only way to satisfy the format

ASCII Character Format	Format numbers	1010, 1110, 1020, 1120	
	General usage	Sending/receiving ASCII characters or 8-bit data. The data is packed two characters per 4*x* register, the first character in the most significant eight bits of the register and the second character in the eight least significant bits.	
	Usage in a write operation	No auto CR/LF	Format satisfied after *n* data fields output from registers
		Auto CR/LF	Format satisfied after *n* characters output from registers and CR/LF output
	Usage in a read operation	No auto CR/LF	Format satisfied after *n* characters input to registers
		Auto CR/LF	Format satisfied after *n* characters input to registers or CR/LF received in buffer
Integer (1 ... 4) Format	Format numbers	1031 ... 1034, 1131 ... 1134, 1041 ... 1044, 1141 ... 1144	
	General usage	Sending/receiving integer data fields. The data is packed as 1 ... 4 digits (depending on format number selected) per 4*x* register and is right-justified with the first digit in the data field in the leftmost position	
	Usage in a write operation	No auto CR/LF	Format satisfied after *n* characters output from registers
		Auto CR/LF	Format satisfied after *n* data fields output from registers and CR/LF output
	Usage in a read operation	No auto CR/LF	Format satisfied after *n* integers input to registers
		Auto CR/LF	Format satisfied after *n* integers input to registers or CR/LF received in buffer

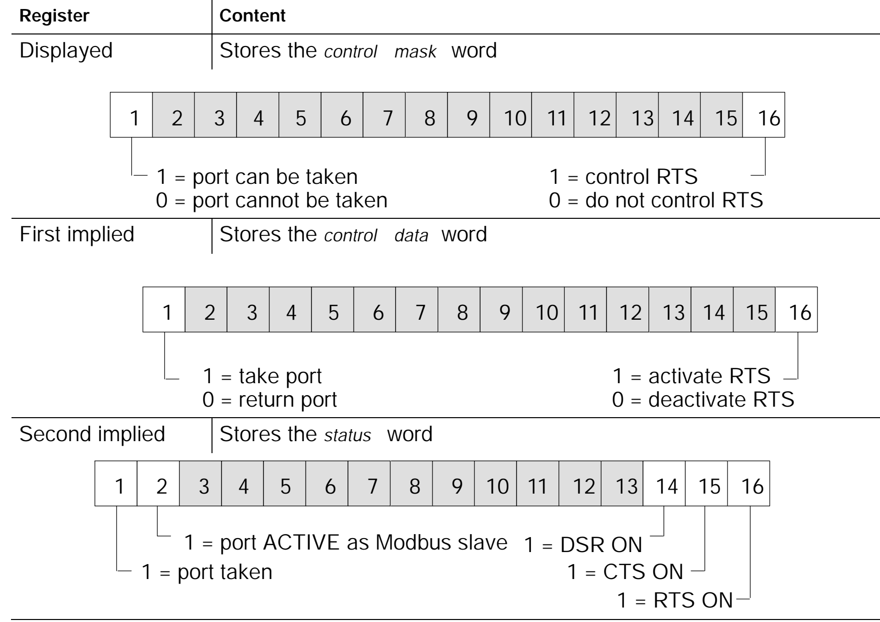
Hex (1 ... 4) Format	Format numbers	1051 ... 1054, 1151 ... 1154, 1061 ... 1064, 1161 ... 1164	
	General usage	Sending/receiving hex data fields. The data is packed as 1 ... 4 digits (depending on format number selected) per 4*x* register and is right-justified with the first digit in the data field in the leftmost position.	
	Usage in a write operation	No auto CR/LF	Format satisfied after *n* data fields output from registers
		Auto CR/LF	Format satisfied after *n* data fields output from registers and CR/LF output
	Usage in a read operation	No auto CR/LF	Format satisfied after *n* integers input to registers
		Auto CR/LF	Format satisfied after *n* integers input to registers or CR/LF received in buffer
Flush Input Buffer Format	Format numbers	1000	
	General usage	Flushing the input buffer. In the local PLC, the buffer is flushed immediately—i.e., at logic solve time. If a parent is using the comm port of a child for the ASCII operation, the flush is done when the child receives the request from the parent—the parent will send this request at the end of scan	
	Usage in a read operation	All bytes in the input buffer will be discarded	
	General usage	Flushing a number of bytes from the input buffer. In the local PLC, the bytes are flushed immediately. If a parent is using the comm port of child for the ASCII operation, the flush is done when the child receives the request from the parent—the parent will send this request at the end of scan.	
	Usage in a read operation	The specified number of bytes in the input buffer will be discarded.	

15.4.4 Special Set-up Considerations for Control/Monitor Signals Format

To control and monitor the signals used in the messaging communication, specify code 1002 in the first register of the control block (the register displayed in the top node). Via this format, you can control the RTS and CTS lines on the port used for messaging.

Tip In this format, only the local port can be used for messaging—i.e., a parent PLC cannot monitor or control the signals on a child port. Therefore, the port number specified in the fifth implied node of the control block must always be 1.

The first three registers in the *data block* (the displayed register and the first and second implied registers in the middle node) have predetermined content:



Register	Content
Displayed	Stores the *control mask* word
First implied	Stores the *control data* word
Second implied	Stores the *status* word

These three *data block* registers are required for this format, and therefore the allowable range for the *length* value (specified in the bottom node) is 3 ... 255.

15.5 ASCII Character Set

ASCII Character	Decimal Value	Hex Value
Bell	7	07
Linefeed	10	0A
Formfeed	12	0C
Carriage return	13	0D
→	26	1A
←	27	1B
Space	32	20
!	33	21
”	34	22
#	35	23
$	36	24
%	37	25
&	38	26
’	39	27
(	40	28
)	41	29
*	42	2A
+	43	2B
’	44	2C
	45	2D
.	46	2E
/	47	2F
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
:	58	3A
;	59	3B
<	60	3C
=	61	3D
>	62	3E
?	63	3F
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44
E	69	45
F	70	46
G	71	47
H	72	48
I	73	49
J	74	4A
K	75	4B
L	76	4C
M	77	4D
N	78	4E
O	79	4F
P	80	50
Q	81	51
R	82	52
S	83	53
T	84	54
U	85	55
V	86	56
W	87	57
X	88	58
Y	89	59
Z	90	5A
[	91	5B
]	93	5D
^	94	5E

ASCII Character	Decimal Value	Hex Value
_	95	5F
a	97	61
b	98	62
c	99	63
d	100	64
e	101	65
f	102	66
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76

ASCII Character	Decimal Value	Hex Value
w	119	77
x	120	78
y	121	79
z	122	7A
{	123	7B
|	124	7C
}	125	7D
ü	129	81
ä	132	84
ö	148	94
¢	155	9B
£	156	9C
ñ	164	A4
█	219	DB
α	224	E0
β	225	E1
Σ	228	E4
σ	229	E5
μ	230	E6
Ω	234	EA
∞	236	EC
ε	238	EE
÷	246	F6

Chapter 16
Sequential Control Instructions

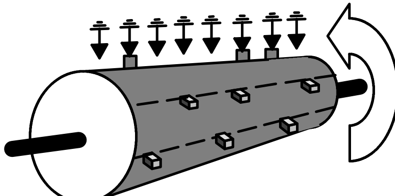
Modicon provides a set of instructions that emulate the operations of a Tenor drum in ladder logic. Two instructions—DRUM and ICMP—along with a DOS-based user interface, are provided in a software loadable package for most PLCs. The functionality of these two loadables is combined in a single instruction—SCIF—which is a standard offering in the Micro and Quantum PLC firmwares.

16.1 The Tenor Drum Model

The two operations described in this chapter—drum sequencing and input comparison (ICMP)—have been designed to emulate mechanical tenor drum operations in ladder logic. The tenor drum sequencer was introduced in the early 1900's and is still used today in applications that require simultaneous control of multiple motors, valves, solenoids, etc. at different steps in a process.

16.1.1 A Mechanical Tenor Drum

It works much like a piano roll. A cylinder consists of a series of rows of cams and flat surfaces. Each row represents a step in a process, and each cam represents a change of state for a mechanical device in the process. The cylinder rotates in a single direction so that each row passes a stationary string of contacts, one row at a time. As the cams in a given row meet the contacts, mechanical state changes take place for that step in the process.



A Mechanical Tenor Drum

Using the instructions described in this chapter, you can set up a step data table with a 16-bit register to represent each step in the process being controlled. The logic scans the table from top to bottom, treating each 1 value in a register like a cam and each 0 like a flat surface in a row on the mechanical tenor drum:

Register	Register Content																	
displayed																		
first implied																		
second implied																		
third implied	Step Data Table Set-up Registers																	
fourth implied																		
fifth implied																		
sixth implied	1	0	1	1	1	0	0	1	1	0	0	1	1	1	0	0	0	step 1
seventh implied	1	0	0	1	1	1	0	0	1	1	0	0	0	0	1	1	1	step 2
eighth implied	0	0	0	0	1	1	0	1	1	0	0	0	0	1	0	0	0	step 3
								⋮										
last	0	1	1	0	1	1	0	0	0	0	1	0	0	0	1	1	0	last step

These instructions combine the concept of the mechanical tenor drum with the added power and flexibility of the PLC to provide:

- ☐ Reduced down-time due to the elimination of several moving parts
- ☐ Sequencing operations that can be easily programmed and maintained
- ☐ More accuracy in terms of timing between process steps
- ☐ More flexibility in setting dwell, clamp, and hold times

Modern drum sequencer applications include tire and rubber molding, injection molding, die casting, plating, bottling, and other batch-oriented uses.

16.1.2 Drum and ICMP Operations

The drum sequencing operation maps a predefined bit pattern to the outputs on the PLC in a sequential, step-by-step fashion. The input comparison operation matches inputs coming from the field devices with a predefined table of bit patterns for each step of the drum.

Using drum and ICMP operations together allows the programmer to fire outputs and compare the status of the inputs against a predefined status. If a mismatch occurs, the process is halted.

16.2 DRUM

The DRUM instruction operates on a table of 4*x* registers containing data representing each step in a sequence. The number of registers associated with this *step data table* depends on the number of steps required in the sequence. You can pre-allocate registers to store data for each step in the sequence, thereby allowing you to add future sequencer steps without having to modify application logic.

DRUM incorporates an output mask that allows you to selectively mask bits in the register data before writing it to coils. This is particularly useful when all physical sequencer outputs are not contiguous on the output module. Masked bits are not altered by the DRUM instruction, and may be used by logic unrelated to the sequencer.

16.2.1 Characteristics

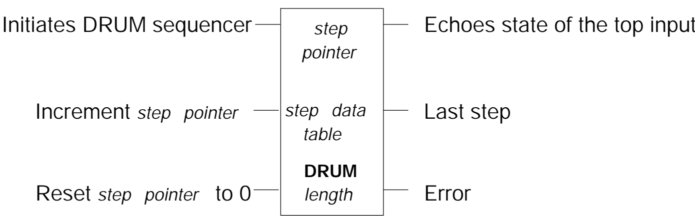
Size
Three nodes high

PLC Compatibility
Available as a loadable for all PLC types except the Micro and Quantum Automation Series PLCs

Opcode
FE hex (default)

16.2.2 Representation

Block Structure



Inputs
DRUM has three control inputs. When the input to the top node is ON, the drum operation is initiated. When the input to the middle node is

ON, the *step pointer* increments to the next step. When the input to the bottom node is ON, the s*tep pointer* is reset to 0.

Outputs

DRUM can produce three possible outputs. The output from the top node echos the state of top input. The output from the middle node goes ON for the last step—i.e., when the *step pointer* value = *length*. The output from the bottom node goes ON if an error is detected.

Top Node Content

The 4*x* register entered in the top node stores the current step number. The value in this register is referenced by the DRUM instruction each time it is solved. If the middle input to the block is ON, the contents of the register in the top node are incremented to the next step in the sequence before the block is solved.

Middle Node Content

The 4*x* register entered in the middle node is the first register in a table of *step data* information. The first six registers in the *step data table* hold constant and variable data required to solve the block:

Register	Register Name	Description
Displayed	*masked output data*	Loaded by DRUM each time the block is solved; contains the contents of the *current step data* register masked with the *output mask* register
First implied	*current step data*	Loaded by DRUM each time the block is solved; contains data from the *step pointer*; causes the block logic to automatically calculate register offsets when accessing step data in the *step data table*
Second implied	*output mask*	Loaded by user before using the block, DRUM will not alter *output mask* contents during logic solve; contains a mask to be applied to the data for each sequencer step
Third implied	*machine ID number*	Identifies DRUM/ICMP blocks belonging to a specific machine configuration; value range: 0 ... 9999 (0 = block not configured); all blocks belonging to same machine configuration have the same *machine ID number*
Fourth implied	*profile ID number*	Identifies profile data currently loaded to the sequencer; value range: 0 ... 9999 (0 = block not configured); all blocks with the same *machine ID number* must have the same *profile ID number*
Fifth implied	*steps used*	Loaded by user before using the block, DRUM will not alter *steps used* contents during logic solve; contains between 1 ... 255 for 16 bit CPUs and 1 ... 999 for 24 bit CPUs, specifying the actual number of steps to be solved; the number must be ≤ *table length* in the bottom node

The remaining registers contain data for each step in the sequence.

Bottom Node Content

The integer value entered in the bottom node is the *length* —i.e., the number of application-specific registers—used in the *step data table* . The *length* can range from 1 ... 255 in a 16-bit CPU and from 1 .. 999 in a 24-bit CPU.

The total number of registers required in the *step data table* is the *length* + 6. The *length* must be ≥ the value placed in the *steps used* register in the middle node.

16.3 ICMP

The ICMP (input compare) instruction provides logic for verifying the correct operation of each step processed by a DRUM instruction. Errors detected by ICMP may be used to trigger additional error-correction logic or to shut down the system.

ICMP and DRUM are synchronized through the use of a common *step pointer* register. As the pointer increments, ICMP moves through its data table in lock step with DRUM. As ICMP moves through each new step, it compares—bit for bit—the live input data to the expected status of each point in its data table.

16.3.1 Characteristics

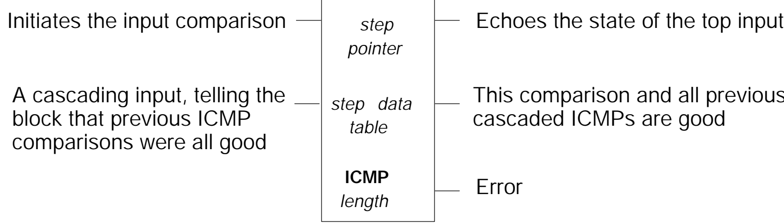
Size
Three nodes high

PLC Compatibility
Available as a loadable for all PLC types except the Micro and Quantum Automation Series PLCs

Opcode
7F hex (default)

16.3.2 Representation

Block Structure



Inputs
ICMP has two control inputs (to the top and middle nodes). When the input to the top node is ON, the ICMP operation is initiated. When the input to the middle node is ON, the instruction passes the compare status to the middle output.

Outputs

ICMP can produce three possible outputs. The output from the top node echos the state of top input. The output from the middle node goes ON to indicate a valid input comparison. The output from the bottom node goes ON if an error is detected.

Top Node Content

The 4*x* register entered in the top node stores the *step pointer* —i.e., the number of the current step in the *step data table* . This value is referenced by ICMP each time the instruction is solved. The value must be controlled externally by a DRUM instruction or by other user logic. The same register must be used in the top node of all ICMP and DRUM instructions that are solved as a single sequencer.

Middle Node Content

The 4*x* register entered in the middle node is the first register in a table of *step data* information. The first eight registers in the table hold constant and variable data required to solve the instruction:

Register	Register Name	Description
Displayed	*raw input data*	Loaded by user from a group of sequential inputs to be used by ICMP for current step
First implied	*current step data*	Loaded by ICMP each time the block is solved; contains a copy of data in the *step pointer* ; causes the block logic to automatically calculate register offsets when accessing step data in the *step data table*
Second implied	*input mask*	Loaded by user before using the block; contains a mask to be ANDed with *raw input data* for each step—masked bits will not be compared; masked data are put in the *masked input data* register
Third implied	*masked input data*	Loaded by ICMP each time the block is solved; contains the result of the ANDed *input mask* and *raw input data*
Fourth implied	*compare status*	Loaded by ICMP each time the block is solved; contains the result of an XOR of the *masked input data* and the *current step data* ; unmasked inputs that are not in the correct logical state cause the associated register bit to go to 1—non-zero bits cause a miscompare, and middle output will not go ON
Fifth implied	*machine ID number*	Identifies DRUM/ICMP blocks belonging to a specific machine configuration; value range: 0 ... 9999 (0 = block not configured); all blocks belonging to same machine configuration have the same *machine ID number*

Sixth implied	*profile ID number*	Identifies profile data currently loaded to the sequencer; value range: 0 ... 9999 (0 = block not configured); all blocks with the same *machine ID number* must have the same *profile ID number*
Seventh implied	*steps used*	Loaded by user before using the block, DRUM will not alter *steps used* contents during logic solve; contains between 1 ... 255 for 16 bit CPUs and 1 ... 999 for 24 bit CPUs, specifying the actual number of steps to be solved; the number must be $\leq$ the *table length* in the bottom node of the ICMP block

The remaining registers contain data for each step in the sequence.

Bottom Node Content

The integer value entered in the bottom node is the *length* —i.e., the number of application-specific registers—used in the *step data table* . The *length* can range from 1 ... 255 in a 16-bit CPU and from 1 .. 999 in a 24-bit CPU.

The total number of registers required in the *step data table* is the *length* + 8. The *length* must be $\geq$ the value placed in the *steps used* register in the middle node.

16.3.3 Cascaded DRUM/ICMP Blocks

A series of DRUM and/or ICMP blocks may be cascaded to simulate a mechanical drum up to 512 bits wide. Programming the same 4*x* register reference into the top node of each related block causes them to cascade and step as a grouped unit without the need of any additional application logic. All DRUM/ICMP blocks with the same register reference in the top node are automatically synchronized. The must also have the same constant value in the bottom node, and must be set to use the same value in the *steps used* register in the middle node.

16.4 SCIF

SCIF performs either a drum sequencing operation or an input comparison (ICMP) using the data defined in the *step data table* . The choice of operation is made by defining the value in the first register of the *step data table* :

- 0 = drum mode (the instruction controls outputs in the drum sequencing application)
- 1 = ICMP mode (the instruction reads inputs to ensure that limit switches, proximity switches, pushbuttons, etc. are properly positioned to allow drum outputs to be fired)

16.4.1 Characteristics

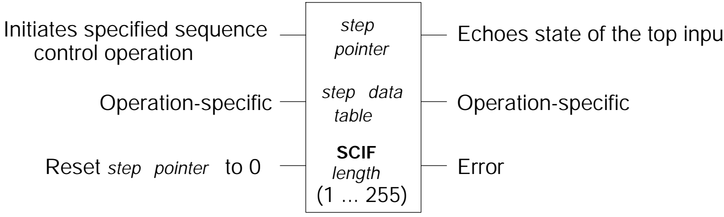
Size
Three nodes high

PLC Compatibility
- Standard in the Micro and Quantum Automation Series PLCs
- Not available in other PLC models

Opcode
3F hex

16.4.2 Representation

Block Structure



Inputs
SCIF has three control inputs. When the input to the top node is ON, the drum or ICMP operation is initiated.

When the input to the middle node is ON in drum mode, the *step pointer* increments to the next step. When this input is ON in ICMP mode, the instruction passes the compare status to the middle output..

When the input to the bottom node is ON in drum mode, the *step pointer* is reset to 0. The bottom input is not used in ICMP mode.

Outputs

SCIF can produce three possible outputs. The output from the top node echos the state of top input.

In drum mode, the output from the middle node goes ON for the last step—i.e., when the *step pointer* = *length*. In ICMP mode, this output goes ON to indicate a valid input comparison.

The output from the bottom node goes ON if an error is detected.

Top Node Content

The 4*x* register entered in the top node contains the *step pointer*—i.e., the number of the current step in the *step data table*.

Middle Node Content

The 4*x* register entered in the middle node is the first register in the *step data table*. The first seven registers in the table hold constant and variable data required to solve the instruction:

Register	Register Name	Description
Displayed	*subfunction type*	0 = drum mode; 1 = ICMP mode (entry of any other value in this register will result in all outputs OFF)
First implied	*masked output data* (in drum mode)	Loaded by SCIF each time the block is solved; the register contains the contents of the *current step data* register masked with the *output mask* register
	raw input data (in ICMP mode)	Loaded by the user from a group of sequential inputs to be used by the block in the current step
Second implied	*current step data*	Loaded by SCIF each time the block is solved; the register contains data from the current step (pointed to by the step pointer)
Third implied	*output mask* (in drum mode)	Loaded by the user before using the block, the contents will not be altered during logic solving; contains a mask to be applied to the data for each sequencer step
	input mask (in ICMP mode)	Loaded by the user before using the block, it contains a mask to be ANDed with *raw input data* for each step—masked bits will not be compared; the masked data are put in the *masked input data* register

Fourth implied	*masked input data* (in ICMP mode)	Loaded by SCIF each time the block is solved, it contains the result of the ANDed *input mask* and *raw input data*
	not used in drum mode	
Fifth implied	*compare status* (in ICMP mode)	Loaded by SCIF each time the block is solved, it contains the result of an XOR of the *masked input data* and the *current step data*; unmasked inputs that are not in the correct logical state cause the associated register bit to go to 1—non-zero bits cause a miscompare and turn ON the middle output from the SCIF block
	not used in drum mode	
Sixth implied	*start of data table**	First of K registers in the table containing the user-specified control data

* This and the rest of the registers represent application-specific step data in the process being controlled.

Bottom Node Content

The integer value entered in the bottom node is the *length*—i.e., the number of application-specific registers—used in the *step data table*. The *length* can range from 1 ... 255.

The total number of registers required in the *step data table* is the *length* + 7. The *length* must be ≥ the value placed in the *steps used* register in the middle node.

16.5 A Sequence Control Example Using the SCIF Instruction

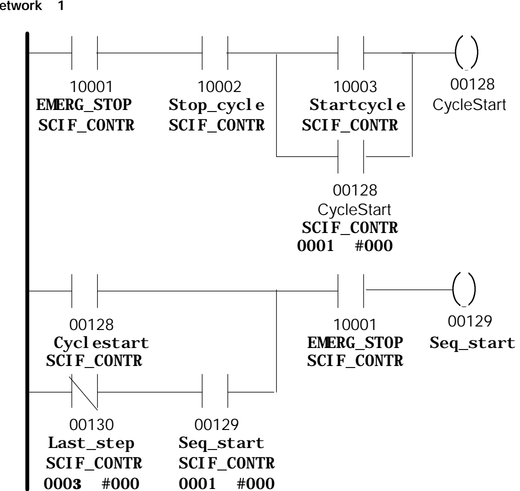
This three-network ladder logic application example shows how SCIF blocks can be used in both drum and ICMP modes. The logic in network 1 starts and stops the sequencer cycle. Once the *Start Cycle* pushbutton is pressed, the logic cycles the drum sequencer until either the *Cycle Stop* pushbutton or *E-stop* pushbutton is pressed.

If *Cycle Stop* is requested, the drum sequence continues until the last step in the step data table has been completed. If *E-stop* is pressed, the drum sequencing stops immediately on the current step.

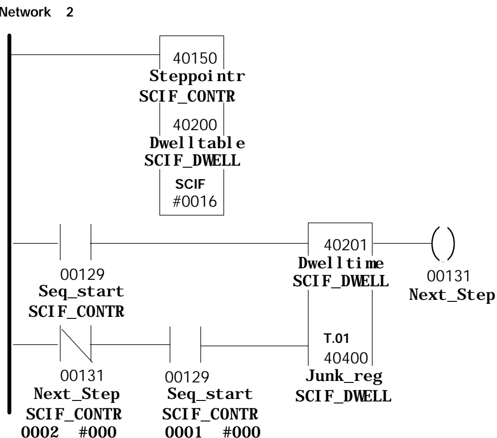
Tip: In some applications, this *E-stop* implementation may not be desirable. If an immediate stop on the current step is not desirable in your application during an emergency shutdown, you should modify the logic to suit your specific requirements.

Caution: Running this example will fire live outputs. Use this example only on a simulator, not on live machinery.

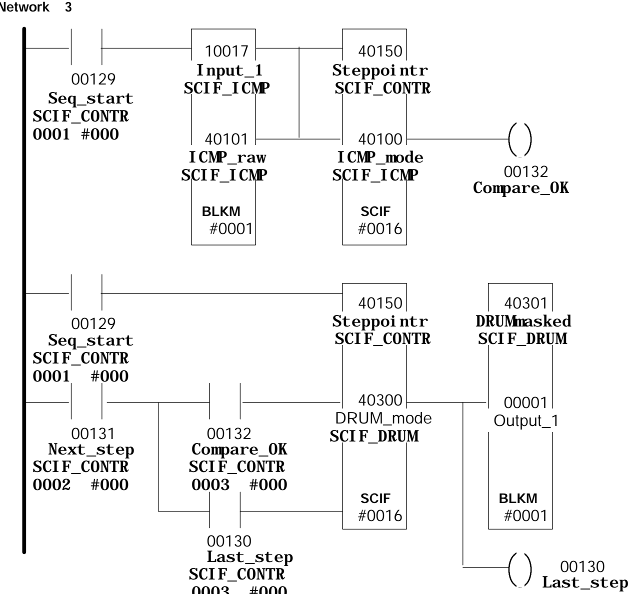
Network 1 controls the starting and stopping of the drum example. Coil 00128—Cyclestart SCIF_CONTR— indicates that the SCIF cycle has started. Coil 00129—Seq_start SCIF_CONTR—indicates that the SCIF sequence has started or restarted.



Network 2 controls the dwell time used at each step of the drum. Coil 00131— `Next_step SCIF_CONTR`—increments the SCIF pointer to the next step:



Network 3 holds the ICMP and drum functions that compare system inputs to a predetermined value and to fire the outputs of the drum. The BLKM moves the feedback inputs that the ICMP-mode SCIF next to it will monitor in its middle-node register. This SCIF then compares the status of the feedback inputs to the expected result. Coil 00132 indicates that the SCIF ICMP inputs equal the desired preset.



Chapter 17
The Checksum Instruction

Several PLCs that *do not* support Modbus Plus come with a standard checksum (CKSM) instruction. CKSM has the same opcode as the MSTR instruction and is not provided in executive firmwares for PLCs that support Modbus Plus.

17.1 CKSM

The CKSM (checksum) instruction provides you with the ability to program four types checksum calculations in ladder logic:

- Straight check
- Binary addition check
- Cyclical redundancy check (CRC-16)
- Longitudinal redundancy check (LRC)

The checksum algorithms handle both 8-bit and 16-bit data. If 8 bits are used, the high-order byte in the register must be 0.

- In a straight checksum calculation, all bytes (high and low) are summed, and the least significant eight bits are returned
- A binary checksum calculation is a 16-bit sum of all registers
- An LRC is a straight checksum that is then two's complemented
- A CRC-16 calculation is a 16 bit cyclical checksum performed on the least significant bytes of the *source* registers

17.1.1 Characteristics

Size
Three nodes high

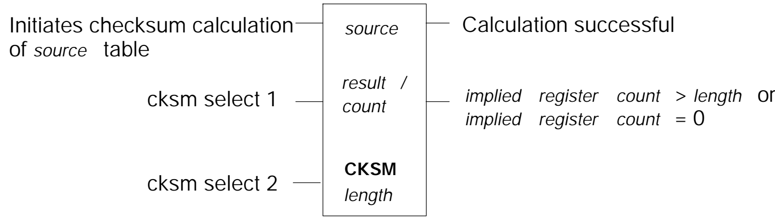
PLC Compatibility

- Standard in most PLCs that do not support Modbus Plus—Exceptions: the 984A, 984B, and 984X Chassis Mount PLCs
- Available as a loadable in PLCs where it is not standard

Opcode
BF hex

17.1.2 Representation

Block Structure



Inputs

CKSM has three control inputs. The states of the inputs indicate the type of checksum calculation to be performed:

CKSM Calculation	Top Input	Middle Input	Bottom Input
Straight Check	ON	OFF	ON
Binary Addition Check	ON	ON	ON
CRC-16	ON	ON	OFF
LRC	ON	OFF	OFF

Ouputs

CKSM can produce one of two possible outputs. The output from the top node goes ON when the checksum calculation is completed. The output from the bottom node goes ON if the an illegal implied register count is detected.

Top Node Content

The 4*x* register entered in the top node is the first holding register in the *source* table. The checksum calculation is performed on the registers in this table.

Middle Node Content

The 4*x* register entered in the middle node is the first of two contiguous 4*x* registers.

- The displayed register stores the result of the checksum calculation
- The implied register posts a value that specifies the number of registers selected from the *source* table as input to the calculation; the value in posted in the implied register must be ≤ *length* of source table

Bottom Node Content

The integer value entered in the bottom node specifies the *length* —i.e., the number of 4*x* registers—in the *source* table. The *length* is in the range 1 ... 255.

Chapter 18
The Modbus Plus Master Instruction

- MSTR Overview
- MSTR Function Error Codes
- *Read* and *Write* MSTR Operations
- *Get Local Statistics* MSTR Operation
- *Clear Local Statistics* MSTR Operation
- *Write Global Data* MSTR Operation
- *Read Global Data* MSTR Operation
- *Get Remote Statistics* MSTR Operation
- *Clear Remote Statistics* MSTR Operation
- *Reset Option Module* MSTR Operation
- *Read CTE* (Config Extension) MSTR Operation
- *Write CTE* (Config Extension) MSTR Operation
- Modbus Plus Network Statistics
- TCP/IP EtherNet Statistics

18.1 MSTR Overview

PLCs that support networking communications capabilities over Modbus Plus and Ethernet have a special MSTR (master) instruction with which nodes on the network can initiate message transactions.

The MSTR instruction allows you to initiate one of 12 possible network communications operations over the network. Each operation is designated by a code. Certain MSTR operations are supported on some networks and not on others:

MSTR Operation	Code	Modbus Plus	TCP/IP Ether-Net	SY/MAX Ethernet
Write data	1	x	x	x
Read data	2	x	x	x
Get local statistics	3	x	x	not supported
Clear local statistics	4	x	x	not supported
Write global database	5	x	not supported	not supported
Read global database	6	x	not supported	not supported
Get remote statistics	7	x	x	not supported
Clear remote statistics	8	x	x	not supported
Peer Cop health	9	x	not supported	not supported
Reset Option Module	10	not supported	x	x
Read CTE (config extension)	11	not supported	x	x
Write CTE (config extension)	12	not supported	x	x

Up to four MSTR instructions can be simultaneously active in a ladder logic program. More than four MSTRs may be programmed to be enabled by the logic flow—as one active MSTR block releases the resources it has been using and becomes deactivated, the next MSTR operation encountered in logic can be activated.

18.1.1 Characteristics

Size

Three nodes high

PLC Compatibility

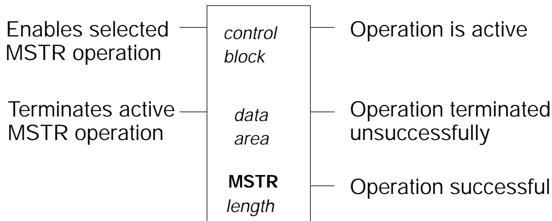
- Standard in PLCs that have built-in Modbus Plus capabilities (Modbus Plus functionality only)
- Standard in all Quantum PLCs with Modbus Plus functionality and/or TCP/IP and SY/MAX EtherNet option modules

- ☐ Available as a loadable in chassis mount PLCs (Modbus Plus functionality only)

Opcode
BF hex

18.1.2 Representation

Block Structure



Inputs
MSTR has two control inputs. The input to the top node enables the instruction when it is ON. The input to the middle node terminates the active operation when it is ON

Outputs
MSTR can produce three possible outputs. The output from the top node echoes the state of the top input—i.e., it goes ON while the instruction is active. The output from the middle node echoes the state of the middle input—i.e., it goes ON if the the MSTR operation is terminated prior to completion. The output from the bottom node goes ON when an MSTR operation has been completed successfully.

Top Node Content
The 4*x* register entered in the top node is the first of several (network-dependant) holding registers that comprise the network *control block* . The *control block* structure differs according to the network in use:

Control Block **for Modbus Plus**

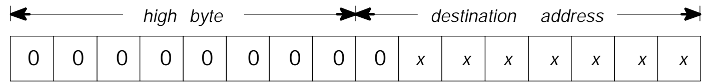
Register	Content
Displayed	Identifies one of nine MSTR operations legal for Modbus Plus (1 ... 9)
First implied	Displays error status
Second implied	Displays length (number of registers transferred)
Third implied	Displays MSTR operation-dependent information

Fourth implied

The Routing 1 register, used to designate the address of the destination node for a network transaction. The register display is implemented logically in the 984 PLCs and physically for the Quantum PLCs:

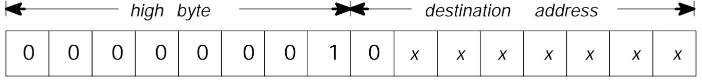
984 PLCs

For an S985-002 card in a 984 chassis mount PLC, a value of 0 in the high byte indicates that the MSTR instruction is destined for the S985 card set for PLC port #2; for a PLC with built-in Modbus Plus, a value of 0 in the high byte indicates that the MSTR is destined for the on-board Modbus Plus port:



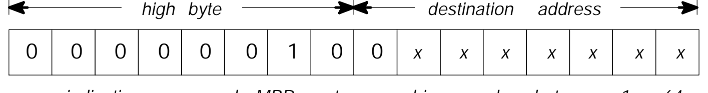
binary value between 1 ... 64

For two S985-002 cards in a 984 chassis mount PLC, a value of 1 in the high byte indicates that the MSTR instruction is destined for the second S985 card's assigned buffer space; for an S985-00 configuration in a PLC with built-in Modbus Plus, a value of 1 in the high byte indicates that the MSTR is destined for the S985 card set for comm port #2:



indicating a second MBP port *binary value between 1 ... 64*

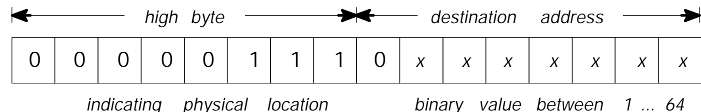
For two S985-000 cards in a 984 PLC with built-in Modbus Plus, a value of 2 in the high byte indicates that the MSTR instruction is destined for the second S985 card's assigned buffer space:



indicating a second MBP port *binary value between 1 ... 64*

Quantum Automation Series PLCs

To target a Modbus Plus Network Option module (NOM) in a Quantum PLC backplane as the destination of an MSTR instruction, the value in the high byte represents the physical slot location of the NOM—e.g., if the NOM resides in slot 7 in the backplane, the high byte of routing register 1 would look like this:



indicating physical location *binary value between 1 ... 64*

Note: If you have created a logic program using an MSTR instruction for a 984 PLC and want to port it to a Quantum Automation Series PLC without having to edit the routing 1 register value, make sure that NOM #1 is installed in slot 1 of the Quantum backplane (and if a NOM #2 is used, that it is installed in slot 2 of the backplane). If you try to run the ported application with the NOMs in other slots without modifying the register, an F001 status error will appear, indicating the wrong destination node.

Fifth implied	The Routing 2 register
Sixth implied	The Routing 3 register
Seventh implied	The Routing 4 register
Eighth implied	The Routing 5 register
Ninth implied	not applicable
Tenth implied	not applicable
Eleventh implied	not applicable

Control Block **for TCP/IP EtherNet**

Register	Content
Displayed	Identifies one of nine MSTR operations legal for TCP/IP (1 ... 4, 7, 8, 10 ... 12)
First implied	Displays error status
Second implied	Displays length (number of registers transferred)
Third implied	Displays MSTR operation-dependent information
Fourth implied	High byte: MBP-to-EtherNet Transporter (MET) Map index
	Low byte: Quantum backplane slot address of the NOE module
Fifth implied	Byte 4 of the 32-bit destination IP Address
Sixth implied	Byte 3 of the 32-bit destination IP Address
Seventh implied	Byte 2 of the 32-bit destination IP Address
Eighth implied	Byte 1 of the 32-bit destination IP Address

Control Block **for SY/MAX EtherNet**

Register	Content
Displayed	Identifies one of five MSTR operations legal for SY/MAX (1, 2, 10 ... 12)
First implied	Displays error status
Second implied	Displays Read/Write length (number of registers transferred)
Third implied	Displays Read/Write base address
Fourth implied	High byte: drop number
	Low byte: Quantum backplane slot address of the NOE module
Fifth implied	Destination drop number (or set to FF hex)
Sixth implied	Terminator (set to FF hex)

Middle Node Content

The 4*x* register entered in the middle node is the first in a group of contiguous holding registers that comprise the *data area*. For operations that provide the communication processor with data—such as a Write operation—the *data area* is the source of the data. For operations that acquire data from the communication processor—such as a Read operation—the *data area* is the destination for the data.

In the case of the EtherNet Read and Write CTE operations (see sections 18.12 and 18.13), the middle node stores the contents of the EtherNet configuration extension table in a series of registers.

Bottom Node Content

The integer value entered in the bottom node specifies the *length*—i.e., the maximum number of registers—in the *data area*. The *length* must be in the range 1 ... 100.

18.2 MSTR Function Error Codes

If an error occurs during an MSTR operation, a hexadecimal error code will be displayed in the first implied register in the *control block* (the top node). Function error codes are network-specific.

18.2.1 Modbus Plus and SY/MAX EtherNet Error Codes

The form of the function error code for Modbus Plus and SY/MAX EtherNet transactions is *Mmss* , where

- *M* represents the major code
- *m* represents the minor code
- *ss* represents a subcode

Hex Error Code	Meaning
1001	User has aborted the MSTR element
2001	An unsupported operation type has been specified in the *control block*
2002	One or more *control block* parameter has been changed while the MSTR element is active (applies only to operations that take multiple scans to complete) *Control block* parameters may be changed only when the MSTR element is not active
2003	Invalid value in the length field of the *control block*
2004	Invalid value in the offset field of the *control block*
2005	Invalid values in the length and offset fields of the *control block*
2006	Invalid slave device data area
2007	Invalid slave device network area
2008	Invalid slave device network routing
2009	Route equal to your own address
200A	Attempting to obtain more global data words than available
30ss*	Modbus slave exception response
4001	Inconsistent Modbus slave response
5001	Inconsistent network response
6mss**	Routing failure
* The ss subfield in error code 30ss is:	
ss Hex Value	**Meaning**
01	Slave device does not support the requested operation
02	Nonexistent slave device registers requested
03	Invalid data value requested
04	

05	Slave has accepted long-duration program command
06	Function can't be performed now—a long-duration command in effect
07	Slave rejected long-duration program command
08 ... 255	

** The m subfield in error code 6mss is an index into the routing information indicating where an error has been detected—a value of 0 indicates the local node, a 2 the second device on the route, etc. The ss subfield in error code 6mss is:

ss Hex Value	Meaning
01	No response received
02	Program access denied
03	Node off-line and unable to communicate
04	Exception response received
05	Router node data paths busy
06	Slave device down
07	Bad destination address
08	Invalid node type in routing path
10	Slave has rejected the command
20	Initiated transaction forgotten by slave device
40	Unexpected master output path received
80	Unexpected response received
F001	Wrong destination node specified for the MSTR operation

18.2.2 SY/MAX specific Error Codes

Three additional types of errors may be reported in the MSTR instruction when SY/MAX EtherNet is being used. The error codes have the following designations:

- ☐ 71*xx* errors: Errors detected by the remote SY/MAX device
- ☐ 72*xx* errors: Errors detected by the server
- ☐ 73*xx* errors: Errors detected by the Quantum translator

Hex Error Code	Meaning
7101	Illegal opcode detected by the remote SY/MAX device
7103	Illegal address detected by the remote SY/MAX device
7109	Attempt to write a read only register detected by the remote SY/MAX device
710F	Receiver overflow detected by the remote SY/MAX device
7110	Invalid length detected by the remote SY/MAX device
7111	Remote device inactive, not communicating (occurs after retries and time out have been exhausted) detected by the remote SY/MAX device

7113	Invalid parameter on a read operation detected by the remote SY/MAX device
711D	Invalid route detected by the remote SY/MAX device
7149	Invalid parameter on a write operation detected by the remote SY/MAX device
714B	Illegal drop number detected by the remote SY/MAX device
7201	Illegal opcode detected by the SY/MAX server
7203	Illegal address detected by the SY/MAX server
7209	Attempt to write to a read only register detected by the SY/MAX server
720F	Receiver overflow detected by the SY/MAX server
7210	Invalid length detected by the SY/MAX server
7211	Remote device inactive, not communicating (occurs after retries and time out have been exhausted) detected by the SY/MAX server
7213	Invalid parameter on a read operation detected by the SY/MAX server
721D	Invalid route detected by the SY/MAX server
7249	Invalid parameter on a write operation detected by the SY/MAX server
724B	Illegal drop number detected by the SY/MAX server
7301	Illegal opcode in an MSTR block request by the Quantum translator
7303	Read/Write QSE module status (200 route address out of range)
7309	Attempt to write to a read only register when performing a status write (200 route)
731D	Invalid rout detected by Quantum translator. Valid routes are: dest_drop, 0xFF 200, dest_drop, 0xFF 100+drop, dest_drop, 0xFF All other routing values generate an error
734B	One of the following errors has occurred:, or No CTE (configuration extension) table was configured No CTE table entry was created for the QSE Module slot number No valid drop was specified The QSE Module was not reset after the CTE was created* When using an MSTR instruction, no valid slot or drop was specified
	* After writing and configuring the CTE and downloading it to the QSE Module, you must reset the QSE Module to make the changes take effect.

18.2.3 TCP/IP EtherNet Error Codes

An error in an MSTR routine over TCP/IP EtherNet may produce one of the following errors in the MSTR *control block*:

Hex Error Code	Meaning
1001	User has aborted the MSTR element
2001	An unsupported operation type has been specified in the *control block*
2002	One or more *control block* parameter has been changed while the MSTR element is active (applies only to operations that take multiple scans to complete) *Control block* parameters may be changed only when the MSTR element is not active
2003	Invalid value in the length field of the *control block*
2004	Invalid value in the offset field of the *control block*
2005	Invalid values in the length and offset fields of the *control block*
2006	Invalid slave device data area
3000	Generic Modbus fail code
30ss*	Modbus slave exception response
4001	Inconsistent Modbus slave response
* The ss subfield in error code 30ss is:	
ss Hex Value	**Meaning**
01	Slave device does not support the requested operation
02	Nonexistent slave device registers requested
03	Invalid data value requested
04	
05	Slave has accepted long-duration program command
06	Function can't be performed now—a long-duration command in effect
07	Slave rejected long-duration program command

An error on the TCP/IP EtherNet network itself may produce one of the following errors in the MSTR *control block*:

Hex Error Code	Meaning
5004	Interrupted system call
5005	I/O error
5006	No such address
5009	The socket descriptor is invalid
500C	Not enough memory
500D	Permission denied
5011	Entry exists
5016	An argument is invalid
5017	An internal table has run out of space
5020	The connection is broken
5023	This operation would block and the socket is nonblocking
5024	The socket is nonblocking and the connection cannot be completed

5025	The socket is nonblocking and a previous connection attempt has not yet completed
5026	Socket operation on a nonsocket
5027	The destination address is invalid
5028	Message too long
5029	Protocol wrong type for socket
502A	Protocol not available
502B	Protocol not supported
502C	Socket type not supported
502D	Operation not supported on socket
502E	Protocol family not supported
502F	Address family not supported
5030	Address is already in use
5031	Address not available
5032	Network is down
5033	Network is unreachable
5034	Network dropped connection on reset
5035	The connection has been aborted by the peer
5036	The connection has been reset by the peer
5037	An internal buffer is required, but cannot be allocated
5038	The socket is already connected
5039	The socket is not connected
503A	Can't send after socket shutdown
503B	Too many references; can't splice
503C	Connection timed out
503D	The attempt to connect was refused
5040	Host is down
5041	The destination host could not be reached from this node
5042	Directory not empty
5046	NI_INIT returned 1
5047	The MTU is invalid
5048	The hardware length is invalid
5049	The route specified cannot be found
504A	Collision in select call; these conditions have already been selected by another task
504B	The task id is invalid

18.2.4 CTE Error Codes for SY/MAX and TCP/IP EtherNet

The following error codes are returned if there is a problem with the EtherNet configuration extension table (CTE) in your program configuration.

Hex Error Code	Meaning
7001	The is no EtherNet configuration extension
7002	The CTE is not available for access
7003	The offset is invalid
7004	The offset + length is invalid
7005	Bad data field in the CTE

18.3 *Read* and *Write* MSTR Operations

An MSTR Write operation transfers data from a master source device to a specified slave destination device on the network. An MSTR Read operation transfers data from a specified slave source device to a master destination device on the network. Read and Write use one data master transaction path and may be completed over multiple scans.

18.3.1 Network Implementation

The MSTR Read and Write operations (type 2 or 1, respectively, in the displayed register of the top node) can be implemented on the Modbus Plus, TCP/IP EtherNet, and SY/MAX EtherNet networks.

Note: You need to understand the routing procedures used by the network you are using when you program an MSTR instruction. A full discussion of Modbus Plus routing path structures is given in *Modbus Plus Network Planning and Installation Guide*. If TCP/IP or SY/MAX EtherNet routing is being implemented, it must be accomplished via standard third-part Ethernet IP router products.

18.3.2 Control Block Utilization

In a Read or Write operation, the registers in the MSTR *control block* (the top node) contain the information that differs depending on the type of network you are using:

***Control Block* for Modbus Plus**

Register	Function	Content
Displayed	Operation type	1 = Write; 2 = Read
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Length	Write = number of registers to be sent to slave Read = number of registers to be read from slave
Third implied	Slave device data area	Specifies starting 4*x* register in the slave to be read from or written to (1 = 40001, 49 = 40049)
Fourth ... Eighth implied	Routing 1 ... 5	Designates the first ... fifth routing path addresses, respectively; the last nonzero byte in the routing path is the destination device

Note: If you attempt to program the MSTR to Read or Write its own station address on a Modbus Plus network, an error will be generated in the first implied register of the control block. It is possible to attempt a Read/Write operation to a nonexistent register in the slave device. The slave will detect this condition and report it—this may take several scans.

Control Block **for TCP/IP EtherNet**

<table>
<tr><th>Register</th><th>Function</th><th colspan="2">Content</th></tr>
<tr><td>Displayed</td><td>Operation type</td><td colspan="2">1 = Write; 2 = Read</td></tr>
<tr><td rowspan="4">First implied</td><td rowspan="4">Error status</td><td colspan="2">Displays a hex value indicating an MSTR error:</td></tr>
<tr><td>Exception response, where response size is correct</td><td>Exception code + 3000</td></tr>
<tr><td>Exception response, where response size is incorrect</td><td rowspan="2">4001</td></tr>
<tr><td>Read/Write</td></tr>
<tr><td>Second implied</td><td>Length</td><td colspan="2">Write = number of registers to be sent to slave
Read = number of registers to be read from slave</td></tr>
<tr><td>Third implied</td><td>Slave device data area</td><td colspan="2">Specifies starting 4x register in the slave to be read from or written to (1 = 40001, 49 = 40049)</td></tr>
<tr><td>Fourth implied</td><td>Low byte</td><td colspan="2">Quantum backplane slot address of the network adapter module</td></tr>
<tr><td>Fifth ... eighth implied</td><td>Destination</td><td colspan="2">Each register contains one byte of the 32-bit IP address</td></tr>
</table>

Control Block **for SY/MAX EtherNet**

Register	Function	Content
Displayed	Operation type	1 = Write; 2 = Read
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Length	Write = number of registers to be sent to slave Read = number of registers to be read from slave
Third implied	Slave device data area	Specifies starting 4*x* register in the slave to be read from or written to (1 = 40001, 49 = 40049)
Fourth implied	Slot ID	Quantum backplane slot address of the network adapter module
		Destination drop number
Fifth ... eighth implied	Terminator	FF hex

18.4 *Get Local Statistics* MSTR Operation

The Get local statistics operation obtains information related to the local node—where the MSTR has been programmed. This operation takes one scan to complete and does not require a data master transaction path.

18.4.1 Network Implementation

The Get Local Statistics operation (type 3 in the displayed register of the top node) can be implemented for Modbus Plus and TCP/IP EtherNet networks. It is not used for SY/MAX EtherNet.

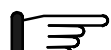
- ☐ See page 367 for the listing of available Modbus Plus network statistics
- ☐ See page 18.15 for the listing of TCP/IP EtherNet network statistics

18.4.2 Control Block Utilization

In a Get local statistics operation, the registers in the MSTR *control block* (the top node) contain the information that differs depending on the type of network you are using:

Control Block **for Modbus Plus**

Register	Function	Content
Displayed	Operation type	3
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Length	Starting from *offset*, the number of words of statistics from the local processor's statistics table; the *length* must be $> 0 \leq$ *data area*
Third implied	Offset	An offset value relative to the first available word in the local processor's statistics table—if the offset is specified as 1, the function obtains statistics starting with the second word in the table
Fourth implied	Routing 1	If this is the second of two local nodes, set the high byte to a value of 1



Note: If you are using the MSTR instruction for Modbus Plus networking and your PLC does not support Modbus Plus option modules (S985s or NOMs), the fourth implied register is not used.

Control Block **for TCP/IP EtherNet**

Register	Function	Content
Displayed	Operation type	3
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Length	Starting from *offset* , the number of words of statistics from the local processor's statistics table; the *length* must be > 0 ≤ *data area*
Third implied	Offset	An offset value relative to the first available word in the local processor's statistics table —if the offset is specified as 1, the function obtains statistics starting with the second word in the table
Fourth implied	Slot ID	Quantum backplane slot address of the network adapter module
Fifth ... Eighth implied	Not applicable	

18.5 *Clear Local Statistics* MSTR Operation

The Clear local statistics operation clears statistics relative to the local node—where the MSTR has been programmed. This operation takes one scan to complete and does not require a data master transaction path.

18.5.1 Network Implementation

The Clear Local Statistics operation (type 4 in the displayed register of the top node) can be implemented for Modbus Plus and TCP/IP EtherNet networks. It is not used for SY/MAX EtherNet.

- See page 367 for the listing of available Modbus Plus network statistics
- See page 18.15 for the listing of TCP/IP network statistics

18.5.2 Control Block Utilization

In a Clear local statistics operation, the registers in the MSTR *control block* (the top node) differ according to the type of network in use:

Control Block **for Modbus Plus**

Register	Function	Content
Displayed	Operation type	4
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Not applicable	
Third implied		
Fourth implied	Routing 1	If this is the second of two local nodes, set the high byte to a value of 1

Note: If you are using the MSTR instruction for Modbus Plus networking and your PLC does not support Modbus Plus option modules (S985s or NOMs), the fourth implied register is not used.

Control Block **for TCP/IP EtherNet**

Register	Function	Content
Displayed	Operation type	4
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Not applicable	
Third implied		
Fourth implied	Slot ID	Quantum backplane slot address of the network adapter module
Fifth ... Eighth implied	Not applicable	

18.6 *Write Global Data* MSTR Operation

The Write global data operation transfers data to the communications processor in the current node so that it can be sent over the network when the node gets the token. All nodes on the local network link can receive this data. This operation takes one scan to complete and does not require a data master transaction path.

18.6.1 Network Implementation

The Write global data operation (type 5 in the displayed register of the top node) can be implemented only for Modbus Plus networks.

18.6.2 Control Block Utilization

The registers in the MSTR *control block* (the top node) are used in a Write global data operation:

Register	Function	Content
Displayed	Operation type	5
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Length	Specifies the number of registers from the data area to be sent to the comm processor; the value of the *length* must be $\leq$ 32 and must not exceed the size of the *data area*
Third implied	Not applicable	
Fourth implied	Routing 1	If this is the second of two local nodes, set the high byte to a value of 1

Note: If your PLC does not support Modbus Plus option modules (S985s or NOMs), the fourth implied register is not used.

18.7 *Read Global Data* MSTR Operation

The Read global data operation gets data from the communications processor in any node on the local network link that is providing global data. This operation may require multiple scans to complete if global data is not currently available from the requested node. If global data is available, the operation completes in a single scan. No master transaction path is required.

18.7.1 Network Implementation

The Read global data operation (type 6 in the displayed register of the top node) can be implemented only for Modbus Plus networks.

18.7.2 Control Block Utilization

The registers in the MSTR *control block* (the top node) are used in a Read global data operation:

Register	Function	Content
Displayed	Operation type	6
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Length	Specifies the number of words of global data to be requested from the comm processor designated by the routing 1 parameter; the value of the *length* must be > 0 $\leq$ 32 and must not exceed the size of the *data area*
Third implied	Available words	Contains the number of words available from the requested node; the value is automatically updated by internal software
Fourth implied	Routing 1	The low byte specifies the address of the node whose global data are to be returned (a value between 1 ... 64); if this is the second of two local nodes, set the high byte to a value of 1

☞ **Note:** If your PLC does not support Modbus Plus option modules (S985s or NOMs), the high byte of the fourth implied register is not used and the high-byte bits must all be set to 0.

18.8 *Get Remote Statistics* MSTR Operation

The Get remote statistics operation obtains information relative to remote nodes on the network. This operation may require multiple scans to complete and does not require a master data transaction path.

18.8.1 Network Implementation

The Get Remote Statistics operation (type 7 in the displayed register of the top node) can be implemented for Modbus Plus and TCP/IP EtherNet networks. It is not used for SY/MAX EtherNet.

Note: You need to understand the routing procedures used by the network you are using when you program an MSTR instruction. A full discussion of Modbus Plus routing path structures is given in *Modbus Plus Network Planning and Installation Guide* . If TCP/IP routing is being implemented, it must be accomplished via standard third-part Ethernet IP router products.

18.8.2 Control Block Utilization

In a Get remote statistics operation, the registers in the MSTR *control block* (the top node) contain information that differs according to the network in use:

Control Block **for Modbus Plus**

Register	Function	Content
Displayed	Operation type	7
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Length	Starting from an offset, the number of words of statistics to be obtained from a remote node; the *length* must be > 0 ≤ total number of statistics available (54) and must not exceed the size of the *data area*
Third implied	Offset	Specifies an offset value relative to the first available word in the statistics table; the value must not exceed the number of statistic words available
Fourth ... eighth implied	Routing 1 ... 5	Designates the first ... fifth routing path addresses, respectively; the last nonzero byte in the routing path is the destination device

The remote comm processor always returns its complete statistics table when a request is made, even if the request is for less than the full

table. The MSTR instruction then copies only the amount of words you have requested to the designated 4*x* registers.

Control Block **for TCP/IP EtherNet**

Register	Function	Content
Displayed	Operation type	7
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Length	Starting from *offset* , the number of words of statistics from the local processor's statistics table; the *length* must be > 0 ≤ *data area*
Third implied	Offset	An offset value relative to the first available word in the local processor's statistics table—if the offset is specified as 1, the function obtains statistics starting with the second word in the table
Fourth implied	Low byte	Quantum backplane slot address of the network adapter module
Fifth ... Eighth implied	Destination	Each register contains one byte of the 32-bit IP address

18.9 *Clear Remote Statistics* MSTR Operation

The Clear remote statistics operation clears statistics related to a remote network node from the *data area* in the local node. This operation may require multiple scans to complete and uses a single data master transaction path.

18.9.1 Network Implementation

The Clear remote statistics operation (type 8 in the displayed register of the top node) can be implemented for Modbus Plus and TCP/IP EtherNet networks. See page 367 for the listing of available network statistics. It is not used for SY/MAX EtherNet.

18.9.2 Control Block Utilization

In a Clear remote statistics operation, the registers in the MSTR *control block* (the top node) contain information that differs according to the network in use:

***Control Block* for Modbus Plus**

Register	Function	Content
Displayed	Operation type	8
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Not applicable	
Third implied		
Fourth ... eighth implied	Routing 1 ... 5	Designates the first ... fifth routing path addresses, respectively; the last nonzero byte in the routing path is the destination device

Note: You need to understand Modbus Plus routing path procedures before programming an MSTR block. A full discussion of routing path structures is given in *Modbus Plus Network Planning and Installation Guide*.

***Control Block* for TCP/IP EtherNet**

Register	Function	Content
Displayed	Operation type	8
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Not applicable	
Third implied		
Fifth ... Eighth implied	Destination	Each register contains one byte of the 32-bit IP address

18.10 ***Peer Cop Health*** MSTR Operation

The peer cop health operation reads selected data from the peer cop communications health table and loads that data to specified 4*x* registers in state RAM. The peer cop communications health table is 12 words long, and the words are indexed via this MSTR operation as words 0 ... 11.

18.10.1 Network Implementation

The Clear remote statistics operation (type 8 in the displayed register of the top node) can be implemented only for Modbus Plus networks.

18.10.2 Control Block Utilization

The registers in the MSTR *control block* (the top node) contain the following information in a Peer cop health operation:

Register	Function	Content
Displayed	Operation type	9
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Data Size	Number of words requested from peer cop table (range 1 ... 12)
Third implied	Index	First word from the table to be read (range 0 ... 11, where 0 = the first word in the peer cop table and 11 = the last word in the table)
Fourth implied	Routing 1	If this is the second of two local nodes, set the high byte to a value of 1

☞ **Note:** If your PLC does not support Modbus Plus option modules (S985s or NOMs), the fourth implied register is not used.

18.10.3 Peer Cop Communications Health Status Information

The peer cop communications health table comprises 12 contiguous registers that can be indexed in an MSTR operation as words 0 ... 11. Each bit in each of the table words is used to represent an aspect of communications health relative to a specific node on the Modbus Plus network.

The bits in words 0 ... 3 represent the health of the global input communication expected from nodes 1 ... 64. The bits in words 4 ... 7 represent the health of the output from a specific node. The bits in words 8 ... 11 represent the health of the input to a specific node:

Type of Status	Word Index	Bit-to-Network Node Relationship															
Global Input	0	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
	1	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
	2	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
	3	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
Specific Output	4	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
	5	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
	6	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
	7	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
Specific Input	8	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
	9	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
	10	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
	11	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49

The state of a peer cop health bit reflects the current communication status of its associated node. A health bit is set when its associated node accepts inputs for its peer copped input data group or hears that another node has accepted specific output data from the its peer copped output data group. A health bit is cleared when no communication has occurred for its associated data group within the configured peer cop health time-out period.

All health bits are cleared when the Put Peer Cop interface command is executed at PLC start-up time. Table values are not valid until at least one full token rotation cycle has been completed after execution of the Put Peer Cop interface command. The health bit for a given node is always zero when its associated peer cop entry is null.

18.1 1 *Reset Option Module* MSTR Operation

The Reset option module operation causes a Quantum NOE option module to enter a reset cycle to reset its operational environment.

18.1 1.1 Network Implementation

The Reset option module operation (type 10 in the displayed register of the top node) can be implemented for TCP/IP and SY/MAX Ethernet networks, accessed via the appropriate network adapter. Modbus Plus networks do not use this operation.

18.1 1.2 Control Block Utilization

In a Reset option module operation, the registers in the MSTR *control block* (the top node) differ according to the network in use:

***Control Block* for TCP/IP EtherNet**

Register	Function	Content
Displayed	Operation type	10
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Not applicable	
Third implied		
Fourth implied	Slot ID	Number displayed in the low byte, in the range 1 ... 16 indicating the slot in the local backplane where the option module resides
Fifth ... Eighth implied	Not applicable	

***Control Block* for SY/MAX EtherNet**

Register	Function	Content
Displayed	Operation type	10
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Not applicable	
Third implied		
Fourth implied	Slot ID	Quantum backplane slot address of the network adapter module
Fifth ... eighth implied	Not applicable	

18.12 *Read CTE* (Config Extension Table) MSTR Operation

The Read CTE operation reads a given number of bytes from the Ethernet configuration extension table to the indicated buffer in PLC memory. The bytes to be read begin at a byte offset from the beginning of the CTE. The content of the EtherNet CTE table is displayed in the middle node of the MSTR block.

18.12.1 Network Implementation

The Read CTE operation (type 11 in the displayed register of the top node) can be implemented for TCP/IP and SY/MAX Ethernet networks, accessed via the appropriate network adapter. Modbus Plus networks do not use this operation.

18.12.2 Control Block Utilization

In a Read CTE operation, the registers in the MSTR *control block* (the top node) differ according to the network in use:

***Control Block* for TCP/IP EtherNet**

Register	Function	Content
Displayed	Operation type	11
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Not applicable	
Third implied		
Fourth implied	Map index	Either a value displayed in the high byte of the register or not used
	Slot ID	Number displayed in the low byte, in the range 1 ... 16 indicating the slot in the local backplane where the option module resides
Fifth ... Eighth implied	Not applicable	

Control Block for SY/MAX EtherNet

Register	Function	Content
Displayed	Operation type	11
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Data Size	Number of words transferred
Third implied	Base Address	Byte offset in PLC register structure indicating where the CTE bytes will be written
Fourth implied	High byte	Quantum backplane slot address of the NOE module
	Low byte	Terminator (FF hex)
Fifth ... eighth implied	Not applicable	

18.12.3 CTE Display Implementation

The values in the EtherNet configuration extension table (CTE) are displayed in a series of registers in the middle node of the MSTR instruction when a Read CTE operation is implemented. The middle node contains the first of 11 contiguous 4*x* registers. The registers display the following CTE data:

Parameter	Register	Content
Frame type	Displayed	1 = 802.3
		2 = EtherNet
IP address	First implied	First byte of the IP address
	Second implied	Second byte of the IP address
	Third implied	Third byte of the IP address
	Fourth implied	Fourth byte of the IP address
Subnetwork mask	Fifth implied	Hi word
	Sixth implied	Low word
Gateway	Seventh implied	First byte of the gateway
	Eighth implied	Second byte of the gateway
	Ninth implied	Third byte of the gateway
	Tenth implied	Fourth byte of the gateway

18.13 *Write CTE* (Config Extension Table) MSTR Operation

The Write CTE operation reads an indicated number of bytes from PLC memory, starting at a specified byte address, to an indicated Ethernet configuration extension table at a specified offset. The content of the EtherNet CTE table is displayed in the middle node of the MSTR block.

18.13.1 Network Implementation

The Write CTE operation (type 12 in the displayed register of the top node) can be implemented for TCP/IP and SY/MAX Ethernet networks, via the appropriate network adapter. Modbus Plus networks do not use this operation.

18.13.2 Control Block Utilization

In a Read CTE operation, the registers in the MSTR *control block* (the top node) differ according to the network in use:

***Control Block* for TCP/IP EtherNet**

Register	Function	Content
Displayed	Operation type	12
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Not applicable	
Third implied		
Fourth implied	Map index	Either a value displayed in the high byte of the register or not used
	Slot ID	Number displayed in the low byte, in the range 1 ... 16 indicating the slot in the local backplane where the option module resides
Fifth ... Eighth implied	Not applicable	

Control Block for SY/MAX EtherNet

Register	Function	Content
Displayed	Operation type	12
First implied	Error status	Displays a hex value indicating an MSTR error, when relevant
Second implied	Data Size	Number of words transferred
Third implied	Base Address	Byte offset in PLC register structure indicating where the CTE bytes will be written
Fourth implied	High byte	Quantum backplane slot address of the NOE module
	Low byte	Destination drop number
Fifth implied	Terminator	FF hex
Sixth ... eighth implied	Not applicable	

18.13.3 CTE Display Implementation

The values in the EtherNet configuration extension table (CTE) are displayed in a series of registers in the middle node of the MSTR instruction when a Write CTE operation is implemented. The middle node contains the first of 11 contiguous 4*x* registers. The registers display the following CTE data:

Parameter	Register	Content
Frame type	Displayed	1 = 802.3
		2 = EtherNet
IP address	First implied	First byte of the IP address
	Second implied	Second byte of the IP address
	Third implied	Third byte of the IP address
	Fourth implied	Fourth byte of the IP address
Subnetwork mask	Fifth implied	Hi word
	Sixth implied	Low word
Gateway	Seventh implied	First byte of the gateway
	Eighth implied	Second byte of the gateway
	Ninth implied	Third byte of the gateway
	Tenth implied	Fourth byte of the gateway

18.14 Modbus Plus Network Statistics

The following table shows the statistics available on the Modbus Plus network. You may acquire this information by using the appropriate MSTR operation or by using Modbus function code 8.

Note: When you issue the *Clear local* or *Clear remote statistics* operations, only words 13 ... 22 are cleared.

Word	Bits	Meaning
00		Node type ID
	0	Unknown node type
	1	PLC node
	2	Modbus bridge node
	3	Host computer node
	4	Bridge Plus node
	5	Peer I/O node
01	0 ... 11	Software version number in hex (to read, strip bits 12 15 from word)
	12 ... 14	Reserved
	15	Defines Word 15 error counters (see Word 15) Most significant bit defines use of error counters in Word 15. Least significant half of upper byte, plus lower byte, contain software version. 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Software version number (in hex) Word 15 error counters (*see* Word 15)
02		Network address for this station

03		MAC state variable:
	0	Power up state
	1	Monitor offline state
	2	Duplicate offline state
	3	Idle state
	4	Use token state
	5	Work response state
	6	Pass token state
	7	Solicit response state
	8	Check pass state
	9	Claim token state
	10	Claim response state
04		Peer status (LED code); provides status of this unit relative to the network:
	0	Monitor link operation
	32	Normal link operation
	64	Never getting token
	96	Sole station
	128	Duplicate station
05		Token pass counter; increments each time this station gets the token
06		Token rotation time in ms
07	LO	Data master failed during token ownership bit map
	HI	Program master failed during token ownership bit map
08	LO	Data master token owner work bit map
	HI	Program master token owner work bit map
09	LO	Data slave token owner work bit map
	HI	Program slave token owner work bit map
10	HI	Data slave/get slave command transfer request bit map
11	LO	Program master/get master rsp transfer request bit map
	HI	Program slave/get slave command transfer request bit map
12	LO	Program master connect status bit map
	HI	Program slave automatic logout request bit map
13	LO	Pretransmit deferral error counter
	HI	Receive buffer DMA overrun error counter
14	LO	Repeated command received counter
	HI	Frame size error counter

15		If Word 1 bit 15 is *not set*, Word 15 has the following meaning:
	LO	Receiver collision abort error counter
	HI	Receiver alignment error counter
		If Word 1 bit 15 is *set*, Word 15 has the following meaning:
	LO	Cable A framing error
	HI	Cable B framing error
16	LO	Receiver CRC error counter
	HI	Bad packet length error counter
17	LO	Bad link address error counter
	HI	Transmit buffer DMA underrun error counter
18	LO	Bad internal packet length error counter
	HI	Bad MAC function code error counter
19	LO	Communication retry counter
	HI	Communication failed error counter
20	LO	Good receive packet success counter
	HI	No response received error counter
21	LO	Exception response received error counter
	HI	Unexpected path error counter
22	LO	Unexpected response error counter
	HI	Forgotten transaction error counter
23	LO	Active station table bit map, nodes 1 ... 8
	HI	Active station table bit map, nodes 9 ...16
24	LO	Active station table bit map, nodes 17 ... 24
	HI	Active station table bit map, nodes 25 ... 32
25	LO	Active station table bit map, nodes 33 ... 40
	HI	Active station table bit map, nodes 41 ... 48
26	LO	Active station table bit map, nodes 49 ... 56
	HI	Active station table bit map, nodes 57 ... 64
27	LO	Token station table bit map, nodes 1 ... 8
	HI	Token station table bit map, nodes 9 ... 16
28	LO	Token station table bit map, nodes 17 ... 24
	HI	Token station table bit map, nodes 25 ... 32
29	LO	Token station table bit map, nodes 33 ... 40
	HI	Token station table bit map, nodes 41 ... 48
30	LO	Token station table bit map, nodes 49 ... 56
	HI	Token station table bit map, nodes 57 ... 64

31	LO	Global data present table bit map, nodes 1 ... 8
	HI	Global data present table bit map, nodes 9 ... 16
32	LO	Global data present table bit map, nodes 17 ... 24
	HI	Global data present table bit map, nodes 25 ... 32
33	LO	Global data present table bit map, nodes 33 ... 40
	HI	Global data present table bit map, nodes 41 ... 48
34	LO	Global data present table map, nodes 49 ... 56
	HI	Global data present table bit map, nodes 57 ... 64
35	LO	Receive buffer in use bit map, buffer 1 8
	HI	Receive buffer in use bit map, buffer 9 ... 16
36	LO	Receive buffer in use bit map, buffer 17 ... 24
	HI	Receive buffer in use bit map, buffer 25 ... 32
37	LO	Receive buffer in use bit map, buffer 33 ... 40
	HI	Station management command processed initiation counter
38	LO	Data master output path 1 command initiation counter
	HI	Data master output path 2 command initiation counter
39	LO	Data master output path 3 command initiation counter
	HI	Data master output path 4 command initiation counter
40	LO	Data master output path 5 command initiation counter
	HI	Data master output path 6 command initiation counter
41	LO	Data master output path 7 command initiation counter
	HI	Data master output path 8 command initiation counter
42	LO	Data slave input path 41 command processed counter
	HI	Data slave input path 42 command processed counter
43	LO	Data slave input path 43 command processed counter
	HI	Data slave input path 44 command processed counter
44	LO	Data slave input path 45 command processed counter
	HI	Data slave input path 46 command processed counter
45	LO	Data slave input path 47 command processed counter
	HI	Data slave input path 48 command processed counter
46	LO	Program master output path 81 command initiation counter
	HI	Program master output path 82 command initiation counter
47	LO	Program master output path 83 command initiation counter
	HI	Program master output path 84 command initiation counter
48	LO	Program master command initiation counter
	HI	Program master output path 86 command initiation counter

49	LO	Program master output path 87 command initiation counter
	HI	Program master output path 88 command initiation counter
50	LO	Program slave input path C1 command processed counter
	HI	Program slave input path C2 command processed counter
51	LO	Program slave input path C3 command processed counter
	HI	Program slave input path C4 command processed counter
52	LO	Program slave input path C5 command processed counter
	HI	Program slave input path C6 command processed counter
53	LO	Program slave input path C7 command processed counter
	HI	Program slave input path C8 command processed counter

18.15 TCP/IP Ethernet Statistics

A TCP/IP EtherNet board responds to *Get Local Statistics* and *Set Local Statistics* commands with the following information:

Word	Meaning
00 ... 02	MAC address
03	Board Status
04 and 05	Number of receiver interrupts
06 and 07	Number of transmitter interrupts
08 and 09	Transmit timeout error count
10 and 11	Collision detect error count
12 and 13	Missed packets
14 and 15	Memory error
16 and 17	Number of times driver has restarted lance
18 and 19	Receive framing error
20 and 21	Receiver overflow error
22 and 23	Receive CRC error
24 and 25	Receive buffer error
26 and 27	Transmit silo underflow
28 and 29	Late collision
30 and 31	Lost carrier
32 and 33	Number of retries
34 and 35	IP address

Chapter 19
Ladder Logic Subroutines

- ☐ Subroutine Overview
- ☐ JSR
- ☐ LAB
- ☐ RET
- ☐ A Subroutine Example
- ☐ CTIF
- ☐ Some Cautionary Notes About Subroutines

19.1 Subroutine Overview

In the Quantum PLCs and in several 984 PLCs, the JSR instruction (section 19.2) can be used to issue a call from the scheduled flow of ladder logic to a subroutine in the last (unscheduled) logic segment. Two additional instructions within the subroutine segment itself are used to mark the beginning and end of each subroutine. The LAB function (section 19.3) labels the starting point of the subroutine. The RET instruction (section 19.4) returns you from the subroutine network to the position in scheduled logic where the JSR call was issued.

19.1.1 The Value of Subroutines

Ladder logic subroutines allow you to save memory space in the user logic table in cases where you need to implement the same logic functions multiple times in a single scan. You need only create the logic once, store it in the logic segment reserved for subroutines, and call it from user logic whenever it is needed.

Subroutines can also be helpful in reducing total scan time. Portions of logic that require only infrequent solution in logic scans can be placed in the subroutine segment and called from user logic only on those scans where it is needed.

19.1.2 Where to Store Subroutines in Ladder Logic

All ladder logic subroutines must be built in the *last* segment of user logic. This segment must be removed from the segment scheduler—it is not part of the regular order-of-solve table and is reserved for subroutine and interrupt handling (Chapter 20) logic.

☞ **Note:** This means that you must specify at least one more segment than is required for regular user logic in the configuration table.

Controllers that support subroutines provide as many as 255 address locations for subroutine ladder logic. Each subroutine must start at the beginning of a network in the last logic segment. There is no set limit on the number of networks in the subroutine segment, and since the segment is unscheduled only the subroutine logic called by a specific interrupt will be solved.

19.2 JSR

When the logic scan encounters an enabled JSR instruction, it stops the normal logic scan and jumps to the specified *source* subroutine in the last (unscheduled) segment of ladder logic.

You can use a JSR instruction anywhere in user logic, even within the subroutine segment. The process of calling one subroutine from another subroutine is called *nesting*. The system allows you to nest up to 100 subroutines— however, we recommend that you use no more than three nesting levels. You may also perform a recursive form of nesting called *looping*, whereby a JSR call within the subroutine recalls the same subroutine.

19.2.1 Characteristics

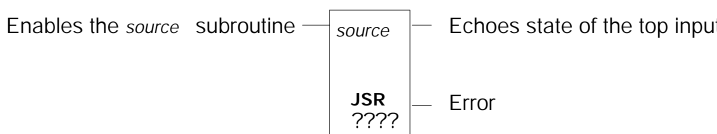
Size
Two nodes high

PLC Compatibility
- Not available in the 984A/B/X Chassis Mount PLCs
- Standard in all other PLC types

Opcode
DE

19.2.2 Representation in Ladder Logic

Block Structure



Input
The input to the top node enables the *source* subroutine specified by the number in the top node.

Output

JSR has two outputs. The output from the top node echoes the state of the top input. The output from the bottom node goes ON to indicate an error in the subroutine jump.

Top Node Content

The top node contains the *source* pointer, the indicator of the subroutine to which the logic scan will jump. The *source* may be: specified as:

- Specified explicitly as an integer value in the range 1 ... 255 for a PLC with a 16-bit CPU
- Specified explicitly as an integer value in the range 1 ... 1023 for a PLC with a 24-bit CPU
- Stored in a 4*x* holding register as a value in the range 1 ... 255 for a PLC with a 16-bit CPU
- Stored in a 4*x* holding register as a value in the range 1 ... 1023 for a PLC with a 24-bit CPU

Bottom Node Content

The bottom node displays a string of four question marks—always enter the constant value 1 in this node.

19.3 LAB

The LAB instruction is used to label the starting point of a subroutine or an interrupt handler. This instruction must be programmed in row 1, column 1 of a network in the subroutine segment (the last, unscheduled segment of ladder logic). LAB is a one-node function block.

LAB also serves as a default return from the subroutine or interrupt handler in the preceding networks. If the PLC is executing a series of subroutine or interrupt handler networks and it reaches a network that begins with LAB instruction, it assumes that the previous subroutine or interrupt handler is finished, and it returns the logic scan to the scheduled logic.

19.3.1 Characteristics

Size

One node high

PLC Compatibility

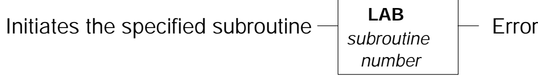
- Not available in the 984A/B/X Chassis Mount PLCs
- Standard in all other PLC types

Opcode

BE hex

19.3.2 Representation in Ladder Logic

Block Structure



Input

The input to the top node initiates the subroutine or interrupt handler specified by the number in the bottom node.

Outputs

The output from the top node goes ON to indicate an error in the initiation of the specified subroutine or interrupt handler.

Node Content

The integer value entered in the node identifies the *subroutine* (or interrupt handler) *number* you are about to execute. The value can range from 1 ... 255 for a PLC with a 16-bit CPU or 1 ... 1023 for a PLC with a 24-bit CPU.

If more than one network begins with a LAB instruction with the same *subroutine* value, the lowest-numbered network is used as the starting point for the subroutine.

19.4 RET

The RET instruction may be used to conditionally return to scheduled logic at the node immediately following the most recently executed JSR block or at the point where the interrupt occurred. This instruction can be implemented only from within the subroutine segment—the (unscheduled) last segment in the user logic program.

Note: If a subroutine does not contain a RET block, either a LAB block or the end-of-logic (whichever comes first) serves as the default return from the subroutine or interrupt handler.

19.4.1 Characteristics

Size

One node high

PLC Compatibility

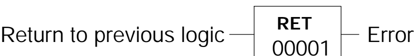
- Not available in the 984A/B/X Chassis Mount PLCs
- Standard in all other PLC types

Opcode

FE hex

19.4.2 Representation in Ladder Logic

Block Structure



Input

When the input to the node is ON, RET returns the logic scan to the node immediately following the most recently executed JSR instruction or to the point where the interrupt occurred in the logic scan.

Output

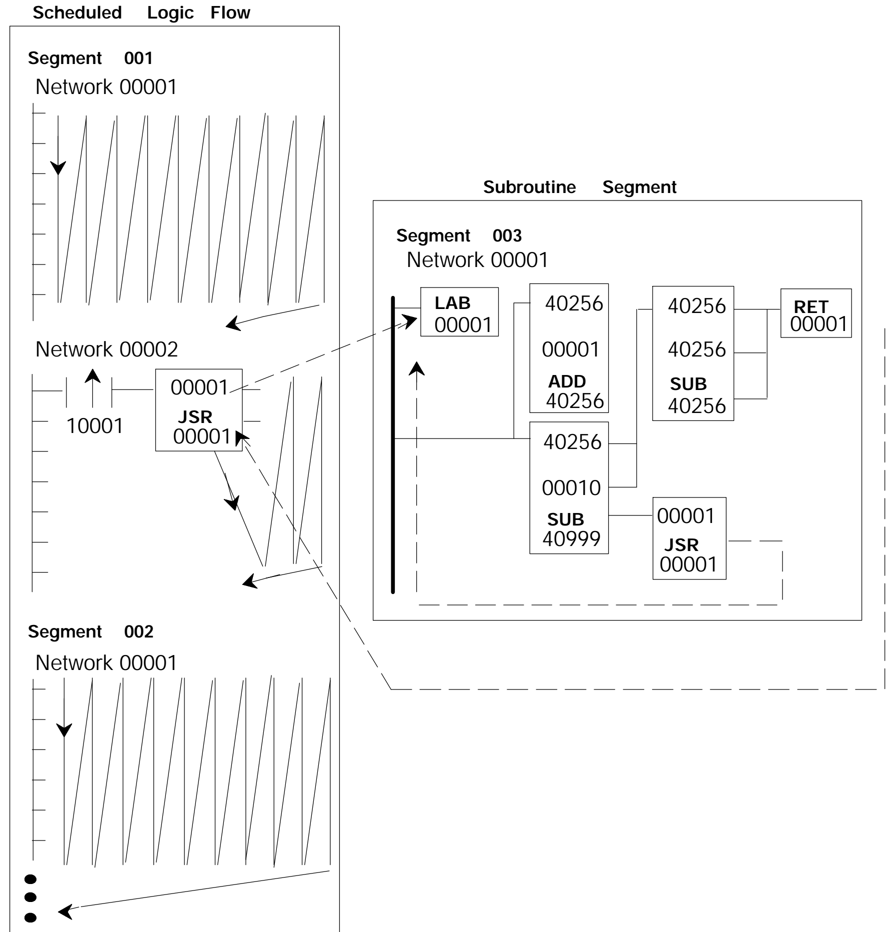
The output from the top node goes ON to indicate an error in the specified subroutine or interrupt handler.

Node Content

The node contains the constant value 00001.

19.5 A Subroutine Example

The example below shows a series of three user logic networks, the last of which is used for an up-counting subroutine. Segment 3 has been removed from the order-of-solve table in the segment scheduler:



When input 10001 to the JSR block in network 2 of segment 1 transitions from OFF to ON, the logic scan jumps to subroutine #1 in network 1 of segment 3.

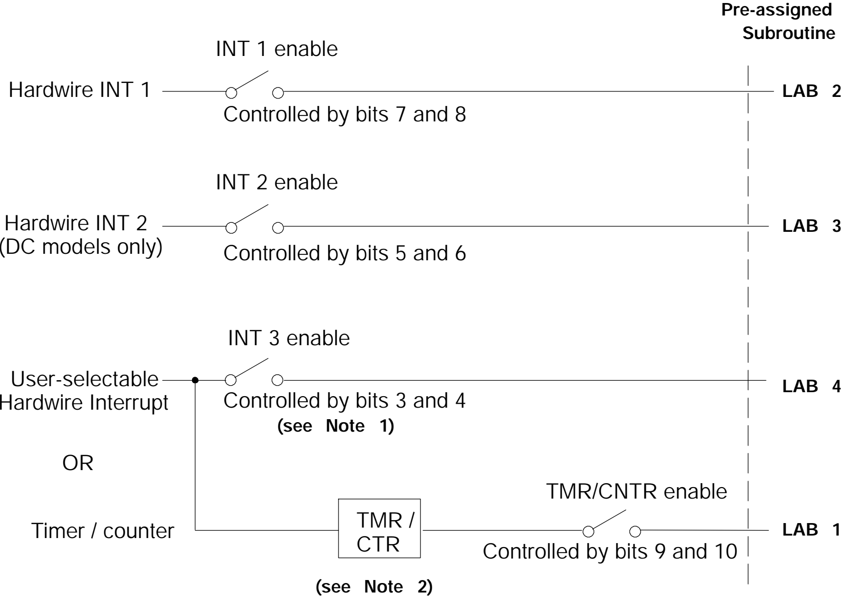
The subroutine will internally loop on itself ten times, counted by the ADD block. The first nine loops end with the JSR block in the subroutine (network 1 of segment 3) sending the scan back to the LAB block. Upon completion of the tenth loop, the RET block sends the logic scan back to the scheduled logic at the JSR node in network 2 of segment 1.

19.6 CTIF

The CTIF instruction is used with the Micro PLCs to set up the inputs for hard-wired interrupt and/or hard-wired counter/timer operations. This instruction always starts and finishes in the same scan.

The CTIF instruction is a configuration/operation tool for Modicon Micro PLCs that contain hardware interrupts (all models except the 110CPU311 Models). The actual counter/timer and interrupts are located in the PLC hardware, and the CTIF instruction is what is used to set up this hardware.

The illustrations below show how the *configuration switches* interact with the interrupt functions.



Note 1. INT 3 is available only when the timer / counter is not used.

Note 2. Bits 15 and 16 select the mode (TMR or CTR). In CTR mode, pulses on the input are counted. In TMR mode, the input acts as a timer gate and must be high to time.

Input Type	CPU Availability	State RAM References	Subroutine
User-selectable timer/ counter interrupt	All 411s, 512s and 612s	10081—updated once/ scan; 10084—updated at the start of each subroutine	Subroutine #1
Hardwire interrupt 1	All 411s, 512s, and 612s	10082—updated once/ scan; 10085—updated at the start of each subroutine	Subroutine #2
Hardwire interrupt 2	Only units that use DC power	10083—updated once/ scan; 10086—updated at the start of each subroutine	Subroutine #3
User-selectable inter-rupt 3	All 411s, 512s, and 612s	10081—updated once/ scan; 10084—updated at the start of each subroutine	Subroutine #4

19.6.1 Characteristics

Size

Two nodes high

PLC Compatibility

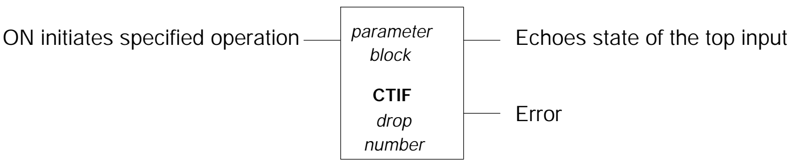
- Standard in the Micro PLCs
- Not available in any other models

Opcode

1F hex

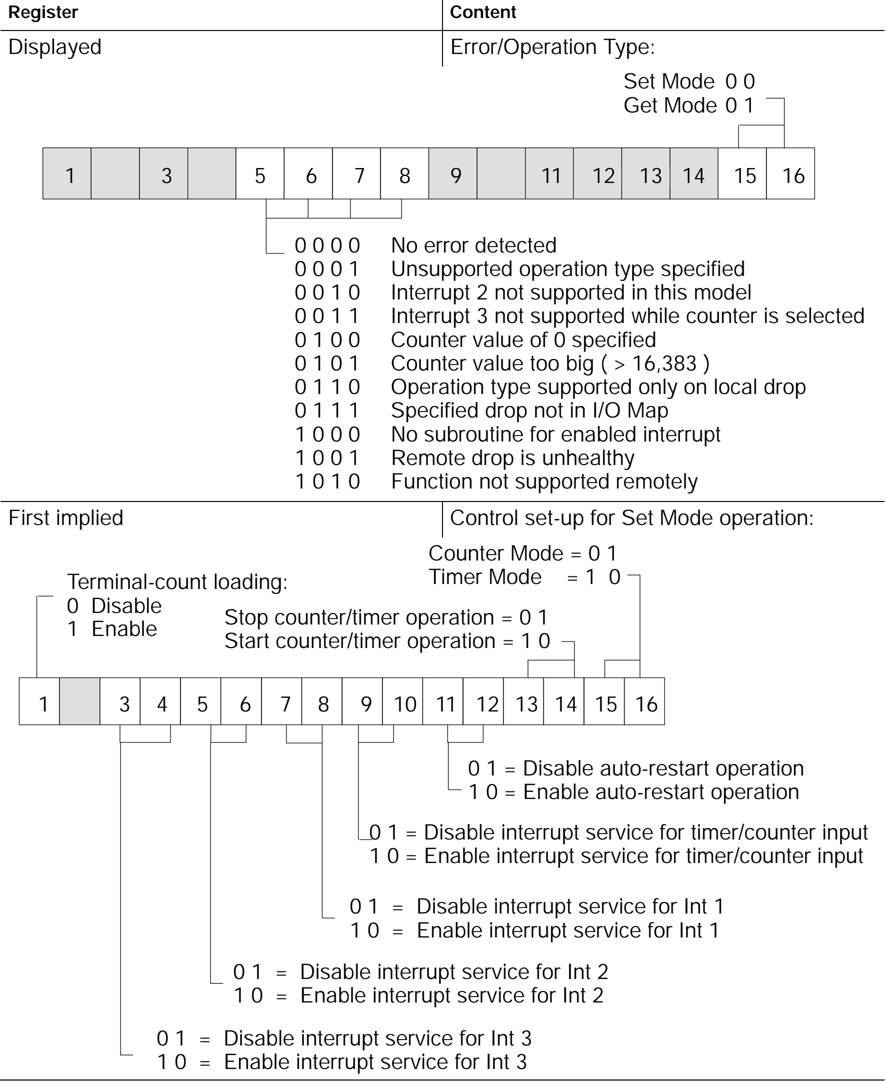
19.6.2 Representation in Ladder Logic

Block Structure

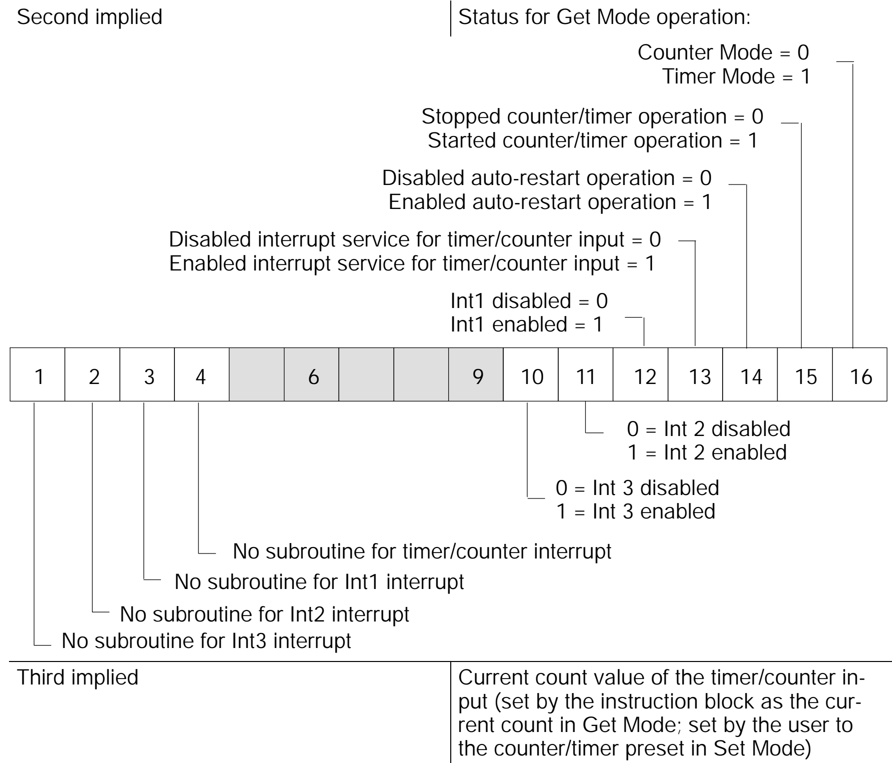


Top Node Content

The 4*x* register entered in the top node is the first of four contiguous holding register in the CTIF *parameter block*. The registers in the *parameter block* are utilized as follows:



Register
Content
Displayed
Error/Operation Type:
Set Mode 0 0
Get Mode 0 1
1
3
5
6
7
8
9
11
12
13
14
15
16
0 0 0 0 No error detected
0 0 0 1 Unsupported operation type specified
0 0 1 0 Interrupt 2 not supported in this model
0 0 1 1 Interrupt 3 not supported while counter is selected
0 1 0 0 Counter value of 0 specified
0 1 0 1 Counter value too big (> 16,383)
0 1 1 0 Operation type supported only on local drop
0 1 1 1 Specified drop not in I/O Map
1 0 0 0 No subroutine for enabled interrupt
1 0 0 1 Remote drop is unhealthy
1 0 1 0 Function not supported remotely
First implied
Control set-up for Set Mode operation:
Counter Mode = 0 1
Timer Mode = 1 0
Terminal-count loading:
0 Disable
1 Enable
Stop counter/timer operation = 0 1
Start counter/timer operation = 1 0
1
3
4
5
6
7
8
9
10
11
12
13
14
15
16
0 1 = Disable auto-restart operation
1 0 = Enable auto-restart operation
0 1 = Disable interrupt service for timer/counter input
1 0 = Enable interrupt service for timer/counter input
0 1 = Disable interrupt service for Int 1
1 0 = Enable interrupt service for Int 1
0 1 = Disable interrupt service for Int 2
1 0 = Enable interrupt service for Int 2
0 1 = Disable interrupt service for Int 3
1 0 = Enable interrupt service for Int 3



Bottom Node Content

The integer value entered in the bottom node indicates the *drop number* where the operation will be performed. The *drop number* is in the range 1 ... 5.

19.7 Some Cautionary Notes About Subroutines

You should always keep your subroutine logic as straightforward as possible for debugging purposes. The power flow displayed on your programming panel is invalid in the subroutine networks and is therefore more difficult to troubleshoot.

Note: We recommend that you debug your ladder logic programs before making them subroutines.

Counters work on a state change basis—when the top input transitions from OFF to ON. Timers do not function properly from within a subroutine unless that subroutine is executed on every scan.

Note: Multiple scan functions do not function from within a subroutine.

Caution: We strongly recommend that you *do not* control real-world outputs from within a ladder logic subroutine. Control of such coils would be possible only when the subroutine was executed.

Chapter 20
Ladder Logic Interrupt Handling for Quantum PLCs

- Overview
- Interval Timer Interrupt (ITMR) Instruction
- Interrupt Mask/Unmask Instructions
- Immediate I/O (IMIO) Instruction

20.1 Overview

The following instructions are designed for a variety of functions known generally as fast I/O updating. They fall into four categories:

- Interval timer interrupt generation (see section 20.2)
- Interrupt masking and unmasking (see section 20.3)
- Immediate I/O access (see section 20.4)

These functional categories are described in detail in the following sections of this chapter.

☞ **Note:** These instructions are designed to be run in Quantum Automation Series PLCs. They are not supported in 984 CPUs.

20.1.1 Interrupt-related Performance

The instructions described in this chapter operate with minimum processing overhead. The performance of interrupt-related instructions is especially critical. Using a interval timer interrupt (ITMR) instruction adds about 6% to the scan time of the scheduled ladder logic—this increase does not include the time required to execute the interrupt handler subroutine associated with the interrupt.

The following table shows the minimum and maximum interrupt latency times you can expect:

Interrupt Latency Times

ITMR overhead	No work to do	60 μs/ms
Response time	Minimum	98 μs
	Maximum during logic solve and Modbus command reception	400 μs
Total overhead (not counting normal logic solve time)		155 μs

These latency times assume only one interrupt at a time.

20.1.2 Instructions that Cannot Be Used in an Interrupt Handler

The following (*nonreenterant*) ladder logic instruction cannot be used inside an interrupt handler subroutine:

- MSTR
- READ/WRIT
- PCFL/EMTH
- Equation Networks
- T1.0/T0.1/T.01 timers (will not set error bit 2, timer results invalid)
- User loadables (will not set error bit 2)

If any of these instructions are placed in an interrupt handler, the subroutine will be aborted, the error output on the ITMR instruction (page 390) that generated the interrupt will go ON, and bit 2 in the status register will be set.

20.2 Interval Timer Interrupt (ITMR) Instruction

The ITRM instruction allows you to define an interval timer that generates interrupts into the normal ladder logic scan and initiates the execution of an interrupt handling subroutine. The user-defined interrupt handler is a ladder logic subroutine created in the last, unscheduled segment of ladder logic with its first network marked by a LAB instruction (see section 19.3). Subroutine execution is asynchronous to the normal scan cycle.

Up to 16 ITRM instructions can be programmed in an application. Each interval timer can be programmed to initiate the same or different interrupt handler subroutines, controlled by the JSR/LAB method described in Chapter 19.

Each instance of the interval timer is delayed for a programmed interval while the PLC is running, then generates a processor interrupt when the interval has elapsed.

An interval timer can execute at any time during normal logic scan, including system I/O updating or other system housekeeping operations. The resolution of each interval timer is 1 ms. An interval can be programmed in units of 1 ms, 10 ms, 100 ms, or 1 s. An internal counter increments at the specified resolution.

20.2.1 Characteristics

Size

Two nodes high

PLC Compatibility

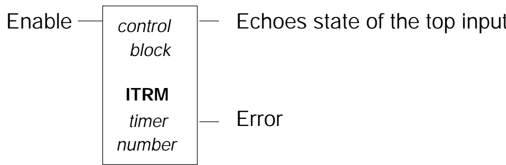
- Standard in all Quantum PLCs
- Not available in other PLCs

Opcode

45_{16}

20.2.2 Representation in Ladder Logic

Block Structure



Input

When the top input is energized, the ITRM instruction is enabled. It begins counting the programmed time interval. When that interval has expired the counter is reset and the designated error handler logic executes.

When the top input is not energized, the following events occur:

- All indicated errors are cleared
- The timer is stopped
- The time count is either reset or held, depending on the state of bit 15 of the first register in the *control block* (the displayed register in the top node)
- Any pending masked interrupt is cleared for this timer

Outputs

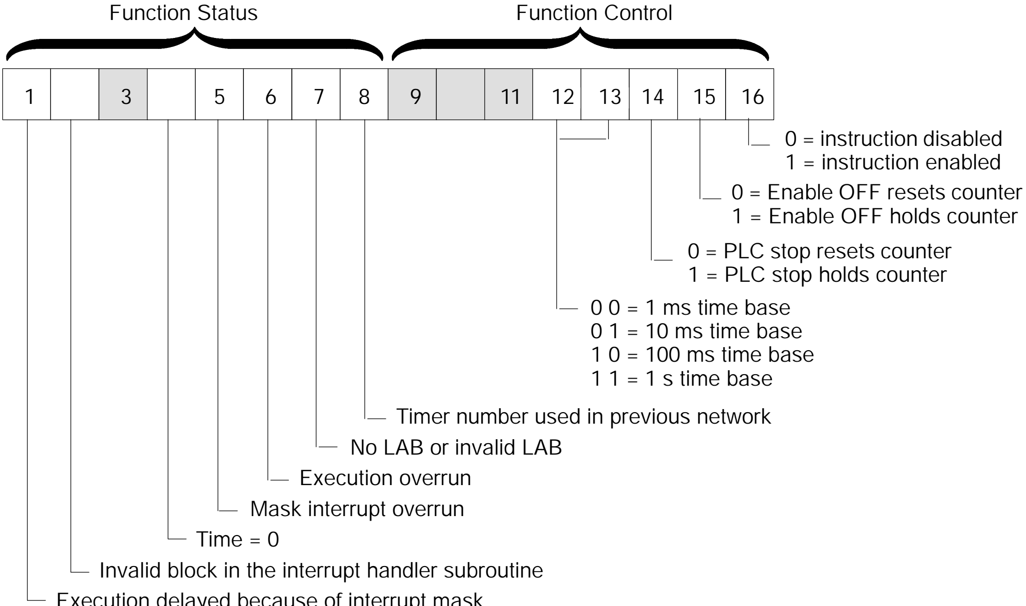
ITRM has two outputs. The output from the top node echoes the state of the top input.

The output from the bottom node goes ON when an error occurs. The source of the error may be in the programmed parameters or a runtime execution error.

Top Node Content

The top node contains the the first of three contiguous 4*x* registers in the ITRM *control block*. These registers are used to specify the parameters required to program each ITRM instruction.

The lower eight bits of the first (displayed) register in the *control block* allow you to specify function control parameters, and the upper eight bits are used to display function status:



In the second register of the *control block* , specify a value representing the interval at which the ITRM instruction will generate interrupts and initiate the execution of the interrupt handler. The interval will be incremented in the units specified by bits 12 and 13 of the first *control block* register—i.e., 1 ms, 10 ms, 100 ms, or 1 s units.

In the third register of the *control block* , specify a value indicating the label (LAB) number that will start the interrupt handler subroutine, The number must be in the range 1 ... 1023.

Note: We recommend that the size of the logic subroutine associated with the LAB be minimized so that the application does not become interrupt-driven.

Bottom Node Content

In the bottom node, specify a value in the range 1 ... 16, indicating the *timer number* assigned to this ITMR instruction. The number entered here must be unique with respect to all other ITMR instructions in the application.

20.3 Interrupt Mask/Unmask Instructions

Three interrupt control instructions are available to help protect data in both the normal (scheduled) ladder logic and the (unscheduled) interrupt handling subroutine logic. These are the Interrupt Disable (ID) instruction, the Interrupt Enable (IE) instruction, and the Block Move with Interrupts Disabled (BMDI) instruction.

An interrupt that is executed in the time frame after an ID instruction has been solved and before the next IE instruction has been solved is buffered. The execution of a buffered interrupt takes place at the time the IE instruction is solved. If two or more interrupts of the same type occur between the ID ... IE solve, the *mask interrupt overrun* error bit is set, and the subroutine initiated by the interrupts is executed only one time.

The BMDI instruction can be used to mask both a timer-generated and local I/O-generated interrupts, perform a single block data move, then unmask the interrupts. It allows for the exchange of a block of data either within the subroutine or at one or more places in the scheduled logic program.

BMDI instructions can be used to reduce the time between the disable and enable of interrupts. For example, BMDI instructions can be used to protect the data used by the interrupt handler when the data is updated or read by Modbus, Modbus Plus, Peer Cop, or Distributed I/O (DIO).

20.3.1 ID Characteristics

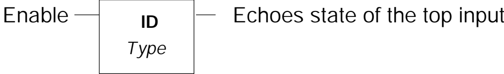
Size
One node high

PLC Compatibility

- Standard in Quantum PLCs
- Not available in other PLCs

Opcode
47_{16}

Block Structure



Input
When the input is energized, the ID instruction masks timer-generated and/or local I/O-generated interrupts.

Outputs
The output echoes the state of the input.

Node Content
Enter a constant integer in the range 1 ... 3 in the node. The value represents the *type* of interrupt to be masked by the ID instruction, where:

Integer Value	Interrupt Type
3	Timer interrupt masked
2	Local I/O module interrupt masked
1	Both interrupt types masked

20.3.2 IE Characteristics

Size

One node high

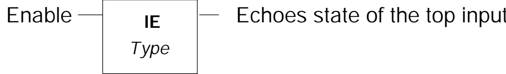
PLC Compatibility

- Standard in Quantum PLCs
- Not available in other PLCs

Opcode

48_{16}

Block Structure



Input

When the input is energized, the IE instruction unmasks interrupts from the timer or local I/O module and responds to the pending interrupts by executing the designated subroutines.

Outputs

The output echoes the state of the input.

Node Content

Enter a constant integer in the range 1 ... 3 in the node. The value represents the *type* of interrupt to be unmasked by the IE instruction, where:

Integer Value	Interrupt Type
3	Timer interrupt unmasked
2	Local I/O module interrupt unmasked
1	Both interrupt types unmasked

20.3.3 BMDI Characteristics

Size

Three nodes high

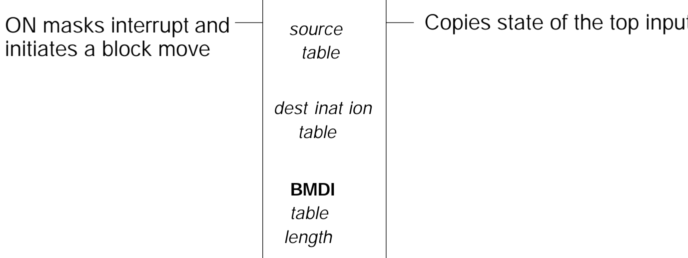
PLC Compatibility

- Standard in Quantum PLCs
- Not available in other PLCs

Opcode

49_{16}

Block Structure



Input

BMDI has one control input (to the top node). This input masks the interrupt, initiates a block move (BLKM) operation, then unmasks the interrupts.

Output

BMDI produces one output (from the top node), which echoes the state of the top input.

Top Node Content

The top node specifies the *source table* that will have its contents copied in the block move. The node may reference:

- The first 0*x* reference in a table of contiguous coils or discrete outputs
- The first 1*x* reference in a table of contiguous discrete inputs
- The first 3*x* reference in a table of contiguous input registers
- The first 4*x* reference in a table of contiguous holding registers

Middle Node Content

The middle node specifies the *destination table* where the contents of the *source table* will be copied in the block move. The node may reference:

- The first 0*x* reference in a table of contiguous coils or discrete outputs
- The first 4*x* reference in a table of contiguous holding registers

Bottom Node Content

The integer value entered in the bottom node specifies the *table size*—i.e., the number of registers—in the *source* and *destination* tables; they are of equal length. The *table length* is in the range 1 ... 100.

20.4 Immediate I/O (IMIO) Instruction

The IMIO instruction permits access of specified I/O modules from within ladder logic. This differs from normal I/O processing, where inputs are accessed at the beginning of the logic solve for the segment in which they are used and outputs are updated at the end of the segment's solution. The I/O modules being accessed must reside in the local backplane with the Quantum PLC or the local racks 1-4 of the Compact PLC.

In order to use IMIO instructions, the local I/O modules to be accessed must be designated in the I/O Map in your panel software.

20.4.1 Characteristics

Size

Two nodes high

PLC Compatibility

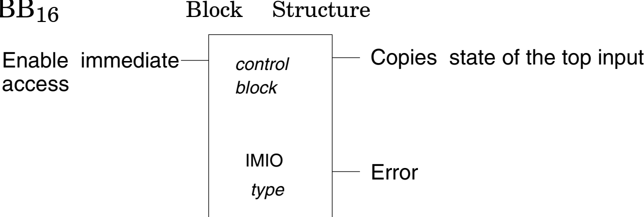
- Standard in Quantum PLCs
- Modicon 32-bit E Compact PLCs (E984-258/265/275/285)

  NOTE:
  This IMIO function block will NOT work with the following Compact I/O modules due to hardware design restrictions inherent with these modules.
  - AS-BADU-204
  - AS-BADU-205
  - AS-BADU-206
  - AS-BADU-216

Opcode

BB_{16}

Block Structure



Input

IMIO has one control input (to the top node) that enables the immediate I/O access when it is ON.

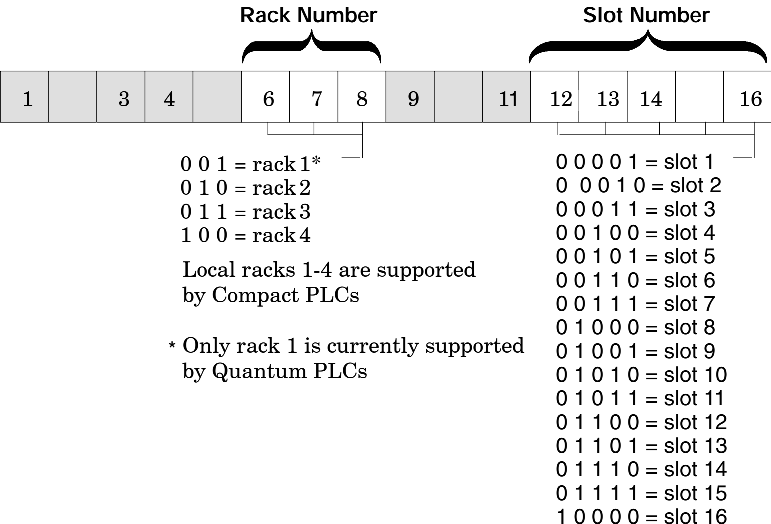
Output

IMIO produces two outputs. The output from the top node echoes the state of the top input. The output from the bottom node goes ON when the instruction reports an error The nature of the error is indicated by a code in the error status register in the IMIO *control block* , which is described below.

Top Node Content

The 4x register in the top node is the first of two contiguous registers in the IMIO *control block*. The first (the displayed) register in the *control block* specifies the physical address of the I/O module to be accessed. The second (the implied) register in the *control block* logs the error status, which is maintained by the instruction.

The high byte of the displayed register in the *control block* allows you to specify which rack the I/O module to be accessed resides in, and the low byte allow you to specify slot number within the specified rack where the I/O module resides.



The implied register in the *control block* will contain an error code when the instruction detects an error. This register is maintained by the IMIO instruction.

Error Code	Meaning
2001	Invalid *type* specified in the bottom node
2002	Problem with the specified I/O slot -- either an invalid slot number entered in the displayed register of the *control block* or the I/O Map does not contain the correct module definition for this slot
2003	A *type* 3 operation is specified in the bottom node, and the module is not bidirectional (see page 400)
F001	Specified I/O module is not healthy

Bottom Node Content

Enter a constant integer in the range 1 ... 3 in the bottom node. The value represents the *type* of operation to be performed by the IMIO instruction, where:

Integer Value	Type of Immediate Access
1	Input operation—transfers data from the the specified module to state RAM
2	Output operation—transfers data from state RAM to the specified module
3	I/O operation—does both input and output if the specified module is bidirectional

Chapter 21
Closed Loop Control Instructions

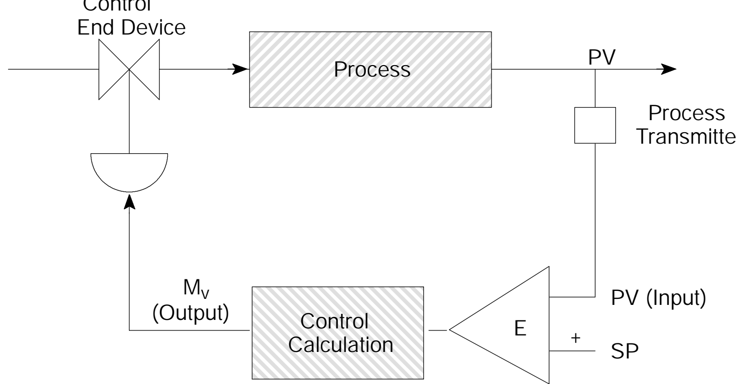
A process control function library (PCFL) instruction is provided in some PLCs with large memory and processing speed abilities. This instruction provides functions for performing a wide range of process control applications. For many other PLCs, the PID2 instruction is provided for proportional-integral-derivative (PID) calculations.

21.1 A Closed Loop Control System

An analog closed loop control system is one in which the deviation from an ideal process condition is measured, analyzed, and adjusted in an attempt to obtain (and maintain) zero error in the process condition. Provided with the Enhanced Instruction Set is a proportional-integral-derivative function block called PID2, which allows you to establish closed loop (or *negative feedback*) control in ladder logic.

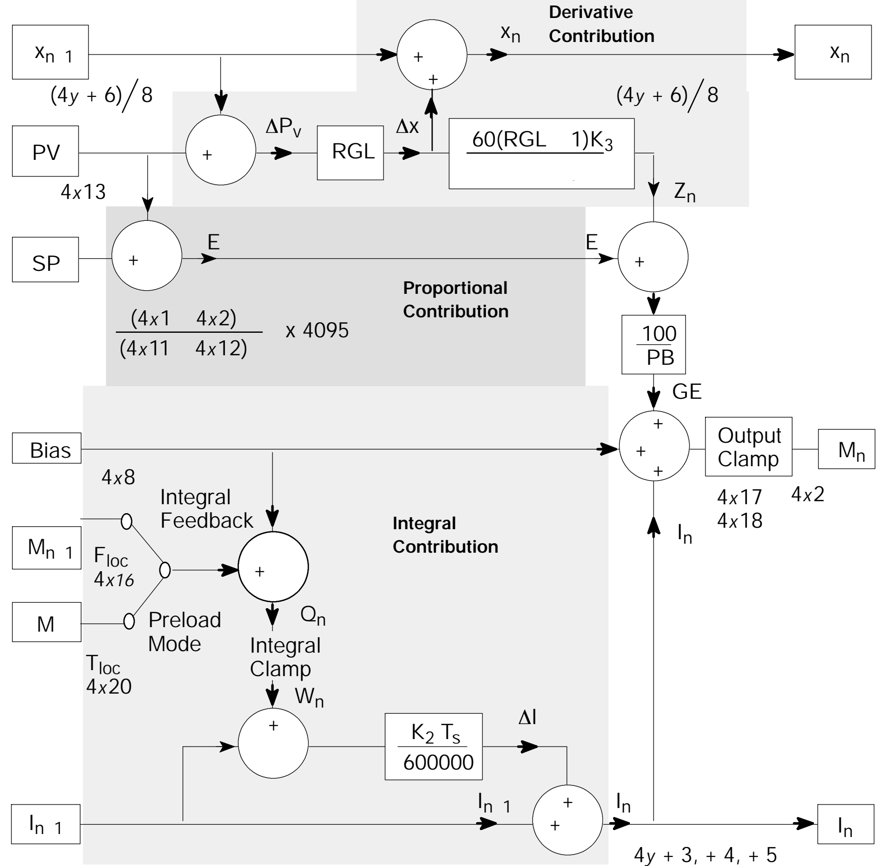
21.1.1 Set Point and Process Variable

The desired (zero error) control point, which you will define in the PID2 block, is called the *set point* (SP). The conditional measurement taken against SP is called the process variable (PV). The difference between the SP and the PV is the *deviation* or *error* (E). E is fed into a control calculation that produces a *manipulated variable* (M_V) used to adjust the process so that PV = SP (and, therefore, E = 0).



21.2 PID2

The PID2 instruction implements an algorithm that performs proportional-integral-derivative operations. The algorithm tunes the closed loop operation in a manner similar to traditional pneumatic and analog electronic loop controllers. It uses a rate gain limiting (RGL) filter on the PV as it is used for the derivative term only, thereby filtering out higher-frequency PV noise sources (random and process generated).



where:

- E = error, expressed in raw analog units
- SP = set point, in the range 0 ... 4095
- PV = process variable, in the range 0 ... 4095
- x = filtered PV

K_2 = integral mode gain constant, expressed in 0.01 min ^{1}
K_3 = derivative mode gain constant, expressed in hundredths of a minute
RGL = rate gain limiting filter constant, in the range 2 ... 30
T_s = solution time, expressed in hundredths of a second
PB = proportional band, in the range 5 ... 500%
$bias$ = loop output bias factor, in the range 0 ... 4095
M = loop output
GE = gross error, the proportional-derivative contribution to the loop output
Z = derivative mode contribution to GE
Q_n = unbiased loop output
F = feedback value, in the range 0 ... 4095
I = integral mode contribution to the loop output
I_{low} = anti-reset-windup low SP, in the range 0 ... 4095
I_{high} = anti-reset-windup high SP, in the range 0 ... 4095

$$K_1 = \frac{100}{PB}$$

Note: The integral mode contribution calculation actually integrates the difference of the output and the integral sum—this is effectively the same as integrating the error.

Proportional Control

With proportional-only control (P), you can calculate the manipulated variable by multiplying error by a proportional constant, K_1, then adding a bias:

$$M_v = K_1E + bias$$

However, process conditions in most applications are changed by other system variables so that the bias does not remain constant; the result is offset error, where PV is constantly offset from the SP. This condition limits the capability of proportional-only control.

Note: The value in the integral term—in registers 4*y* + 3, 4*y* + 4, and 4*y* + 5—is always used, even when the integral mode is not enabled. Using this value is necessary to preserve bumpless transfer between modes. If you wish to disable bumpless transfer, these three registers must be cleared.

Proportional-Integral Control

To eliminate this offset error without forcing you to manually change the bias, an integral function can be added to the control equation:

$$M_v = K_1(E + K_2 \int_0^t E\Delta t)$$

Proportional-integral control (PI) eliminates offset by integrating E as a function of time. K_1 is the integral constant expressed as rep/min. As long as $E \neq 0$, the integrator increases (or decreases) its value, adjusting M_v. This continues until the offset error is eliminated.

Proportional-Integral-Derivative Control

You may want to add derivative functionality to the control equation to minimize the effects of frequent load changes or to override the integral function in order to get to the SP condition more quickly:

$$M_v = K_1(E + K_2 \int_0^t E\Delta t + K_3 \frac{\Delta PV}{\Delta t})$$

Proportional-integral-derivative (PID) control can be used to save energy in the process or as a safety valve in the event of a sudden, unexpected change in process flow. K_3 is the derivative time constant expressed as min. ΔPV is the change in the process variable over a time period of Δt.

21.2.1 Characteristics

Size

Three nodes high

PLC Compatibility

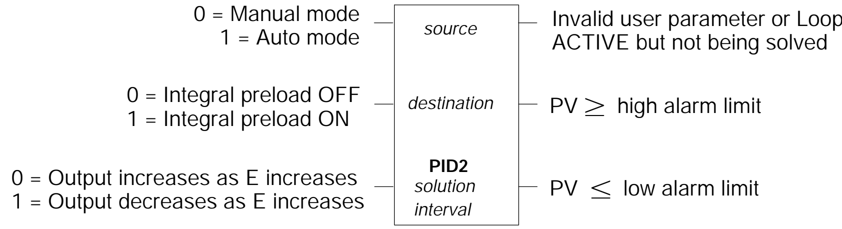
Standard in all PLC types except the 984A/B/X Chassis Mounts, where it is available as a loadable

Opcode

5E hex

21.2.2 Representation

Block Structure



Inputs

PID2 has three control inputs. The state of the input to the top node determines whether the operation will be initiated automatically or manually. The state of the input to the middle node indicates whether or not an integral preload is used. The state of the input to the bottom node indicates whether the output from the operation will increase or decrease as the error increases.

Outputs

PID2 can produce three possible outputs, each indicating an error condition.

Top Node Content

The 4*x* register entered in the top node is the first of 21 contiguous holding registers in a *source* block. The contents of the fifth ... eighth implied registers determine whether the operation will be P, PI, or PID:

Operation	Fifth Implied	Sixth Implied	Seventh Implied	Eighth Implied
P	ON			ON
PI	ON	ON		
PID	ON	ON	ON	

The *source* block comprises the following register assignments:

Register	Name	Content
Displayed	Scaled PV	Loaded by the block each time it is scanned; a linear scaling is done on register 4*x* + 13 using the high and low ranges from registers 4*x* + 11 and 4*x* + 12: Scaled PV = $\frac{4x13}{4095}$ x (4*x*11 4*x*12) + 4*x*12

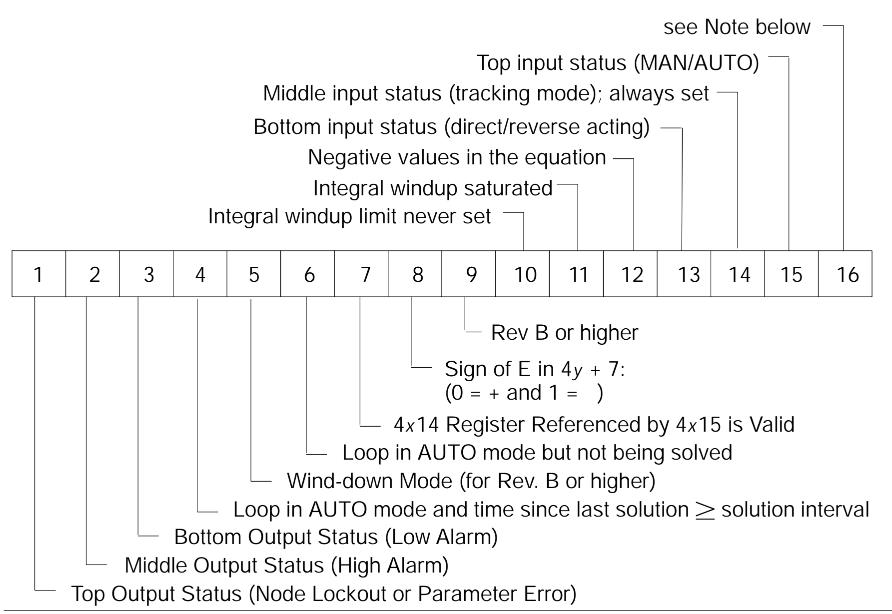
First implied	SP	You must specify the set point in engineering units; the value must be < value in the 11th implied register and > value in the 12th implied register
Second implied	M_V	Loaded by the block every time the loop is solved; it is clamped to a range of 0 ... 4095, making the output compatible with an analog output module; the manipulated variable register may be used for further CPU calculations such as cascaded loops
Third implied	High Alarm Limit	Load a value in this register to specify a high alarm for PV (at or above SP); enter the value in engineering units within the range specified in the 11th and 12th implied registers
Fourth implied	Low Alarm Limit	Load a value in this register to specify a low alarm for PV (at or below SP); enter the value in engineering units within the range specified in the 11th and 12th implied registers
Fifth implied	Proportional Band	Load this register with the desired proportional constant in the range 5 ... 500; the smaller the number, the larger the proportional contribution; a valid number is required in this register for PID2 to operate
Sixth implied	Reset Time Constant	Load this register to add integral action to the calculation; enter a value between 0000 ... 9999 to represent a range of 00.00 ... 99.99 repeats/min; the larger the number, the larger the integral contribution; a value > 9999 stops the PID2 calculation
Seventh implied	Rate Time Constant	Load this register to add derivative action to the calculation; enter a value between 0000 ... 9999 to represent a range of 00.00 ... 99.99 min; the larger the number, the larger the derivative contribution; a value > 9999 stops the PID2 calculation
Eighth implied	Bias	Load this register to add a bias to the output; the value must be between 000 .... 4095, and added directly to M_V, whether the integral term is enabled or not
Ninth implied	High Integral Windup Limit	Load this register with the upper limit of the output value (between 0 ... 4095) where the *anti-reset windup* takes effect; the updating of the integral sum is stopped if it goes above this value—this is normally 4095
10th implied	Low Integral Windup Limit	Load this register with the lower limit of the output value (between 0 ... 4095) where the anti-reset windup takes effect—this is normally 0
11th implied	High Engineering Range	Load this register with the highest value for which the measurement device is spanned—e.g., if a resistance temperature device ranges from 0 ... 500 degrees C, the high engineering range value is 500; the range must be given as a positive integer between 0001 ... 9999, corresponding to the raw analog input 4095
12th implied	Low Engineering Range	Load this register with the lowest value for which the measurement device is spanned; the range must be given as a positive integer between 0 ... 9998, and it must be less than the value in the 11th implied register; it corresponds to the raw analog input 0

13th implied	Raw Analog Measurement	The logic program loads this register with PV; the measurement must be scaled and linear in the range 0 ... 4095
14th implied	Pointer to Loop Counter Register	The value you load in this register points to the register that counts the number of loops solved in each scan; the entry is determined by discarding the most significant digit in the register where the controller will count the loops solved/scan—e.g., if the PLC does the count in register 41236, load 1236 into the 14th implied register; the same value must be loaded into the 14th implied register in every PID2 block in the logic program
15th implied	Maximum Number of Loops	Solved In a Scan: If the 14th implied register contains a non-zero value, you may load a value in this register to limit the number of loops to be solved in one scan
16th implied	Pointer To Reset Feedback Input:	The value you load in this register points to the holding register that contains the value of feedback (F); drop the 4 from the feedback register and enter the remaining four digits in this register; integration calculations depend on the F value being should F vary from 0 ... 4095
17th implied	Output Clamp—High	The value entered in this register determines the upper limit of M_V—this is normally 4095
18th implied	Output Clamp—Low	The value entered in this register determines the lower limit of M_V—this is normally 0
19th implied	Rate Gain Limit (RGL) Constant	The value entered in this register determines the effective degree of derivative filtering; the range is from 2 ... 30; the smaller the value, the more filtering takes place
20th implied	Pointer to Integral Preload	The value entered in this register points to the holding register containing the track input (T) value; drop the 4 from the tracking register and enter the remaining four digits in this register; the value in the T register is connected to the input of the integral lag whenever the auto bit and integral preload bit are both true

Middle Node Content

The 4*y* register entered in the middle node is the first of nine contiguous holding register used for PID2 calculations. You do not need to load anything into these registers:

Register	Name	Content
Displayed	Loop Status Register	Twelve of the 16 bits in this register are used to define loop status:



Note: Bit 16 is set after initial startup or installation of the loop. If you clear the bit, the following actions take place in one scan:

- The loop status register is reset
- The current value in the real-time clock is stored in the first implied register
- Values the the third ... fifth registers are cleared
- The value in the13th implied register x 8 is stored in the sixth implied register
- The seventh and eighth implied registers are cleared

Register	Name	Content
First implied	Error (E) Status Bits	This register displays PID2 error codes:

Code	Explanation	Check these Registers in the Top Node
0000	No errors, all validations OK	None
0001	Scaled SP above 9999	First implied
0002	High alarm above 9999	Third implied
0003	Low alarm above 9999	Fourth implied

0004	Proportional band below 5	Fifth implied
0005	Proportional band above 500	Fifth implied
0006	Reset above 99.99 r/min	Sixth implied
0007	Rate above 99.99 min	Seventh implied
0008	Bias above 4095	Eighth implied
0009	High integral limit above 4095	Ninth implied
0010	Low integral limit above 4095	10th implied
0011	High engineering unit (E.U.) scale above 9999	11th implied
0012	Low E.U. scale above 9999	12th implied
0013	High E.U. below low E.U.	11th and 12th implied
0014	Scaled SP above high E.U.	First and 11th implied
0015	Scaled SP below low E.U.	First and 12th implied
0016*	Maximum loops/scan > 9999	15th implied
0017	Reset feedback pointer out of range	16th implied
0018	High output clamp above 4095	17th implied
0019	Low output clamp above 4095	18th implied
0020	Low output clamp above high output clamp	17th and 18th implied
0021	RGL below 2	19th implied
0022	RGL above 30	19th implied
0023**	Track F pointer out of range	20th implied with middle input ON
0024**	Track F pointer is zero	20th implied with middle input ON
0025*	Node locked out (short of scan time)	None
NOTE: If lockout occurs often and the parameters are all valid, increase the maximum number of loops/scan. Lockout may also occur if the counting registers in use are not cleared as required.		
0026*	Loop counter pointer is zero	14th and 15th implied
0027	Loop counter pointer out of range	14th and 15th implied

* Activated by maximum loop feature—i.e., only if 4*x*15 is not zero.

** Activated only if the track feature is ON—i.e., the middle input of the PID2 block is receiving power while in AUTO mode.

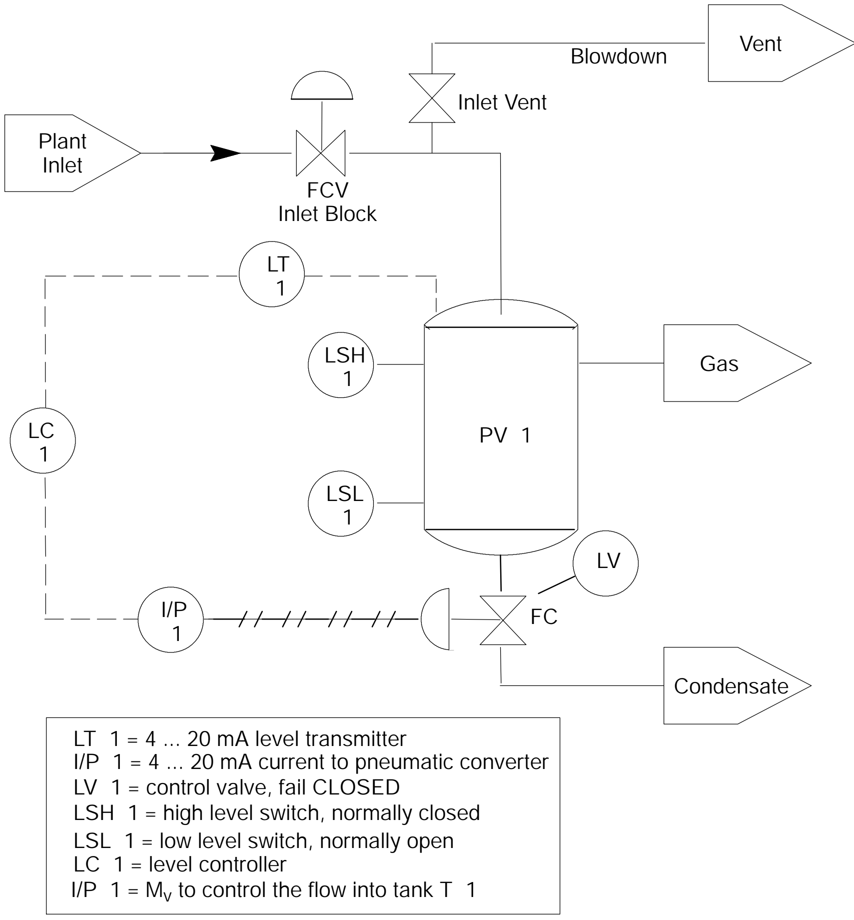
Register	Name	Content
Second implied	Loop Timer Register	This register stores the real-time clock reading on the system clock each time the loop is solved: the difference between the current clock value and the value stored in the register is the *elapsed* time; if elapsed time $\geq$ *solution interval* (10 times the value given in the bottom node of the PID2 block), then the loop should be solved in this scan
Third implied	For Internal Use	Integral (integer portion)
Fourth implied	For Internal Use	Integral—fraction 1 ($^1/_{3,000}$)
Fifth implied	For Internal Use	Integral—fraction 2 ($^1/_{600,000}$)
Sixth implied	P_V x 8 (Filtered)	This register stores the result of the filtered analog input (from register 4*x*14) multiplied by 8; this value is useful in derivative control operations
Seventh implied	Absolute Value of E	This register, which is updated after each loop solution, contains the absolute value of (SP PV); bit 8 in register 4*y* + 1 indicates the sign of E
Eighth implied	For Internal Use	Current solution interval

Bottom Node Content

The bottom node indicates that this is a PID2 function and contains a number ranging from 1 ... 255, indicating how often the function should be performed. The number represents a time value in tenths of a second—for example, the number 17 indicates that the PID function should be performed every 1.7 s.

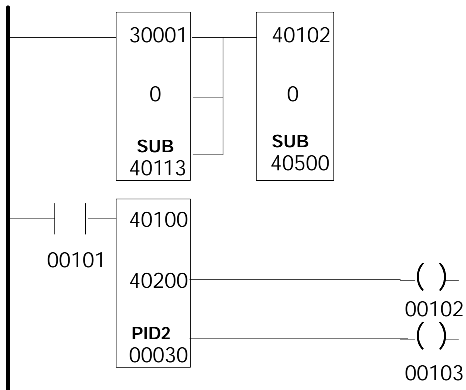
21.2.3 A PID2 Level Control Example

Here is a simplified P&I diagram for an inlet separator in a gas processing plant. There is a two-phase inlet stream—liquid and gas.



The liquid is dumped from the tank to maintain a constant level. The control objective is to maintain a constant level in the separator. The phases must be separated before processing; separation is the role of the inlet separator, PV 1. If the level controller, LC 1, fails to perform its job, the inlet separator could fill, causing liquids to get into the gas stream; this could severely damage devices such as gas compressors.

The level is controlled by device LC 1, a 984 controller connected to an analog input module; I/P 1 is connected to an analog output module. We can implement the control loop with the following 984 ladder logic:



The first SUB block is used to move the analog input from LT 1 to the PID2 analog input register, 40113. The second SUB block is used to move the PID2 output M_V to the traffic copped output I/P 1. Coil 00101 is used to change the loop from AUTO to MANUAL mode, if desired. For AUTO mode, it should be ON.

Specify the set point in mm for input scaling (E.U.). The full input range will be 0 ... 4000 mm (for 0 ... 4095 raw analog). Specify the register content of the top node in the PID2 block as follows:

Register	Content		Comments
	Numeric	Meaning	
40100		Scaled PV (mm)	PID2 writes this
40101	2000	Scaled SP (mm)	Set to 2000 mm (half full) initially
40102	0000	Loop output (0 ... 4095)	PID2 writes this; keep it set to 0 to be safe
40103	3500	Alarm High Set Point (mm)	If the level rises above 3500 mm, coil 00102 goes ON
40104	1000	Alarm Low Set Point (mm)	If the level drops below 1000 mm, coil 00103 goes ON
40105	0100	PB (%)	The actual value depends on the process dynamics
40106	0500	Integral constant (5.00 repeats/min)	The actual value depends on the process dynamics
40107	0000	Rate time constant (per min)	Setting this to 0 turns off the derivative mode
40108	0000	Bias (0 ... 4095)	This is set to 0, since we have an integral term
40109	4095	High windup limit (0 ... 4095)	Normally set to the maximum
40110	0000	Low windup limit (0 ... 4095)	Normally set to the minimum

Register	Content		Comments
	Numeric	Meaning	
40111	4000	High engineering range (mm)	The scaled value of the process variable when the raw input is at 4095
40112	0000	Low engineering range (mm)	The scaled value of the process variable when the raw input is at 0
40113		Raw analog measure (0 ... 4095)	A copy of the input from the analog input module register (30001) copied by the first SUB
40114	0000	Offset to loop counter register	Zero disables this feature. Normally, this is not used
40115	0000	Max loops solved per scan	See register 40114
40116	0102	Pointer to reset feedback	If you leave this as zero, the PID2 function automatically supplies a pointer to the loop output register. If the actual output (40500) could be changed from the value supplied by PID2, then this register should be set to 500 (40500) to calculate the integral properly
40117	4095	Output clamp high (0 ... 4095)	Normally set to maximum
40118	0000	Output clamp low (0 ... 4095)	Normally set to minimum
40119	0015	Rate Gain Limit Constant (2 ... 30)	Normally set to about 15. The actual value depends on how noisy the input signal is. Since we are not using derivative mode, this has no effect on PID2
40120	0000	Pointer to track input	Used only if the PRELOAD feature is used. If the PRELOAD is not used, this is normally zero

The values in the registers in the 40200 destination block are all set by the PID2 block.

21.3 PCFL

The PCFL instruction gives you access to a library of process control functions utilizing analog values. PCFL operations fall into three major categories:

- Advanced calculations
- Signal processing
- Regulatory control

A PCFL function is selected from a list of alphabetical indicators in a pulldown menu in the panel software, and the indicator is displayed in the top node of the instruction (see the table on pages 416 and 417 for a list of function indicators and descriptions).

PCFL uses the same FP library as EMTH. If the PLC that you are using for PCFL does not have the onboard 80x87 math coprocessor chip, calculations take a comparatively long time to execute. PLCs with the math coprocessor can solve PCFL calculations ten times faster than PLCs without the chip. Speed, however, should not be an issue for most traditional process control applications where solution times are measured in seconds, not milliseconds.

21.3.1 Characteristics

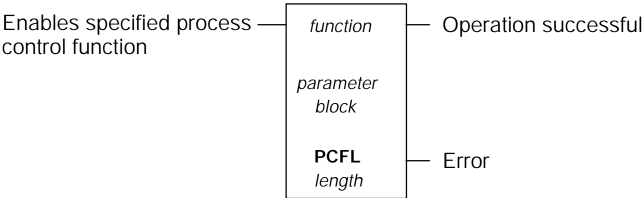
Size
Three nodes high

PLC Compatibility
- Standard in the E984-685 and E984-785 Slot Mount PLCs and in the Quantum Automation Series PLCs. Some operations are not available on all of the above PLC types; for details see the table beginning on page 416
- Not available in other PLC types

Opcode
7B hex

21.3.2 Representation

Block Structure



Input

PCFL has one control input (to the top node), which enables the specifies process control operation when it is ON.

Outputs

PCFL produces one of two possible outputs (from the top or bottom node). Power is passed to the output from the top node if the process control operation completes successfully. Power is passed to the output from the bottom node if an error is encountered in the process control operation.

Top Node Content

An indicator for the selected PCFL library function is specified in the top node:

Operation	Indicator	Description	PLCs Supported
Advanced Calculations	AVER	Average weighted inputs	E785, Q785, Quantum, AT, and VME
	CALC	Calculate preset formula	E785, Q785, Quantum, AT, and VME
	EQN	Formatted equation calculator	E785, Q785, Quantum, AT, and VME
Signal Processing	ALARM	Central alarm handler for a P_V input	E785, Q785, Quantum, AT, and VME
	AIN	Convert inputs to scaled engineering units	E785, Q785, Quantum, AT, and VME
	AOUT	Convert outputs to values in the 0 ... 4095 range	E785, Q785, Quantum, AT, and VME
	DELAY	Time delay queue*	E785, Q785, Quantum, AT, and VME
	LKUP	Look-up table	E785, Q785, Quantum, AT, and VME
	INTEG	Integrate input at specified interval*	E785, Q785, Quantum, AT, and VME

	LLAG	First-order lead/lag filter*	E785, Q785, Quantum, AT, and VME
	LIMIT	Limiter for the P_V (low/low, low, high, high/high)	E785, Q785, Quantum, AT, and VME
	LIMV	Velocity limiter for changes in the P_V (low, high) *	E785, Q785, Quantum, AT, and VME
	MODE	Put input in auto or manual mode	E785, Q785, Quantum, AT, and VME
	RAMP	Ramp to set point at a constant rate*	E785, Q785, Quantum, AT, and VME
	RMPLN	Logarithmic ramp to set point (~$^2/_3$ closer to set point for each time constant)*	E785, Q785, Quantum, AT, and VME
	RATE	Derivative rate calculation over a specified time*	E785, Q785, Quantum, AT, and VME
	SEL	High/low/average input selection	E785, Q785, Quantum, AT, and VME
Regulatory Control	KPID	Comprehensive ISA non-interacting proportional-integral-derivative (PID)*	Q785, Quantum, and VME
	ONOFF	Specifies ON/OFF values for deadband	E785, Q785, Quantum, AT, and VME
	PID	PID algorithms*	E785, Q785, Quantum, AT, and VME
	PI	ISA non-interacting PI (with halt/manual/auto operation features)*	Quantum and VME
	RATIO	Four-station ratio controller	Quantum and VME
	TOTAL	Totalizer for metering flow*	Quantum and VME

* Time-dependent operations.

Middle Node Content

The 4*x* register entered in the middle node is the first in a block of contiguous holding register where the *parameters* for the specified PCFL operation are stored. The ways that the various PCFL operations implement the *parameter block* are described on the following pages.

Bottom Node Content

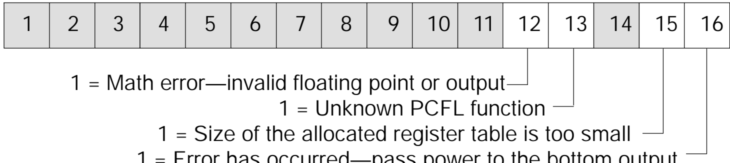
The integer value entered in the bottom node specifies the *length* —i.e., the number of registers—of the PCFL *parameter block* . The maximum allowable *length* will vary depending on the function you specify.

21.3.3 Input and Output Flags

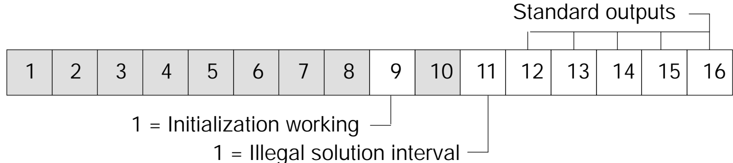
Within the parameter block of each PCFL function are two registers used for input and output status.

Output Flags

In all PCFL functions, bits 12 ... 16 of the output status register define the following standard output flags:

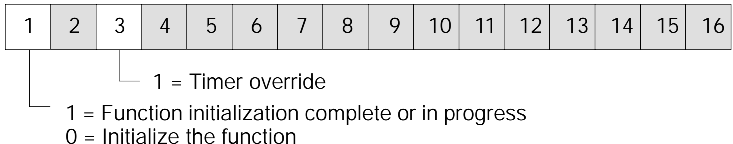


For time-dependent PCFL functions, bits 9 and 11 are also used as follows:



Input Flags

In all PCFL functions, bits 1 and 3 of the input status register define the following standard input flags:



21.4 PCFL Advanced Calculations

Advanced calculations are used for general mathematical purposes and are not limited to process control applications. With advanced calculations, you can create custom signal processing algorithms, derive states of the controlled process, derive statistical measures of the process, etc.

Simple math routines have already been offered in the EMTH instruction (see Chapter 7). The calculation capability included in PCFL is a textual equation calculator for writing custom equations instead of programming a series of math operations one by one.

21.4.1 AVER

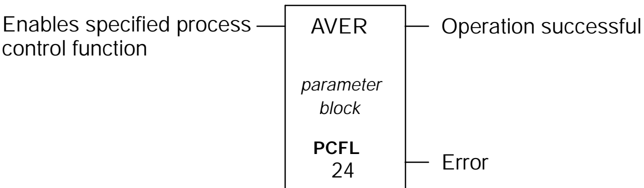
The AVER function calculates the average of up to four weighted inputs, via the following formula:

$$result = \frac{(k + (w_1 \times In_1) + (w_2 \times In_2) + (w_3 \times In_3) + (w_4 \times In_4))}{1 + w_1 + w_2 + w_3 + w_4}$$

where

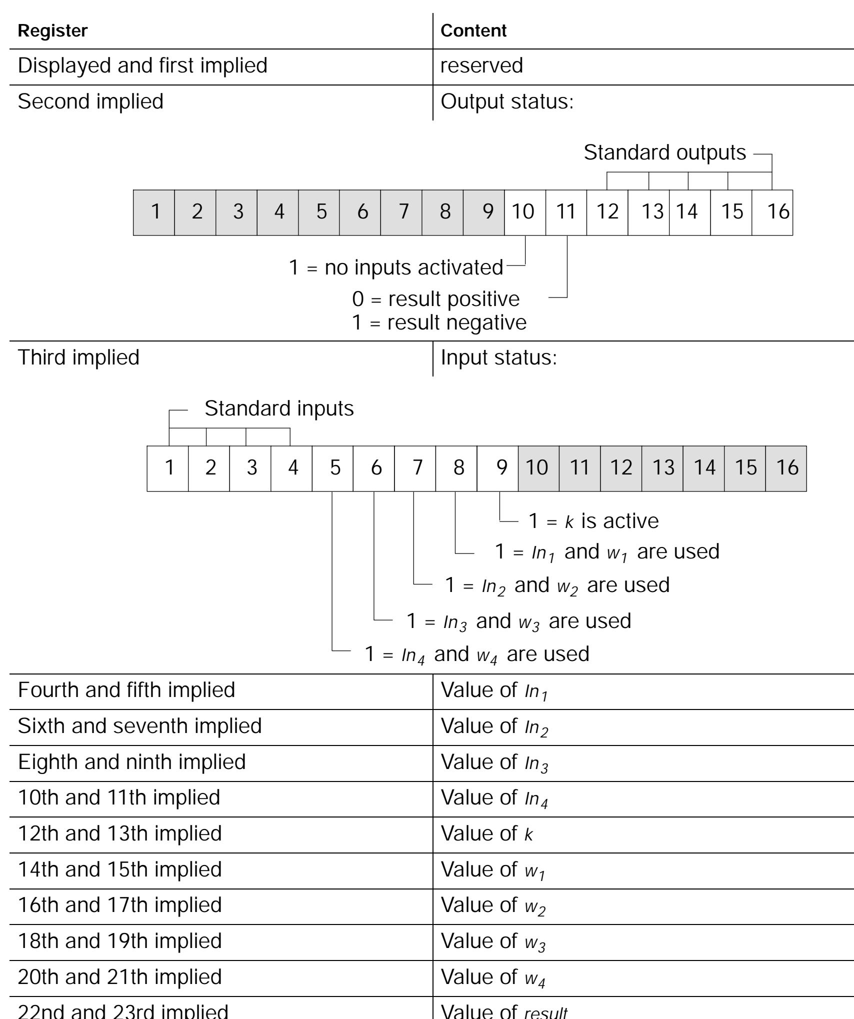
w_1 ... w_4 are the weights
In_1 ... In_4 are the inputs
k is a constant

Block Structure



Parameter Block Assignment

The *length* of the AVER *parameter block* is 24 registers:



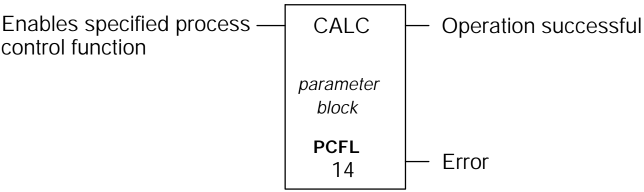
Register	Content
Displayed and first implied	reserved
Second implied	Output status:
	Bits 1–16; bits 12–16: Standard outputs; bit 10: 1 = no inputs activated; bit 11: 0 = result positive, 1 = result negative
Third implied	Input status:
	Bits 1–16; bits 1–4: Standard inputs; bit 9: 1 = k is active; bit 8: 1 = In_1 and w_1 are used; bit 7: 1 = In_2 and w_2 are used; bit 6: 1 = In_3 and w_3 are used; bit 5: 1 = In_4 and w_4 are used
Fourth and fifth implied	Value of In_1
Sixth and seventh implied	Value of In_2
Eighth and ninth implied	Value of In_3
10th and 11th implied	Value of In_4
12th and 13th implied	Value of k
14th and 15th implied	Value of w_1
16th and 17th implied	Value of w_2
18th and 19th implied	Value of w_3
20th and 21th implied	Value of w_4
22nd and 23rd implied	Value of *result*

A weight can be used only when its corresponding input is enabled—e.g., the 20th and 21st implied registers (which contain the value of w_4) can be used only when the 10th and 11th implied registers (which contain In_4) are enabled. The I in the denominator is used only when the constant is enabled.

21.4.2 CALC

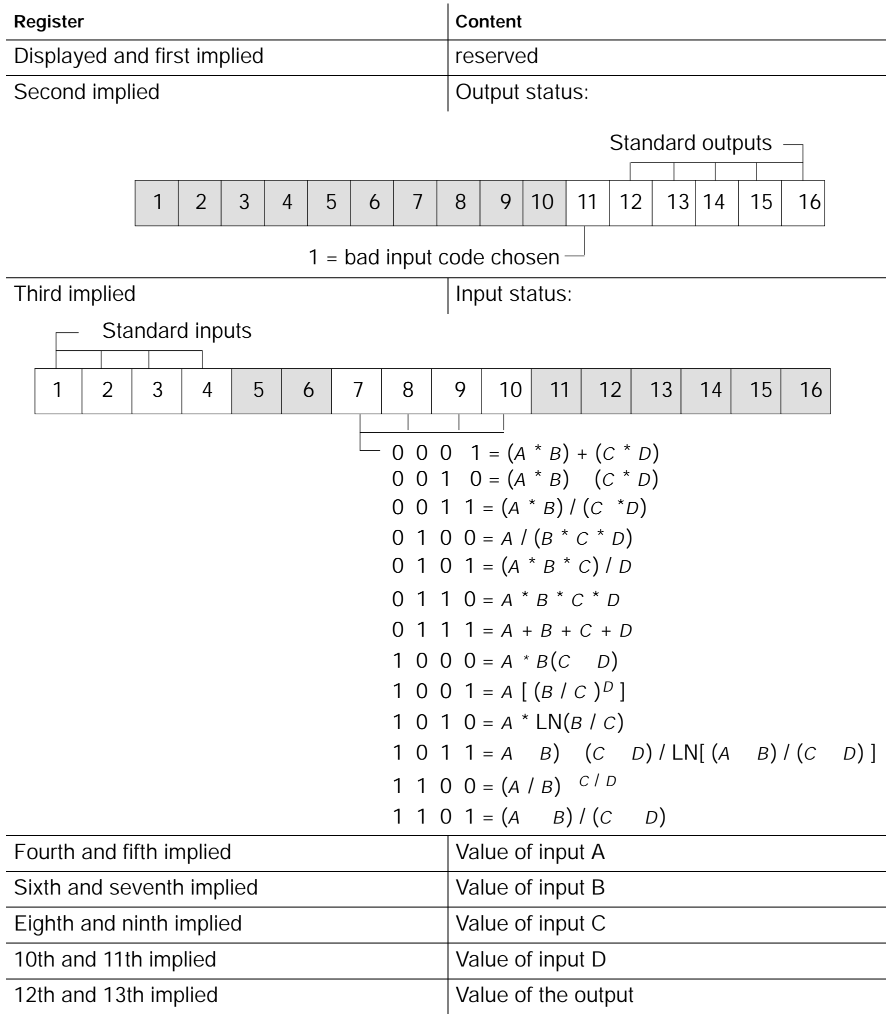
The CALC function calculates a preset formula with up to four inputs, each characterized in a separate register of the *parameter block*.

Block Structure



Parameter Block Assignment

The *length* of the CALC *parameter block* is 14 registers, with the following assignments:



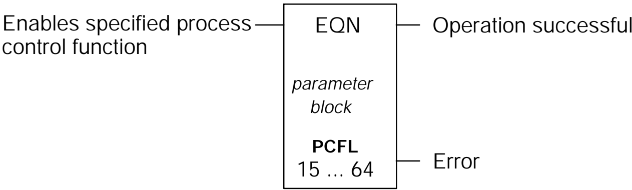
Register	Content
Displayed and first implied	reserved
Second implied	Output status: Bits 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 Standard outputs (bits 12–16) 1 = bad input code chosen (bit 11)
Third implied	Input status: Bits 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 Standard inputs (bits 1–4) Bits 7–10: 0 0 0 1 = (A * B) + (C * D) 0 0 1 0 = (A * B) (C * D) 0 0 1 1 = (A * B) / (C *D) 0 1 0 0 = A / (B * C * D) 0 1 0 1 = (A * B * C) / D 0 1 1 0 = A * B * C * D 0 1 1 1 = A + B + C + D 1 0 0 0 = A * B(C D) 1 0 0 1 = A [(B / C)^{D}] 1 0 1 0 = A * LN(B / C) 1 0 1 1 = A B) (C D) / LN[(A B) / (C D)] 1 1 0 0 = (A / B) $^{C / D}$ 1 1 0 1 = (A B) / (C D)
Fourth and fifth implied	Value of input A
Sixth and seventh implied	Value of input B
Eighth and ninth implied	Value of input C
10th and 11th implied	Value of input D
12th and 13th implied	Value of the output

21.4.3 EQN

The EQN function is a formatted equation calculator. You must define the equation in the *parameter block* with various codes that specify operators, input selection, and inputs.

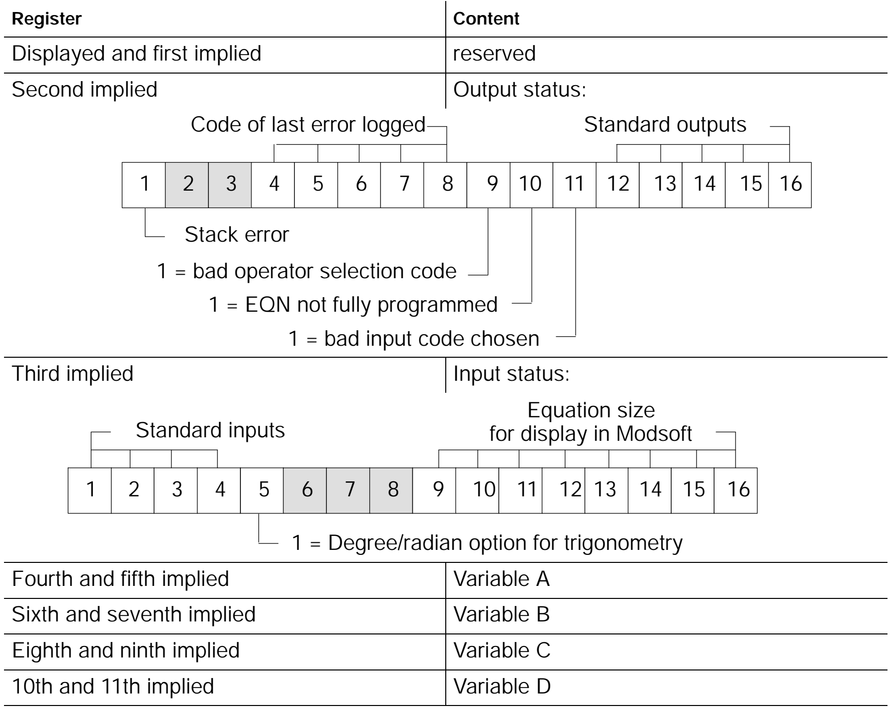
EQN is used for equations that have four or fewer variables but do not fit into the CALC format. It complements the CALC function by letting you input an equation with floating point and integer inputs as well as operators.

Block Structure



Parameter Block Assignment

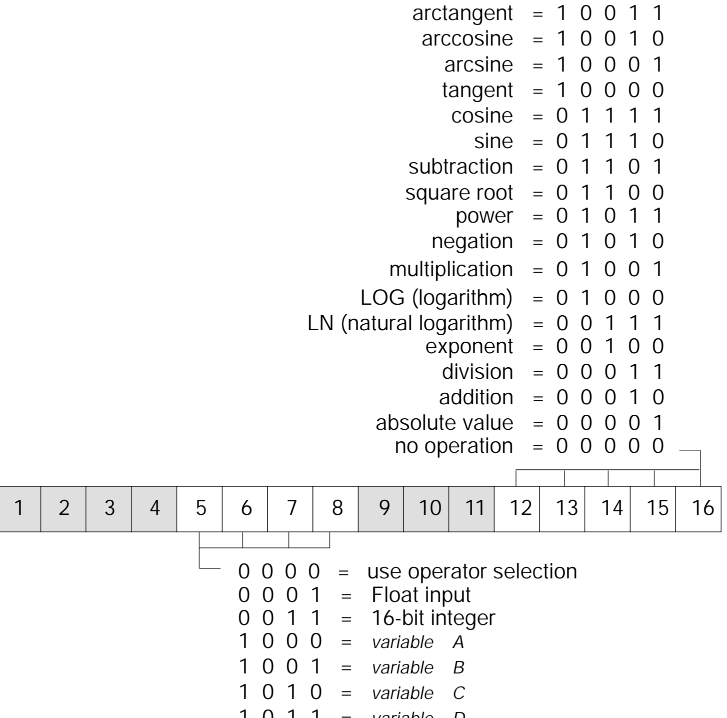
The *length* of the EQN *parameter block* can be as high as 64 registers:



Register	Content
Displayed and first implied	reserved
Second implied	Output status:
Third implied	Input status:
Fourth and fifth implied	Variable A
Sixth and seventh implied	Variable B
Eighth and ninth implied	Variable C
10th and 11th implied	Variable D

12th and 13th implied	Output
14th implied	First formula code
15th implied	Second possible formula code
. . .	. . .
63rd implied	Last possible formula code

Each formula code in the EQN function defines either an input selection code or an operator selection code. Bits 5 ... 8 in the appropriate register can be used to define an input selection, and bits 12 ... 16 can be used to define an operator selection:



21.5 PCFL Signal Processing Functions

Signal processing functions are used to manipulate process and derived process signals. They can do this in a variety of ways; they linearize, filter, delay, and otherwise modify a signal. This category would include functions such as an Analog Input/Output, Limiters, Lead/Lag, and Ramp generators.

21.5.1 ALARM

The ALARM function gives you a central block for alarm handling where you can set high (H), low (L), high high (HH), and low low (LL) limits on a process variable. ALARM lets you specify:

- A choice of normal or deviation operating mode
- Whether to use H/L or both H/L and HH/LL limits
- Whether or not to use deadband (DB) around the limits

Normal Operating Mode

In normal mode, ALARM operates directly on the input. Normal is the default condition.

Deviation Operating Mode

In deviation mode, ALARM operates on the change between the current input and the last input.

Deadband

When enabled, the DB option is incorporated into the HH/H/LL/L limits. These calculated limits are inclusive of the more extreme range—e.g., if the input has been in the high range, the output remains high and does not transition when the input hits the calculated H limit.

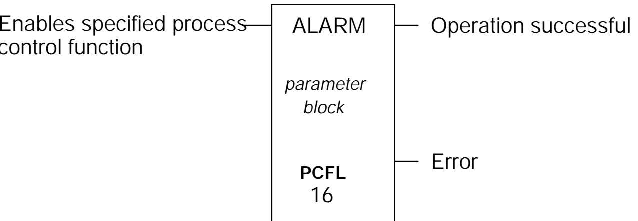
The DB option is recognized only in normal mode. An error is logged in the status output register if DB is specified in deviation mode.

Operations

A flag is set when the input or deviation equals or crosses the corresponding limit. If the DB option is used, the HH, H, LL, L limits are adjusted internally for crossed-limit checking and hysteresis.

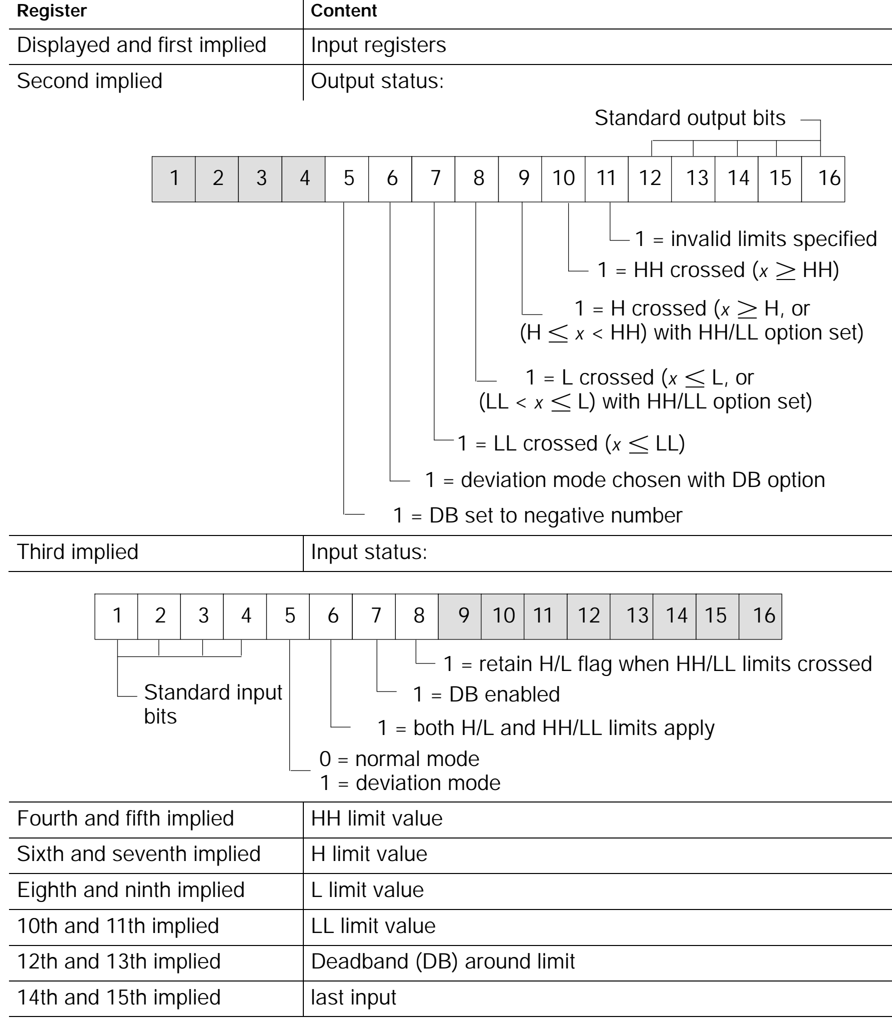
ALARM automatically tracks the last input—even when you specify normal mode—to facilitate a smooth transition to deviation mode.

Block Structure



Parameter Block Assignment

The *length* of the ALARM *parameter block* is 16 registers:



Register	Content
Displayed and first implied	Input registers
Second implied	Output status:
Third implied	Input status:
Fourth and fifth implied	HH limit value
Sixth and seventh implied	H limit value
Eighth and ninth implied	L limit value
10th and 11th implied	LL limit value
12th and 13th implied	Deadband (DB) around limit
14th and 15th implied	last input

21.5.2 AIN

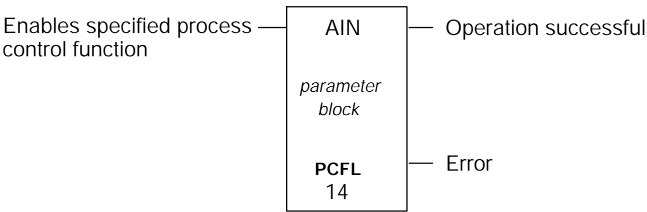
The AIN function scales the raw input produced by analog input modules to engineering values that can be used in the subsequent calculations. Three scaling options are available:

- ☐ Auto input scaling
- ☐ Manual input scaling
- ☐ Implementing process square root on the input to linearize the signal before scaling

AIN supports the following range resolutions:

		Range		
Device Type	**Resolution**	**Valid**	**Under**	**Over**
984 PLCs	4096 Normal	0 ... 4,096	0	4,096
	Elevated	4,096 ... 8,192	4,096	8,192
	8192 Normal	0 ... 8,192	0	8,192
	Offset	0 ... 6,000	0xC000	0x8000
	Unipolar	0 ... 7,500	0xC000	0x8000
	Bipolar	0 ... 15,000	0xC000	0x8000
	Scaled decimal	0 ... 10,000	0xC000	0x8000
Quantum Engineering Ranges	±10 V	768 ... 64,768	767	64,769
	±5 V	16,768 ... 48,768	16,767	48,769
	0 ... 10 V	0 ... 64,000	0	64,001
	0 ... 5 V	0 ... 32,000	0	32,001
	1 ... 5 V	6,400 ... 32,000	6,399	32,001
Quantum Thermocouple	TC degrees	454 ... +3,308		
	TC 0.1 degrees	4,540 ... +32,767		
	TC Raw Units	0 ... 65,535		
Quantum Voltmeter	±10 V	10,000 ... +10,000	10,001	+10,001
	±5 V	5,000 ... +5,000	5,001	+5,001
	0 ... 10 V	0 ... 10,000	0	10,001
	0 ... 5 V	0 ... 5,000	0	5,001
	1 ... 5 V	1,000 ... 5,000	999	5,001

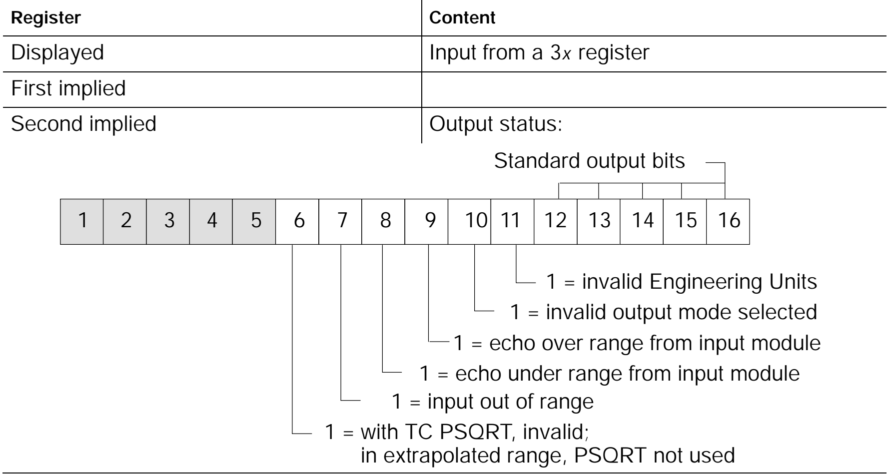
Block Structure



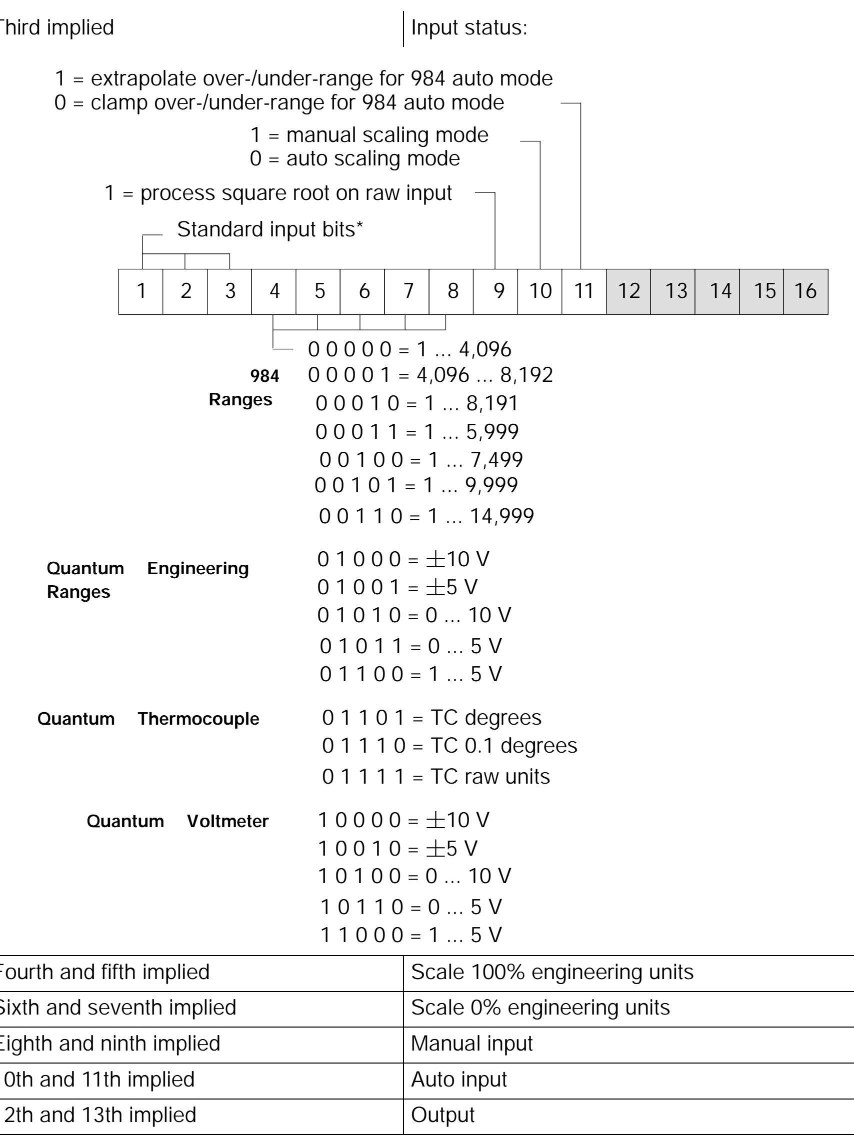
Parameter Block Assignment

The *length* of the AIN *parameter block* is 14 registers:

Register	Content
Displayed	Input from a 3*x* register
First implied	
Second implied	Output status:



Third implied	Input status:



Fourth and fifth implied	Scale 100% engineering units
Sixth and seventh implied	Scale 0% engineering units
Eighth and ninth implied	Manual input
10th and 11th implied	Auto input
12th and 13th implied	Output

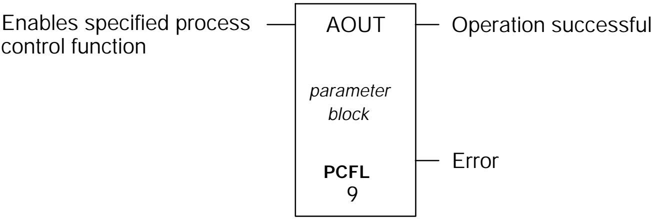
* Bit 4 in the third implied register is nonstandard use.

21.5.3 AOUT

The AOUT function is an interface for calculated signals for output modules. It converts the signal to a value in the range 0 ... 4,096, using the formula:

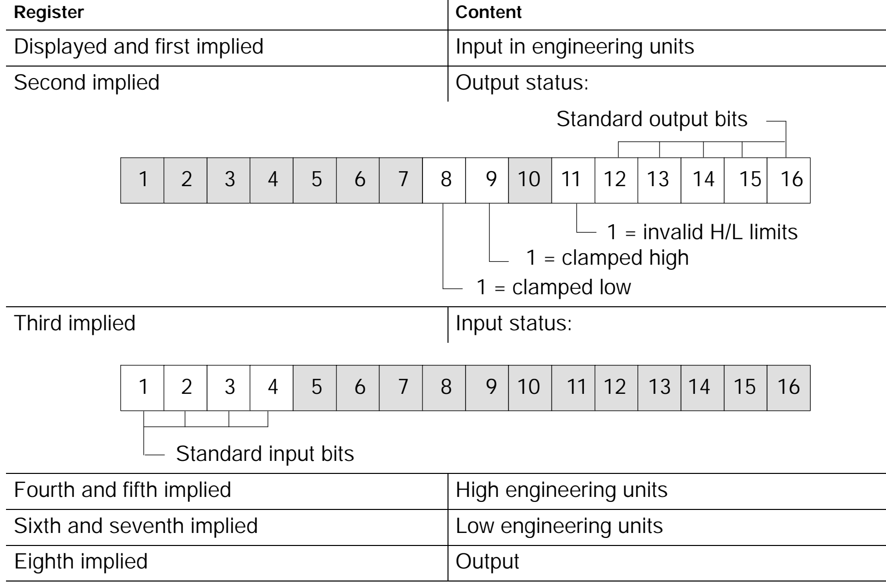
$$\text{output} = \frac{\textit{scale} \times (\textit{input} \quad \textit{low engineering unit})}{(\textit{high engineering unit} \quad \textit{low engineering unit})}$$

Block Structure



Parameter Block Assignment

The *length* of the AOUT *parameter block* is nine registers:



Register	Content
Displayed and first implied	Input in engineering units
Second implied	Output status: Standard output bits (12–16); bit 11: 1 = invalid H/L limits; bit 9: 1 = clamped high; bit 8: 1 = clamped low
Third implied	Input status: Standard input bits (1–4)
Fourth and fifth implied	High engineering units
Sixth and seventh implied	Low engineering units
Eighth implied	Output

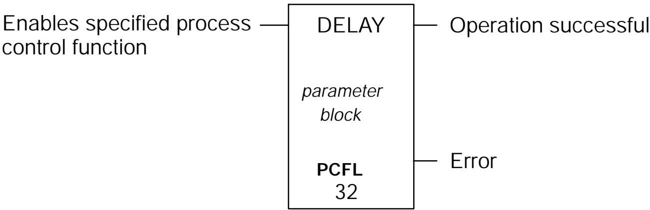
21.5.4 DELA Y

The DELAY function can be used to build a series of readings for time-delay compensation in the logic. Up to 10 sampling instances can be used to delay an input.

All values are carried along in registers, where register *x*[0] contains the current sampled input. The 10th delay period does not need to be stored. When the 10th instance in the sequence takes place, the value in register *x*[9] can be moved directly to the output.

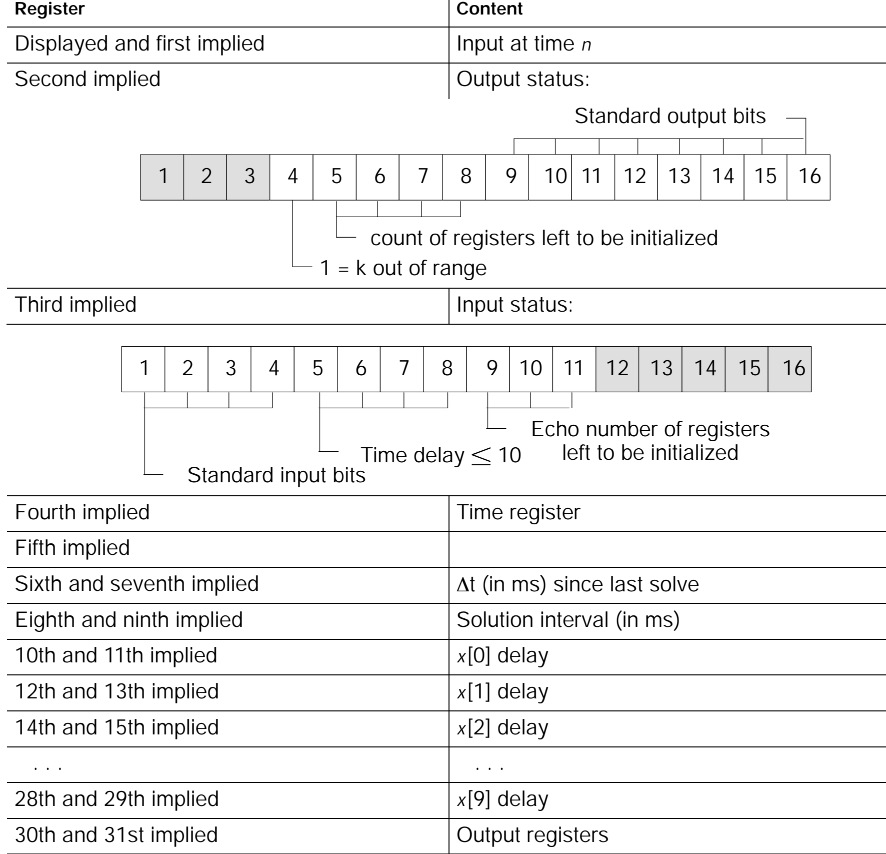
A DXDONE message is returned when the calculation is complete. The function can be reset by toggling the first-scan bit.

Block Structure



Parameter Block Assignment

The *length* of the DELAY *parameter block* is 32 registers:



Register	Content
Displayed and first implied	Input at time *n*
Second implied	Output status:
Third implied	Input status:
Fourth implied	Time register
Fifth implied	
Sixth and seventh implied	Δt (in ms) since last solve
Eighth and ninth implied	Solution interval (in ms)
10th and 11th implied	*x*[0] delay
12th and 13th implied	*x*[1] delay
14th and 15th implied	*x*[2] delay
. . .	. . .
28th and 29th implied	*x*[9] delay
30th and 31st implied	Output registers

21.5.5 LKUP

The LKUP function establishes a look-up table using a linear algorithm to interpolate between points. LKUP can handle variable point intervals and variable numbers of points.

If the input (x) is outside the specified range of points, the output (y) is clamped to the corresponding output $y0$ or yn. If the specified *parameter block length* is too small or if the number of points is out of range, the function does not check the xn because the information from that pointer is invalid.

Points to be interpolated are determined by a binary search algorithm starting near the center of x data. The search is valid for $x1 \leq x < xn$. The variable x may occur multiple times with the same value—the value chosen from the look-up table is the first instance found. For example, if the table is:

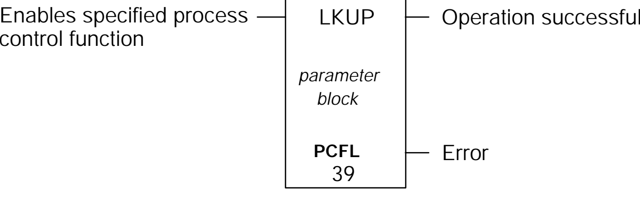
x	*y*
10.0	1.0
20.0	2.0
30.0	3.0
30.0	3.5
40.0	4.0

then an input of 30.0 finds the first instance of 30.0 and assigns 3.0 as the output. An input of 31.0 would assign the value 3.55 as the output.

No sorting is done on the contents of the look-up table. Independent variable table values should be entered in ascending order to prevent unreachable gaps in the table.

The function returns a DXDONE message when the operation is complete.

Block Structure



Parameter Block Assignment

The *length* of the LKUP *parameter block* is 39 registers:

Register	Content
Displayed and first implied	Input
Second implied	Output status:

Output status bits 1–16: bits 12–16 = Standard output bits; bit 10: 1 = input clamped—i.e., out of table's range; bit 11: 1 = invalid number of points

Register	Content
Third implied	Input status:

Input status bits 1–16: bits 1–4 = Standard input bits

Register	Content
Fourth implied	Number of point pairs
Fifth and sixth implied	Point $x1$
Seventh and eighth implied	Point $y1$
Ninth and tenth implied	Point $x2$
11th and 12th implied	Point $y2$
. . .	. . .
33rd and 34th implied	Point $x8$
35th and 36th implied	Point $y8$
37th and 38th implied	Output

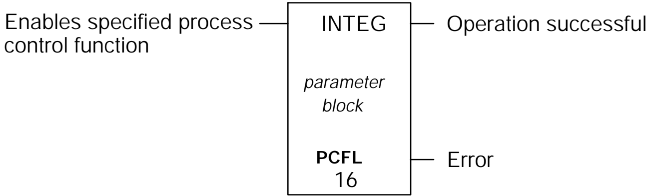
21.5.6 INTEG

The INTEG function is used to integrate over a specified time interval. No protection against integral wind-up is provided in this function. INTEG is time-dependent—e.g., if you are integrating at an input value of 1/sec, it matters whether it operates over one second (in which case the result is 1) or over one minute (in which case the result is 60).

You can set flags to either initialize or restart the function after an undetermined down-time, and you can reset the integral sum if you wish. If you set the *initialize* flag, you must specify a reset value (zero or the last output in case of power failure), and calculations will be skipped for one sample.

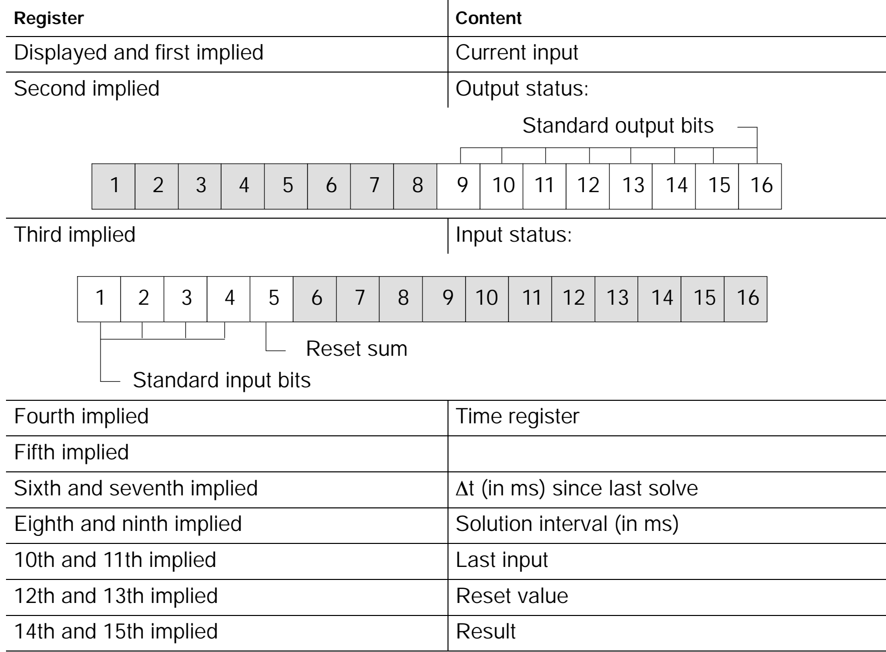
The function returns a DXDONE message when the operation is complete.

Block Structure



Parameter Block Assignment

The *length* of the INTEG *parameter block* is 16 registers:



Register	Content
Displayed and first implied	Current input
Second implied	Output status: Standard output bits 9–16
Third implied	Input status: Standard input bits 1–4; Reset sum bit 5
Fourth implied	Time register
Fifth implied	
Sixth and seventh implied	Δt (in ms) since last solve
Eighth and ninth implied	Solution interval (in ms)
10th and 11th implied	Last input
12th and 13th implied	Reset value
14th and 15th implied	Result

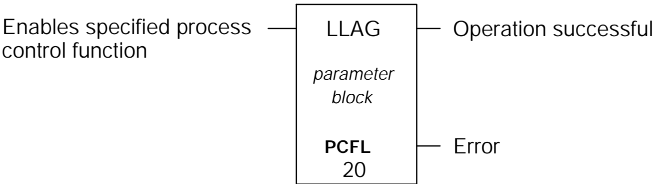
21.5.7 LLAG

The LLAG function provides dynamic compensation for a known disturbance. It usually appears in a feed-forward algorithm or as a dynamic filter. LLAG passes the input through a filter comprising a lead term (a numerator) and a lag term (a denominator) in the frequency domain, then multiplies it by a gain. Lead, lag, gain, and solution interval must be user-specified.

For best results, use lead and lag terms that are $\geq 4 * \Delta t$. This will ensure sufficient granularity in the output response.

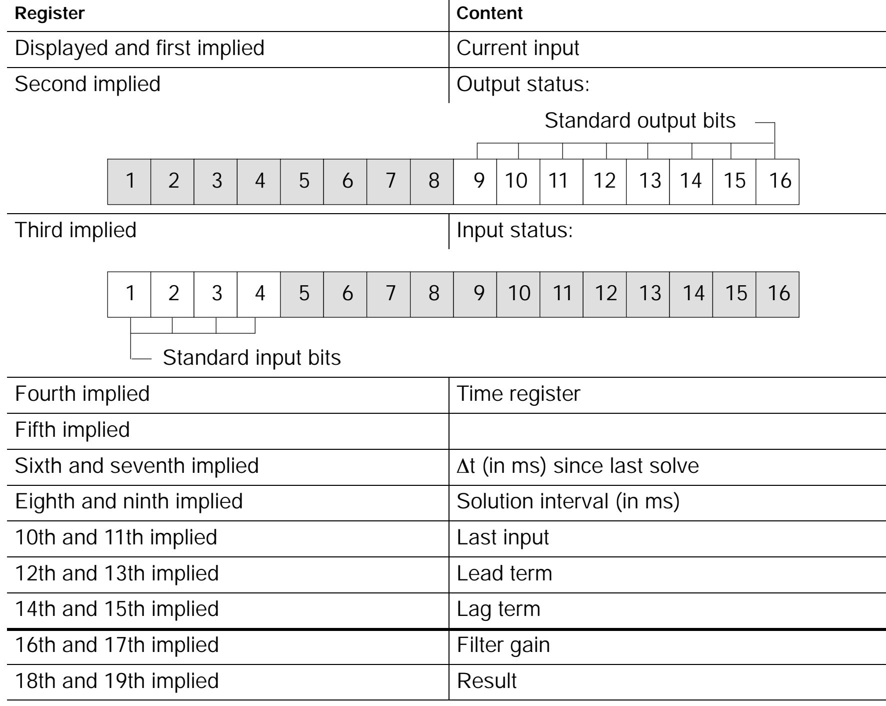
LLAG returns a DXDONE message when the operation completes.

Block Structure



Parameter Block Assignment

The *length* of the LLAG *parameter block* is 20 registers:



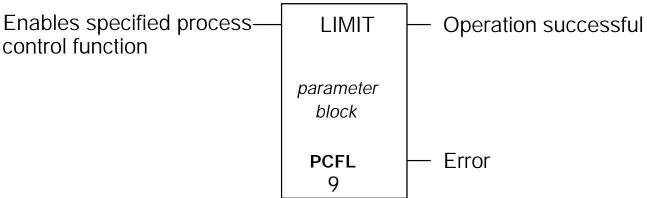
Register	Content
Displayed and first implied	Current input
Second implied	Output status:
Third implied	Input status:
Fourth implied	Time register
Fifth implied	
Sixth and seventh implied	Δt (in ms) since last solve
Eighth and ninth implied	Solution interval (in ms)
10th and 11th implied	Last input
12th and 13th implied	Lead term
14th and 15th implied	Lag term
16th and 17th implied	Filter gain
18th and 19th implied	Result

21.5.8 LIMIT

The LIMIT function limits the input to a range between a specified high and low value. If the high or low limit is reached, the function sets an H or L flag and clamps the output.

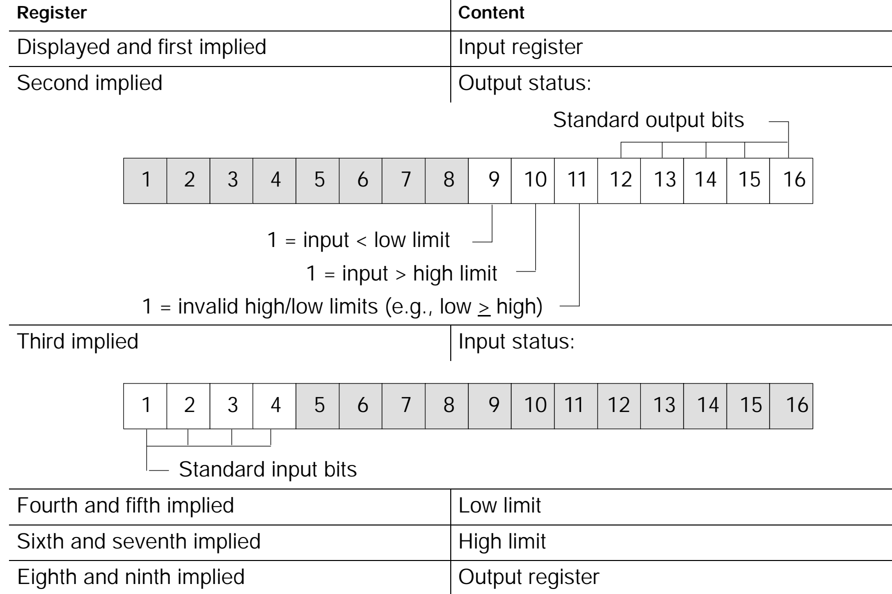
LIMIT returns a DXDONE message when the operation is complete.

Block Structure



Parameter Block Assignment

The *length* of the LIMIT *parameter block* is nine registers:



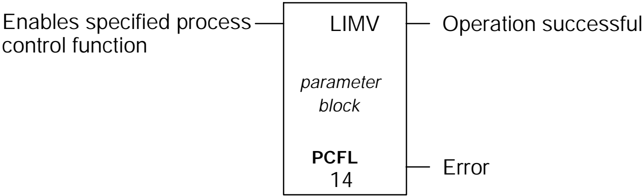
Register	Content
Displayed and first implied	Input register
Second implied	Output status:
Third implied	Input status:
Fourth and fifth implied	Low limit
Sixth and seventh implied	High limit
Eighth and ninth implied	Output register

21.5.9 LIMV

The LIMV function limits the velocity of change in the input variable between a specified high and low value. If the high or low limit is reached, the function sets an H or L flag and clamps the output.

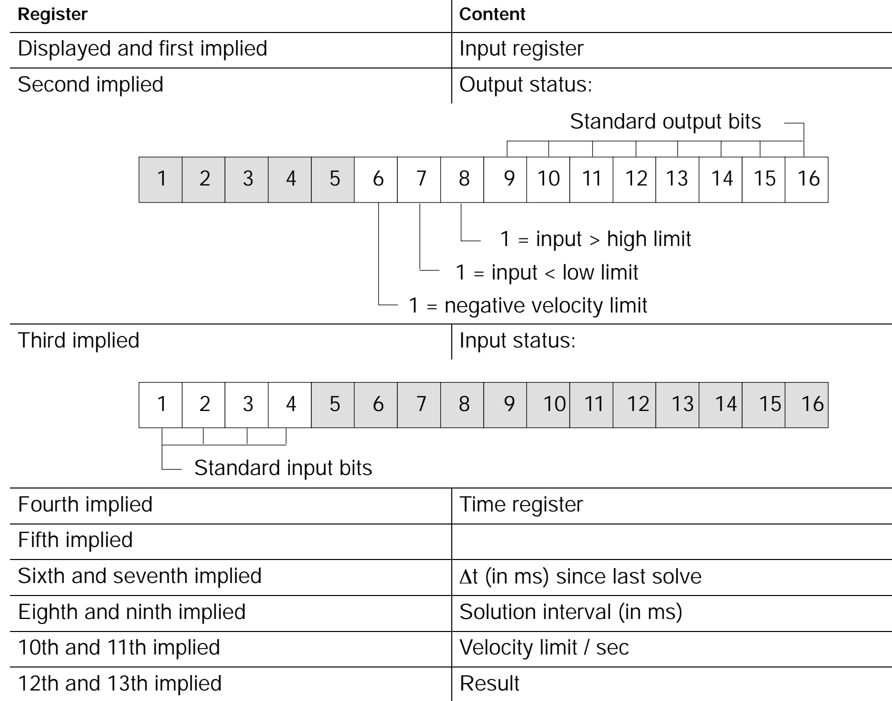
LIMV returns a DXDONE message when the operation is complete.

Block Structure



Parameter Block Assignment

The *length* of the LIMV *parameter block* is 14 registers:



Register	Content
Displayed and first implied	Input register
Second implied	Output status: Standard output bits 9–16; bit 8: 1 = input > high limit; bit 7: 1 = input < low limit; bit 6: 1 = negative velocity limit
Third implied	Input status: Standard input bits 1–4
Fourth implied	Time register
Fifth implied	
Sixth and seventh implied	Δt (in ms) since last solve
Eighth and ninth implied	Solution interval (in ms)
10th and 11th implied	Velocity limit / sec
12th and 13th implied	Result

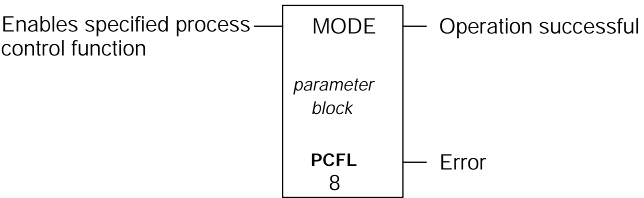
21.5.10 MODE

The MODE function sets up a manual or automatic station for enabling and disabling data transfers to the next block. The function acts like a BLKM instruction (see page 192), moving a value to the output register.

In auto mode, the input is copied to the output. In manual mode, the output is overwritten by a user entry.

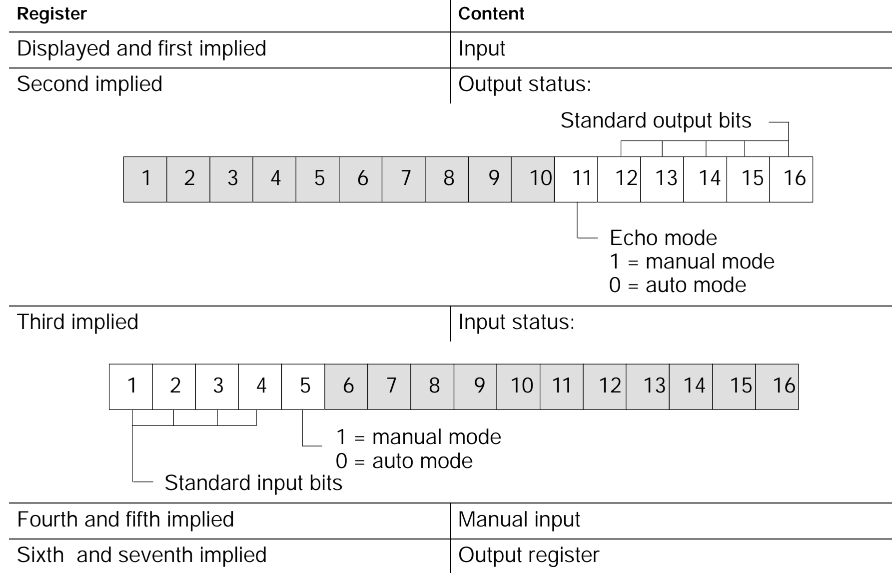
MODE returns a DXDONE message when the operation completes.

Block Structure



Parameter Block Assignment

The *length* of the MODE *parameter block* is eight registers:



Register	Content
Displayed and first implied	Input
Second implied	Output status:
Third implied	Input status:
Fourth and fifth implied	Manual input
Sixth and seventh implied	Output register

21.5.1 1 RAMP

The RAMP function allows you to ramp up linearly to a target set point at a specified approach rate. You need to specify:

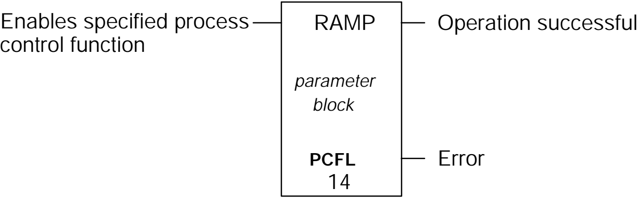
- The target set point, in the same units as the contents of the input register are specified
- The sampling rate
- A positive rate toward the target set point—negative rates are illegal

The direction of the ramp depends on the relationship between the target set point and the input—i.e., if x < SP, the ramp is up; if x > SP, the ramp is down.

You may use a flag to initialize after an undetermined down-time. The function will store a new sample, then wait for one cycle to collect the second sample. Calculations will be skipped for one cycle and the output will be left as is, after which the ramp will resume.

RAMP terminates when the entire ramping operation is complete (over multiple scans) and returns a DXDONE message.

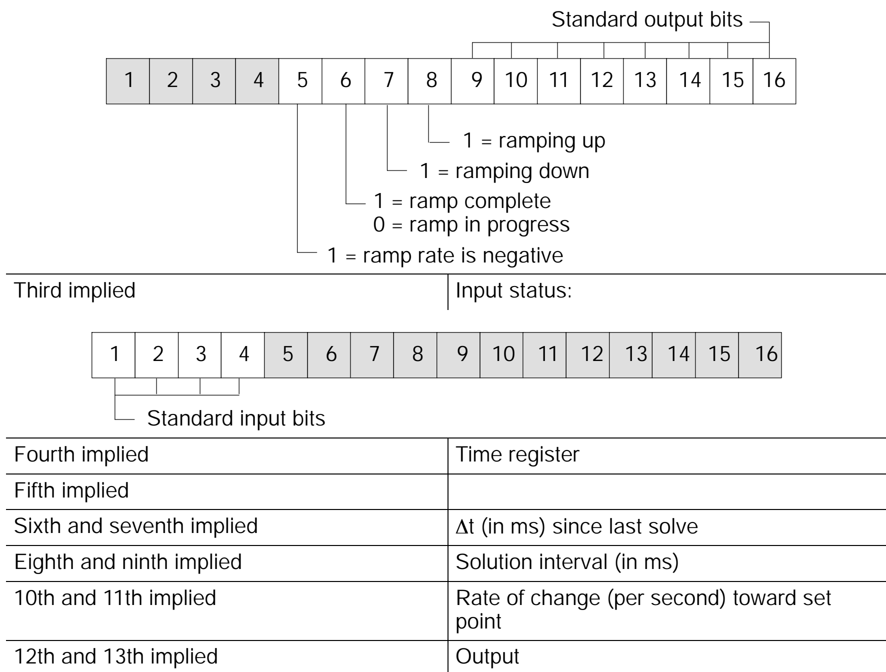
Block Structure



Parameter Block Assignment

The *length* of the RAMP *parameter block* is 14 registers:

Register	Content
Displayed and first implied	Set point (input)
Second implied	Output status:



Third implied	Input status:
Fourth implied	Time register
Fifth implied	
Sixth and seventh implied	Δt (in ms) since last solve
Eighth and ninth implied	Solution interval (in ms)
10th and 11th implied	Rate of change (per second) toward set point
12th and 13th implied	Output

21.5.12 RMPLN

The RMPLN function allows you to ramp up logarithmically to a target set point at a specified approach rate. At each successive call, it calculates the output until it is within a specified deadband (DB). DB is necessary because the incremental distance the ramp crosses decreases with each solve.

You need to specify:

- The target set point, in the same units as the contents of the input register are specified
- The sampling rate
- The time constant used for the logarithmic ramp, which is the time it takes to reach 63.2% of the new set point

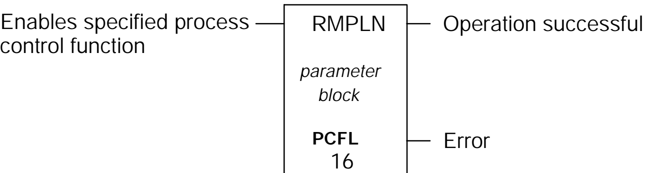
For best results, use a that is $\geq 4 * \Delta t$. This will ensure sufficient granularity in the output response.

You may use a flag to initialize after an undetermined down-time. The function will store a new sample, then wait for one cycle to collect the

second sample. Calculations will be skipped for one cycle and the output will be left as is, after which the ramp will resume.

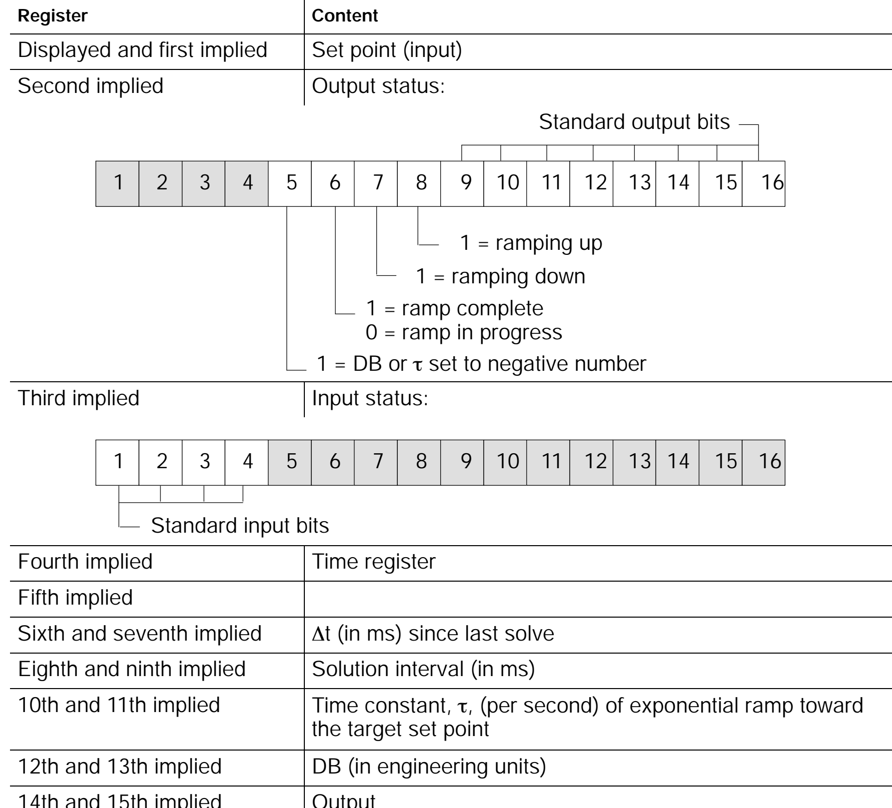
RMPLN terminates when the input reaches the target set point $\pm$ the specified DB and returns a DXDONE message.

Block Structure



Parameter Block Assignment

The *length* of the RMPLN *parameter block* is 16 registers:



Register	Content
Displayed and first implied	Set point (input)
Second implied	Output status:
Third implied	Input status:
Fourth implied	Time register
Fifth implied	
Sixth and seventh implied	Δt (in ms) since last solve
Eighth and ninth implied	Solution interval (in ms)
10th and 11th implied	Time constant, τ, (per second) of exponential ramp toward the target set point
12th and 13th implied	DB (in engineering units)
14th and 15th implied	Output

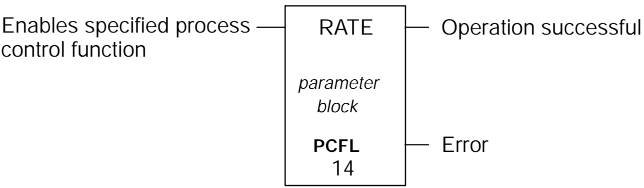
21.5.13 RATE

The RATE function calculates the rate of change over the last two input values. If you set an initialization flag, the function records a sample and sets the appropriate flags.

If a divide-by-zero operation is attempted, the function returns a DXERROR message.

It returns a DXDONE message when the operation completes successfully.

Block Structure



Parameter Block Assignment

The *length* of the RATE *parameter block* is 14 registers:

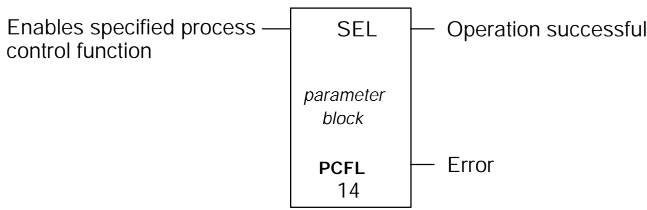
Register	Content
Displayed and first implied	Current input
Second implied	Output status: Standard output bits: 9–16 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Third implied	Input status: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 Standard input bits: 1–4
Fourth implied	Time register
Fifth implied	
Sixth and seventh implied	Δt (in ms) since last solve
Eighth and ninth implied	Solution interval (in ms)
10th and 11th implied	Last input
12th and 13th implied	Result

21.5.14 SEL

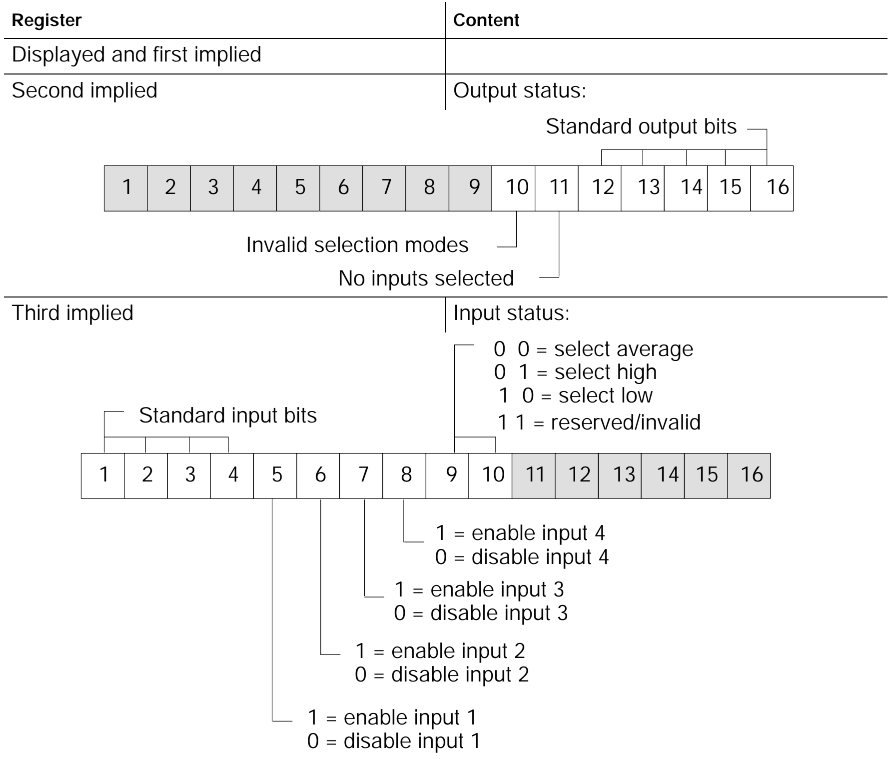
The SEL function compares up to four inputs and makes a selection based upon either the highest, lowest, or average value. You choose the inputs to be compared and the comparison criterion. The output is a copy of the selected input.

SEL returns a DXDONE message when the operation is complete.

Block Structure



Parameter Block Assignment
The *length* of the SEL *parameter block* can be up to 14 registers:



Register	Content
Displayed and first implied	
Second implied	Output status:
Third implied	Input status:

Fourth and fifth implied	Input 1
Sixth and seventh implied	Input 2
Eighth and ninth implied	Input 3
10th and 11th implied	Input 4
12th and 13th implied	Output

21.6 PCFL Regulatory Functions

Regulatory functions perform closed loop control in a variety of applications. Typically, this is a PID (proportional integral derivative) negative feedback control loop. The PID functions in PCFL offer varying degrees of functionality. Function 75, PID, has the same general functionality as the PID2 instruction but uses floating point math and represents some options differently. PID is beneficial in cases where PID2 is not suitable because of numerical concerns such as round-off.

☞ **Note:**

21.6.1 General Equations

$Y = YP + YI + YD + Bias$ integral bit ON

$Y = YP + YD + Bias + BT$ integral bit OFF

with the following high/low output limits:

$$Y_{high} \leq Y \leq Y_{low}$$

with

$.....YP \,,\ YI \,,\ YD = f(XD)$

$XD = SP \quad X \pm (GRZ * (1 \quad KGRZ))$ gain reduction zone used

$XD = SP \quad X$ gain reduction zone not used

Proportional Calculation

$YP = KP * XD$ proportional bit ON

$YP = 0$

Integral Calculation

$$YI = YI + KP * \frac{\Delta t}{TI} * \frac{XD_1 + XD}{2}$$

integral bit ON

$YI = 0$

Derivative Calculation

$DXD = X_1 \quad X$ base derivative or PV

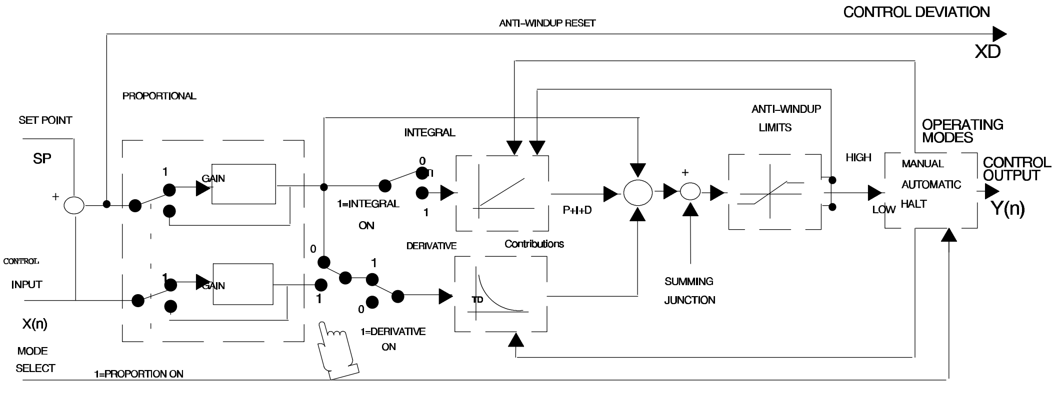
$DXD = XD \quad XD_1$

$$YD = \frac{(TD1 * YD) + (TD * KP * DXD)}{\Delta t + TD1} \quad \text{derivative bit ON}$$

$$YD = 0$$

where:

Y = manipulated variable output
YP = proportional part of the calculation
YI = integral part of the calculation
YD = derivative part of the calculation
$Bias$ = constant added to input
BT = bumpless transfer register
SP = set point
KP = proportional gain
Δt = time since last solve
TI = integral time constant
TD = derivative time constant
$TD1$ = derivative time lag
XD = error term, deviation
XD_1 = previous error term
X = process input
X_1 = previous process input



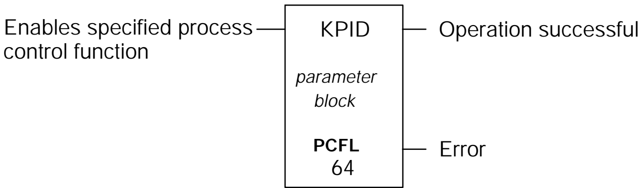
0 = base Derivitave on XD
1 = base Derivative on X

21.6.2 KPID

The KPID function offers a superset of the functionality of the PID function, with additional features that include:

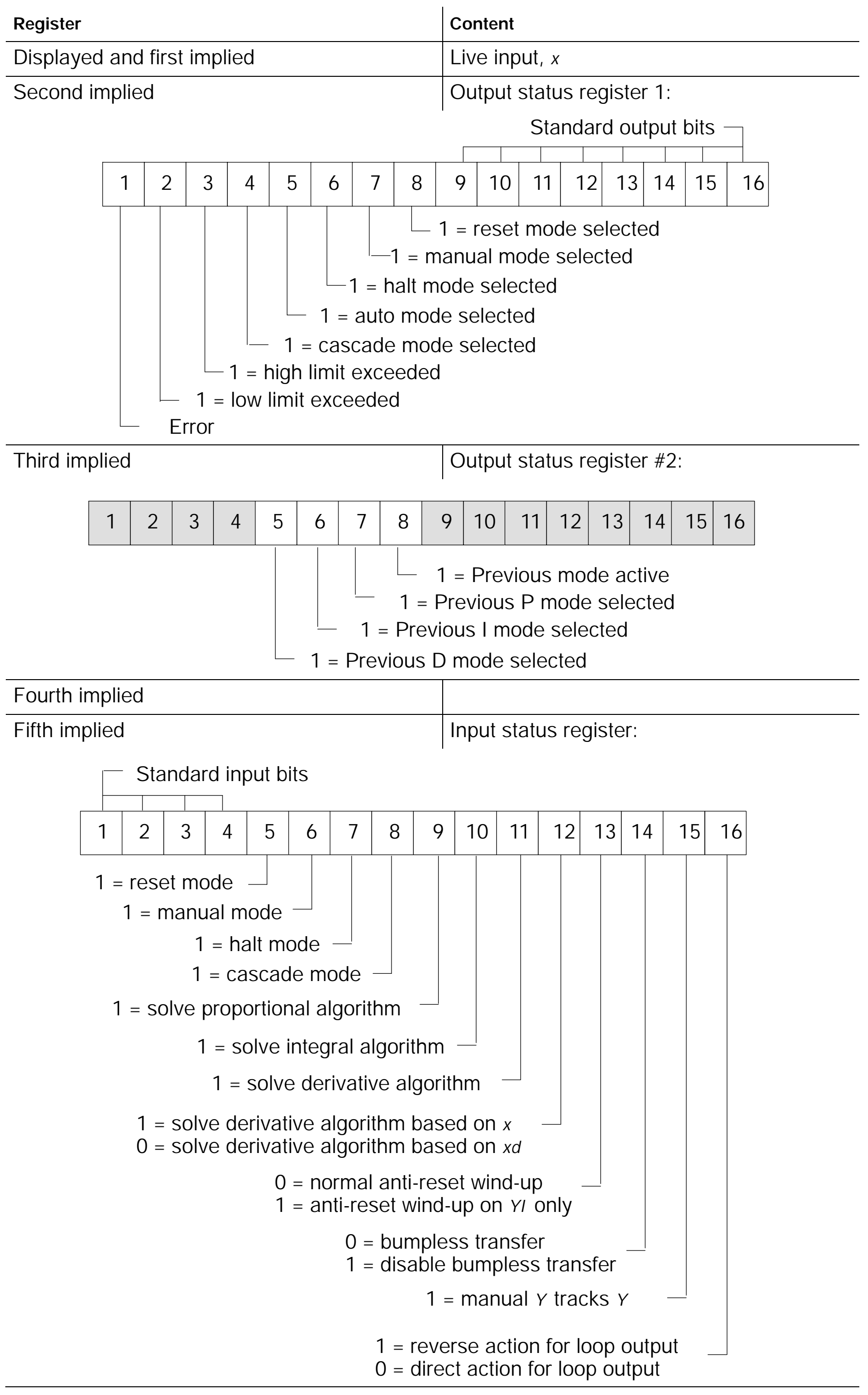
- ☐ A gain reduction zone
- ☐ A separate register for bumpless transfer when the integral term is not used
- ☐ A reset mode
- ☐ An external set point for cascade control
- ☐ Built-in velocity limiters for set point changes and changes to a manual output
- ☐ A variable derivative filter constant
- ☐ Optional expansion of anti-reset wind-up limits

Block Structure



Parameter Block Assignment

The *length* of the KPID *parameter block* is 64 registers:



Register	Content
Displayed and first implied	Live input, *x*
Second implied	Output status register 1: Bits 9–16: Standard output bits Bit 8: 1 = reset mode selected Bit 7: 1 = manual mode selected Bit 6: 1 = halt mode selected Bit 5: 1 = auto mode selected Bit 4: 1 = cascade mode selected Bit 3: 1 = high limit exceeded Bit 2: 1 = low limit exceeded Bit 1: Error
Third implied	Output status register #2: Bit 8: 1 = Previous mode active Bit 7: 1 = Previous P mode selected Bit 6: 1 = Previous I mode selected Bit 5: 1 = Previous D mode selected
Fourth implied	
Fifth implied	Input status register: Bits 1–4: Standard input bits Bit 5: 1 = reset mode Bit 6: 1 = manual mode Bit 7: 1 = halt mode Bit 8: 1 = cascade mode Bit 9: 1 = solve proportional algorithm Bit 10: 1 = solve integral algorithm Bit 11: 1 = solve derivative algorithm Bit 12: 1 = solve derivative algorithm based on *x*; 0 = solve derivative algorithm based on *xd* Bit 13: 0 = normal anti-reset wind-up; 1 = anti-reset wind-up on *YI* only Bit 14: 0 = bumpless transfer; 1 = disable bumpless transfer Bit 15: 1 = manual *Y* tracks *Y* Bit 16: 1 = reverse action for loop output; 0 = direct action for loop output

Input Parameters	sixth and seventh implied	Proportional rate, *KP*
	Eighth and ninth implied	Reset time, *TI*
	10th and 11th implied	Derivative action time, *TD*
	12th and 13th implied	Delay time constant, *TD1*
	14th and 15th implied	Gain reduction zone, *GRZ*
	16th and 17th implied	Gain reduction in GRZ, *KGRZ*
	18th and 19th implied	Limit rise of manual set point value
	20th and 21st implied	Limit rise of manual output
	22nd and 23rd implied	High limit for *Y*
	24th and 25th implied	Low limit for *Y*
	26th and 27th implied	Expansion for anti-reset wind-up limits
Inputs	28th and 29th implied	External set point for cascade
	30th and 31st implied	Manual set point
	32nd and 33rd implied	Manual *Y*
	34th and 35th implied	Reset for *Y*
	36th and 37th implied	Bias
Outputs	38th and 39th implied	Bumpless transfer register, *BT*
	40th and 41st implied	Calculated control difference (error term), *XD*
	42nd implied	Previous operating mode
	43rd and 44th implied	Δt (in ms) since last solve
	45th and 46th implied	Previous system deviation, *XD_1*
	47th and 48th implied	Previous input, *X_1*
	49th and 50th implied	Integral part for Y, *YI*
	51st and 52nd implied	Differential part for Y, *YD*
	53rd and 54th implied	Set point, *SP*
	55th and 56th implied	Proportional part for Y, *YP*
	57th implied	Previous operating status
Timing Information	58th implied	10 ms clock at time *n*
	59th implied	
	60th and 61st implied	Solution interval (in ms)
Output	62nd and 63rd implied	Manipulated output variable, *Y*

21.6.3 ONOFF

The ONOFF function is used to control the output signal between fully ON and fully OFF conditions so that a user can manually force the output ON or OFF. You can control the output via either a direct or reverse configuration:

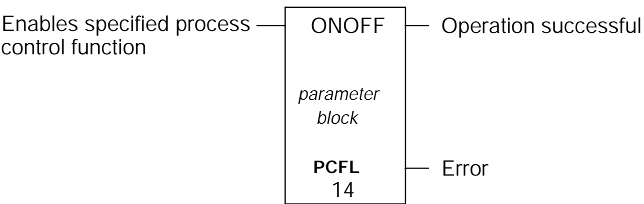
- In a direct configuration, the output will be set to ON when the input < (YSP DB), and to OFF when the input > (SP + DB)

- In a reverse configuration, the output will be set to ON when the input > (SP + DB), and to OFF when the input < (SP DB)

Manual Override

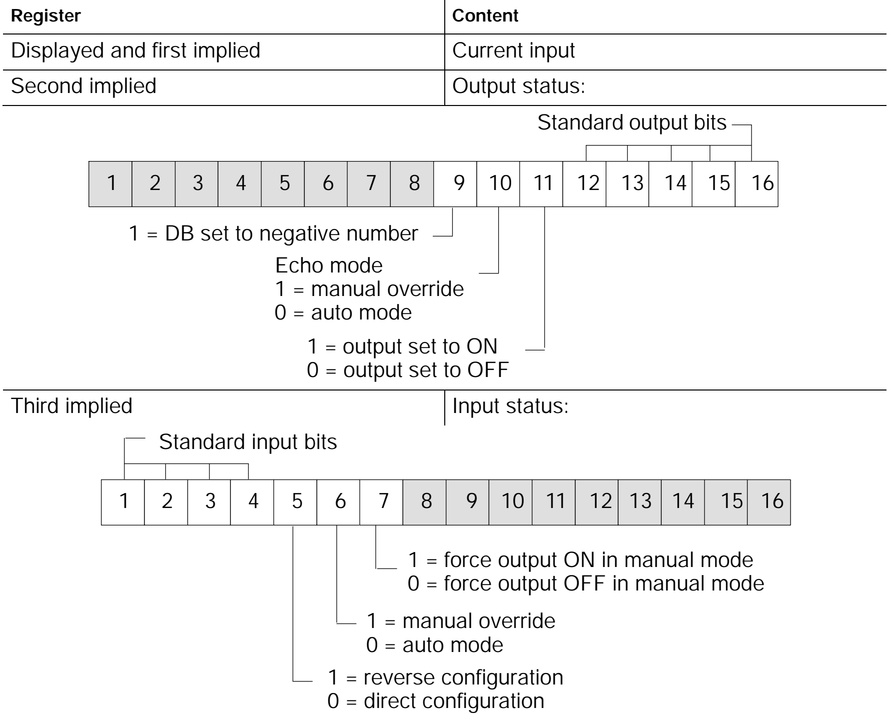
Two bits in the input status register (the third implied register in the *parameter block*) are used for manual override. When bit 6 is set to 1, manual mode is enforced. In manual mode, a 0 in bit 7 forces the output OFF, and a 1 in bit 7 forces the output ON. The state of bit 7 has meaning only in manual mode.

Block Structure



Parameter Block Assignment

The *length* of the ONOFF *parameter block* is 14 registers:



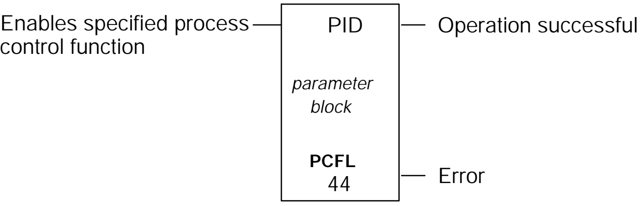
Register	Content
Displayed and first implied	Current input
Second implied	Output status:
Third implied	Input status:

Fourth and fifth implied	Set point, *SP*
Sixth and seventh implied	Deadband (DB) around SP
Eighth and ninth implied	Fully ON (maximum output)
10th and 11th implied	Fully OFF (minimum output)
12th and 13th implied	Output, ON or OFF

21.6.4 PID

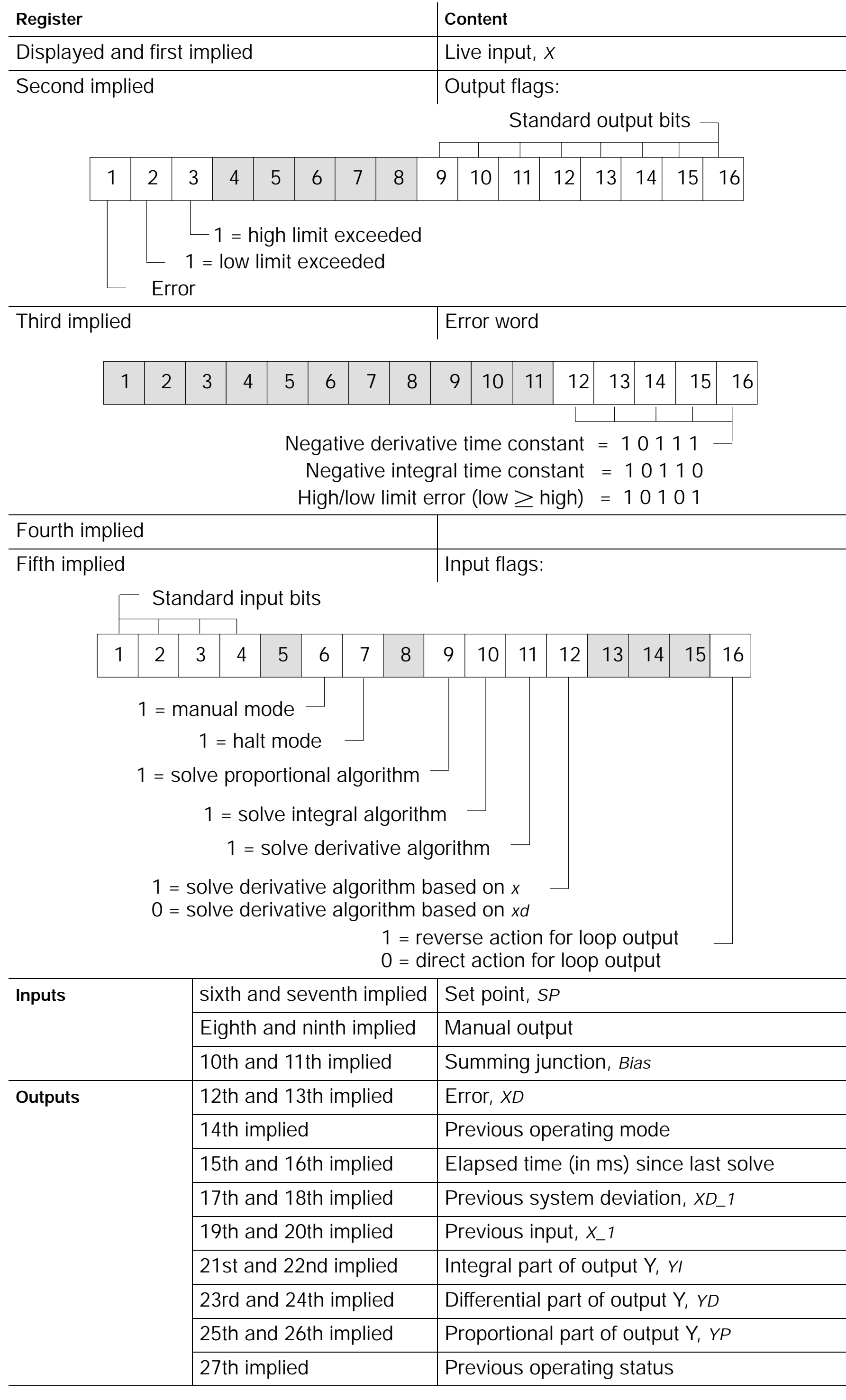
The PID function performs ISA non-interacting proportional-integral-derivative (PID) operations using floating point math. Because it uses FP math (unlike PID2), round-off errors are negligible.

Block Structure



Parameter Block Assignment

The *length* of the PID *parameter block* is 44 registers:

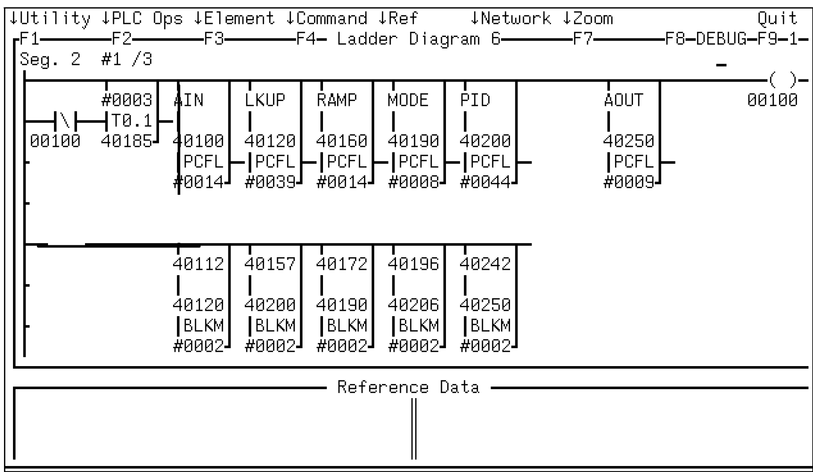


Register		Content
Displayed and first implied		Live input, *X*
Second implied		Output flags:
Third implied		Error word
Fourth implied		
Fifth implied		Input flags:
Inputs	sixth and seventh implied	Set point, *SP*
	Eighth and ninth implied	Manual output
	10th and 11th implied	Summing junction, *Bias*
Outputs	12th and 13th implied	Error, *XD*
	14th implied	Previous operating mode
	15th and 16th implied	Elapsed time (in ms) since last solve
	17th and 18th implied	Previous system deviation, *XD_1*
	19th and 20th implied	Previous input, *X_1*
	21st and 22nd implied	Integral part of output Y, *YI*
	23rd and 24th implied	Differential part of output Y, *YD*
	25th and 26th implied	Proportional part of output Y, *YP*
	27th implied	Previous operating status

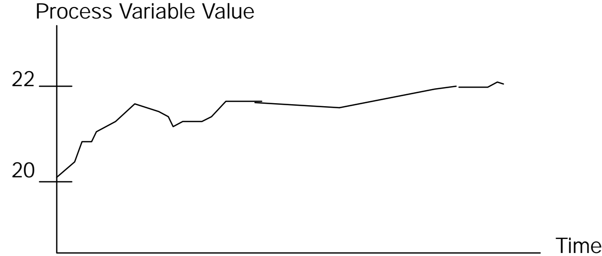
Timing Information	28th implied	Current time
	29th implied	
	30th and 31st implied	Solution interval (in ms)
Inputs	32nd and 33rd implied	Proportional gain, *KP*
	34th and 35th implied	Reset time, *TI*
	36th and 37th implied	Derivative action time, *TD*
	38th and 39th implied	High limit on output Y
	40th and 41st implied	Low limit on output Y
Outputs	42nd and 43rd implied	Manipulated control output, *Y*

21.6.5 A PID Example

This example illustrates how a typical PID loop could be configured using PCFL function 75. The calculation begins with the AIN function, which takes raw input simulated to cause the output to run between approximately 20 and 22 when the engineering unit scale is set to 0 ... 100.

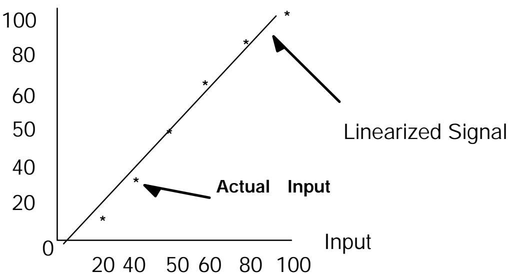


The process variable over time should look something like this:



Main PID Ladder Logic

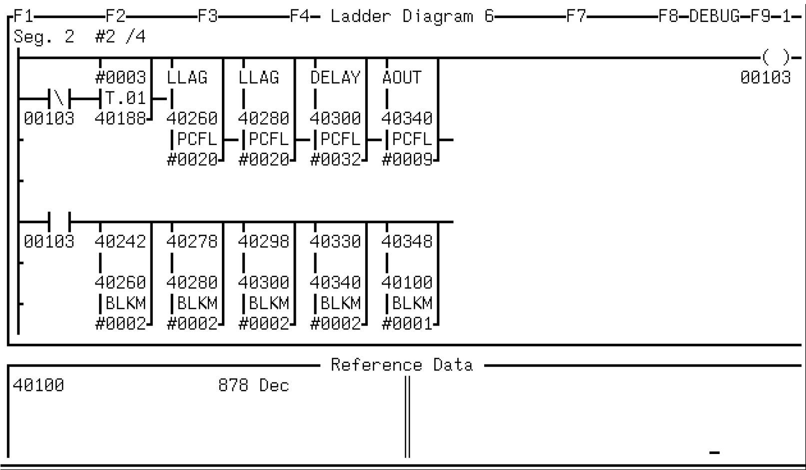
The AIN output is block moved to the LKUP function, which is used to scale the input signal. We do this because the input sensor is not likely to produce highly linear readings; the result is an ideal linear signal:



The look-up table output is block moved to the PID function. RAMP is used to control the rise (or fall) of the set point for the PID controller with regard to the rate of ramp and the solution interval. In this example, the set point is established in another logic section to simulate a remote setting. The MODE function is placed after the RAMP so that we can switch between the RAMP-generated set point or a manual value.

Simulated Process

The PID function is actually controlling the process simulated by this logic:



The process simulator is comprised of two LLAG functions that act as a filter and input to a delay queue that is also a PCFL function block. This arrangement is the equivalent of a second-order process with dead time.

The solution intervals for the LLAG filters do not affect the process dynamics and were chosen to give fast updates. The solution interval for the DELAY queue is set at 1000 ms with a delay of 5 intervals—i.e., 5 s. The LLAG filters each have lead terms of 4 s and lag terms of 10 s. The gain for each is 1.0.

In process control terms the transfer function can be expressed as:

$$G_p(S) = \frac{(4S + 1)\,(4S + 1)\,e^{5S}}{(10S + 1)\,(10S + 1)}$$

The AOUT function is used only to convert the simulated process output control value into a range of 0 ... 4,095, which simulates a field device. This integer signal is used as the process input in the first network.

PID Parameters

The PID controller is tuned to control this process at 20.0, using the Ziegler-Nichols tuning method. The resulting controller gain is 2.16, equivalent to a proportional band of 46.3%.

The integral time is set at 12.5 s/repeat (4.8 repeats/ min). The derivative time is initially 3 s, then reduced to 0.3 s to de emphasize the derivative effect.

An AOUT function is used after the PID. It conditions the PID control output by scaling the signal back to an integer for use as the control value.

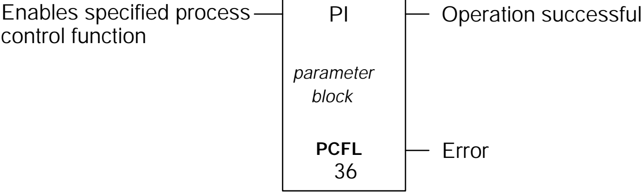
The entire control loop is preceded by a 0.1 s timer. The target solution interval for the entire loop is 1 s, and the full solve is 1 s. However, the nontime-dependent functions that are used (AIN, LKUP, MODE, and AOUT) do not need to be solved every scan. To reduce the scan time impact, these functions are scheduled to solve less frequently. The example has a loop solve every 3 s, reducing the average scan time dramatically.

Note: It is still important to be aware of the maximum scan impact. When programming other loops, you will not want all of the loops to solve on the same scan.

21.6.6 PI

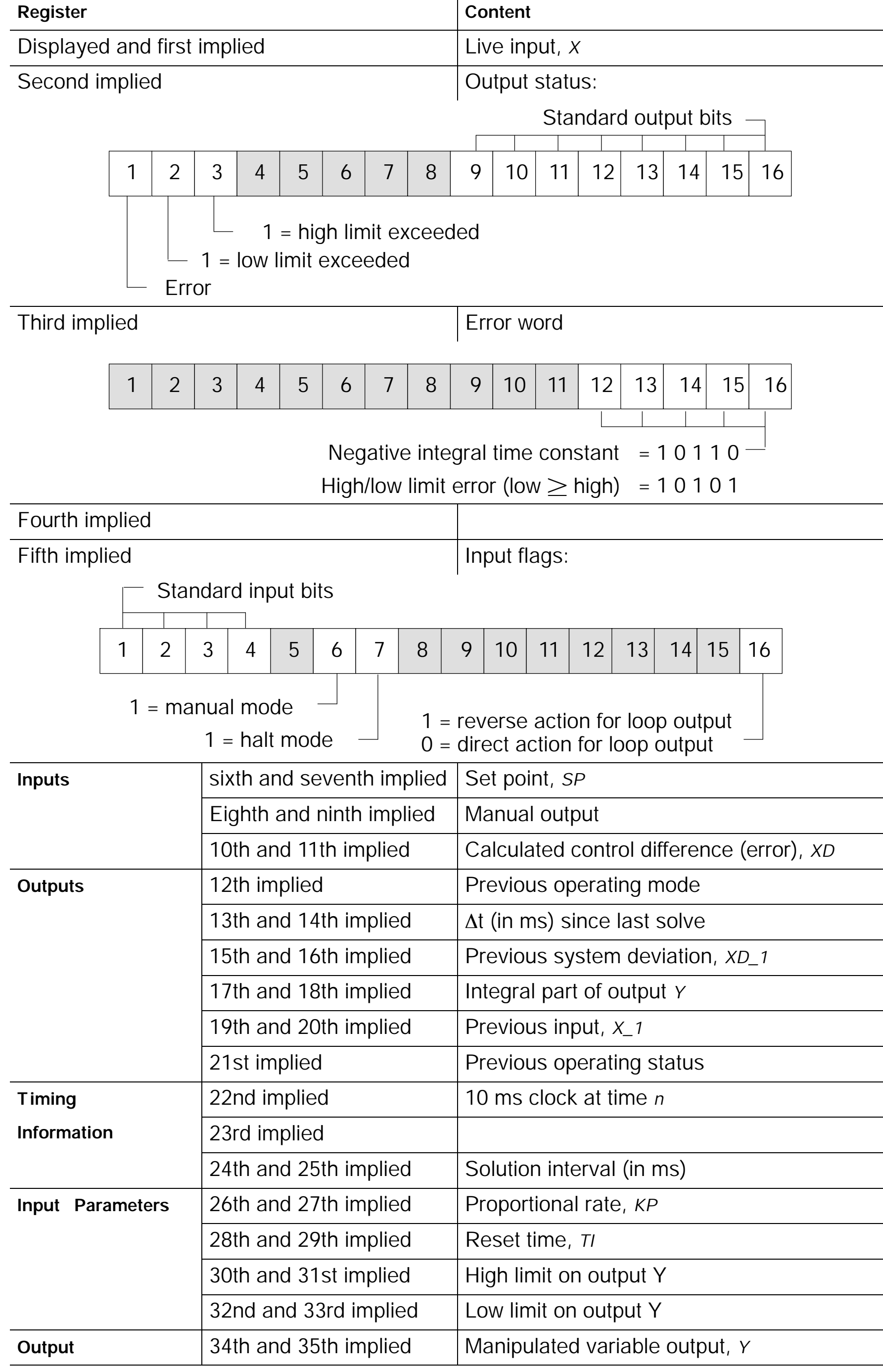
The PI function performs a simple proportional-integral operations using floating point math. It features halt/manual/auto operation modes. It is similar to the PID (page 450) and KPID (page 446) functions but does not contain as many options. It is available for higher-speed loops or inner loops in cascade strategies.

Block Structure



Parameter Block Assignment

The *length* of the PI *parameter block* is 36 registers:



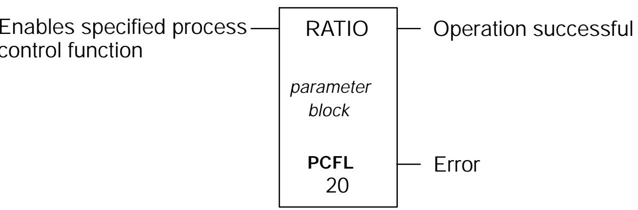
Register		Content
Displayed and first implied		Live input, *X*
Second implied		Output status: Bits 1–16; Standard output bits: 9–16; bit 3: 1 = high limit exceeded; bit 2: 1 = low limit exceeded; bit 1: Error
Third implied		Error word: Bits 1–16; bits 12–16: Negative integral time constant = 1 0 1 1 0; High/low limit error (low ≥ high) = 1 0 1 0 1
Fourth implied		
Fifth implied		Input flags: Bits 1–16; Standard input bits: 1–4; bit 6: 1 = manual mode; bit 7: 1 = halt mode; bit 16: 1 = reverse action for loop output, 0 = direct action for loop output
Inputs	sixth and seventh implied	Set point, *SP*
	Eighth and ninth implied	Manual output
	10th and 11th implied	Calculated control difference (error), *XD*
Outputs	12th implied	Previous operating mode
	13th and 14th implied	Δt (in ms) since last solve
	15th and 16th implied	Previous system deviation, *XD_1*
	17th and 18th implied	Integral part of output *Y*
	19th and 20th implied	Previous input, *X_1*
	21st implied	Previous operating status
Timing Information	22nd implied	10 ms clock at time *n*
	23rd implied	
	24th and 25th implied	Solution interval (in ms)
Input Parameters	26th and 27th implied	Proportional rate, *KP*
	28th and 29th implied	Reset time, *TI*
	30th and 31st implied	High limit on output Y
	32nd and 33rd implied	Low limit on output Y
Output	34th and 35th implied	Manipulated variable output, *Y*

21.6.7 RA TIO

The RATIO function provides a four-station ratio controller. Ratio control can be used in applications where one or more raw ingredients are dependent on a primary ingredient. The primary ingredient is measured, and the measurement is converted to engineering units via an AIN function (see page 426). The converted value is used to set the target for the other ratioed inputs.

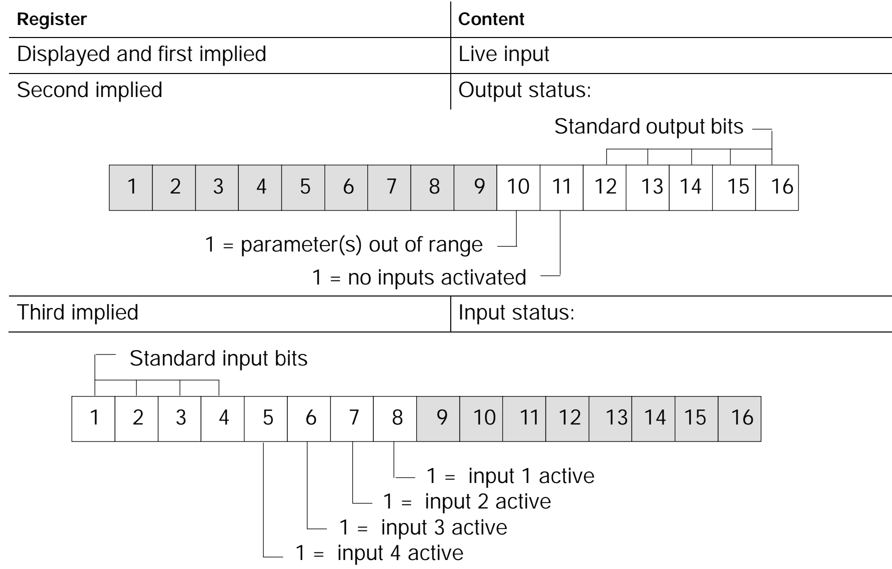
Outputs from the ratio controller can provide set points for other controllers. They can also be used in an open loop structure for applications where feedback is not required.

Block Structure



Parameter Block Assignment

The *length* of the RATIO *parameter block* is 20 registers:



Register	Content
Displayed and first implied	Live input
Second implied	Output status:
Third implied	Input status:

Fourth and fifth implied	Ratio for input 1
Sixth and seventh implied	Ratio for input 2
Eighth and ninth implied	Ratio for input 3
10th and 11th implied	Ratio for input 4
12th and 13th implied	Output for input 1
14th and 15th implied	Output for input 2
16th and 17th implied	Output for input 3
18th and 19th implied	Output for input 4

21.6.8 TOTAL

The TOTAL function provides a material totalizer for batch processing reagents. The input signal contains the units of weight or volume per unit of time. The totalizer integrates the input over time. The algorithm reports three outputs:

- The integration sum
- The remainder left to meter in
- The valve output (in engineering units)

The function uses up to three different set points—a *trickle flow* set point, a *target* set point, and an *auxiliary trickle flow* set point. The target set point is for the full amount to be metered in. Here the output will be turned OFF.

The trickle flow set point is the cut-off point when the output should be decreased from full flow to a percentage of full flow so that the target set point is reached with better granularity.

The auxiliary trickle flow set point is optional. It is used to gain another level of granularity. If this set point is enabled, the output is reduced further to 10% of the trickle output.

The totalizer works from zero as a base point. The set point must be a positive value.

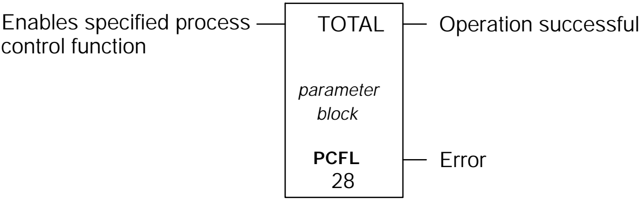
In normal operation, the valve output is set to 100% flow when the integrated value is below the trickle flow set point. When the sum crosses the trickle flow set point, the valve flow becomes a programmable percentage of full flow. When the sum reaches the desired target set point, the valve output is set to 0% flow.

Set points can be relative or absolute. With a relative set point, the deviation between the last summation and the set point is used. Otherwise, the summation is used in absolute comparison to the set point.

There is a halt option to stop the system from integrating.

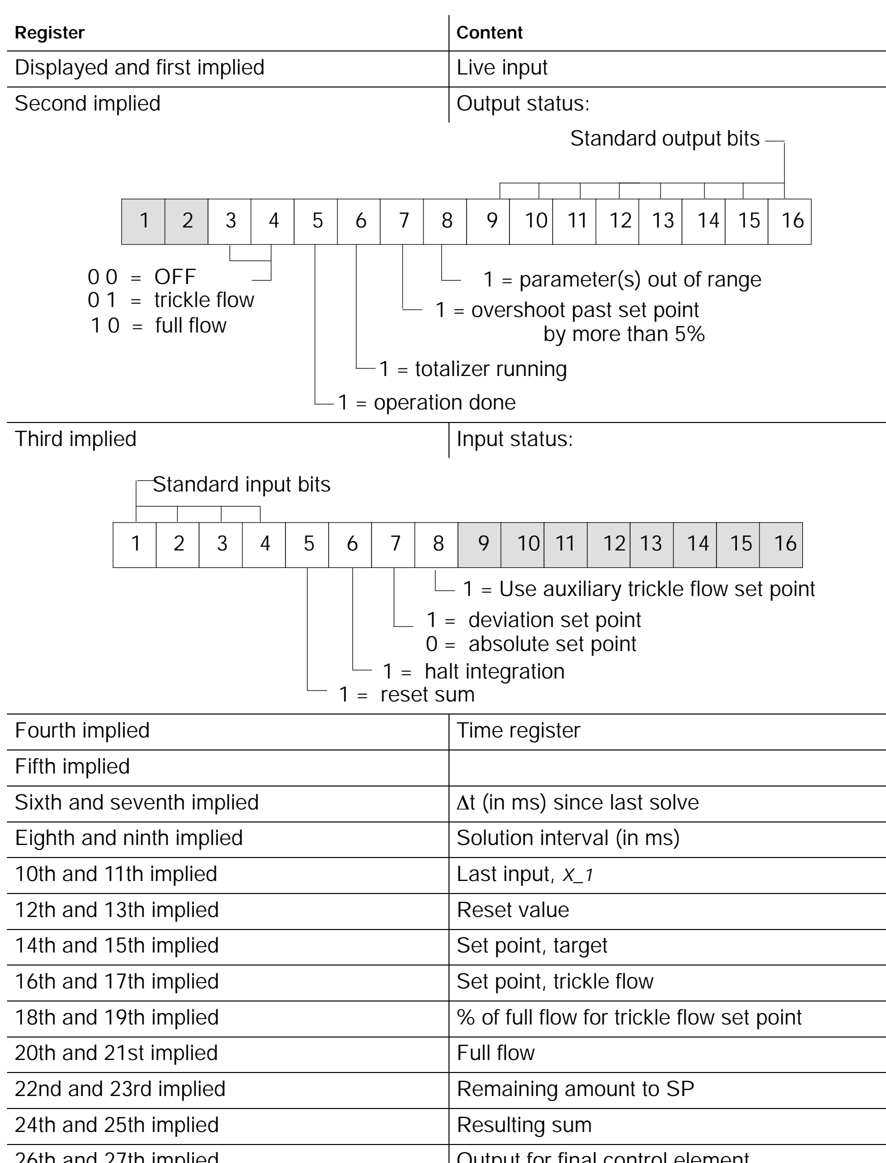
When the operation has finished, the output summation is retained for future use. You have the option of clearing this sum. In some applications, it is important to save the sum—e.g., if the meters or load cells cannot handle the full batch in one charge and measurements are split up, if there are several tanks to fill for a batch and you want to keep track of batch and production sums.

Block Structure



Parameter Block Assignment

The *length* of the TOTAL *parameter block* is 28 registers:



Register	Content
Displayed and first implied	Live input
Second implied	Output status: Bits 1–16; Standard output bits (9–16) Bits 3–4: 0 0 = OFF; 0 1 = trickle flow; 1 0 = full flow Bit 5: 1 = operation done Bit 6: 1 = totalizer running Bit 7: 1 = overshoot past set point by more than 5% Bit 8: 1 = parameter(s) out of range
Third implied	Input status: Bits 1–16; Standard input bits (1–4) Bit 5: 1 = reset sum Bit 6: 1 = halt integration Bit 7: 1 = deviation set point; 0 = absolute set point Bit 8: 1 = Use auxiliary trickle flow set point
Fourth implied	Time register
Fifth implied	
Sixth and seventh implied	Δt (in ms) since last solve
Eighth and ninth implied	Solution interval (in ms)
10th and 11th implied	Last input, *X_1*
12th and 13th implied	Reset value
14th and 15th implied	Set point, target
16th and 17th implied	Set point, trickle flow
18th and 19th implied	% of full flow for trickle flow set point
20th and 21st implied	Full flow
22nd and 23rd implied	Remaining amount to SP
24th and 25th implied	Resulting sum
26th and 27th implied	Output for final control element

Chapter 22
Loadable Instructions

- Loadable Software Packages
- HSBY
- CALL
- MBUS
- PEER
- Custom Loadables
- Extended Math Loadables
- EARS
- EUCA

22.1 Loadable Software Packages

Software loadable functions are available to support optional control modules such as the coprocessing and Hot Standby capabilities, and to support special applications such as drum sequencing and the event/alarm recording system (EARS).

22.1.1 Loadable Support for Controller Option Modules

Loadable	Part Number*	Option Module	PLCs Supported
HSBY	SW-AP9X-RXA	AM-R911-000	chassis mounts
	SW-AP98-RXA	AS-S911-800	984-680/685/780/785 slot mounts, host based
CHS		140 CHS 110 00	Quantum
ESI		140 ESI 062 10	Quantum
CALL	SW-AP9X-CXB	AM-C986-004	chassis mounts
MBUS/PEER	SW-AP9X-AXA	AM-S975-100	chassis mounts
	SW-AP98-AXA	AM-S975-820	984-685/780/785 slot mounts, host based
MSTR**	SW-AP9X-MBP	AM-S985-0*x*0	chassis mounts

* When the X in the above software part numbers is a T, the medium is a P190 tape; when the X is a D, the software media are 5.25 in and 3.5 in diskettes.

** The MSTR function that is a loadable for the chassis mount controllers is functionally identical to the MSTR block provided in firmware for the 984-385/485/685/785 PLCs.

22.1.2 Other Loadable Functions

Loadable	Part Number*	Software Capability	PLCs Supported
DRUM/ICMP	SW-SA*x*9-001	Sequence control	chassis mounts
	SW-AP98-S*x*A		slot mounts, host based
FN*xx*	SW-AP98-GDA	Custom loadable	slot mounts, host based, Quantum
Loadables Library**	SW-AP9*x*-D*x*A	includes MATH, DMTH, TBLK, BLKT, CKSM, and PID2	chassis mounts
PID2**	SW-AP9*x*-2*x*a	PID2 closed loop control	chassis mounts
EARS	SW-AP9D-EDA	Event/alarm recording system	All PLCs

* When the *x* in the above software part numbers is a T, the medium is a P190 tape; when the *x* is a D, the software media are 5.25 in and 3.5 in diskettes.

** TBLK, BLKT, CKSM, and PID2 are functionally identical to those instructions of the same name provided in firmware for the 984-385/485/685/785 PLCs.

This chapter describes all the loadable functions that support option modules except:

- MSTR, which is described in Chapter 18
- The sequence control loadables DRUM and ICMP, which are described in Chapter 16 (see pages 320 and 323, respectively)
- The MATH and DMTH instructions, which are described in Chapter 7 (see pages 139 and 143, respectively)
- The BLKT and TBLK instructions, which are described in Chapter 9 (see pages 193 and 196, respectively)
- The PID2 instruction, which is described in Section 21.2 (page 403)
- The CKSM function, which is described in Chapter 17

22.2 HSBY

The HSBY loadable instruction manages a 984 Hot Standby control system. This instruction must be placed in network 1 of segment 1 in the application logic for both the primary and standby controllers. It allows you to program a *nontransfer area* in system state RAM—an area that protects a serial group of registers in the standby controller from being modified by the primary controller.

Through the HSBY instruction you can access two registers—a *command* register and a *status* register—that allow you to monitor and control Hot Standby operations. The status register is the third register in the nontransfer area you specify.

22.2.1 Characteristics

Size

Three nodes high

PLC Compatibility

Available as a loadable in all 984 PLC types that support Hot Standby

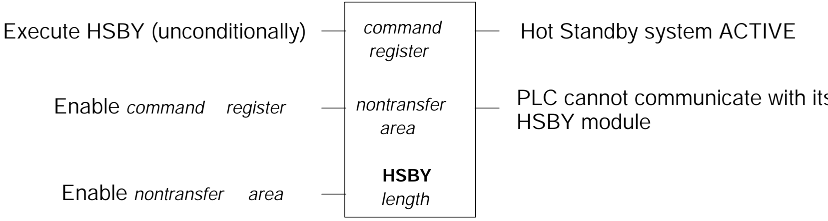
Does not support Hot Standby in any Quantum PLCs (see the CHS instruction on page 468)

Opcode

FF hex (default)

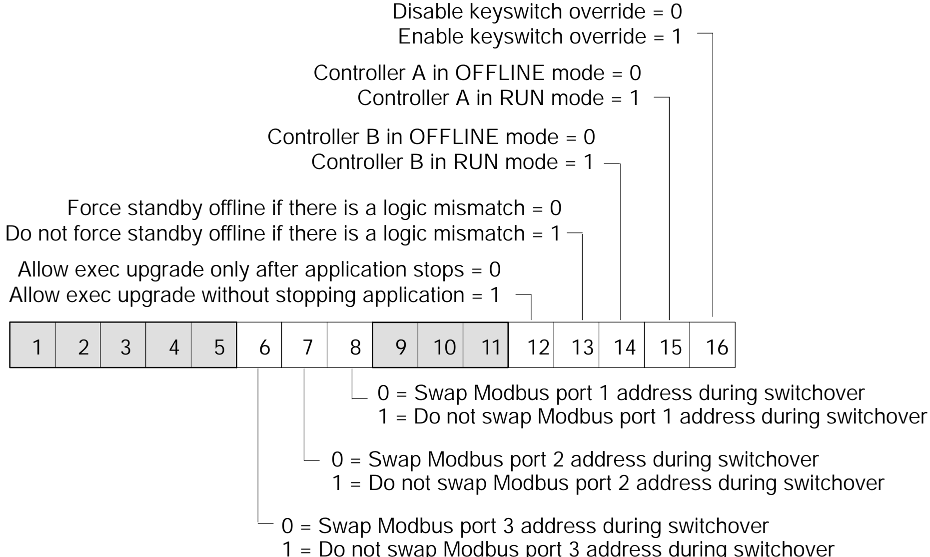
22.2.2 Representation

Block Structure



Top Node Content

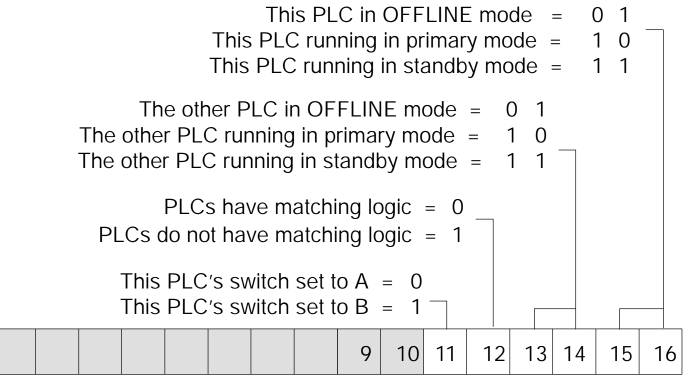
The 4*x* register entered in the top node is the HSBY command register; eight bits in this register may be configured and controlled via your panel software:



Middle Node Content

The 4*x* register entered in the middle node is the first register reserved for the *nontransfer area* in state RAM. The first three registers in the nontransfer area are special registers:

Register	Content
Displayed and first implied	*reverse transfer registers* for passing information from the standby to the primary PLC
Second implied	HSBY *status register* :



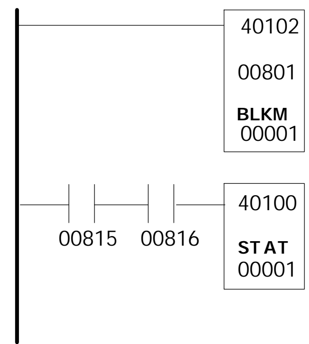
The content of the remaining registers is application-specific; the *length* is defined in the bottom node.

Bottom Node Content

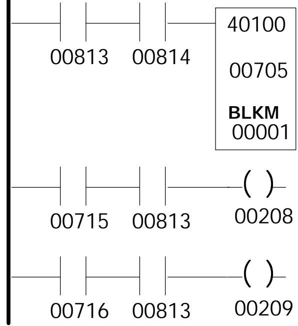
The integer value entered in the bottom node defines the *length* —i.e., the number of registers—of the HSBY *nontransfer area* in state RAM. The *length* must be at least four registers; in the range from 4 ... 255 registers in a 16-bit CPU, and in the range 4 ... 8000 registers in a 24-bit CPU.

22.2.3 An HSBY Reverse Transfer Example

The two networks below are for a primary controller that monitors two fault lamps and a reverse transfer that sends status data from the standby controller to the primary controller. The first network must be network 2 of segment 1; the second network must *not* be in segment 1.



Network 2, Must be segment 1



Network must not be in Segment 1

The first BLKM function transfers the HSBY status register (40102) to internal coils, starting at 00801. The STAT instruction, which is enabled if the other controller is in standby mode, sends one status register word from the standby controller to a reverse transfer register (40100) in the primary controller.

22.3 CHS

The logic in the CHS loadable instruction is the engine that drives the Hot Standby capability in a Quantum PLC system. Unlike the 984 HSBY instruction (page 464), the use of the CHS instruction in the ladder logic program is optional. However, the loadable software itself must be installed in the Quantum PLC in order for a Hot Standby system to be implemented.

22.3.1 How to Configure a Quantum Hot Standby System

In a Quantum PLC system that is programmed via Modsoft, there are two alternative methods available to configure Hot Standby capability:

Method 1 Program the CHS instruction in network 1, segment 1 of your ladder logic program and unconditionally connect the top node to the power rail via a horizontal short (as the HSBY instruction is programmed in a 984 Hot Standby system)

Method 2 Define the Hot Standby configuration parameters in a series of Hot Standby configuration extension screens in Modsoft (Version 2.3 or higher)

You may use the configuration extension screens to define and control the Hot Standby configuration while inserting a CHS instruction in ladder logic to access the CHS Zoom screen in Modsoft. The CHS Zoom screen allows you to access the Hot Standby command and status registers, and it is an easy way to perform PLC executive upgrades without shutting down the system. For more details, refer to the *Quantum CHS 110 Hot Standby Planning and Installation Guide*.

Method 1: Hot Standby System Configuration via the CHS Instruction

Method 1 is particularly useful if you are porting Hot Standby code from a 984 application to a Quantum application. The structure of the CHS instruction is almost exactly the same as the HSBY instruction. You simply remove the HSBY instruction from the 984 ladder logic and replace it with a CHS instruction in the Quantum logic.

If you are using the CHS instruction in ladder logic, the only difference between it and the HSBY instruction is the use of an output from the bottom node. This output senses whether or not method 2 has been used. If the Hot Standby configuration extension screens have been used to define the Hot Standby configuration, the configuration parameters in the screens will override any different parameters defined by the CHS instruction at system startup.

Method 2: The Modsoft Configuration Extension Screens

Method 2 is designed to make the Hot Standby configuration process more versatile. The details of the configuration are all defined in a pair of configuration extension screens in Modsoft. Although the CHS software must be loaded to the PLC, the instruction itself does not need to be entered in the ladder logic program. If you use method 2 and insert a CHS instruction in the logic, the parameters defined in the configuration extension screens will override attempts you have made to configure the Hot Standby system in ladder logic at the time of startup.

Major advantages of method 2 include:

- Your ability to increase the amount of state RAM data that is transferred between the primary and standby PLCs on every scan; all configured state RAM in your system (16K, 32K, or 64K words, depending on the type of Quantum PLC you are using) can be made part of the state RAM transfer area

- Your ability to reduce the amount of state RAM data in the transfer area to a small amount of critical I/O; the minimum amount of state RAM data that needs to be scheduled for transfer in every scan is 16 registers of 4*x* data

- Your ability to chunk a certain amount of noncritical state RAM data to be transferred in pieces over multiple scans; this approach can be used to reduce the impact of the Hot Standby system on scan time

- When the CHS instruction is not used in ladder logic, network 1 of segment 1 is free for other purposes; this may be necessary for certain special applications such as MSL, where another instruction is required in network 1, segment 1

For a detailed discussion of the issues related to the configuration extension capabilities of a Quantum Hot Standby system, refer to the *Quantum CHS 110 Hot Standby Planning and Installation Guide*.

22.3.2 CHS Instruction Characteristics

Size

Three nodes high

PLC Compatibility

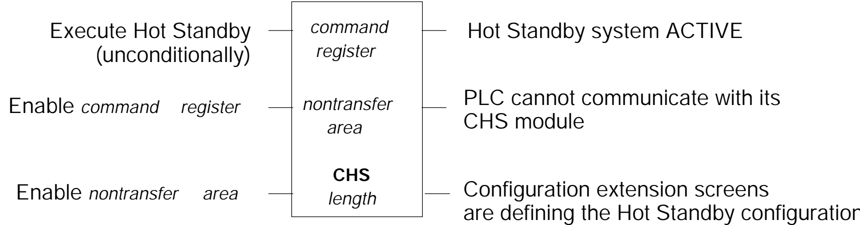
Available as a loadable in all Quantum PLC types (186, 386, 486, etc.) that support Hot Standby

Does not support Hot Standby in any non-Quantum PLCs (see the HSBY instruction on page 464)

Opcode

22.3.3 Representation

Block Structure



Inputs

When the CHS instruction is inserted in ladder logic to control the Hot Standby configuration parameters, its top node must be connected directly to the power rail by a horizontal short. No control logic, such as contacts, should be placed between the rail and the input to the top node.

The middle node enables the command register. This input must be ON for the Hot Standby system to be functional.

The bottom input enables the nontransfer area. If this input is OFF, the nontransfer area will not be used, and the Hot Standby status register will not exist.

Caution: Although it is legal to enable and disable the nontransfer area while the Hot Standby system is running, we strongly discourage this practice. It can lead to erratic behavior in the Hot Standby system.

Outputs

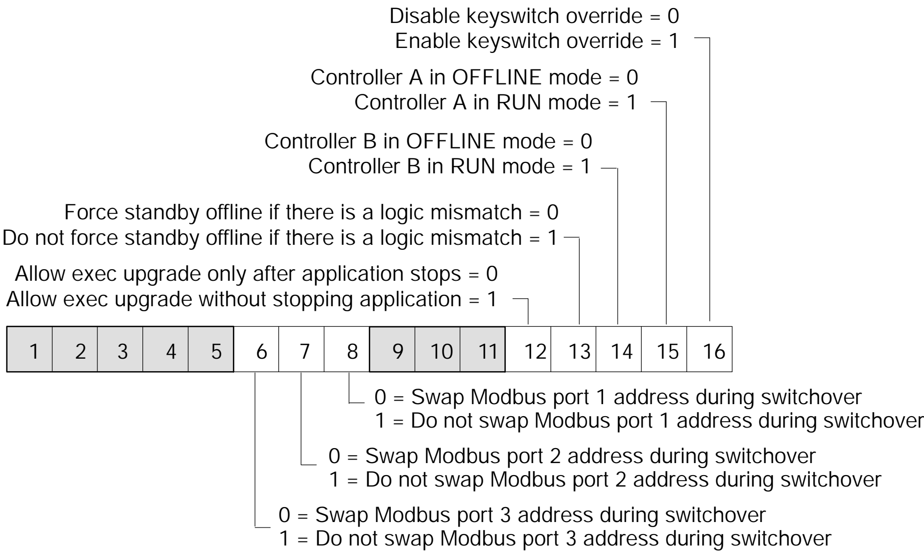
The output from the top node goes ON to indicate that the Hot Standby system is running.

The output from the middle node goes ON if the system detects a system interface error while the ladder logic is being solved.

The output from the bottom node goes ON when the Hot Standby system configuration has been set by the Hot Standby configuration extension capability in Modsoft. The configuration parameters may be changed during system runtime via the CHS Zoom screen or a Modsoft reference data editor (RDE); however the original configuration parameters will be reset if the system is powered down and then restarted.

Top Node Content

The 4*x* register entered in the top node is the Hot Standby *command register*; eight bits in this register are used to configure and control Hot Standby system parameters:



The Hot Standby command register must be outside of the *nontransfer area* of state RAM.

Middle Node Content

The 4*x* register entered in the middle node is the first register in the *nontransfer area* of state RAM. The nontransfer area must contain at least four registers, the first three of which have a predefined usage:

Register	Content
Displayed and first implied	*reverse transfer registers* for passing information from the standby to the primary PLC
Second implied	CHS *status register* :

This PLC in OFFLINE mode = 0 1
This PLC running in primary mode = 1 0
This PLC running in standby mode = 1 1

The other PLC in OFFLINE mode = 0 1
The other PLC running in primary mode = 1 0
The other PLC running in standby mode = 1 1

PLCs have matching logic = 0
PLCs do not have matching logic = 1

This PLC's switch set to A = 0
This PLC's switch set to B = 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1 = middle output ON (indicating an error condition)
1 = top output ON (indicating Hot Standby system is running)

The content of the remaining registers is application-specific; the *length* is defined in the bottom node.

The 4*x* registers in the nontransfer area are never transferred from the primary to the standby PLC during the logic scans. One reason for scheduling additional registers in the nontransfer area is to reduce the impact of state RAM transfer on the total system scan time.

Bottom Node Content

The integer value entered in the bottom node defines the *length* —i.e., the number of registers—of the Hot Standby *nontransfer area* in state RAM. The *length* must be in the range 4 ... 8000 registers.

22.4 CALL

A CALL instruction activates an immediate or deferred DX function from a library of functions defined by function codes. The Copro copies the data and function code into its local memory, processes the data, and copies the results back to Controller memory.

22.4.1 Characteristics

Size
Three nodes high

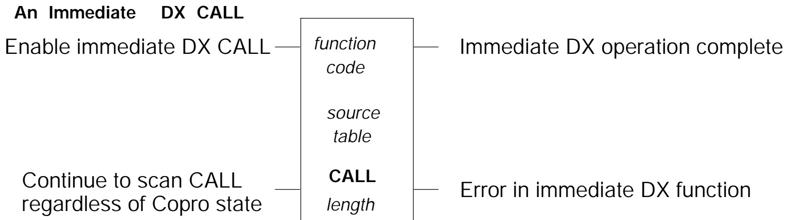
PLC Compatibility
Available as a loadable in PLC types that support a C986 Copro module

Opcode
5F hex (default)

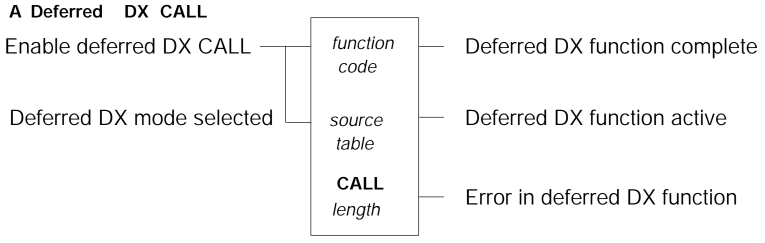
22.4.2 Representation

Block Structure
The inputs and outputs are different, depending on whether you call an immediate DX function or a deferred DX function:

An Immediate DX CALL



A Deferred DX CALL



Inputs

The input to the top node is used to initiate the CALL. The instruction calls a deferred DX when the input to the middle node is enabled and an immediate DX when no middle input is programmed. The input to the bottom node is used with an immediate DX function to keep scanning the instruction regardless of the state of the top input.

Outputs

The output from the top node goes ON when the function completes successfully. The output from the middle node, which is used only with deferred DX functions, goes ON to indicate that the function is in process. The output from the bottom node will go ON if an error is detected in the function.

Top Node Content

The top node is used to specify the *function code* to be executed. It may be entered explicitly as a constant or as a value in a 4*x* holding register. The codes fall into two ranges: 0 ... 499 are for *user-definable* DXs, and 500 ... 9999 are for *system* DXs (immediate and deferred) provided by Modicon:

Immediate DX Functions

Name	Code	Function
f_config	500	Obtain Copro configuration data
f_2md_fl	501	Convert a two-register long integer to 64-bit floating point
f_fl_2md	502	Convert floating point to two-register long integer
f_4md_fl	503	Convert a four-register long integer to floating point
f_fl_4md	504	Convert floating point to four-register long integer
f_1md_fl	505	Convert a one-register long integer to floating point
f_fl_1md	506	Convert floating point to one-register long integer
f_exp	507	Exponential function
f_log	508	Natural logarithm
f_log10	509	Base 10 logarithm
f_pow	510	Raise to a power
f_sqrt	511	Square root
f_cos	512	Cosine
f_sin	513	Sine
f_tan	514	Tangent
f_atan	515	Arc tangent x
f_atan2	516	Arc tangent y/x
f_asin	517	Arc sine
f_acos	518	Arc cosine
f_add	519	Add
f_sub	520	Subtract
f_mult	521	Multiply

f_div	522	Divide
f_deg_rad	523	Convert degrees to radians
f_rad_deg	524	Convert radians to degrees
f_swap	525	Swap byte positions within a register
f_comp	526	Floating point compare
f_dbwrite	527	Write Copro register database from PLC
f_dbread	528	Read Copro register database from PLC

Deferred DX Functions

Name	Code	Function
f_config	500	Obtain Copro configuration data
f_d_dbwr	501	Write Copro register database from PLC
f_d_dbrd	502	Read Copro register database from PLC
f_dgets	515	Issue dgets() on comm line
f_dputs	516	Issue dputs() on comm line
f_sprintf	518	Generate a character string
f_sscanf	519	Interpret a character string
f_egets	520	IEEE-488 gets() function
f_eputs	521	IEEE-488 puts() function
f_ectl	522	IEEE-488 error control function

Middle Node Content

The 4*x* register in the middle node is the first in a block of registers to be passed to the Copro for processing; the number of registers in the block is defined in the bottom node.

22.5 ESI

ESI is an optional loadable instruction that can be used in a Quantum PLC system to support operations using a 140 ESI 062 10 Quantum ASCII module. The PLC can use the ESI instruction to invoke the module. The power of the loadable is its ability to cause a sequence of commands over one or more logic scans

22.5.1 ESI-Driven Command Sequences

Via the ESI instruction, the PLC can invoke the ESI 062 ASCII module to:

- Read an ASCII message from a serial port on the ESI 062 module, then perform a sequence of Get Data transfers from the module to the PLC
- Write an ASCII message to a serial port on the ESI 062 module after having performed a sequence of Put Data transfers to the variable data registers in the module
- Perform a sequence of Get Data transfers (up to 16,384 registers of data from the ESI 062 module to the PLC); one Get Data transfer will move up to 10 data registers each time the instruction is solved
- Perform a sequence of Put Data (up to 16,384 registers of data to the ESI 062 module from the PLC) one Put Data transfer will move up to 10 registers of data each time the instruction is solved
- Abort the the ASCII message currently running

The ESI 062 is a 12-word bidirectional module. User logic can write or read up to 12 words to/from the module each time the ESI instruction is scanned. Information is transferred between the PLC and the module through a routine data area consisting of a 12-word command structure and a 12-word response structure. The command structure implements 4*x* output data, and the response structure uses 3*x* input data. See sections 22.5.5 ... 22.5.9 for more details on the command/response structures.

A non-volatile message table and a block of 16K registers for volatile variable data reside in the ESI 062 module.

Use of the ESI loadable instruction in ladder logic is optional. Other standard instructions may be used to manage the messaging operations between the PLC and the ESI 062 module. The ESI instruction can simplify the logic needed to execute these commands, especially when large blocks of registers, transferred over multiple scans, are involved.

22.5.2 Characteristics

Size

Three nodes high

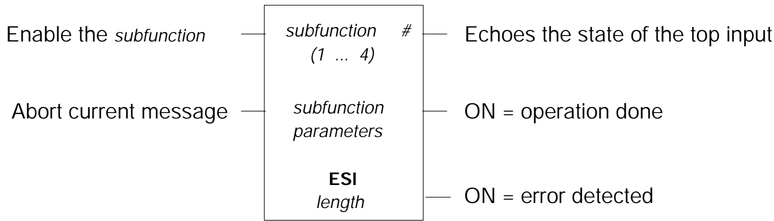
PLC Compatibility

Available as a loadable in all Quantum PLC types (186, 386, 486, etc.)

Not supported in any non-Quantum PLCs

22.5.3 Representation

Block Structure



Inputs

ESI has two inputs, to the top and middle nodes. When the input to the top node is powered ON, it enables the ESI instruction and starts executing the command indicated by the *subfunction* code in the top node.

When the input to the middle node is powered ON, an Abort command is issued. If a message is running when the Abort command is received, the instruction will complete; if a data transfer is in process when the Abort command is received, the transfer will stop and the instruction will complete.

Outputs

ESI has three outputs. The output from the top node echoes the state of the top input. The output from the middle node goes ON for one scan when the *subfunction* operation specified in the top node is completed,

timed out, or aborted. The bottom node goes ON for one scan if an error has been detected. Error checking is the first thing that is performed on the instruction when it is enabled, it it is completed before the *subfunction* is executed. For more details on error checking, see section 22.5.4.

Top Node

The top node may contain either a 4*x* register or an integer. The integer or the value in the register must be in the range 1 ... 4. It represents one of four possible *subfunction* command sequences to be executed by the instruction:

Subfunction	Command Sequence
1	One Read ASCII Message command followed by multiple Get Data commands
2	Multiple Put Data commands followed by one Write ASCII Message command
3	Zero or more Get Data commands
4	Zero or more Put Data commands

A fifth command, Abort ASCII Message, can be initiated by enabling the middle input to the ESI instruction.

Middle Node

The middle node contains the first 4*x* register in a list of contiguous registers that define the *subfunction parameters* needed to run the command sequence:

Register	Parameter	Contents
Displayed	ESI status register	Returned error codes (section 22.5.4)
First implied	Address of the first 4*x* register in the command structure	Register address minus the leading 4 and any leading zeros, as specified in the I/O Map (e.g., 1 represents register 400001)
Second implied	Address of the first 3*x* register in the command structure	Register address minus the leading 3 and any leading zeros, as specified in the I/O Map (e.g., 7 represents register 300007)
Third implied	Address of the first 4*x* register in the PLC's data register area	Register address minus the leading 4 and any leading zeros (e.g., 100 representing register 400100)
Fourth implied	Address of the first 3*x* register in the PLC's data register area	Register address minus the leading 3 and any leading zeros (e.g., 1000 representing register 301000)
Fifth implied	Starting register for data register area in module	Number in the range 0 ... 3FFF hex
Sixth implied	Data transfer count	Number in the range 0 ... 4000 hex

Seventh implied	ESI timeout value, in 100 ms increments	Number in the range 0 ... FFFF hex, where 0 means no timeout
The last two registers below are used only when the Read/W rite ASCII Messages commands are being executed (*subfunction* indicated in the top node is 1 or 2)		
Eighth implied	ASCII message number	Number in the range 1 ... 255 dec
Ninth implied	ASCII port number	1 or 2

Bottom Node

The bottom node contains the *length* of the table in the middle node—i.e., the number of *subfunction parameter* registers. For Read/Write operations, the *length* must be 10 registers. For Put/Get operations, the required *length* is eight registers; 10 may be specified and the last two registers will be unused.

22.5.4 Error Checking

The command sequence executed by the ESI 062 module (specified by the *subfunction* value in the top node of the ESI instruction) needs to go through a series of error checking routines before the actual command execution begins. If an error is detected, a message is posted in the register displayed in the middle node. The following table lists possible error message codes and their meanings:

Error Code (dec)	Meaning
0001	Unknown *subfunction* specified in the top node
0010	ESI instruction has timed out (exceeded the time specified in the eighth register of the *subfunction parameter* table
0101	Error in the Read ASCII Message sequence
0102	Error in the Write ASCII Message sequence
0103	Error in the Get Data sequence
0104	Error in the Put Data sequence
1000	*length* specified in the bottom node is too small
1001	Nonzero value in both the 4*x* and 3*x* data offset parameters
1002	Zero value in both the 4*x* and 3*x* data offset parameters
1003	4*x* or 3*x* data offset parameter out of range
1004	4*x* or 3*x* data offset plus transfer count out of range
1005	3*x* data offset parameter set for Get Data
1101	Output registers from the offset parameter out of range
1102	Input registers from the offset parameter out of range
2001	Error reported from the ESI 062 module

Once the parameter error checking has completed without finding an error, the ESI module begins to execute the command sequence.

22.5.5 The Read ASCII Message Command

A Read ASCII command causes the ESI 062 module to read incoming data from one of its serial ports and store the data in internal variable data registers. The serial port number is specified in the tenth (ninth implied) register of the *subfunction parameters* table. The ASCII message number to be read is specified in the ninth (eighth implied) register of the *subfunction parameters* table. The received data is stored in the 16K variable data space in user-programmed formats.

When the top node of the ESI instruction is 1, the PLC invokes the module and causes it to execute one Read ASCII command followed by a sequence of Get Data commands (transferring up to 16,384 registers of data) from the module to the PLC.

Read ASCII Message Command Structure

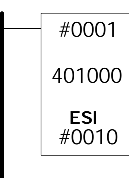
Word	Content (hex)	Meaning
0	01*PD*	*P* = port number (1 or 2); *D* = data count
1	*xxxx*	Starting register number, in the range 0 ... 3FFF
2	00*xx*	Message number, where *xx* is in the range 1 ... FF (1 ... 255 dec)
3 ... 11	Not used	

Read ASCII Message Response Structure

Word	Content (hex)	Meaning
0	01*PD*	Echoes command word 0
1	*xxxx*	Echoes starting register number from command word 1
2	00*xx*	Echoes message number from command word 2
3	*xxxx*	Data word 1
4	*xxxx*	Data word 2
...	...	...
11	*xxxx*	Module status (see section 22.5.10) or data word 9

A Comparative Read ASCII Message/Put Data Example

Below is an example of how an ESI loadable instruction can simplify your logic programming task in an ASCII read application. Assume that the 12-point bidirectional ESI 062 module has been I/O mapped to 400001 ... 400012 output registers and 300001 ... 300012 input registers. We want to read ASCII message #10 from port 1, then transfer four words of data to registers 400501 ... 400504 in the PLC.



The *subfunction parameter* table begin at register 401000 . Enter the following parameters in the table:

Register	Parameter Value	Description
401000	*nnnn*	ESI status register
401001	1	I/O mapped output starting register (400001)
401002	1	I/O mapped input starting register (300001)
401003	501	Starting register for the data transfer (400501)
401004	0	No 3*x* starting register for the data transfer
401005	100	Module start register
401006	4	Number of registers to transfer
401007	600	timeout = 60 s
401008	10	ASCII message number
401009	1	ASCII port number

With these parameters entered to the table, the ESI instruction will handle the read and data transfers automatically in one scan.

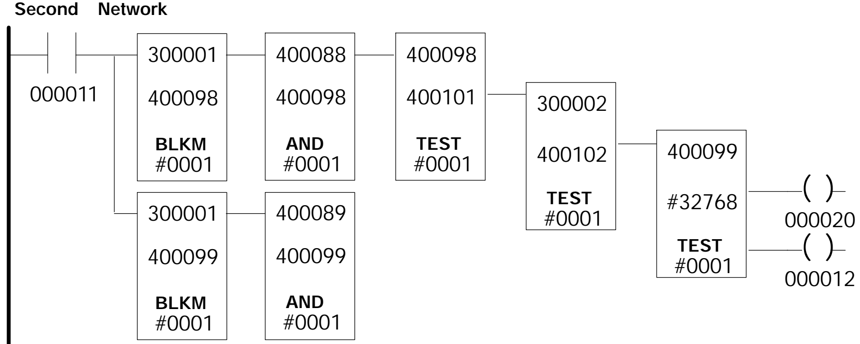
The same task could be accomplished in ladder logic without the ESI loadable, but it would require the following three networks to set up the command and transfer parameters, then copy the data. Registers 400101 ... 400112 are used as workspace for the output values. Registers 400201 ... 400212 are initial Read ASCII Message command values. Registers 400501 ... 400504 are the data space for the received data from the module.

First Network

000011
000011
000011
400201
400101
BLKM
#0012
400101
400001
BLKM
#0012

Register	Value (hex)	Description
400201	0114	Read ASCII Message command, Port 1, Four registers
400202	0064	Module's starting register
400203	*nnnn*	Not valid: data word 1
...	...	...
400212	*nnnn*	Not valid: data word 10

The first network starts up the Read ASCII Message command by turning ON coil 000011 forever. It moves the Read ASCII Message command into the workspace, then moves the workspace to the output registers for the module.



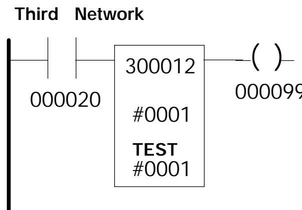
Register	Value (hex)	Description
400098	*nnnn*	Workspace for response word
400099	*nnnn*	Workspace for response word
400088	7FFF	Response word mask
400089	8000	Status word valid bit mask

As long as coil 000011 is ON, Read ASCII Message response word 0 in the input register is tested to make sure it is the same as command word 0 in the workspace. This is done by ANDing response word 0 in the input register with 7FFF hex to get rid of the status word valid bit (bit 15) in response word 0.

The module start register in the input register is also tested against the module start register in the workspace to make sure that are the same.

If both these tests show matches, test the status word valid bit in response word 0. To do this, AND response word 0 in the input register with 8000 hex to get rid of the echoed command word 0 information. If the ANDed result equals the status word valid bit, coil 000020 is turned ON indicating an error and/or status in the module status word.

If the ANDed result is not the status word valid bit, coil 000012 is turned ON indicating that the message is done and that you can start another command in the module.



If coil 000020 is ON, this third network will test the module status word for busy status. If the module is busy, do nothing. If the module status word is greater than 1 (busy), a detected error has been logged in the high byte and coil 000099 will be turned ON. At this point, you need to determine what the error is using some error-handling logic that you have developed.

22.5.6 Write ASCII Message

In a Write ASCII Message command, the ESI 062 module writes an ASCII message to one of its serial ports. The serial port number is specified in the tenth (ninth implied) register of the *subfunction parameters* table. The ASCII message number to be written is specified in the ninth (eighth implied) register of the *subfunction parameters* table.

When the top node of the ESI instruction is 2, the PLC invokes the module and causes it to execute one Write ASCII command. Before starting the Write command, subfunction 2 executes a sequence of Put Data transfers (transferring up to 16,384 registers of data) from the PLC to the module.

Write ASCII Message Command Structure

Word	Content (hex)	Meaning
0	02*PD*	*P* = port number (1 or 2); *D* = data count
1	*xxxx*	Starting register number, in the range 0 ... 3FFF
2	00*xx*	Message number, where *xx* is in the range 1 ... FF (1 ... 255 dec)
3	*xxxx*	Data word 1
4	*xxxx*	Data word 2
...	...	...
11	*xxxx*	Data word 9

Write ASCII Message Response Structure

Word	Content (hex)	Meaning
0	02*PD*	Echoes command word 0
1	*xxxx*	Echoes starting register number from command word 1
2	00*xx*	Echoes message number from command word 2
3	0000	
...	...	
10	0000	
11	*xxxx*	Module status (see section 22.5.10)

22.5.7 Get Data

A Get Data command transfers up to 10 registers of data from the ESI 062 module to the PLC each time the ESI instruction is solved in ladder logic. The total number of words to be read is specified in word 0 of the Get Data command structure (the *data count*). The data is returned in increments of 10 in words 2 ... 11 in the Get Data response structure.

If a sequence of Get Data commands is being executed in conjunction with a Read ASCII Message command (via subfunction 1), up to nine registers are transferred when the instruction is solved the first time. Additional data are returned in groups of ten registers on subsequent solves of the instruction until all the data has been transferred.

If there is an error condition to be reported (other than a command syntax error), it is reported in word 11 in the Get Data response structure. If the command has requested 10 registers and the error needs to be reported, only nine registers of data will be returned in words 2 ... 10, and word 11 will be used for error status.

Get Data Command Structure

Word	Content (hex)	Meaning
0	030*D*	*D* = data count
1	*xxxx*	Starting register number, in the range 0 ... 3FFF
2 ... 11	Not used	

Get Data Response Structure

Word	Content (hex)	Meaning
0	030*D*	Echoes command word 0
1	*xxxx*	Echoes starting register number from command word 1
2	*xxxx*	Data word 1
3	*xxxx*	Data word 2
...	...	...
11	*xxxx*	Module status (see section 22.5.10) or data word 10

Note: If the data count and starting register number that you specify are valid but some of the registers to be read are beyond the valid register range, only data from the registers in the valid range will be read. The data count returned in word 0 of the response structure will reflect the number of valid data registers returned, and an error code (1280 hex) will be returned in the module status word (word 11 in the response table).

22.5.8 Put Data (Subfunction 4)

A Put Data command writes up to 10 registers of data to the ESI 062 module from the PLC each time the ESI instruction is solved in ladder logic. The total number of words to be written is specified in word 0 of the Put Data command structure (the *data count*). The data is returned in increments of 10 in words 2 ... 11 in the Put Data command structure. The command is executed sequentially until command word 0 changes to another command other than Put Data (040D hex).

Put Data Command Structure

Word	Content (hex)	Meaning
0	040*D*	*D* = data count
1	*xxxx*	Starting register number, in the range 0 ... 3FFF
2	*xxxx*	Data word 1
3	*xxxx*	Data word 2
...	...	...
11	*xxxx*	Data word 10

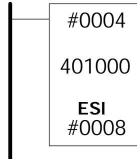
Put Data Response Structure

Word	Content (hex)	Meaning
0	040*D*	Echoes command word 0
1	*xxxx*	Echoes starting register number from command word 1
2	0000	
...	...	
10	0000	
11	*xxxx*	Module status (see section 22.5.10)

Note: If the data count and starting register number that you specify are valid but some of the registers to be written are beyond the valid register range, only data from the registers in the valid range will be written. The data count returned in word 0 of the response structure will reflect the number of valid data registers returned, and an error code (1280 hex) will be returned in the module status word (word 11 in the response table).

A Comparative Put Data Example

Below is an example of how an ESI loadable instruction can simplify your logic programming task in a Put Data application. Assume that the 12-point bidirectional ESI 062 module has been I/O mapped to 400001 ... 400012 output registers and 300001 ... 300012 input registers. We want to put 30 PLC data registers, starting at register 400501, to the ESI 062 module starting at location 100.



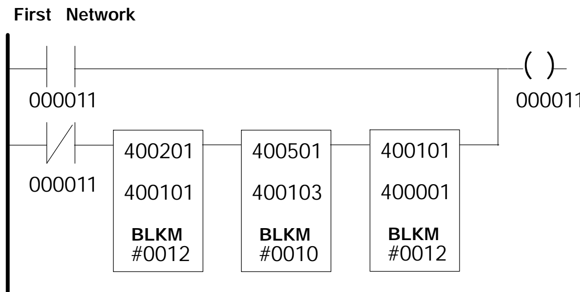
The *subfunction parameter* table begin at register 401000 . Enter the following parameters in the table:

Register	Parameter Value	Description
401000	*nnnn*	ESI status register
401001	1	I/O mapped output starting register (400001)
401002	1	I/O mapped input starting register (300001)
401003	501	Starting register for the data transfer (400501)
401004	0	No 3*x* starting register for the data transfer
401005	100	Module start register
401006	30	Number of registers to transfer
401007	0	timeout = never
401009	1	ASCII port number

With these parameters entered to the table, the ESI instruction will handle the data transfers automatically over three ESI logic solves.

The same task could be accomplished in ladder logic without the ESI loadable, but it would require the following four networks to set up the command and transfer parameters, then copy data multiple times until the operation is complete. Registers 400101 ... 400112 are used as workspace for the output values. Registers 400201 ... 400212 are initial Put Data command values. Registers 400501 ... 400530 are the data registers to be sent to the module.

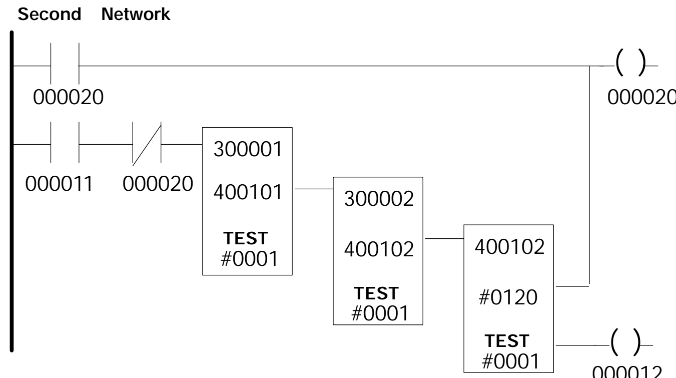
First Network



Register	Value (hex)	Description
400201	040A	Put Data command, 10 registers
400202	0064	Module's starting register
400203	*nnnn*	Not valid: data word 1
...	...	...
400212	*nnnn*	Not valid: data word 10

The first network starts up the transfer of the first 10 registers by turning ON coil 000011 forever. It moves the initial Put Data command into the workspace, moves the first 10 registers (400501 ... 400510) into

the workspace, and then moves the workspace to the output registers for the module.



As long as coil 000011 is ON and coil 000020 is OFF, Put Data response word 0 in the input register is tested to make sure it is the same as the command word in the workspace. The module start register in the input register is also tested to make sure it is the same as the module start register in the workspace.

If both these tests show matches, the current module start register is tested against what would be the module start register of the last Put Data command for this transfer. If the test shows that the current module start register is greater than or equal to the last Put Data command, coil 000020 goes ON indicating that the transfer is done. If the test shows that the current module start register is less than the last Put Data command, coil 000012 indicating that the next 10 registers should be transferred.

Third Network

000012

400102
#0100
TEST
#0001

400102
#0110
TEST
#0001

400511
400103
BLKM
#0010

400521
400103
BLKM
#0010

As long as coil 000012 is ON, there is more data to be transferred. The module start register needs to be tested from the last command solve to determine which set of 10 registers to transfer next. For example, if the

last command started with module register 400110, then the module start register for this command is 400120.

Fourth Network

000012
#0010
400102
AD16
400102
400101
400001
BLKM
#0012

As long as coil 000012 is ON, add 10 to the module start register value in the workspace and move the workspace to the output registers for the module to start the next transfer of 10 registers.

22.5.9 Abort (Middle Input ON)

When the middle input to the ESI instruction is powered ON, the instruction aborts a running ASCII Read or Write message. The serial port buffers of the module are not affected by the Abort, only the message that is currently running.

Abort Command Structure

Word	Content (hex)
0	0900
2 ... 11	not used

Abort Response Structure

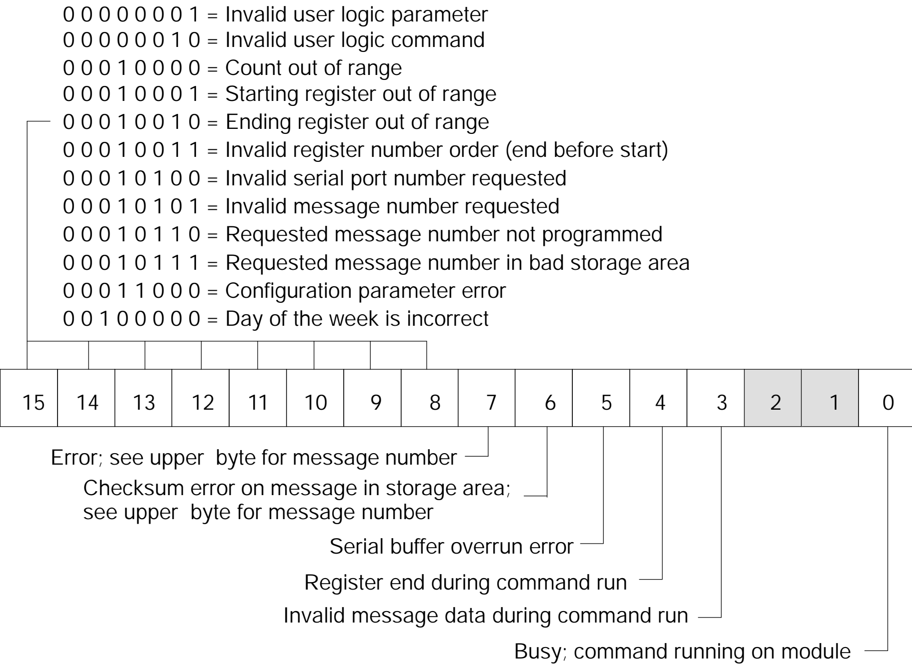
Word	Content (hex)	Meaning
0	0900	Echoes command word 0
2	0000	
...	...	
10	0000	
11	*xxxx*	Module status (see section 22.5.10)

22.5.10 Module Status Word

The module status word (word 11 in the response structure) contains valid module status information when bit 15 of word 0 (in the response structure) is set. The state of this bit can be used to distinguish

whether word 11 in the response structure is being used for data or status.

The low byte of the module status word defines status conditions. The high byte defines module status error conditions (when bit 7 is set).



22.6 MBUS

The S975 Modbus II Interface option modules use two loadable function blocks—MBUS and PEER (see page 496). MBUS is used to initiate a single transaction with another device on the Modbus II network. In an MBUS transaction, you are able to read or write discrete or register data.

PLCs on a Modbus II network can handle up to 16 transactions simultaneously. Transactions include incoming (unsolicited) messages as well as outgoing messages. Thus, the number of message initiations a PLC can manage at any time is 16 *# of incoming messages*.

A transaction cannot be initiated unless the S975 has enough resources for the entire transaction to be performed. Once a transaction has been initiated, it runs until a reply is received, an error is detected, or a timeout occurs. A second transaction cannot be started in the same scan that the previous transaction completes unless the middle input is ON. A second transaction cannot be initiated by the same MBUS instruction until the first transaction has completed.

22.6.1 Characteristics

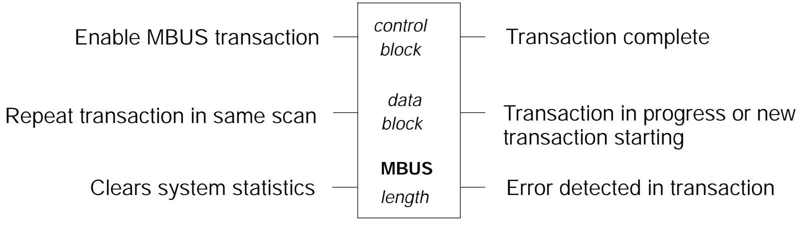
Size
Three nodes high

PLC Compatibility
Available as a loadable in PLC types that support an S975 Modbus II module

Opcode
1F hex (default)

22.6.2 Representation

Block Structure



Top Node Content

The 4*x* register entered in the top node is the first of seven contiguous registers in the MBUS *control block* :

Register	Function	
Displayed	Address of destination device (range: 0 ... 246)	
4*x* + First implied		
Second implied	Function code for requested action:	
	01	*Read discretes*
	02	*Read registers*
	03	*Write discrete outputs*
	04	*Write register outputs*
	255	*Get system statistics*
Third implied	Discrete or register reference type:	
	0	Discrete output (0*x*)
	1	Discrete input (1*x*)
	3	Input register (3*x*)
	4	Holding register (4*x*)
Fourth implied	Reference number—e.g., if you placed a 4 in the third implied register and you place a 23 in this register, the reference will be holding register 40023	
Fifth implied	Number of words of discrete or register references to be read or written; the length limits are:	
	Read register	251 registers
	Write register	249 registers
	Read coils	7,848 discretes
	Write coils	7,800 discretes
Sixth implied	Time allowed for a transaction to be completed before an error is declared; expressed as a multiple of 10 ms—e.g., *100* indicates 1,000 ms; the default timeout is 250 ms	

Middle Node Content

The middle node is the first 4*x* register in a *data block* to be transmitted or received in the MBUS transaction.

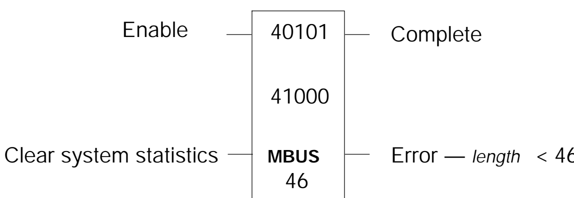
Bottom Node Content

The number of words reserved for the *data block* is entered as a constant value in the bottom node. This number does not imply a data transaction length, but it can restrict the maximum allowable number of register or discrete references to be read or written in the transaction. The maximum number of words that may be used in the specified transaction is:

- 251 for reading registers (one register/word)
- 249 for writing registers (one register/word)
- 490 for reading discretes using 24-bit CPUs: 255 for reading discretes using 16-bit CPUs (up to 16 discretes/word)
- 487 for writing discretes using 24-bit CPUs; 255 for reading discretes using 16-bit CPUs (up to 16 discretes/word)

22.6.3 The MBUS *Get Statistics* Function

Issuing function code 255 in the second implied register of the MBUS *control block* obtains a copy of the Modbus II local statistics—a series of 46 contiguous register locations where data describing error and system conditions is stored. To use MBUS for a *get statistics* operation, set the *length* in the bottom node to 46—a *length* < 46 returns an error (the bottom output will go ON), and a *length* > 46 reserves extra registers that cannot be used. For example:



Register 40101 is the first register in the MBUS control block, making register 40103 the control register that defines the MBUS function code. By entering a value of 255 in register 40103, you implement a *get*

statistics function. Registers 41000 ... 41045 are then filled with the following system statistics:

Statistic	Register	Content
Token bus controller (TBC)	41000	Number of tokens passed by this station
	41001	Number of tokens sent by this station
	41002	Number of time the TBC has failed to pass token and has not found a successor
	41003	Number of times the station has had to look for a new successor
Software-maintained receive statistics	41004	TBC-detected error frames
	41005	Invalid request with response frames
	41006	Applications message too long
	41007	Media access control (MAC) address out of range
	41008	Duplicate application frames
	41009	Unsupported logical link control (LLC) message types
	41010	Unsupported LLC address
TBC-maintained error counters	41011	Receive noise bursts (no start delimiter)
	41012	Frame check sequence errors
	41013	E-bit error in end delimiter
	41014	Fragmented frames received (start delimiter not followed by end delimiter)
	41015	Receive frames too long
	41016	Discarded frames because there is no receive buffer
	41017	Receive overruns
	41018	Token pass failures
Software-maintained transmit errors	41019	Retries on request with response frames
	41020	All retries performed and no response received from unit
Software-maintained receive errors	41021	Bad transmit request
	41022	Negative transmit confirmation
User logic transaction errors	41023	Message sent but no application response
	41024	Invalid MBUS/PEER logic
Manufacturing message format standard	41025	Command not executable
	41026	Data not available
(MMFS) errors	41027	Device not available
	41028	Function not implemented
	41029	Request not recognized
	41030	Syntax error
	41031	Unspecified error
	41032	Data request out of bounds
	41033	Request contains invalid 984 address
	41034	Request contains invalid data type
	41035	None of the above

Background statistics	41036	Invalid MBUS/PEER request
	41037	Number of unsupported MMFS message types received
	41038	Unexpected response or response received after time-out
	41039	Duplicate application responses received
	41040	Response from unspecified device
	41041	Number of responses buffered to be processed (in the least significant byte); number of MBUS/PEER requests to be processed (in the most significant byte)
	41042	Number of received requests to be processed (in the least significant byte); number of transactions in process (in the most significant byte)
	41043	S975 scan time in 10 s increments
Software revision	41044	Version level of fixed software (PROMs): major version number in most significant byte; minor version number in least significant byte
	41045	Version of loadable software(EEPROMs): major version number in most significant byte; minor version number in least significant byte

22.7 PEER

The S975 Modbus II Interface option modules use two loadable function blocks—MBUS and PEER (see page 491). The PEER instruction can initiate identical message transactions with as many as 16 devices on Modbus II at one time. In a PEER transaction, you may only write register data.

22.7.1 Characteristics

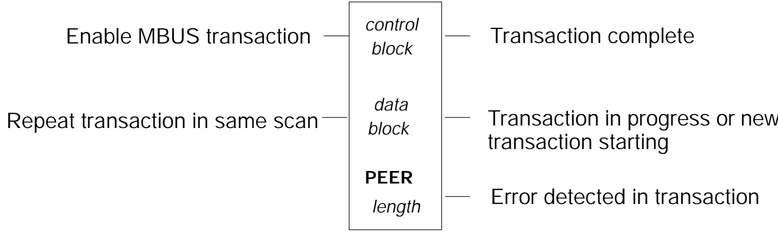
Size
Three nodes high

PLC Compatibility
Available as a loadable in PLC types that support an S975 Modbus II module

Opcode
3F hex (default)

22.7.2 Representation

Block Structure



Top Node Content
The 4*x* register entered in the top node is the first of 19 contiguous registers in the PEER control block:

Register	Function
Displayed	Indicates the status of the transactions at each device, the leftmost bit being the status of device #1 and the rightmost bit the status of device #16: 0 = OK, 1 = transaction error
First implied	Defines the reference to the first 4*x* register to be written to in the receiving device; a 0 in this field is an invalid value and will produce an error (the bottom output will go ON)
Second implied	Time allowed for a transaction to be completed before an error is declared; expressed as a multiple of 10 ms—e.g., *100* indicates 1,000 ms; the default timeout is 250 ms
Third implied	The Modbus port 3 address of the first of the receiving devices; address range: 1 ... 255 (0 = no transaction requested)
Fourth implied	The Modbus port 3 address of the second of the receiving devices; address range: 1 ... 255 (0 = no transaction requested)
. . .	. . .
18th implied	The Modbus port 3 address of the 16th of the receiving devices (address range: 1 ... 255)

Middle Node Content

The 4*x* register entered in the middle node is the first register in a *data block* to be transmitted by the PEER function.

Bottom Node Content

The integer value entered in the bottom node is the *length* —i.e., the number of holding registers—of the *data block* . The *length* can range from 1 ... 249.

22.8 Custom Loadables

The Custom Loadable software package (SW-AP98-GDA) allows you to design your own ladder logic instructions. The operational unit for the custom loadable support software is a three-node instruction block, FN*xx*, where *xx* is an integer in the range 01 ... 99. Up to 99 unique FN*xx* blocks can be created. Within each block, you can design a large number of subfunctions—up to 8192.

22.8.1 Programming Environment

This development package is for experienced C or Assembly Language programmers, and is outside the standard ladder logic programming environment. Custom loadable instructions may be developed on IBM-AT or compatible computers running MS-DOS, Rev. 3.2 or greater. The resulting blocks may be downloaded to a standard disk-based programming panel and used in ladder logic programs.

Creating a Subfunction Library

Each subfunction built into an FN*xx* loadable is comparable to a standard three-node DX function. It requires a certain amount of user logic memory upon installation.

A large number of subfunctions can be written and stored in a subfunction library in the development environment. The size of this library can be far in excess of available memory in the target PLC. Only particular subfunctions for immediate use can be pulled from the library and compiled in the FN*xx* instruction as it is built. The PLC needs enough extra memory to support only the installed subfunction(s).

Naming Subfunctions

In addition to an individual ID number, each subfunction in a customized block must be assigned a name. The name may contain from one to four alphabetical characters, either upper or lower case. The programmer creates a separate file—the subfunction list file—where a subfunction ID number is linked to its subfunction name, and the name can be used by utility tools to access and display the subfunction and its specific characteristics.

Assigning Opcodes to Functions

Each FN*xx* function must be assigned an opcode in the valid range of Modicon opcodes that is not used by any other instruction currently installed in the PLC. If you have designed multiple custom loadables but intend to download only some of them together at any one time,

then you need only assign as many unique opcodes as there are custom functions downloaded at any one time. However, you must inform the user how to change opcodes using the *lodutil* utility as one function is withdrawn and replaced by another. The fact that you are able to create so many subfunctions within one function allows you to work around the finite limit of available opcodes.

22.8.2 Characteristics

Size

Three nodes high

PLC Compatibility

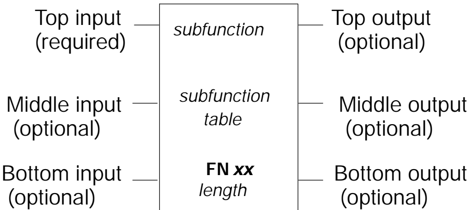
Available as a loadable in all PLC types except the 984A/B/X Chassis Mounts

Opcode

5F hex (default)

22.8.3 Representation

Block Structure



Inputs and Outputs

The input to the top node, which will be used to initiate the instruction, must be implemented. The remaining two inputs and all three outputs may or may not be used according to your application requirements.

Top Node Content

The top node can use either a 4*x* holding register or a constant value to identify a *subfunction ID number*. Valid ID numbers range from 0 ... 9999.

As many as 8192 different subfunctions may be designed within a block. When multiple subfunctions are designed within an FN*xx* block,

each subfunction within the block must have a unique ID number, but those numbers do not have to be consecutive.

Middle Node Content

The middle node displays the first 4*x* register in a table of registers to be used by the subfunction. The table may be used to pass data to the subfunction and store results. The table format may be customized for your requirements, and each subfunction developed within the function block may have its own format.

Bottom Node Content

The bottom node defines the function number, which may range from FN01 ... FN99, and uses an integer value to define the *length* —i.e., the number of 4*x* registers—of the subfunction table. The *length* range can range from 1 ... 255 in a 16-bit CPU and from 1 ... 999 in a 24-bit CPU.

22.9 The EARS Loadable

The EARS block is loaded to a PLC used in an alarm/event recording system. An EARS system requires that the PLC work in conjunction with a man-machine interface (MMI) host device that runs a special off-line software package. The PLC monitors a specified group of events for changes in state and logs change data into a buffer. The data is then removed by the host over a high speed network such as Modbus Plus. The two devices comply with a defined handshake protocol that ensures that all data detected by the PLC is accurately represented in the host.

22.9.1 PLC Functions in an Event/Alarm Recording System

When a PLC is employed in an EARS environment, it is set up to maintain and monitor two tables of 4*x* registers, one containing the *current* state of a set of user-defined events and one containing the *history* of the most recent state of these events. Event states are stored as bit representations in the 4*x* registers—a bit value of 1 signifying an ON state and a bit value of 0 signifying an OFF state. Each table can contain up to 62 registers, allowing you to monitor the states of up to 992 events.

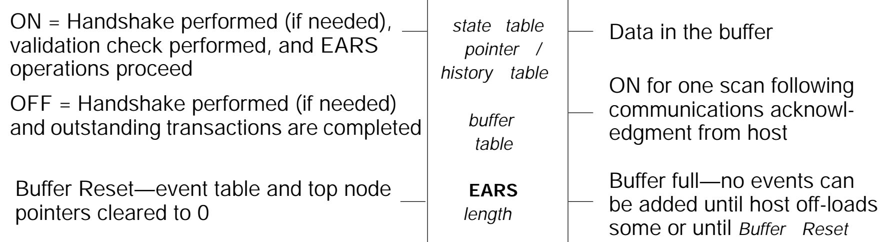
When the PLC detects a change between the current state bit and the history bit for an event, the EARS instruction prepares a two-word message and places it in a buffer where they can be off-loaded to a host MMI. This message contains:

- A time stamp representing the time span from midnight to 24:00 hours in tenths of a second
- A transition flag indicating that the event is either a positive or negative transition with respect to the event state
- A number indicating which event has occurred

22.9.2 Host ↔PLC Interaction

The host MMI device must be able to read and write PLC data registers via the Modbus protocol. A handshake protocol maintains integrity between the host and the circular buffer running in the PLC. This enables the host to receive events asynchronously from the buffer at a speed suitable to the host while the PLC detects event changes and load the buffer at its faster scan rate.

22.9.3 The EARS Block



Top Node Content

The 4*x* register entered in the top node is the first of 64 contiguous registers. The first two registers contain values that specify the location and size of the current state table:

Register	Content
Displayed	Indirect pointer to the current state table—e.g., if the register contains a value of 5, then the state table begins at register 40005; the indirect pointer register must be hard-coded by the programmer
First implied	Contains a value in the range 1 ... 62 that specifies the number of registers in the current state table; this value must be hard-coded by the programmer
Second implied	First register of the history table, and the remaining registers allocated to the top node may be used in the table as required; the history table can provide monitoring for as many as 992 contiguous events (if 16 bits in all the 62 available registers are used)

The the remaining 61 registers are available to store history data. If all the remaining registers are not required for the history table, they may be used elsewhere in the program for other purposes, but they will still be found (by a Modbus search) in the top node of the EARS block.

Middle Node Content

The 4*x* register entered in the middle node is the first in a series of contiguous registers uses as a *buffer table* . The first five registers are used as follows, and the rest contain the circular buffer. The circular buffer uses an even number of registers in the range 2 ... 100:

<table>
<tr><th>Register</th><th>Content</th></tr>
<tr><td>Displayed</td><td>A value that defines the maximum number of registers the circular buffer may occupy</td></tr>
<tr><td>First implied</td><td>The Q_take pointer—the pointer to the next register where the host will go to remove data</td></tr>
<tr><td>Second implied</td><td>The low byte contains the Q_put pointer—the pointer to the register in the circular buffer where the EARS block will begin to place the next state-change data. The high byte contains the last transaction number received.</td></tr>
<tr><td>Third implied</td><td>The Q+count —a value indicating the number of words currently in the circular buffer</td></tr>
<tr><td>Fourth implied</td><td>Status/error codes:
<table>
<tr><th>Code</th><th>Condition</th></tr>
<tr><td>1</td><td>Invalid block length</td></tr>
<tr><td>2</td><td>Invalid clock request</td></tr>
<tr><td>3</td><td>Invalid clock configuration</td></tr>
<tr><td>4</td><td>Invalid state length</td></tr>
<tr><td>5</td><td>Invalid queue put</td></tr>
<tr><td>6</td><td>Invalid queue take</td></tr>
<tr><td>7</td><td>Invalid state</td></tr>
<tr><td>8</td><td>Invalid queue count</td></tr>
<tr><td>9</td><td>Invalid sequence number</td></tr>
<tr><td>10</td><td>Count removed</td></tr>
<tr><td>255</td><td>Bad clock chip</td></tr>
</table></td></tr>
<tr><td>Fifth implied</td><td>First register in the circular buffer where event-change data are stored; each change in event status produces two contiguous registers:</td></tr>
</table>

Event Data Register 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Event Number (1 ... 992)

0 = Negative Transition Event Type
1 = Positive Transition Event Type

Four Most Significant Bits of Event Time Stamp

Event Data Register 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Sixteen Least Significant Bits of Event Time Stamp

The time stamp is encoded in 20 bits as a binary weighted value that represents the time in an increment of 0.1 s, starting from midnight of the day on which the status change was detected:

1 hour = 3,600 seconds = 36,000 tenths of a second, and
24 hours = 86,400 seconds = 864,000 tenths of a second

The following table shows binary weighted values for the time stamp, where *n* is the relative bit position in the 20-bit time scheme:

Event Data Register 1				Event Data Register 2															
19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2^n	*n*	2^n	*n*	2^n	*n*
1	0	256	8	65536	16
2	1	512	9	131072	17
4	2	1024	10	262144	18
8	3	2048	11	524288	19
16	4	4096	12		
32	5	8192	13		
64	6	16384	14		
128	7	32768	15		

Note: The real time clock in the chassis mount controllers has a tenth-of-a-second resolution, but the other 984s have real time clock chips that resolve only to a second. An algorithm is used in EARS to provide a best estimate of tenth-of-a-second resolution—it is accurate in the relative time intervals between events, but it may vary slightly from the real time clock.

Bottom Node Content

The integer value entered in the bottom node is the *length* —i.e., actual number of registers allocated for the circular buffer. The *length* can range from 2 ... 100. Each event requires two registers for data storage. Therefore, if you wish to trap up to 25 events at any given time in the buffer, assign a *length* of 50 in the bottom node.

22.10 EUCA

The use of ladder logic to convert binary-expressed analog data into decimal units can be memory-intensive and scan-time intensive operation. The Engineering Unit Conversion and Alarms (EUCA) loadable is designed to eliminate the need for extra user logic normally required for these conversions. EUCA scales 12 bits of binary data (representing analog signals or other variables) into engineering units that are readily usable for display, data logging, or alarm generation.

Using Y = mX + b linear conversion, binary values between 0 ... 4095 are converted to a scaled process variable (SPV). The SPV is expressed in engineering units in the range 0 ... 9999.

One EUCA instruction can perform up to four separate engineering unit conversions. It also provides four levels of alarm checking on each of the four conversions:

- High absolute (HA)
- High warning (HW)
- Low warning (LW)
- Low absolute (LA)

22.10.1 Characteristics

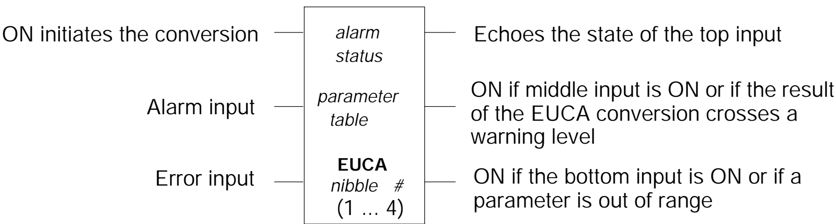
Size
Three nodes high

PLC Compatibility
- Available as a loadable in the E984-685 and E984-785 Slot Mount PLCs and in the Quantum Automation Series PLCs
- Not available in other PLC types

Opcode
01F hex (default)

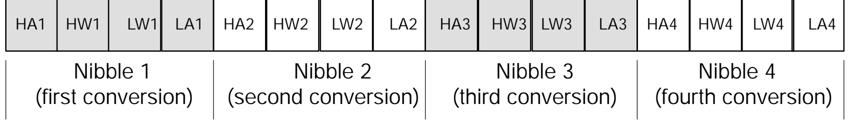
22.10.2 Representation

Block Structure



Top Node Content

The 4*x* register entered in the top node displays the *alarm status* for as many as four EUCA conversions, which can be performed by the instruction. The register in segmented into four four-bit nibbles. Each four-bit nibble represents the four possible alarm conditions for an individual EUCA conversion. The most significant nibble represents the first conversion, and the least significant nibble represents the fourth conversion:



At any given time a nibble selected by the value in the bottom node displays one alarm condition:

- An HA alarm is set when the SPV exceeds the user-defined high alarm value expressed in engineering units
- An HW alarm is set when SPV exceeds a user-defined high warning value expressed in engineering units
- An LW alarm is set when SPV is less than a user-defined low warning value expressed in engineering units
- An LA alarm is set when SPV is less than a user-defined low alarm value expressed in engineering units

Only one alarm condition can exist in any EUCA conversion at any given time. If the SPV exceeds the high warning level the HW bit will be set. If the HA is exceeded, the HW bit is cleared and the HA bit is set. The alarm bit will not change after returning to a less severe condition until the deadband (DB) area has also been exited.

EUCA Middle Node Description

The 4*x* register entered in the middle node is the first of nine contiguous holding registers in the EUCA *parameter table* :

Register	Content	Range
Displayed	Binary value input by the user	0 ... 4095
First implied	SPV calculated by the EUCA block	
Second implied	High engineering unit (HEU), maximum SPV required and set by the user (top of the scale)	LEU < HEU ≤ 9999
Third implied	Low engineering unit (LEU), minimum SPV required and set by the user (bottom end of the scale)	0 ≤ LEU < HEU
Fourth implied	DB area in SPV units, below HA levels and above LA levels that must be crossed before the *alarm status* bit will reset	0 ≤ DB < (HEU LEU)
Fifth implied	HA alarm value in SPV units	HW < HA≤ HEU
Sixth implied	HW alarm value in SPV units	LW < HW < HA
Seventh implied	LW alarm value in SPV units	LA < LW < HW
Eighth implied	LA alarm value in SPV units	LEU ≤LA < LW

Note: An error is generated if any value is out of the range defined above.

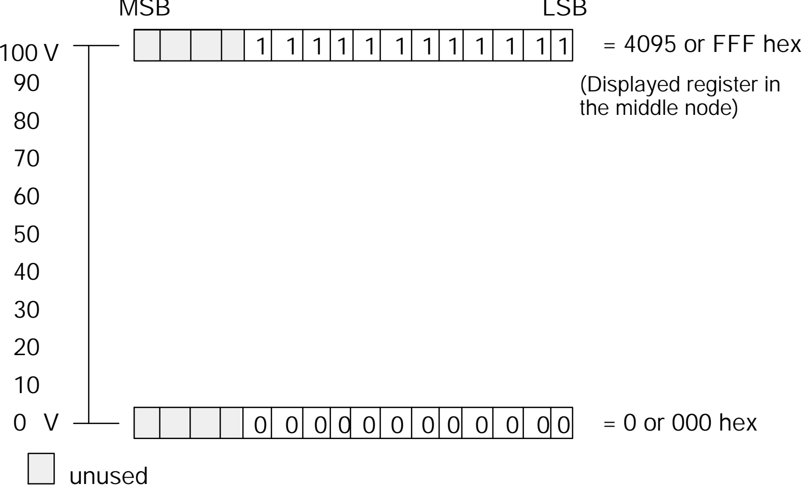
EUCA Bottom Node Description

The integer value entered in the bottom node indicates which one of the four nibbles in the *alarm status* register to use.

22.10.3 A EUCA Example

This example demonstrates the principles of EUCA operation. The binary value is manually input in the displayed register in the middle node, and the result is visually available in the SPV register (the first implied register in the middle node).

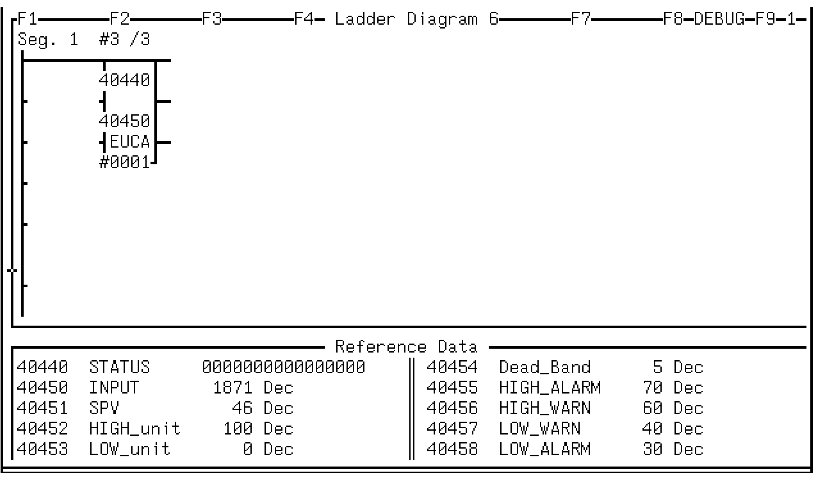
The illustration below shows an input range equivalent of a 0 ... 100 V measure, corresponding to the whole binary 12-bit range:



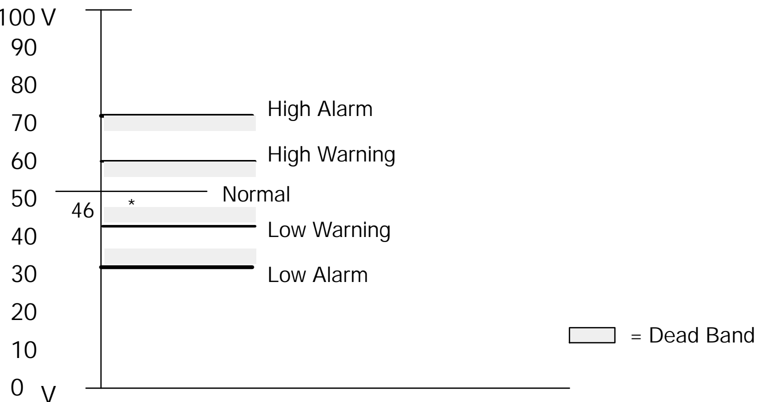
A range of 0 ... 100 V establishes 50 V for nominal operation. EUCA provides a margin on the nominal side of both warning and alarm levels (deadband). If an alarm threshold is exceeded, the alarm bit becomes active and stays active until the signal becomes greater (or less) than the DB setting—5 V in this example.

Setup

Programming the EUCA block is accomplished by selecting the EUCA loadable and writing in the data as illustrated in the Modsoft screen below:

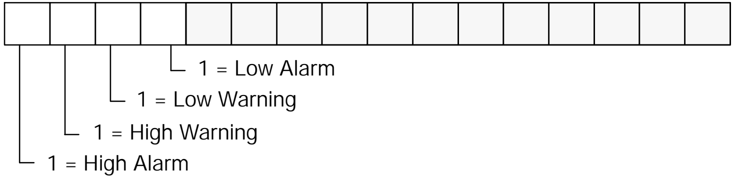


The nine middle-node registers are set using the reference editor. DB is 5 V followed by 10 V increments of high and low warning. The actual high and low alarm is set at 20 V above and below nominal. On a graph, the example looks like this:



Note: The example value shows a decimal 46, which is in the normal range. No alarm is set—i.e., register 40440 = 0.

You can now verify the instruction in a running PLC by entering values in register 40450 that fall into the defined ranges. The verification is done by observing the bit change in register 40440 where:



22.10.4 Example 2

If the input of 0 ... 4095 indicates the speed of a drive system of 0 ... 5000 rpm, you could set up a EUCA instruction as follows:

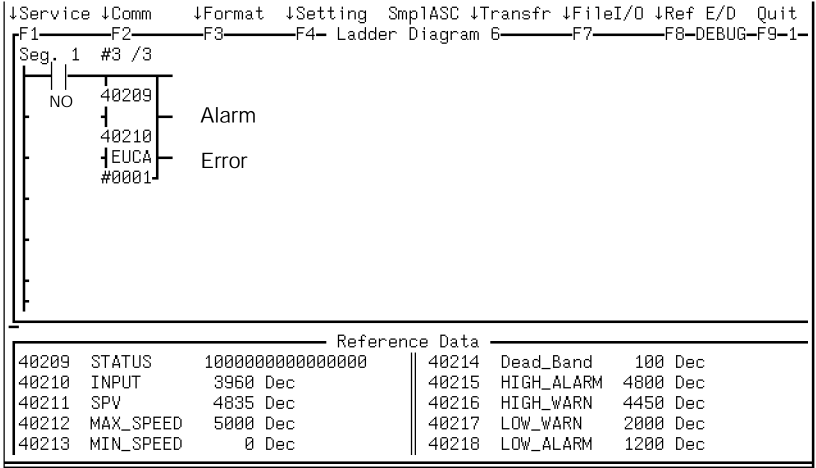
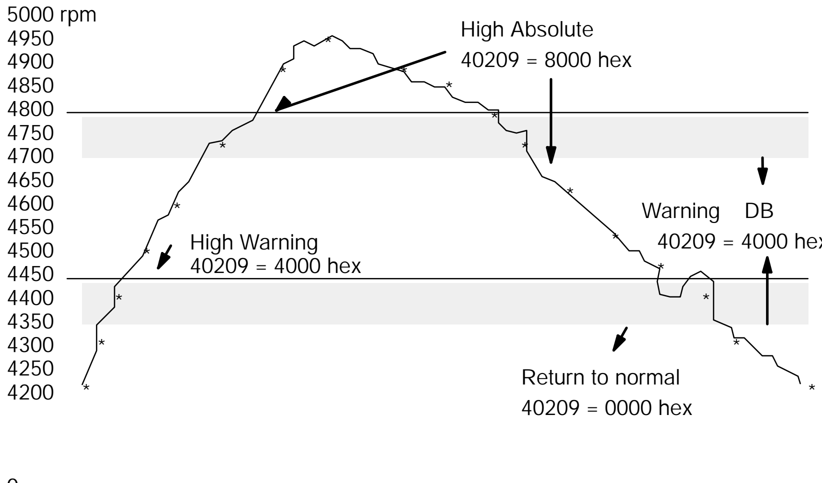


Figure 1

The binary value in 40210 results in an SPV of 4835 decimal, which exceeds the high absolute alarm level, sets the HA bit in 40209, and powers the EUCA alarm node.

Maximum Speed	5,000 rpm
Minimum Speed	0 rpm
DB	100 rpm
HA Alarm	4,800 rpm
HW Alarm	4,450 rpm
LW Alarm	2,000 rpm
LA Alarm	1,200 rpm

The N.O. contact is used to suppress alarm checks when the drive system is shutdown, or during initial start up allowing the system to get above the Low alarm RPM level.



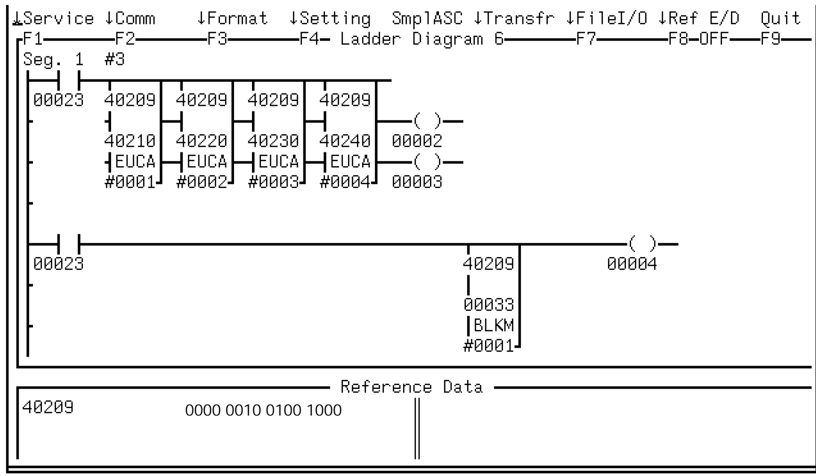
Varying the binary value in register 40210 would cause the bits in nibble 1 of register 40209 to correspond with the changes illustrated above. The DB becomes effective when the alarm or warning has been set—then the signal falls into the DB zone.

The alarm is maintained, thus taking what would be a switch chatter condition out of a marginal signal level. This point is exemplified in the chart above, where after setting the HA alarm and returning to the warning level at 4700 the signal crosses in and out of DB at the warning level (4450) but the warning bit in 40209 stays ON.

The same action would be seen if the signal were generated through the low settings.

22.10.5 Example 3

You can chain up to four EUCA conversions together to make one *alarm status* register. Each conversion writes to the nibble defined in the block bottom node. In the program example below, each EUCA block writes it‘s status (based on the table values for that block) into a four bit (nibble) of the status register 40209.



The status register can then be transferred using a BLKM instruction to a group of discretes wired to illuminate lamps in an alarm enunciator panel.

As you observe the status content of register 40209 you see; no alarm in block 1, an LW alarm in block 2, an HW alarm in Block 3, and an HA alarm in block 4.

The alarm conditions for the four blocks can be represented with the following table settings:

	Conversion 1	Conversion 2	Conversion 3	Conversion 4
Input	40210 = 2048	40220 = 1220	40230 = 3022	40240 = 3920
Scaled #	40211 = 2501	40221 = 1124	40231 = 7379	40241 = 0770
HEU	40212 = 5000	40222 = 3300	40232 = 9999	40242 = 0800
LEU	40213 = 0000	40223 = 0200	40233 = 0000	40243 = 0100
DB	40214 = 0015	40224 = 0022	40234 = 0100	40244 = 0006
Hi Alarm	40215 = 4000	40225 = 2900	40235 = 8090	40245 = 0768
Hi Warn	40216 = 3500	40226 = 2300	40236 = 7100	40246 = 0680
Lo Warn	40217 = 2000	40227 = 1200	40237 = 3200	40247 = 0280
Lo Alarm	40218 = 1200	40228 = 0430	40238 = 0992	40248 = 0230

Appendix A
Optimizing RIO Performance with the Segment Scheduler

- Scan Time
- How to Measure Scan Time
- Maximizing Throughput
- The Order of Solve
- Using the Segment Scheduler to Improve Critical I/O Throughput
- Using the Segment Scheduler to Improve System Performance
- Using the Segment Scheduler to Improve Comm Port Servicing
- Sweep Functions

A.1 Scan Time

The time it takes the PLC to solve the logic program and update the physical system is called *scan time*. It comprises the time it takes the PLC to:

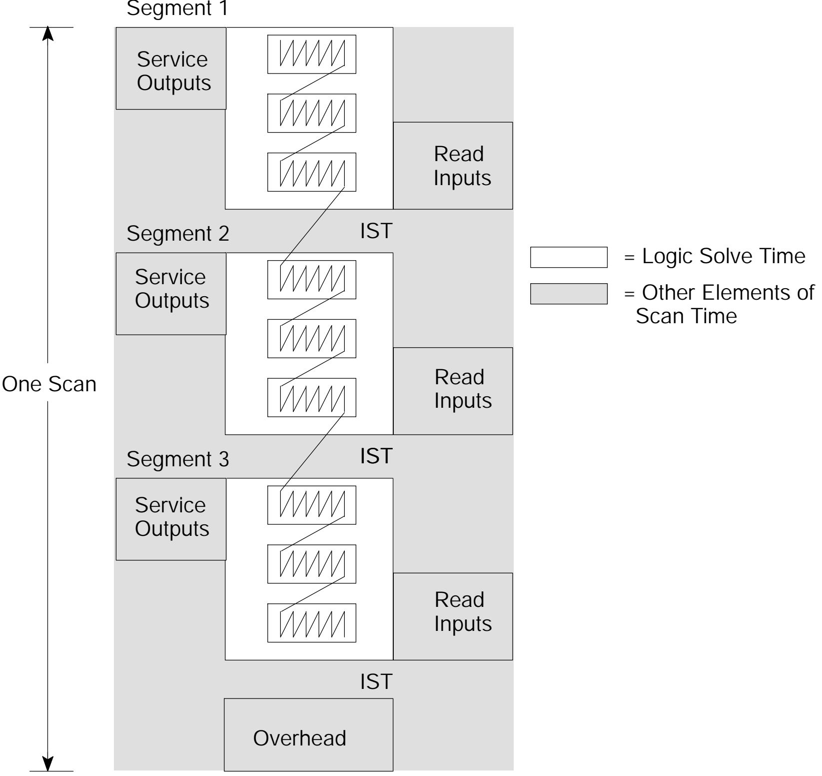
- Solve all scheduled logic—i.e., *logic solve time*
- Service the I/O drops
- Service communication ports and option processors
- Execute intersegment transfer (IST) and system diagnostics

A.1.1 Logic Solve Time

Logic solve time is the time it takes the CPU to solve the elements and instructions used in the logic program. It is a part of the total scan time that is independent of I/O service time and system overhead time. Logic solve time is measured in ms/Kwords of user logic. Various PLC models have different logic solve times, as shown below:

Logic Solve Time	PLC Models	PLC Types
0.75 ms/Kwords	984A, 984B, 984X	Chassis-mount
1.0 ms/Kwords	E984-685/-785, L984-785	Slot-mount
	CPU11302, CPU11303, CPU21304	Quantum Series
1.5 ms/Kwords	AT-984, MC-984	Host-based
	0984-780/-785	Slot-mount
2.0 ms/Kwords	Q984	Host-based
	0984-685	Slot-mount
2.5 ms/Kwords	110CPU51*x* and 110CPU61*x*	Micro
3.0 ms/Kwords	984-385, 984-485, 984-680	Slot-mount
4.25 ms/Kwords	984-A12*x*, 984-A13*x*, 984-A14*x*	Compact
	110CPU311 and 110CPU411	Micro
5.0 ms/Kwords	984-380/-381, 984-480	Slot-mount

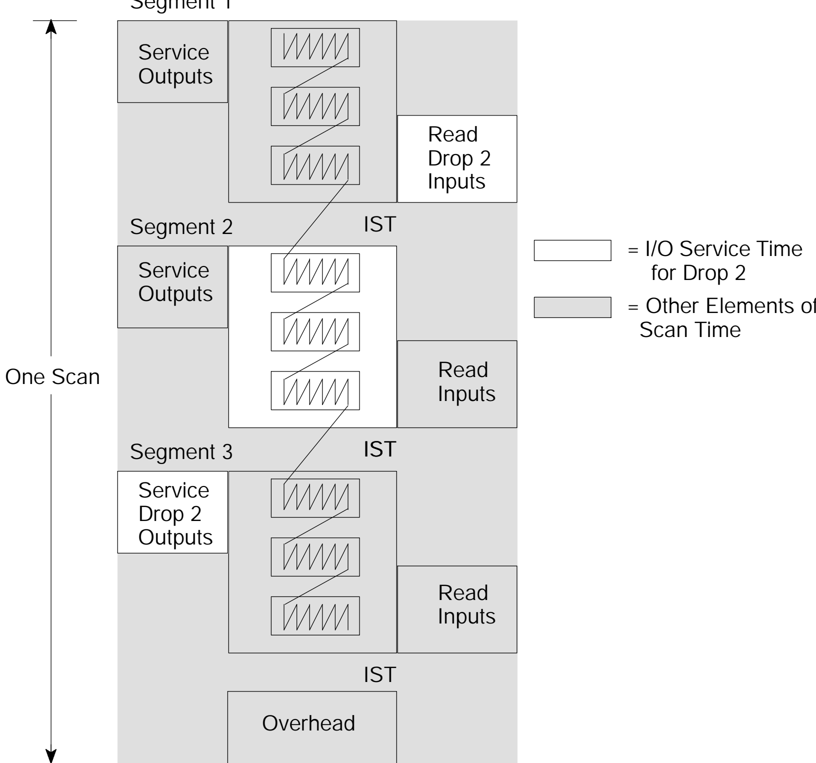
The following illustration shows how logic solve time fits in the overall scan time function:



A.1.2 Servicing I/O

In order to handle system throughput efficiently, the PLC coordinates the solution of logic segments via its CPU and the servicing of I/O drops via its I/O processor. Typically a logic segment is coordinated with a particular I/O drop—for example, the logic networks in segment 2 correspond to the real-world I/O points at drop 2. Inputs are read during the previous segment and outputs are written during the subsequent segment.

This method of I/O servicing assures that the most recent input status is available for logic solve and that outputs are written as soon as possible after logic solve. It ensures predictability between the PLC and the process it is controlling.

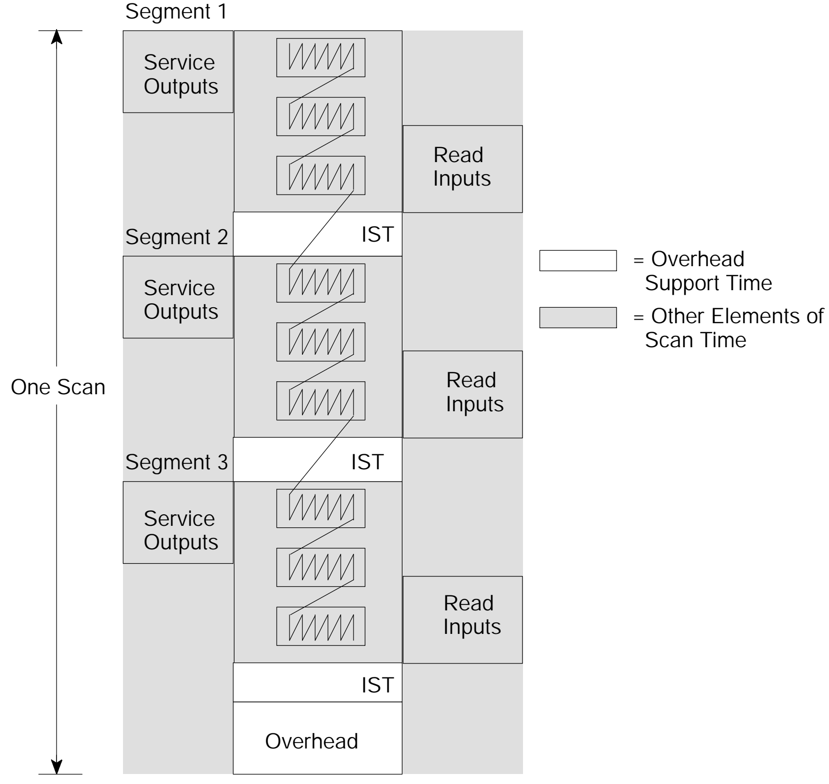


A.1.3 Overhead

An intersegment transfer (IST) occurs between each segment. At this time, the I/O processor and the state RAM exchange data; previous inputs are transferred to state RAM and the next outputs are transferred to the I/O processor. The logic scan and I/O servicing for each segment are coordinated in this fashion. Using direct memory access (DMA), ISTs typically take less than 1 ms/segment.

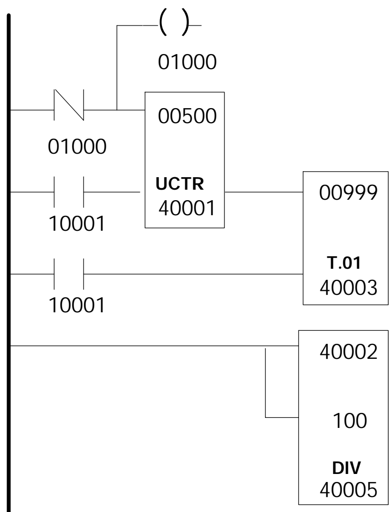
At the end of each scan, input messages to the Modbus communication ports are serviced. The maximum time allotted for comm port servicing is 2.5 ms/scan; typical servicing times are less than 1 ms/scan. If the PLC is using any option processors (C986 Coprocessors or D908 Distributed Communications Processors), they are also serviced at the end of each scan and typically require less than 1 ms/scan.

System diagnostics take from 1 ... 2 ms/scan to run, depending on PLC type.


Segment 1
Service Outputs
Read Inputs
IST
Segment 2
Service Outputs
Read Inputs
IST
Segment 3
Service Outputs
Read Inputs
IST
Overhead
One Scan
= Overhead Support Time
= Other Elements of Scan Time

A.2 How to Measure Scan Time

The following ladder logic circuit can be used in your application program to evaluate system scan time:



The up-counter counts 1000 scans as it transitions 500 times. When the counter has transitioned 500 times, the T.01 timer turns OFF and stores the number of hundredths of seconds it has taken for the counter to transition 500 times (1000 scans) in register 40003.

The value stored in 40002/40003 in the DIV block is then divided by 100 and the result—which represents logic solve time in ms—is stored in register 40005.

Note: 10001 is controlled via a DISABLE or a hard-wired input; if you are running the program in optimized mode, a hard-wired input is required to toggle 10001.

Note: The maximum amount of time allowed for a scan is 250 ms; if the scan has not completed in that amount of time, a *watchdog timer* in the CPU stops the application and sends a timeout error message to the programming panel display. The maximum limit on scan time protects the PLC from entering into an infinite loop.

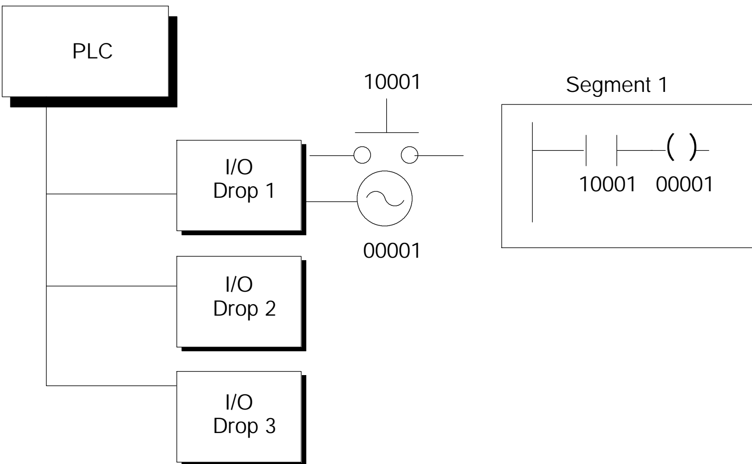
A.3 Maximizing Throughput

The PLC architecture simultaneously solves logic and services I/O drops to optimize system throughput. *Throughput* is the time it takes for a signal received at a field sensing device to be sent as an input to the PLC, processed in ladder logic, and returned as an output signal to a field working device. Throughput time may be longer or shorter than a single scan; it gives you a realistic measure of the system's actual performance.

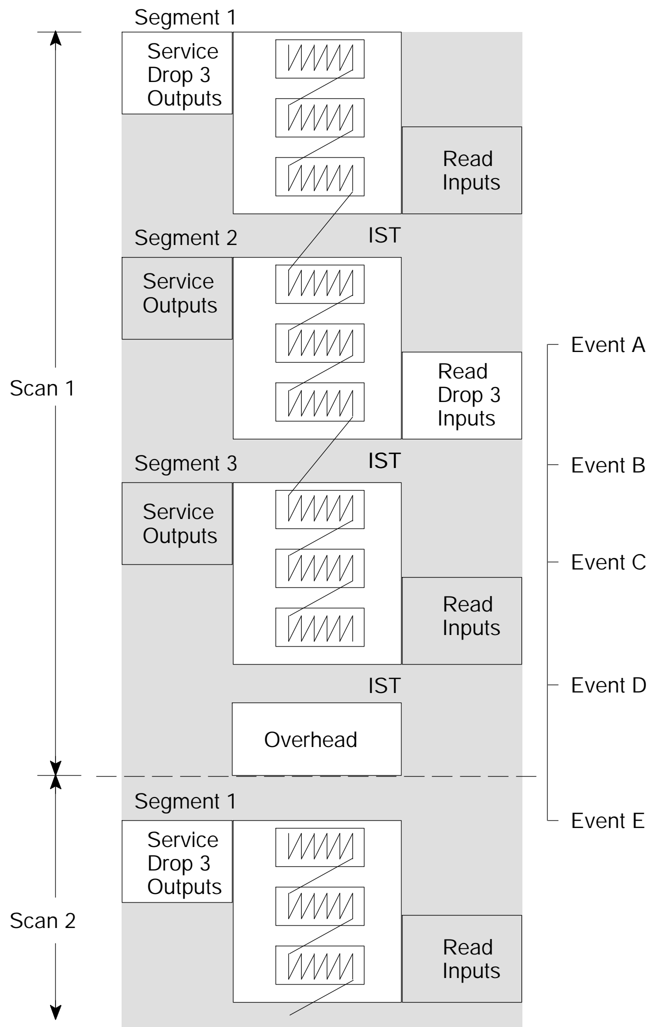
A.3.1 The Ideal Throughput Situation

If the default segment scheduler is in place, the system automatically solves the logic starting at segment 1 and moving sequentially through segment *n*. Throughput is optimized when logic referring to real-world I/O is contained in the segment that corresponds to that I/O drop.

For instance, if you are using I/O in drop 1 of a three-drop system to control a pushbutton that starts a motor, the ideal condition is for logic segment 1 to contain all the appropriate logic:



When all logic segments are coordinated with all physical I/O drops in this manner, the throughput for a given logic segment can be less than one scan. Here is how it can be traced in our scan time model:



The model tracks throughput for drop 3. Throughput in this best case example is about 75% of total scan time. Five benchmark events are shown:

- Event A, where the inputs from drop 3 are available to the I/O processor
- Event B, where the I/O processor transfers data to state RAM
- Event C, where the segment 3 logic networks are solved
- Event D, where data are transferred from state RAM to the I/O processor
- Event E, where the output data are written to the output modules at drop 3

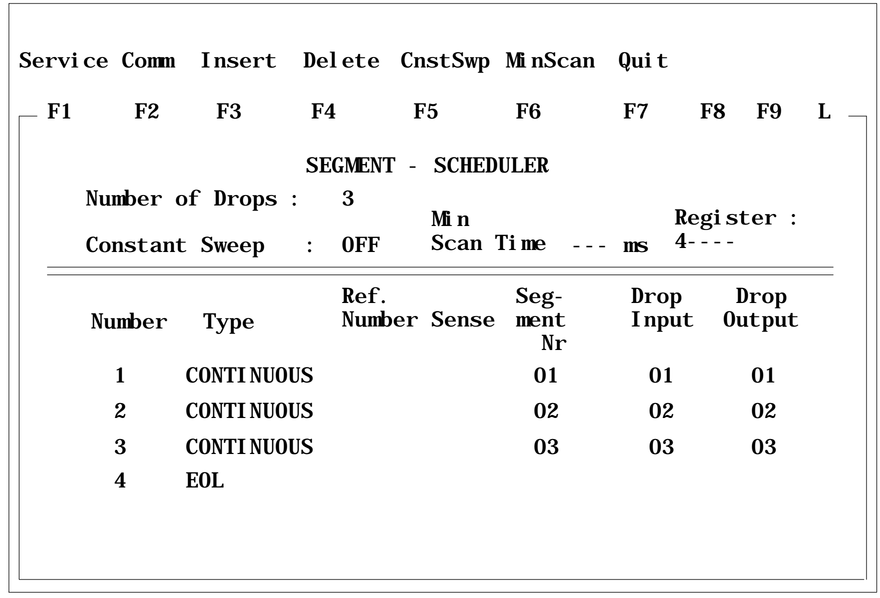
A.4 The Order of Solve

You specify the number of segments and I/O drops with the configurator editor in your panel software package. The default order-of-solve condition is segment 1 through segment *n* consecutively and continuously, once per scan, with the corresponding I/O drops serviced in like order. You are able to change the order of solve using the segment scheduler editor in your panel software package.

There may be times when you can modify the order of solve to improve overall system performance. The segment scheduler can be used effectively to:

- Improve throughput for critical I/O
- Improve overall system performance
- Optimize the servicing of communication ports

Here is what a default order of solve might look like, as seen in the Modsoft segment scheduler editor:



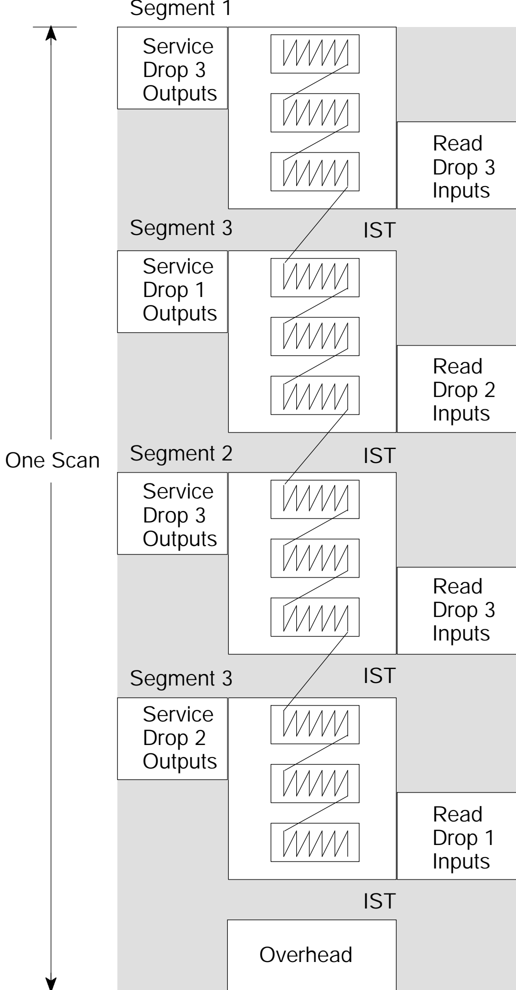
A Default Order-of-Solve Table for a Three-Segment Logic Program

A.5 Using the Segment Scheduler to Improve Critical I/O Throughput

Suppose that your logic program is three segments long and that segment 3 contains logic that is critical to your application—for example, monitoring a proximity switch to verify part presence. Segments 1 and 2 are running noncritical logic such as part count analysis and statistic gathering. The program is running in the standard order-of-solve mode, and you are finding that the PLC is not able to read critical inputs with the frequency desired, thereby causing unacceptable system delay.

Using the segment scheduler editor, you can improve the throughput for the critical I/O at drop 3 by scheduling segment 3 to be solved two (or more) times in the same scan.

Here is an example of a rescheduled logic program, again using our scan time model:



By rescheduling the order-of-solve table, you actually increase the scan time, but more importantly you improve throughput for the critical I/O supported by logic in segment 3. Throughput is the better measure of system performance.

Here is how the Modsoft segment scheduler would show the resulting order-of-solve table:

```
Service Comm  Insert  Delete  CnstSwp MinScan  Quit

 F1     F2     F3     F4      F5      F6      F7    F8   F9   L

                    SEGMENT - SCHEDULER
   Number of Drops :   3
                              Min                  Register :
   Constant Sweep   :  OFF    Scan Time  --- ms    4----

                         Ref.            Segment   Drop    Drop
   Number   Type         Number Sense      Nr      Input   Output

     1    CONTINUOUS                       01       01      01
     2    CONTINUOUS                       03       03      03
     3    CONTINUOUS                       02       02      02
     4    CONTINUOUS                       03       03      03
     5    EOL
```

An Order-of-Solve Table Rescheduled for Critical I/O

A.6 Using the Segment Scheduler to Improve System Performance

When certain areas of a ladder logic program do not need to be solved continually on every scan—for example, an alarm handling routine, a data analysis routine, some diagnostic message routines—they can be designated as *controlled segments* by the segment scheduler editor. Based on the status of an I/O or internal reference, a controlled segment may be scheduled to be skipped, thereby reducing scan time and improving overall system throughput.

For example, suppose that you have some alarm handling logic in segment 2 of a three-segment logic program. You can use the segment scheduler editor to *control* segment 2 based on the status of a coil 00056—if the coil is ON, segment 2 logic will be activated in the scan, and if the coil is OFF the segment will not be solved in the scan. I/O servicing is still performed, regardless of the conditional status. Here is how the Modsoft segment scheduler would show the resulting order-of-solve table:

```
Service Comm  Insert  Delete  CnstSwp MinScan  Quit

 F1     F2     F3     F4      F5      F6      F7     F8   F9   L

                         SEGMENT - SCHEDULER
     Number of Drops :    3
                                   Min                      Register :
     Constant Sweep   :  OFF       Scan Time  --- ms        4----

                           Ref.           Segment    Drop     Drop
      Number    Type       Number Sense    Nr        Input    Output

        1     CONTINUOUS                   01         01       01
        2     CONTINUOUS                   03         03       03
        3     CONTROLLED   00056   ON      02         02       02
        4     CONTINUOUS                   03         03       03
        5     EOL
```

An Order-of-Solve Table Rescheduled for a Controlled Logic Segment

A.7 Using the Segment Scheduler to Improve Communication Port Servicing

When you find that the frequency of standard end-of-scan servicing of communication ports, option processors, or system diagnostics is inadequate for your application requirements, you can increase service frequency by inserting one or more *reset watchdog timer* routines in the order-of-solve table. Each time this routine is encountered by the CPU, it causes all communication ports to be serviced and causes the system diagnostics to be run.

Here is how the Modsoft segment scheduler would show an order-of-solve table where the comm ports are serviced after each segment in the logic program:

```
Service Comm  Insert  Delete  CnstSwp MinScan  Quit
  F1     F2     F3     F4      F5      F6      F7    F8   F9   L

                     SEGMENT - SCHEDULER
    Number of Drops :   3
                                 Min                 Register :
    Constant Sweep   :  OFF      Scan Time  --- ms   4----

                          Ref.           Segment   Drop    Drop
    Number   Type         Number Sense   Nr        Input   Output

      1     CONTINUOUS                   01        01      01
      2     WDT RESET
      3     CONTINUOUS                   02        02      02
      4     WDT RESET
      5     CONTINUOUS                   03        03      03
      6     EOL
```

An Order-of-Solve Table Rescheduled for Three Comm Port Servicings per Scan

A.8 Sweep Functions

Sweep functions allow you to scan a logic program at fixed intervals. They do not make the PLC solve logic faster or terminate scans prematurely.

A.8.1 Constant Sweep

Constant Sweep allows you to set target scan times from 10 ... 200 ms (in multiples of 10). A target scan time is the time between the start of one scan and the start of the next; it is not the time between the end of one scan and the beginning of the next.

Constant Sweep is useful in applications where data must be sampled at constant time intervals. If a Constant Sweep is invoked with a time lapse smaller than the actual scan time, the time lapse is ignored and the system uses its own normal scan rate. The Constant Sweep target scan time encompasses logic solving, I/O and Modbus port servicing, and system diagnostics. If you set a target scan of 40 ms and the logic solving, I/O servicing, and diagnostics require only 30 ms, the PLC will wait 10 ms on each scan. Consult your programming documentation for procedures to invoke a Constant Sweep function.

A.8.2 Single Sweep

The Single Sweep function allows your PLC to execute a fixed number of scans (from 1 ... 15) and then to stop solving logic but continue servicing I/O. This function is useful for diagnostic work—it allows solved logic, moved data, and performed calculations to be examined for errors.

Warning: The Single Sweep function should not be used to debug controls on machine tools, processes, or material handling systems when they are active. Once a specified number of scans has been solved, all outputs are frozen in their last state. Since no logic solving is taking place, the PLC ignores all input information. This can result in unsafe, hazardous, and destructive operation of the machine or process connected to the PLC.

Consult your programming documentation for procedures to invoke Single Sweep functions.

Index

A

AD16 instruction, 84

ADD instruction, 76

ADDDP function, in EMTH, 107

ADDFP function, in EMTH, 123

ADDIF function, in EMTH, 118

addition
  floating point, 123
  floating point and integer values, 118
  signed or unsigned 16-bit, 84
  unsigned integer, 76

AIN (analog in) function, in PCFL, 424

ALARM function, in PCFL, 422

alarm/event warning system, 499

algebraic operators, in an Equation Network, 157

AND circuit, built from contacts and coils, 40

AND instruction, 207

ANLOG function, in EMTH, 115

antilogarithm (base 10) calculation
  using EMTH, 115
  using MATH, 142

AOUT (analog out) function, in PCFL, 426

arccosine calculation, in floating point, 131

ARCOS function, in EMTH, 131

arcsine calculation, in floating point, 130

arctangent calculation, in floating point, 132

ARSIN function, in EMTH, 130

ARTAN function, in EMTH, 132

ASCII character set, 312

ASCII message formats
  for COMM instruction, 308
  for READ/WRIT instructions, 303

AVER (average weighted inputs) function, in PCFL, 417

B

battery coil assignment, in the configurator, 18

BCD instruction, 100

BCD-to-binary format conversion, 100

benchmark performance, for Equation Network operations in ladder logic, 167

binary addition checksum, in ladder logic, 330

binary-to-BCD format conversion, 100

BLKM instruction, 189

BLKT instruction, 192

BMDI instruction, 391, 394

Boolean operations, 207, 210, 213

BROT instruction, 226

C

CALC (calculate preset formula) function, in PCFL, 418

CALL instruction, 471
  loadable part numbers, 460

changing signs, for floating point numbers, 126

CHS instruction, 466

CHSIN function, in EMTH, 126

CKSM instruction, in ladder logic, 330

Clear local statistics, via the MSTR instruction, 349

Clear remote statistics, via the MSTR instruction, 355

clearing bits, in a DX matrix, 224

closed loop control, 400

CMPFP function, in EMTH, 125

CMPIF function, in EMTH, 121

CMPR instruction, 218

CNVDR function, in EMTH, 134

CNVFI function, in EMTH, 122

CNVIF function, in EMTH, 117

CNVRD function, in EMTH, 133

coil usage in a ladder logic program, 38

coils
- 0x, 14
- as displayed in ladder logic, 3
- latched, 36, 37
- memory-retentive, 36
- memory-retentive, 37
- normal, 36

COMM instruction, messaging for Micro PLCs, 306

common logarithm calculation, in floating point, 137

COMP instruction, 216

comparison
- bit patterns in DX matrices, 218
- floating point and integer values, 121
- two floating point values, 125

complementing a bit pattern, 216

conditional segments, as defined by segment scheduler, 524

configuration table, 18

configurator editor, 18

constant sweep, 526

constants, in an Equation Network, 155

contacts
- negative transitional, 34
- normally closed, 33
- normally open, 32
- positive transitional, 33

conversion
- binary-expressed analog data to engineering units, 503
- floating point and integer values, 117, 122
- radians to degrees, 133

COS function, in EMTH, 129

cosine calculation, in floating point, 129

counters
- down, 62
- up, 60

CRC-16 checksum, in ladder logic, 330

CTIF instruction, for hardwired counter/timer/interrupt setup, on the Micro PLCs, 380

custom loadable instruction design, 496

D

data types, in an Equation Network, 154

DCTR instruction, 62

degree-to-radian conversion, in floating point, 134

DELAY function, in PCFL, 427

derivative control, in PID2, 403

DIO health status table, 276

DIO system status, how the DIOH block works, 278

DIOH instruction, 278

disable discrete values in ladder logic, 38

discrete inputs, 1*x*, 14

discrete outputs, 0*x*, 14

DIV instruction, 82

DIVDP function, in EMTH, 110

DIVFI function, in EMTH, 120

DIVFP function, in EMTH, 124

DIVIF function, in EMTH, 119

division
  floating point, 124
  floating point and integer values, 119, 120
  signed or unsigned 16-bit, 93
  unsigned integer, 82

DMTH instruction, loadable part number, 461

double precision addition
  using DMTH, 143
  using EMTH, 107

double precision division
  using DMTH, 146
  using EMTH, 110

double precision multiplication
  using DMTH, 145
  using EMTH, 109

double precision subtraction
  using DMTH, 144
  using EMTH, 108

DRUM instruction, 318
  loadable part numbers, 461

DV16 instruction, 93

E

E. *See* error measurement

EARS instruction, 499

EMTH, overview, 104

EMTH performance benchmarks, compared to Equation Network, in Quantum PLCs, 167

engineering unit conversion, in ladder logic, 503

environment, for programming 984 custom loadables, 496

EQN (formatted equation) function, in PCFL, 420

Equation Network, 149, 150
  benchmark performance, 167

Equation Networks
  algebraic operators, 157
  block structure, 150
  constant data, 155
  data types supported, 154
  functions, 161
  parentheses, 159
  variable data, 154

ERLOG function, in EMTH, 138

error measurement, in a PID2 function, 400

ESI instruction, 474

EUCA instruction, 503

event/alarm warning system, 499

examples
  a scan time evaluation circuit, 516
  CMPR instruction, 220
  COMP instruction, 217
  EUCA operations, 505, 507, 509
  Fahrenheit-to-Centigrade conversion, 102
  ideal throughput, 517
  one second timer, 66
  PID2 level control, 409
  recipe storage, 190
  reporting current system status, 222
  searching for bit values, 188
  sequential control using SCIF instruction, 327
  simple table averaging, 228
  skipping nodes in a network, 283
  subroutine in ladder logic, 378
  using a segment scheduler to improve throughput, 521
  using a segment scheduler to increase port service, 525
  using asegment scheduler for controlled segments, 524

exclusive OR instruction, 213

EXP function, in EMTH, 135

exponential calculation, in floating point, 135

extended memory
in 24-bit CPUs, 288
storage in user memory, 289

F

FIN instruction, 181

floating point + integer addition, 118

floating point - integer subtraction, 118, 120

floating point addition, 123

floating point arccosine calculation, 131

floating point arcsine calculation, 130

floating point arctangent calculation, 132

floating point common logarithm calculation, 137

floating point comparison, 125

floating point conversion
degrees to radians, 134
radians to degrees, 133

floating point cosine calculation, 129

floating point division, 124

floating point error reporting, 138

floating point exponential calculation, 135

floating point format standard, 116

floating point multiplication, 124

floating point natural logarithm calculation, 136

floating point numbers, changing signs, 126

floating point sine calculation, 128

floating point square root, 126

floating point subtraction, 123

floating point tangent calculation, 130

floating point value of pi, 127

floating point x integer multiplication, 119

floating point-to-integer conversion instruction, signed or unsigned 16-bit, 98

floating point/integer division, 119, 120

floating point-integer conversion, 122

FNxx instruction, 496

forcing OFF a discrete value in ladder logic, 38

forcing ON a discrete value in ladder logic, 38

FOUT instruction, 184

FTOI instruction, 98

functions, in an Equation Network, 161

G

Get Data command, via the ESI instruction, 482

Get local statistics, via the MSTR instruction, 347

get Modbus II statistics, with MBUS, 491

Get remote statistics, via the MSTR instruction, 353

H

HLTH instruction, 264

holding registers, 4x, 14

horizontal shorts, 39

HSBY instruction, 462
loadable part numbers, 460

I

I/O map table, 22

IBKR instruction, 198

IBKW instruction, 201
ICMP instruction, 321
loadable part numbers, 461
ID instruction, 391, 392
IE instruction, 391, 393
IMIO instruction, 396
immediate access to I/O from ladder logic, via an interrupt instruction, 396
indirect block reading data, from noncontiguous registers, 198
indirect block writing data, to noncontiguous registers, 201
instruction set
built into select PLCs, 8
loadables for select PLCs, 10
standard for all PLCs, 7
INTEG (integrate over time interval) function, in PCFL, 430
integer-floating point comparison, 121
integer-to-floating point conversion, 117
integer-to-floating point conversion instruction, signed or unsigned 16-bit, 96
integral control, in PID2, 403
interrupt disabling, 391, 392
interrupt enabling, 391, 393
interrupts, interval timer, 388
intersegment transfer (IST), as a part of scan time, 512
interval timer interrupts, 388
IST, 514
ITMR instruction, 388
ITOF instruction, 96

J

JSR instruction, 373
jump to a subroutine, 374

K

KPID function, in PCFL, 444

L

LAB instruction, 375
labeling the start of a subroutine, 375
ladder logic, structure, 2
ladder logic elements, standard for all PLCs, 7
latched coils, 36
LIMIT function, in PCFL, 432
LIMV (limit velocity of change) function, in PCFL, 433
LKUP (lookup table) function, in PCFL, 429
LLAG (dynamic compensation) function, in PCFL, 431
LNFP function, in EMTH, 136
loadable instructions, 460
developing your own custom blocks, 496
LOG function, in EMTH, 114
logarithm (base 10) calculation
using EMTH, 114
using MATH, 141
LOGFP function, in EMTH, 137
logic solve time, as a part of scan time, 512
LRC checksum, in ladder logic, 330

M

manipulated variable, in a PID2 function, 400
masking a timer-generated interrupt, 391, 394
masking an I/O-generated interrupt, 391, 394

MATH instruction, loadable part number, 461
MBIT instruction, 224
MBUS instruction, 489
  loadable part numbers, 460
memory-retentive coils, 36
Modbus II instructions
  MBUS, 489
  PEER, 494
Modbus II local statistics, 492
Modbus Plus, MSTR instruction, 334
Modbus Plus network statictics, 365
MODE function, in PCFL, 434
moving a block of data, in DX tables, 189
moving tables to registers, 195
MSTR error codes, 339
MSTR function, 334
MSTR instruction, loadable part number, 460
MSTR operations
  clear local statistics, 349
  clear remote statistics, 355
  get local statistics, 347
  get remote statistics, 353
  Peer Cop communications health, 357
  read, 345
  read global data, 352
  write, 345
  write global data, 351
MU16 instruction, 90
MUL instruction, 80
MULDP function, in EMTH, 109
MULFP function, in EMTH, 124
MULIF function, in EMTH, 119
multiplication
  floating point, 124
  floating point and integer values, 119
  signed or unsigned 16-bit, 90
  unsigned integer, 80
mv. *See* manipulated variable

N

N.C. contacts, 33
N.O. contacts, 32
natural logarithm calculation, in floating point, 136
NBIT instruction, 49
NCBT instruction, 47
negative numbers, in a floating point calculation, 116
negative transitional contacts, 34
Network statistics, for Modbus Plus, 365
NOBT instruction, 45
node, in ladder logic, 24
nodes, in ladder logic, 2
normal coils, 36
normally closed contacts, 33
normally open contacts, 32

O

ONOFF function, in PCFL, 446
opcodes, 24
  for ladder logic elements and non-DX functiona, 24
  for standard DX functions, 28
  in custom loadable designs, 496
operators, in an Equation Network, 157
OR circuit, built from contacts and coils, 40
OR instruction, 210
order-of-solve table, 517
overhead services, as a part of scan time, 514

P

parentheses, in an Equation Network, 159

PCFL instruction, 413

Peer cop communications health statistics, via the MSTR instruction, 357

PEER instruction, 494
  loadable part numbers, 460

pi, loading the FP value of, 127

PI function
  in EMTH, 127
  in PCFL, 453

PID function, in PCFL, 442, 448

PID2 algorithm, 401

PID2 instruction, 401

positive transitional contacts, 33

POW function, in EMTH, 134

process square root calculation
  using EMTH, 112, 140
  using MATH, 140

process variable, in a PID2 function, 400

proportional control, in PID2, 402

Put Data command, via the ESI instruction, 483

PV. *See* process variable

R

raising an FP number to an integer power, 134

RAMP function, in PCFL, 436

RATE function, in PCFL, 439

RATIO function, in PCFL, 455

RBIT instruction, 53

Read ASCII command, via the ESI instruction, 478

Read global data, via the MSTR instruction, 352

READ instruction, for ASCII communications, 296

Read operations, via the MSTR instruction, 345

reference numbering system, 14

register inputs, 3*x*, 14

register outputs, 4*x*, 14

register-to-table move, 171

reset watchdog timer routine, 525

RET instruction, 377

returning from a subroutine, 377

reverse transfer function, in Hot Standby systems, 464

RIO status table
  for Compact PLC users, 250
  for Micro PLC users, 255
  for S901 users, 234
  for S908 users, 240

RIO system status, how the STAT block works, 232

RMPLN (logarithmic ramp) function, in PCFL, 437

rotating a bit pattern, in a DX matrix, 226

RStF, Modsoft off-line function for SFC, 230

S

SBIT instruction, 51

scan time, 512

scan time evaluation circuit, 516

scanning logic segments, 5

SCIF instruction, 324

seal circuit, built from contacts and coils, 41

search for bit pattern, in a DX table, 187

segment scheduler, 4, 520
  defining order of logic solution, 5
  improving overall system performance, 524

improving overhead servicing frequency, 525
improving throughput for critical I/O, 521

SEL function, in PCFL, 440

SENS instruction, 221

sense of a bit, 221

sequential control functions, cascaded blocks, 323

sequential control instructions
DRUM, 318
ICMP, 321
SCIF, 324

setpoint, in a PID2 function, 400

setting a bit, in a DX matrix, 224

shorts
horizontal, 39
vertical, 39

sign changing, for floating point numbers, 126

sine calculation, in floating point, 128

SINE function, in EMTH, 128

single sweep, 526

SKIP, Modsoft off-line function for SFC, 284

skip constant, Modsoft off-line function for SFC or macros, 284

skip register, Modsoft off-line function for SFC or macros, 284

skipping networks in ladder logic, 282

SKP instruction, 282

SKPC, Modsoft off-line function for SFC or macros, 284

SKPR, Modsoft off-line function for SFC or macros, 284

SP. *See* setpoint

SQRFP function, in EMTH, 126

SQRT function, in EMTH, 111

SQRTP function, in EMTH, 112

square root, floating point, 126

square root calculation
using EMTH, 111
using MATH, 139

SRCH instruction, 187

STAT instruction, 232

state RAM, minimum configuration, 17

SU16 instruction, 86

SUB instruction, 78

SUBDP function, in EMTH, 108

SUBFI function, in EMTH, 120

SUBFP function, in EMTH, 123

SUBIF function, in EMTH, 118

subroutines, in ladder logic, 372

subtraction
floating point, 123
floating point and integer values, 118, 120
signed or unsigned 16-bit, 86
unsigned integer, 78

sweep functions, 526

system overhead, in user memory, 13

T

T.01 instruction, 69

T0.1 instruction, 67

T1.0 instruction, 64

T1MS instruction, 71

table-to-register move, 174

table-to-table move, 177

TAN function, in EMTH, 130

tangent calculation, in floating point, 130

TBLK instruction, 195

TC, Modsoft off-line function for SFC, 229

TEST instruction, 88

throughput, 517

time of day clock assignment, in the configurator, 19

timer
- hundredth-of-a-second, 69
- millisecond, 71
- one-second, 64
- tenth-of-a-second, 67

timer register assignment, in the configurator, 18

TOD assignment, in the configurator, 19

TOTAL function, in PCFL, 456

trace capability, 38

traffic cop table, 22

U

UCTR function, 60

UCTR instruction, 60

user logic, in user memory, 12

user memory, 12
- CMOS RAM storage, 13

V

variables, in an Equation Network, 154

vertical shorts, 39

W

watchdog timer, 516

WRIT instruction, for ASCII communications, 300

Write ASCII command, via the ESI instruction, 481

Write global data, via the MSTR instruction, 351

Write operations, via the MSTR instruction, 345

X

XMRD instruction, 292

XMWT instruction, 290

XOR circuit, built from contacts and coils, 41

XOR instruction, 213