

# SIEMENS

## SIMATIC 545/555/575


### Programming Reference

User Manual


Order Number: PPX:505–8204-2  
Manual Assembly Number: 2806090–0002  
Second Edition

**Safety-Related  
Guidelines**

This manual contains the following notices intended to ensure personal safety, as well as to protect the products and connected equipment against damage.

 <b>DANGER</b>
<b>DANGER</b> indicates an imminently hazardous situation that, if not avoided, will result in death or serious injury. <b>DANGER</b> is limited to the most extreme situations.

 <b>WARNING</b>
<b>WARNING</b> indicates a potentially hazardous situation that, if not avoided, could result in death or serious injury, and/or property damage.

 <b>CAUTION</b>
<b>CAUTION</b> indicates a potentially hazardous situation that, if not avoided, could result in minor or moderate injury, and/or damage to property. <b>CAUTION</b> is also used for property-damage-only accidents.

**Copyright 1998 by Siemens Energy & Automation, Inc.  
All Rights Reserved — Printed in USA**

Reproduction, transmission, or use of this document or contents is not permitted without express consent of Siemens Energy & Automation, Inc. All rights, including rights created by patent grant or registration of a utility model or design, are reserved.

Since Siemens Energy & Automation, Inc., does not possess full access to data concerning all of the uses and applications of customer's products, we do not assume responsibility either for customer product design or for any infringements of patents or rights of others which may result from our assistance.

## MANUAL PUBLICATION HISTORY

SIMATIC 545/555/575 Programming Reference User Manual

Order Manual Number: PPX:505-8204-2

*Refer to this history in all correspondence and/or discussion about this manual.*

---

<b>Event</b>	<b>Date</b>	<b>Description</b>
Original Issue	3/96	Original Issue (2806090-0001)
Second Edition	8/98	Second Edition (2806090-0002)

---

## LIST OF EFFECTIVE PAGES

---

Pages	Description	Pages	Description
Cover/Copyright	Second		
History/Effective Pages	Second		
iii — xxxvi	Second		
1-1 — 1-11	Second		
2-1 — 2-8	Second		
3-1 — 3-14	Second		
4-1 — 4-11	Second		
5-1 — 5-40	Second		
6-1 — 6-187	Second		
7-1 — 7-80	Second		
8-1 — 8-15	Second		
9-1 — 9-45	Second		
A-1 — A-8	Second		
B-1 — B-6	Second		
C-1 — C-15	Second		
D-1 — D-4	Second		
E-1 — E-37	Second		
F-1 — F-1	Second		
G-1 — G-32	Second		
H-1 — H-17	Second		
I-1 — I-12	Second		
Index-1 — Index-10	Second		
Registration	Second		

# Contents

---

## Preface

Introduction .....	xxv
New Features .....	xxvi
How to Use This Manual .....	xxvi
TISOFT Programming Software .....	xxvi
SIMATIC 505 SoftShop for Windows .....	xxvi
Technical Assistance .....	xxviii

## Chapter 1 Series 505 System Overview

<b>1.1 The 545, 555, and 575 Systems .....</b>	<b>1-2</b>
System Components .....	1-2
Using PROFIBUS-DP I/O .....	1-2
Local I/O .....	1-2
Expansion I/O Channels .....	1-2
Series 505 Remote I/O .....	1-2
PROFIBUS-DP I/O .....	1-4
Output Response on PROFIBUS-DP Slave Devices .....	1-4
Assigning I/O Point Numbers .....	1-5
<b>1.2 Program Execution Operations .....</b>	<b>1-6</b>
CPU Scan Operations .....	1-6
Interrupt RLL Execution .....	1-6
Cyclic RLL Execution .....	1-6
Discrete Scan .....	1-6
Analog Task Processing .....	1-8
Cyclic Analog Tasks .....	1-8
Non-cyclic Analog Tasks .....	1-9
Setting the Scan .....	1-10

## Chapter 2 Data Representation

<b>2.1 Definitions .....</b>	<b>2-2</b>
Byte .....	2-2
Word .....	2-2
Long Word .....	2-2
Image Register .....	2-2
I/O Point .....	2-2
<b>2.2 Integers .....</b>	<b>2-3</b>
Signed 16-Bit Integers .....	2-3
Unsigned 16-Bit Integers .....	2-4
Signed 32-Bit Integers .....	2-4
<b>2.3 Real Numbers .....</b>	<b>2-5</b>
<b>2.4 Binary-Coded Decimal .....</b>	<b>2-6</b>
<b>2.5 Format for an Address Stored in a Memory Location .....</b>	<b>2-7</b>

---

## Chapter 3 I/O Concepts

<b>3.1</b>	<b>Reading and Updating the I/O</b> .....	<b>3-2</b>
	Discrete Image Register .....	3-3
	Word Image Register .....	3-5
<b>3.2</b>	<b>Normal I/O Updates</b> .....	<b>3-6</b>
	Discrete Control .....	3-6
	Analog Control .....	3-6
<b>3.3</b>	<b>High Speed I/O Updates</b> .....	<b>3-8</b>
	Immediate I/O .....	3-8
	Modules that Support Immediate I/O .....	3-10
	Configuring Immediate I/O .....	3-10
<b>3.4</b>	<b>Interrupt I/O Operation</b> .....	<b>3-11</b>
	Overview .....	3-11
	Configuring the Interrupt Input Module .....	3-11
<b>3.5</b>	<b>Control Relays</b> .....	<b>3-13</b>
	Using Retentive and Non-retentive Control Relays .....	3-14

## Chapter 4 Controller Memory

<b>4.1</b>	<b>Introduction to Controller Memory</b> .....	<b>4-2</b>
	Overview of Controller Memory Types .....	4-2
	RLL Access to the Memory Types .....	4-3
<b>4.2</b>	<b>Controller Memory Types</b> .....	<b>4-4</b>
	Ladder Memory .....	4-4
	Image Register Memory .....	4-4
	Control Relay Memory .....	4-4
	Special Memory .....	4-4
	Compiled Special (CS) Memory .....	4-4
	Temporary Memory .....	4-4
	Variable Memory .....	4-4
	Constant Memory .....	4-5
	Status Word Memory .....	4-5
	Timer/Counter Memory .....	4-5
	Table Move Memory .....	4-6
	One Shot Memory .....	4-7
	Shift Register Memory .....	4-8
	Drum Memory .....	4-9
	PGTS Discrete Parameter Area .....	4-10
	PGTS Word Parameter Area .....	4-10
	User External Subroutine Memory .....	4-11
	Global Memory: 575 Only .....	4-11
	VME Memory: 575 Only .....	4-11

---

## Chapter 5 Programming Concepts

<b>5.1</b>	<b>RLL Components</b> .....	<b>5-2</b>
	RLL Concept .....	5-2
	RLL Contact .....	5-3
	RLL Coil .....	5-8
	RLL Box Instruction .....	5-12
	RLL Rung Structure .....	5-12
	RLL Scan Principles .....	5-13
<b>5.2</b>	<b>Program Compile Sequence</b> .....	<b>5-14</b>
<b>5.3</b>	<b>Using Subroutines</b> .....	<b>5-16</b>
	RLL Subroutine Programs .....	5-16
	SF Programs .....	5-16
	External Subroutines .....	5-17
<b>5.4</b>	<b>Cyclic RLL</b> .....	<b>5-18</b>
	Overview .....	5-18
	Cyclic RLL Execution .....	5-20
<b>5.5</b>	<b>Interrupt RLL (545/555 only)</b> .....	<b>5-22</b>
	The Interrupt RLL Task .....	5-22
	Operation .....	5-25
	Performance Characteristics .....	5-26
	Troubleshooting .....	5-27
<b>5.6</b>	<b>Using Real-Time Clock Data</b> .....	<b>5-28</b>
	BCD Time of Day .....	5-28
	Binary Time of Day .....	5-30
	Time of Day Status .....	5-31
<b>5.7</b>	<b>Entering Relay Ladder Logic</b> .....	<b>5-32</b>
	SoftShop 505 for Windows .....	5-32
	TISOFT .....	5-32
	Using APT .....	5-32
	COM PROFIBUS .....	5-32
<b>5.8</b>	<b>Doing Run-Time Program Edits</b> .....	<b>5-33</b>
	Editing in Run Mode .....	5-33
	Avoid These Actions During Run-Time Edits .....	5-34
	Additional Considerations When Doing Run-Time Edits .....	5-37
<b>5.9</b>	<b>Password Protection</b> .....	<b>5-39</b>
	Protected Program Elements .....	5-39
	Disabled and Enabled Passwords .....	5-39
	Password Protection Levels .....	5-40
	Determining the Current State of Password .....	5-40
	Password Effect on EEPROM .....	5-40

---

## Chapter 6 RLL Instruction Set

<b>6.1</b>	<b>Safety Considerations</b> .....	<b>6-4</b>
	Overview .....	6-4
	Failure of the Control System .....	6-4
	Inconsistent Program Operation .....	6-5
	Editing an Active Process .....	6-5
<b>6.2</b>	<b>Introduction</b> .....	<b>6-6</b>
<b>6.3</b>	<b>Absolute Value</b> .....	<b>6-11</b>
	ABSV Description .....	6-11
	ABSV Operation .....	6-11
<b>6.4</b>	<b>Add</b> .....	<b>6-12</b>
	ADD Description .....	6-12
	ADD Operation .....	6-12
<b>6.5</b>	<b>Bit Clear</b> .....	<b>6-13</b>
	BITC Description .....	6-13
	BITC Operation .....	6-13
<b>6.6</b>	<b>Bit Pick</b> .....	<b>6-14</b>
	BITP Description .....	6-14
	BITP Operation .....	6-14
<b>6.7</b>	<b>Bit Set</b> .....	<b>6-15</b>
	BITS Description .....	6-15
	BITS Operation .....	6-15
<b>6.8</b>	<b>Convert Binary to BCD</b> .....	<b>6-16</b>
	CBD Description .....	6-16
	CBD Operation .....	6-16
<b>6.9</b>	<b>Convert BCD to Binary</b> .....	<b>6-18</b>
	CDB Description .....	6-18
	CDB Operation .....	6-18
<b>6.10</b>	<b>Compare</b> .....	<b>6-20</b>
	CMP Description .....	6-20
	CMP Operation .....	6-20
<b>6.11</b>	<b>Coils</b> .....	<b>6-22</b>
<b>6.12</b>	<b>Contacts</b> .....	<b>6-23</b>
<b>6.13</b>	<b>Counter (Up Counter)</b> .....	<b>6-24</b>
	CTR Description .....	6-24
	CTR Operation .....	6-24
	Using the Counter Variables .....	6-25



---

<b>6.14</b>	<b>Discrete Control Alarm Timer</b> .....	<b>6-26</b>
	DCAT Description .....	6-26
	DCAT State Changes .....	6-27
	DCAT Operation .....	6-28
	Open (Input On) .....	6-28
	Close (Input Off) .....	6-28
	Using the DCAT Variables .....	6-29
<b>6.15</b>	<b>Date Compare</b> .....	<b>6-30</b>
	DCMP Description .....	6-30
	DCMP Operation .....	6-31
<b>6.16</b>	<b>Divide</b> .....	<b>6-32</b>
	DIV Description .....	6-32
	DIV Operation .....	6-33
<b>6.17</b>	<b>Time Driven Drum</b> .....	<b>6-34</b>
	DRUM Description .....	6-34
	DRUM Operation .....	6-34
	Calculating Counts/Step .....	6-36
	Using DRUM Variables .....	6-36
<b>6.18</b>	<b>Date Set</b> .....	<b>6-38</b>
	DSET Description .....	6-38
	DSET Operation .....	6-39
<b>6.19</b>	<b>Time/Event Driven Drum</b> .....	<b>6-40</b>
	EDRUM Description .....	6-40
	EDRUM Operation .....	6-41
	Calculating Counts/Step .....	6-42
	Timer-triggered Advance Only .....	6-42
	Event-triggered Advance Only .....	6-42
	Timer and Event- Triggered Advance .....	6-43
	Timer or External Event-triggered Advance .....	6-43
	Using EDRUM Variables .....	6-43
<b>6.20</b>	<b>Unconditional End</b> .....	<b>6-44</b>
	END Description .....	6-44
	END Operation .....	6-44
<b>6.21</b>	<b>Conditional End</b> .....	<b>6-45</b>
	ENDC Description .....	6-45
	ENDC Operation .....	6-45
<b>6.22</b>	<b>Go To Subroutine</b> .....	<b>6-46</b>
	GTS Description .....	6-46
	GTS Operation .....	6-46
<b>6.23</b>	<b>Indexed Matrix Compare</b> .....	<b>6-48</b>
	IMC Description .....	6-48
	IMC Operation .....	6-49

---

<b>6.24</b>	<b>Immediate I/O Read/Write</b> .....	<b>6-50</b>
	IORW Description .....	6-50
	IORW Operation .....	6-50
<b>6.25</b>	<b>Jump</b> .....	<b>6-52</b>
	JMP Description .....	6-52
	JMP/JMPE Operation .....	6-52
<b>6.26</b>	<b>Load Address</b> .....	<b>6-54</b>
	LDA Description .....	6-54
	LDA Operation .....	6-55
	Specifying Source .....	6-57
	Specifying Index for Source .....	6-57
	Specifying Destination .....	6-58
	Specifying Index for Destination .....	6-58
<b>6.27</b>	<b>Load Data Constant</b> .....	<b>6-59</b>
	LDC Description .....	6-59
	LDC Operation .....	6-59
<b>6.28</b>	<b>Lock Memory</b> .....	<b>6-60</b>
	LOCK Description .....	6-60
	Acquiring Control of the Lock .....	6-60
	How the Lock Protects Memory .....	6-62
<b>6.29</b>	<b>Motor Control Alarm Timer</b> .....	<b>6-63</b>
	MCAT Description .....	6-63
	MCAT State Changes .....	6-64
	MCAT Operation .....	6-66
	Open Input Turns On .....	6-66
	Close Input Turns On .....	6-66
	Using the MCAT Variables .....	6-67
<b>6.30</b>	<b>Master Control Relay</b> .....	<b>6-68</b>
	MCR Description .....	6-68
	MCR/MCRE Operation .....	6-68
<b>6.31</b>	<b>Maskable Event Drum, Discrete</b> .....	<b>6-72</b>
	MDRMD Description .....	6-72
	MDRMD Operation .....	6-73
	Defining the Mask .....	6-74
	Calculating Counts/Step .....	6-74
	Timer-triggered Advance Only .....	6-74
	Event-triggered Advance Only .....	6-74
	Timer and Event-Triggered Advance .....	6-75
	Timer or External Event-Triggered Advance .....	6-75
	Using MDRMD Variables .....	6-75

---

<b>6.32</b>	<b>Maskable Event Drum, Word</b> .....	<b>6-76</b>
	MDRMW Description .....	6-76
	MDRMW Operation .....	6-78
	Defining the Mask .....	6-79
	Calculating Counts/Step .....	6-79
	Timer-triggered Advance Only .....	6-80
	Event-triggered Advance Only .....	6-80
	Timer and Event-Triggered Advance .....	6-80
	Timer or External Event-triggered Advance .....	6-80
	Using MDRMD Variables .....	6-81
<b>6.33</b>	<b>Move Image Register from Table</b> .....	<b>6-82</b>
	MIRFT Description .....	6-82
	MIRFT Operation .....	6-82
<b>6.34</b>	<b>Move Image Register to Table</b> .....	<b>6-84</b>
	MIRTT Description .....	6-84
	MIRTT Operation .....	6-84
<b>6.35</b>	<b>Move Image Register to Word</b> .....	<b>6-86</b>
	MIRW Description .....	6-86
	MIRW Operation .....	6-86
<b>6.36</b>	<b>Move Element</b> .....	<b>6-88</b>
	MOVE Description .....	6-88
	MOVE Operation .....	6-89
	Specifying Type of Elements .....	6-93
	Specifying Source .....	6-93
	Specifying Index for Source .....	6-93
	Specifying Destination .....	6-94
	Specifying Index for Destination .....	6-94
	Specifying Number of Elements to Move .....	6-95
<b>6.37</b>	<b>Move Word</b> .....	<b>6-96</b>
	MOVW Description .....	6-96
	MOVW Operation .....	6-97
<b>6.38</b>	<b>Multiply</b> .....	<b>6-98</b>
	MULT Description .....	6-98
	MULT Operation .....	6-98
<b>6.39</b>	<b>Move Word from Table</b> .....	<b>6-100</b>
	MWFT Description .....	6-100
	MWFT Operation .....	6-100
<b>6.40</b>	<b>Move Word with Index</b> .....	<b>6-102</b>
	MWI Description .....	6-102
	MWI Operation .....	6-102

---

<b>6.41</b>	<b>Move Word to Image Register</b> .....	<b>6-104</b>
	MWIR Description .....	6-104
	MWIR Operation .....	6-104
<b>6.42</b>	<b>Move Word To Table</b> .....	<b>6-106</b>
	MWTT Description .....	6-106
	MWTT Operation .....	6-106
<b>6.43</b>	<b>NOT</b> .....	<b>6-108</b>
	NOT Description .....	6-108
	NOT Operation .....	6-108
<b>6.44</b>	<b>One Shot</b> .....	<b>6-109</b>
	One Shot Description .....	6-109
	One Shot Operation .....	6-109
<b>6.45</b>	<b>PID Loop</b> .....	<b>6-110</b>
	PID Fast Loop Description .....	6-110
	PID Operation .....	6-110
<b>6.46</b>	<b>Parameterized Go To Subroutine</b> .....	<b>6-112</b>
	PGTS Description .....	6-112
	PGTS Operation .....	6-112
<b>6.47</b>	<b>Parameterized Go To Subroutine (Zero)</b> .....	<b>6-118</b>
	PGTSZ Description .....	6-118
	PGTSZ Operation .....	6-119
<b>6.48</b>	<b>Read Slave Diagnostic (RSD)</b> .....	<b>6-120</b>
	RSD Description .....	6-120
	RSD Operation .....	6-121
<b>6.49</b>	<b>Return from Subroutine</b> .....	<b>6-122</b>
	RTN Description .....	6-122
	RTN Operation .....	6-122
<b>6.50</b>	<b>Subroutine</b> .....	<b>6-123</b>
	SBR Description .....	6-123
	SBR Operation .....	6-123
<b>6.51</b>	<b>Call an SF Program</b> .....	<b>6-126</b>
	SFPGM Description .....	6-126
	SFPGM Operation .....	6-126
	In-line SFPGM Execution .....	6-127
<b>6.52</b>	<b>Call SF Subroutines from RLL</b> .....	<b>6-128</b>
	SFSUB Description .....	6-128
	SFSUB Operation .....	6-129
	In-line SFSUB Execution .....	6-130

---

<b>6.53</b>	<b>Bit Shift Register</b> .....	<b>6-132</b>
	SHRB Description .....	6-132
	SHRB Operation .....	6-133
<b>6.54</b>	<b>Word Shift Register</b> .....	<b>6-134</b>
	SHRW Description .....	6-134
	SHRW Operation .....	6-134
<b>6.55</b>	<b>Skip / Label</b> .....	<b>6-136</b>
	SKP / LBL Description .....	6-136
	SKP / LBL Operation .....	6-138
<b>6.56</b>	<b>Scan Matrix Compare</b> .....	<b>6-140</b>
	SMC Description .....	6-140
	SMC Operation .....	6-141
<b>6.57</b>	<b>Square Root</b> .....	<b>6-142</b>
	SQRT Description .....	6-142
	SQRT Operation .....	6-143
<b>6.58</b>	<b>Search Table For Equal</b> .....	<b>6-144</b>
	STFE Description .....	6-144
	STFE Operation .....	6-144
<b>6.59</b>	<b>Search Table For Not Equal</b> .....	<b>6-146</b>
	STFN Description .....	6-146
	STFN Operation .....	6-146
<b>6.60</b>	<b>Subtract</b> .....	<b>6-148</b>
	SUB Description .....	6-148
	SUB Operation .....	6-148
<b>6.61</b>	<b>Table to Table AND</b> .....	<b>6-149</b>
	TAND Description .....	6-149
	TAND Operation .....	6-149
<b>6.62</b>	<b>Start New RLL Task</b> .....	<b>6-150</b>
	TASK Description .....	6-150
	TASK Operation .....	6-150
<b>6.63</b>	<b>Time Compare</b> .....	<b>6-153</b>
	TCMP Description .....	6-153
	TCMP Operation .....	6-153
<b>6.64</b>	<b>Table Complement</b> .....	<b>6-154</b>
	TCPL Description .....	6-154
	TCPL Operation .....	6-154
<b>6.65</b>	<b>Text</b> .....	<b>6-155</b>
	Text Box Description .....	6-155

---

<b>6.66</b>	<b>Timer</b> .....	<b>6-156</b>
	TMR/TMRF Description .....	6-156
	TMR/TMRF Operation .....	6-156
	Using the Timer Variables .....	6-157
<b>6.67</b>	<b>Table to Table OR</b> .....	<b>6-158</b>
	TOR Description .....	6-158
	TOR Operation .....	6-158
<b>6.68</b>	<b>Time Set</b> .....	<b>6-159</b>
	TSET Description .....	6-159
	TSET Operation .....	6-159
<b>6.69</b>	<b>Table to Word</b> .....	<b>6-160</b>
	TTOW Description .....	6-160
	TTOW Operation .....	6-161
<b>6.70</b>	<b>Table to Table Exclusive OR</b> .....	<b>6-162</b>
	TXOR Description .....	6-162
	TXOR Operation .....	6-163
<b>6.71</b>	<b>Up/Down Counter</b> .....	<b>6-164</b>
	UDC Description .....	6-164
	UDC Operation .....	6-165
	Using the UDC Variables .....	6-166
<b>6.72</b>	<b>Unlock Memory</b> .....	<b>6-167</b>
	UNLCK Description .....	6-167
	UNLCK Operation .....	6-167
<b>6.73</b>	<b>Word AND</b> .....	<b>6-168</b>
	WAND Description .....	6-168
	WAND Operation .....	6-168
<b>6.74</b>	<b>Word OR</b> .....	<b>6-170</b>
	WOR Description .....	6-170
	WOR Operation .....	6-170
<b>6.75</b>	<b>Word Rotate</b> .....	<b>6-172</b>
	WROT Description .....	6-172
	WROT Operation .....	6-172
<b>6.76</b>	<b>Word To Table</b> .....	<b>6-174</b>
	WTOT Description .....	6-174
	WTOT Operation .....	6-175
<b>6.77</b>	<b>Word To Table AND</b> .....	<b>6-176</b>
	WTTA Description .....	6-176
	WTTA Operation .....	6-177

<b>6.78</b>	<b>Word To Table OR</b> .....	<b>6-178</b>
	WTT0 Description .....	6-178
	WTT0 Operation .....	6-179
<b>6.79</b>	<b>Word To Table Exclusive OR</b> .....	<b>6-180</b>
	WTTX0 Description .....	6-180
	WTTX0 Operation .....	6-181
<b>6.80</b>	<b>Word Exclusive OR</b> .....	<b>6-182</b>
	WXOR Description .....	6-182
	WXOR Operation .....	6-182
<b>6.81</b>	<b>External Subroutine Call</b> .....	<b>6-184</b>
	XSUB Description .....	6-184
	XSUB Operation .....	6-185

## Chapter 7 Special Function Programs

<b>7.1</b>	<b>Defining Special Function Programs</b> .....	<b>7-2</b>
	Introduction .....	7-2
	Special Function Program Types .....	7-2
	SF Programs Called from RLL .....	7-3
	SF Programs Called from Loops/Analog Alarms .....	7-3
<b>7.2</b>	<b>Using PowerMath with Special Function Programming</b> .....	<b>7-4</b>
	What is PowerMath? .....	7-4
	32-Bit Signed and 16-Bit Unsigned Integer Math .....	7-4
	SF Operators, Functions, and Instructions .....	7-5
	Why Choose Compiled Mode for an SF Program or Subroutine? .....	7-6
	Why Choose Interpreted Mode for an SF Program or Subroutine? .....	7-6
	What Can Be Compiled? .....	7-7
	How Do SF Programs Execute? .....	7-7
	How Do SF Subroutines Execute? .....	7-8
	CALL Subroutine Statement Execution .....	7-9
<b>7.3</b>	<b>SF Program Statements</b> .....	<b>7-10</b>
<b>7.4</b>	<b>Executing Special Function Programs</b> .....	<b>7-11</b>
	Priority/non-priority SF Programs .....	7-11
	In-Line Execution of Compiled SF Programs .....	7-12
	Cyclic Programs .....	7-12
	Restricted Programs Called by Loops .....	7-12
	Restricted Programs Called by Analog Alarms .....	7-13
<b>7.5</b>	<b>Executing Special Function Subroutines</b> .....	<b>7-14</b>
	Calling SF Subroutines .....	7-14
	Designing SF Subroutines .....	7-14
<b>7.6</b>	<b>Memory Usage by SF Programs</b> .....	<b>7-16</b>
<b>7.7</b>	<b>Entering SF Program Header with TISOFT</b> .....	<b>7-18</b>

---

<b>7.8</b>	<b>Reporting SF Program or SFSUB RLL Instruction Errors</b> .....	<b>7-20</b>
	Reporting Errors with the SFEC Variable .....	7-20
	Reporting Errors with Discrete Points .....	7-20
	Reporting Errors with V or WY Memory .....	7-21
<b>7.9</b>	<b>Entering Special Function Programming Statements</b> .....	<b>7-22</b>
<b>7.10</b>	<b>Convert BCD to Binary</b> .....	<b>7-24</b>
	BCDBIN Description .....	7-24
	BCDBIN Operation .....	7-24
<b>7.11</b>	<b>Convert Binary Inputs to BCD</b> .....	<b>7-25</b>
	BINBCD Description .....	7-25
	BINBCD Operation .....	7-25
<b>7.12</b>	<b>Call Subroutine</b> .....	<b>7-26</b>
	CALL Description .....	7-26
	CALL Operation .....	7-26
<b>7.13</b>	<b>Correlated Data Table</b> .....	<b>7-28</b>
	CDT Description .....	7-28
	CDT Operation .....	7-29
<b>7.14</b>	<b>Exit on Error</b> .....	<b>7-30</b>
	EXIT Description .....	7-30
	EXIT Operation .....	7-30
<b>7.15</b>	<b>Fall Through Shift Register—Input</b> .....	<b>7-31</b>
	F TSR-IN Description .....	7-31
	F TSR-IN Operation .....	7-32
<b>7.16</b>	<b>Fall through Shift Register—Output</b> .....	<b>7-35</b>
	F TSR-OUT Description .....	7-35
	F TSR-OUT Operation .....	7-36
<b>7.17</b>	<b>Go To/Label Function</b> .....	<b>7-39</b>
<b>7.18</b>	<b>IF/IIF/THEN/ELSE Functions</b> .....	<b>7-40</b>
	IF/THEN/ELSE Description .....	7-40
	IF Operation .....	7-40
<b>7.19</b>	<b>Integer Math Operations</b> .....	<b>7-42</b>
	IMATH Description .....	7-42
	IMATH Operation .....	7-43
<b>7.20</b>	<b>Lead/Lag Operation</b> .....	<b>7-44</b>
	LEAD/LAG Description .....	7-44
	LEAD/LAG Operation .....	7-45



---

<b>7.21</b>	<b>Real/Integer Math Operations</b> .....	<b>7-46</b>
	MATH Description .....	7-46
	MATH Operation .....	7-47
	Using Word Indexing .....	7-48
	Using Element Indexing .....	7-49
	Indexing Loop and Analog Alarm Variables .....	7-49
	Using Multiple Subscripts .....	7-49
	MATH Examples .....	7-50
<b>7.22</b>	<b>Pack Data</b> .....	<b>7-51</b>
	PACK Description .....	7-51
	PACK TO Operation .....	7-52
	PACK FROM Operation .....	7-54
<b>7.23</b>	<b>Pack Analog Alarm Data</b> .....	<b>7-56</b>
	PACKAA Description .....	7-56
	PACKAA Operation .....	7-57
<b>7.24</b>	<b>Pack Loop Data</b> .....	<b>7-58</b>
	PACKLOOP Description .....	7-58
	PACKLOOP Operation .....	7-58
<b>7.25</b>	<b>Pack Ramp/Soak Data</b> .....	<b>7-60</b>
	PACKRS Description .....	7-60
	PACKRS Operation .....	7-60
<b>7.26</b>	<b>Pet Scan Watchdog</b> .....	<b>7-66</b>
	PETWD Description .....	7-66
<b>7.27</b>	<b>Printing</b> .....	<b>7-68</b>
	PRINT Description .....	7-68
	PRINT Operation .....	7-68
<b>7.28</b>	<b>Return from SF Program/Subroutine</b> .....	<b>7-71</b>
<b>7.29</b>	<b>Scaling Values</b> .....	<b>7-72</b>
	SCALE Description .....	7-72
	SCALE Operation .....	7-73
<b>7.30</b>	<b>Sequential Data Table</b> .....	<b>7-74</b>
	SDT Description .....	7-74
	SDT Operation .....	7-74
<b>7.31</b>	<b>Synchronous Shift Register</b> .....	<b>7-76</b>
	SSR Description .....	7-76
	SSR Operation .....	7-76
<b>7.32</b>	<b>Unscaling Values</b> .....	<b>7-78</b>
	UNSCALE Description .....	7-78
	UNSCALE Operation .....	7-78
<b>7.33</b>	<b>Comment</b> .....	<b>7-80</b>

---

## Chapter 8 Programming Analog Alarms

8.1	Overview .....	8-2
8.2	<b>Analog Alarm Programming and Structure</b> .....	<b>8-4</b>
	Analog Alarm Numbers and Variable Names .....	8-4
	Programming Tables .....	8-4
	Analog Alarm C-Flags .....	8-5
8.3	<b>Specifying Analog Alarm V-Flag Address</b> .....	<b>8-6</b>
	Alarm V-Flag Address .....	8-6
8.4	<b>Specifying Analog Alarm Sample Rate</b> .....	<b>8-7</b>
	Sample Rate .....	8-7
8.5	<b>Specifying Analog Alarm Process Variable Parameters</b> .....	<b>8-8</b>
	Process Variable Address .....	8-8
	PV Range Low/High .....	8-8
	PV is Bipolar 20% Offset .....	8-8
	Square Root of PV .....	8-8
8.6	<b>Specifying Analog Alarm Deadband</b> .....	<b>8-9</b>
	Alarm Deadband .....	8-9
8.7	<b>Specifying Analog Alarm Process Variable Alarm Limits</b> .....	<b>8-10</b>
	PV Alarms: Low-low, Low, High, High-high .....	8-10
8.8	<b>Specifying Analog Alarm Setpoint Parameters</b> .....	<b>8-11</b>
	Remote Setpoint .....	8-11
	Clamp SP Limits .....	8-11
8.9	<b>Specifying Analog Alarm Special Function Call</b> .....	<b>8-12</b>
	Special Function .....	8-12
8.10	<b>Specifying Analog Alarm Setpoint Deviation Limits</b> .....	<b>8-13</b>
	Deviation Alarms: Yellow, Orange .....	8-13
8.11	<b>Specifying Other Analog Alarm Process Variable Alarms</b> .....	<b>8-14</b>
	Rate of Change Alarm .....	8-14
	Broken Transmitter Alarm .....	8-14

## Chapter 9 Programming Loops

9.1	Overview .....	9-2
9.2	<b>Using the PID Loop Function</b> .....	<b>9-4</b>
	Manual Mode .....	9-4
	Auto Mode .....	9-4
	Cascade Mode .....	9-4
	Changing Loop Mode .....	9-5

---

<b>9.3</b>	<b>Loop Algorithms</b> .....	<b>9-6</b>
	PID Position Algorithm .....	9-6
	PID Velocity Algorithm .....	9-7
<b>9.4</b>	<b>Programming Loops</b> .....	<b>9-8</b>
	Loop Numbers and Variable Names .....	9-8
	Programming Tables .....	9-8
	Loop C-Flags .....	9-9
<b>9.5</b>	<b>Specifying Loop PID Algorithm</b> .....	<b>9-10</b>
	Pos/Vel PID Algorithm .....	9-10
<b>9.6</b>	<b>Specifying Loop V-Flag Address</b> .....	<b>9-11</b>
	Loop V-Flag Address .....	9-11
<b>9.7</b>	<b>Specifying Loop Sample Rate</b> .....	<b>9-12</b>
	Sample Rate .....	9-12
<b>9.8</b>	<b>Specifying Loop Process Variable Parameters</b> .....	<b>9-13</b>
	Process Variable Address .....	9-13
	PV Range Low/high .....	9-13
	PV is Bipolar 20% Offset .....	9-13
	Square Root of PV .....	9-13
<b>9.9</b>	<b>Specifying Loop Ramp/Soak Profile</b> .....	<b>9-14</b>
	Defining Ramp/Soak Operation .....	9-14
	Defining Ramp/Soak Steps .....	9-14
	Controlling the Ramp/Soak Operation .....	9-14
	Ramp/Soak for SP .....	9-15
	Programming Ramp/Soak .....	9-15
<b>9.10</b>	<b>Specifying Loop Output Parameters</b> .....	<b>9-18</b>
	Loop Output Address .....	9-18
	Output is Bipolar .....	9-18
	20% Offset on Output .....	9-18
<b>9.11</b>	<b>Specifying Loop Alarm Deadband</b> .....	<b>9-19</b>
	Alarm Deadband .....	9-19
<b>9.12</b>	<b>Specifying Loop Process Variable Alarm Limits</b> .....	<b>9-20</b>
	PV Alarms Low-low, Low-high, High-high .....	9-20
<b>9.13</b>	<b>Specifying Loop Setpoint Parameters</b> .....	<b>9-21</b>
	Remote Setpoint .....	9-21
	Clamp SP Limits .....	9-21

---

<b>9.14</b>	<b>Specifying Loop Tuning Parameters</b> .....	<b>9-22</b>
	Loop Gain, Reset, Rate .....	9-22
	Removing Integral Action .....	9-22
	Removing Derivative Action .....	9-22
	Removing Proportional Action .....	9-22
	Freeze Bias .....	9-23
	Adjust Bias .....	9-24
<b>9.15</b>	<b>Specifying Loop Derivative Gain Limiting</b> .....	<b>9-25</b>
	Limiting Coefficient .....	9-25
<b>9.16</b>	<b>Specifying Loop Special Function Call</b> .....	<b>9-26</b>
	Special Calculation/ Special Function .....	9-26
	Calculation Scheduled on Setpoint .....	9-26
	Calculation Scheduled on Process Variable .....	9-26
	Calculation Scheduled on Output .....	9-27
<b>9.17</b>	<b>Specifying Loop Locked Changes</b> .....	<b>9-28</b>
	Lock Setpoint, Auto/Manual, Cascade .....	9-28
<b>9.18</b>	<b>Specifying Loop Error Operation</b> .....	<b>9-29</b>
	Error Operation .....	9-29
	Error Deadband .....	9-29
	No Error Calculation .....	9-29
<b>9.19</b>	<b>Specifying Reverse Acting Loops</b> .....	<b>9-30</b>
	Reverse Acting .....	9-30
	Direct-Acting Loop .....	9-30
	Reverse-Acting Loop .....	9-30
<b>9.20</b>	<b>Specifying Loop Setpoint Deviation Limits</b> .....	<b>9-31</b>
	Deviation Alarms Yellow, Orange .....	9-31
<b>9.21</b>	<b>Specifying Other Loop Process Variable Alarms</b> .....	<b>9-32</b>
	Rate of Change Alarm .....	9-32
	Broken Transmitter Alarm .....	9-32
<b>9.22</b>	<b>Using SmarTune Automatic Loop Tuning (555 CPUs Only)</b> .....	<b>9-34</b>
	Overview of SmarTune .....	9-34
	The Loop Tuning Process Equation .....	9-35
	The Proportional Component .....	9-35
	The Integral Component .....	9-36
	The Derivative Component .....	9-37
	Variable Parameters .....	9-38
	Value Parameters .....	9-43

---

## Appendix A Memory and Variable Types

A.1	RLL Variable Access .....	A-2
A.2	SF Program Variable Access .....	A-3

## Appendix B RLL Memory Requirements

B.1	Memory Requirements .....	B-2
-----	---------------------------	-----

## Appendix C Controller Performance

C.1	Calculating Performance .....	C-2
	Calculating Normal Scan Time .....	C-2
	Calculating the Cyclic RLL Execution Time .....	C-4
	Total Scan Time Including Cyclic RLL .....	C-5
C.2	Tuning the Timeline .....	C-8
	Basic Strategy .....	C-8
	Using Peak Elapsed Time Words .....	C-8
	Using the Status Words .....	C-9
	Concepts to Remember When Tuning Timeline .....	C-10
C.3	RLL Execution Times .....	C-12
C.4	SF Program Statement Execution Times .....	C-13

## Appendix D Loop and Analog Alarm Flag Formats

D.1	Loop Flags .....	D-2
D.2	Analog Alarm Flags .....	D-4

## Appendix E Selected Application Examples

E.1	Using the SHRB .....	E-2
E.2	Using the SHRW .....	E-4
E.3	Using the TMR .....	E-6
E.4	Using the BITP .....	E-10
E.5	Using the DRUM .....	E-11
E.6	Using the EDRUM .....	E-13
E.7	Using the MIRW .....	E-17
E.8	Using the MWIR .....	E-20

E.9	Using the MWTT .....	E-24
E.10	Using the MWFT .....	E-26
E.11	Using the WXOR .....	E-28
E.12	Using the CBD .....	E-30
E.13	Using the CDB .....	E-32
E.14	Using the One Shot .....	E-33
E.15	Using the DCAAT .....	E-34
E.16	Using Status Words .....	E-37

## Appendix F Special Function Program Error Codes

### Appendix G Status Words

STW01: Non-fatal Errors .....	G-2
STW02: Base Controller Status .....	G-3
STW03 – STW09: PROFIBUS-DP Slave Status .....	G-4
STW10: Dynamic Scan Time .....	G-4
STW11 – STW138: I/O Module Status .....	G-5
STW11 – STW138: (continued) .....	G-7
STW139: Discrete Force Count .....	G-8
STW140: Word Force Count .....	G-8
STW141 – STW144: Date, Time, and Day of Week .....	G-8
STW145 – STW146: Receive and Timeout Errors .....	G-11
STW147: PROFIBUS-DP Slave Errors .....	G-11
STW148: PROFIBUS-DP Bus Communication Errors .....	G-11
STW149 - STW160: Reserved .....	G-11
STW161: Special Function Processor Fatal Errors .....	G-12
STW162: Special Function Processor Non-fatal Errors .....	G-13
STW163: RLL Subroutine Stack Overflow .....	G-14
STW164 – STW165: L-Memory Checksum C0 .....	G-14
STW166 – STW167: L-Memory Checksum C1 .....	G-14
STW168: Dual RBC Status .....	G-15
STW169 – STW175: Reserved .....	G-16
STW176: Dual Power Supply Status .....	G-16
STW177 – STW183: Reserved .....	G-17
STW184: Module Mismatch Indicator .....	G-17
STW185 – STW191: Reserved .....	G-17
STW192: Discrete Scan Execution Time .....	G-17
STW193 – STW199: Reserved .....	G-17
STW200: User Error Cause .....	G-18
STW201: First Scan Flags .....	G-19

STW202: Application Mode Flags (A – P) .....	G-20
STW203: Application Mode Flags (Q – Z) .....	G-21
STW204: Application Installed Flags (A – P) .....	G-22
STW205: Application Installed Flags (Q – Z) .....	G-23
STW206 – STW207: U-Memory Checksum C0 .....	G-24
STW208 – STW209: U-Memory Checksum C1 .....	G-24
STW210: Base Poll Enable Flags .....	G-25
STW211 – STW217: PROFIBUS-DP Slave Enable Flags .....	G-26
STW218: My Application ID .....	G-26
STW219: RLL Task Overrun .....	G-26
STW220: Interrupting Slots in Local Base .....	G-27
STW221: Module Interrupt Request Count .....	G-27
STW222: Spurious Interrupt Count .....	G-27
STW223 – STW225: Binary Time-of-Day .....	G-28
STW226: Time-of-Day Status .....	G-28
STW227 – STW228: Bus Error Access Address .....	G-30
STW229 – STW230: Bus Error Program Offset .....	G-30
STW231 PROFIBUS-DP I/O System Status .....	G-31
STW232 – STW238: PROFIBUS-DP Slave Diagnostic .....	G-31
STW239 – STW240: CS-Memory Checksum C0 .....	G-32
STW241 – STW242: CS-Memory Checksum C1 .....	G-32

## Appendix H External Subroutine Development

<b>H.1</b>	<b>Designing the External Subroutine .....</b>	<b>H-2</b>
	Program Code Requirements .....	H-2
	Loading the Subroutine .....	H-3
<b>H.2</b>	<b>U-Memory Format .....</b>	<b>H-4</b>
	Header .....	H-4
	Code and Constant Data .....	H-5
	Modifiable Data .....	H-5
	User Stack .....	H-5
<b>H.3</b>	<b>Guidelines for Creating C Language Subroutines .....</b>	<b>H-6</b>
	Debugging the External Subroutine .....	H-6
	Static Data Initialization .....	H-7
	Accessing Discrete/Word Variables .....	H-10
	Floating Point Operations .....	H-11
	Unsupported C Language Features .....	H-11
<b>H.4</b>	<b>Developing an External Subroutine — Example .....</b>	<b>H-12</b>
	Example Header File .....	H-12
	Example Subroutine Source .....	H-14
	Preparing the Load Module .....	H-14
	Loading U-Memory .....	H-16
	Using the External Subroutines in RLL .....	H-16

---

## Appendix I Interboard Communications for the 575

<b>I.1</b>	<b>Using Applications to Enable CPUs to Exchange Data</b> .....	<b>I-2</b>
	Applications .....	I-2
	Overview .....	I-4
	G-Memory Areas .....	I-4
	Required and Optional Applications .....	I-5
	Locking Mode Transitions for Two or More Applications .....	I-6
<b>I.2</b>	<b>Using Direct VMEbus Access to Communicate with Third-Party Boards</b> .....	<b>I-8</b>
	Accessing VMEbus Masters and Slaves Directly .....	I-8
<b>I.3</b>	<b>Coordinating Access to Shared Memory</b> .....	<b>I-10</b>
	Using Locks .....	I-10



## List of Figures

---

1-1	Components for the 545/555/575 System .....	1-3
1-2	Discrete Scan Sequence .....	1-7
1-3	Analog Task Scan Sequence .....	1-8
1-4	Timing Relationship of the Controller Scan Operations .....	1-11
2-1	Format of Signed 16-Bit Integers .....	2-3
2-2	Format of Unsigned 16-Bit Integers .....	2-4
2-3	Format of Signed 32-Bit Integers .....	2-4
2-4	Format of Real Numbers .....	2-5
2-5	Example of Binary-Coded Decimal Values .....	2-6
2-6	Example of Storing an Address .....	2-8
3-1	Image Register Update .....	3-2
3-2	Discrete Image Register .....	3-3
3-3	Word Image Register .....	3-5
3-4	Relation of Hardwired Field Devices and the RLL Program .....	3-7
3-5	Immediate I/O Update .....	3-8
3-6	IORW Instruction .....	3-9
3-7	Immediate I/O Configuration Chart .....	3-10
3-8	Control Relay .....	3-13
3-9	Control Relay Operation .....	3-14
4-1	Controller Memory Types .....	4-2
4-2	PGTS Discrete Parameter Area .....	4-10
4-3	PGTS Word Parameter Area .....	4-10
5-1	Single Rung of a Relay Ladder Logic Program .....	5-2
5-2	Power Flow and the Contact .....	5-3
5-3	Operation of Normal Contact and Electro-mechanical Relay .....	5-4
5-4	Operation of a NOT-ed Contact and Electro-mechanical Relay .....	5-6
5-5	Power Flow and the Coil .....	5-8
5-6	Example of a Box Instruction .....	5-12
5-7	How Relay Ladder Logic Is Solved .....	5-13
5-8	RLL Program Compile Process .....	5-14
5-9	Examples of Cyclic RLL Design .....	5-19
5-10	Example of Cyclic RLL Execution Interrupt .....	5-20
5-11	Relationship of Cyclic RLL Execution Time to Cycle Time .....	5-20
5-12	When Cycle Time Changes Take Effect .....	5-21
5-13	Examples of Cyclic RLL Design .....	5-22
5-14	Status Word 220 Format .....	5-23
5-15	Example RLL Interrupt Program .....	5-24
5-16	Status Word Location of Time Data .....	5-28
5-17	Clock Data Example .....	5-29
5-18	Binary Time of Day .....	5-30
5-19	Time-of-Day Status Word .....	5-31

## List of Figures (continued)

---

6-1	RLL Instruction Format	6-6
6-2	ABSV Format	6-11
6-3	ADD Format	6-12
6-4	BITC Format	6-13
6-5	BITP Format	6-14
6-6	BITS Format	6-15
6-7	CBD Format	6-16
6-8	Examples of CBD Operation	6-17
6-9	CDB Format	6-18
6-10	Examples of CDB Operation	6-19
6-11	CMP Format	6-20
6-12	Coil Format	6-22
6-13	Contact Format	6-23
6-14	CTR Format	6-24
6-15	DCAT Format	6-26
6-16	DCMP Format	6-30
6-17	DIV Format	6-32
6-18	Division Example	6-33
6-19	DRUM Format	6-35
6-20	DSET Format	6-38
6-21	EDRUM Format	6-40
6-22	END Format	6-44
6-23	ENDC Format	6-45
6-24	GTS Format	6-46
6-25	Example Call to Subroutine	6-47
6-26	IMC Format	6-48
6-27	IORW Format	6-50
6-28	JMP Format	6-52
6-29	Example of JMP Zone of Control	6-53
6-30	LDA Format	6-54
6-31	Examples of the LDA Instruction	6-56
6-32	Address/Index Resolution	6-57
6-33	LDC Format	6-59
6-34	LOCK Format	6-60
6-35	Example of the LOCK Instruction	6-62
6-36	MCAT Format	6-63
6-37	MCR Format	6-68
6-38	Example of MCR Control of a Box	6-69
6-39	Example of the MCR Zone of Control	6-70
6-40	MDRMD Format	6-72
6-41	MDRMW Format	6-77
6-42	MIRFT Format	6-82

---

6-43	Example of MIRFT Operation .....	6-83
6-44	MIRTT Format .....	6-84
6-45	Example of MIRTT Operation .....	6-85
6-46	MIRW Format .....	6-86
6-47	Example of MIRW Operation .....	6-87
6-48	MOVE Format .....	6-88
6-49	Examples of the MOVE Instruction .....	6-90
6-50	Address/Source Index Resolution .....	6-94
6-51	Address/Destination Index Resolution .....	6-95
6-52	MOVW Format .....	6-96
6-53	The MOVW Operation .....	6-97
6-54	MULT Format .....	6-98
6-55	Multiplication Example .....	6-99
6-56	MWFT Format .....	6-100
6-57	The MWFT Operation .....	6-101
6-58	MWI Format .....	6-102
6-59	The MWI Operation .....	6-103
6-60	MWIR Format .....	6-104
6-61	The MWIR Format .....	6-105
6-62	MWTT Format .....	6-106
6-63	The MWTT Operation .....	6-107
6-64	NOT Format .....	6-108
6-65	NOT Example .....	6-108
6-66	One Shot Format .....	6-109
6-67	PID Format .....	6-110
6-68	PGTS Format .....	6-112
6-69	PGTS Instruction Example 2 .....	6-114
6-70	PGTS Instruction Example 1 .....	6-117
6-71	PGTSZ Format .....	6-118
6-72	RSD Instruction Format .....	6-120
6-73	RTN Format .....	6-122
6-74	SBR Format .....	6-123
6-75	SBR Example .....	6-124
6-76	SFPGM Format .....	6-126
6-77	SFSUB Format .....	6-128
6-78	SHRB Format .....	6-132
6-79	SHRB Example .....	6-133
6-80	SHRW Format .....	6-134
6-81	SHRW Operation .....	6-135
6-82	SKP / LBL Format .....	6-137
6-83	Example of SKP Zone of Control .....	6-139
6-84	SMC Format .....	6-140

## List of Figures (continued)

---

6-85	SQRT Format	6-142
6-86	STFE Format	6-144
6-87	STFN Format	6-146
6-88	SUB Format	6-148
6-89	TAND Format	6-149
6-90	TASK Format	6-150
6-91	Examples of TASK Design	6-151
6-92	TCMP Format	6-153
6-93	TCPL Format	6-154
6-94	Text Box Format	6-155
6-95	TMR/TMRF Format	6-156
6-96	TOR Format	6-158
6-97	TSET Format	6-159
6-98	TTOW Format	6-160
6-99	TXOR Format	6-162
6-100	UDC Format	6-164
6-101	UNLCK Format	6-167
6-102	WAND Format	6-168
6-103	Result of ANDing Bits	6-168
6-104	Result of ANDing Two Words	6-169
6-105	WOR Format	6-170
6-106	Result of ORing Bits	6-170
6-107	Result of ORing Two Words	6-171
6-108	WROT Format	6-172
6-109	WROT Operation	6-172
6-110	Result of a WROT Operation	6-173
6-111	WTOT Format	6-174
6-112	WTTA Format	6-176
6-113	WTOO Format	6-178
6-114	WTXO Format	6-180
6-115	WXOR Format	6-182
6-116	Result of an Exclusive OR of Bits	6-183
6-117	Result of an Exclusive OR of Two Words	6-183
6-118	XSUB Format	6-184
6-119	Example of the XSUB Instruction	6-186
7-1	SFPGM Instruction Format	7-11
7-2	Special Function Program Format	7-18
7-3	Word Specification for SF Program Errors	7-21
7-4	Example of Valid Entries for the FTSR-IN Statement	7-23
7-5	BCDBIN Format	7-24
7-6	Example of BCDBIN Operation	7-24

---

7-7	BINBCD Format .....	7-25
7-8	Example of BINBCD Operation .....	7-25
7-9	CALL Format .....	7-26
7-10	CDT Format .....	7-28
7-11	CDT Statement Example .....	7-29
7-12	EXIT Format .....	7-30
7-13	FTSR-IN Format .....	7-31
7-14	Example of FTSR-IN Operation .....	7-34
7-15	FTSR-OUT Format .....	7-35
7-16	Example Of FTSR-OUT Operation .....	7-38
7-17	GOTO/LABEL Format .....	7-39
7-18	Example of GOTO/LABEL Statements .....	7-39
7-19	IF Format .....	7-40
7-20	Example of IF/THEN/ELSE Statements .....	7-41
7-21	IMATH Format .....	7-42
7-22	IMATH Statement Example .....	7-43
7-23	LEAD/LAG Format .....	7-44
7-24	MATH Format .....	7-46
7-25	MATH Statement Example .....	7-48
7-26	PACK Format .....	7-51
7-27	Example of PACKing Multiple Blocks of Bits Into Table .....	7-51
7-28	Example of PACKing Bits Into Table .....	7-52
7-29	Example of PACKing Words Into Table .....	7-53
7-30	Example of PACKing Bits and Words Into Table .....	7-53
7-31	Example of PACKing Bits from a Table .....	7-54
7-32	Example of PACKing Multiple Blocks of Bits from a Table .....	7-54
7-33	Example of PACKing Words from a Table .....	7-55
7-34	Example of PACKing Bits and Words from a Table .....	7-55
7-35	PACKAA Format .....	7-56
7-36	Example of PACKAA TO Table Operation .....	7-57
7-37	Example of PACKAA FROM Table Operation .....	7-57
7-38	PACKLOOP Format .....	7-58
7-39	PACKRS Format .....	7-60
7-40	Address Format — Short Form .....	7-62
7-41	Short Form Address Example .....	7-62
7-42	Address Format — Long Form .....	7-63
7-43	Long Form Address Example .....	7-63
7-44	Example of PACKRS to a Table in V-Memory .....	7-64
7-45	Example of PACKRS from a Table in V-Memory .....	7-65
7-46	PRINT Format .....	7-68
7-47	Example of the RETURN Statement .....	7-71
7-48	SCALE Format .....	7-72

## List of Figures (continued)

7-49	SCALE Example .....	7-73
7-50	SDT Format .....	7-74
7-51	SDT Statement Example .....	7-75
7-52	SSR Format .....	7-76
7-53	Example of SSR Operation .....	7-77
7-54	UNSCALE Format .....	7-78
7-55	UNSCALE Example .....	7-79
7-56	Comment Format .....	7-80
8-1	Example of Analog Alarm Application .....	8-2
8-2	Analog Alarm Programming Table .....	8-4
8-3	Example of Alarm Deadband For Analog Alarms .....	8-9
8-4	Example of Broken Transmitter Alarm .....	8-15
9-1	Example of Loop Control .....	9-2
9-2	Loop Programming Table .....	9-8
9-3	Example Ramp/Soak Cycle .....	9-14
9-4	Ramp/Soak Programming Table .....	9-16
9-5	Ramp/Soak Table Examples .....	9-17
9-6	Example of Alarm Deadband For Loops .....	9-19
9-7	Loop Response to the Freeze Bias Option .....	9-23
9-8	Loop Response to the Adjust Bias Option .....	9-24
9-9	Examples of Direct- and Reverse-Acting Control .....	9-30
9-10	Example of Broken Transmitter Alarm .....	9-33
9-11	Proportional Band .....	9-35
9-12	Steady State Error .....	9-36
9-13	Ideal Process Variable Curve .....	9-37
9-14	Example of Activation/Deactivation of Auto Tuning Process .....	9-39
C-1	Loop/Analog Alarm Execution Time for the 545/575* .....	C-7
E-1	SHRB Application Example .....	E-2
E-2	RLL for SHRB Application Example .....	E-3
E-3	20-Bit Shift Register in Discrete Image Register .....	E-3
E-4	SHRW Application Example .....	E-4
E-5	RLL for SHRW Application Example .....	E-5
E-6	TMR Application Example .....	E-6
E-7	RLL for TMR Application Example #1 .....	E-7
E-8	Timing Diagram for TMR Application Example #2 .....	E-8
E-9	RLL for TMR Application Example #2 .....	E-8
E-10	Timing Diagram for TMR Application Example #3 .....	E-9
E-11	RLL for TMR Application Example #3 .....	E-9
E-12	RLL for BITP Application Example .....	E-10
E-13	RLL for DRUM Application Example .....	E-12

---

E-14	RLL for EDRUM Application Example .....	E-15
E-15	MIRW Application Example .....	E-17
E-16	RLL for MIRW Application Example .....	E-19
E-17	RLL for MWIR Application Example (continued on next 2 pages) .....	E-21
E-18	MWTT Application Example .....	E-24
E-19	RLL for MWTT Application Example .....	E-25
E-20	RLL for MWFT Application Example .....	E-27
E-21	RLL for WXOR Application Example .....	E-28
E-22	RLL for CBD Application Example .....	E-31
E-23	RLL for CDB Application Example .....	E-32
E-24	RLL for One Shot Application Example .....	E-33
E-25	Constructing a One Shot From RLL .....	E-33
E-26	DCAT Application Example .....	E-34
E-27	RLL for DCAT Application Example .....	E-35
E-28	RLL for Status Word Application Example .....	E-37
G-1	Example of Status Word Reporting Scan Time .....	G-4
G-2	Example of Status Word Reporting a Module Failure .....	G-7
G-3	Example of Status Words Reporting Time .....	G-10
H-1	Externally Developed Subroutine Code Format .....	H-5
H-2	Initialization Routine Required for Microtec C .....	H-8
H-3	Example of Passing a Discrete Value .....	H-10
H-4	Example of Passing a Pointer .....	H-10
H-5	Example of Passing Normal Values .....	H-10
H-6	Example Assembly Language Header File .....	H-12
H-7	Example Subroutine Source File .....	H-14
H-8	Example Commands for Preparing the Load Module .....	H-14
H-9	Example Link Command File .....	H-15
H-10	Example Subroutine Call for Static Variable Initialization .....	H-16
H-11	Example Call to a Subroutine .....	H-16
I-1	Typical CPU Application .....	I-2
I-2	Accessing G-Memory .....	I-4
I-3	Example of Mode-locked Applications .....	I-6
I-4	Example of Locks and Their Uses .....	I-11
I-5	RLL Example for Locks .....	I-12

## List of Tables

---

1	Release Levels .....	xxv
2-1	Data Type Codes for Controller Memory Areas .....	2-7
3-1	Discrete/Word I/O Permitted .....	3-4
3-2	Logical Points Corresponding to Interrupt Inputs 9 – 16 .....	3-12
3-3	Control Relays Permitted .....	3-13
5-1	RLL Instructions and Condition After Edit .....	5-38
6-1	RLL Functional Groups .....	6-7
6-2	DCAT States .....	6-27
6-3	RSD Buffer Format .....	6-120
7-1	SF Program Statements .....	7-10
7-2	Specifying Real or Integer Parameters .....	7-15
7-3	SF Statement Field Entry Definitions .....	7-22
7-4	Specifying Real or Integer Parameters .....	7-27
7-5	IMATH Operators .....	7-42
7-6	Order of Precedence for IMATH Operators .....	7-43
7-7	MATH Operators .....	7-46
7-8	MATH Intrinsic Functions .....	7-47
7-9	Order of Precedence for MATH Operators .....	7-48
7-10	Analog Alarm Variables .....	7-56
7-11	Loop Variables .....	7-59
8-1	Analog Alarm C-Flags (ACFH and ACFL) .....	8-5
8-2	Analog Alarm V-Flags (AVF) .....	8-6
9-1	Loop C-Flags (LCFH and LCFL) .....	9-9
9-2	Loop V-Flags (LVF) .....	9-11
9-3	Loop Ramp/Soak Flags (LRSF) .....	9-16
9-4	Variable Parameters .....	9-38
9-5	Status Code Bit Values .....	9-41
9-6	Value Parameters .....	9-43
A-1	Controller Variable Types .....	A-2
A-2	Variable Names and Types Used in SF Programs .....	A-3
A-2	Variable Names and Types Used in SF Programs (continued) .....	A-4
A-2	Variable Names and Types Used in SF Programs (continued) .....	A-5
A-3	Bit Format for Words AACK and LACK .....	A-7
B-1	RLL Memory Requirements .....	B-2



---

C-1	Performance and Overrun Indicators .....	C-9
C-2	SF Statement Execution Times for the 545/575 .....	C-13
C-2	SF Statement Execution Times for the 545/575 (continued) .....	C-14
C-2	SF Statement Execution Times for the 545/575 (continued) .....	C-15
D-1	Loop V-Flags (LVF) .....	D-2
D-2	Loop C-Flags (LCFH and LCFL) .....	D-3
D-3	Analog Alarm V-Flags (AVF) .....	D-4
D-4	Analog Alarm C-Flags (ACFH and ACFL) .....	D-4
F-1	Special Function Error Codes .....	F-1
G-1	Status Words 11 Through 138 .....	G-5
G-2	Receive Errors and Timeout Errors for STW145 and STW146 .....	G-11
H-1	Linker Command Functions .....	H-15

# Preface

## Introduction

The *SIMATIC 545/555/575 Programming Reference Manual* contains the information that you need to design an application program for any of these Series 505™ programmable controllers:

- 545–1103, 545–1104, 545–1105, and 545–1106
- 555–1103, 555–1104, 555–1105, and 555–1106
- 575–2104, 575–2105, and 575–2106

This manual describes the complete instruction set for the SIMATIC® controllers listed above.

Additionally, this manual assumes that the programming software and the controller are at the current release at the time of publication, as listed in Table 1. If your controller is at a newer firmware release level, the Release Notes included with your controller or firmware upgrade kit may document new features not covered in this manual.

Table 1 Release Levels

Controller/Software	Release	PowerMath™	SmarTune™
545–1103, 545–1104	4.0	—	—
545–1105, 545–1106	4.2	—	—
555–1103, 555–1104	4.0	—	—
555–1105, 555–1106	5.0	Yes	Yes
575–2104	4.0	—	—
575–2105, 575–2106	5.0	Yes	—
TISOFT	6.3	Yes	—
SoftShop	2.2	Yes	Yes

Refer to the *SIMATIC TI505 Programming Reference User Manual* (PPX:505–8104–x) for information on the following controllers:

- 545–1101 and 545–1102
- 555–1101 and 555–1102
- 575–2101, 575–2102, and 575–2103
- SIMATIC 525/535
- SIMATIC 520C/530C/530T
- SIMATIC 560/565/560T/565P

---

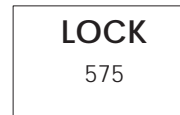
## New Features

Beginning with firmware release 5.0, the 555–1105 and 555–1106 CPUs support several new features, including PowerMath™, SmarTune™, and “fast” PID loops. The 575–2105 and –2106 CPUs support PowerMath.

Additionally, the material about 575 Interboard Communication which was formerly located in the *SIMATIC TI575 System Manual* has been moved to this manual. It is located in Appendix I. For more information, refer to the *SIMATIC 575 Interboard Communication Specification* (PPX:575–8103–x).

## How to Use This Manual

Relay Ladder Logic (RLL) instructions are identified by a mnemonic in a tab in the upper outside corner of the page. In cases where the instruction is restricted to a certain controller or subset of controllers, the tab identifies which controllers can use the instruction. For instance, the LOCK instruction is supported by 575 controllers only, as shown in the example below:



To help you in your program design tasks, Status Words and performance data for all controller models are provided in the appendices.

This manual is not intended to be a primer on RLL or Special Function (SF) programming techniques. If you are not familiar with the techniques of RLL programming or of loop dynamics, you should refer to other documentation or call your Siemens Energy & Automation, Inc., distributor or sales office for technical assistance.

If you need assistance in contacting your distributor or sales office in the United States, call 1–800–964–4114. Training classes in RLL and Special Function programming are available at a number of locations. Contact your distributor for more information. Because there are references to various hardware components, you should review the appropriate hardware and installation manuals for your controller as you design your programs.

## SIMATIC 505 SoftShop for Windows

SIMATIC 505 SoftShop™ for Windows® is a Windows-based programming software for IBM®-compatible personal computers that supports all SIMATIC 505 series programmable controllers. SoftShop for Windows Release 2.2 or greater is required to support all the new features in the 555–1105 and 555–1106 CPUs. Refer to your SoftShop manual for detailed instructions about how to enter a program.

## TISOFT Programming Software

TISOFT™ Release 6.3 or later, is an MS-DOS-based programming software package that supports the full instruction set for SIMATIC 505 controllers. TISOFT, however, does not support SmarTune automatic loop tuning, the PID box instruction, PID loops above 64, or analog alarms above 128.

---

## Manual Contents

Topics are listed below by chapter:

- Chapter 1 gives an overview of the components of the Series 505 systems, local remote I/O, PROFIBUS-DP I/O, the concept of I/O numbering and the hardware/software interface.
- Chapter 2 describes the formats used to represent data types.
- Chapter 3 describes how I/O is read and updated.
- Chapter 4 describes the various controller memory types.
- Chapter 5 presents programming concepts.
- Chapter 6 describes the RLL instructions.
- Chapter 7 describes the Special Function Program statements.
- Chapter 8 describes analog alarm programming.
- Chapter 9 describes loop programming.
- Appendix A lists all the variables used by Series 505 controllers.
- Appendix B lists the RLL instructions, the amount of memory each requires, and instruction numbering guidelines.
- Appendix C gives information needed to calculate controller program scan times.
- Appendix D provides the formats for the loop and analog alarm flags.
- Appendix E gives application examples for selected RLL instructions.
- Appendix F lists the Special Function Program error codes.
- Appendix G lists the status words supported by the Series 505 controllers.
- Appendix H describes how to design an external subroutine, and includes an example subroutine.
- Appendix I describes interboard communication for 575 controllers.

---

Technical  
Assistance

For technical assistance, contact your Siemens Energy & Automation, Inc., distributor or sales office. If you need assistance in contacting your sales agent or distributor in the United States, call 1-800-964-4114.

For additional technical assistance, call the Siemens Technical Services Group in Johnson City, Tennessee at 423-461-2522, or contact them by e-mail at **simatic.hotline@sea.siemens.com**. For technical assistance outside the United States, call 49-911-895-7000.

# Chapter 1

## Series 505 System Overview

---

<b>1.1</b>	<b>The 545, 555, and 575 Systems</b> .....	<b>1-2</b>
	System Components .....	1-2
	Using PROFIBUS-DP I/O .....	1-2
	Local I/O .....	1-2
	Expansion I/O Channels .....	1-2
	Series 505 Remote I/O .....	1-2
	PROFIBUS-DP I/O .....	1-4
	Output Response on PROFIBUS-DP Slave Devices .....	1-4
	Assigning I/O Point Numbers .....	1-5
<b>1.2</b>	<b>Program Execution Operations</b> .....	<b>1-6</b>
	CPU Scan Operations .....	1-6
	Interrupt RLL Execution .....	1-6
	Cyclic RLL Execution .....	1-6
	Discrete Scan .....	1-6
	Analog Task Processing .....	1-8
	Cyclic Analog Tasks .....	1-8
	Non-cyclic Analog Tasks .....	1-9
	Setting the Scan .....	1-10

## 1.1 The 545, 555, and 575 Systems

---

**System Components** The programmable controller interacts with your equipment through input/output (I/O) modules that relay information between the equipment and the programmable controller. When you design your program, you need to know the physical and logical configuration of these I/O modules, how your equipment is connected to them, and how they are addressed and accessed. The relationships among the system components of the 545, 555, and 575 systems are illustrated in Figure 1-1. The 575 system has essentially the same functionality as the 545/555, with the distinction that the 575 local base is a VMEbus. For details about hardware components and installation, refer to the *SIMATIC 545/555/575 System Manual* PPX:505-8201-x).

---

**NOTE:** In this manual, a feature, unless it is explicitly restricted, applies to all systems.

---

**Using PROFIBUS-DP I/O** The 545, 555, and 575 programmable control systems now possess the capability to communicate with PROFIBUS-DP I/O and other devices that meet the PROFIBUS standard (DIN 19245, Part 3). The Series 505 CPUs described in this manual contain a PROFIBUS-DP I/O communication port (via an optional annex card for the 575 and 545-1103/-1105 CPUs) that supports all I/O devices that conform to the PROFIBUS-DP standard.

For information about the PROFIBUS-DP I/O port, see the *SIMATIC 545/555/575 System Manual*. To configure PROFIBUS-DP I/O, Series 505 users must use the COM PROFIBUS configuration utility in conjunction with TISOFT; consult the *SIMATIC 505 TISOFT2 User Manual*.

**Local I/O** Local I/O comprises those modules located in the same base assembly as the programmable controller. The base containing the local I/O is numbered 0. Only Series 505 I/O modules can be installed in the local base.

**Expansion I/O Channels** Two channels are available for expansion I/O. The Series 505 remote I/O channel supports Series 505 and Series 500 remote I/O. The PROFIBUS-DP I/O channel supports PROFIBUS-DP I/O slaves and field devices, and also Series 505 remote I/O (by means of the 505 PROFIBUS-DP RBC, PPX:505-6870).

**Series 505 Remote I/O** When you use the Series 505 remote I/O channel, you can connect up to 15 additional base assemblies with remote I/O modules to the system. These are numbered 1-15.

---

**NOTE:** The 575 CPU requires an annex card, PPX:575-2126, in order to use the Series 505 remote I/O channel. The 545-1103/-1105 CPU cannot use the Series 505 remote I/O channel.

---

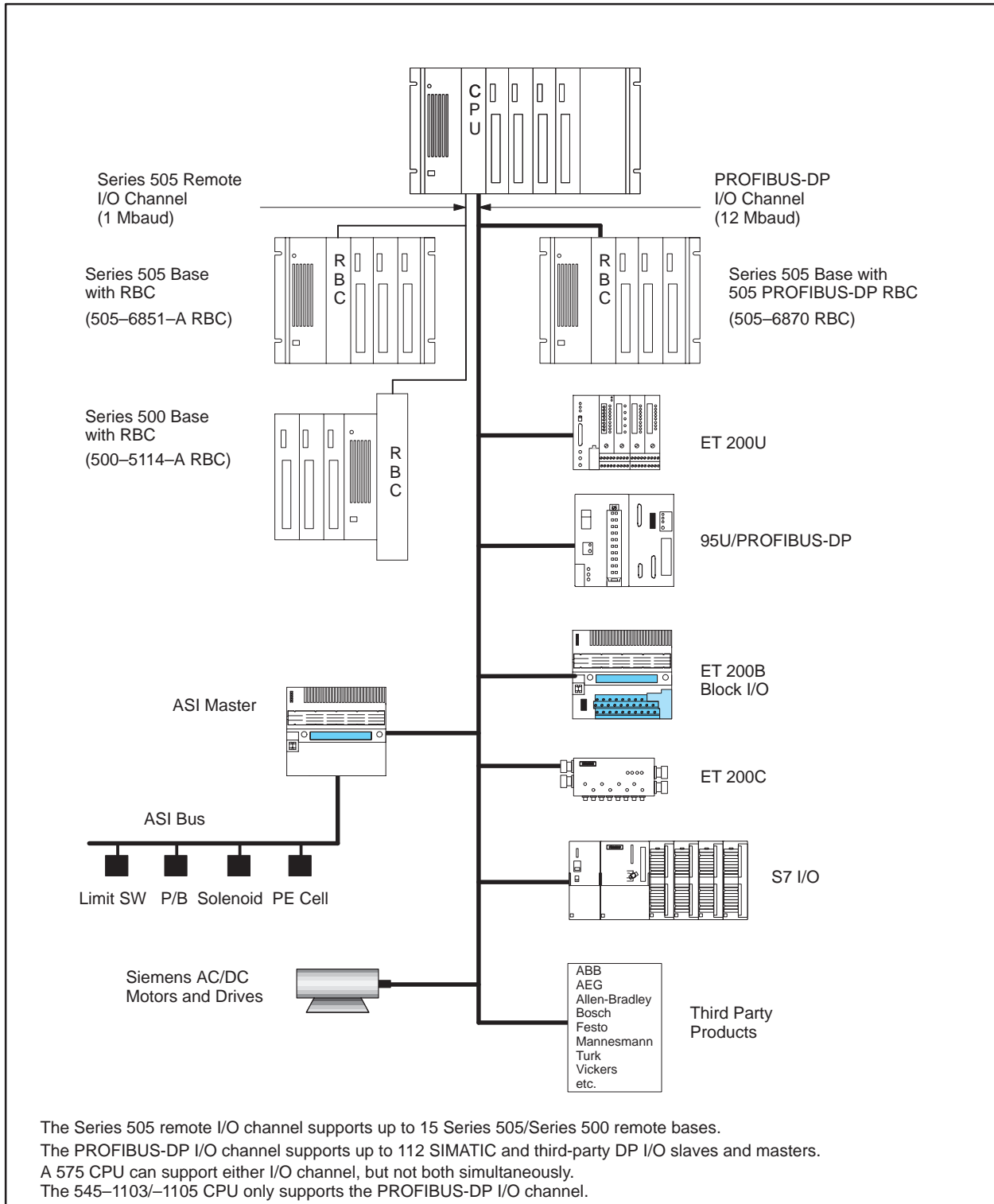


Figure 1-1 Components for the 545/555/575 System



## The 545, 555, and 575 Systems (continued)

---

### PROFIBUS-DP I/O

When you use the PROFIBUS-DP I/O channel, you can connect up to 32 PROFIBUS-DP-compatible I/O slaves and masters with the 545–1103/–1105 CPUs, or up to 112 slaves and masters with the 545–1104/–1106, 555, and 575 CPUs.

---

**NOTE:** Special Function modules cannot be installed on the PROFIBUS-DP I/O channel.

---

### Output Response on PROFIBUS-DP Slave Devices

The response of output points on ET200B and other PROFIBUS-DP slave devices that are connected to the PROFIBUS-DP I/O channel is different from the response of devices on the normal 505 I/O channel.

If network communications are interrupted, or if you power-cycle the master CPU, the outputs of most PROFIBUS-DP slaves momentarily transition to a zero state when communications are reestablished.

### **WARNING**

The outputs of most PROFIBUS-DP slaves momentarily transition to a zero state when communications are reestablished following a power cycle or interruption in network communications.

If output points are expected to retain the last valid state received from the controlling device prior to the communication interruption, erratic operation of your process may result. This could cause unpredictable operation by the controller or network, which could result in death or serious injury to personnel, and/or damage to equipment.

Follow the solutions listed below to avoid unexpected process operation.

- Define your system to withstand the output transition to a zero state when communications are reestablished following a power cycle or interruption in network communications.
- Use slaves that support the “Fail-Safe” feature, which is a recent enhancement to the PROFIBUS-DP Standard DIN 19245 Part 3. The Fail-Safe feature, combined with an appropriately designed control program, can prevent the slave outputs from transitioning to zero when you power up the CPU or reestablish network communications. The 505–6870 Remote Base Controller provides the Fail-Safe feature.

If fail-safe slaves are not used, you must ensure successful process control through your program design or other external means in the event of an error condition such as a loss of communication to slave devices due to cable breaks or a power-cycle of the controlling CPU.

For additional information, contact your distributor.

---

## Assigning I/O Point Numbers

You must assign the I/O point and slot numbers from the I/O Configuration Chart on your programming device. The programmable controller does not update discrete or word I/O points in non-configured I/O modules. Refer to your TISOFT user manual for instructions about configuring the I/O.

For the 545–1103/–1105, a maximum of 1024 I/O points can be assigned. Up to 4096 control relays are available.

For the 545–1104/–1106, a maximum of 2048 I/O points can be assigned. Of these, up to 1024 can be discrete or word points, which must be numbered 1–1024. The next 1024 points are discrete only. Up to 32,768 control relays are available.

For the 555 CPUs, a maximum of 8192 I/O points can be assigned in any mix of discrete and word I/O. Up to 32,768 control relays are available.

For the 575 CPUs, a maximum of 8192 I/O points can be assigned. Up to 23,552 control relays are available.

You do not need to assign I/O point numbers consecutively. For example, in a remote system, Base 2 can be assigned I/O points 897–960. If a base is configured and the modules in the base do not match the configuration, the programmable controller logs a non-fatal error. Misconfigured modules are not accessed by your program. Inputs are read as 0; outputs are ignored.

A Special Function Module is divided into the I/O portion and the special function portion. When a Special Function Module is inserted into a system, the special function portion of the module is automatically logged in, and can send data to and receive data from the controller.

---

**NOTE:** You must configure the I/O portion so that the controller updates the I/O points. Non-special function modules are not logged in automatically.

---

## 1.2 Program Execution Operations

---

### CPU Scan Operations

The 545 and 555 controllers execute four scan operations during the programmable controller scan.

- Interrupt RLL execution
- Discrete scan
- Cyclic RLL execution
- Analog task processing

The 575 controllers share the same operations, except that the 575 CPUs do not execute interrupt RLL.

### Interrupt RLL Execution

The interrupt I/O feature allows you to program an immediate response to a field input transition (interrupt request) from your application. Interrupt I/O operation requires the use of at least one Interrupt Input Module (e.g., PPX:505-4317) installed in the local base. See Section 3.4 for more information on interrupt I/O operation.

### Cyclic RLL Execution

A cyclic RLL program consists of a section of ladder logic, usually short for quick cycle times, that runs independently of the main RLL program. Cyclic RLL is executed periodically throughout the entire programmable controller scan, interrupting the discrete scan and the analog scan as necessary. Because the execution of a cyclic RLL task is not synchronized with the I/O update, use the immediate I/O instructions to access the I/O.

### Discrete Scan

The discrete scan consists of three primary tasks that are executed sequentially and at a rate that can be user-specified.

**Normal I/O Update.** During the normal I/O cycle update, the programmable controller writes data from the image registers to the outputs, and stores data from the inputs into the image registers. The length of the I/O update cycle is dependent upon the number of bases and types of modules (analog, discrete, or intelligent). All I/O points are fully updated each scan.

**Main Ladder Logic Cycle.** The programmable controller executes the main RLL task.

**Special Function Module Communication.** Communication with special function (SF) modules, e.g., NIM, BASIC, PEERLINK™, etc., consists of the following actions:

- Service requests from a previous scan for which processing has been completed are transmitted to the SF modules.
- Remote bases are polled for initial SF module service requests.
- Remote base communication ports are polled for service requests.
- Service requests from SF modules and remote base communication ports are processed.

---

Each SF module that requires service increases the scan time, depending upon the type of module and task. Each type of module is allowed a certain number of service requests per scan. Once these are completed, this function is terminated. Some service requests can be deferred, and these are processed during the analog task time slice described in Figure 1-2.

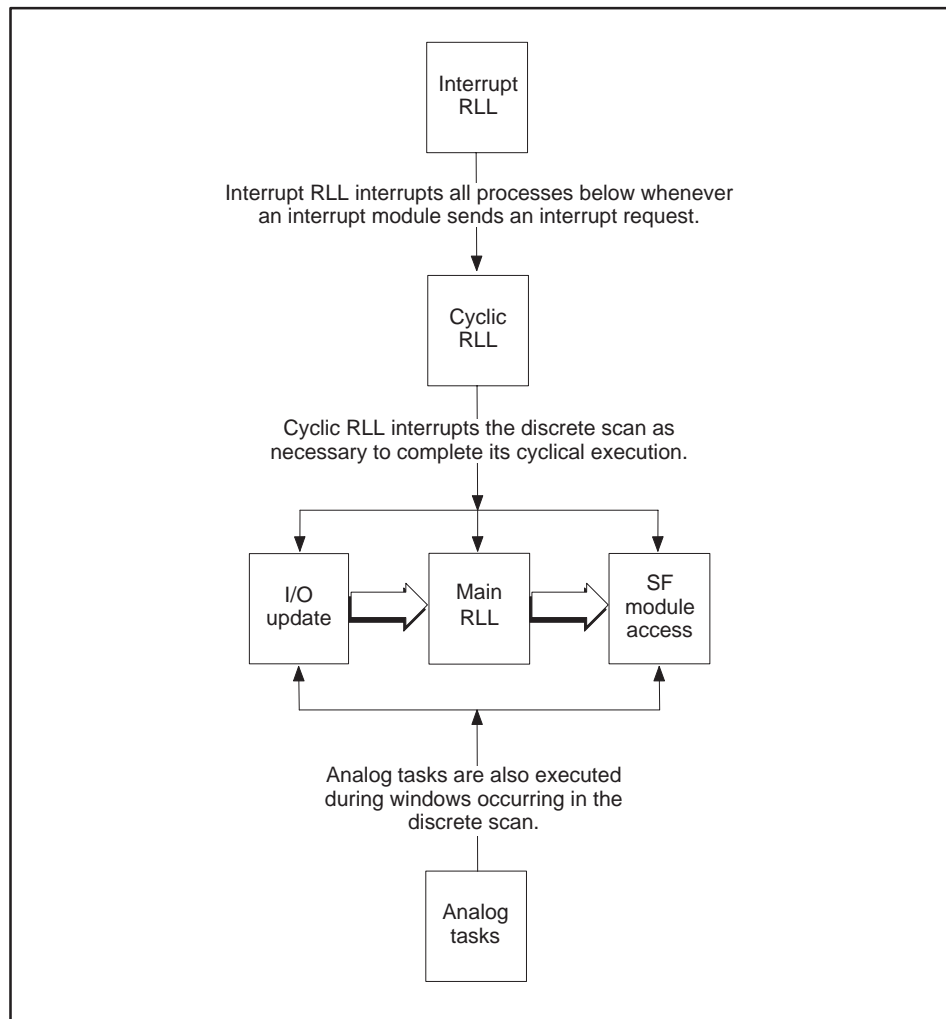


Figure 1-2 Discrete Scan Sequence

## Program Execution Operations (continued)

### Analog Task Processing

The analog portion of the scan is composed of five general types of tasks (Figure 1-3), which are cyclical or non-cyclical in their execution.

Analog tasks are guaranteed execution once per scan, following the discrete scan, provided there is processing to be done. Analog tasks are also processed during windows of suspended activity that occur during the normal I/O and SF portions of the scan. RLL execution is not interrupted by analog tasks.

You can adjust the amount of time spent per controller scan for all analog tasks, except diagnostics, with a programming unit and using AUX Function 19. The time allocation for a given analog task is referred to as its time slice.

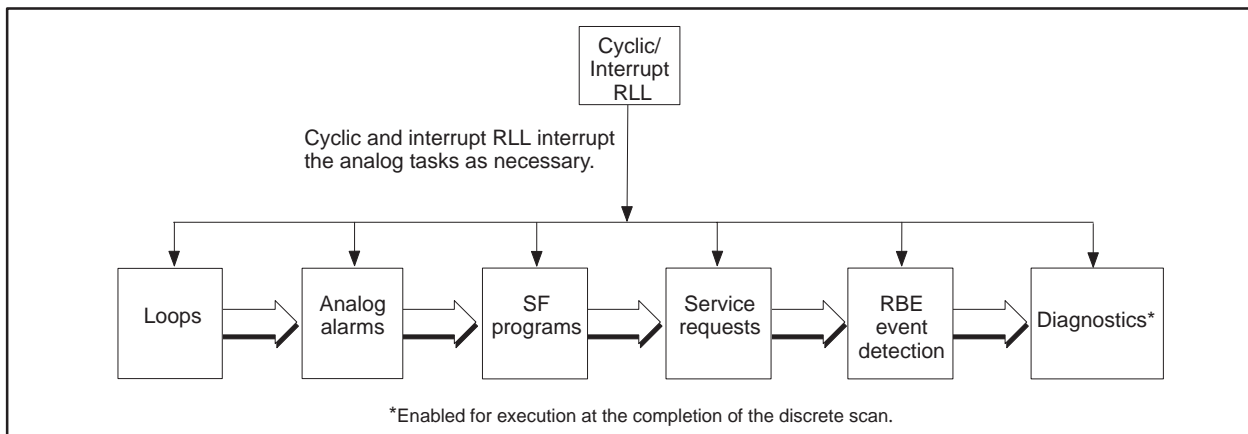


Figure 1-3 Analog Task Scan Sequence

### Cyclic Analog Tasks

The following types of processes are executed cyclically. Each has a sample rate which determines how often it is executed.

- Loops
- Analog alarms
- Cyclic SF programs

The programmable controller has a separate task to execute each type of cyclic process. When enabled, each cyclic process is placed in the execution queue that is managed by the task responsible for executing that type of process.

---

The cyclic processes are time-ordered in their individual queues according to when each process is rescheduled for execution, relative to the other cyclic processes within the same queues. The process with the highest priority (closest to overrunning) is executed first. The process is executed until it is completed or until the time specified for that particular task's time slice expires. If the executing process is completed before the time slice expires, the process with the next highest priority is executed. If the time slice expires before the process is completed, the process (and the task) is put on hold in its current position.

The programmable controller then advances to the next analog task. When the programmable controller sequences through its operations and returns to an analog task with a cyclic process on hold, the process resumes execution from the hold point, unless a higher priority process was scheduled since the last respective time slice. If a process in a cyclic time slice is not finished executing when it is scheduled to execute again, an overrun flag is set.

Restricted SF programs, which are called by loops or analog alarms, are executed from within the loop or analog alarm tasks. Therefore, their execution time is included within the loop or analog alarm time slice.

---

**NOTE:** For CPUs that support PowerMath, while a compiled SF program is executing, a higher priority process on the queue will not execute until the program terminates.

---

SF subroutines, which are called by SF programs or other SF subroutines, are processed during the calling program's time slice.

#### Non-cyclic Analog Tasks

The following types of processes are executed non-cyclically:

- Priority/Non-priority SF programs.
- RLL-requested SF subroutines.
- Service request messages.
- Report by Exception (RBE) event detection.
- Run-time diagnostics.

## Program Execution Operations (continued)

---

Priority and Non-Priority SF Programs are non-cyclic processes that are queued when the SFPGM RLL box instruction receives power flow. There is an analog task that executes priority SF programs, and another analog task that executes non-priority SF programs. These processes are executed in the order that they are queued in the appropriate task's execution queue. When the programmable controller completes one of these processes, it removes the process from the respective queue and turns on the SFPGM output. There are no overrun flags associated with these processes.

RLL-requested SF Subroutines are queued into one of two SFSUB queues when the SFSUB RLL box instruction receives power flow. One queue handles SFSUB 0 instructions and the other handles all other SFSUB instructions.

Service Requests received from the communication ports are placed on one of two communications queues. Read and write commands are placed on the priority communication queue for fastest response. Commands that may require several scans to complete, e.g., program edits and the TISOFT FIND function, are placed in a non-priority communications queue.

Report By Exception event detection task only executes when the programmable controller is used with SIMATIC® PCS™, Release 3.0 or later. The RBE event detection task monitors PCS-defined process events and notifies PCS when an event is detected.

Run-time Diagnostics are enabled for execution at the completion of the discrete scan. The time slice for diagnostics is 1 ms and cannot be changed.

### Setting the Scan

The 545/555/575 scan is defined as the time between normal I/O updates. You can set the scan for the controller as follows.

- **Fixed** — The programmable controller starts a new discrete scan at the specified time interval. The controller executes the discrete scan once and then cycles to the analog scan portion, executing the analog tasks at least one time. If the analog tasks are completed within the specified time, the controller goes into a loop mode (processing analog tasks or idling) until time to start the next scan.

A scan overrun status bit is set (bit 14 in Status Word 1) if the total execution time for the discrete scan portion and the first execution of the analog scan portion exceeds the fixed scan time.

- **Variable** — The programmable controller executes all tasks once and then starts a new scan. All discrete and analog tasks are guaranteed one execution per scan. Specify variable scan for the fastest possible execution of the discrete scan.

- **Variable with upper limit** — The programmable controller executes the discrete scan once and then executes the analog tasks. The controller remains in the analog portion of the scan as long as there are analog tasks to be done. When the upper time limit expires, or no analog tasks require processing, a new scan is begun.

The analog scan portion is executed at least one time. A scan overrun status bit is set if the total execution time for the discrete scan portion and the first execution of the analog scan portion exceeds the upper limit.

Cycle time for the cyclic RLL can be a fixed value or a user-specified variable. As a variable, the cycle time can be changed by logic in your application program. If the cyclic RLL completes execution in less than the specified cycle time, execution does not resume until the next cycle begins. The programmable controller scan time is extended by the amount of time to execute the cyclic RLL multiplied by the number of times the cyclic RLL is executed during the programmable controller scan.

The timing relationship of the scan operations is shown in Figure 1-4. Refer to the Appendix C for details about how to configure the time slices.

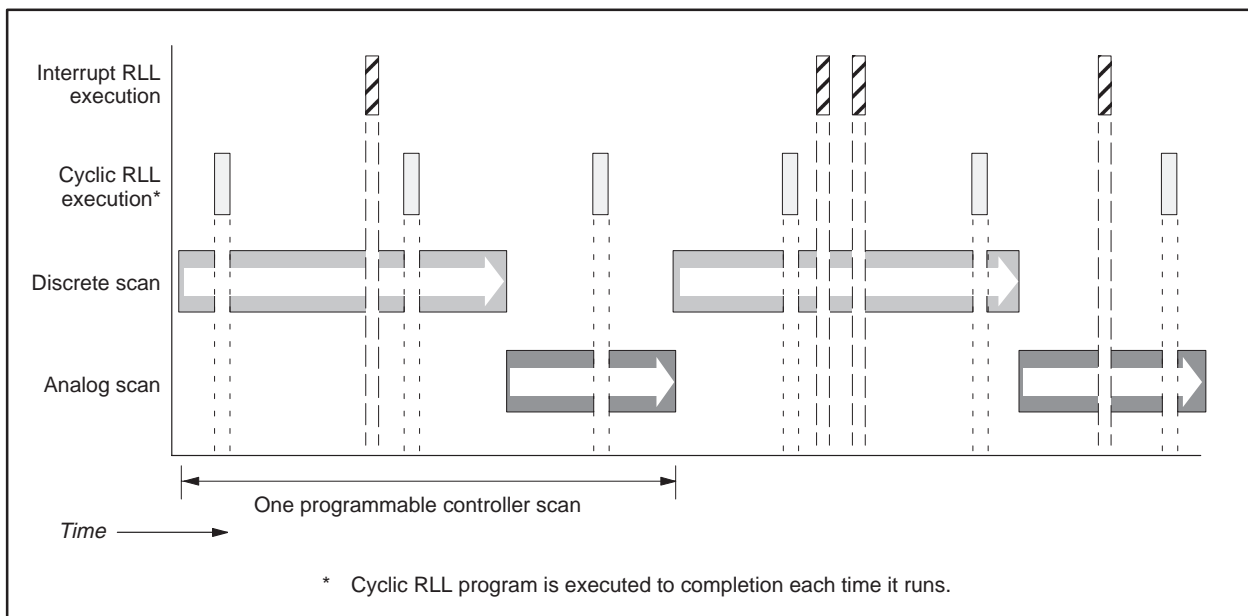


Figure 1-4 Timing Relationship of the Controller Scan Operations



# Chapter 2

## Data Representation

---

<b>2.1</b>	<b>Definitions</b> .....	<b>2-2</b>
	Byte .....	2-2
	Word .....	2-2
	Long Word .....	2-2
	Image Register .....	2-2
	I/O Point .....	2-2
<b>2.2</b>	<b>Integers</b> .....	<b>2-3</b>
	Signed 16-Bit Integers .....	2-3
	Unsigned 16-Bit Integers .....	2-4
	Signed 32-Bit Integers .....	2-4
<b>2.3</b>	<b>Real Numbers</b> .....	<b>2-5</b>
<b>2.4</b>	<b>Binary-Coded Decimal</b> .....	<b>2-6</b>
<b>2.5</b>	<b>Format for an Address Stored in a Memory Location</b> .....	<b>2-7</b>

## 2.1 Definitions

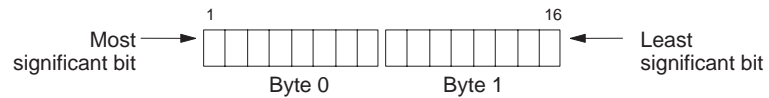
---

The terms listed below are used throughout this manual and have the following meanings.

**Byte** A byte consists of 8 contiguous bits.

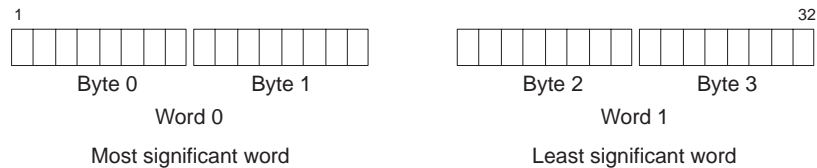


**Word** A word consists of 2 contiguous bytes, 16 bits.



For example, the contents of V-Memory word V100 occupy 16 contiguous bits; the word output WY551 occupies 16 contiguous bits.

**Long Word** A long word consists of 2 contiguous words, 32 bits, that represent a single value.



For example, the contents of V-Memory long word V693 occupy two contiguous words (32 bits), V693 and V694. The next available address is V695, which can represent a word (16 bits) or another long word (32 bits).

**Image Register** The image register is a reserved memory area used to store the value of all discrete (on/off) and word I/O data. Discrete I/O data is contained in the discrete image register. Word I/O data is stored in the word image register. See Section 3.1 for a more complete discussion of the function of the image register.

**I/O Point** An I/O point consists of an I/O type and a reference number that represent a location in the image register. An I/O point that represents a discrete bit in the discrete image register is identified as an X or Y I/O type. An I/O point that represents a word in the word image register is identified as a WX or WY I/O type.

## 2.2 Integers

### Signed 16-Bit Integers

Signed integers are stored as 16-bit words in the two's complement format as shown in Figure 2-1. The 16-bit format allows you to store values ranging from  $-32,768$  to  $+32,767$  (decimal integer values). When bit 1 (the sign bit) is 0, the number is positive; when bit 1 is 1, the number is negative.

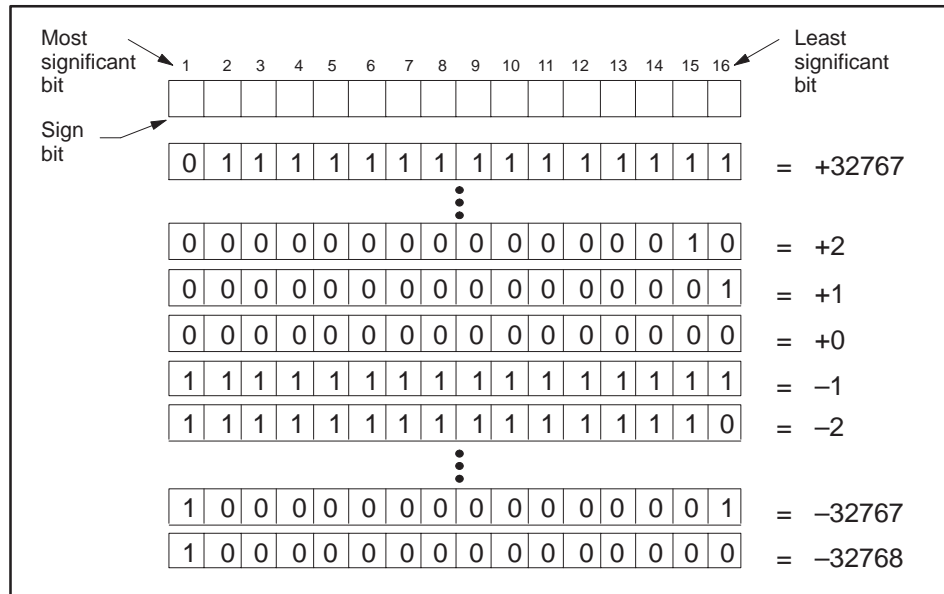


Figure 2-1 Format of Signed 16-Bit Integers

## Integers (continued)

### Unsigned 16-Bit Integers

You can display data on your programming unit as an unsigned integer. The 16-bit format allows you to display integer values ranging from 0 to 65535 as shown in Figure 2-2.

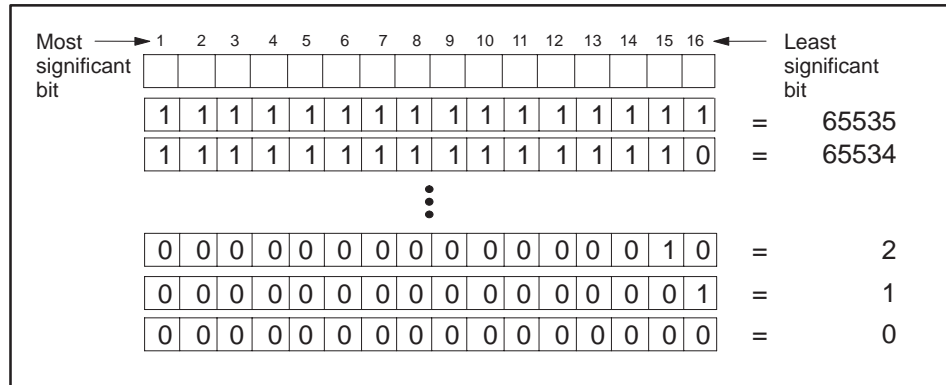


Figure 2-2 Format of Unsigned 16-Bit Integers

### Signed 32-Bit Integers

Thirty-two bit signed long word integers are stored as 32-bit long words in the two's complement format, as shown in Figure 2-3:

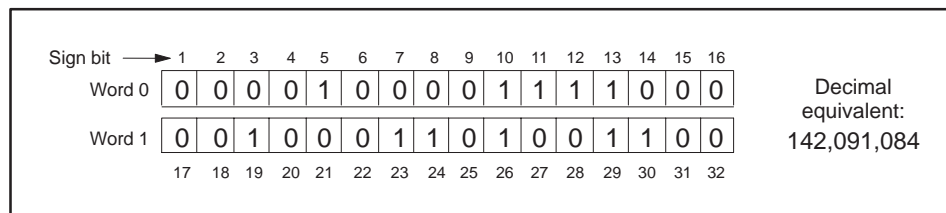


Figure 2-3 Format of Signed 32-Bit Integers

## 2.3 Real Numbers

---

Real numbers are stored in the single-precision 32-bit (two words) binary format (Figure 2-4). Refer to ANSI/IEEE Standard 754–1985 for details about the format.

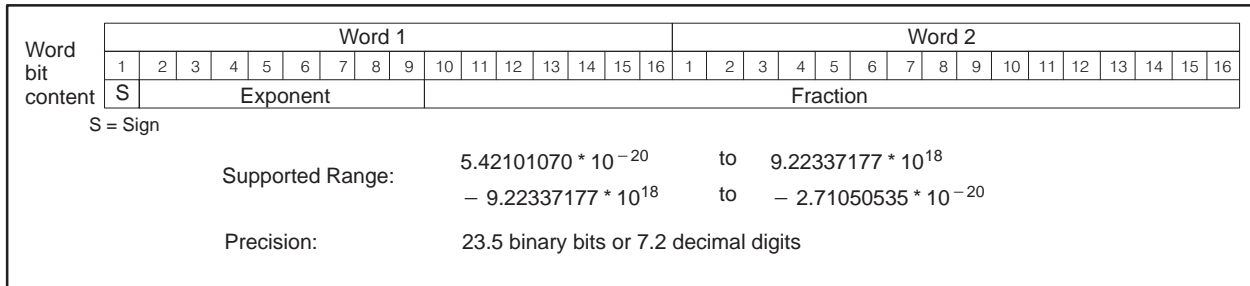


Figure 2-4 Format of Real Numbers

## 2.4 Binary-Coded Decimal

---

Individual BCD digits from a BCD field device are stored in a word in groups of four bits. For example, the number 0582 is stored as shown in Figure 2-5:

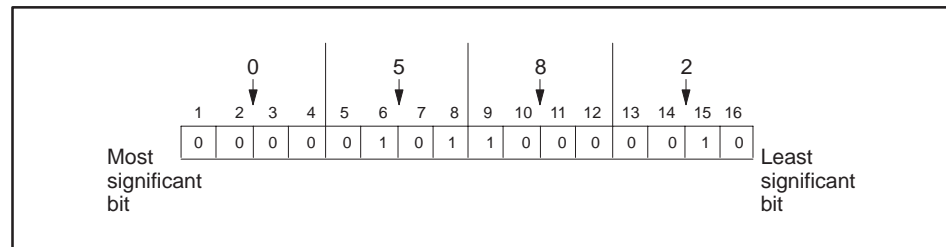


Figure 2-5 Example of Binary-Coded Decimal Values

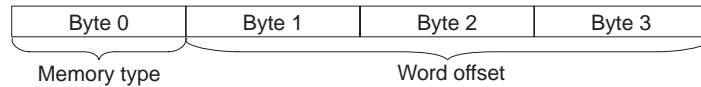
Each digit of the BCD value must be less than or equal to 9. The binary values 1010, 1011, 1100, 1101, 1110, and 1111 are invalid.

Normally, you would convert a BCD value to the binary format, as described in Section 6.9, using the resulting value elsewhere in your program.

## 2.5 Format for an Address Stored in a Memory Location

The Load Address (LDA) instruction allows you to store a memory address in a memory location. A description of LDA and examples of its usage are given in Section 6.26.

When you use LDA to store an address in a memory location, one long word is required, as shown below.



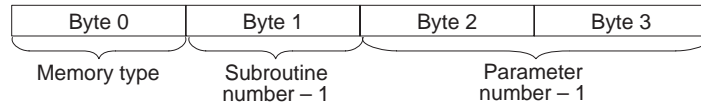
The memory data type is contained in byte 0. The word offset relative to the base address for the data type is contained in bytes 1–3. Data type codes are listed in Table 2-1.

Table 2-1 Data Type Codes for Controller Memory Areas

Memory Area	Data Type (Hex)	Memory Area	Data Type (Hex)
Subroutine work area	00	Application G Global Variables	E8
Variable	01	Application H Global Variables	E7
Constant	02	Application I Global Variables	E6
Word Input	09	Application J Global Variables	E5
Word Output	0A	Application K Global Variables	E4
Timer/Counter Preset	0E	Application L Global Variables	E3
Timer/Counter Current	0F	Application M Global Variables	E2
Drum Step Preset	10	Application N Global Variables	E1
Drum Step Current	11	Application O Global Variables	E0
Drum Count Preset	12	Application P Global Variables	DF
Status Word	1A	Application Q Global Variables	DE
Drum Count Current	1B	Application R Global Variables	DD
VME A24 Space	D3	Application S Global Variables	DC
VME A16 Space	D4	Application T Global Variables	DB
My Global Variables	EF	Application U Global Variables	DA
Application A Global Variables	EE	Application V Global Variables	D9
Application B Global Variables	ED	Application W Global Variables	D8
Application C Global Variables	EC	Application X Global Variables	D7
Application D Global Variables	EB	Application Y Global Variables	D6
Application E Global Variables	EA	Application Z Global Variables	D5
Application F Global Variables	E9		

## Format for an Address Stored in a Memory Location (continued)

The format for logical addresses in the subroutine work areas differs from the other data types, as shown below.



For example, WY77 is stored in V100 and V101 as shown in Figure 2-6. The code for the WY data type is 0A. The decimal offset for the 77th word is 76, which is 00004C in hex.

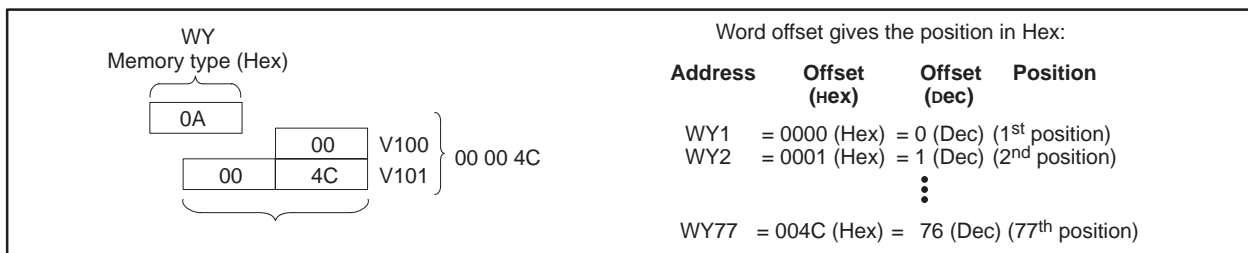


Figure 2-6 Example of Storing an Address

**NOTE:** An address always references a word boundary.

### ! WARNING

**The address that is copied to the destination is a logical, not a physical, address. The misuse of this address could cause an unsafe condition that could result in death or serious injury to personnel, and/or damage to equipment.**

**Do not use this address as a pointer within an external subroutine.**

**NOTE:** The data type codes are provided to give assistance when you decode information displayed in TISOFT. You do not have to enter a data type when you program an LDA. For example, to load V-Memory address V15, enter V15 in field A of the LDA instruction, not 0100 000E.



# Chapter 3

## I/O Concepts

---

<b>3.1</b>	<b>Reading and Updating the I/O</b> .....	<b>3-2</b>
	Discrete Image Register .....	3-3
	Word Image Register .....	3-5
<b>3.2</b>	<b>Normal I/O Updates</b> .....	<b>3-6</b>
	Discrete Control .....	3-6
	Analog Control .....	3-6
<b>3.3</b>	<b>High Speed I/O Updates</b> .....	<b>3-8</b>
	Immediate I/O .....	3-8
	Modules that Support Immediate I/O .....	3-10
	Configuring Immediate I/O .....	3-10
<b>3.4</b>	<b>Interrupt I/O Operation</b> .....	<b>3-11</b>
	Overview .....	3-11
	Configuring the Interrupt Input Module .....	3-11
<b>3.5</b>	<b>Control Relays</b> .....	<b>3-13</b>
	Using Retentive and Non-retentive Control Relays .....	3-14

## 3.1 Reading and Updating the I/O

In normal operation the controller updates outputs, reads inputs, and solves the user application program. The I/O update is shown in Figure 3-1. The Series 505 controllers have reserved memory areas for storing the value of all discrete and word I/O points. Discrete I/O values are contained in the discrete image register, which provides storage for all discrete (on/off) I/O points. Word values are stored in the word image register, which provides storage for word and analog data.

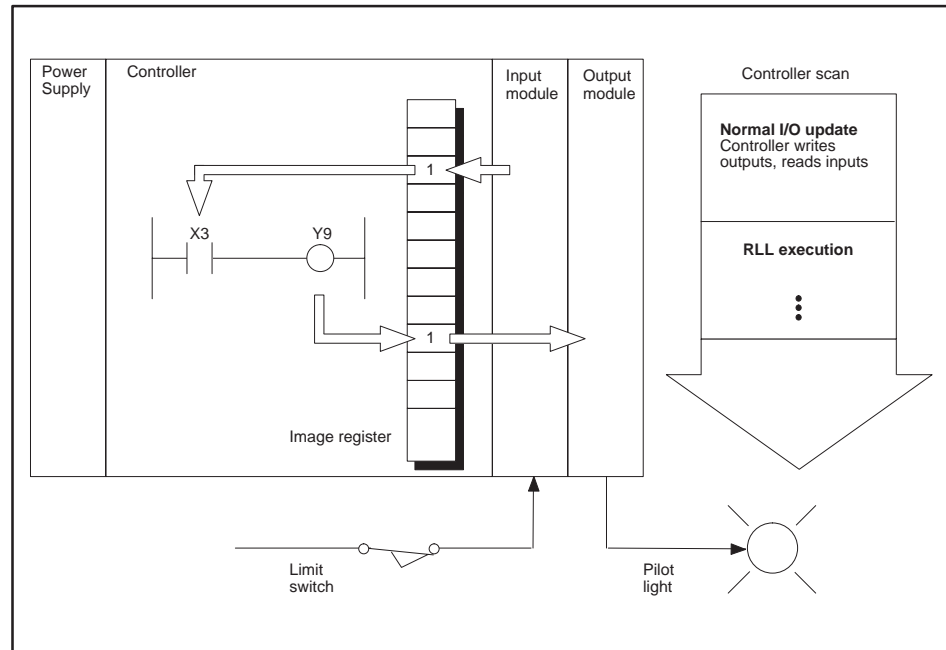


Figure 3-1 Image Register Update

Following the I/O update, the image registers hold the latest value of all discrete and word inputs. As the user program is executed, new values for discrete/word outputs are stored in the image registers. At the completion of the user program, the controller begins a new cycle. The I/O is updated: the results of the last program execution are written from the image registers to the discrete/word outputs, and new values are read for use in the user program. Then the user program is executed.

---

**Discrete Image Register**

An area of memory within the controller called the discrete image register (Figure 3-2) is reserved for maintaining the status of all discrete inputs and outputs.

As a troubleshooting tool, you can use a programming device to force an I/O point on or off. A record of the forced state of a discrete I/O point is kept by the force attribute bit, also shown in Figure 3-2. There is a one-bit location for each of the discrete I/O points. If a discrete I/O point is forced to a particular state, that point remains in that state and does not change until it is forced to the opposite state or is unforced. A power cycle does not alter the value of a forced discrete I/O point as long as the controller battery is good.

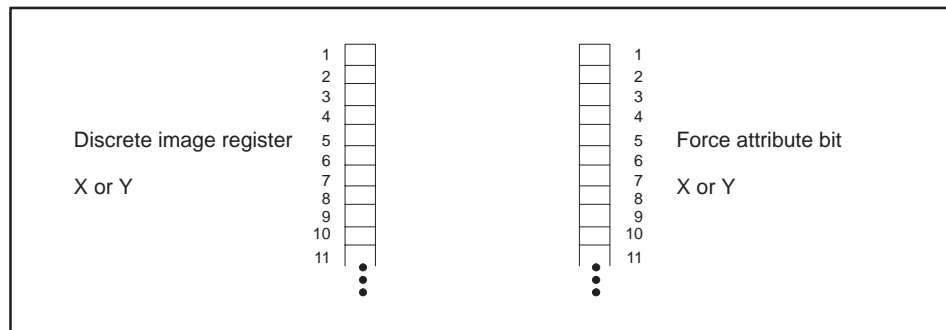


Figure 3-2 Discrete Image Register

## Reading and Updating the I/O (continued)

---

The size of the discrete image register depends upon your controller model (see Table 3-1). Although the discrete and word I/O modules have separate image registers, they are used in the same physical I/O slots. Therefore, the total number of both discrete and word I/O cannot exceed the number listed for your controller model.

Table 3-1 Discrete/Word I/O Permitted

<b>CPU Model</b>	<b>Discrete / Word I/O</b>
PPX:545-1103/-1105	1024
PPX:545-1104/-1106	2048/1024*
All 555 CPUs	8192
All 575 CPUs	8192

\* The 545-1104/-1106 CPUs support 2048 points. Of these, 1024 can be any combination of discrete/word points. The second 1024 points are discrete only.

### **CAUTION**

**Xs and Ys use the same discrete image register.**

**If you assign an input module to an X image register point and an output module to the same Y image register point, your program may not be able to affect the output module's actions.**

**Do not assign the same reference number to both an input (X) and an output (Y).**

**Word Image Register**

The word image register (Figure 3-3) is an area of memory within the controller that is reserved for holding the contents of all 16-bit word inputs and outputs. The size of the word image register depends upon your controller model. The total number of discrete and word I/O cannot exceed the number listed for your controller model.

As a troubleshooting tool, word I/O can be forced. The record of the forced state of word I/O is kept by a force attribute bit, shown in Figure 3-3. There is a one-bit location for each of the word I/O points. If an I/O word is forced, then the value contained within that word does not change until the word either is forced to a different value or is unforced. A power cycle does not alter the value of a forced I/O word as long as the controller battery is good.

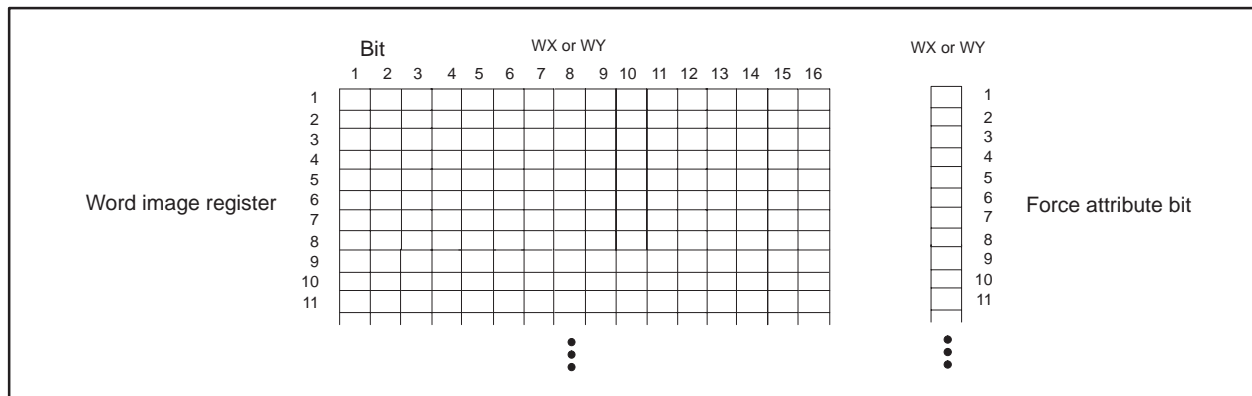


Figure 3-3 Word Image Register

<b>⚠ CAUTION</b>
<p><b>WXs and WYs use the same word image register.</b></p> <p><b>If you assign an input module to an WX image register point and an output module to the same WY image register point, your program may not be able to affect the output module's actions.</b></p> <p><b>Do not assign the same reference number to both an input (WX) and an output (WY).</b></p>

## 3.2 Normal I/O Updates

---

**Discrete Control** To relate the hardwired connections on the equipment that you are controlling to the program inputs and outputs, you need to focus on the function of the image register. For normal I/O updates, the interface between the software RLL program and the physical hardware occurs within the image register. Refer to Figure 3-4 for an example of the discrete operation in which a limit switch controls the state of a pilot light.

**Analog Control** Analog control is similar in operation to discrete control except that data is transmitted as 16-bit words. An analog input signal is converted by the analog input module into a 16-bit digital word. This word of data is written to the word image register.

The controller solves the RLL logic, executing all the necessary tasks relating to the data. If controlling an analog output is the function of the program, then a word of data is written to the word image register.

The controller writes the word from the image register to the analog output module during the normal I/O cycle portion of the scan. The module converts the 16-bit digital word into an analog signal, and sends the analog signal to the appropriate field device.

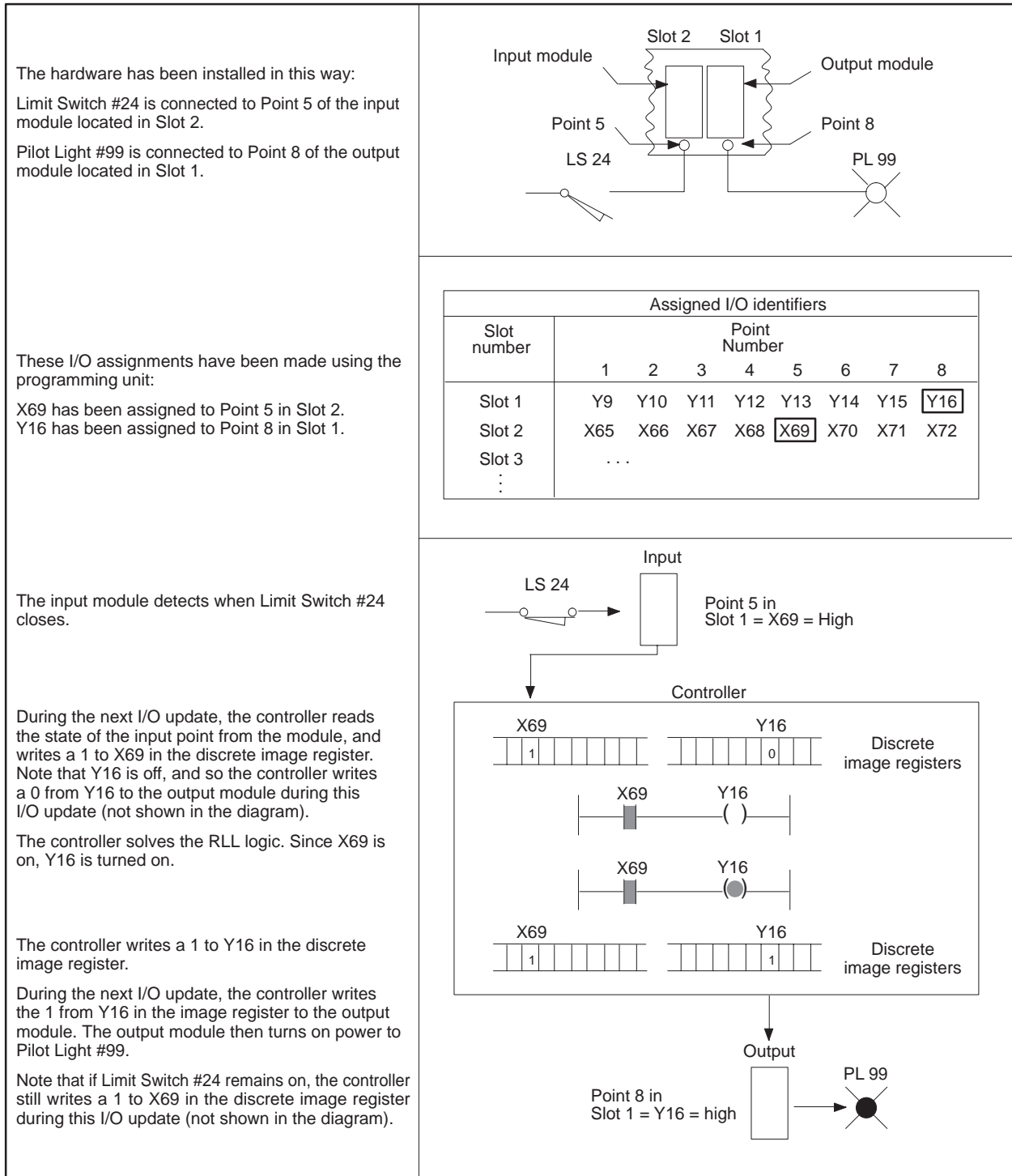


Figure 3-4 Relation of Hardwired Field Devices and the RLL Program

### 3.3 High Speed I/O Updates

#### Immediate I/O

The immediate I/O feature allows your RLL application program to access an I/O point in a local I/O module multiple times per controller scan. This feature enables you to sample fast-changing inputs more often, providing a faster response to the application.

Figure 3-5 illustrates the operation for immediate contacts and immediate coils.

- Use an immediate contact when you want to read an input point directly from the input module as part of the power flow computation. The input discrete image register is not updated as the result of an immediate contact.
- Use an immediate coil when you want to simultaneously write the result of a power flow computation to the output discrete image register as well as to the output module.

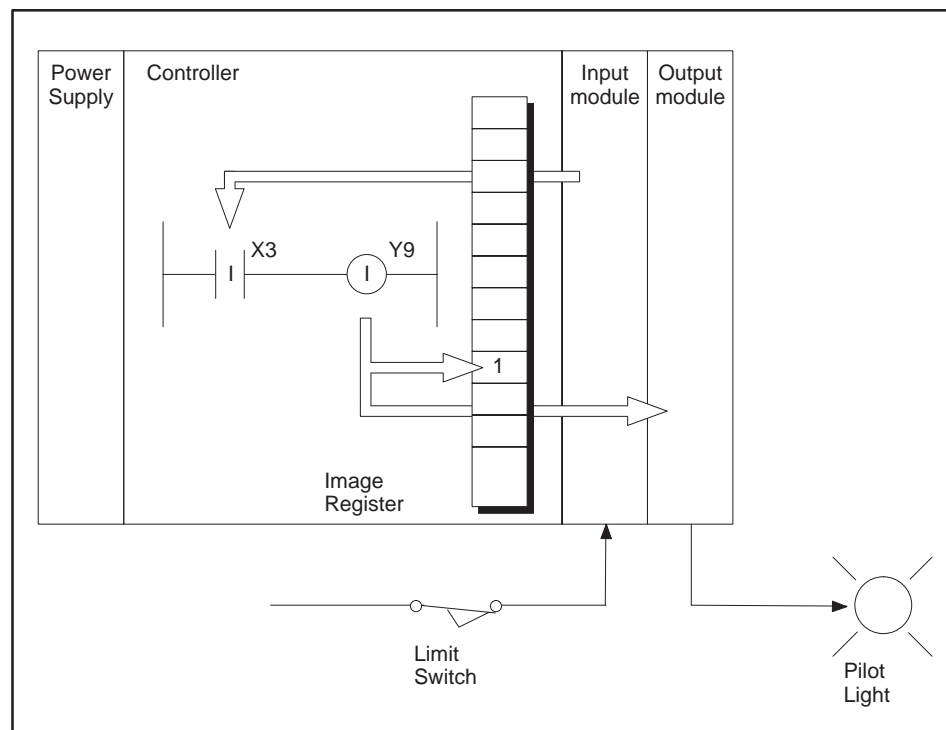


Figure 3-5 Immediate I/O Update



Figure 3-6 illustrates the operation of the IORW (immediate I/O read/write) instruction. For further discussion on immediate I/O read/write, see section 6.24.

- Use an IORW instruction specifying a discrete input image register address (e.g., X1) or a word input image register address (e.g., WX50) to read a block of I/O point values from a module into the referenced image register. The entire block must be contained in a single module.
- Use an IORW instruction specifying a discrete output image register address (e.g., Y17) or a word output image register address (e.g., WY22) to write a block of I/O point values from the referenced image register to a module. The entire block must be contained in a single module.

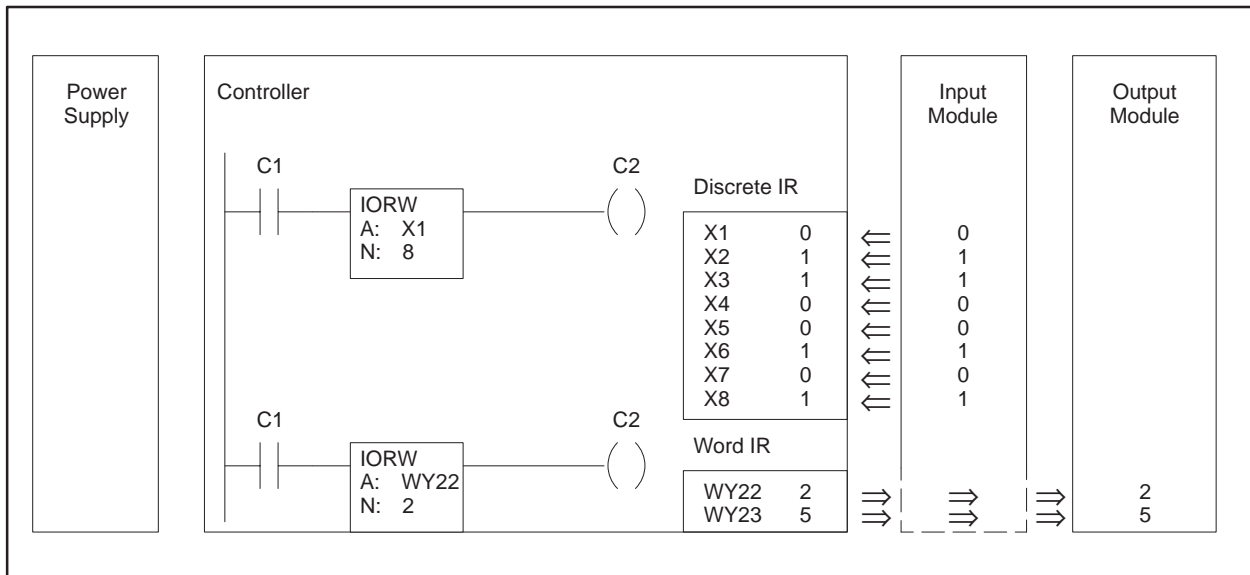


Figure 3-6 IORW Instruction

## High Speed I/O Updates (continued)

Modules that Support Immediate I/O

The 545 and 555 support immediate I/O for all non-SF Series 505 I/O modules. The 575 supports immediate I/O for all VME-compatible I/O modules.

Configuring Immediate I/O

When you configure I/O for the controller, do not assign the same number to both a discrete point and a word point if you intend to access these points as immediate I/O. For example, if you design your program to access X1 immediately, do not configure the word point WX1. See the example I/O Configuration Chart in Figure 3-7.

**NOTE:** Immediate I/O is supported only in modules that are installed in the local base (Base 0).

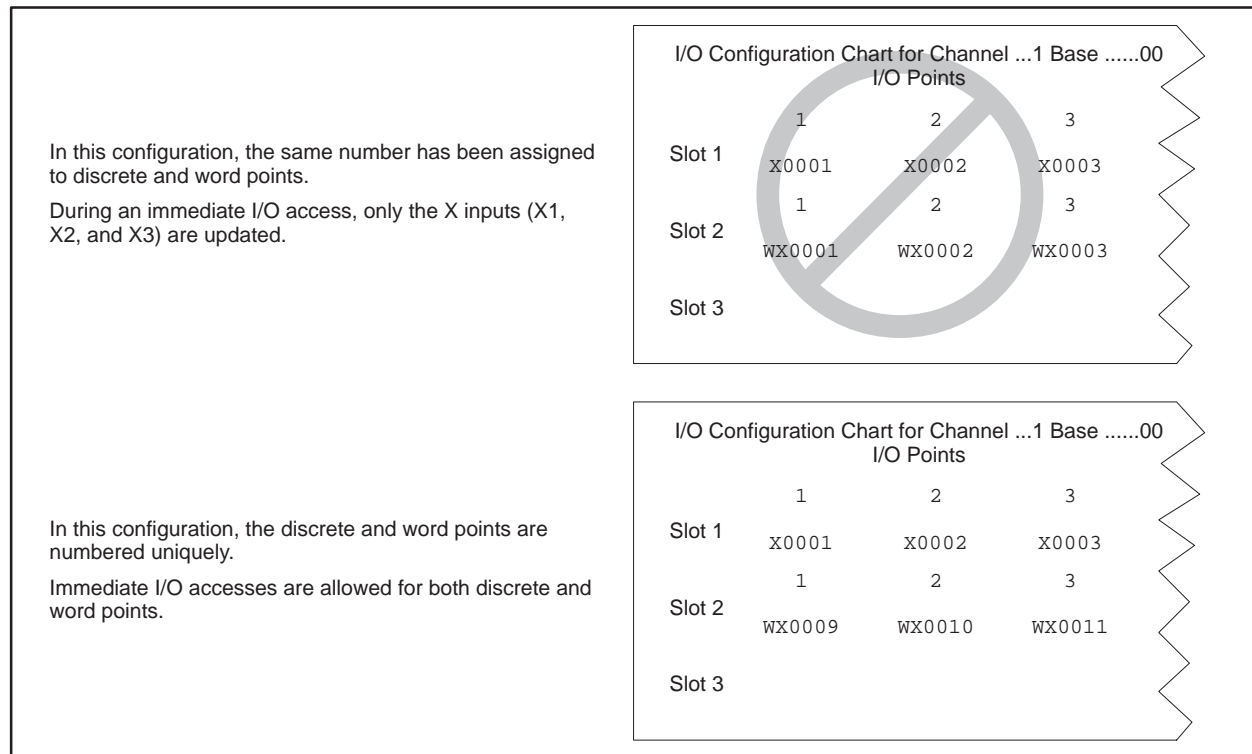


Figure 3-7 Immediate I/O Configuration Chart

## 3.4 Interrupt I/O Operation

---

Overview	<p>The interrupt I/O feature allows your application program to be executed immediately in response to a field input transition generated by your application. Refer to Section 5.5 for more information on interrupt I/O operation.</p>
Configuring the Interrupt Input Module	<p>The interrupt I/O functionality requires a 545 or 555, with at least one Isolated Interrupt Input Module installed in the local base. This module has 16 isolated discrete input points, 8 of which can be configured to generate an interrupt on the occurrence of an off-to-on transition, an on-to-off transition, or a transition in either direction.</p> <p>The Interrupt Input Module has dipswitches that are used to select the signal behavior at a pair of input points that will cause an interrupt to be generated by the module. You must correctly select the interrupt type for the points being used in the interrupt module by using these dipswitches. (The points are not individually configurable.) See the section on “Configuring the Module Operating Mode” in the <i>Isolated Interrupt Discrete Input Module User Manual</i> for a description on how to set the configuration switches.</p> <p>To be used as an interrupt module, this module must be installed in the local base of the system (i.e., the base in which the controller is located or Base 0), and at least one pair of the configurable input points must be specified to be interrupting. Multiple interrupt modules can be used in the local base of the system.</p> <p>When the module powers up with interrupting input points configured, it is logged in by the controller as a 32-point discrete module (24X – 8Y). The points are a mixture of physical field input points and logical (internal) points used for status purposes, as described below.</p> <ul style="list-style-type: none"><li>• Points 1–8: Non-interrupting field inputs (these points cannot be used as interrupting inputs).</li><li>• Points 9–16: Configurable field inputs (can be interrupting or non-interrupting, based upon the settings of the interrupt type switches on the module).</li><li>• Points 17–24: Logical (internal) inputs that indicate which of the interrupting field inputs has generated an interrupt to the controller. A value of ON for a given logical input indicates that the module has generated an interrupt due to the detection of a transition matching the configuration of the corresponding field input. Each of these points corresponds to one of the interrupting field inputs, as shown in Table 3-2.</li></ul>

## Interrupt I/O Operation (continued)

---

- Points 25–32: Logical (internal) outputs that act as individual interrupt enables for each of the interrupting field inputs. Turning on a given output point enables interrupt operation on the corresponding field input, as shown in Table 3-2.

Table 3-2 Logical Points Corresponding to Interrupt Inputs 9 – 16

Physical Input Points (X)	Logical Interrupt Status Inputs (X)	Logical Interrupt Enable Outputs (Y)
9	17	25
10	18	26
11	19	27
12	20	28
13	21	29
14	22	30
15	23	31
16	24	32

The interrupt status points (17–24) are used by the interrupt RLL program to distinguish between interrupt events from each of the configured interrupt input points. See Section 5.5 for more information. The interrupt enable output points (25–32) give you the option of selectively enabling or disabling interrupts under program control. The Interrupt Input Module powers up with all interrupt inputs disabled, so the interrupt enable outputs must be turned on to allow interrupts to be generated by the module.

In order for the controller to accept interrupt requests from an interrupt module, you must correctly configure the module into the I/O map of the controller, using, for example, the I/O Configuration function of TISOFT. The controller ignores interrupt requests from an incorrectly configured module.

---

**NOTE:** For applications requiring quick response to interrupt events, disable the 10-ms filtering option provided by the module (set by dipo switches on the module) for the interrupting points used in that type of application. See the *Isolated Interrupt Discrete Input Module User Manual* for details about the setup and usage of this module.

---

### 3.5 Control Relays

Control relays are single-bit internal memory locations (Figure 3-8) and do not represent actual hardwired devices. A given control relay may be retentive or non-retentive. A retentive control relay maintains its value during a power outage if a good battery is installed and enabled. Non-retentive control relays are initialized to zero (off) following a power outage. The number of available control relays depends upon your controller model. See Table 3-3.

Table 3-3 Control Relays Permitted

<i>Non-retentive</i>	<i>Retentive</i>
C1–C768	C769–C1024
C1025–C1792	C1793–C2048
C2049–C2816	C2817–C3072
C3073–C3840	C3841–C4096
C4097–C4864 <sup>1</sup>	C4865–C5120 <sup>1</sup>
C5121–C5888 <sup>1</sup>	C5889–C6144 <sup>1</sup>
C6145–C6912 <sup>1</sup>	C6913–C7168 <sup>1</sup>
C7169–C7936 <sup>1</sup>	C7937–C8192 <sup>1</sup>
	C8193–C10240 <sup>1</sup>
C10241–C32768 <sup>1, 2</sup>	

<sup>1</sup> Applies to all CPUs except the 545–1103 and 545–1105.  
<sup>2</sup> For the 575 CPUs, the range of non-retentive Cs is C10241–23552.

As a troubleshooting tool, control relays can be forced. The force attribute bit, also shown in Figure 3-8, provides a single-bit memory location for storing the forced status of control relays. If a control relay has been forced, the control relay retains that forced status during a power cycle as long as the battery is good.

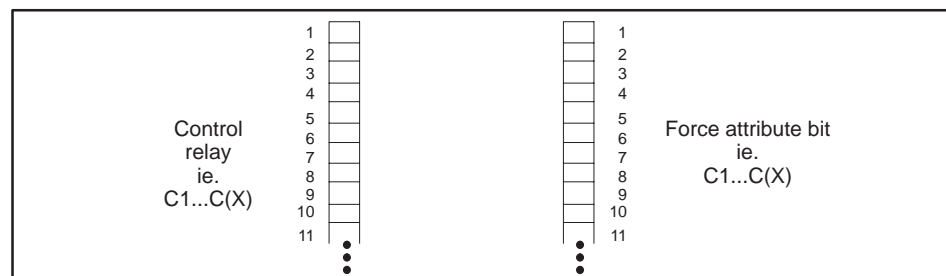


Figure 3-8 Control Relay

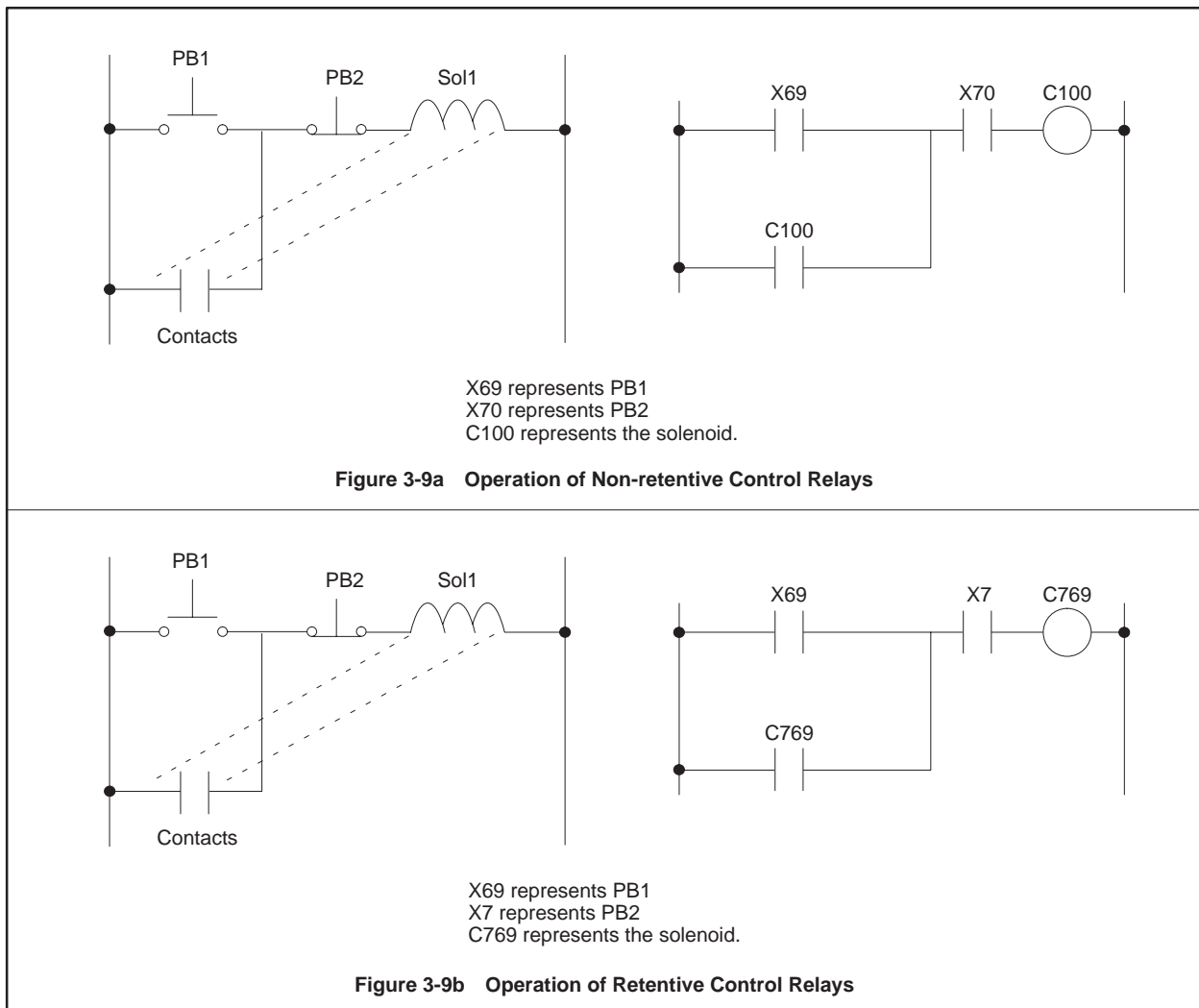
Control relays are retentive or non-retentive. The state of retentive relays does not change during a power loss when the back-up battery is good. Non-retentive relays are turned off if power to the controller is lost.

## Control Relays (continued)

### Using Retentive and Non-retentive Control Relays

The difference in operation between retentive and non-retentive control relays is illustrated in Figure 3-9. The starter circuit shown in Figure 3-9a requires a manual start. The normally open push-button #1 must be pressed. In the event of a power loss, a manual restart is required. The equivalent RLL design, built with non-retentive control relay C100, functions the same way, requiring a manual restart after power loss.

The starter circuit shown in Figure 3-9b also requires a manual start, but in the event of a power loss, restart occurs automatically. Push-button #2 breaks the circuit. The equivalent RLL design, built with retentive control relay C769, also restarts automatically after power loss.



**Figure 3-9 Control Relay Operation**

# Chapter 4

## Controller Memory

---

<b>4.1</b>	<b>Introduction to Controller Memory</b> .....	<b>4-2</b>
	Overview of Controller Memory Types .....	4-2
	RLL Access to the Memory Types .....	4-3
<b>4.2</b>	<b>Controller Memory Types</b> .....	<b>4-4</b>
	Ladder Memory .....	4-4
	Image Register Memory .....	4-4
	Control Relay Memory .....	4-4
	Special Memory .....	4-4
	Compiled Special (CS) Memory .....	4-4
	Temporary Memory .....	4-4
	Variable Memory .....	4-4
	Constant Memory .....	4-5
	Status Word Memory .....	4-5
	Timer/Counter Memory .....	4-5
	Table Move Memory .....	4-6
	One Shot Memory .....	4-7
	Shift Register Memory .....	4-8
	Drum Memory .....	4-9
	PGTS Discrete Parameter Area .....	4-10
	PGTS Word Parameter Area .....	4-10
	User External Subroutine Memory .....	4-11
	Global Memory: 575 Only .....	4-11
	VME Memory: 575 Only .....	4-11

## 4.1 Introduction to Controller Memory

---

### Overview of Controller Memory Types

Controller memory is composed of several functional types (Figure 4-1). You can configure the amount of memory dedicated to each type, depending upon your application. The configurable memory sizes are given in the *SIMATIC 545/555/575 System Manual*.

User Control Program	<p>User Program Memory</p> <ul style="list-style-type: none"><li>• Ladder (L) Memory stores RLL program</li><li>• Special (S) Memory stores loops, analog alarms, SF Programs</li><li>• User (U) Memory stores user-defined subroutines</li><li>• Compiled Special (CS) Memory compiles SF programs and SF subroutines for execution by the floating-point math coprocessor in CPUs equipped with PowerMath™.</li></ul>
User Data	<p>Data Area Memory</p> <ul style="list-style-type: none"><li>• Variable (V) Memory stores variable data</li><li>• Constant (K) Memory stores constant data</li><li>• Global (G) and VME Memory is used for VME data transfers (applies to 575 only)</li></ul>
System Operation	<p>System Memory</p> <ul style="list-style-type: none"><li>• RLL instruction tables: drum, timer/counter, shift register, etc.</li><li>• Image registers and control relays</li><li>• Subroutine parameter areas</li><li>• SF Program temporary memory</li><li>• Status Word memory</li></ul>

Figure 4-1 Controller Memory Types



---

**RLL Access to the  
Memory Types**

The various memory types are described in the pages that follow. Memory types are classified for RLL programming purposes in the following ways:

- **Writeable** — This memory type is read/write. It can be used for both input and output fields of RLL instructions.
- **Readable** — This memory type is read only. It can be used only for the input fields of RLL instructions.
- **No access** — RLL instructions have no access to this memory.

Table A-1 lists the RLL access restrictions for variables that are stored in the various memory types.

## 4.2 Controller Memory Types

---

<b>Ladder Memory</b>	A block of memory within the controller is reserved for the RLL program. This memory type is called Ladder Memory (L-Memory). Each RLL instruction used in the program requires one or more 16-bit words of L-Memory. Refer to Appendix B for a detailed list of the number of words required by each instruction.
<b>Image Register Memory</b>	A block of memory within the controller is reserved for maintaining the status of discrete inputs/outputs. This memory type is called the discrete image register. A word image register holds the values of word inputs/outputs. Refer to Section 3.1 for information about how the image registers operate.
<b>Control Relay Memory</b>	A block of memory within the controller is reserved for control relays. Control relays are single-bit internal memory locations and do not represent actual hardwired devices. Refer to Section 3.5 for information about how the control relays operate.
<b>Special Memory</b>	A block of memory within the controller may be allocated for loops, analog alarms, and Special Function programs. This memory type is called Special Memory (S-Memory). All loop and analog alarm parameters are stored in S-Memory when you program the loop or analog alarm. Likewise, when you create a Special Function program or subroutine, the program is stored in S-Memory.
<b>Compiled Special (CS) Memory</b>	For controllers that support PowerMath, you can configure a block of memory within the controller called Compiled Special Memory (CS-Memory) to execute compiled Special Function programs and subroutines. When an SF program or subroutine is marked as compiled, it is translated to the native instruction set of the CPU's microprocessor. The compiled code is then executed whenever the SF program or subroutine is scheduled for execution.
<b>Temporary Memory</b>	A block of memory within the controller is temporarily reserved during run time whenever a Special Function program is run. One block is allocated for each SF program that is being run. This memory type is 16 words in length and is called Temporary Memory (T-Memory) since it is not saved when the program has completed running. The controller writes data related to the Special Function program to the first seven words. You can read this data and/or write over it if you choose. You can use all 16 words just as you would use Variable Memory, except no data is saved when the program has completed.
<b>Variable Memory</b>	A block of memory within the controller may be allocated for user operations. This memory type is called Variable Memory (V-Memory). For example, you can do a math operation and store the result in V-Memory. You can enter values directly into V-Memory with a programming unit.

---

**Constant Memory**      A block of memory within the controller may be allocated for constants (unchanging data). This memory type is called Constant Memory (K-Memory). You can use a programming unit to load a table of data into K-Memory and read the table during run time whenever you need the data for an operation.

**Status Word Memory**      A block of memory within the controller is allocated for storing status information relating to controller operations. This information is stored in one or more status words: STW1, STW2, etc. These status words can be used in the RLL program to signal and/or correct alarm conditions. See Appendix E for examples. Refer to Appendix G for a list of the status words supported by your controller model.

**Timer/Counter Memory**      A block of memory within the controller is reserved for the operation of the timer/counter group of RLL instructions, including the following.

- Timer (TMR, TMRF)
- Discrete Control Alarm Timer (DCAT)
- Up/Down Counter (UDC)
- Counter (CTR)
- Motor Control Alarm Timer (MCAT)

 **WARNING**

When you assign a number to a timer, counter, up/down counter, or discrete/motor control alarm timer, be sure that you do not use that number for any other timer, counter, up/down counter, or discrete/motor control alarm timer. For example, if you configure a Timer 6 (TMR6), do not configure any other operation, e.g., a counter (CTR) or a discrete control alarm timer (DCAT) with the number 6.

Assigning the same number more than once could cause unpredictable operation by the controller, which could result in death or serious injury to personnel, and/or damage to equipment.

Do not use the same reference number more than once for timer, counter, up/down counter, and discrete/motor control alarm timer instructions.

## Controller Memory Types (continued)

---

This memory type is divided into areas for storing two types of information. This information consists of Timer/Counter Preset (TCP) data and Timer/Counter Current (TCC) data. When you designate a preset value for one of the instructions in this group, this value is stored as a 16-bit word in TCP-Memory. When the instruction is actually operating, the current time or count is stored as a 16-bit word in TCC-Memory.

---

**NOTE:** If you use an operator interface to change the time/counter values, the new values are not changed in the original RLL program. If the RLL presets are ever downloaded, e.g., as the result of a complete restart (TISOFT Aux 12) or an edit of the network containing the Timer/Counter instruction, the changes made with the operator interface are replaced by the values in the RLL program.

---

### Table Move Memory

A block of memory within the controller is reserved for the operation of the table-move instructions, including the following:

- Move Word To Table (MWTT)
- Move Word From Table (MWFT)

### **WARNING**

When you assign a number to a table-move instruction, be sure that you do not use that number for any other table-move instruction. For example, if you configure a Move Word To Table #1 (MWTT1) do not configure a Move Word From Table #1 (MWFT1).

Assigning the same reference number to more than one table-move instruction could cause unpredictable operation by the controller, which could result in death or serious injury to personnel, and/or damage to equipment.

Do not use the same reference number more than once for a table-move instruction.

This memory type consists of one word per table-move instruction configured. This word is used to maintain the current count of moves done since the MWTT or MWFT instruction was last reset.

---

## One Shot Memory

A block of memory within the controller is reserved for the operation of the various instructions of the One-Shot group, including the following:

- One Shot
- Time Set
- Date Set

### **WARNING**

When you assign a number to a One-Shot instruction, be sure that you do not use that number for any other One-Shot instruction type. For example, do not configure more than one OS11.

Assigning the same number for more than one One-Shot instruction type can cause unpredictable operation by the controller, which could result in death or serious injury to personnel, and/or damage to equipment.

Do not use the same number more than once for the same instruction type (e.g., use it only once in One Shot, in Timer Set, etc.).

This memory type consists of one byte per configured One-Shot instruction. This byte is used to save the previous state of the instruction input.

Because the instructions in the One-Shot group use different bits of one byte, these instructions can be assigned identical reference numbers. That is, if you configure a One Shot #11 (OS11) you can configure a Date Set #11.

## Controller Memory Types (continued)

---

### Shift Register Memory

A block of memory within the controller is reserved for the operation of the shift registers, which include the following:

- Bit Shift Register (SHRB)
- Word Shift Register (SHRW)

### **WARNING**

When you assign a number to a shift register, be sure that you do not use that number for any other shift register type. For example, do not configure SHRB11 and SHRW11.

Assigning the same number for more than one shift register could cause unpredictable operation by the controller, which could result in death or serious injury to personnel, and/or damage to equipment.

**Do not assign the same reference number to more than one shift-register instruction.**

This memory type consists of one byte per shift register. This byte is used to save the previous state of the instruction input.

---

## Drum Memory

A block of memory within the controller is reserved for the operation of the various drum types, including the following:

- Drum (DRUM)
- Maskable Event Drum Discrete (MDRMD)
- Event Drum (EDRUM)
- Maskable Event Drum Word (MDRMW)

### WARNING

When you assign a number to a drum-type instruction, be sure that you do not use that number for any other drum-type instruction. For example, if you configure a Maskable Event Drum Word #1 (MDRMW1), do not configure an Event Drum #1 (EDRUM1).

Assigning the same reference number to more than one drum-type instruction could cause unpredictable operation by the controller, which could result in death or serious injury to personnel, and/or damage to equipment.

Do not assign the same reference number to more than one drum-type instruction.

Drum memory is divided into areas for storing the following types of information:

- Drum Step Preset (DSP)
- Drum Count Preset (DCP)
- Drum Step Current (DSC)
- Drum Count Current (DCC)

When you specify step and counts-per-step (count preset) values for a drum type, the step preset is stored as a 16-bit word in DSP-Memory, and the counts-per-step values are stored as 16 consecutive 16-bit words in DCP-Memory (except for the DRUM). For the DRUM instruction, counts-per-step values are stored in L-Memory; DCP is not used.

When the instruction is actually operating, the current step is stored as a 16-bit word in DSC-Memory. The current count for this step is stored as a 16-bit word in DCC-Memory.

---

**NOTE:** If you use an operator interface to change the drum preset values (DSP or DCP), the new values are not changed in the original RLL program. If the RLL presets are ever downloaded, e.g., as the result of a complete restart (TISOFT Aux 12) or an edit of the network containing the drum instruction, the changes made with the operator interface are replaced by the values in the RLL program.

---

## Controller Memory Types (continued)

### PGTS Discrete Parameter Area

The Parameterized Go To Subroutine (PGTS) discrete parameter area (Figure 4-2) is an area of memory within the controller that is reserved for holding the status of discrete bits referenced as parameters in a PGTS RLL instruction. Because up to 32 PGTS subroutines can be programmed, the controller has 32 discrete parameter areas, each capable of storing the status for 20 discrete parameters. When you use a parameter in the subroutine, refer to discrete points as  $B_n$  where  $n$  = the parameter number.

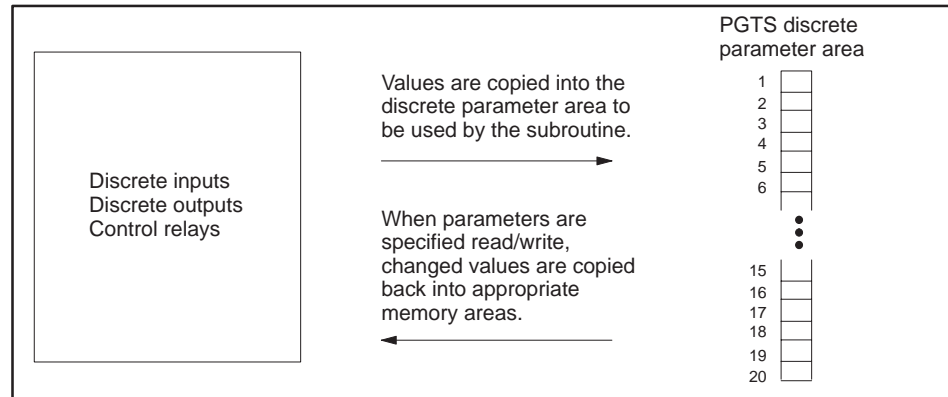


Figure 4-2 PGTS Discrete Parameter Area

### PGTS Word Parameter Area

The PGTS word parameter area (Figure 4-3) is an area of memory within the controller that is reserved for holding the contents of 16-bit words referenced as parameters in a PGTS RLL instruction. Because up to 32 PGTS subroutines can be programmed, the controller has 32 word parameter areas, each capable of storing the status for 20 word parameters. When you use a parameter in the subroutine, refer to words as  $W_n$ , where  $n$  = the parameter number.

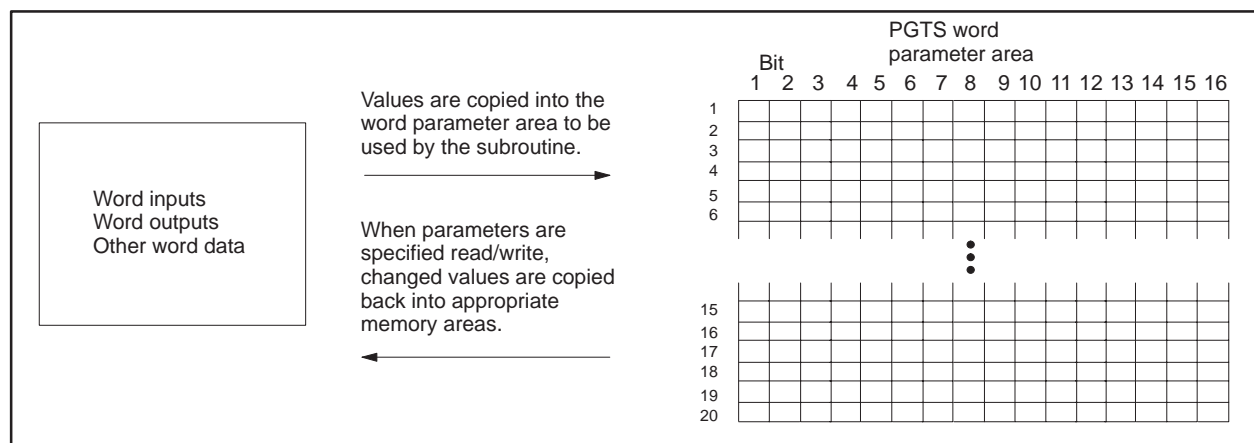


Figure 4-3 PGTS Word Parameter Area



---

User External Subroutine Memory	A block of memory within the controller may be allocated for storing externally developed programs written in C, Pascal, assembly language, etc. This memory type is called User Memory (U-Memory). The size of U-Memory is user configurable.
Global Memory: 575 Only	The 575 CPU allocates a 32K-word block of memory to allow you to transfer data over the VME backplane. This memory type is called Global Memory (G-Memory). Refer to Appendix I for more information about G-Memory.
VME Memory: 575 Only	The 575 controller also allows access to physical VME addresses using the VMM-Memory or VMS-Memory. <ul style="list-style-type: none"> <li>• VMM corresponds to VME address modifier 39 (standard non-privileged data access).</li> <li>• VMS corresponds to VME address modifier 29 (short non-privileged access).</li> </ul>

 **CAUTION**

The 575 controller allows you to use a VME address (VMM or VMS) as a parameter to most word-oriented RLL instructions, e.g., ADD, SUB, or MOVW, etc.

When a VME address is used and is not recognized by any installed board, a VMEbus error occurs. If the instruction that used the address was other than MOVE or XSUB (with the U-Memory header's E bit set to 1—see Appendix H), the controller enters the Fatal Error mode, freezes analog outputs and clears discrete outputs.

Use the XSUB or MOVE instruction to access the VME address.

# Chapter 5

## Programming Concepts

---

<b>5.1</b>	<b>RLL Components</b> .....	<b>5-2</b>
	RLL Concept .....	5-2
	RLL Contact .....	5-3
	RLL Coil .....	5-8
	RLL Box Instruction .....	5-12
	RLL Rung Structure .....	5-12
	RLL Scan Principles .....	5-13
<b>5.2</b>	<b>Program Compile Sequence</b> .....	<b>5-14</b>
<b>5.3</b>	<b>Using Subroutines</b> .....	<b>5-16</b>
	RLL Subroutine Programs .....	5-16
	SF Programs .....	5-16
	External Subroutines .....	5-17
<b>5.4</b>	<b>Cyclic RLL</b> .....	<b>5-18</b>
	Overview .....	5-18
	Cyclic RLL Execution .....	5-20
<b>5.5</b>	<b>Interrupt RLL (545/555 only)</b> .....	<b>5-22</b>
	The Interrupt RLL Task .....	5-22
	Operation .....	5-25
	Performance Characteristics .....	5-26
	Troubleshooting .....	5-27
<b>5.6</b>	<b>Using Real-Time Clock Data</b> .....	<b>5-28</b>
	BCD Time of Day .....	5-28
	Binary Time of Day .....	5-30
	Time of Day Status .....	5-31
<b>5.7</b>	<b>Entering Relay Ladder Logic</b> .....	<b>5-32</b>
	SoftShop 505 for Windows .....	5-32
	TISOFT .....	5-32
	Using APT .....	5-32
	COM PROFIBUS .....	5-32
<b>5.8</b>	<b>Doing Run-Time Program Edits</b> .....	<b>5-33</b>
	Editing in Run Mode .....	5-33
	Avoid These Actions During Run-Time Edits .....	5-34
	Additional Considerations When Doing Run-Time Edits .....	5-37
<b>5.9</b>	<b>Password Protection</b> .....	<b>5-39</b>
	Protected Program Elements .....	5-39
	Disabled and Enabled Passwords .....	5-39
	Password Protection Levels .....	5-40
	Determining the Current State of Password .....	5-40
	Password Effect on EEPROM .....	5-40

## 5.1 RLL Components

---

Depending upon your controller model, you can choose from several programming languages to write your application program. The basic language that is common to all the Series 505 controllers is Relay Ladder Logic (RLL). In addition, the Special Function (SF) programming language provides a high-level statement-driven language that can be used for floating-point math calculations and can call externally developed subroutines that are written in other high-level programming languages, such as C, or Pascal.

For a description of these other programming methods, refer to Section 5.3 for the external subroutines, and Chapter 7 for SF programs.

### RLL Concept

RLL is similar in form and interpretation to the relay diagram. Two vertical lines represent power and return rails. Connections between the rails (the ladder rungs) contain circuit components that represent switches, control relays, solenoids, etc.

The primary function of the RLL program is to control the state of an output, based on one or more input conditions. An example is shown in Figure 5-1. This is done at the level of a ladder rung.

In Figure 5-1, the controller tests the input condition, which is represented by the contacts X20 and X21. When either of the contacts is evaluated as true, it is defined as having power flow and the circuit is complete to the next component on the rung, coil Y33. When coil Y33 receives power flow, the output condition is true, and the circuit is complete to the return rail.

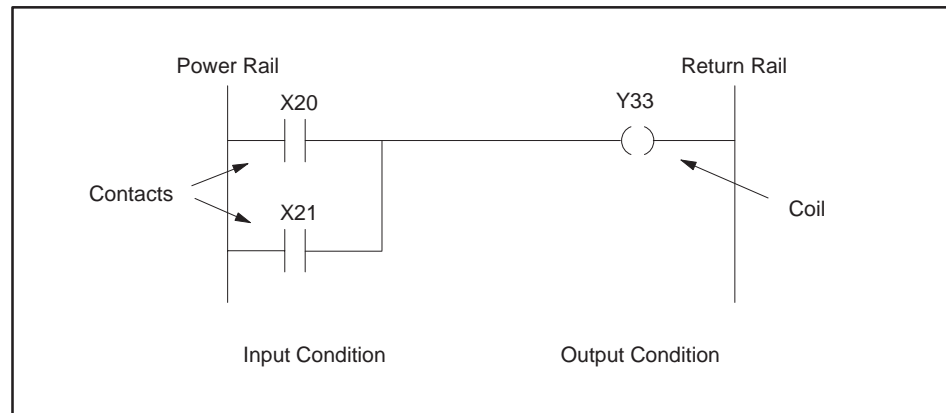


Figure 5-1 Single Rung of a Relay Ladder Logic Program

---

## RLL Contact

A contact can be used anywhere in the program to represent a condition that needs to be tested. It can represent an actual field input or an internal memory location. When representing a field input, the contact is referenced by an address in one of the image registers. When representing an internal memory location, the contact is referenced by an address in one of the other RLL-readable memory locations, such as the control relays.

In Figure 5-2, the address for the contact is X1, a point in the discrete image register. When X1 contains a 1, the contact evaluates as true or on; when X1 contains a 0, the contact evaluates as false or off.

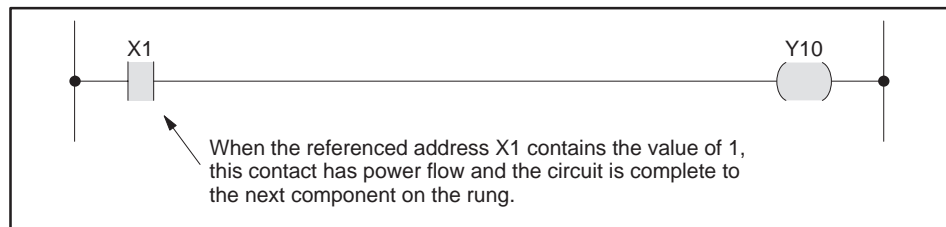


Figure 5-2 Power Flow and the Contact

The normal contact is symbolized by  $\text{—} | \text{—}$  in the RLL program. Use the normal contact when your application requires the referenced address to equal 1 in order to turn the output on.

- If the referenced address equals 1, the normal contact closes and passes power flow.
- If the referenced address equals 0, the normal contact remains open and does not pass power flow.
- Use the normal contact to represent field devices that operate like a limit switch. When the limit switch closes, the normal contact closes and passes power flow.

The operation of the normal contact is compared to that of an electro-mechanical relay in Figure 5-3.

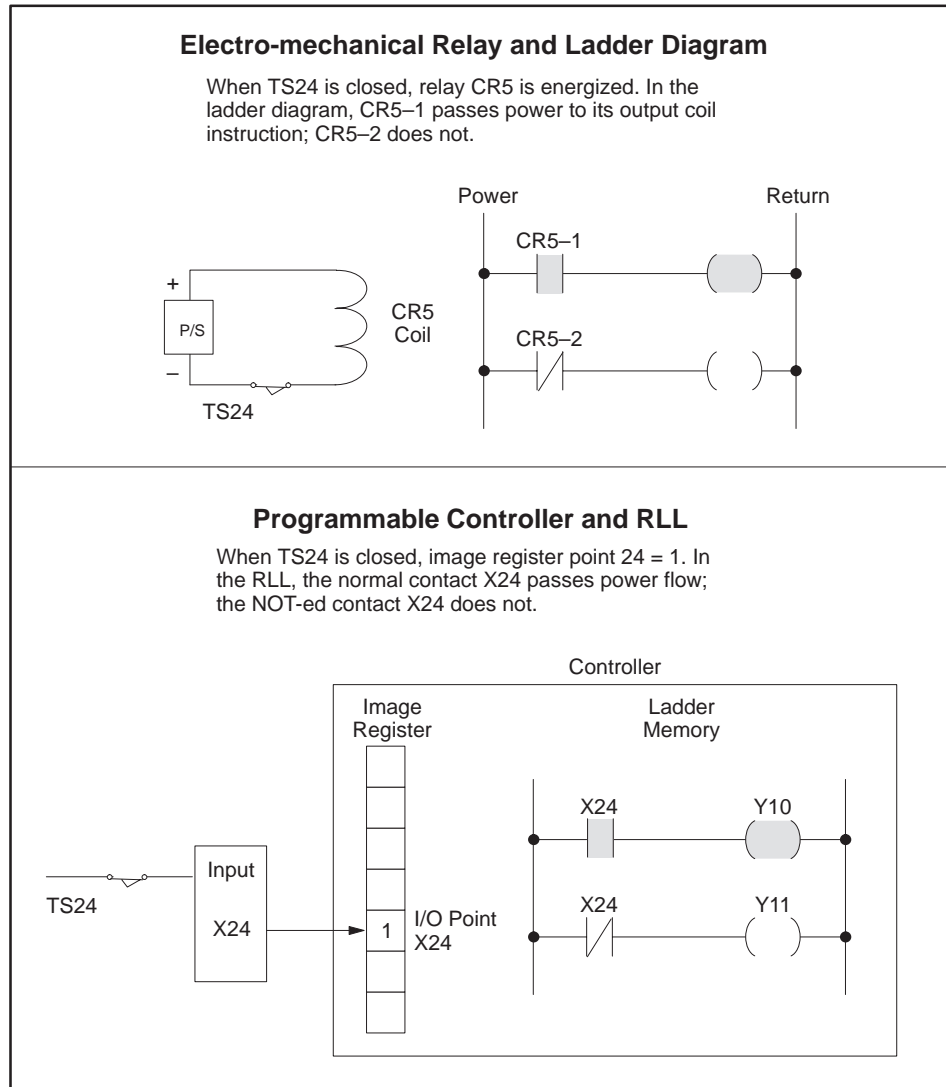



Figure 5-3 Operation of Normal Contact and Electro-mechanical Relay

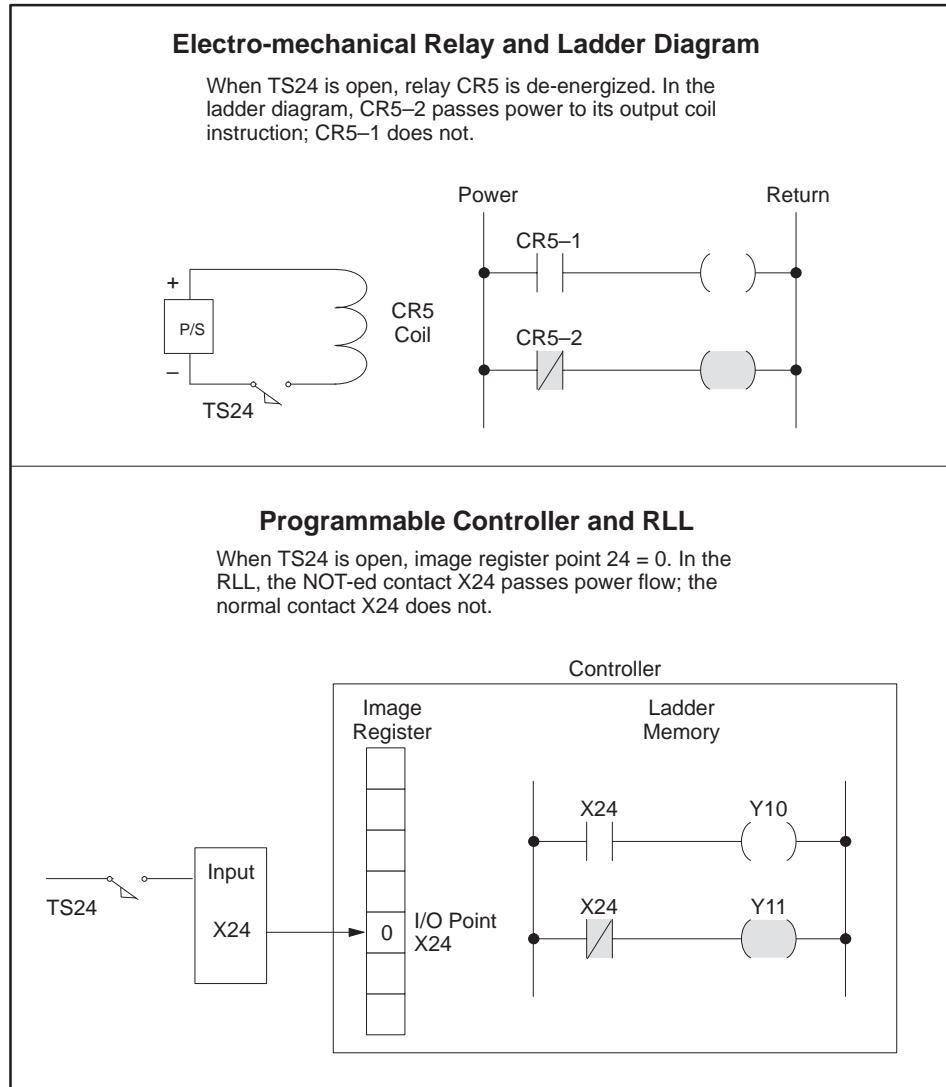
---

The NOT-ed contact is symbolized by  in the RLL program. Use the NOT-ed contact when your application requires the referenced address to equal 0 in order to turn the output on.

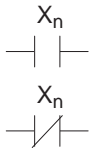
- If the referenced address equals 0, the NOT-ed contact remains closed and passes power flow.
- If the referenced address equals 1, the NOT-ed contact opens and interrupts power flow.

The operation of the NOT-ed contact is compared to that of an electro-mechanical relay in Figure 5-4.

Several different types of contacts are available to enable you to create the program control that you need for your application. These types of contacts are described on Pages 5-7 and 5-8.

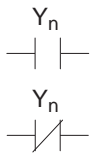


**Figure 5-4 Operation of a NOT-ed Contact and Electro-mechanical Relay**

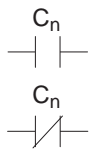


An X contact corresponds to a point in the discrete image register. The X contact represents an input from a field device, for example, a limit switch.

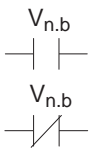
<b>⚠ CAUTION</b>
<p><b>Xs and Ys use the same discrete image register.</b></p> <p><b>If you assign an input module to an X image register point and an output module to the same Y image register point, your program may not be able to affect the output module's actions.</b></p> <p><b>Do not assign the same reference number to both an input (X) and an output (Y).</b></p>



A Y contact corresponds to a point in the discrete image register. The status of a Y contact is determined by the status of the Y output coil that has the same address as the Y contact.



A C contact represents a control relay. Control relays are internal memory locations and do not represent actual hard-wired field devices. The control relay is used to provide control for other RLL instructions.

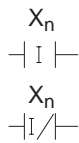


A bit-of-word contact represents an individual bit in any readable word, such as a V- or WX-Memory location. Power flow in a bit-of-word contact is determined by the state of the bit b (1–16) that it represents.

For example, the bit-of-word contact  $\overset{V100.13}{\text{---|---}}$  is closed when bit 13 in V100 equals 1.



## RLL Components (continued)

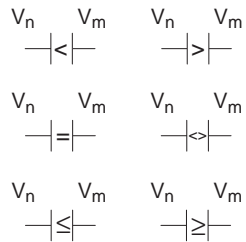


An immediate X contact corresponds to a discrete point in an I/O module and is updated from the I/O module immediately. The immediate X contact can be updated any time during the controller scan, and is not limited to the normal I/O update portion of the timeline.

---

**NOTE:** Only the power flow for an immediate X contact is updated. The value in the image register is not updated.

---



The power flow through a relational contact depends upon the relational condition that exists between the values contained in two readable words, such as V- or WX-Memory locations. When the relational condition is true, the contact is closed. When the relational condition is not true, then the contact is open.

For example, the relational contact  $\begin{array}{c} V1 & V25 \\ \text{---} | < | \text{---} \end{array}$  is closed when the content of V1 is less than the content of V25.

The word on the right of the contact symbol can be a signed integer (INT, -32768 to 32767) or an unsigned integer (UINT, 0 to 65535).

The relational contact  $\begin{array}{c} V112 & 941 \\ \text{---} | = | \text{---} \end{array}$  is closed when the content of V112 is equal to 941.

### RLL Coil

A coil can be used anywhere in the program to represent an output that needs to be controlled. It can represent an actual field device or an internal memory location. When representing a field device, the coil is referenced by an address in one of the image registers. When representing an internal memory location the coil is referenced by an address in one of the other RLL-writeable memory locations, such as control relay memory.

In Figure 5-5, the address for the coil is Y10, a point in the discrete image register. When the coil is true or on, the controller writes a 1 to Y10; when the coil is not true or off, the controller writes a 0 to Y10.

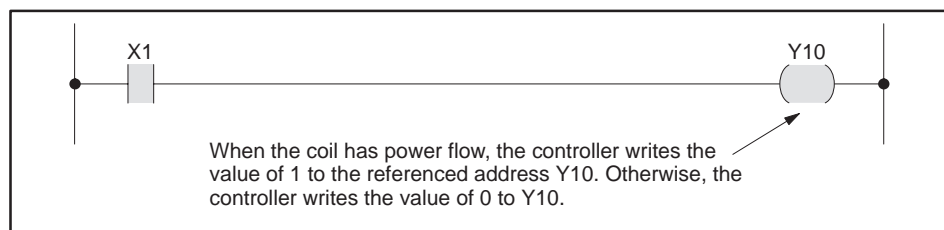


Figure 5-5 Power Flow and the Coil

---

The Normal Coil is symbolized by  $\text{---}(\ )\text{---}$  in the RLL program. Use the normal coil when your application requires the referenced address to equal 1 when the coil has power flow.

- When the rung logic passes power flow to the normal coil, the coil turns on and the referenced address equals 1.
- When the rung logic does not pass power flow to the normal coil, the coil remains off and the referenced address equals 0.
- When the normal coil is on, a normal contact that references the same address also turns on. A NOT-ed contact that references the same address turns off.
- Use the normal coil to represent field devices that operate like a solenoid. When the normal coil has power flow, the solenoid is energized.

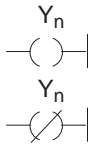
The NOT-ed coil is symbolized by  $\text{---}(\text{X})\text{---}$  in the RLL program. Use the NOT-ed coil when your application requires the referenced address to equal 0 when the coil has power flow.

- When the rung logic does not pass power flow to the NOT-ed coil, the coil remains energized and the referenced address equals 1.
- When the rung logic passes power flow to the NOT-ed coil, the coil is de-energized and the referenced address equals 0.
- When the NOT-ed coil has power flow, a normal contact that references the same address turns off. A negative contact that references the same address turns on.
- The NOT-ed coil does not have any actual field device counterpart. Use the NOT-ed coil in a situation when you want the output to turn off when the NOT-ed coil has power flow.

Several different types of coils are available to enable you to create the program control that you need for your application. These types of coils are described on Pages 5-10 and 5-11.

## RLL Components (continued)

---



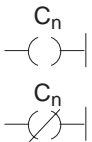
A Y coil corresponds to a point in the discrete image register. The Y coil can represent an output to a field device or an internal control relay.

### CAUTION

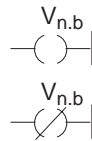
**Xs and Ys use the same discrete image register.**

**If you assign an input module to an X image register point and an output module to the same Y image register point, your program may not be able to affect the output module's actions.**

**Do not assign the same reference number to both an input (X) and an output (Y).**

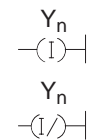


A C coil represents a control relay. Control relays are internal memory locations and do not represent actual hard-wired field devices. The control relay is used to provide control for other RLL instructions.



A bit-of-word coil represents an individual bit in any writeable word, such as a V- or WY-Memory location. Power flow in a bit-of-word coil determines the state of the bit b (1–16) that it represents.

For example, when this bit-of-word coil  is on, bit 2 in V18 is set to 1. When the coil is off, bit 2 in V18 is cleared to 0.



An immediate Y coil operates as a normal Y coil with the additional function that an immediate I/O module update is done when the instruction (coil) is executed. The immediate Y coil is updated any time during the controller scan, and is not limited to the normal I/O update portion of the timeline.

---

**NOTE:** Both the image register and the I/O module are updated when the immediate Y coil is executed.

---

---

$Y_n$   
 $\text{-(SET)}$   
 $Y_n$   
 $\text{-(RST)}$

When it has power flow, a SET Y coil sets a specified bit to one. Otherwise, the bit remains unchanged. When it has power flow, a RST (Reset) Y coil clears a specified bit to zero. Otherwise, the bit remains unchanged.

$C_n$   
 $\text{-(SET)}$   
 $C_n$   
 $\text{-(RST)}$

When it has power flow, a SET C coil sets a specified bit to one. Otherwise, the bit remains unchanged. When it has power flow, a RST (Reset) C coil clears a specified bit to zero. Otherwise, the bit remains unchanged.

$Y_n$   
 $\text{-(SETI)}$   
 $Y_n$   
 $\text{-(RSTI)}$

The SET immediate Y coil operates the same as the set Y coil, except that the specified bit is updated immediately, like the immediate Y coil. The RST (Reset) immediate Y coil operates the same as the reset Y coil, except that the specified bit is updated immediately, like the immediate Y coil.

$V_{n,b}$   
 $\text{-(SET)}$   
 $V_{n,b}$   
 $\text{-(RST)}$

The SET bit-of-word coil operates the same as the set coil, except that the specified bit is contained in a writeable word, such as a V- or WY-Memory location. The RST (Reset) bit-of-word coil operates the same as the reset coil, except that the specified bit is contained in a writeable word.

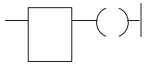
---

**NOTE:** If the referenced bit is only used by set/reset coils, then the bit acts as a latch.

---

## RLL Components (continued)

### RLL Box Instruction



The RLL box instructions are pre-programmed functions that extend the capabilities of your program beyond the RLL relay-type contact and coil instructions. The box instructions are described in detail in Chapter 6.

The counter, shown in Figure 5-6, is an example of a box instruction.

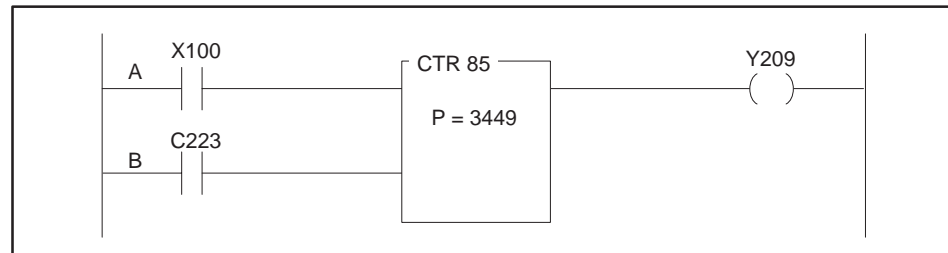
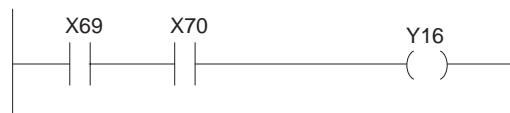


Figure 5-6 Example of a Box Instruction

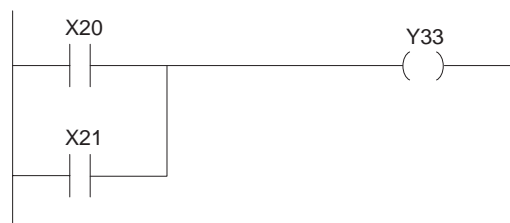
The counter is enabled by the lower input line, B in the figure. Then off/on transitions on the upper input line A are counted as pulses. When the pulse count reaches the preset value of 3449, the output coil is turned on.

### RLL Rung Structure

You can design a rung in combinations of series and parallel structures to provide the required logic for controlling the output. The rung shown below represents a series circuit. When both input conditions are true, the output is true. In terms of programming logic, the two input conditions are ANDed:  $Y16 = (X69 \bullet X70)$ .



This rung represents a parallel circuit. When either input condition is true, the output is true. In terms of programming logic, the two input conditions are ORed:  $Y33 = (X20 + X21)$ .



## RLL Scan Principles

When processing an RLL program that contains no cyclic or interrupt RLL tasks, the sequence of controller operation is summarized in these three stages.

- The controller reads all inputs, and
- The controller solves the RLL, and
- The controller writes all outputs.

The controller solves all the logic in an RLL rung before proceeding to the next rung, as shown in Figure 5-7. Refer to Section 3.3 for a discussion of cyclic RLL and Section 3.4 for a discussion of interrupt RLL operation.

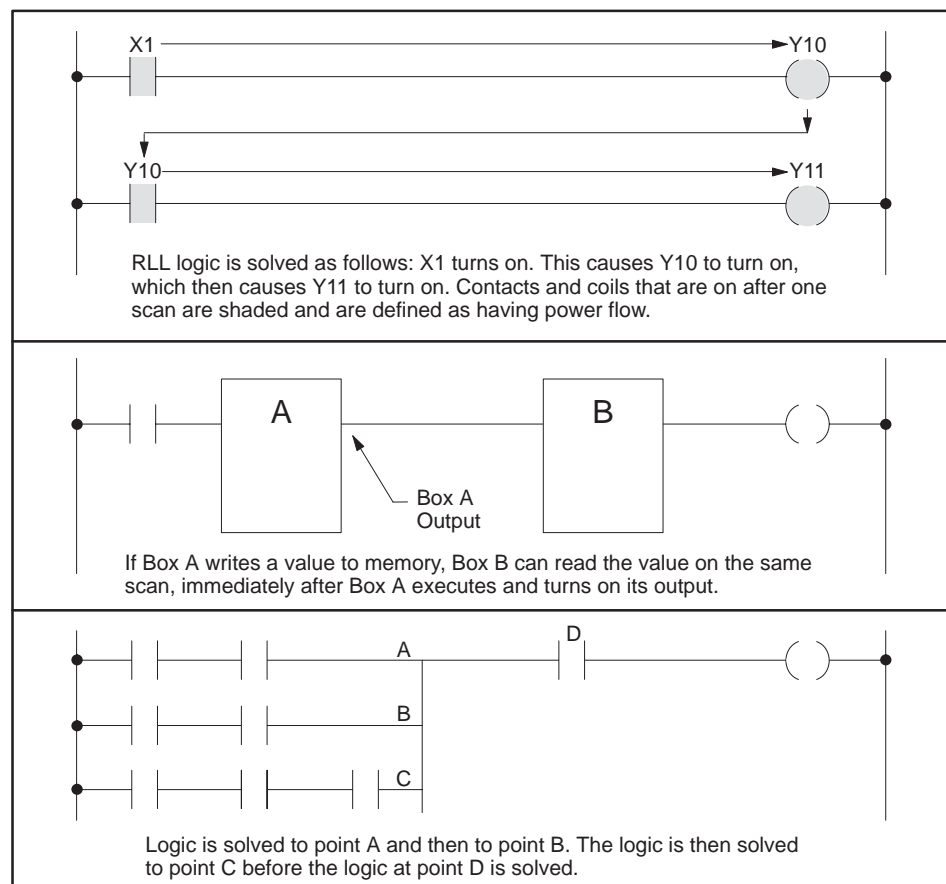


Figure 5-7 How Relay Ladder Logic Is Solved

## 5.2 Program Compile Sequence

If an RLL program has been modified, it is compiled when the controller mode changes from PROGRAM to RUN or from EDIT to RUN. The compile sequence for an RLL program is illustrated in Figure 5-8. Note the effect of the END and SBR RLL instructions on the compile process.

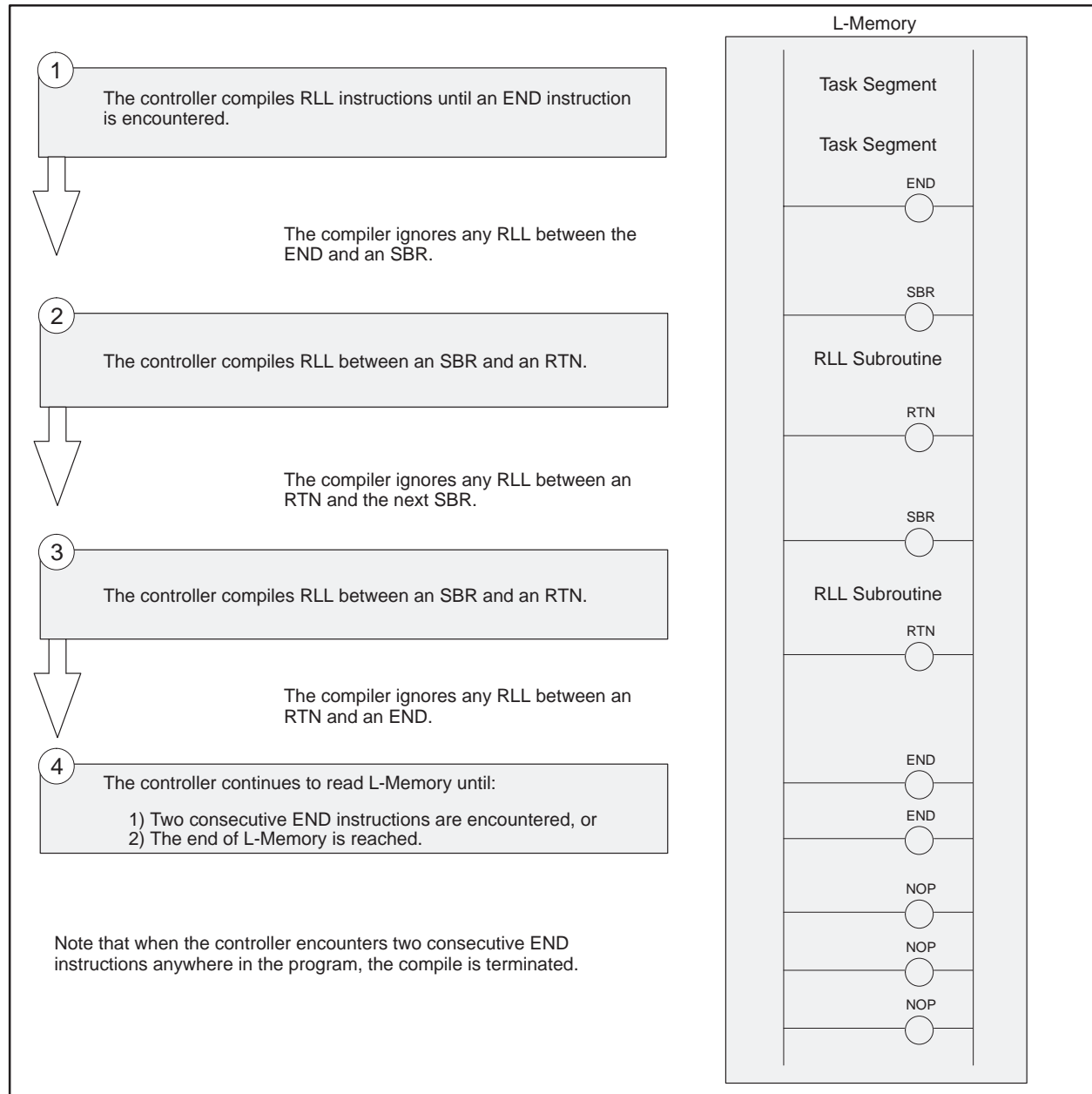


Figure 5-8 RLL Program Compile Process

---

Remember these rules as you design the RLL program.

- The TASK instruction, not an END instruction, separates task segments.
- All TASKs must be located before the first END.
- The zone of control for a SKP is limited to the task segment or subroutine in which the SKP is used. That is, the matching LBL must be defined after the SKP and in the same task segment or subroutine as the SKP.
- An END instruction separates RLL subroutines, if any, from the rest of the program.
- Subroutines must be terminated with an unconditional RTN instruction.
- Two consecutive END instructions terminate the compile process. Otherwise, the controller scans all of L-Memory. If the RLL program is significantly smaller than configured L-Memory, terminate the program with two END instructions to reduce the scan bump caused by a change to RUN mode after a run-time edit.

---

**NOTE:** The online FIND function does not search past two consecutive END instructions. You must position your cursor after the two ENDS when you search for an item occurring after two END instructions.

---



## 5.3 Using Subroutines

---

The 545, 555, and 575 controllers provide several levels of subroutine support for your application program. Program subroutines can be designed as an RLL structure stored in L-Memory, a Special Function (SF) program located in S-Memory, or an externally developed program (written in C, Pascal, or certain other high-level languages) stored in U-Memory.

### RLL Subroutine Programs

You use the SBR, and RTN ladder logic instructions to create an RLL subroutine that can be called from the main RLL program. The SBR instruction marks the start of the subroutine; the RTN instruction marks the end of the subroutine. The GTS instruction transfers program control to the subroutine and RTN returns control to the instruction that follows the calling GTS instruction after the subroutine has executed.

The PGTS ladder logic instruction operates similarly to the GTS instruction. You use PGTS to call a section of the RLL program that is preceded by an SBR and execute it. Unlike GTS, the PGTS allows you to pass parameters to a subroutine.

Refer to Chapter 6 for more information about using the RLL subroutine instructions.

### SF Programs

A Special Function program consists of a set of high-level, statement-driven programming instructions that can be called from loops, analog alarms, or from the RLL program, much like a GOSUB subroutine in a BASIC program or a procedure in a C language program. Typically, the types of operations that you execute within an SF program either cannot be done with the RLL instruction set, or they involve complex RLL programming. Such operations include floating point math, If/Then conditional statements, table transfers, data consolidation, etc.

Refer to Chapter 7 for more information about designing and writing SF programs.

---

External  
Subroutines

Use the XSUB instruction to pass appropriate parameters to an externally developed subroutine and then call the subroutine for execution. The external subroutine can be developed offline in a non-RLL programming language, such as C or Pascal.

Refer to Appendix H for more information about designing and writing external subroutines.

 **WARNING**

When you call an external subroutine, the built-in protection features of the controller are by-passed.

Control devices can fail in an unsafe condition that could result in death or serious injury to personnel, and/or damage to equipment.

You must take care in testing the external subroutine before introducing it to a control environment. Failure to do so may cause undetected corruption of controller memory and unpredictable operation by the controller.

### Overview

The cyclic RLL function allows you to partition the RLL program into a cyclic RLL task and a main RLL task. When used with the immediate I/O feature, the cyclic RLL task can provide very high rates of sampling for critical inputs.

The TASK instruction, described in Chapter 6, is used to partition an RLL program into a main RLL task and a cyclic RLL task.

An RLL application program that contains a cyclic RLL task must be designed as follows.

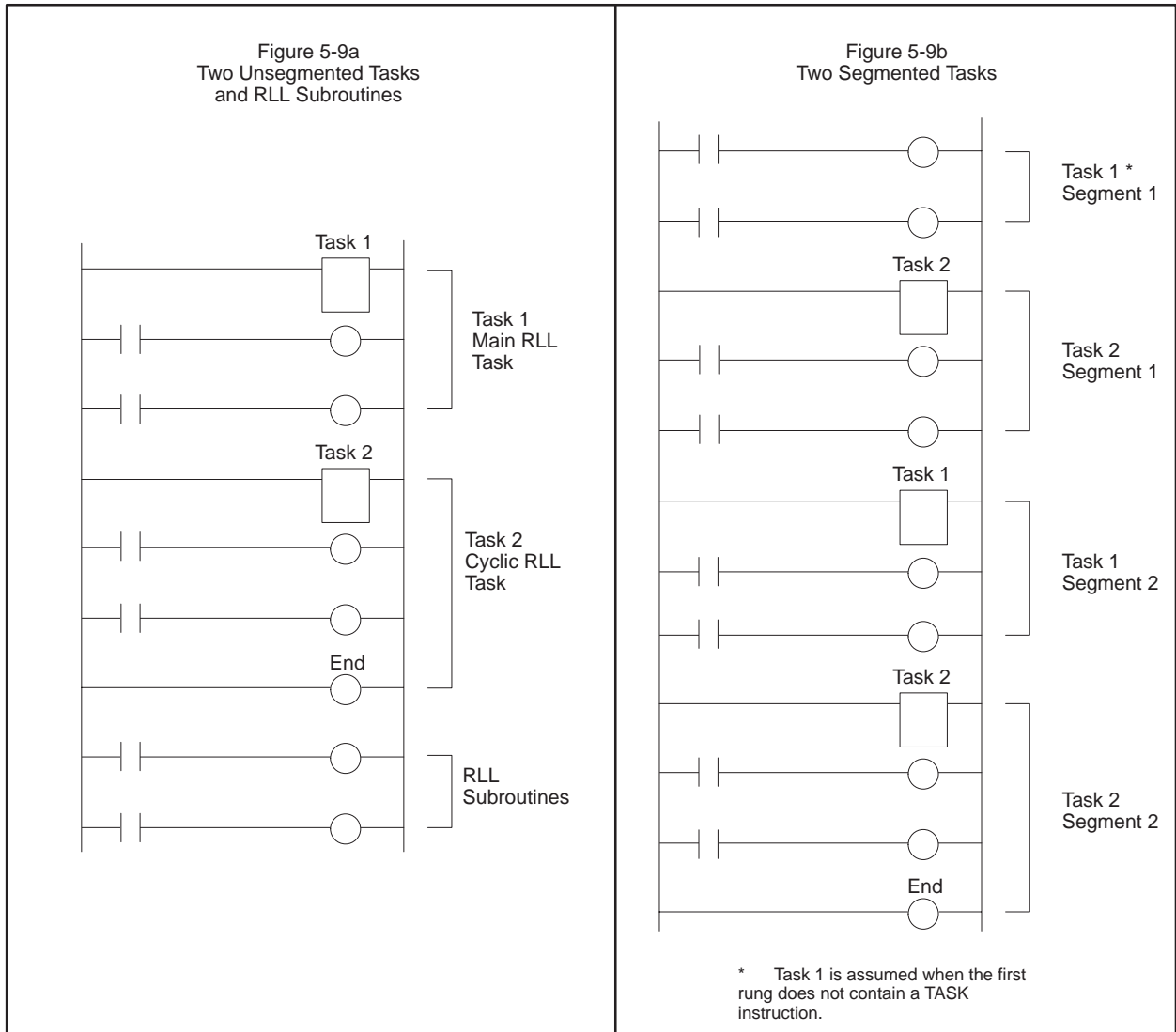
- The application program can consist of two or three RLL tasks: the main RLL task, the cyclic RLL task, and an optional interrupt RLL task. Each RLL task is preceded by the TASK(n) instruction, where n = 1 designates the main task, n = 2 designates the cyclic task, and n = 8 designates the interrupt task. Refer to Figure 5-9a.
- The A field of the TASK2 instruction specifies the cycle time of the cyclic task in milliseconds. The range for this field is 0–65535. You can specify cycle time as a constant for A or as a readable variable, where the run-time content of the variable establishes the cycle time.
- A task can consist of multiple segments, each preceded by a TASK instruction. The segments do not have to be contiguous (Figure 5-9b). All segments for a TASK2 are executed within the cycle time specified in the TASK2 instruction for the first segment in the program. Values specified in subsequent segments are ignored.

When the cyclic RLL task does not complete execution within the specified cycle time, the appropriate status word bits are set. These are described in Appendix G.

---

**NOTE:** You can use any of the RLL instructions in a cyclic RLL task. Using cyclic RLL for immediate I/O applications and keeping the cyclic RLL task as small as possible minimizes the impact to the normal RLL scan.

---



**Figure 5-9 Examples of Cyclic RLL Design**

## Cyclic RLL (continued)

### Cyclic RLL Execution

An RLL program that contains a cyclic RLL task is executed as follows.

- The cyclic RLL task is executed periodically throughout the entire controller scan, interrupting the discrete scan and the analog scan as necessary.

**NOTE:** The execution of a cyclic RLL task is not synchronized with the normal I/O update or the normal RLL execution. If a cyclic RLL task uses a value computed by the normal RLL task, you must plan your program carefully to ensure correct operation when the value is not fully determined. For example, the cyclic RLL task can run between the execution of the ADD and SUB boxes in Figure 5-10.

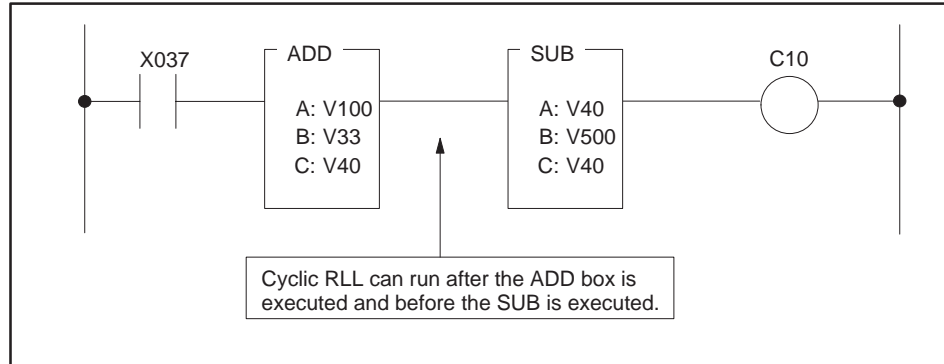


Figure 5-10 Example of Cyclic RLL Execution Interrupt

- If the cyclic RLL completes execution in less than the time specified by cycle time, execution does not resume until cycle time expires (Figure 5-11).

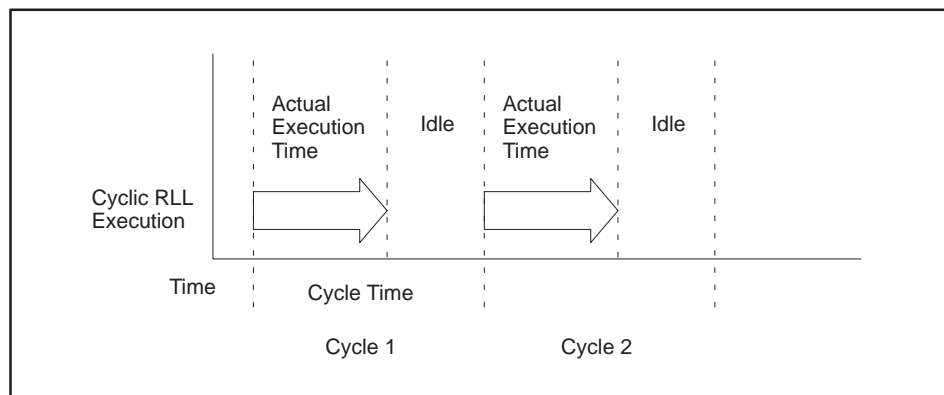


Figure 5-11 Relationship of Cyclic RLL Execution Time to Cycle Time

- Cycle time can be a constant or a variable. As a variable, the cycle time can be changed by logic in the main program, logic in the cyclic RLL task itself, or by other processes. The new cycle time does not take effect until the current execution of the cyclic RLL task has completed. See the example in Figure 5-12.
- If cycle time expires before a cyclic task completes, an overrun is reported in STW219, and the cycle that should have executed upon the expiration of A is skipped.

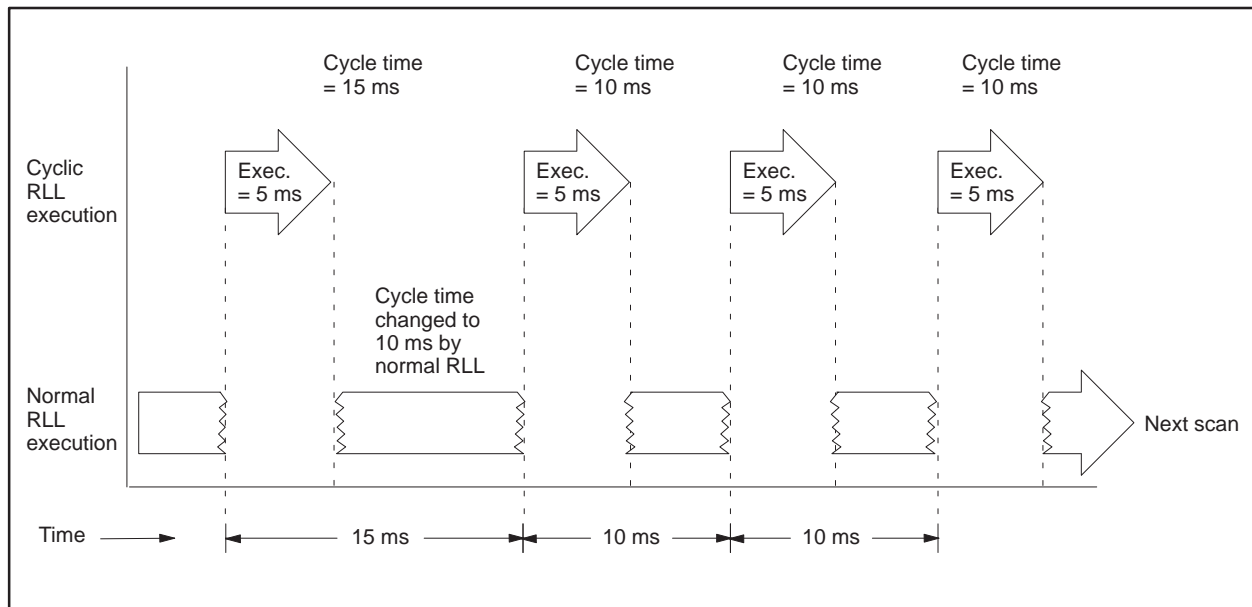


Figure 5-12 When Cycle Time Changes Take Effect

Refer to Chapter 6 for more information about how to use the TASK instruction.

## 5.5 Interrupt RLL (545/555 only)

### The Interrupt RLL Task

The interrupt RLL task (available on the 545 and 555 only) is the user program entity that is executed upon the occurrence of an interrupt request from an interrupt module. You can create only one interrupt task, and within it, you must include the RLL instructions required to handle all of the possible interrupt events in your application.

The TASK instruction, described in Chapter 6, is used to partition the interrupt RLL task from the main and cyclic RLL tasks. The interrupt RLL task is denoted as TASK 8 and can be composed of either one segment or multiple segments in the controller's L-Memory area, but it must be located before the first END statement of the program. Refer to Figure 5-13 for examples of user program partitioning.

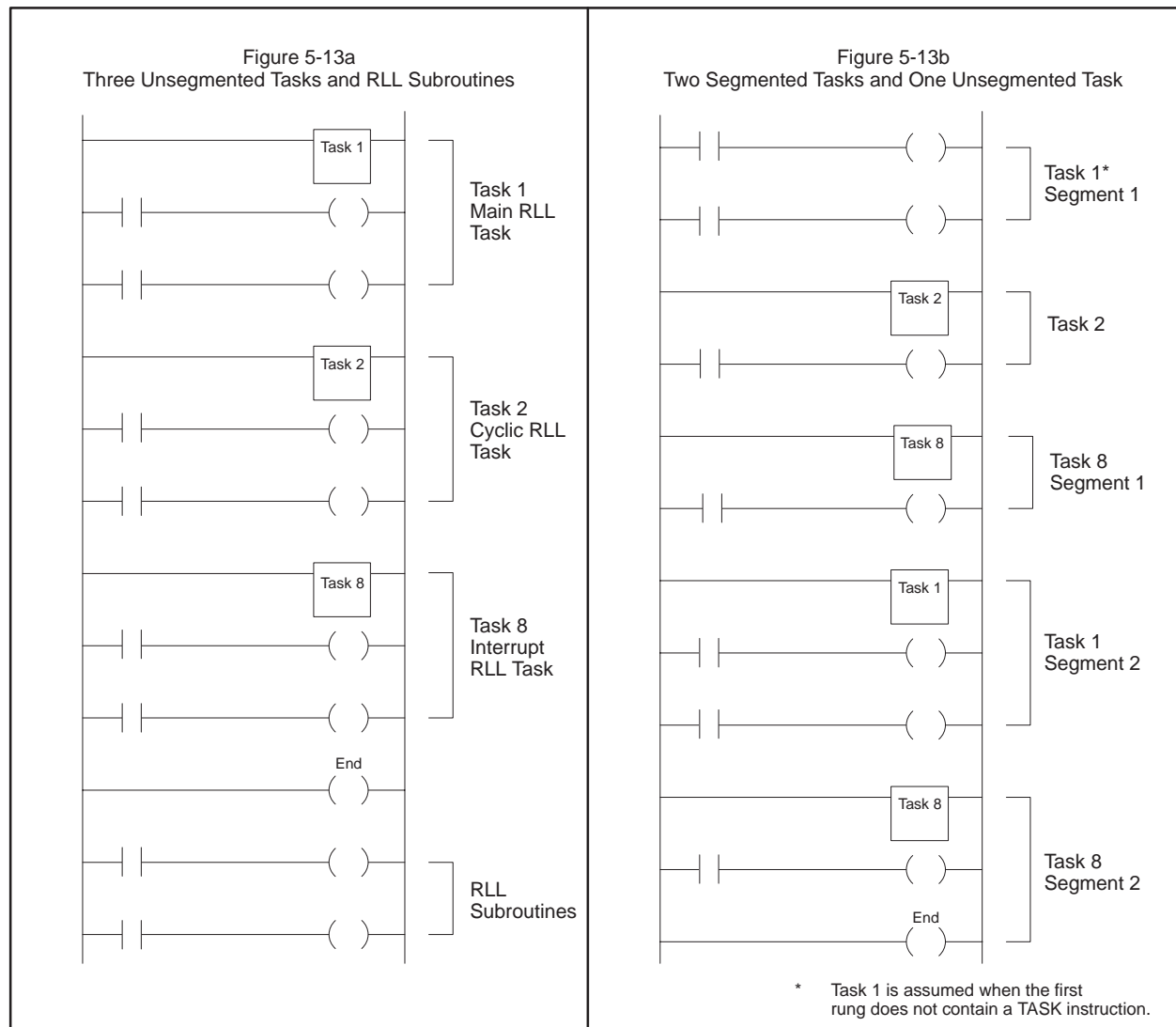


Figure 5-13 Examples of Cyclic RLL Design

TASK 8 of your RLL program is executed whenever the controller receives an interrupt request from one or more interrupt modules installed in the local base. An interrupt request is generated by a module when one or more of its field inputs undergoes a transition matching the transition type configured for the inputs.

Since multiple field inputs may simultaneously undergo transitions in your system, a given interrupt request issued to the controller can result from transitions occurring simultaneously at multiple inputs on one or more modules. Therefore, your TASK 8 program must be written to handle interrupts from multiple sources in a single execution pass. Your program must incorporate the status word STW220 and the module's interrupt status points to determine the source(s) of a given interrupt request.

When an interrupt request occurs, the controller determines which modules are involved (or "participating") in that request and places that information into status word STW220 in the format shown in Figure 5-14. If you are using more than one interrupt module, you must use the values stored in STW220 in your TASK 8 program to make decisions on whether or not the interrupt handlers for a module should be executed. (Remember, more than one module may be generating interrupt requests simultaneously.)

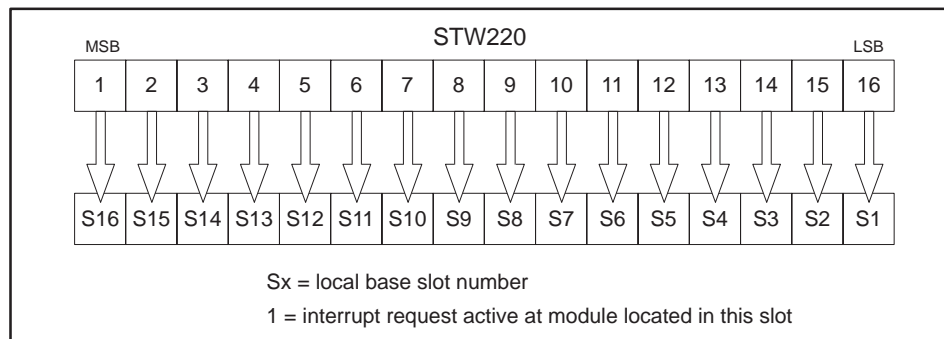


Figure 5-14 Status Word 220 Format

Once the TASK 8 program has determined that a module is involved in the current interrupt request, it must determine which of the module's field inputs were responsible for the generation of that request. The interrupt status points (points 17 – 24) of the Interrupt Input module provide that information. You can use the IORW instruction, described in Chapter 6, to read the interrupt status points of the module and store their values into the image register of the controller. Your program can then use these values to make decisions on which interrupt handlers to execute and which ones to bypass.



## Interrupt RLL (545/555 only) (continued)

Refer to Figure 5-15 for an example of an RLL program that uses STW220, IORW instructions, TASK 8, and the interrupt status input points of the module to execute handlers for inputs participating in the current interrupt request and to bypass handlers for non-participating inputs.

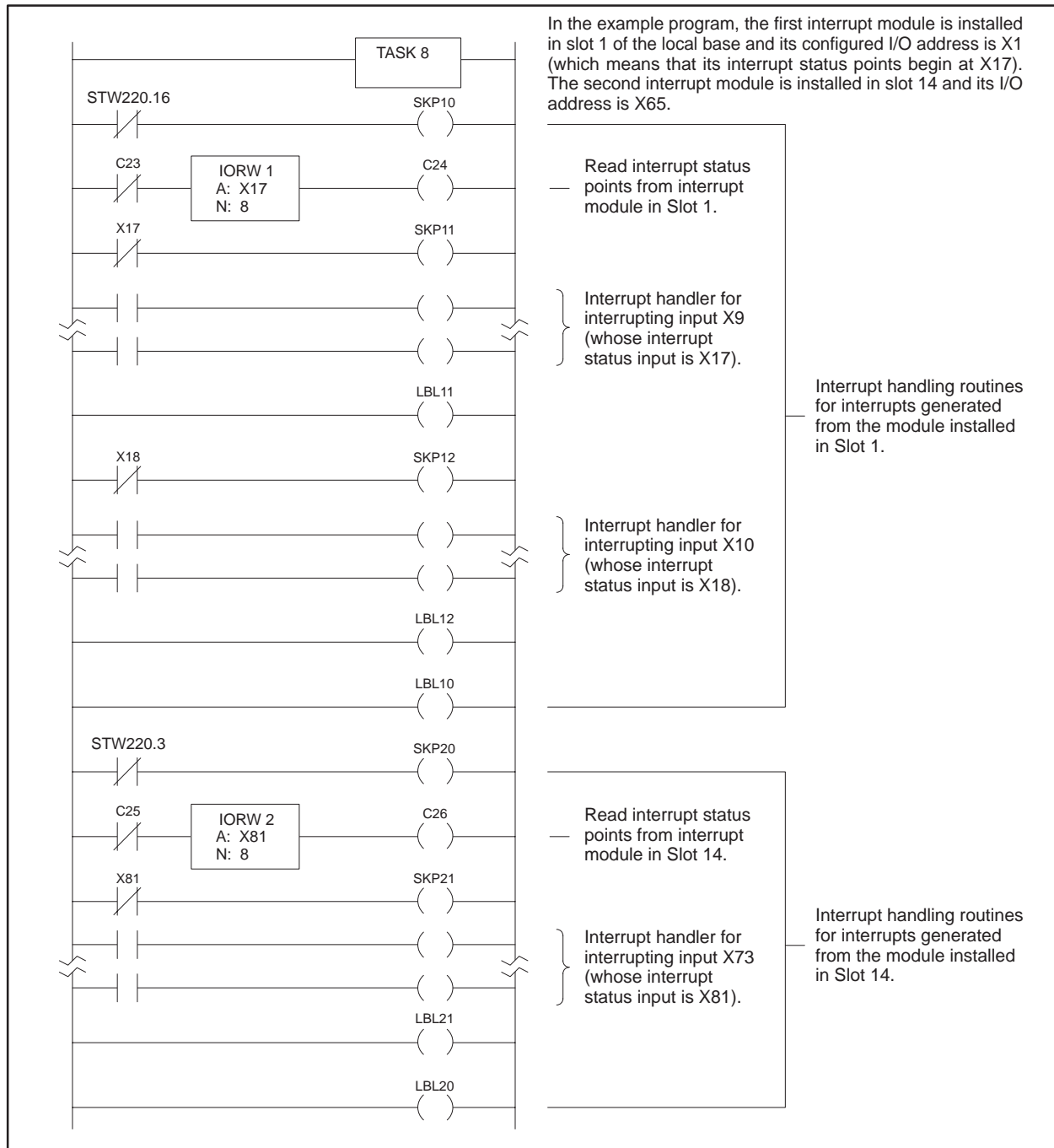


Figure 5-15 Example RLL Interrupt Program

---

## Operation

A number of qualifying conditions determine whether the controller executes the interrupt RLL task upon the occurrence of an interrupt request. The interrupt RLL task is not executed if the following are true:

- The controller is in the PROGRAM or FAULT modes.
- The controller is in the process of switching from EDIT to RUN mode.
- The controller is in the process of reconfiguring I/O.
- Interrupt requests are received from a module that is failed, not configured, or incorrectly configured in the controller's I/O map.

If interrupt requests occur simultaneously from both a correctly configured module and an incorrectly configured module, only the bit in STW220 corresponding to the correctly configured module is set to 1 before the interrupt RLL task is executed. (Bit positions corresponding to slots not participating in the current interrupt request also contain a 0.)

Upon detection of an interrupt request, the controller performs the following sequence of actions:

- Acknowledges the interrupt request, to clear the interrupt request backplane signal and to obtain an indication of which modules are participating in this interrupt request.
- Determines whether each participating module is qualified to issue interrupts (based upon configuration and failure state, as described above), and then writes the resultant bit pattern into STW220.
- Executes the interrupt RLL task if the qualifying conditions are met.
- Sends a rearm signal to each participating module, to clear the current interrupt request and to allow new interrupt requests.

## Interrupt RLL (545/555 only) (continued)

---

### Performance Characteristics

The interrupt input feature is designed for rapid response to external events, which is implemented by servicing interrupt requests at a very high priority. Because of this emphasis, you must take care to minimize the length of the interrupt RLL program in order to avoid affecting other time-dependent functions in the controller.

---

**NOTE:** Excessive time spent by the controller executing interrupt RLL can delay the execution of loops, analog alarms, and cyclic SF programs, extend the scan time of the controller, degrade the performance of the communication ports and remote I/O, and possibly result in a timeout of the scan watchdog timer, causing the controller to enter FAULT mode.

---

The amount of interrupt RLL execution time is determined both by the length of the TASK 8 program and the rate of interrupt requests. The execution time of your TASK 8 program can be determined by using the Ladder Logic Instruction Execution Time data in the Release Notes which accompanied your controller or firmware upgrade kit. The rate of interrupt requests is solely dependent upon your application.

It is important to know that the maximum delay through the Interrupt Input Module of an interrupt event is 0.5 ms (with 10-ms filter off) and that the maximum delay time in the controller in reacting to the interrupt generated by the Interrupt Input Module is also 0.5 ms. Therefore, the TASK 8 interrupt RLL program begins execution within 1 ms of the occurrence of a signal transition detectable by the Interrupt Input Module (assuming that no other interrupt inputs are being processed).

Using the above information, the minimum acceptable sustained interval between interrupt requests is as follows:

$$\text{Interrupt interval}_{\min} \text{ (in ms)} = 2 * (\text{TASK 8 max. execution time} + 1)$$

For example, if the maximum execution time of your TASK 8 program is 0.75 ms, then the controller can continuously handle interrupt requests occurring at intervals down to  $(2 * (.75 + 1))$  or 3.5 ms. The controller can handle bursts of interrupt events occurring at shorter intervals but sustained interrupt activity occurring at intervals shorter than the recommended time will result in system degradation.

---

## Troubleshooting

Successful operation of the interrupt input feature depends upon the following conditions.

- The interrupt input module is correctly configured.
- The I/O configuration stored in the controller for the Interrupt Input Module is correct.
- The interrupt RLL task is correctly designed and implemented.

Each interrupt module installed in the local base must be correctly configured in the I/O map of the controller. When in the interrupt mode, each module logs in as having 24 discrete inputs and 8 discrete outputs. Additionally, the module must not be reporting itself as failed.

The example of an interrupt RLL task shown in Figure 5-15 provides a guide for the development of your interrupt RLL task. If problems with the execution of your interrupt RLL task occur, verify that your logic for determining the source of the interrupt request is correct. Remember the following points:

- STW220 identifies which interrupt modules in the local base have an active interrupt request. Use STW220 to determine which module or modules triggered the current execution of the interrupt RLL.
- The status of each internal point (17 – 24) of the Interrupt module indicates the interrupting points responsible for generating the current request. Use the immediate I/O read instruction (IORW) to read the interrupt status point values from the module. (Refer to Table 3-2 on page 3-12 and the *Interrupt Input Module User Manual*.)

Also, remember to enable the interrupting points used in your application. This is done in the normal RLL (TASK 1) program. You must set the interrupt enable output points in the module to allow operation of the interrupting input points that you are using (see Table 3-2).

Status word STW221 can assist you in tracking down problems with interrupt input operation. STW221 contains a count of interrupts generated by modules on the local base. Whenever a module generates an interrupt request to the controller, STW221 is incremented by one (even though the module may have multiple actively interrupting points). Interrupt requests increment STW221 in any operating mode of the controller (except FAULT). For example, you can debug some of the interrupt operation in PROGRAM mode by manually causing a signal transition of the correct direction at a field input on the interrupt module and verifying that STW221 increments. (The interrupt RLL task is not executed since the controller is in PROGRAM mode.) This validates that the interrupt module is detecting the field input transition and is generating an interrupt to the controller and that the controller recognizes the interrupt. This does not validate that the module is correctly configured in the I/O map or that your interrupt RLL program is correct.

## 5.6 Using Real-Time Clock Data

### BCD Time of Day

Status Words 141–144 contain the status of the real-time clock at the start of the last I/O update. The real-time clock data includes the following information:

- Year (two digits), month, day of month, and day of week
- Hour, minute, second, and fraction of second, in 24-hour format

The clock data is stored in the status words in BCD format and is updated at the start of the I/O cycle, once per controller scan. The clock is backed up by battery and continues to keep time during a power shutdown.

You can use the Move Element (MOVE byte), or Word Rotate (WROT) and the Word AND (WAND) instructions to obtain specific segments of the status words containing the individual time items, such as minutes or seconds, for use in your RLL program.

Figure 5-16 shows the location of each item of information available with the clock status words. Each division in the figure represents four bits.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
STW141	Year–Tens				Year–Units				Month–Tens				Month–Units			
STW142	Day–Tens				Day–Units				Hour–Tens				Hour–Units			
STW143	Minute–Tens				Minute–Units				Second–Tens				Second–Units			
STW144	Second–Tenths				Second–Hundredths <sup>1</sup>				0				Day of Week			

<sup>1</sup> Always 0 for 575.

Figure 5-16 Status Word Location of Time Data

Figure 5-17 illustrates clock information on the date: Monday, 5 October, 1992, at 6:39:51.76 P.M. Note that the 24-hour format is used and Sunday is assumed to be day 1.

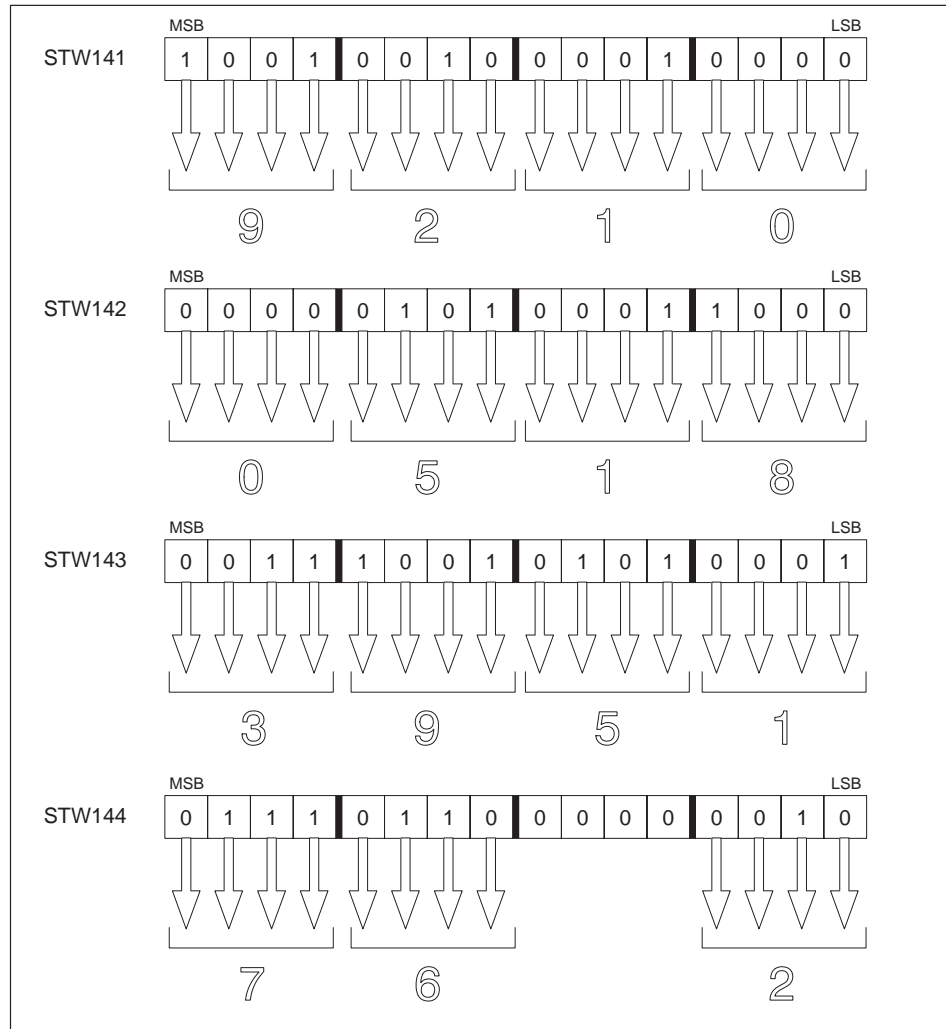


Figure 5-17 Clock Data Example

## Using Real-Time Clock Data (continued)

**Binary Time of Day** Binary time of day is contained in status words STW223 through STW225. STW223 and STW224 contain a 32-bit binary representation of the relative millisecond of the current day. STW225 contains a 16-bit binary representation of the current day relative to 1-January-1984 (day 0). Figure 5-18 shows the binary time-of-day status words.

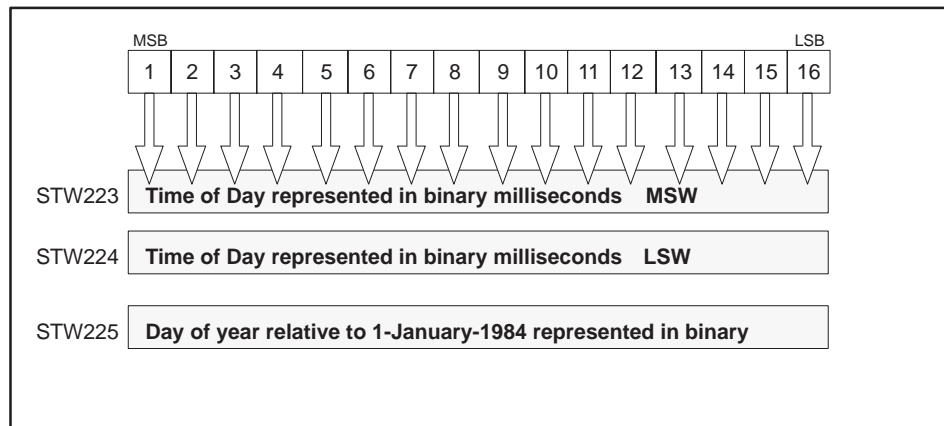


Figure 5-18 Binary Time of Day

## Time of Day Status

STW226 contains the time of day status. See Figure 5-19. The status word contains the following information:

- Bit 1 is a 1 when the current time is prior to the time reported on the last Task 1 RLL scan.
- Bit 10 is a 1 when the time has been set and is valid.
- Bit 11 is a 1 when the time of day is synchronized over a network.
- Bits 12 and 13 define the time resolution as follows:
  - 01 = 0.01 second
  - 10 = 0.1 second
  - 11 = 1.0 second
- Bit 14 is a 1 when there is a time synchronization error. This bit is set if the CPU does not receive a time update from the network at the expected time.
- Bit 15 is a 1 when there is no time-synchronization input from the time transmitter network.

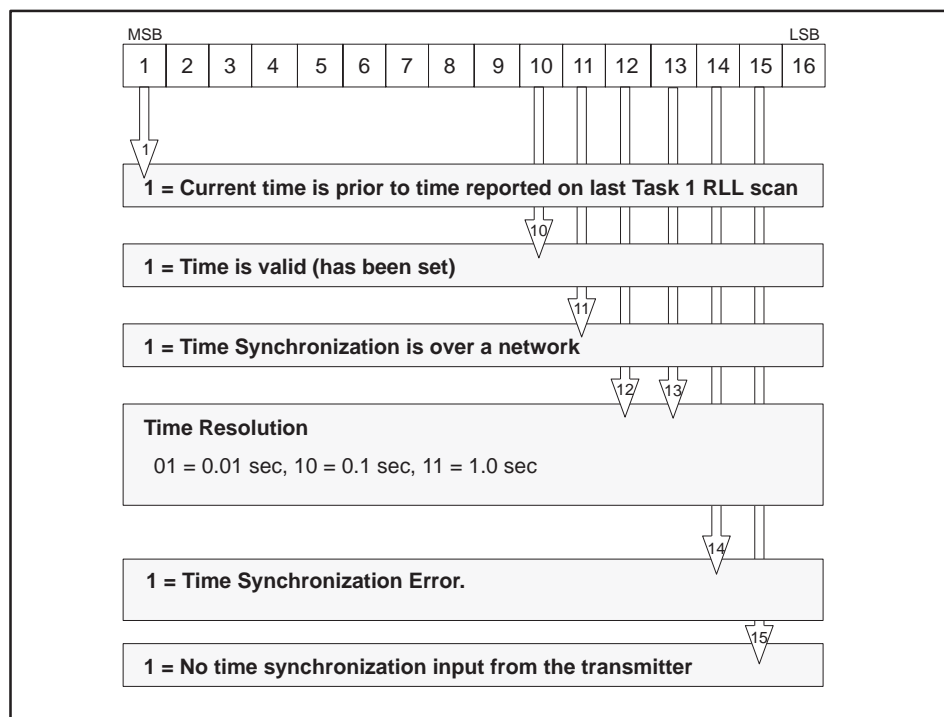


Figure 5-19 Time-of-Day Status Word



## 5.7 Entering Relay Ladder Logic

---

### SoftShop 505 for Windows

SIMATIC 505 SoftShop™ for Windows® is a Windows-based programming software that supports all SIMATIC 505 series programmable controllers. It provides familiar Windows features such as menu-driven commands, tool bars, and point-and-click functions to help simplify creating and editing your application program. Refer to your SoftShop manual for detailed instructions about how to enter a program.

SoftShop for Windows Release 2.1 or greater is required to support all the new features in the 555–1105 and 555–1106 CPUs.

### TISOFT

You can use the TISOFT programming software to create and edit your application program. TISOFT allows you to work directly in the ladder logic environment as you design the RLL program. For loops, analog alarms, and SF programs, TISOFT presents menu-driven programming tools.

To program the features described in this manual, you need TISOFT 6.3 or greater, which runs on an IBM® PC/AT™ compatible personal computer. Refer to your TISOFT manual for detailed instructions about how to enter a program.

---

**NOTE:** TISOFT 6.3 does not support the following features in the 555–1105 and 555–1106 CPUs: SmarTune™, PID loops above 64, the RLL PID Fast Loop box instruction, or analog alarms above 128.

---

### Using APT

You can use the SIMATIC Application Productivity Tool — APT™ to program your controller. APT is a graphic programming environment that eliminates the need for you to work in relay ladder logic when you create your application program. APT presents a familiar structure for process engineers, allowing them to become more closely involved in up-front design work. When the APT program is compiled, an RLL program is produced, generating the language with which the electrician or maintenance person is already familiar. Refer to the APT manual set for more information.

### COM PROFIBUS

The COM PROFIBUS configuration utility is a Windows® 3.1-based tool for configuring PROFIBUS-DP I/O slaves. Refer to the *SIMATIC 505 TISOFT2 User Manual* and the *SIMATIC S5 ET 200 Distributed I/O System Manual* for information about using COM PROFIBUS.

## 5.8 Doing Run-Time Program Edits

---

Your controller allows you to edit the RLL control program of a process that is running. This section provides guidelines for doing run-time edits.

### **WARNING**

Care must be exercised when doing run-time edits.

Incorrect actions can result in the failure of the process being controlled, which could result in death or serious injury to personnel, and/or damage to equipment.

Carefully plan any run-time edits to an active process. Avoid doing run-time edits to an active process if at all possible.

### Editing in Run Mode

Run-time edits to the RLL program are made in the EDIT mode. The controller enters the EDIT mode automatically when you enter the first edit change. While in EDIT mode, the process is controlled by the RLL program as it existed prior to the controller entering the EDIT mode.

### **WARNING**

TISOFT or SoftShop supports some controller models that do not support certain RLL instructions and/or memory configurations. It may allow you to enter unsupported RLL instructions, and depending upon memory configuration, may allow you to enter unsupported memory addresses for RLL instructions. Be aware that, if you do a run-time edit and enter an unsupported RLL instruction or an unsupported memory address, the controller enters PROGRAM mode and freezes all outputs.

This could cause unpredictable operation, which could result in death or serious injury to personnel, and/or damage to equipment.

Refer to the documentation for your controller model to see which memory types are supported, and what their maximum size can be. Use the syntax check function to validate a program before setting the controller to RUN mode.

You can modify one or more networks, as required, to accomplish the complete modification. After all required modifications are complete, request a SYNTAX CHECK to verify that the changes compile correctly. If errors are detected by SYNTAX CHECK, you can correct these errors and then re-execute the SYNTAX CHECK. This process can be repeated until the syntax check is successful, at which time you can set the controller to the RUN mode.

## Doing Run-Time Program Edits (continued)

---

When you select RUN mode, the controller compiles the edited RLL program. If you did not run the SYNTAX CHECK and errors are detected during the RLL compilation, the controller transitions to the PROGRAM mode, freezing the outputs in their current state. Actions that result in an error are listed in the “Avoid These Actions During Run-Time Edit” Section on pages 5-34 to 5-36. If no errors are detected during the RLL compilation, the controller transitions to the RUN mode and the newly-edited RLL program assumes control of the process.

---

**NOTE:** The process experiences a temporary scan extension during the compilation of the edited program. The length of the scan extension depends upon the size of the RLL program (30–70 ms per K-words of programmed RLL on a 545 or a 575, and approximately have of that time on a 575).

---

### Avoid These Actions During Run-Time Edits

The actions listed in this section cause the controller to enter the PROGRAM mode with outputs frozen in their current state, if present when RUN mode is selected from EDIT mode. For users of TISOFT 4.2 or later, or SoftShop, these conditions are detected and can be corrected prior to selecting RUN mode when you use the SYNTAX CHECK function.

### **WARNING**

The conditions that are described on the following pages can cause the process to become uncontrolled, which could result in death or serious injury to personnel, and/or damage to equipment.

It is your responsibility to provide for a safe recovery in the event of the occurrence of any of these conditions.

Be sure to observe the guidelines under the System Commissioning section of the Safety Considerations document (2588015–0003) included with your documentation.

SKP instruction without a corresponding LBL The LBL associated with a SKP instruction must exist within the same program segment (SBR or TASK) as the SKP instruction. If this is not the case, the controller transitions to PROGRAM mode and freeze the outputs.

*Solution* Ensure that both instructions have been entered before selecting RUN mode.

---

**SBR instruction without a terminating RTN** A subroutine must be terminated by an unconditional RTN instruction. If this is not the case, the controller will transition to PROGRAM mode and freeze the outputs.

*Solution* Ensure that both instructions have been entered before selecting RUN mode.

**GTS, PGTS or PGTSZ without corresponding SBR** The subroutine referenced by a GTS or PGTS(Z) instruction must be defined before it can be referenced. If this is not the case, the controller will transition to PROGRAM mode and freeze the outputs.

*Solution* Ensure that both instructions have been entered before selecting RUN mode.

**Use of unsupported features** Your RLL program must not use an instruction that is not supported by the software release installed in your controller, nor may it reference undefined or unconfigured data elements.

TISOFT and SoftShop have been designed to support a wide range of controllers. Since a given controller may not support all instructions supported by TISOFT or SoftShop, it is possible to enter an instruction that is not supported by your controller. If you enter an unsupported instruction or reference an unconfigured variable location, the CPU will transition to PROGRAM mode and freeze the outputs.

*Solution* Ensure that the instruction that you intend to use is supported by the software release installed in your controller. Use the SYNTAX CHECK function to verify the program before selecting RUN mode.

## Doing Run-Time Program Edits (continued)

---

**Exceeding L-Memory** When you edit an RLL program, it is possible for the edited program to exceed L-Memory. This can occur in two ways, as described below.

**First**, when you modify or insert a new network, the networks following the edited network are “pushed down” toward higher L-Memory addresses. If the configured L-Memory capacity is exceeded, one or more networks at the end of the program will be deleted to make room for the edit. TISOFT and SoftShop provide a warning of this condition prior to entering the editing change. After selecting RUN mode, the controller enters RUN mode, assuming none of the other conditions described above is present.

*Solution* Prior to making run-time edits, ensure that L-Memory can hold the entire program. With TISOFT 6.3 or greater, select AUX 28 (or select the SoftShop menu command **PLC Utilities** → **PLC Status...**) to determine the memory availability status of your controller. Otherwise, follow the steps below:


1. Determine the configured L-Memory size by using the TISOFT Memory Configuration function. Remember to convert K bytes (shown on the Memory Configuration display) to K words (1 word = 2 bytes).
2. Find the end of the RLL program.
3. Subtract the rung number of the NOP, which follows the last network of your program, from the configured K words of L-Memory that you determined in step 1. This is the amount of available L-Memory.
4. If the size of the additional logic exceeds the amount of available L-Memory, do not do the run-time edit.

**Second**, configured L-Memory can be exceeded when the compiled RLL program is more than twice as large as the uncompiled program. When you configure L-Memory, the system allocates two bytes for the compiled program for every byte of RLL memory. Usually this is sufficient to ensure that the compile does not run out of memory. However, if your RLL program contains a high percentage of SKP instructions relative to contacts and coils, it is possible to exceed the allocated compiled program memory. If this happens following an edit, the controller transitions to PROGRAM mode and freezes the outputs at the current state.

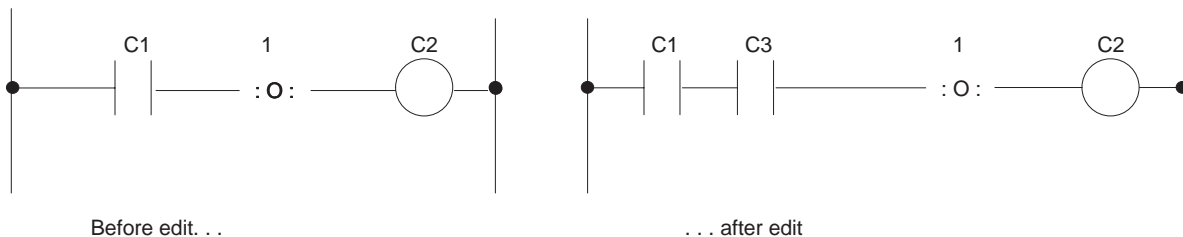
*Solution* With TISOFT 6.3 or greater, select AUX 28 (or select the SoftShop menu command **PLC Utilities** → **PLC Status...**) to determine the memory availability status of your controller. Otherwise, you can use the SYNTAX CHECK function to detect this problem before selecting the RUN mode.

**Additional Considerations When Doing Run-Time Edits**

When you edit an existing network, TISOFT or SoftShop deletes the existing network and then inserts the edited network in its place. If the existing (pre-edit) network has an instruction with retained state information, and if this instruction remains in the network after the edit, unexpected results may be obtained following the edit. These unexpected results occur due to initialization of the state information for the “retained state” instruction.

 <b>WARNING</b>
<p>When editing an existing network, TISOFT or SoftShop deletes the existing network and then inserts the edited network in its place.</p> <p>If the existing (pre-edit) network has an instruction with retained state information, and if this instruction remains in the network after the edit, you could experience unexpected results (following the edit) that could result in death or serious injury to personnel, and/or damage to equipment.</p> <p>Table 5-1 lists RLL instructions with retained state information along with the initialization performed by these instructions when they are compiled on the to-RUN transition following an edit. If you must edit a network containing one of these instructions, you must consider the effect upon the process caused by this initialization and ensure that the process state can safely handle this effect. Additional information concerning state initialization can be found in Section 4.2.</p>

For example, consider the following edit operation:



In this edit, the intent is to add a dependency on C3 for the C2 output. Due to the edit, however, the C2 output may be unexpectedly driven for one scan. This will occur, for example, if C1 is on during the edit process and both C1 and C3 are on when the controller enters the run mode following the edit.

## Doing Run-Time Program Edits (continued)

Table 5-1 lists the RLL instructions that have retained state and also gives their initial state on the first run-mode scan following the edit operation.

Table 5-1 RLL Instructions and Condition After Edit

<b>Instruction</b>	<b>Initial Condition After Run-Time Edit</b>
CTR	Initialized to require a 0 to 1 transition of the count input with TCP (count preset) set to the instruction's preset value and TCC (current count) set to 0.
DCAT	TCP (time preset) and TCC (time remaining) are set to the preset value in the DCAT instruction; i.e., the alarm timer is restarted.
DRUM	DSP (preset step) and DSC (current step) are set to the preset step specified in the DRUM instruction. DCC (current count) is set to the programmed count for his preset step. (The process is now controlled by the preset step.)
DSET	Initialized to require a 0 to 1 transition of the input.
EDRUM	The count preset values for each of the drum's steps are copied from the EDRUM instruction to the corresponding DCP (count preset) variables. DSP (preset step) and DSC (current step) are set to the preset step specified by the instruction and DCC (current count) is set to the programmed count for this preset step. Finally, the jog input is initialized to require a 0 to 1 transition. (The process is now controlled by the preset step.)
MCAT	TCP (time preset) and TCC (time remaining) are set to the preset value in the MCAT instruction; i.e., the alarm timer is restarted.
MDRMD MDRMW	The count preset values for each of the drum's steps are copied from the MDRUM/MDRUMW instruction to the corresponding DCP (count preset) variables. DSP (present step) and DSC (current step) are set to the preset step specified by the instruction and DCC (current count) is set to the programmed count for this preset step. Finally, the jog input is initialized to require a 0 to 1 transition. (The process is now controlled by the preset step.)
MWFT MWTT	The table pointer is set to the table base and the move count is set to 0.
OS	Initialized to set the output on the first scan for which the input is a 1.
SHRB SHRW	Initialized to require a 0 to 1 transition on the input.
TMR	TCP (time preset) and TCC (time remaining) are set to the preset value in the TMR/TMRF instruction; i.e., the timer is restarted.
TSET	Initialized to require a 0 to 1 transition of the input.
UDC	Initialized to require a 0 to 1 transition of the count input with TCP (count preset) set to the upper limit specified in the UDC instruction and TCC (current count) set to 0.

## 5.9 Password Protection

---

---

**NOTE:** Use your programming software (SoftShop or TISOFT) to enable the password protection feature. Refer to *SIMATIC 505 TISOFT2 User Manual (PPX:TS505-8101-x)* or the SoftShop user manual for password protection programming information.

---

### Protected Program Elements

The password protection feature allows you to protect the following elements of the application program from unauthorized access:

- Memory configuration
- I/O configuration
- Scan tuning parameters (scan watchdog, scan type, time-line values, etc.)
- RLL Program, including constants (K-Memory)
- Loop Configurations
- Analog Alarm Configurations
- Special Function Programs and Subroutines
- User External Subroutines
- Application Dependencies (575 only)
- Password Protection Level

### Disabled and Enabled Passwords

The programmable controller may be in one of three states of password protection:

- **No Password:** The application program is not protected. Any user may enter an initial password.
- **Disabled Password:** The application program is not protected. Only an authorized user may change or delete the password. Any user may enable the password.
- **Enabled Password.** The application program is protected according to the protection level assigned to the password (see below). If a protected operation is attempted from any communications port, the operation is denied and an error response is given. Only an authorized user may change, delete, or disable the password.



## Password Protection (continued)

---

Password Protection Levels	Three levels of protection are available when a password has been entered and enabled.
	<ul style="list-style-type: none"><li data-bbox="495 378 1433 451">• <b>No Access:</b>                    The application program cannot be read or modified.</li><li data-bbox="495 472 1433 546">• <b>Read-only Access:</b>            The application program can be read but it cannot be modified.</li><li data-bbox="495 567 1433 598">• <b>Full Access:</b>                    The application program is not protected.</li></ul>
Determining the Current State of Password	The application program may dynamically determine the current state of password protection by examining status bits defined in STW1. (See Appendix G, Status Words.)
Password Effect on EEPROM	When the application program is stored in EEPROM the password information is stored as well. If an application program stored in EEPROM is password protected, the password will be automatically enabled following a power cycle or whenever you select to run out of EEPROM.

# Chapter 6

## RLL Instruction Set

---

6.1	Safety Considerations .....	6-4
6.2	Introduction .....	6-6
6.3	Absolute Value .....	6-11
6.4	Add .....	6-12
6.5	Bit Clear .....	6-13
6.6	Bit Pick .....	6-14
6.7	Bit Set .....	6-15
6.8	Convert Binary to BCD .....	6-16
6.9	Convert BCD to Binary .....	6-18
6.10	Compare .....	6-20
6.11	Coils .....	6-22
6.12	Contacts .....	6-23
6.13	Counter (Up Counter) .....	6-24
6.14	Discrete Control Alarm Timer .....	6-26
6.15	Date Compare .....	6-30
6.16	Divide .....	6-32
6.17	Time Driven Drum .....	6-34
6.18	Date Set .....	6-38
6.19	Time/Event Driven Drum .....	6-40
6.20	Unconditional End .....	6-44
6.21	Conditional End .....	6-45
6.22	Go To Subroutine .....	6-46
6.23	Indexed Matrix Compare .....	6-48
6.24	Immediate I/O Read/Write .....	6-50
6.25	Jump .....	6-52
6.26	Load Address .....	6-54

---

6.27	Load Data Constant .....	6-59
6.28	Lock Memory .....	6-60
6.29	Motor Control Alarm Timer .....	6-63
6.30	Master Control Relay .....	6-68
6.31	Maskable Event Drum, Discrete .....	6-72
6.32	Maskable Event Drum, Word .....	6-76
6.33	Move Image Register from Table .....	6-82
6.34	Move Image Register to Table .....	6-84
6.35	Move Image Register to Word .....	6-86
6.36	Move Element .....	6-88
6.37	Move Word .....	6-96
6.38	Multiply .....	6-98
6.39	Move Word from Table .....	6-100
6.40	Move Word with Index .....	6-102
6.41	Move Word to Image Register .....	6-104
6.42	Move Word To Table .....	6-106
6.43	NOT .....	6-108
6.44	One Shot .....	6-109
6.45	PID Loop .....	6-110
6.46	Parameterized Go To Subroutine .....	6-112
6.47	Parameterized Go To Subroutine (Zero) .....	6-118
6.48	Read Slave Diagnostic (RSD) .....	6-120
6.49	Return from Subroutine .....	6-122
6.50	Subroutine .....	6-123
6.51	Call an SF Program .....	6-126
6.52	Call SF Subroutines from RLL .....	6-128
6.53	Bit Shift Register .....	6-132
6.54	Word Shift Register .....	6-134

---

6.55	Skip / Label .....	6-136
6.56	Scan Matrix Compare .....	6-140
6.57	Square Root .....	6-142
6.58	Search Table For Equal .....	6-144
6.59	Search Table For Not Equal .....	6-146
6.60	Subtract .....	6-148
6.61	Table to Table AND .....	6-149
6.62	Start New RLL Task .....	6-150
6.63	Time Compare .....	6-153
6.64	Table Complement .....	6-154
6.65	Text .....	6-155
6.66	Timer .....	6-156
6.67	Table to Table OR .....	6-158
6.68	Time Set .....	6-159
6.69	Table to Word .....	6-160
6.70	Table to Table Exclusive OR .....	6-162
6.71	Up/Down Counter .....	6-164
6.72	Unlock Memory .....	6-167
6.73	Word AND .....	6-168
6.74	Word OR .....	6-170
6.75	Word Rotate .....	6-172
6.76	Word To Table .....	6-174
6.77	Word To Table AND .....	6-176
6.78	Word To Table OR .....	6-178
6.79	Word To Table Exclusive OR .....	6-180
6.80	Word Exclusive OR .....	6-182
6.81	External Subroutine Call .....	6-184

## 6.1 Safety Considerations

---

### Overview

A programmable controller is a programmed system. When you create or modify the control program, you must be aware that your program affects control actions that manipulate physical devices. If the program contains errors, these errors can cause the controlled equipment to operate in unpredictable ways. This could cause harm to anyone who uses the equipment, damage to the controlled equipment, or both. You must ensure that the control program is correct before you introduce it to the operational environment of the controlled process. Read this section carefully before you create or modify the control program.

### Failure of the Control System

The Series 505 controllers are highly reliable systems. However, you must be aware that these systems can fail. If a failure occurs, and if the control system is able to respond to the failure, the controller enters the Fatal Error mode. The Fatal Error mode sets all the discrete outputs to zero (off) and freezes all the word outputs at their values when the failure was detected. Your control system design must take the Fatal Error mode into consideration and ensure that the controlled environment can react safely if a Fatal Error occurs.

### **WARNING**

It is possible that the system could fail without being able to execute the Fatal Error actions. It is also possible for the system to continue to operate while producing incorrect results.

Operating and producing incorrect results could cause unpredictable controller behavior that could result in death or serious injury to personnel, and/or damage to equipment.

You must provide for manual overrides in those cases where operator safety could be jeopardized or where equipment damage is possible because of a failure. Refer to the safety considerations sheet (2583015-0003).

---

**NOTE:** Some user program errors can also cause the controller to enter the Fatal Error mode. Examples include corruption of SF instruction control blocks retained in V-Memory and VMEbus bus errors (for 575 only; see page 4-11).

---

---

**Inconsistent  
Program Operation**

You must ensure the correctness of your control program before you introduce it to the controlled process. An incorrect control program can cause the process to act incorrectly or inconsistently. Although any number of programming errors can cause control problems, one of the more subtle problems occurs with the incorrect assignment of instruction numbers for box instructions that have retained state information. The timer, counter, and drum instructions are examples of these instructions. Section 4.2 lists the various memory areas in the controller where retained state information is maintained. Section 4.2 also lists the restrictions that exist in assigning instruction numbers for the boxes that reference these areas. You must design your program to accommodate these requirements.

 **WARNING**

**Incorrect assignment of instruction numbers for retained state instructions could result in inconsistent controller action.**

**If this occurs, it could cause unpredictable controller action that could result in death or serious injury to personnel, and/or damage to equipment.**

**You must ensure that instruction numbers are assigned uniquely for boxes with retained state information. Refer to Section 4.2.**

**Editing an Active  
Process**

Performing edits on an active process involves a number of considerations that are detailed in Section 5.8. You must read and fully understand this information before you make any edits to the control program of an active process.

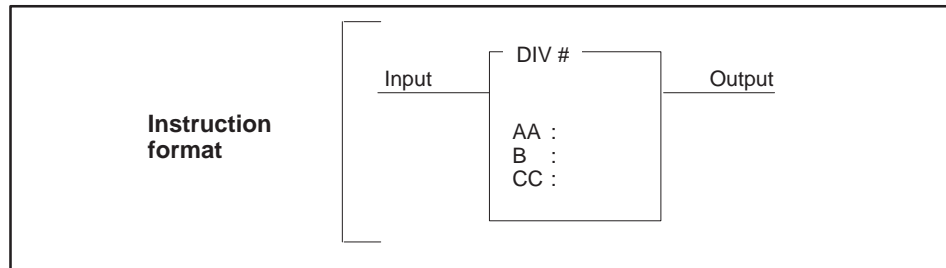
 **WARNING**

**Incorrect application of run-time edits could cause the controller to transition to the program mode, freezing both discrete and word outputs at their current status.**

**This could cause failure of the process that could result in death or serious injury to personnel, and/or damage to equipment.**

**Avoid doing run-time edits if you can. If you cannot avoid doing a run-time edit, then ensure that you have read and fully understood Section 5.8, and that your edits conform to the requirements of that section.**

This chapter describes the RLL instruction set that is supported by the 545, 555, and 575 controllers. Figure 6-1 shows how the instructions are illustrated. The fields that you use to program the instruction are defined below.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
AA	Any readable word	Memory location for the dividend. This is a long word. AA holds the 16 most significant bits, and AA + 1 holds the 16 least significant bits.
	or constant (-32768 to +32767)	Value of the dividend if a constant is used.
B	Any readable word	Memory location of the divisor (one word).
	or constant (-32768 to +32767)	Value of the divisor if a constant is used.
CC	Any writeable word	Memory location for the result. CC holds the quotient (a word); CC + 1 holds the remainder (a word).

Figure 6-1 RLL Instruction Format

Instruction Format illustrates how the instruction appears on the programming unit.

Field contains the various fields used to define an instruction. For a field that is denoted by a single character, e.g., B, the entry defines one word. If you enter V110 for field B in the division example (Figure 6-1), the controller reads the word at V110. For a field that is defined by a double character, e.g., AA, the entry consists of one long word. If you enter V55 for field AA in Figure 6-1, the controller reads the long word at V55 and V56.

Some fields are defined by two characters that are descriptors for the field. For example, TD = table destination; AI = an index into field A. For these fields, the parameter description specifies the field size.

Valid Values lists the valid constants and memory locations that can be used in this field.

A writeable memory location is defined as any memory location to which an RLL instruction can both read and write (Section A.1).

A readable memory location is defined as any memory location that an RLL instruction can read (Section A.1).

Function describes the purpose of the field.

Following an instruction's format and description, the function of the instruction is described.

RLL instructions are presented alphabetically for ease in reference. Table 6-1 lists the RLL instructions by functional groups.

Table 6-1 RLL Functional Groups

Operation Type	Instruction	Function	Page
Electro-mechanical Replacement	Coil	Normal or NOT output coil; control relay; set/reset coil; immediate coil; bit-of-word coil.	6-22
	Contact	Normal or NOT contact; control relay; immediate contact; bit-of-word contact; relational contact.	6-23
	NOT	Inverts power flow.	6-108
	MCR/MCRE	Master control relay.	6-68
	JMP/JMPE	Freezes outputs in zone of control.	6-52
	SKP/LBL	Selectively enable/disable program segments during scan.	6-136
	SHRB	Bit shift register.	6-132
	TMR/TMRF	Times events.	6-156
	DCAT	Discrete control alarm timer.	6-26
	MCAT	Motor control alarm timer.	6-63
	CTR	Counts recurring events.	6-24
	UDC	Counts events up or down.	6-164
	DRUM	Simulates electro-mechanical stepper switch.	6-34
	EDRUM	Simulates electro-mechanical stepper switch. Can be indexed by timer, event, or timer and event.	6-40
	MDRMD	Drum; uses configurable mask to control coils.	6-72
MDRMW	Drum; uses configurable mask to write to words.	6-76	



Table 6-1 RLL Functional Groups (continued)

Operation Type	Instruction	Function	Page
Bit Manipulation	BITC	Clears a specified bit.	6-13
	BITS	Sets a specified bit.	6-15
	BITP	Examines status of a specified bit.	6-14
	WAND	Does logical bit-by-bit AND on two words.	6-168
	WOR	Does logical bit-by-bit OR on two words.	6-170
	WXOR	Does logical bit-by-bit EXCLUSIVE OR on two words.	6-182
	WROT	Rotates the 4-segment bits of a word.	6-172
	SMC	Compares status of discrete points with a set of specified bit patterns.	6-140
	IMC	Compares status of discrete points with a specified bit pattern in a set of patterns.	6-48
	Bit-of-word contact	Examines status of a specified bit	6-23
Bit-of-word coil	Copies power flow to the specified bit	6-22	
BCD Conversions	CDB	Converts BCD inputs to binary.	6-18
	CBD	Converts binary to BCD value.	6-16
Word Move Instructions	LDC	Loads a constant to a memory location.	6-59
	LDA	Copies the logical address of a memory location into a memory location.	6-54
	MIRW	Copies bit status from control relays or discrete image register to a word.	6-86
	MWIR	Copies bits of a word to the discrete image register, or the control relay memory.	6-104
	MOVW	Copies words from one location to another.	6-96
	MOVE	Copies bytes, words, or long words from a source location to a destination location.	6-88
	MWTT	Copies a word to a table.	6-106
	MWFT	Copies a word from a table.	6-100
	SHRW	Word shift register.	6-134
MWI	Copies words from one location to another using indexed addresses.	6-102	

Table 6-1 RLL Functional Groups (continued)

Operation Type	Instruction	Function	Page
Math Instructions	ADD	Addition.	6-12
	SUB	Subtraction.	6-148
	MULT	Multiplication.	6-98
	DIV	Division.	6-32
	SQRT	Square Root.	6-142
	CMP	Compare.	6-20
	ABSV	Take absolute value of a word.	6-11
	Relational Contacts	Power flow depends on relational condition that exists between values in two readable words.	6-23
Table Instructions	MIRTT	Copies status of control relays or discrete image register bits to table.	6-84
	MIRFT	Copies a table into the control relay memory or discrete image register.	6-82
	TAND	ANDs the corresponding bits in two tables.	6-149
	TOR	ORs the corresponding bits in two tables.	6-158
	TXOR	Does an EXCLUSIVE OR on the corresponding bits in two tables.	6-162
	TCPL	Inverts status of each bit in a table.	6-154
	WTOT	Copies a word into a table.	6-174
	TTOW	Copies a word from a table.	6-160
	WTTA	ANDs bits of a word with the bits of a word in a table.	6-176
	WTTO	ORs bits of a word with the bits of a word in a table.	6-178
	WTTXO	Does an EXCLUSIVE OR on the bits of a word with the bits of a word in a table.	6-180
	STFE	Searches for a word in a table equal to a specified word.	6-144
	STFN	Searches for a word in a table not equal to a specified word.	6-146
Clock Instructions	DCMP	Compares current date with a specified date.	6-30
	TCMP	Compares current time with a specified time.	6-153
	TSET	Sets time in real-time clock.	6-159
	DSET	Sets date in real-time clock.	6-38

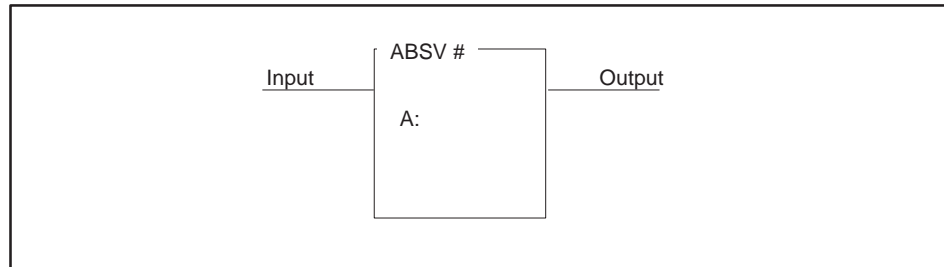
## Introduction (continued)

Table 6-1 RLL Functional Groups (continued)

Operation Type	Instruction	Function	Page
Subroutine Instructions	GTS	Calls a subroutine.	6-46
	PGTS	Calls an RLL subroutine and passes parameters to it.	6-112
	PGTSZ	Calls an RLL subroutine and passes parameters to it. Discrete parameters indicated as outputs are cleared when the subroutine is not executed.	6-118
	SBR	Designates the beginning of an RLL subroutine.	6-123
	RTN	Returns control from an RLL subroutine to the main RLL program.	6-122
	XSUB	Calls an externally developed subroutine and passes parameters to it.	6-184
	SFPGM	Calls a special function program from RLL.	6-126
	SFSUB	Calls a special function subroutine from RLL.	6-128
Miscellaneous Instructions	OS (One Shot)	Turns on output for a single scan.	6-109
	END	Unconditionally terminates a scan.	6-44
	ENDC	Terminates a scan conditionally.	6-45
	LOCK	Used together and provide a mechanism whereby multiple applications in the 575 system can coordinate access to shared resources.	6-60
	UNLCK		6-167
	PID	Performs the PID fast loop function	6-110
	RSD	Transfers a PROFIBUS-DP slave's current diagnostic to user memory.	6-120
	TASK	Start a new RLL program segment.	6-150
	TEXT	Places textual information into L-Memory.	6-155
Immediate I/O Instructions	Immediate Contact/Coil	Immediate I/O update.	6-22
	SETI/RSTI Coil	Immediate set/reset of a bit.	6-23 6-22
	IORW	Does immediate read or write to discrete or word I/O.	6-50

### 6.3 Absolute Value

**ABSV Description** The ABSV instruction (Figure 6-2) calculates the absolute value of a signed integer.



Field	Valid Values	Function
#	0–65535	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any writeable word	Specifies word that contains number of which absolute value is calculated.

Figure 6-2 ABSV Format

**ABSV Operation** When the input is turned on, the ABSV box executes. If the input remains on, the instruction executes on every scan. The operation executed is  $A = |A|$ .

- If  $A \geq 0$ , A is not changed, and the output turns on.
- If  $-32768 < A < 0$ , A is replaced with the value  $(0 - A)$  and the output turns on.
- If  $A = -32768$ , A does not change, and the output is off.

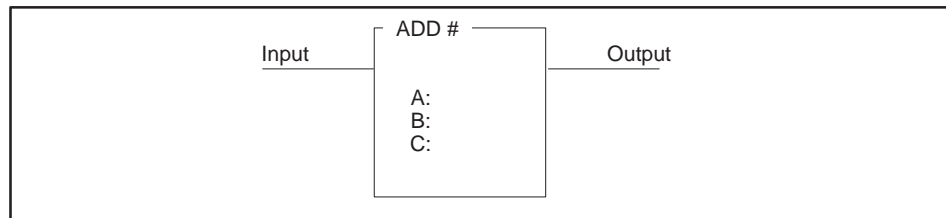
When the input is off, the instruction does not execute, and there is no power flow at the box output.

**See Also** These RLL instructions can also be used for math operations.

ADD	CMP	DIV	MULT	SQRT	SUB
Relational Contact					

6.4 Add

**ADD Description** The ADD instruction (Figure 6-3) adds a signed integer in memory location A to a signed integer in memory location B, and stores the result in memory location C.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any readable word	Memory location for first addend (a word).
B	Any readable word or constant (-32768 to +32767)	Memory location for second addend (a word).
		Value to be added if a constant is used.
C	Any writeable word	Specifies memory location for the sum (a word).

Figure 6-3 ADD Format

**ADD Operation** When the input is on, the ADD box is executed. If the input remains on, the instruction is executed on every scan. The operation executed is:  $C = A + B$ .

If  $-32768 \leq \text{sum} \leq 32767$ , then the output is turned on. Otherwise, the output is turned off, indicating an addition overflow, and C contains the truncated (16 bits) sum.

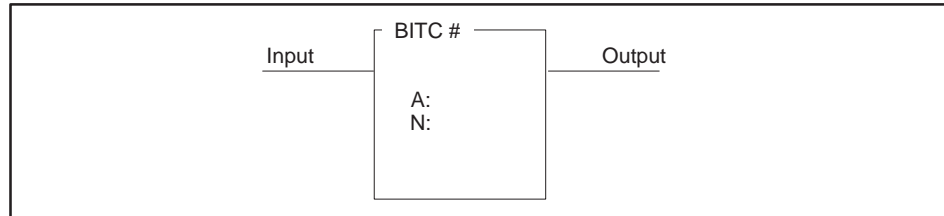
If the input is off, the instruction is not executed, and there is no power flow at the box output.

**See Also** These RLL instructions can also be used for math operations.

ABSV	CMP	DIV	MULT	SQRT	SUB
Relational Contact					

## 6.5 Bit Clear

**BITC Description** The Bit Clear instruction (Figure 6-4) clears a specified bit to zero.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any writeable word	Specifies memory location of word containing bit to be cleared.
N	1–16	Specifies bit position. The most significant bit (MSB) = 1; the least significant bit (LSB) = 16.

Figure 6-4 BITC Format

**BITC Operation** When the input is on, the BITC box executes. If the input remains on, the instruction executes on every scan. The operation executed is Bit N of word A is cleared to 0.

The output is turned on during each scan in which the instruction is executed.

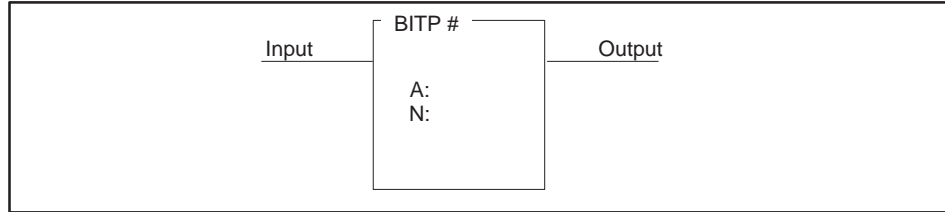
If the input is off, the instruction does not execute, and there is no power flow at the box output.

**See Also** These RLL instructions are also used for bit manipulation.

BITP	BITS	IMC	SMC	WAND	WOR
WROT	WXOR	Bit-of-Word Contact/Coil		Set/Reset Coil	

6.6 Bit Pick

**BITP Description** The Bit Pick instruction (Figure 6-5) examines the status of a specified bit.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any readable word	Specifies memory location of word containing bit to be examined.
N	1-16	Specifies bit position. The most significant bit (MSB) = 1; the least significant bit (LSB) = 16.

Figure 6-5 BITP Format

**BITP Operation** When the input is turned on, the BITP box executes. If the input remains on, the instruction executes on every scan. The operation executed is the status of bit N of word A is checked.

- The output is turned on if the selected bit is 1.
- The output is turned off if the selected bit is 0.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

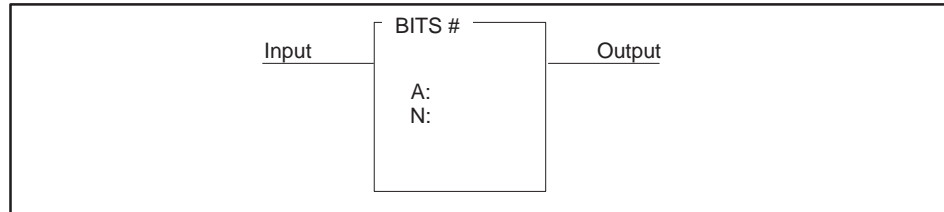
**See Also** These RLL instructions are also used for bit manipulation.

BITC	BITS	IMC	SMC	WAND	WOR
WROT	WXOR	Bit-of-Word Contact/Coil		Set/Reset Coil	

Refer to Section E.4 for an application example of the BITP.

## 6.7 Bit Set

**BITS Description** The Bit Set instruction (Figure 6-6) sets a specified bit to one.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any writeable word	Specifies memory location of word containing bit to be set to one.
N	1–16	Specifies bit position. The most significant bit (MSB) = 1; the least significant bit (LSB) = 16.

Figure 6-6 BITS Format

**BITS Operation** When the input is on, the BITS box executes. If the input remains on, the instruction executes on every scan. The operation executed is Bit N of word A is set to 1.

- The output is turned on during each scan in which the instruction is executed.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

**See Also** These RLL instructions are also used for bit manipulation.

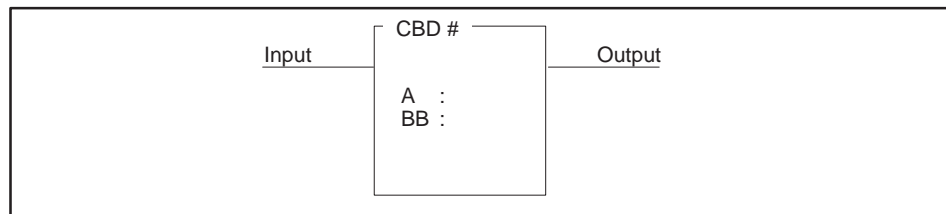
BITC	BITP	IMC	SMC	WAND	WOR
WROT	WXOR	Bit-of-Word Contact/Coil		Set/Reset Coil	



## 6.8 Convert Binary to BCD

### CBD Description

The Convert Binary to BCD instruction (Figure 6-7) converts a binary representation of an integer to an equivalent Binary Coded Decimal (BCD) value. That is, a 16-bit word representing an integer is converted into a 32-bit word in which each group of four bits represents a BCD digit. Values up to 32,767 are converted to equivalent BCD values.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any readable word	Specifies memory location of integer to be converted.
BB	Any writeable word	Specifies memory location of the BCD word after conversion. BB contains the most significant 16 bits, and BB + 1 contains the least significant 16 bits.

Figure 6-7 CBD Format

### CBD Operation

When the input is on, the CBD box executes. If the input remains on, the instruction executes on every scan. The operation of the CBD is described below and illustrated in Figure 6-8.

- If A contains an integer 0–32767, the value is converted to BCD and stored in BB and BB + 1 as shown below, and the box output is turned on.



- If A is not in the range 0–32767, there is no power flow at the box output, and BB and BB + 1 do not change.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

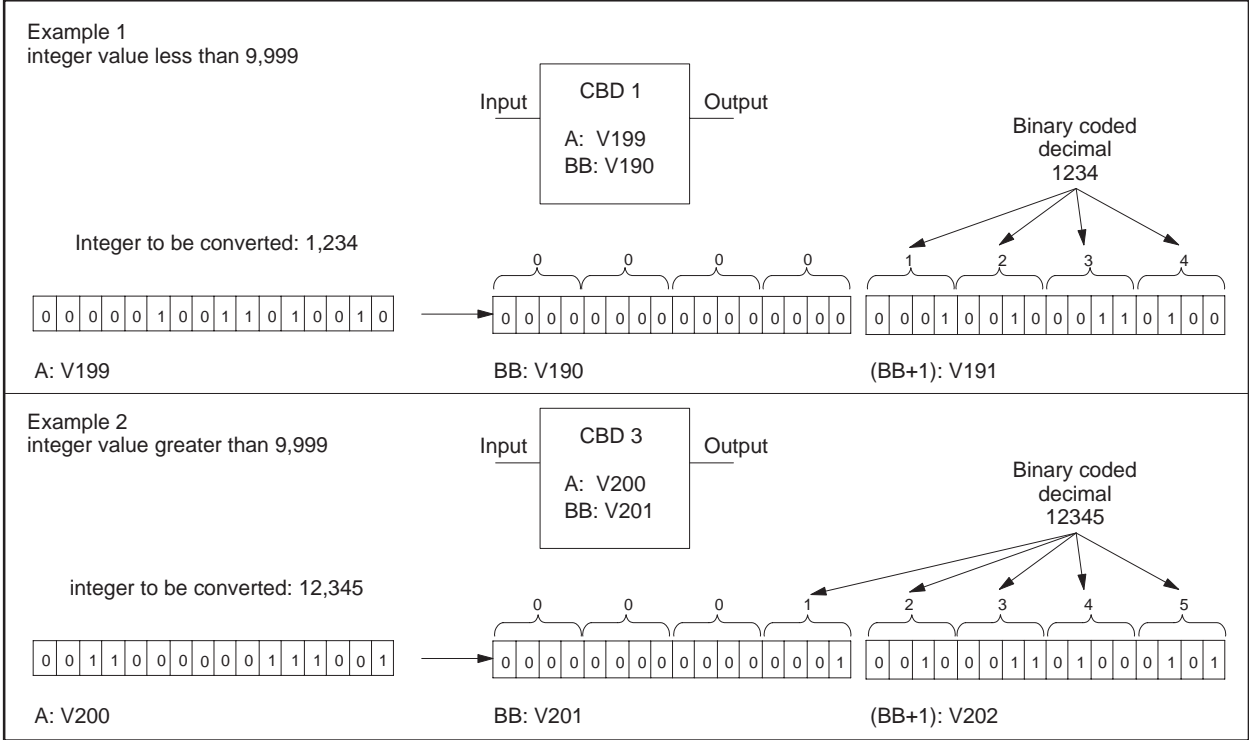


Figure 6-8 Examples of CBD Operation

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

See Also

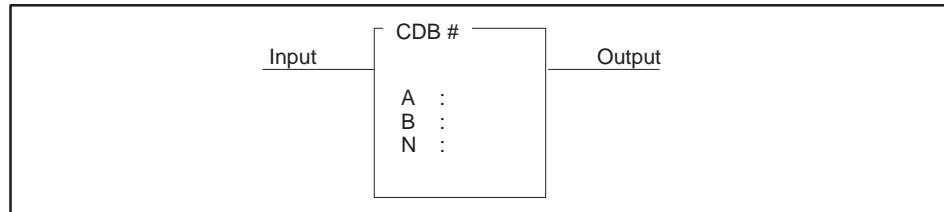
This RLL instruction can also be used for BCD conversions.

CDB

Refer to Section E.12 for an application example of the CBD.

## 6.9 Convert BCD to Binary

**CDB Description** The Convert BCD to Binary instruction (Figure 6-9) converts BCD inputs to the binary representation of the equivalent integer.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any readable word	Specifies memory location of BCD word to be converted.
B	Any writeable word	Specifies memory location of the integer after conversion.
N	1–4	Number of digits to be converted.

Figure 6-9 CDB Format

**CDB Operation** When the input turns on the CDB box executes. If the input remains on, the instruction executes on every scan. The operation of the CDB follows and illustrated in Figure 6-10:

- The number of digits (N) of the BCD value located in A, is converted to its equivalent binary integer value and stored in B.
- N may range from 1–4, and the BCD digit count is from right to left. For example, if N = 2 and the BCD number in A = 4321, then 21 is converted, and the value stored in B is 00010101.
- The output turns on after the instruction executes if the digits of the input word are valid. Each digit of the BCD value in A must be less than or equal to 9. The binary values 1010, 1011, 1100, 1101, 1110, and 1111 are invalid.

If the digits of the input word are not valid, the instruction does not execute, and the output does not turn on.

If the input is off, the instruction does not execute and there is no power flow at the box output.

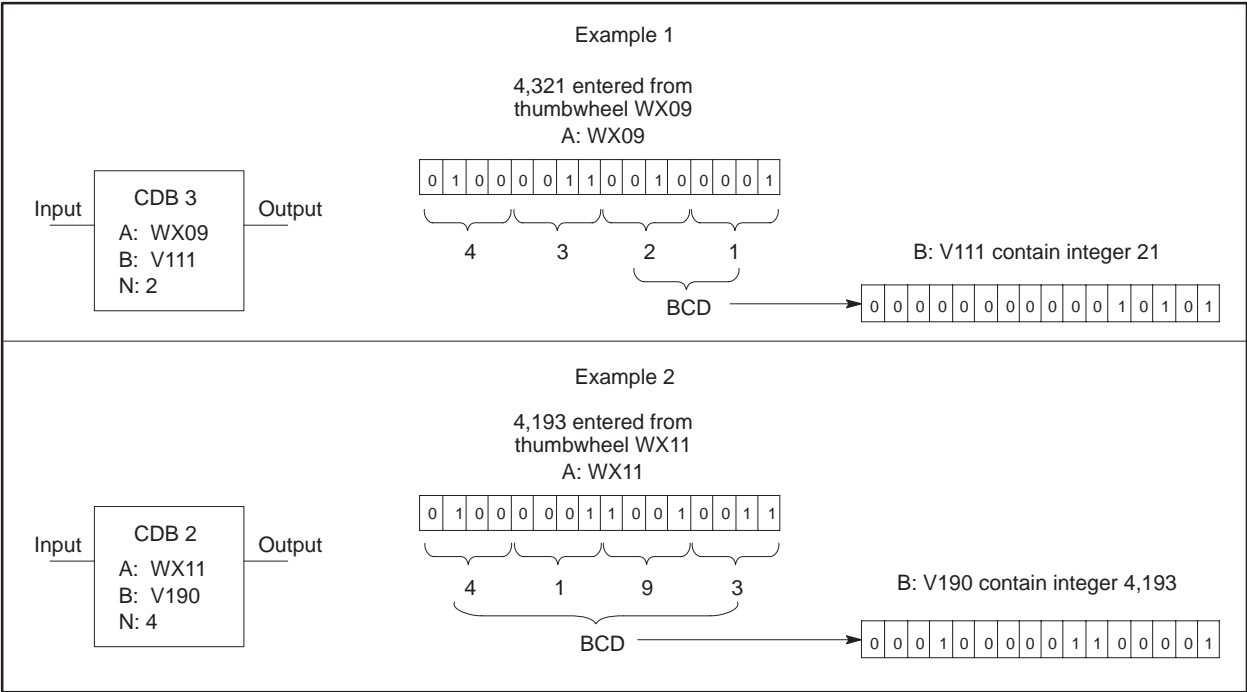


Figure 6-10 Examples of CDB Operation

See Also

This RLL instruction can also be used for BCD conversions.

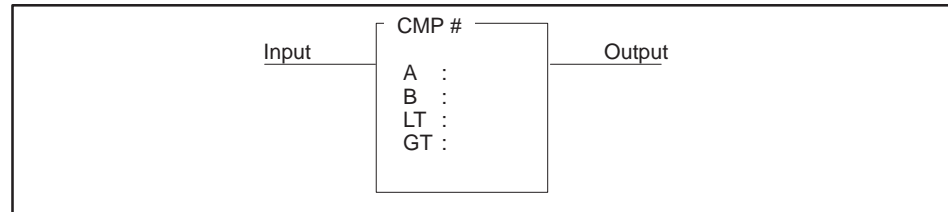
CBD

Refer to Section E.13 for an application example of the CDB.

## 6.10 Compare

### CMP Description

The Compare instruction (Figure 6-11) compares a signed integer value in memory location A with a signed integer value in memory location B.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A B	Any readable word	Memory locations of the values being compared.
LT	C, Y, B or blank	Coil or relay to be turned on if $A < B$ . If you do not want any contacts turned on, designate this coil as C0 or leave it blank.
GT	C, Y, B or blank	Coil or relay to be turned on if $A > B$ . If you do not want any contacts turned on, designate this coil as C0 or leave it blank.

Figure 6-11 CMP Format

### CMP Operation

If the input is on, the value in A is compared to the value in B with the results listed below. A and B do not change as a result of this instruction.

- If  $A < B$ , LT is turned on, GT is turned off, and there is no power flow at the box output.
- If  $A > B$ , GT is turned on, LT is turned off, and there is no power flow at the box output.
- If  $A = B$ , GT and LT are turned off, and the output is turned on.

If the input is off, the GT and LT coils are turned off and there is no power flow at the box output.

---

**NOTE:** The Compare instruction computes power flow based on the equality test. To compute power flow for two conditions (e.g.,  $\geq$ ), additional RLL is required, or you can use the relational contacts.

---

**See Also**

These RLL instructions can also be used for math operations.

ABSV	ADD	DIV	MULT	SQR	SUB
Relational Contact					

6.11 Coils

The various types of RLL coils that are supported are shown in Figure 6-12. Refer to Section 5.1 for a detailed description of their operation.

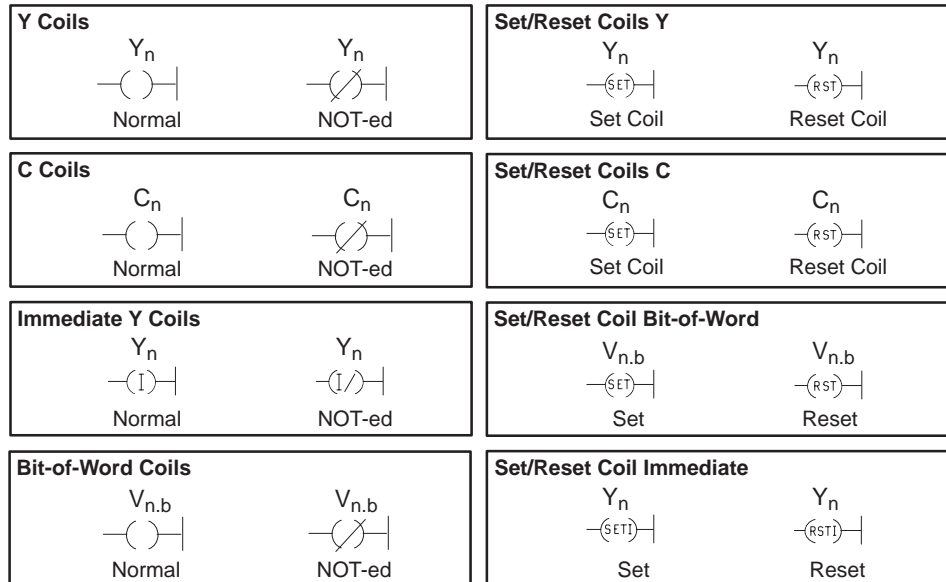


Figure 6-12 Coil Format

See Also

These RLL instructions can also be used for electro-mechanical replacement.

Contacts	CTR	DCAT	DRUM	EDRUM	JMP
MCA	MCR	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			

## 6.12 Contacts

The various types of RLL contacts that are supported are shown in Figure 6-13. Refer to Section 5.1 for a detailed description of their operation.

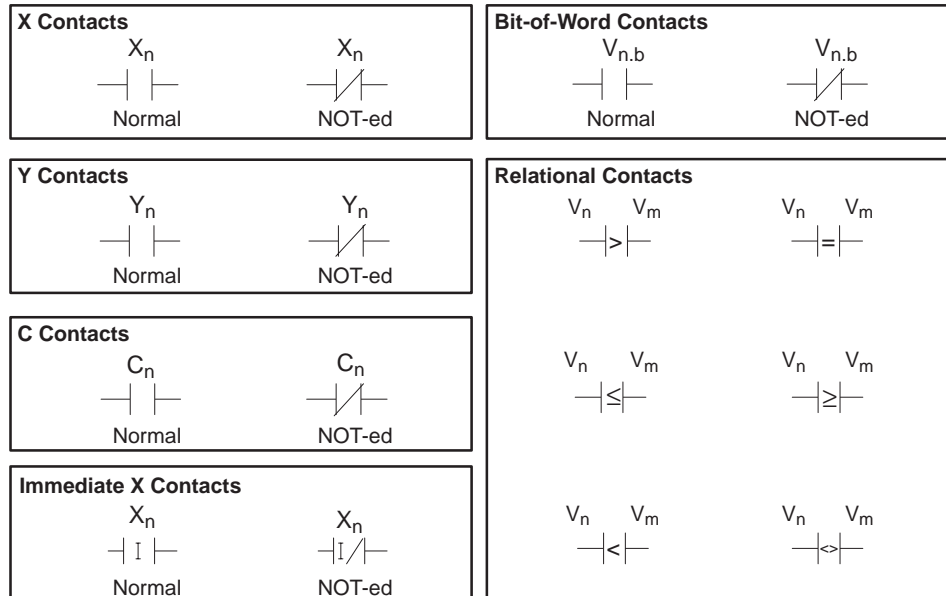


Figure 6-13 Contact Format

See Also

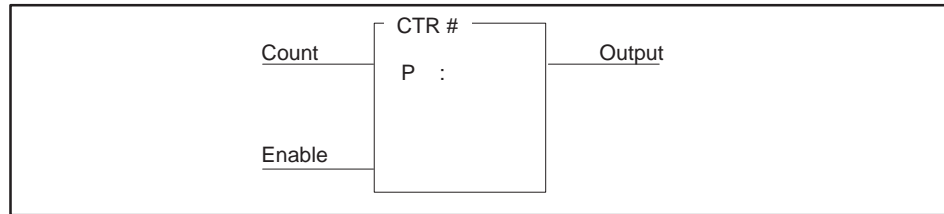
These RLL instructions can also be used for electro-mechanical replacement.

Coils	CTR	DCAT	DRUM	EDRUM	JMP
MCA	MCR	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			



## 6.13 Counter (Up Counter)

**CTR Description** The Counter instruction (Figure 6-14) counts recurring events. The counter output turns on after the counter counts up to a preset number, making it an “up counter.”



Field	Valid Values	Function
#	Varies with configured memory	Instruction reference number. Refer to your controller user manual for the number of counters supported. The assigned instruction number must conform to the requirements of the timer/counter memory discussed on page 4-5 in Section 4.2.
P	0–32767	Preset value of the maximum value (0–32,767) to which the counter counts. The counter does not count events beyond the preset value.

Figure 6-14 CTR Format

**CTR Operation** When the Enable is on:

- The counter is incremented by one each time the Count input transitions from off to on
- The output is turned on when the count reaches the preset value.

When enable is off, the current count is set to zero and the output is turned off.

---

**Using the Counter Variables**

The counter's preset value is stored in TCP# and its current count is stored in TCC#. Current values are retained following loss of power provided that the controller battery backup is enabled. Other RLL instructions can be used to read or write to the counter variables. You can also use an operator interface to read or write to the counter variables. While you are programming the counter, you are given the option of protecting the preset values from changes made with an operator interface.

---

**NOTE:** If you use an operator interface to change TCP, the new TCP value is not changed in the original RLL program. If the RLL presets are ever downloaded, the changes made with the operator interface are replaced by the original values in the RLL program.

---

**See Also**

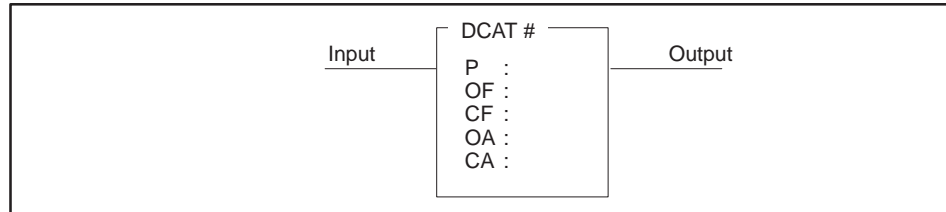
These RLL instructions can also be used for electro-mechanical replacement.

Coils	Contacts	DCAT	DRUM	EDRUM	JMP
MCAT	MCR	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			

## 6.14 Discrete Control Alarm Timer

**DCAT Description** The Discrete Control Alarm Timer (Figure 6-15) is designed for use with a single input, double feedback device. The input to the DCAT box is derived from the logic that determines the state of the device. The output of the DCAT box controls the device.

You can use the DCAT to replace the several rungs of logic that are required to time the field device's operation and generate alarms in case of failure.



Field	Valid Values	Function
#	Varies with configured memory	Instruction reference number. Range depends on memory configured for timers/counters. The assigned instruction number must conform to the requirements of the timer/counter memory discussed on page 4-5 in Section 4.2.
P	0000.1–3276.7 sec.	Time allowed for the device to open or close
OF	X, Y, C, B	Open Feedback — input from a field device or a control relay that senses when the device being controlled has opened to specified position.
CF	X, Y, C, B	Close Feedback — input from a field device or a control relay that senses when the device being controlled has closed to specified position.
OA	Y, C, B	Open Alarm — control relay or output that turns on if the input to the DCAT is on, and the Open Feedback input does not turn on before the DCAT timer times out.
CA	Y, C, B	Close Alarm — control relay or output that turns on if the input to the DCAT has turned off and the Close Feedback input does not turn on before the DCAT timer times out.

Figure 6-15 DCAT Format

**DCAT State Changes**

The state changes for the DCAT are shown in Table 6-2. The DCAT output always equals the state of the input.

**Table 6-2 DCAT States**

Input Condition 1 = open 0 = close X = do not care	IF			THEN		
	Feedback OF	AND CF	Timer Action	Alarm Status OA	CA	Output
1	0	1	timing	0	0	1
1	0	0	timing	0	0	1
1	1	0	reset <sup>2</sup>	0	0	1
1	0	0	timed out <sup>1</sup>	1	0	1
0	1	0	timing	0	0	0
0	0	0	timing	0	0	0
0	0	1	reset <sup>2</sup>	0	0	0
0	0	0	timed out	0	1	0
X	1	1	X	1	1	<sup>3</sup>
<sup>1</sup> Timed out: timer has timed a full preset value of time without a sensor closing. <sup>2</sup> Reset: timer is at preset value and is not timing. <sup>3</sup> Follows input.						

**NOTE:** The DCAT output and alarms are under the control of the JMP or MCR. Unexpected alarm conditions may occur when the DCAT exists within the zone of control of a JMP or MCR.

## Discrete Control Alarm Timer (continued)

---

DCAT Operation	The DCAT timer times down from the preset value specified in P that is stored in TCP-Memory. The timer current time is stored in TCC-Memory.
Open (Input On)	<p>When the input to the DCAT transitions from off to on, the time delay is set to the preset value defined by P, both alarm outputs OA and CA turn off, and the DCAT output turns on.</p> <p>While the input remains on, the timer counts down until the OF input turns on or the timer times out.</p> <p>If the OF input turns on before the timer times out, the time delay is set to zero and the OA remains off.</p> <p>If OF does not turn on before the timer times out, OA turns on.</p> <p>If OF turns on before the timer times down, but then goes off again while the input is on, OA turns on. The OA turns off if OF then turns on again.</p>
Close (Input Off)	<p>When the input to the DCAT transitions from on to off, the DCAT output turns off, the time delay is set to the preset value defined by P, and both alarm outputs OA and CA turn off.</p> <p>While the input remains off, the timer counts down until the CF input turns on or the timer times out.</p> <p>If the CF input turns on before the timer times out, the time delay is set to zero and the CA remains off.</p> <p>If CF does not turn on before the timer times out, CA turns on.</p> <p>If CF turns on before the timer times out, but then goes off again while the input is off, CA turns on. The CA turns off if CF then turns on again.</p>

---

**NOTE:** If both OF and CF are on simultaneously, both OA and CA turn on.

---

---

**Using the DCAT Variables**

The DCAT preset value is stored in TCP# and the time remaining until time out is stored in TCC#. Other RLL instructions can be used to read or write to these variables. You can also use an operator interface to read or write to the DCAT variables. While you are programming the DCAT, you are given the option of protecting the preset values from changes made with an operator interface.

---

**NOTE:** If you use an operator interface to change TCP, the new TCP value is not changed in the original RLL program. If the RLL presets are ever downloaded, the changes made with the operator interface are replaced by the original values in the RLL program.

---

**See Also**

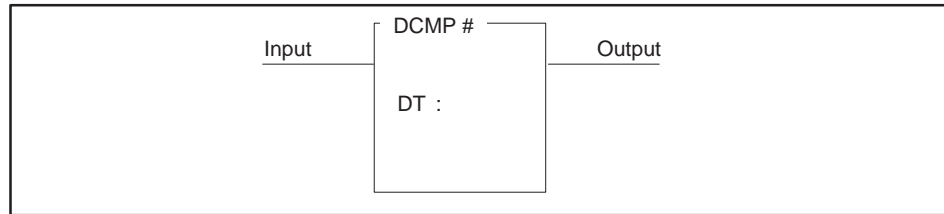
These RLL instructions can also be used for electro-mechanical replacement.

Coils	Contacts	CTR	DRUM	EDRUM	JMP
MCAT	MCR	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			

Refer to Section E.15 for an application example of the DCAT.

## 6.15 Date Compare

**DCMP Description** The Date Compare instruction (Figure 6-16) compares the current date of the real-time clock with the values contained in the designated memory locations.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
DT	V, W, (G, VMS, VMM, 575)	Designates the memory locations containing date to be compared to date in real-time clock.* V(DT) = Year — BCD 0000-0099. V(DT+1) = Month — BCD 0001-0012. V(DT+2) = Day of month — BCD 0001-0031. V(DT+3) = Day of week — BCD 0001-0007. Enter the hexadecimal value of 00FF for any of the fields (year, month, day, etc.) that you want to exclude from the compare operation.

\* BCD values are entered using the HEX data format.

Figure 6-16 DCMP Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

---

**DCMP Operation**      When there is power flow to the input of the DCMP instruction, the current date in the real-time clock is compared to that contained in the designated memory locations. If a match occurs, the instruction's output is turned on.

When the input is off, the comparison is not executed and there is no power flow at the box output.

**See Also**              These RLL instructions can also be used for date/time functions.

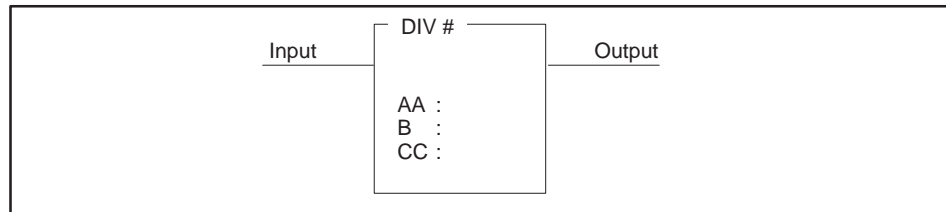
DSET	TCMP	TSET
------	------	------



6.16 Divide

DIV Description

The Divide instruction (Figure 6-17) divides a 32-bit (long word) signed integer stored in memory locations AA and AA + 1, by a 16-bit signed integer in memory location B. The quotient is stored in memory location CC, and the remainder is stored in CC + 1.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
AA	Any readable word	Memory location for the dividend. This is a long word. AA holds the 16 most significant bits, and AA + 1 holds the 16 least significant bits. When a variable is used, the dividend can range from –2,147,483,648 to +2,147,483,647.
	or constant (–32768 to +32767)	Value of the dividend if a constant is used.
B	Any readable word	Memory location for the divisor (one word). When a variable is used, the divisor can range from –32,768 to +32,767, but cannot be zero.
	or constant (–32768 to +32767)	Value of the divisor if a constant is used.
CC	Any writeable word	Memory location for the result. CC holds the quotient (a word); CC+1 holds the remainder (a word). Both quotient and remainder must range from –32,768 to +32,767 to be valid.

Figure 6-17 DIV Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**DIV Operation**

When the input is on, the DIV box is executed. If the input remains on, the operation executes on every scan. The operation of the DIV, that is illustrated in Figure 6-18, follows:

$$[CC \text{ (quotient)}, CC + 1 \text{ (remainder)}] = (AA, AA + 1) \div B$$

If B is non-zero, the division is done and the output turns on. Otherwise, the output turns off, and the contents of CC and CC + 1 do not change.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

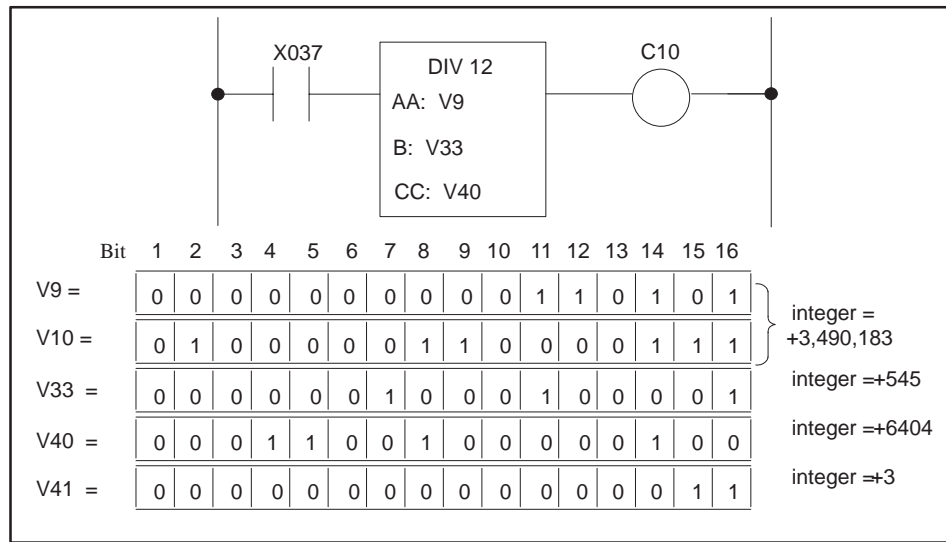


Figure 6-18 Division Example

**See Also**

These RLL instructions can also be used for math operations.

ABS	ADD	CMP	MULT	SQR	SUB
Relational Contact					

## 6.17 Time Driven Drum

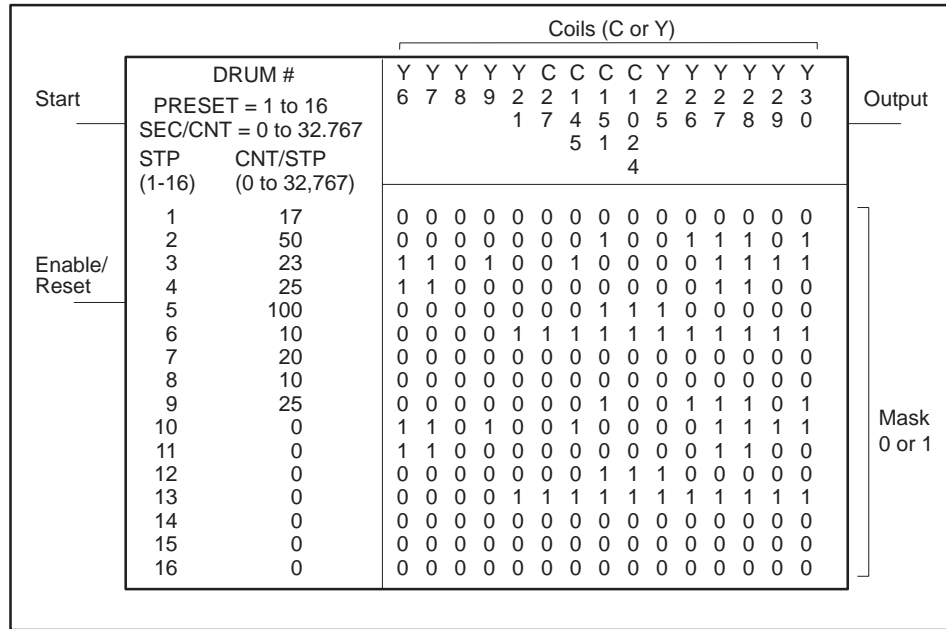
---

**DRUM Description**      The Drum (Figure 6-19) simulates an electro-mechanical stepper switch or drum. It provides 15 output coils and 16 steps that are operated on multiples of the time base set up for the drum. Each step controls all 15 output coils.

**DRUM Operation**      The drum functions as described below.

When the drum begins to run, it starts at the step specified by the Drum Step Preset that is stored in DSP-Memory. The drum current step is stored in DSC-Memory. The counts per step, set in the CNT/STP field, are stored in L-Memory and cannot be changed without re-programming the drum. The current count (counts remaining for a step) is stored in DCC-Memory.

- The drum is enabled when the Enable/Reset input is on.
- When the Enable/Reset is on and the Start input turns on, the drum begins to run. The drum begins at the step specified by DSP and remains at this step until DCC counts down to zero.
- When DCC for a step reaches zero, the drum advances to the next step, and the coils are turned on/off according to the drum mask for the new step. Each 1 in the mask designates that a coil is to be turned on, while each 0 designates that a coil is to be turned off.
- The drum output comes on and remains on after the last programmed step is executed. The last programmed step is the last step with a non-zero CNT/STP value (step 9 in Figure 6-19). The drum remains at the last step until you reset the drum.
- When the Enable/Reset turns off, the drum output turns off, and the drum returns to the step specified in DSP.
- If the Start input turns off but Enable/Reset remains on, the drum remains at the current step (DSC), and DCC holds its current count. All coils maintain the condition specified by the drum mask for this step.
- When the drum is at the Preset step, the output coils follow the states specified by the drum mask for that step, even if the Enable/Reset input is off. Take care to program the mask with a bit pattern that is a safe (home) state for the Preset step.



Field	Valid Values	Function
#	Varies with configured memory	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of the drum memory discussed on page 4-9 in Section 4.2.
PRESET	1-16	Step to which the drum returns when reset.
SEC/CNT	0-32.767 sec.	Time base. Amount of time for one count.
Coils	Y, C, B, or blank	Coils controlled by drum. C0 represents no coil.
STP	1-16	Step number.
CNT/STP	0-32767	Specifies time that drum remains at step. Actual time/step equals CNT/STP × SEC/CNT in seconds.
Mask	0-1	Mask controls coils turned on (1) or off (0).

Figure 6-19 DRUM Format

## Time Driven Drum (continued)

---

### Calculating Counts/Step

Set the Counts/Step for the time that the drum must remain on a step according to one of the following equations.

$$\text{If SEC/CNT is not 0, } \Rightarrow \quad CNT/STP = \frac{\text{step time}}{\text{SEC/CNT}}$$

$$\text{If SEC/CNT is 0, } \Rightarrow \quad CNT/STP = \frac{\text{step time}}{\text{scan time}}$$

For example, if Step 2 is to remain on for 5 seconds and you have set the SEC/CNT at 0.20 seconds, then  $CNT/STP = 25$  as shown.

$$CNT/STP = \frac{5.0}{0.2} \quad CNT/STP = 25$$

### Using DRUM Variables

Other RLL instructions can be used to read or write to the DRUM variables. Use care when programming instructions that can alter or read these variables. You can also use an operator interface to read or write to the DRUM variables.

During its operation, the DRUM uses the count preset value that was stored in L-Memory when the DRUM was programmed. Therefore, a new value for count preset that is written by RLL or by an operator interface has no effect on DRUM operation.

It is possible to read/write data to/from drum memory areas for an unprogrammed drum, using these memory locations like V-Memory. However, if you use TISOFT to display values in DSP or DSC memory, any value not in the range of 1–16 is displayed as 16. By contrast, an APT program can display values that are greater than 16 for these variables.

---

**NOTE:** If you use an operator interface to change drum preset values, the new values are not changed in the original RLL program. If the RLL presets are ever downloaded, the changes made with the operator interface are replaced by the original values in the RLL program.

---

---

**See Also**

These RLL instructions can also be used for electro-mechanical replacement.

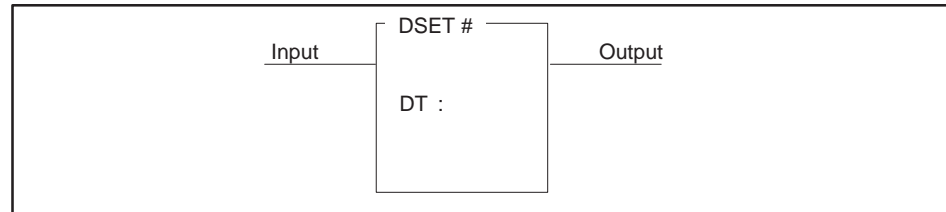
Coils	Contacts	CTR	DCAT	EDRUM	JMP
MCAT	MCR	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			

Refer to Section E.5 for an application example of the DRUM.

## 6.18 Date Set

## DSET Description

The Date Set instruction (Figure 6-20) sets the date portion of the real-time clock to the values contained in designated memory locations.



Field	Valid Values	Function
#	1 to number of one shots.	Instruction reference number. The assigned instruction number must conform to the requirements of the one-shot memory discussed on page 4-7 in Section 4.2.
DT	V, W, (G, VMS, VMM, 575)	Designates the memory locations containing date to be written into the real-time clock*. V(DT) = Year — BCD 0000–0099. V(DT+1) = Month — BCD 0001–0012. V(DT+2) = Day of month — BCD 0001–0031. V(DT+3) = Day of week — BCD 0001–0007.

\* BCD values are entered using the HEX data format.

Figure 6-20 DSET Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

---

**DSET Operation**

When the input to the DSET instruction transitions from off to on, the date portion of the real-time clock is set to the values contained within the three consecutive memory locations designated by DT, and the output is turned on.

---

**NOTE:** The time of day status words (STW141–144 and STW223–225) do not reflect the date change until the next RLL scan.

---

When the input is off, the instruction does not execute and there is no power flow at the box output.

**See Also**

These RLL instructions can also be used for date/time functions.

DCMP	TCMP	TSET
------	------	------



## 6.19 Time/Event Driven Drum

**EDRUM Description** The Time/Event Drum instruction (Figure 6-21) simulates an electro-mechanical stepper switch or drum. The EDRUM can be indexed by a timer only, an event contact only, or a time and event. A jog input enables you to allow either time or an event to advance the drum a step. The EDRUM provides 15 coils and 16 steps that are operated on multiples of the drum time base. Each step controls all 15 output coils.

		Coils (C or Y)															Output	
Event Drum #		Y	Y	Y	Y	Y	C	C	C	C	C	Y	Y	Y	C	Y		
Start	PRESET = 1-16	6	7	8	9	1	1	1	1	5	5	2	2	2	2	3		
	SEC/CNT = 0 to 32.767						0	3	4	5	7	8	6	7	8	9	0	
	STP CNT EVENT (1-16) (0-32,767) (X, Y or C)																	
Jog	1 17 X25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	2 50	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	1	
Enable/ Reset	3 23	1	1	0	1	0	0	1	0	0	0	0	1	1	1	1	1	
	4 25	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	
	5 100 Y45	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	
	6 10	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	
	7 20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	8 10 X34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	9 25 C50	0	0	0	0	0	0	1	0	0	1	1	1	0	1	0	1	
	10 0	1	1	0	1	0	0	1	0	0	0	0	1	1	1	1	1	
	11 0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	
	12 0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	
	13 0 X95	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	
	14 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	15 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	16 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Field	Valid Values	Function
#	Varies with configured memory	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of drum memory discussed on page 4-9 in Section 4.2.
PRESET	1-16	Step to which the drum returns when reset.
SEC/CNT	0-32.767	Time base. Amount of time in seconds for one count.
EVENT	X, Y, C, B	Discrete point that starts countdown of a step and that advances the drum to the next step when count equals zero.
Coils	Y, C, B, or blank	Coils controlled by drum. C0 represents no coil.
STP	1-16	Step number.
CNT	0-32767	Specifies time that drum remains at step. Actual time/step equals CNT × SEC/CNT in seconds.
Mask	0-1	Mask controls coils turned on (1) or off (0).

Figure 6-21 EDRUM Format

---

**EDRUM Operation**

When the drum begins to run, it starts at the step specified by the Drum Step Preset that is stored in DSP-Memory. The drum current step is stored in DSC-Memory. The counts per step, set in the CNT/STP field, is stored in DCP-Memory. The drum current count is stored in DCC-Memory.

- The drum is enabled when the Enable/Reset input is on.
- When the Enable/Reset is on and the Start input turns on, the drum begins to run. The drum begins at the step specified by DSP and advances to the next step depending upon operation of the timer and/or event.
- When the drum advances a step, coils turns on or off according to the mask for the new step. Each 1 in the mask designates that a coil is to turn on, while each 0 designates that a coil is to turn off.
- The drum output turns on, and remains on, after the last programmed step is executed. The last programmed step is the last step having an event programmed or having a non-zero CNT/STP preset value (step 13 in Figure 6-21). The event must be on (if one was programmed for this step) and DCC must be zero. If the event turns off after DCC reaches zero, the drum output remains on and the EDRUM remains at the last programmed step until the drum is reset.
- When the Enable/Reset turns off, the drum output turns off, and the drum returns to the step specified in DSP.
- If the Start input turns off and Enable/Reset remains on, the drum remains at the current step (DSC), and DCC holds its current count. All coils maintain the condition specified by the drum mask.
- When the drum is at the Preset step, the output coils follow the states specified by the drum mask for that step, even if the Enable/Reset input is off. Take care to program the mask with a bit pattern that is a safe (home) state for the Preset step.
- The drum advances to the next step immediately if the Jog input transitions from off to on and the Enable/Reset input is also on.

## Time/Event Driven Drum (continued)

---

### Calculating Counts/Step

Set the Counts for the time that the drum must remain on a step according to one of the following equations.

$$\text{If SEC/CNT is not 0, } \Rightarrow \quad CNT = \frac{\text{step time}}{\text{SEC/CNT}}$$

$$\text{If SEC/CNT is 0, } \Rightarrow \quad CNT = \frac{\text{step time}}{\text{scan time}}$$

For example, if Step 2 is to remain on for 5 seconds and you have set the SEC/CNT at 0.20 seconds, then  $CNT/STP = 25$  as shown.

$$CNT = \frac{5.0}{0.2} \qquad CNT = 25$$

### Timer-triggered Advance Only

For a step having timer operation only, set the CNT preset value (DCP) greater than 0, and do not program a contact or coil in the event field for this step. The drum remains at this step until the DCC counts down to zero. When DCC reaches zero, the drum advances to the next step.

### Event-triggered Advance Only

For a step having event operation only, set the CNT preset value (DCP) for the step to 0, and program a contact or coil in the event field for this step. The drum remains at this step until the contact or coil specified by the event turns on. The drum then advances to the next step.

**Timer and Event-Triggered Advance**

For a step having timer and event operation, set the CNT preset value (DCP) for the step greater than 0 and program a contact or coil in the event field for this step. The following actions occur.

- The timer counts down during every scan in which the event is on. If the event turns off, the DCC holds its current value. DCC resumes counting down when the event turns on again. Timing is the same as for a time-triggered advance.
- When DCC reaches zero, the drum advances to the next step.

**Timer or External Event-triggered Advance**

For a step having timer or external event operation, set the CNT preset value (DCP) for the step greater than 0. Do not program a contact or coil in the event field for this step. Design the RLL program such that an event external to the drum turns on the Jog input. The drum advances to the next step based on either the drum timer or the external event.

**Using EDRUM Variables**

Other RLL instructions can be used to read or write to the EDRUM variables. Use care when programming instructions that can alter or read these variables. You can also use an operator interface to read or write to the EDRUM variables.

It is possible to read/write data to/from drum memory areas for an unprogrammed drum, using these memory locations like V-Memory. However, if you use TISOFT to display values in DSP or DSC memory, any value not in the range of 1–16 is displayed as 16. By contrast, an APT program can display values that are greater than 16 for these variables.

---

**NOTE:** If you use an operator interface to change drum preset values, the new values are not changed in the original RLL program. If the RLL presets are ever downloaded, the changes made with the operator interface are replaced by the original values in the RLL program.

---

**See Also**

These RLL instructions can also be used for electro-mechanical replacement.

Coils	Contacts	CTR	DCAT	DRUM	JMP
MCAT	MCR	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			

Refer to Section E.6 for an application example of the EDRUM.

## 6.20 Unconditional End

---

**END Description** The END instruction (Figure 6-22) unconditionally terminates the scan.

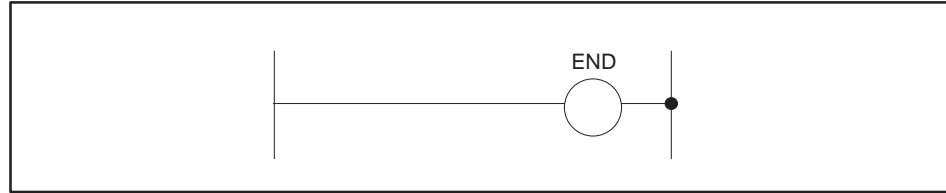


Figure 6-22 END Format

**END Operation** Always terminate your program with the END instruction. When a controller executes an END instruction, the program scan terminates. No instructions occurring after an END executes.

- The controller program scan is always terminated by the unconditional end.
- No other elements can be on the same rung with an END.

If you use an RLL subroutine, place an END instruction between the last rung of the main RLL program and the first rung of the subroutine.

Do not use an END instruction to separate RLL tasks. The TASK instruction indicates that a new RLL task is beginning.

**See Also** This RLL instruction can also be used for terminating the scan.

ENDC

## 6.21 Conditional End

**ENDC Description** The ENDC instruction (Figure 6-23) can terminate the program scan under specific conditions. Since any instructions after an active ENDC instruction are not executed, this instruction can be used to decrease scan time.

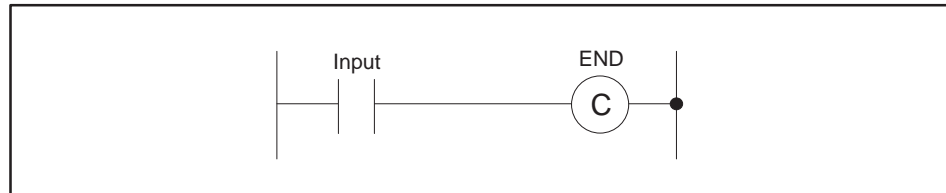


Figure 6-23 ENDC Format

**ENDC Operation** When the ENDC instruction executes, the current program scan terminates. ENDC operates in conjunction with an input and is executed only when there is power flow at the input. When the input is off, the ENDC instruction is not executed, and the program scan is not terminated.

When the ENDC instruction is active, ladder logic following the ENDC is not executed and outputs following the ENDC are frozen. An active ENDC functions as an end statement for MCRs and JMPs that precede it, if it is in their zones of control. Outputs between the MCR or JMP and the ENDC remain under the control of the MCR or JMP.

For an ENDC contained within a SKP zone of control, the ENDC is overridden if the SKP receives power flow.

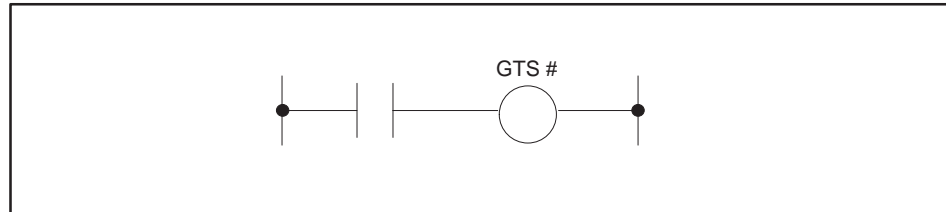
**See Also** This RLL instruction can also be used for terminating the scan.

END

## 6.22 Go To Subroutine

### GTS Description

The GTS instruction (Figure 6-24) enables you to write RLL programs preceded by a subroutine number and call them to be used where needed. The subroutine number is entered after the GTS to designate the subroutine to be executed.



Field	Valid Values	Function
#	1–255	Subroutine reference number.

Figure 6-24 GTS Format

### GTS Operation

When there is power flow to the input of the GTS instruction, the RLL program calls the subroutine indicated by the GTS number. For example, when GTS44 has power flow to the input, execution of RLL jumps to SBR44. If there is no power flow to the input, the GTS instruction does not execute, and RLL program does not jump to the subroutine.

### WARNING

The instructions required to define a subroutine, such as END, RTN, SBR, GTS, and PGTS/PGTSZ, must be entered the way that the controller expects, or else the controller changes from RUN to PROGRAM mode and freezes outputs in their current status, which can cause unexpected controller operation.

Unexpected controller operation can result in death or serious injury to personnel, and/or damage to equipment.

When you do a run-time edit with TISOFT (Rel 4.2 or later), enter all the instructions required to define a subroutine (END, RTN, SBR, GTS or PGTS/PGTSZ) before setting the controller to RUN mode; also, use the TISOFT syntax check function to validate a program before placing the controller in RUN mode. When you do a run-time edit using an earlier release of TISOFT, you must enter the instructions in this order: END, RTN, SBR, GTS or PGTS/PGTSZ.

An example of a subroutine call is shown in Figure 6-25.

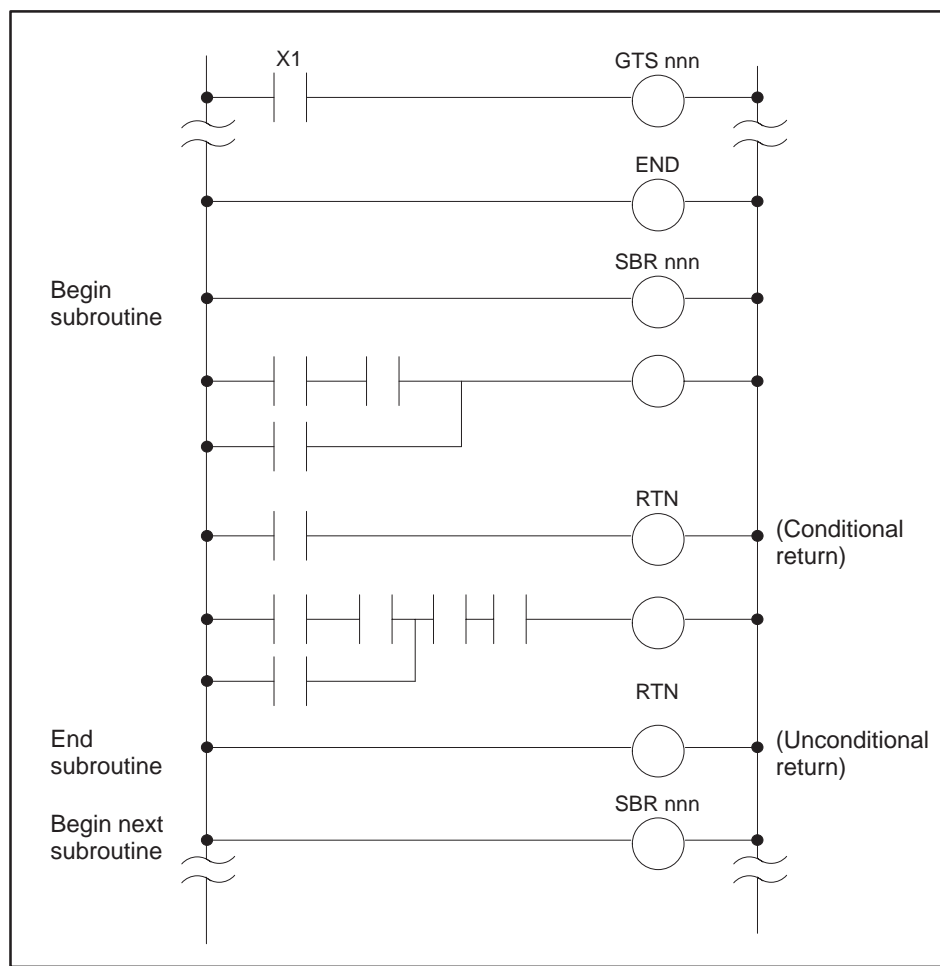


Figure 6-25 Example Call to Subroutine

See Also

These RLL instructions are also used for subroutine operations.

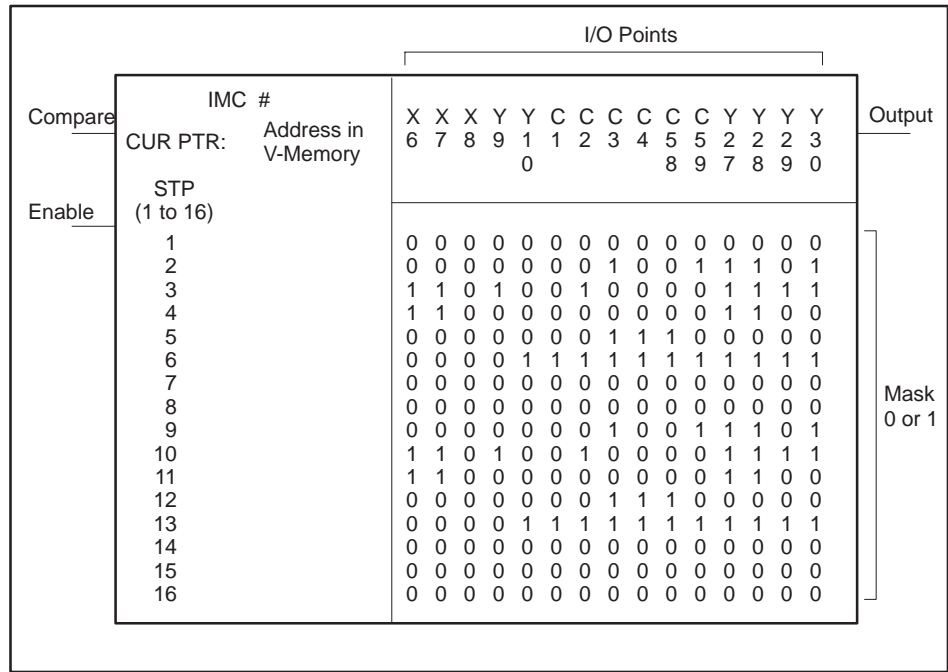
PGTS	PGTSZ	RTN	SBR	SFPGM	XSUB
------	-------	-----	-----	-------	------



## 6.23 Indexed Matrix Compare

### IMC Description

The Indexed Matrix Compare instruction, Figure 6-26, compares a predefined 15-bit mask pattern to the status of up to 15 discrete points. The mask to be compared is selected from a field of up to 16 masks by the step number currently located in CUR PTR. If a match is found, the output turns on.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
CUR PTR	V, W, (G, VMS, VMM, 575)	Memory location of the step number of the mask to be compared to the discrete points.
STP	1-16	Specifies step number of the mask.
I/O Points	X, Y, C, B, or blank	The discrete points to be compared to the mask.

Figure 6-26 IMC Format

**IMC Operation**

The IMC operation is described below.

- If Enable is off, the controller automatically writes 1 to the CUR PTR address regardless of the signal state of the Compare input. The output is turned off.
- If Enable is on and Compare is on, the current status of up to 15 X, Y, or C points is checked against the predefined bit pattern identified by the step number loaded into CUR PTR. If a match is found, the box output turns on. Otherwise, the box output turns off.

---

**NOTE:** If the CUR PTR value is out of range (greater than 16 or less than 1), the controller uses the mask for step 16 to compare to the discrete points.

---

- If Enable is on and Compare is off, the instruction does not execute, and there is no power flow at the box output.

**See Also**

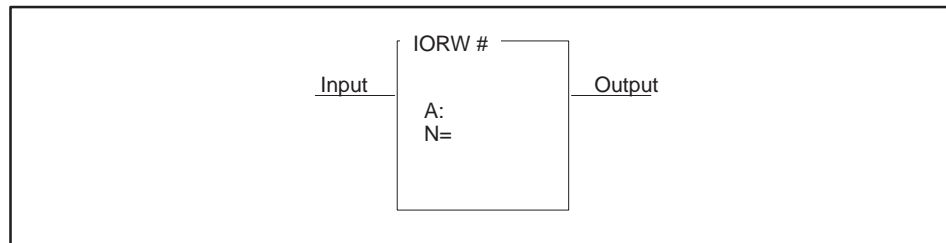
These RLL instructions are also used for bit manipulation.

BITC	BITS	BITP	SMC	WAND	WOR
WROT	WXOR	Bit-of-Word Contact/Coil			

## 6.24 Immediate I/O Read/Write

### IORW Description

The IORW instruction (Figure 6-27) allows you to do an immediate read or write to discrete or word I/O modules on the local base. For inputs, the data transfer is directly from the I/O module(s) into the image register. For outputs, the data transfer is directly from the image register to the I/O modules. Refer to Section 3.3 for more information about using immediate I/O in a program.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	X, Y, WX, WY	Designates I/O starting address. If a discrete point ( $X_n$ or $Y_n$ ), then $n - 1$ must be a multiple of 8.
N	Up to number of points supported by module.	Designates number of points to move. If A is a discrete point ( $X_n$ or $Y_n$ ) then N must be a multiple of 8. All points must reside within the same I/O module.

Figure 6-27 IORW Format

### IORW Operation

When the input is on, the IORW box is executed. If the input remains on, the instruction is executed on every scan.

- The data transfer takes place when the instruction is executed in RLL.  
 For inputs (Xs and WXs), the status of the specified number of points is copied from the I/O module to the image register.  
 For outputs (Ys and WYs), the status of the specified number of points is copied from the image register to the I/O module.

- Output status follows input status, unless an error occurs.

For inputs: when the module is not present or does not match I/O configuration, the specified input points in the image register are cleared to zero and the output turns off.

For outputs: when the module is not present or does not match I/O configuration, the status of the specified output points in the image register is not copied to the I/O module and the output turns off.

If the input is off, the instruction does not execute and there is no power flow at the box output.

**NOTE:** When the IORW copies Y values from the image register to a module, the current state of the Y points in the image register are written to the module. If you want these Ys to be controlled by an MCR or a JMP, the MCR or JMP must be used to control the coils that write to the Ys. The IORW operation is not directly affected by MCRs and JMPs.

**See Also**

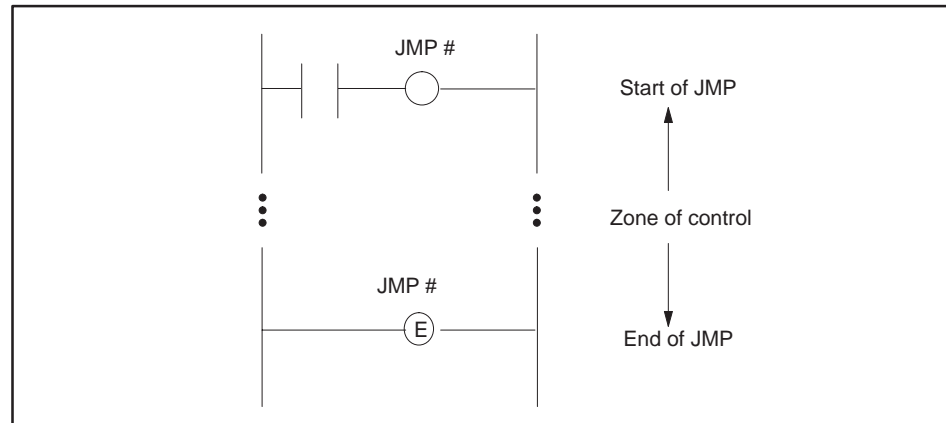
These RLL instructions are also used for immediate I/O applications.

Immediate Contact/Coil	Immediate Set/Reset Coil	TASK
------------------------	--------------------------	------

## 6.25 Jump

### JMP Description

The Jump instruction (Figure 6-28) is used to freeze the values of the discrete image register points of the controlled outputs in the JMP's zone of control. This instruction is often used when duplication of outputs is required and the outputs are controlled by different logic.



Field	Valid Values	Function
#	1–8	Instruction reference number. Numbers can be repeated.

Figure 6-28 JMP Format

### JMP/JMPE Operation

The JMP operates as an output update-enable instruction. The JMP must have power flow, and not be nested within the zone of control of a JMP not having power flow, for ladder logic in the JMP zone of control to change the status of outputs.

- Discrete outputs between a JMP and its corresponding JMPE do not change when the JMP loses power flow.
- JMPE marks the end of the zone of control for the JMP having the same reference number. If you do not use the JMPE, the remainder of the program is placed under the control of the JMP. You can make the JMPE conditional by placing a contact on the same rung as the JMPE.
- When an MCR loses power flow, JMP instructions within the MCR's zone of control are overridden. That is, all outputs in the MCR's zone of control turn off when the MCR loses power flow, even when the outputs are frozen in an ON state by a JMP. This includes rung outputs within the rung, such as those specified within a drum.

Refer to Section 6.55 for information about the action of the JMP in conjunction with the SKP instruction.

In Figure 6-29, a JMP is located on rung A, and its zone of control is terminated by JMPE (End Jump) on rung D.

- When JMP 5 has power flow, the ladder logic within its zone of control, (rungs B and C), is executed normally.
- When JMP 5 does not have power flow, all RLL instructions in the JMP zone of control still execute normally, but outputs are not changed.
- Discrete outputs and control relays contained within an instruction, such as a drum, for example, are also controlled by the JMP. In Figure 6-29, Y6, Y7, Y8, C1, C2, and C3, as well as Y12 and Y451, are frozen when the JMP loses power flow.

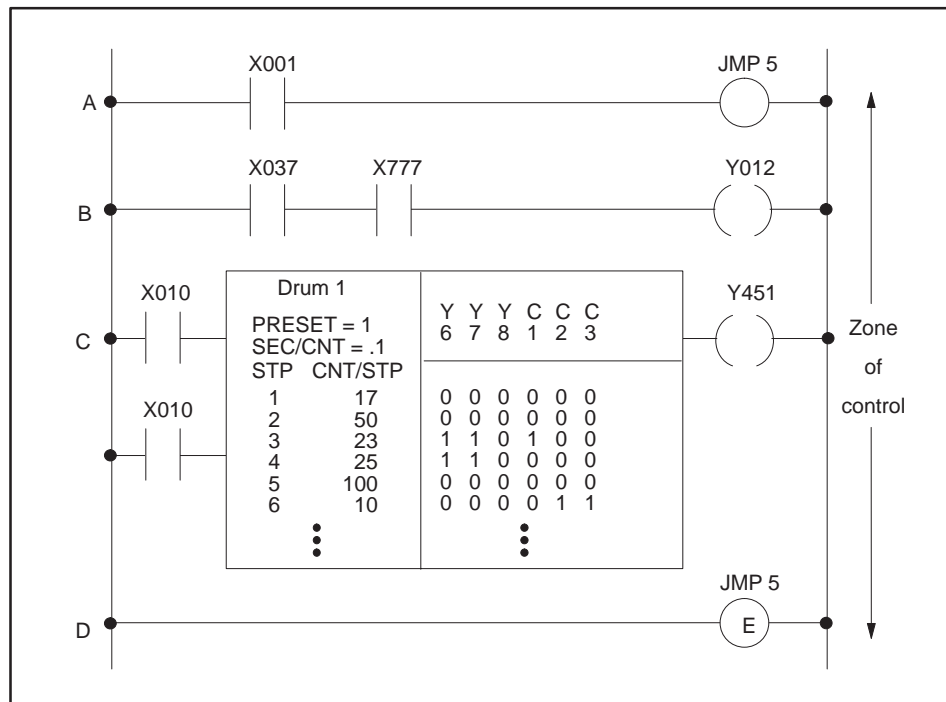


Figure 6-29 Example of JMP Zone of Control

See Also

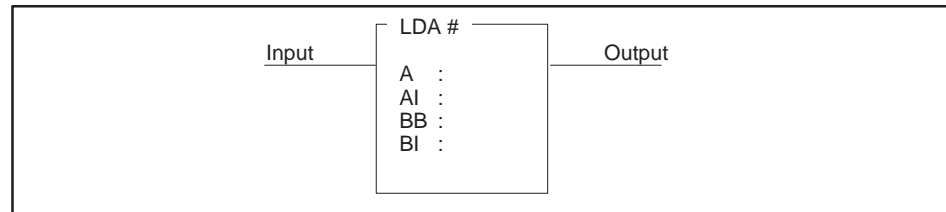
These RLL instructions are also used for electro-mechanical replacement.

Coils	Contacts	CTR	DCAT	DRUM	EDRUM
MCAT	MCR	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			

## 6.26 Load Address

### LDA Description

The Load Address instruction (Figure 6-30) copies the logical address of a memory location into a specified memory location (a long word). Use the LDA as a preparatory statement to the MOVE instruction, when the indirect addressing option is needed.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
A	Any readable word	Specifies the source address. See "Specifying Source" below.
AI	Blank, unsigned constant (0 to 65535) or any readable word	Specifies an index to be applied to the source address. See "Specifying Index for Source" below.
BB	For direct address: any writeable word For indirect address: any readable word	Specifies destination. See "Specifying Destination" below.
BI	Blank, unsigned constant (0 to 65535) or any readable word	Specifies index to be applied to destination address. See "Specifying Index for Destination" below.

Figure 6-30 LDA Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

---

## LDA Operation

When the input is turned on the LDA box executes. If the input remains on, the instruction executes on every scan. The operation of LDA follows and is illustrated in Figure 6-31.

- The address of the memory location specified in A is copied to the destination specified in BB.

 <b>WARNING</b>
--

The address that is copied to the destination is a logical address, not a physical address.

Using this address as a pointer within an external subroutine can cause unpredictable operation by the controller, which could result in death or serious injury to personnel, and/or damage to equipment.

Do not use this address as a pointer within an external subroutine.

- The output is turned on and bit 11 of STW01 is turned off after the instruction executes, unless an error occurs.

When the destination location is not valid, bits 6 and 11 in STW01 turn on, and (if bit 6 of STW01 was off) STW200 is set to a value of 5. The destination contents do not change.

When the input is off, the instruction is not executed and there is no power flow at the box output. In this case bit 11 of STW01 is turned off.



Load Address (continued)

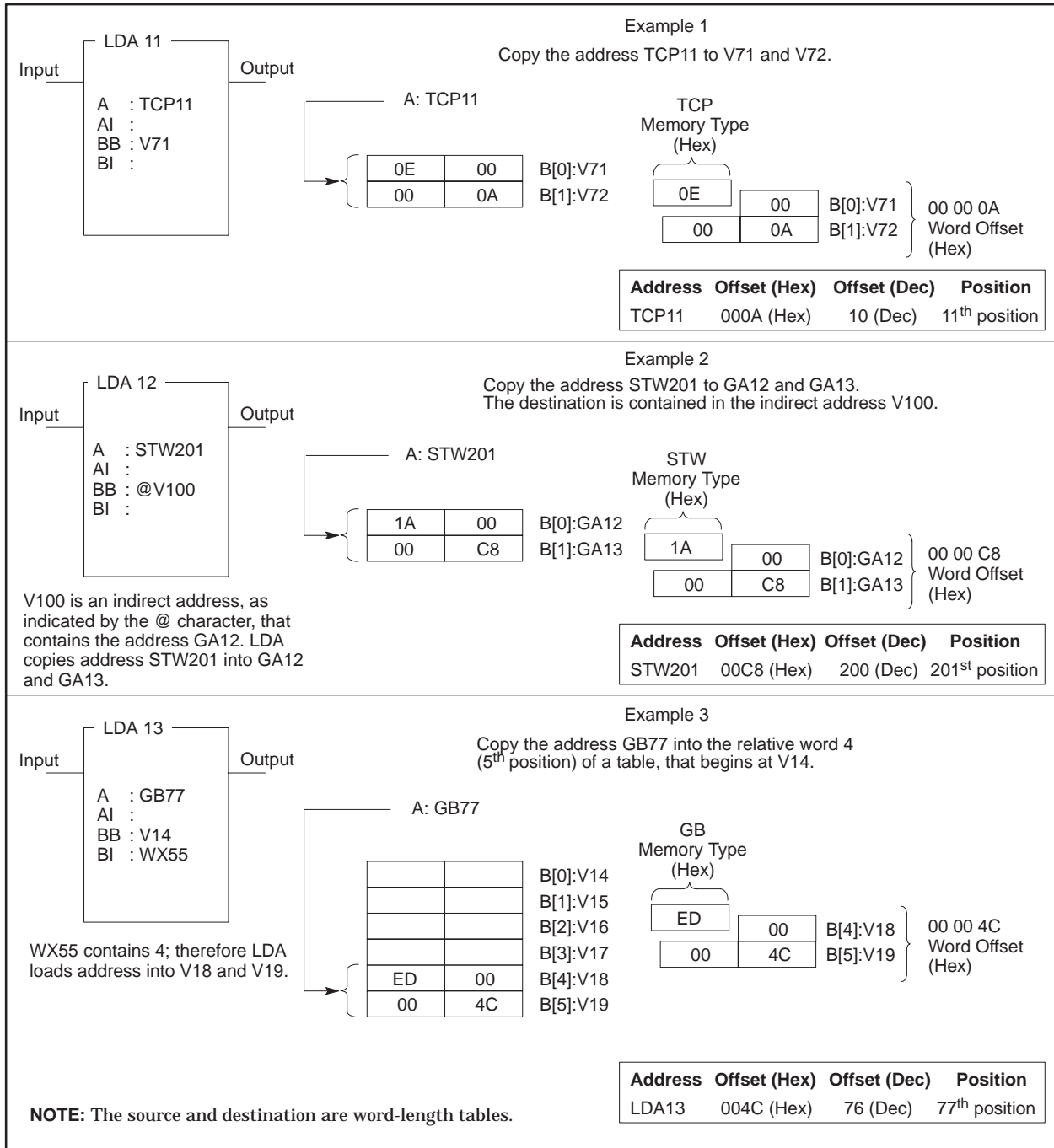


Figure 6-31 Examples of the LDA Instruction

**Specifying Source**

You can specify one of the following elements in A.

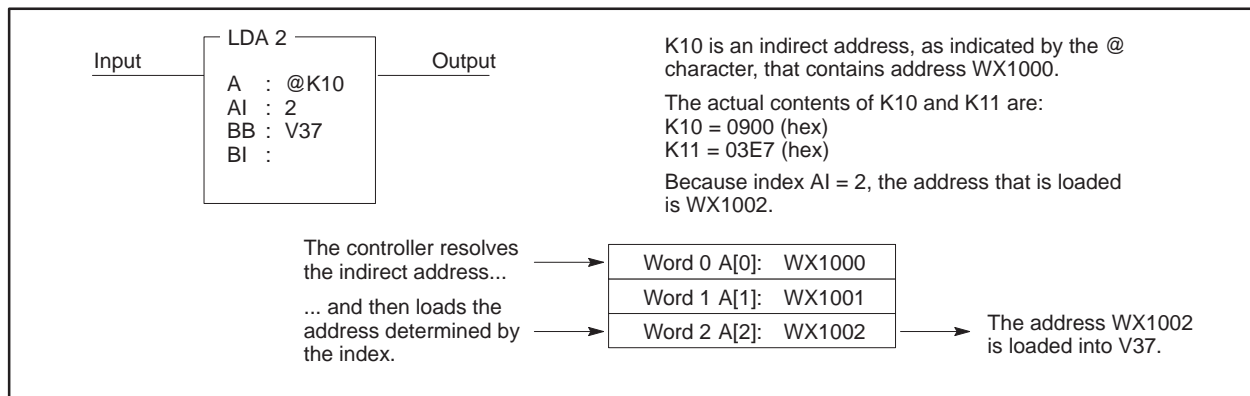
- **Direct address** — Specify any readable word, e.g., V100. LDA copies the logical address for this word into the destination.
- **Indirect address** — Specify any readable word and designate it an indirect address by preceding the address with the @ character, e.g., @V929. The long word at this indirect address must contain another address, and LDA copies this second logical address into the destination.

**Specifying Index for Source**

Use the optional field AI as an index into a table when you want to copy an address that is in a table. AI designates the relative word, in the table referenced by A, the address of which is to be copied. The element at A<sub>0</sub> is the first element in the table. You can specify one of the following in AI.

- **Constant index (range 0 to 65535)** — You can leave AI blank or enter zero and no indexing is done.
- **Variable index** — Specify any readable word. The content of this word is an unsigned integer (0 to 65535) that gives the value of the index.

If an indirect source address is indexed, the controller first resolves the address and then indexes it. See Figure 6-32.



**Figure 6-32 Address/Index Resolution**

## Load Address (continued)

---

### Specifying Destination

You can specify one of the following elements in BB.

- **Direct address** — Specify any writeable word, e.g., V631. LDA copies the logical address specified by A into the long word at this address.
- **Indirect address** — Specify any readable word and designate it an indirect address by preceding the address with the @ character, e.g., @V929. The long word at this indirect address must contain another address, and LDA copies the address specified by A into the memory location specified by this second address. You can enter a readable word, e.g., a K-Memory address, into field BB, but the second address referenced by the address in BB must be a writeable word.

### Specifying Index for Destination

Use the optional field BI as an index into a table when you want to copy an address into a word in a table. BI designates the relative word in a table referenced by BB, into which the source is copied. The element at BB<sub>0</sub> is the first element in the table.

You can specify one of the following in BI.

- **Constant index (range = 0 to 65535)** — You can leave BI blank or enter zero, and no indexing is done.
- **Variable index** — Specify any readable address. The content of this address is an unsigned integer (0 to 65535) that gives the value of the index.

If an indirect destination address is indexed, the controller first resolves the address and then indexes it. See Figure 6-32.

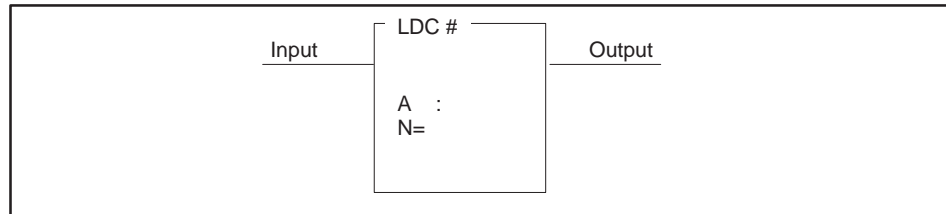
### See Also

These RLL instructions are also used for word moves.

LDC	MIRW	MOVE	MOVW	MWFT	MWI
MWIR	MWTT	SHRW			

## 6.27 Load Data Constant

**LDC Description**      The Load Data Constant instruction (Figure 6-33) loads a (positive integer) constant into the designated memory location.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any writeable word	Memory location where constant is stored.
N	0-32767	Data constant (integer) to be loaded.

Figure 6-33 LDC Format

**LDC Operation**      When the input turns on, the LDC box executes. If the input remains on, the instruction is executed on every scan.

- The data constant designated by N is loaded into the memory location specified by A.
- When the function executes, the output turns on.

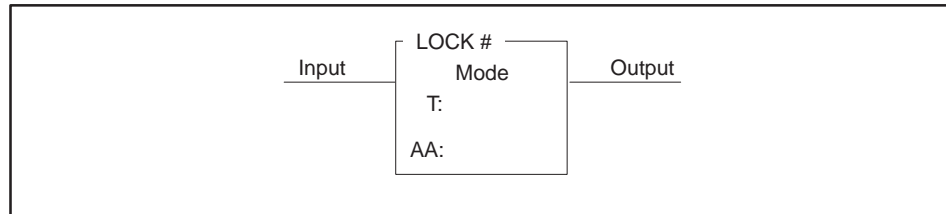
If the input is off, the instruction does not execute, and there is no power flow at the box output.

**See Also**      These RLL instructions are also used for word moves.

LDA	MIRW	MOVE	MOVW	MWFT	MWI
MWIR	MWTT	SHRW			

## 6.28 Lock Memory

**LOCK Description** The LOCK instruction (Figure 6-34) works with the UNLCK instruction to provide a means whereby multiple applications in the 575 system coordinate access to shared resources, generally G-Memory data blocks.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
Mode	Exclusive or Shared	An exclusive lock signals other application programs that the resource is unavailable for reading or writing. A shared lock signals other application programs that the resource locations are available for reading only.
T	0-3276.7	Time in milliseconds for an application program to attempt to acquire control of the lock.
AA	G, VMS, VMM	Memory location (two words) where lock structure is stored. Use the same address for associated UNLCK instruction.

Figure 6-34 LOCK Format

**Acquiring Control of the Lock** The process by which an application program acquires control of a lock is described below.

- You must initialize the lock data structure prior to its first use. Initialization consists of setting both AA and AA+1 to zero.

---

**NOTE:** It is recommended that an application initialize all lock data structures residing in its application space (G-Memory owned by the application) on any scan in which the first scan status word (STW201) indicates a transition from program to run, and on any scan in which the first scan status word indicates a power-up restart or complete restart. If you use this method, be sure to follow these programming practices for the indicated first scan conditions:

- Reset all lock-held states associated with the application.
- Do not attempt to acquire any lock in another application's space.

For this method to operate correctly, all applications sharing a given lock data structure must be mode-locked, and all restarts involving these applications must specify the mode-locked option.

---

- When the input is on, the application attempts to acquire the lock. If the lock is not available, the application continues to attempt acquisition of the lock (the scan is suspended in the process) until the lock is acquired or the specified timeout (T) has expired. A value of 0 for T results in a single attempt to obtain the lock. A value of 3276.7 indicates that the application tries until it obtains the lock or the scan watchdog fatal error occurs.

If the application obtains the lock before the timeout expires, the output turns on and the scan continues.

If the timeout expires before the application obtains the lock, the output turns off and the scan continues.

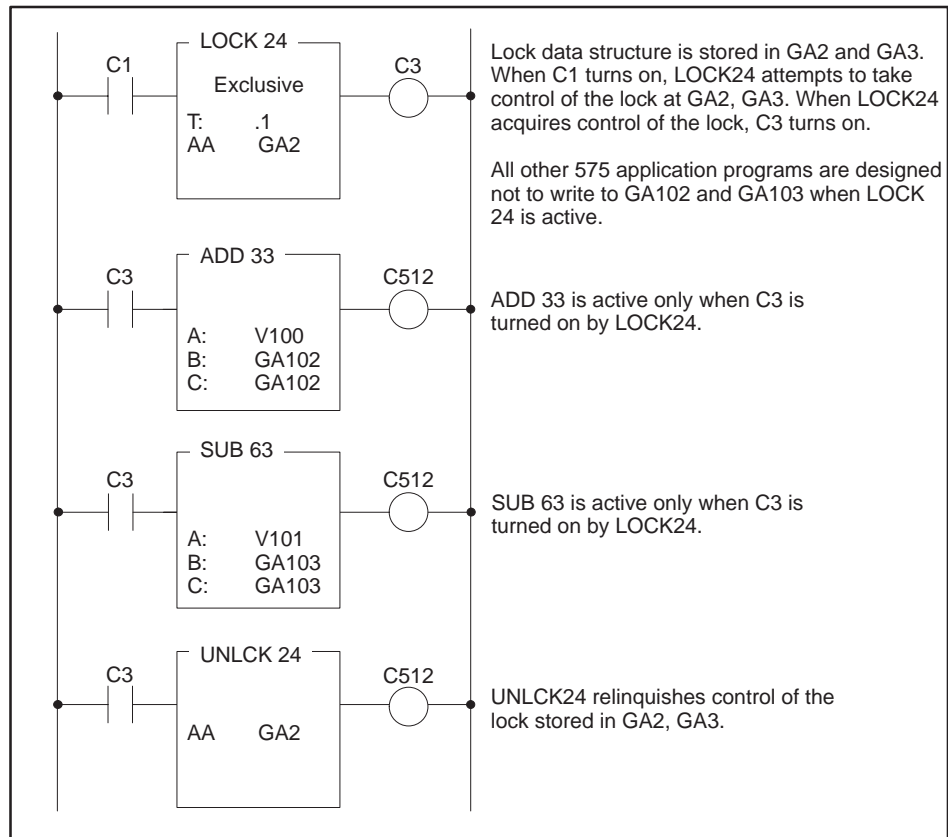
- When an application program attempts to acquire control of the lock, the value in AA (AA+1) is examined. If this value indicates that the lock is free, control of the lock passes to the inquiring application program, the output turns on, and RLL execution continues at the next rung.
- When an application program obtains control of the lock, the LOCK instruction increments the value of a lock/unlock counter. The UNLCK instruction decrements the lock/unlock counter when an application program relinquishes control of a lock. If the counter is not equal to zero at the end of the RLL scan, Bit 6 in STW01 is set to 1 and a value of 3 is written to STW200.
- If the input is off, the instruction does not execute and there is no power flow at the box output.

**Lock Memory (continued)**

**How the Lock Protects Memory**

LOCK does not specify the G-Memory locations that are protected, nor does LOCK actually prevent an application from reading or writing to these G-Memory locations. You must determine which G-Memory locations require lock protection and design your program code not to read from or write to these locations when control of the lock cannot be acquired. Refer to Figure 6-35 for an example of the LOCK instruction operation.

- When you program an exclusive lock, no other application program can acquire control of the lock. Use this capability in programs that update (write to) the shared resource protected by the lock.
- When you program a shared lock, more than one application program can acquire control of the lock. Use this capability in programs that read the shared resource protected by the lock.



**Figure 6-35 Example of the LOCK Instruction**

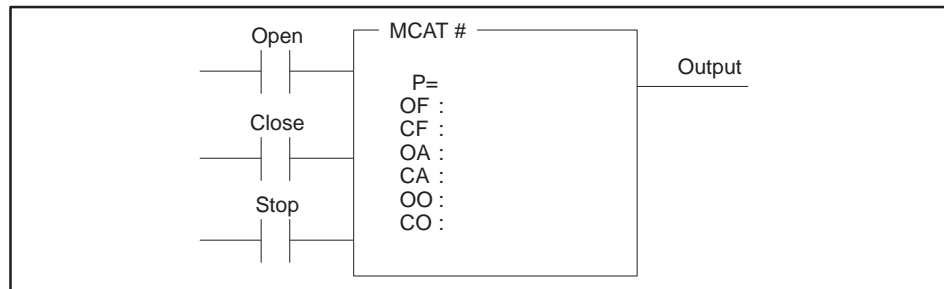
**See Also**

This RLL instruction is also used to coordinate access to shared resources.

**UNLCK**

## 6.29 Motor Control Alarm Timer

**MCAT Description** The MCAT instruction (Figure 6-36) is designed for use with a double input, double feedback device. The MCAT operates similarly to the DCAT instruction. However, the MCAT provides additional functions to operate motor-driven devices that drive in opposite directions. You can use the MCAT to replace several rungs of logic that are required to time the field device's operation and generate alarms in case of failure.



Field	Valid Values	Function
#	Varies with configured memory	Instruction reference number. Range depends on memory configured for timers/counters. The assigned instruction number must conform to the requirements of the timer/counter memory discussed on page 4-5 in Section 4.2.
P	0.1–3276.7	Time allowed for device being controlled to open or close.
OF	X, Y, C, B	Open Feedback — Input from field device that senses when device being controlled has opened.
CF	X, Y, C, B	Close Feedback — Input from field device that senses when device being controlled has closed.
OA	Y, C, B	Open Alarm — Turns on if Open input to the MCAT is on and Open Feedback (OF) input does not turn on before the MCAT timer times out.
CA	Y, C, B	Close Alarm — Turns on if Close input to the MCAT has turned on and Close Feedback (CF) does not turn on before MCAT timer times out.
OO	Y, C, B	Open Output — Opens device being controlled.
CO	Y, C, B	Close Output — Closes device being controlled.

Figure 6-36 MCAT Format



## Motor Control Alarm Timer (continued)

---

### MCAT State Changes

The following state changes for the MCAT are evaluated in the order listed. If a condition is true, the specified actions are executed, and all remaining conditions are not tested or executed.

1. If both OF and CF are on then  
OO turns off, CO turns off,  
OA turns on, CA turns on,  
MCAT output turns off, and  
TCC is set to zero.
2. If Stop input is on and/or both Open input and Close input are on simultaneously, then  
OO turns off, CO turns off,  
OA turns off, CA turns off,  
MCAT output turns on, and  
MCAT timer is disabled.
3. If open was not been commanded, the timer did not time down, and the Open input transitions from off to on while the Close input and the Stop input are both off, then  
OO turns on, CO turns off,  
OA turns off, CA turns off,  
MCAT output turns on, and  
MCAT timer is reset.
4. If open was commanded, the Close and Stop inputs and OF are all off, and the timer did not time down, then  
OO turns on, CO turns off,  
OA turns off, CA turns off,  
MCAT output turns on, and  
MCAT timer times down by the amount of the previous scan.
5. If open was commanded, the Close and Stop inputs are off, and OF is on, then  
OO turns off, CO turns off,  
OA turns off, CA turns off,  
MCAT output turns on, and  
MCAT timer is marked as timed down. This provides for immediate alarming in case the OF input turns off prior to a subsequent close command.

- 
6. If open was commanded, the Close and Stop inputs and OF are all off, and if the timer has timed down, then  
OO turns off,                      CO turns off,  
OA turns on,                      CA turns off, and  
MCAT output is turned off.
  7. If close was commanded, the timer did not time down, and the Close input transitions from off to on while the Open and Stop inputs are both off, then  
OO turns off,                      CO turns on,  
OA turns off,                      CA turns off,  
MCAT output turns on, and  
MCAT timer is reset.
  8. If close was commanded, the Open and Stop inputs and CF are all off, and the timer has not timed down, then  
OO turns off,                      CO turns on,  
OA turns off,                      CA turns off,  
MCAT output turns on, and  
MCAT timer times down by the amount of the previous scan.
  9. If close was commanded, if the Open and Stop inputs are off, and CF is on, then  
OO turns off,                      CO turns off,  
OA turns off,                      CA turns off,  
MCAT output turns on, and  
MCAT timer is marked as timed down. This provides for immediate alarming in case the CF input turns off prior to a subsequent open or stop command.
  10. If close has been commanded, if the Open and Stop inputs and CF are all off, and the timer has timed down, then  
OO turns off,                      CO turns off,  
OA turns off,                      CA turns on, and  
MCAT output turns off.
  11. If none of the above conditions is true, then  
OO turns off,                      CO turns off,  
OA turns off,                      CA turns off, and  
MCAT output turns on.

## Motor Control Alarm Timer (continued)

---

MCAT Operation	The MCAT timer times down from the preset value specified in P that is stored in TCP-Memory. The time current time is stored in TCC-Memory.
Open Input Turns On	<p>When the Open input transitions from off to on and the Close and Stop inputs are both off, the OO turns on and the timer starts. Once triggered, OO remains on independent of the Open input until one of the following events occurs.</p> <ul style="list-style-type: none"><li>• The timer times to 0. The OA turns on, and the OO turns off.</li><li>• The OF turns on while the CF remains off. The OO turns off, and the timer resets to 0. If OF turns on and then turns off, the OA comes on immediately (no time delay) the next time the box is executed.</li><li>• The Stop input turns on. The OO, CO, OA, and CA turn off, and the timer stays where it was when Stop was received. If the Stop input turns off while the Open input is on, then the timer starts at the preset value again—not at the value when the Stop input turned on.</li><li>• The Close input turns on after the Open input turns off. The CO turns on and the timer starts counting at the preset. The OO is turned off.</li></ul>
Close Input Turns On	<p>When the Close input transitions from off to on, while the Open Command and Stop Command Inputs are both off, the CO turns on and the timer starts. CO turns on the motor that closes the valve. Once triggered, the CO remains on, independent of the Close input, until one of the following events occurs.</p> <ul style="list-style-type: none"><li>• The timer times to 0. The CA turns on, and the CO turns off.</li><li>• The CF turns on while the OF remains off. The CO turns off, and the timer resets. If CF turns on and then turns off, the CA comes on immediately (no time delay) the next time the box executes.</li><li>• The Stop input turns on. The OO, CO, OA, and CA turn off.</li><li>• The Open input turns on after the Close input turns off. The OO turns on. The CO turns off.</li></ul>

The condition in which both the Close and Open inputs are on simultaneously is treated as a Stop. The input remaining on when the other turns off is seen as a transition from off to on, and the MCAT enters the appropriate state.

When the Stop input overlaps an Open or Close input, the Stop overrides as long as it is on. When the Stop turns off, the remaining input is seen as a transition from off to on and drives the MCAT to the corresponding state.

The condition in which both Feedback signals are on simultaneously is an error condition. Both Open and Close Alarms turn on, and both Open and Closed Outputs turn off. Removing the conflicting feedback signals does not clear the Open and Close Alarms. One of the MCAT inputs (Open, Close, or Stop) must change state in order to clear the error state.

The box output is always on except during an alarm or error condition.

**Using the MCAT Variables**

You can use other RLL instructions to read or write to the MCAT variables. You can also use an operator interface to read or write to the MCAT variables. While you are programming the MCAT, you are given the option of protecting the preset values from changes made with an operator interface.

---

**NOTE:** If you use an operator interface to change TCP, the new TCP value is not changed in the original RLL program. If the RLL presets are ever downloaded, the changes made with the operator interface are replaced by the original values in the RLL program.

---

**See Also**

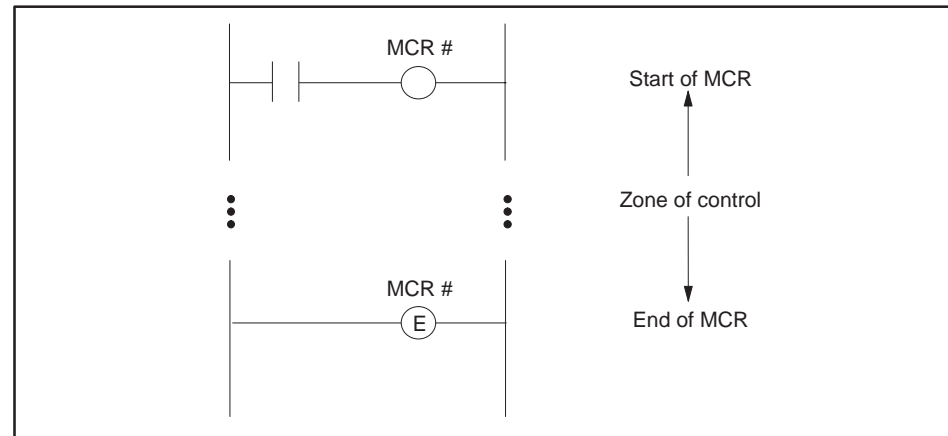
These RLL instructions are also used for electro-mechanical replacement.

Contacts	Coils	CTR	DCAT	DRUM	EDRUM
JMP	MCR	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			

## 6.30 Master Control Relay

### MCR Description

The Master Control Relay (Figure 6-37) is used to turn off blocks of outputs controlled by segments of RLL programs. This is done by clearing the discrete image register points of the controlled outputs to zero.



Field	Valid Values	Function
#	1-8	Instruction reference number. Numbers can be repeated; however, plan logic carefully when nesting MCRs.

Figure 6-37 MCR Format

### MCR/MCRE Operation

The MCR operates as an output-enable instruction.

- The MCR must have power flow, and must not be nested within the zone of control of an MCR not having power flow, for discrete outputs in the MCR zone of control to turn on or stay on.
- The MCR controls the coils and discrete outputs of boxes, e.g., CMP, DCAT, MCAT, drums, etc., in its zone of control.
- MCRE marks the end of the zone of control for the MCR having the same reference number. If you do not use the MCRE, the remainder of the program is placed under the control of the MCR.

You can make the MCRE conditional by placing a contact on the same rung as the MCRE. If you do this, be sure that the contact that controls the conditional MCRE is not controlled by the MCR.

## ⚠ WARNING

Using MCR to replace a hardwired mechanical master control relay that is used for an emergency stop function could jeopardize your control of your process.

Control devices can fail in an unsafe condition that could result in death or serious injury to personnel, and/or damage to equipment.

Never use the MCR to replace a hardwired mechanical master control relay used for an emergency stop function.

Although the MCR controls the coils and discrete outputs of box instructions within its zone of control, it does not control the power rail. Therefore, box instructions continue to operate normally. In order to disable a box, use an MCR-controlled coil output as a normal contact on the same rung that contains the box. See Figure 6-38.

In Figure 6-38 the ADD is controlled by contact C2 when MCR2 is on. When MCR2 is off, the ADD does not execute, regardless of the state of C2.

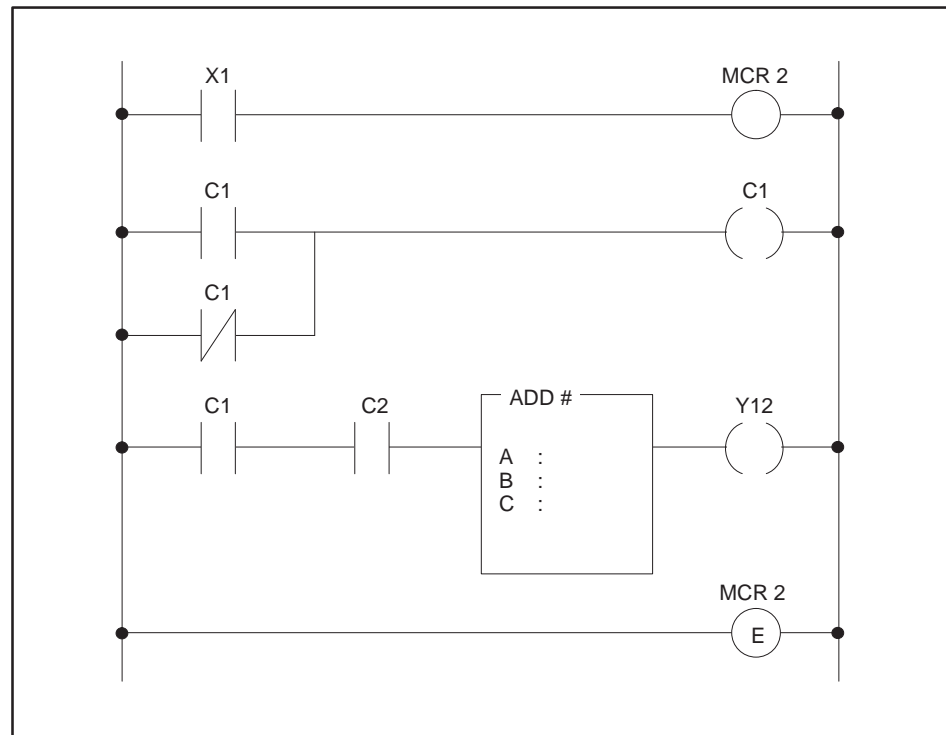


Figure 6-38 Example of MCR Control of a Box

Refer to Section 6.25 and Section 6.55 for information about the action of the MCR in conjunction with the JMP and SKP instructions.

## Master Control Relay (continued)

In Figure 6-39, an MCR is located on rung A, and its zone of control is terminated by the End Master Control Relay MCRE on rung D.

- When MCR2 has power flow, the ladder logic within its zone of control, (rungs B and C), executes normally.
- When MCR2 does not have power flow, all RLL instructions still execute normally, but outputs are turned off.
- Any Ys and Cs contained within an instruction, e.g., a drum, also turn off. In Figure 6-39, when the MCR2 loses power flow, Y6, Y7, Y8, C1, C2, and C3, as well as Y12 and Y451, turn off.

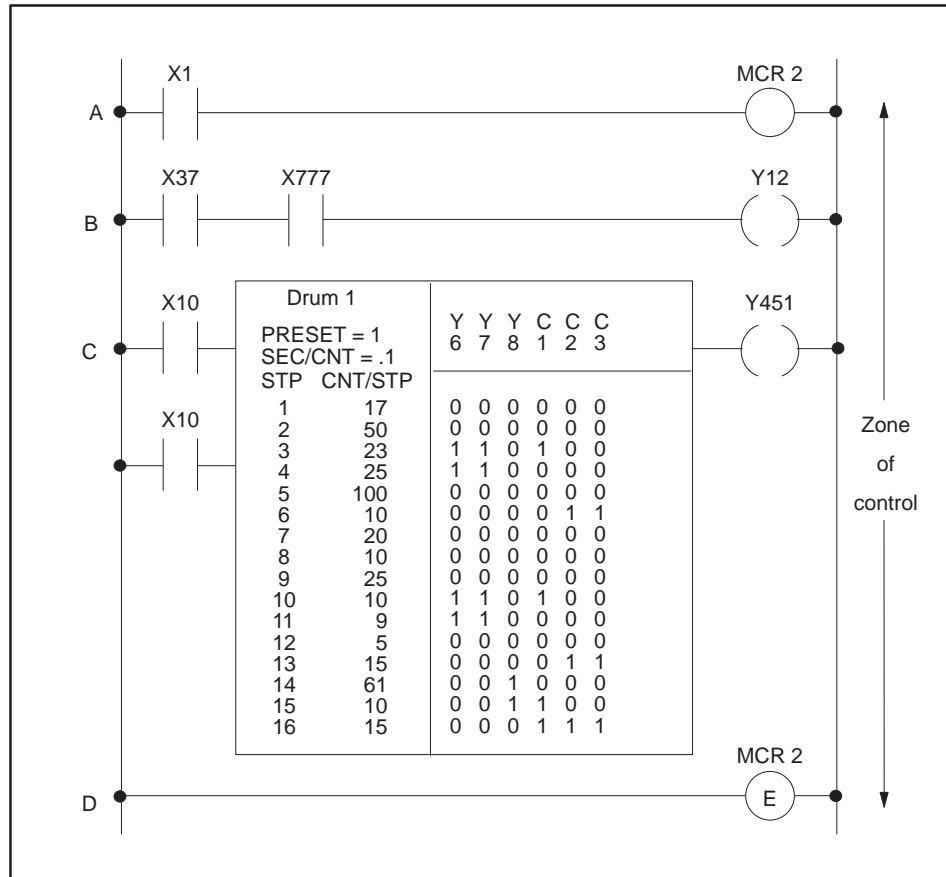


Figure 6-39 Example of the MCR Zone of Control

---

**NOTE:** If a SET or a RST coil is within the zone of control of an active MCR (having no power flow), it stops the SET or RST from changing the state of its associated operand. (That is, SET and RST execution is disabled by the active MCR.)

---

**See Also**

These RLL instructions are also used for electro-mechanical replacement.

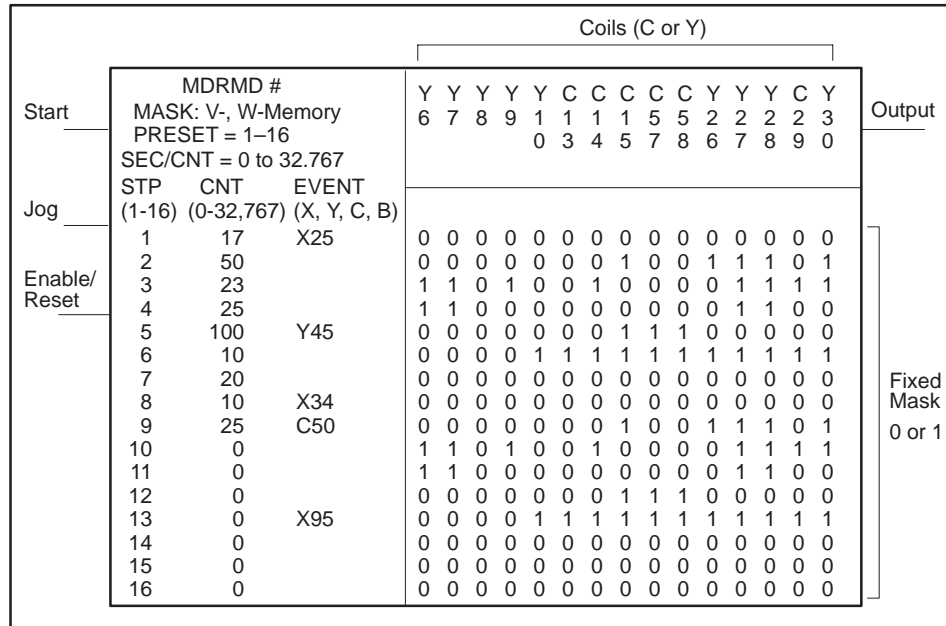
Coils	Contacts	CTR	DCAT	DRUM	EDRUM
JMP	MCAT	MDRMD	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			



### 6.31 Maskable Event Drum, Discrete

**MDRMD Description**

The MDRMD instruction (Figure 6-40) operates similarly to the event drum. The MDRMD, however, is capable of specifying a configurable mask for each step, that allows selection of the coils to be under the control of the fixed mask in each MDRMD step.



Field	Valid Values	Function
#	Varies with configured memory	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of the drum memory discussed on page 4-9 in Section 4.2.
MASK	V, W, (G, VMS, VMM, 575)	First word of a 16-word table that contains the configurable mask output patterns.
PRESET	1-16	Step to which the drum returns when reset.
SEC/CNT	0-32.767	Time base. Amount of time in seconds for one count.
EVENT	X, Y, C, B	Discrete point that starts countdown of a step and that advances the drum to the next step when count equals zero.
Coils	Y, C, B, blank	Coils controlled by drum. C0 represents no coil.
STP	1-16	Step number.
CNT	0-32767	Specifies time that drum remains at step. Actual time/step equals CNT × SEC/CNT in seconds.
Mask	0-1	Mask turns coils on (1) or off (0) according to bit pattern in configurable mask.

Figure 6-40 MDRMD Format

---

**MDRMD Operation**

When the drum begins to run, it starts at the step specified by the Drum Step Preset that is stored in DSP-Memory. The current step is stored in DSC-Memory. The counts per step, set in the CNT field, is stored in DCP-Memory. The current count is stored in DCC-Memory.

- The drum is enabled when the Enable/Reset input is on.
- When the Enable/Reset is on and the Start input turns on, the drum begins to run. The drum begins at the step specified by DSP and advances to the next step based on operation of the timer and/or event.
- When the drum advances a step, coils turn on/off according to the fixed mask and the current bit pattern in the configurable mask.
- The drum output comes on, and remains on, after the last programmed step is executed. The last programmed step is the last step having an event programmed or having a non-zero CNT preset value (step 13 in Figure 6-40). The event must be on (if one was programmed for this step) and DCC must be zero. If the event goes off after DCC reaches zero, the drum output remains on and the MDRMD remains at the last programmed step until the drum is reset.
- When the Enable/Reset turns off, the drum output turns off, and the drum returns to the step specified in DSP.
- If the Start input turns off and Enable/Reset remains on, the drum remains at the current step (DSC), and DCC holds its current count. All coils specified in the configurable mask maintain the condition specified by the fixed mask.
- When the drum is at the Preset step, the coils specified in the configurable mask follow the states specified by the fixed mask for that step, even if the Enable/Reset input is off. Take care to program the mask with a bit pattern that is a safe (home) state for the Preset step.
- The drum advances to the next step immediately if the Jog input transitions from off to on and the Enable/Reset input is also on.

You can use the MDRMD in applications that require a configurable on/off pattern for the drum coils. To do this, specify all ones (1s) for the fixed mask of every programmed step of the MDRMD and precede the MDRMD instruction with the necessary instruction(s) to turn off unconditionally all the MDRMD's coils. The configurable mask table in memory must then contain the on/off patterns that are to be written to the coils for each step.

## Maskable Event Drum, Discrete (continued)

### Defining the Mask

The configurable mask is specified for each step by a memory location in the mask field of the instruction. The configurable mask is located in 16 consecutive memory locations (allocated after entry of the first address). The first location corresponds to step 1 of the drum; the second, to step 2, etc. The mask is defined as being configurable because you can change the mask by writing data to the memory locations.

The configurable mask allows selection of the coils to be controlled by the fixed mask. When a bit of the configurable mask is on (set to 1), the fixed mask controls the corresponding coil. When a bit of the configurable mask is off (set to 0), the corresponding coil is left unchanged by the MDRMD.

The mapping between the configurable mask and the coils is shown below. To match corresponding bits in the mask, coils are numbered from left to right.

Configurable mask word bit position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MDRMD coil #		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Bit 1 of the configurable mask word is unused.

### Calculating Counts/Step

Set the Counts/Step for the time that the drum must remain on a step according to one of the following equations. (See also p. 6-42 for an example.)

$$\text{If SEC/CNT is not 0, } \Rightarrow \quad CNT = \frac{\text{step time}}{\text{SEC/CNT}}$$

$$\text{If SEC/CNT is 0, } \Rightarrow \quad CNT = \frac{\text{step time}}{\text{scan time}}$$

### Timer-triggered Advance Only

For a step having timer operation only, set the CNT preset value (DCP) greater than 0, and do not program a contact or coil in the event field for this step. The drum remains at this step until the DCC counts down to zero. When DCC reaches zero, the drum advances to the next step.

### Event-triggered Advance Only

For a step having event operation only, set the CNT preset value (DCP) for the step equal to 0, and program a contact or coil in the event field for this step. The drum remains at this step until the contact or coil specified by the event turns on. Then the drum then advances to the next step.

**Timer and Event-Triggered Advance**

For a step having timer and event operation, set the CNT preset value (DCP) for the step greater than 0 and program a contact or coil in the event field for this step. The following actions occur.

- The timer counts down during every scan in which the event is on. If the event turns off, the DCC holds its current value. DCC resumes counting down when the event turns on again. Timing is the same as for a time-triggered advance.
- When DCC reaches zero, the drum advances to the next step.

**Timer or External Event-Triggered Advance**

For a step having timer or external event operation, set the CNT preset value (DCP) for the step greater than 0. Do not program a contact or coil in the event field for this step. Design the RLL program such that an event external to the drum turns on the Jog input. The drum advances to the next step based on either the drum timer or the external event.

**Using MDRMD Variables**

Other RLL instructions can be used to read or write to the MDRMD variables. Use care when programming instructions that can alter or read these variables. You can also use an operator interface to read or write to the MDRMD variables.

It is possible to read/write data to/from drum memory areas for an unprogrammed drum, using these memory locations like V-Memory. However, if you use TISOFT to display values in DSP or DSC memory, any value not in the range of 1–16 is displayed as 16. By contrast, an APT program can display values that are greater than 16 for these variables.

---

**NOTE:** If you use an operator interface to change drum preset values, the new values are not changed in the original RLL program. If the RLL presets are ever downloaded, the changes made with the operator interface are replaced by the original values in the RLL program.

---

**See Also**

These RLL instructions are also used for electro-mechanical replacement.

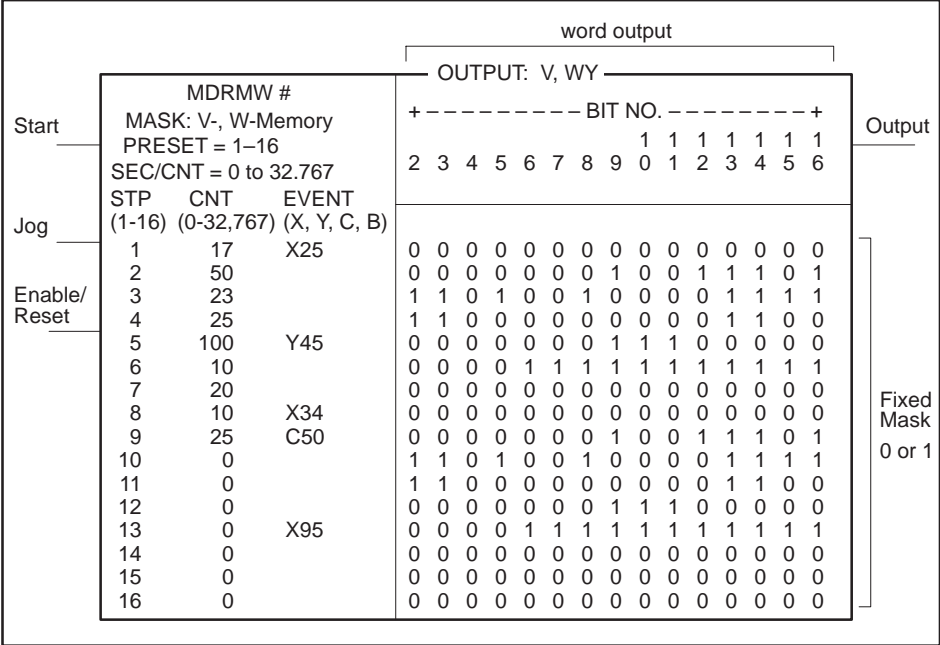
Coils	Contacts	CTR	DCAT	DRUM	EDRUM
JMP	MCAT	MCR	MDRMW	NOT	SHRB
SKP/LBL	TMR	UDC			

## 6.32 Maskable Event Drum, Word

---

### MDRMW Description

The MDRMW instruction (Figure 6-41) operates similarly to the event drum, but the MDRMW writes data to a word instead of to individual coils. The MDRMW also is capable of specifying a configurable mask for each step. This allows the selection of the bits in the word to be changed by the fixed mask in each MDRMW step.



Field	Valid Values	Function
#	Varies with configured memory	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of the drum memory discussed on page 4-9 in Section 4.2.
MASK	V, W, (G, VMS, VMM, 575)	First word of a 16-word table that contains the configurable mask output patterns.
PRESET	1-16	Step to which the drum returns when reset.
SEC/CNT	0-32.767	Time base. Amount of time in seconds for one count.
EVENT	X, Y, C, B	Discrete point that starts countdown of a step and that advances the drum to the next step when count equals zero.
OUTPUT	WY, V, TCP, TCC, G, W, VMS, VMM, DSP, DSC, DCP, DCC	Word location to which the drum writes. Bit 1 is always set to zero.
STP	1-16	Step number.
CNT	0-32767	Specifies time that drum remains at step. Actual time/step equals CNT x SEC/CNT in seconds.
Mask	0-1	Mask gives the value of the bits of the output word.

Figure 6-41 MDRMW Format

## Maskable Event Drum, Word (continued)

---

- MDRMW Operation**      When the drum begins to run, it starts at the step specified by the Drum Step Preset that is stored in DSP-Memory. The current step is stored in DSC-Memory. The counts per step, set in the CNT field, are stored in DCP-Memory. The current count is stored in DCC-Memory.
- The drum is enabled when the Enable/Reset input is on.
  - When the Enable/Reset is on and the Start input turns on, the drum begins to run. The drum begins at the step specified by DSP and advances to the next step based on operation of the timer and/or event.
  - When the drum advances a step, individual bits of the output word turn on/off based on the fixed mask and the current bit pattern in the configurable mask.
  - The drum output comes on, and remains on, after the last programmed step has been executed. The last programmed step is the last step having an event programmed or having a non-zero CNT preset value (step 13 in Figure 6-41). The event must be on (if one was programmed for this step) and DCC must be zero. If the event goes off after DCC reaches zero, the drum output remains on and the MDRMW remains at the last programmed step until the drum is reset.
  - When the Enable/Reset turns off, the drum output turn off, and the drum returns to the step specified in DSP.
  - If the Start input turns off and Enable/Reset remains on, the drum remains at the current step (DSC), and DCC holds its current count. All bits specified in the configurable mask maintain the condition specified by the fixed mask.
  - When the drum is at the Preset step, the bits specified in the configurable mask follow the states specified by the fixed mask for that step, even if the Enable/Reset input is off. Take care to program the mask with a bit pattern that is a safe (home) state for the Preset step.
  - The drum advances to the next step immediately if the Jog input transitions from off to on and the Enable/Reset input is also on.

## Defining the Mask

The configurable mask is specified for each step by a memory location in the mask field of the instruction. The configurable mask is located in 16 consecutive memory locations (allocated after entry of the first address). The first location corresponds to step 1 of the drum; the second, to step 2, etc. The mask is defined as being configurable because you can change the mask by writing data to the memory locations.

The configurable mask allows selection of the individual bits in the output word that are set/cleared by the fixed mask. When a bit of the configurable mask is on (set to 1), the fixed mask sets/clears the corresponding bit. When a bit of the configurable mask is off (set to 0), the corresponding bit is left unchanged by the MDRMW.

The mapping between the configurable mask and the individual bits in the output word is shown below.

Configurable mask word bit position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Output word bit #		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Bit 1 of the configurable mask word is not used.  
Bit 1 of the output word is not used and is always equal to zero.

## Calculating Counts/Step

Set the Counts/Step for the time that the drum must remain on a step according to one of the following equations. (See also p. 6-42 for an example.)

$$\text{If } SEC/CNT \text{ is not } 0, \Rightarrow CNT = \frac{\text{step time}}{SEC/CNT}$$

$$\text{If } SEC/CNT \text{ is } 0, \Rightarrow CNT = \frac{\text{step time}}{\text{scan time}}$$



## Maskable Event Drum, Word (continued)

---

Timer-triggered Advance Only	For a step having timer operation only, set the CNT preset value (DCP) greater than 0, and do not program a contact or coil in the event field for this step. The drum remains at this step until the DCC counts down to zero. When DCC reaches zero, the drum advances to the next step.
Event-triggered Advance Only	For a step having event operation only, set the CNT preset value (DCP) for the step equal to 0, and program a contact or coil in the event field for this step. The drum remains at this step until the contact or coil specified by the event turns on. The drum then advances to the next step.
Timer and Event-Triggered Advance	<p>For a step having timer and event operation, set the CNT preset value (DCP) for the step greater than 0 and program a contact or coil in the event field for this step. The following actions occur.</p> <ul style="list-style-type: none"><li>• The timer counts down during every scan in which the event is on. If the event turns off, the DCC holds its current value. DCC resumes counting down when the event turns on again. Timing is the same as for a time-triggered advance.</li><li>• When DCC reaches zero, the drum advances to the next step.</li></ul>
Timer or External Event-triggered Advance	For a step having timer or external event operation, set the CNT preset value (DCP) for the step greater than 0. Do not program a contact or coil in the event field for this step. Design the RLL program so that an event external to the drum turns on the Jog input. The drum advances to the next step based on either the drum timer or the external event.

---

**Using MDRMD Variables**

Other RLL instructions can be used to read or write to the MDRMW variables. Use care programming instructions that can alter or read these variables. You can also use an operator interface to read from or write to the MDRMW variables.

It is possible to read/write data from/to drum memory areas for an unprogrammed drum, using these memory locations like V-Memory. However, if you use TISOFT to display values in DSP or DSC memory, any value not in the range of 1–16 is displayed as 16. By contrast, an APT program can display values that are greater than 16 for these variables.

---

**NOTE:** If you use an operator interface to change drum preset values, the new values are not changed in the original RLL program. If the RLL presets are ever downloaded, the changes made with the operator interface are replaced by the original values in the RLL program.

---

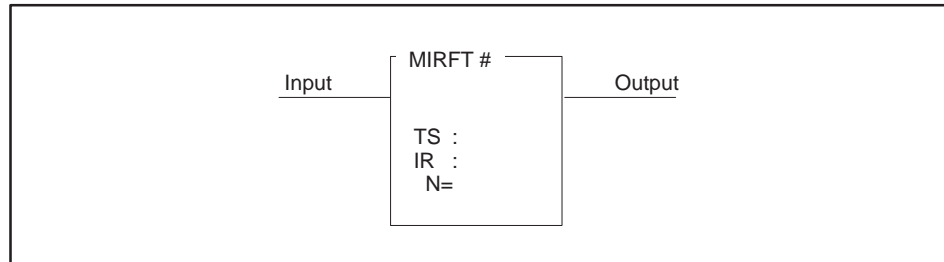
**See Also**

These RLL instructions are also used for electro-mechanical replacement.

Coils	Contacts	CTR	DCAT	DRUM	EDRUM
JMP	MCAT	MCR	MDRMD	NOT	SHRB
SKP/LBL	TMR	UDC			

### 6.33 Move Image Register from Table

**MIRFT Description** The Move Image Register from Table instruction (Figure 6-42) allows you to copy information into the control relays or the discrete image register from a table of consecutive word locations.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
TS	Any readable word	Starting address of source table.
IR	X, Y, C, B	Starting address of the control relays or the discrete image register. Must begin on an 8-point boundary (1, 9, 17, etc.)
N	1–256	Length of table in words.

Figure 6-42 MIRFT Format

**NOTE:** If you plan to use this instruction in a subroutine (using B-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**MIRFT Operation** When the input is on, the MIRFT box executes. If the input remains on, the operation executes every scan. The operation of the MIRFT follows and is illustrated in Figure 6-43.

- The values of up to 256 (N) words (16–4096 bits) are copied, starting at the memory location specified by TS.

The copy is placed in the control relays or the discrete image register. The LSB of the first word is copied into the point specified by IR.

The beginning point in the control relays or the discrete image register must be on an eight-point boundary (1, 9, 17, etc.).

- All words are copied into the control relays or the image register on each scan.
- The output turns on when the instruction executes.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

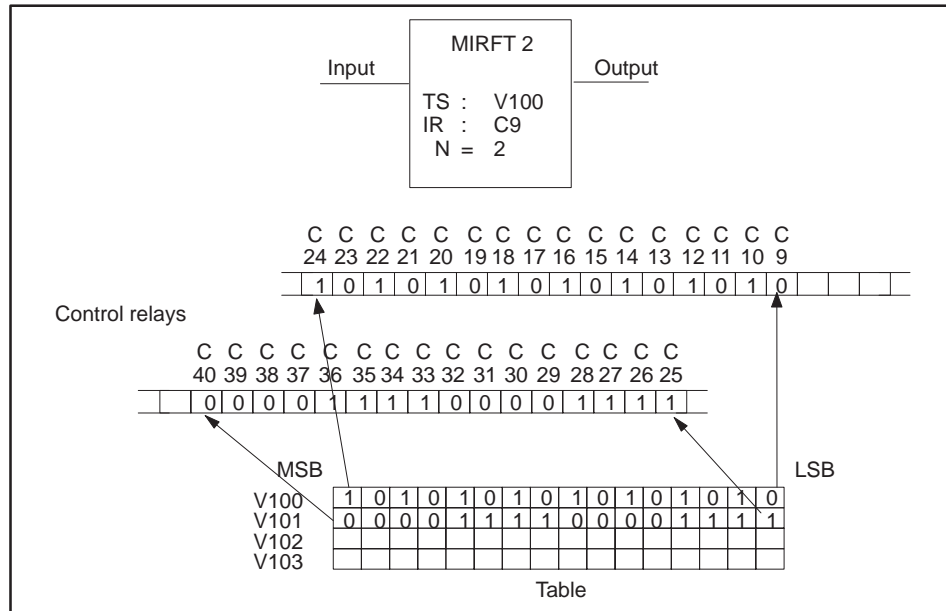


Figure 6-43 Example of MIRFT Operation

See Also

These RLL instructions are also used for table operations.

MIRTT	STFE	STFN	TAND	TCPL	TOR
TTOW	TXOR	WTOT	WTTA	WTTO	WTTXO

## 6.34 Move Image Register to Table

### MIRTT Description

The Move Image Register to Table instruction (Figure 6-44) allows you to copy information from the control relays or the discrete image register to a table of consecutive word locations.

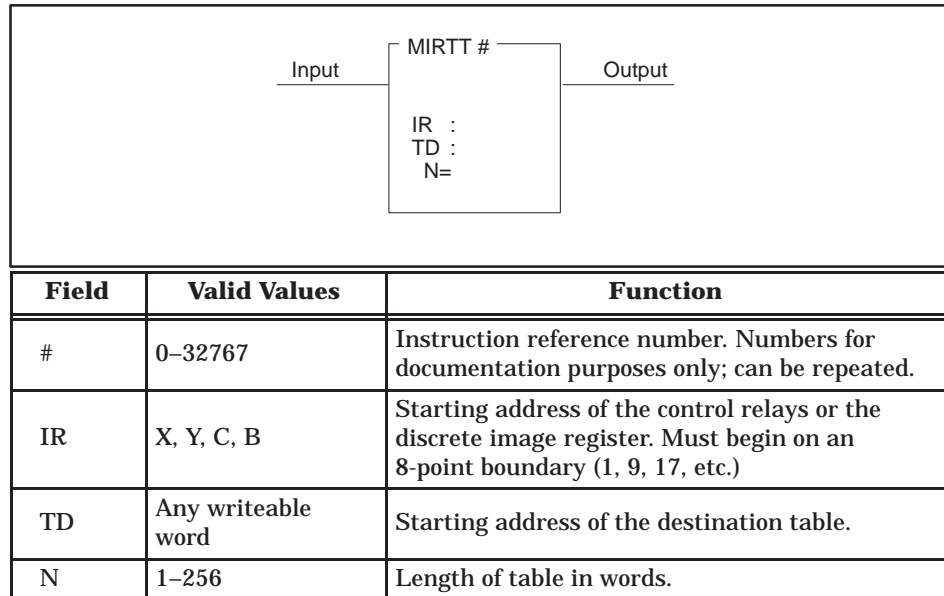


Figure 6-44 MIRTT Format

**NOTE:** If you plan to use this instruction in a subroutine (using B-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

### MIRTT Operation

When the input is on, the MIRTT box executes. If the input remains on, the operation executes on every scan. The operation of the MIRTT follows and is illustrated in Figure 6-45.

- The On/Off state of up to 4096 bits (256 words × 16 bits) is copied from the control relays or the discrete image register, starting at the bit address specified by IR.

The starting point must be on an 8-point boundary (1, 9, 17, etc.). Bits are copied in groups of 16.

The copy begins with the lowest numbered bit address and is placed into word locations, beginning with the LSB of the word specified by TD.

- All bits are copied into the word locations each scan. There must be a sufficient number of discrete points to copy all bits into the table of N words.
- The output turns on when the instruction executes.

If the input is off, the instruction does not execute and there is no power flow at the box output.

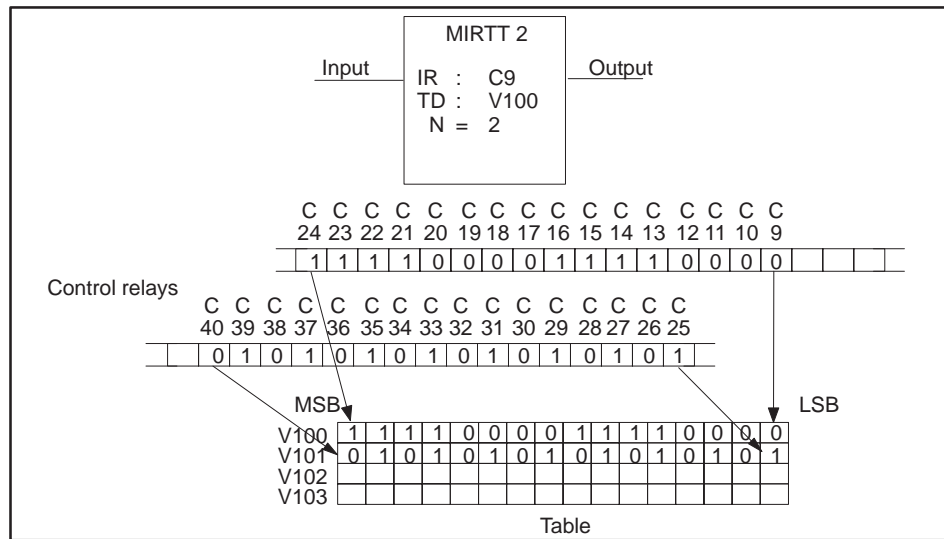


Figure 6-45 Example of MIRT Operation

See Also

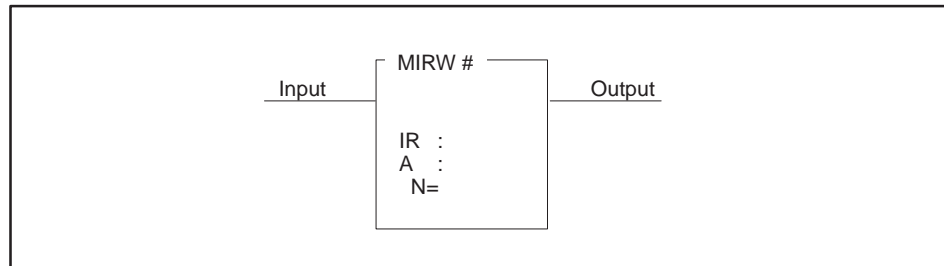
These RLL instructions are also used for table operations.

MIRFT	STFE	STFN	TAND	TCPL	TOR
TTOW	TXOR	WTOT	WTTA	WTTO	WTTXO

## 6.35 Move Image Register to Word

### MIRW Description

The Move Image Register to Word instruction (Figure 6-46) copies a specified number of bits from the discrete image register or the control relay memory locations to a designated word memory location. Up to 16 bits are copied in a single scan.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
IR	X, Y, C, B	Starting address of the control relays or the discrete image register bits to be copied.
A	Any writeable word	Specifies word memory location to which bits are copied.
N	1–16	Number of bits to be copied.

Figure 6-46 MIRW Format

**NOTE:** If you plan to use this instruction in a subroutine (using B-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

### MIRW Operation

When the input is on, the MIRW box executes. If the input remains on, the operation executes on every scan. The operation of the MIRW box follows and is illustrated in Figure 6-47.

- Up to 16 bits (N) are copied, beginning with the lowest numbered address, that is specified by IR.
- The bits are moved into the word memory location specified by A, beginning with the LSB of the word. If fewer than 16 bits are moved, the remaining bits are set to 0. All bits are copied during a single scan.
- The output turns on when the instruction executes.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

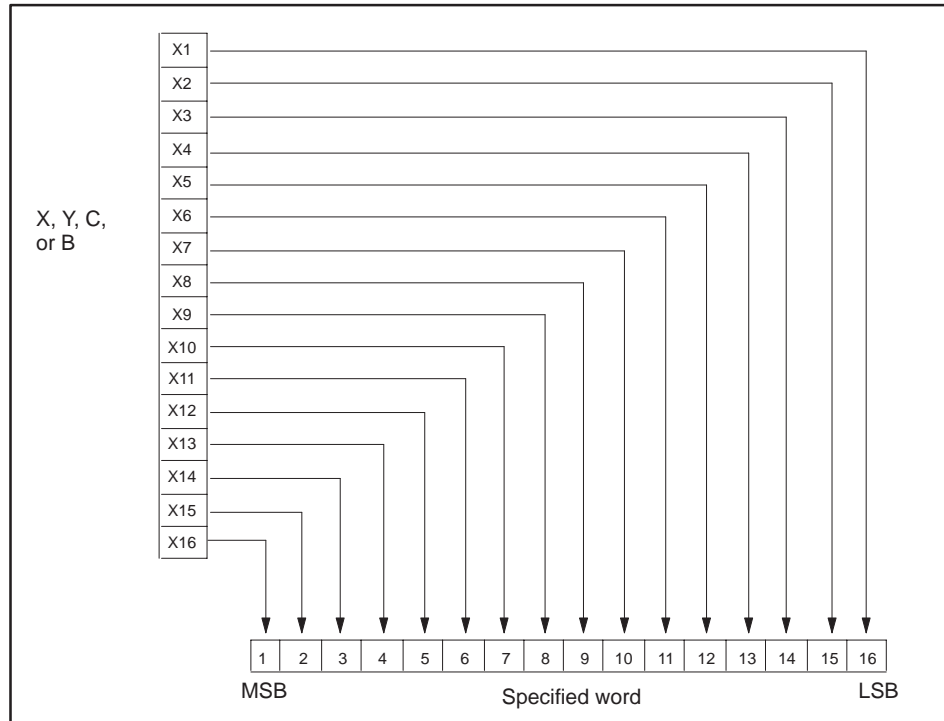


Figure 6-47 Example of MIRW Operation

See Also

These RLL instructions are also used for word moves.

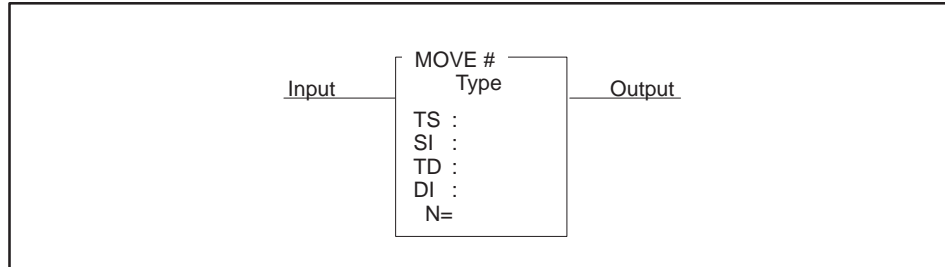
LDA	LDC	MOVE	MOVW	MWFT	MWI
MWIR	MWTT	SHRW			

Refer to Section E.7 for an application example of the MIRW.



6.36 Move Element

**MOVE Description** The Move Element instruction (Figure 6-48) copies data elements (bytes, words, or long words) from a source location to a destination location.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
Type	Byte, Word, or Long Word	Specifies type of the element(s) to be copied: byte = 8 bits, word = 16 bits, long word = 32 bits.
TS	Signed constant (range varies with size of element) or Any readable word	Specifies source element to be copied. Can be a constant, a direct address, or an indirect address (a memory location containing the address of another memory location).
SI	Blank, Unsigned constant (0 to 65535) or Any readable word	Optional index. Designates that the SI <sup>th</sup> element in a table referenced by TS is to be copied. The element at TS is zero (0).
TD	For direct address: Any writeable word For indirect address: Any readable word	Specifies the destination of the copy. TD can be a direct address or an indirect address (a long word containing the address of another memory location).
DI	Blank, Unsigned constant (0 to 65535) or Any readable word	Optional index. Designates the relative element in a table referenced by TD, into which the element is copied. The element at TD is zero (0).
N	Unsigned constant (1 to 32767) or Any readable word	Specifies number of elements to be copied.

Figure 6-48 MOVE Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

---

**MOVE Operation**

When the input is on, the MOVE box executes. If the input remains on, the instruction executes on every scan. The operation of MOVE is described below and illustrated in Figure 6-49.

- The element(s) specified in A are copied to the destination specified in B.
- The output turns on and STW01 bit 11 turns off after the instruction executes, unless an error occurs. See notes below.

---

**NOTE:** If the count is invalid or any referenced data element is undefined, the user program error bit (6) and the instruction failed bit (11) in STW01 are set to 1. If this is the first program error encountered on the current RLL scan, the value 5 (Table overflow) is written to STW200. Finally, power flow is turned off and the RLL scan continues with the next instruction of the current network. The contents of the destination are not changed.

---

---

**NOTE:** For the 575, if a MOVE instruction attempts to access a non-existent VMEbus address, a VMEbus error occurs. If this is the first VMEbus error, the offending address is written to STW227-STW228 and STW229-STW230 is cleared. Next, the user program error bit (6) and the instruction failed bit (11) in STW01 are set to 1 and, if this is the first program error encountered on the current RLL scan, the value 7 (VMEbus error) is written to STW200. The controller then continues execution with the next RLL instruction of the current network after turning power flow off. If the VMEbus error occurred in the middle of the MOVE operation, a partial move occurred.

---

When the input is off, the instruction does not execute and there is no power flow at the box output. Bit 11 of STW01 turns off.

# MOVE

## Move Element (continued)

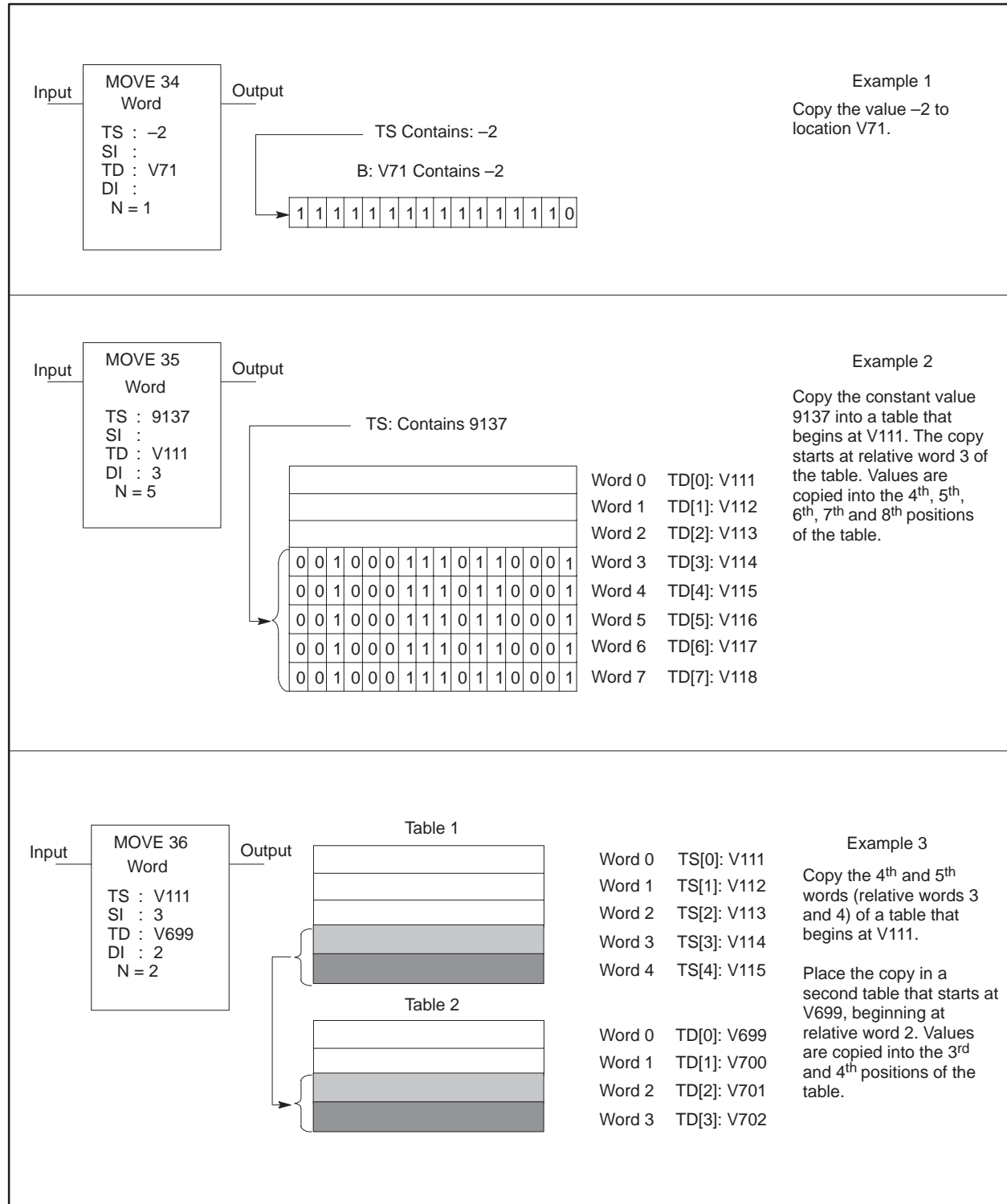


Figure 6-49 Examples of the MOVE Instruction

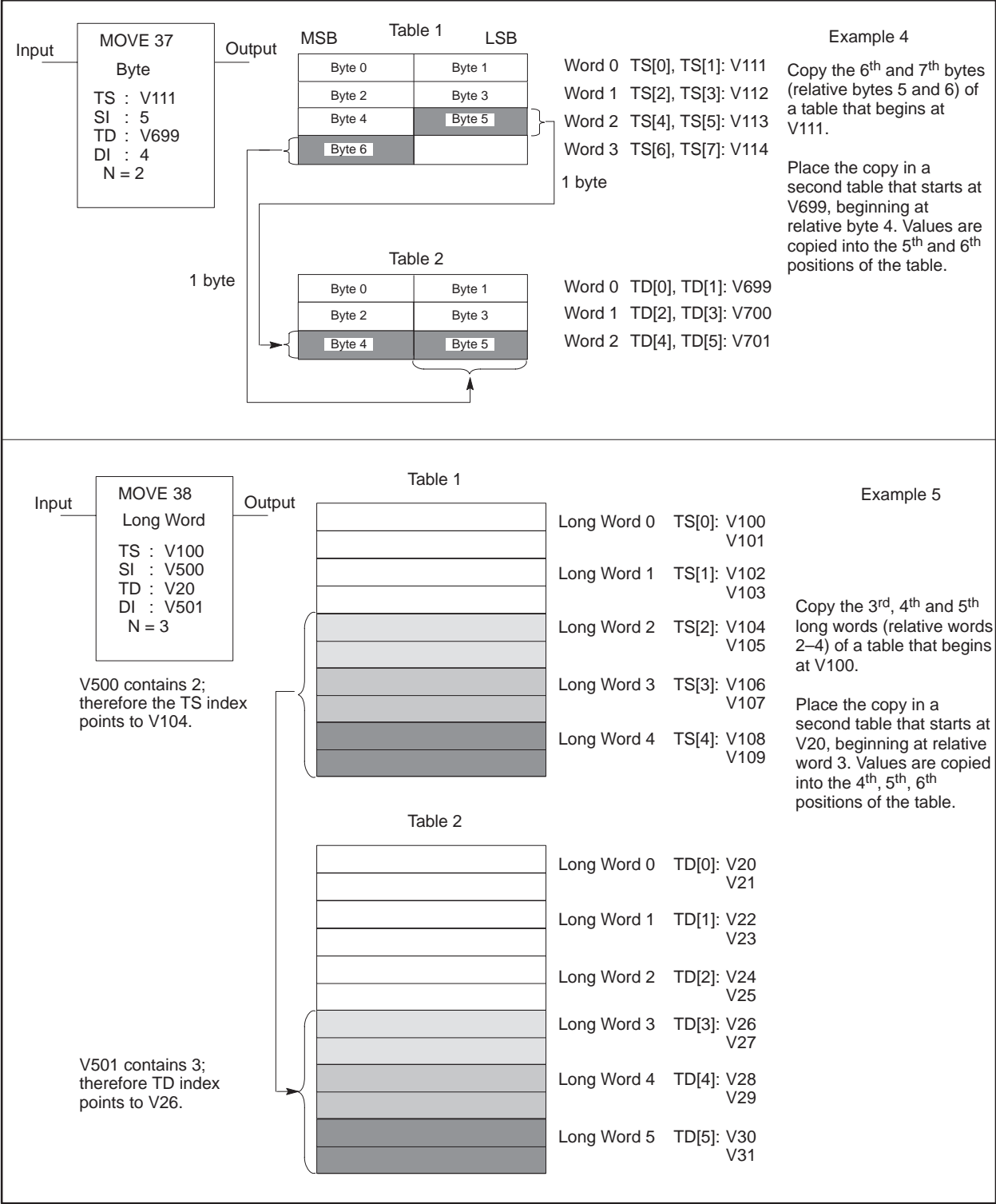


Figure 6-49 Examples of the MOVE Instruction (continued)

# MOVE

## Move Element (continued)

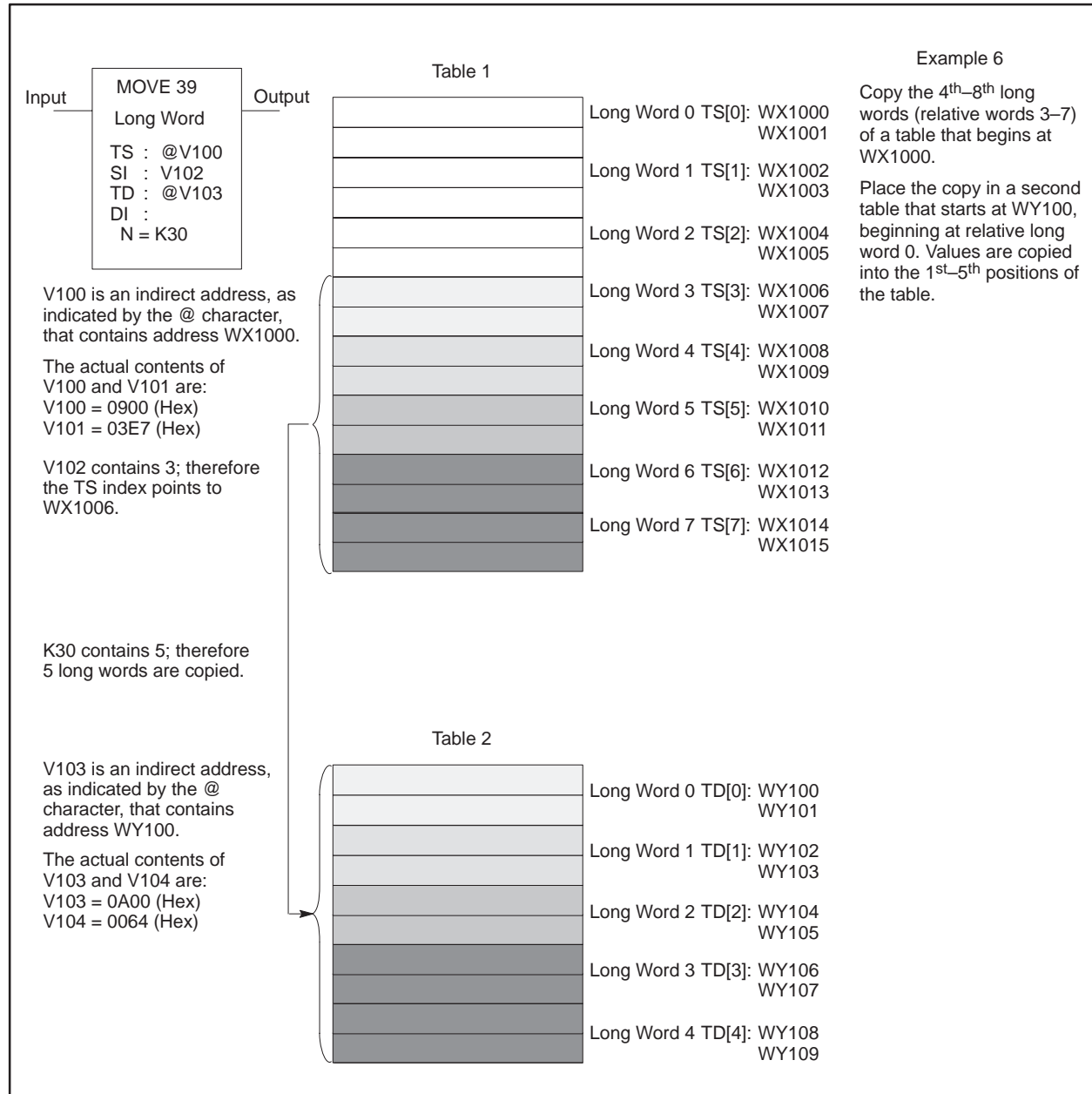


Figure 6-49 Examples of the MOVE Instruction (continued)

---

**Specifying Type of Elements**

Designate the type of the data elements to be moved.

- Byte — The element is 8 bits long.
- Word — The element is 16 bits long.
- Long word — The element is 32 bits long.

**Specifying Source**

You can specify any of the following elements in TS.

- Constant value (range is determined by the data element type) — Specify any signed integer. When the MOVE executes, the specified value is copied to each element of the destination table.
- Direct address — Specify any readable word and designate it a direct address. MOVE copies the contents of the memory location(s), starting at this address, to the destination.
- Indirect address — Specify any readable word and designate it an indirect address by preceding the address with the “@” character, e.g., @V929. The long word at this indirect address must contain another address, and MOVE copies the contents of the memory location(s), starting at this second address, to the destination.

Use the LDA instruction to load an address into a memory location.

**Specifying Index for Source**

Use the first optional field SI as an index into a table when you want to copy elements of a table to a destination. SI designates the relative element, in the table referenced by TS, that is to be copied. The element at TS<sub>0</sub> is the first element in the table. You can specify one of the following in SI.

- Constant index (range = 0 to 65535) — You can leave IN blank or enter 0 and no indexing is done.
- Variable index — Specify any readable word. The content of this word is an unsigned integer (0–65535) that gives the element number of the first element to copy.

If an indirect source address is indexed, the controller first resolves the address and then indexes it. See Figure 6-50.

## Move Element (continued)

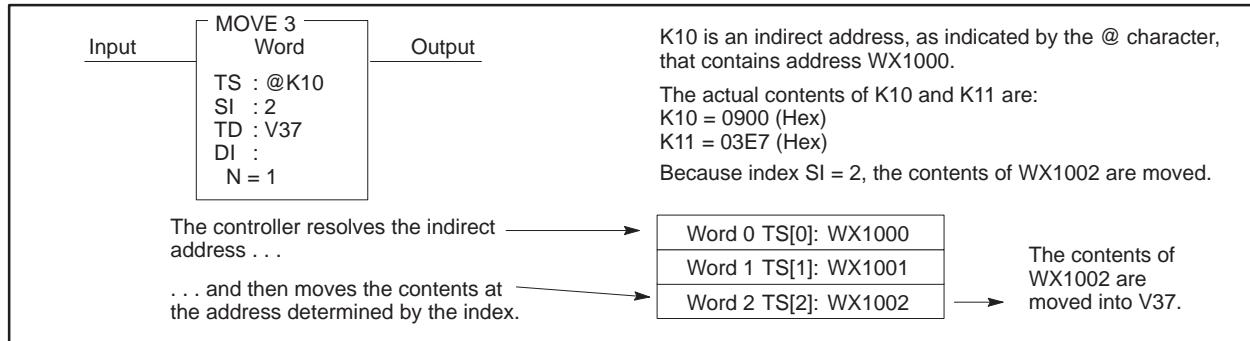


Figure 6-50 Address/Source Index Resolution

## Specifying Destination

You can specify one of the following elements in TD.

- **Direct address** — Specify any writable word and designate it a direct address. MOVE copies the source element(s) into the memory location(s) starting at this address.
- **Indirect address** — Specify any readable word and designate it an indirect address by preceding the address with the @ character, e.g., @V929. The long word at this indirect address must contain another address, and MOVE copies the source element(s) into the memory location(s), starting at this second address. Use the LDA instruction to load an address into a memory location. You can enter a readable word, e.g., a K-Memory address into field TD, but the second address referenced by the address in TD must be a writable word.

## Specifying Index for Destination

Use the second optional field DI as an index into a table when you want to copy an element(s) into a table. DI designates the relative element in a table, referenced by TD, into which the source is copied. The element at TD<sub>0</sub> is the first element in the table.

You can specify one of the following in DI.

- **Constant index** (range = 0 to 65535) — You can leave DI blank or enter 0 and no indexing is done.
- **Variable index** — Specify any readable word. The content of this address is an unsigned integer (0 to 65535) that gives the element number of the first element in the table to which the source element(s) is copied.

If an indirect destination address is indexed, the controller first resolves the address and then indexes it. See Figure 6-51.

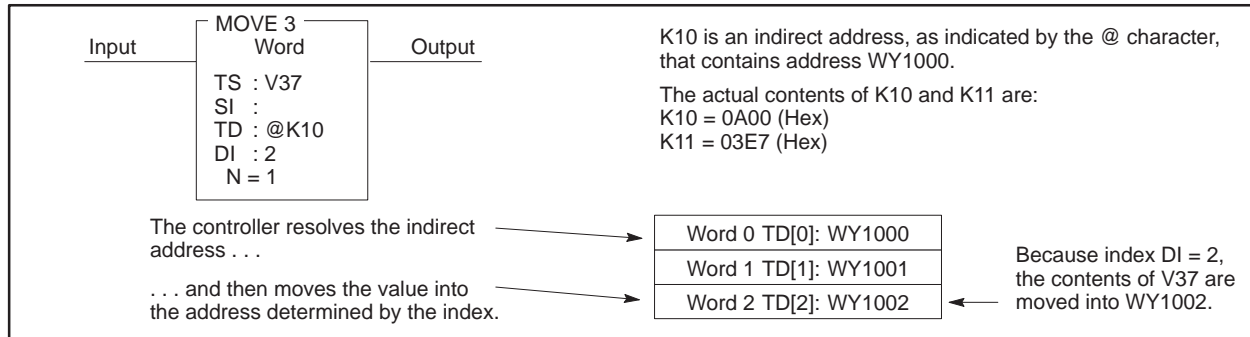


Figure 6-51 Address/Destination Index Resolution

### Specifying Number of Elements to Move

Designate the number of elements to be copied in the count field N. You can specify one of the following in N.

- **Constant count:** Specify an unsigned integer in the range 1–32767.
- **Variable count:** Enter any readable word. The value of the count is determined by the contents of this word when the MOVE executes. The count range is 0–32767, where 0 means that no elements move.

### See Also

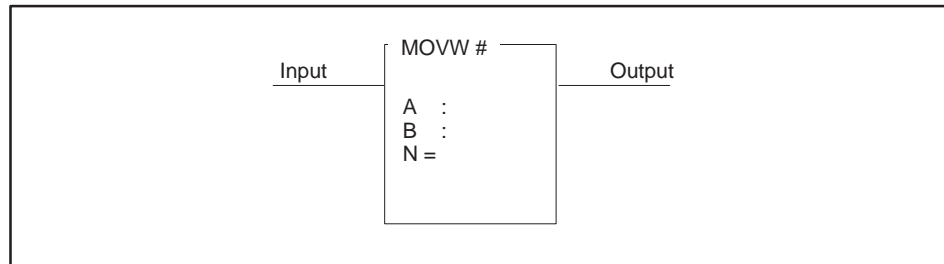
These RLL instructions are also used for word moves.

LDA	LDC	MIRW	MOVW	MWFT	MWI
MWIR	MWTT	SHRW			



## 6.37 Move Word

**MOVW Description** The Move Word instruction (Figure 6-52) copies up to 256 contiguous words from one location to another. The starting memory location for the words to be moved is specified by A, and the starting memory location for their destination is specified by B. All words are copied in a single scan.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any readable word or constant (-32768 to +32767)	Starting memory location for the words to be copied. Value to be copied if a constant is used.
B	Any writeable word	Starting memory location for the destination.
C	1-256	Number of words to be copied.

Figure 6-52 MOVW Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**MOVW Operation**

When the input is on, the MOVW box executes. If the input remains on, the operation executes on every scan. The operation of MOVW follows and is illustrated in Figure 6-53.

- A table of up to 256 (N) words, with a starting memory location specified by A, is copied.

If a constant value is specified in A, then the constant is copied to all destination locations.

- The words are copied to a destination beginning at the memory location designated by B.
- The output turns on when the instruction executes.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

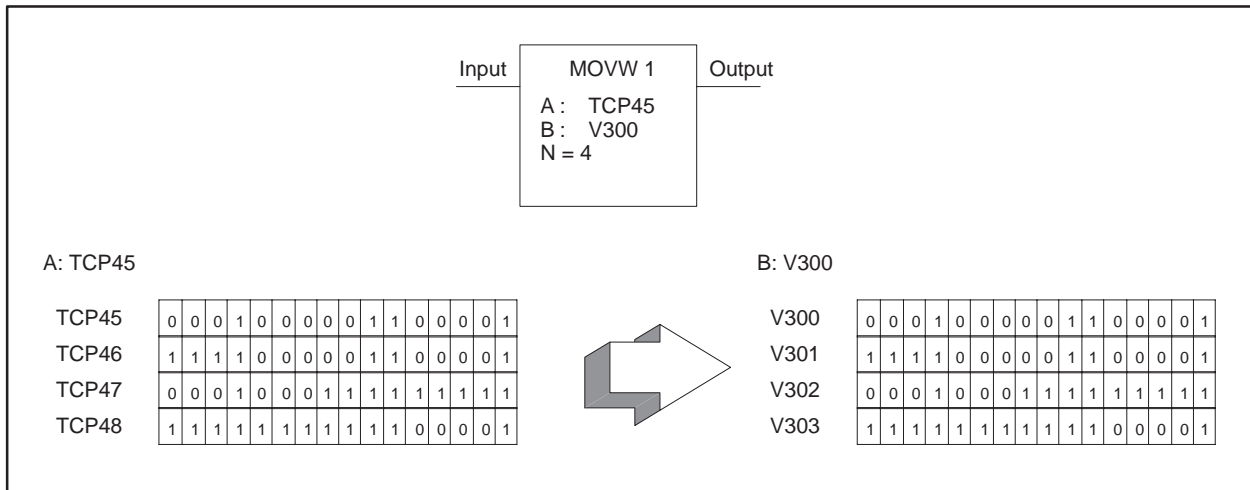


Figure 6-53 The MOVW Operation

**See Also**

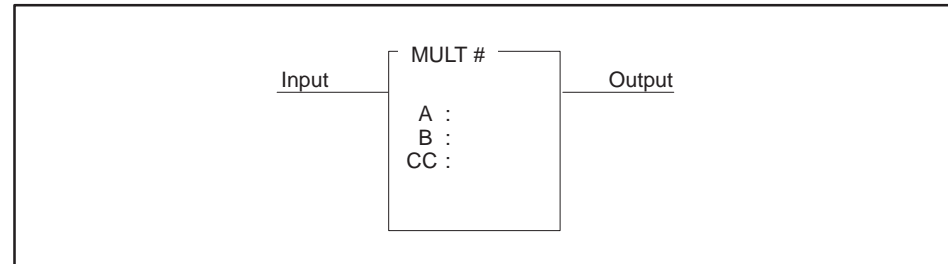
These RLL instructions are also used for word moves.

LDA	LDC	MIRW	MOVE	MWFT	MWI
MWIR	MWTT	SHRW			

## 6.38 Multiply

### MULT Description

The Multiply instruction (Figure 6-54) multiplies a signed integer in memory location A by a signed integer in memory location B. The product is stored in one long word, CC and CC + 1.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any readable word	Memory location for the multiplicand (a word).
B	Any readable word or constant (–32768 to +32767)	Memory location for the multiplier (a word).
		Value of the multiplier if a constant is used.
CC	Any writeable long word	Memory location for the product (a long word). CC holds the 16 most significant bits, and CC + 1 holds the 16 least significant bits.

Figure 6-54 MULT Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

### MULT Operation

When the input is on, the MULT box executes. If the input remains on, the operation executes on every scan. The operation of the MULT that is illustrated in Figure 6-55 is  $(CC, CC + 1) = A \times B$ .

- The values in A and B are not affected by the operation.
- When the multiplication executes, the output turns on.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

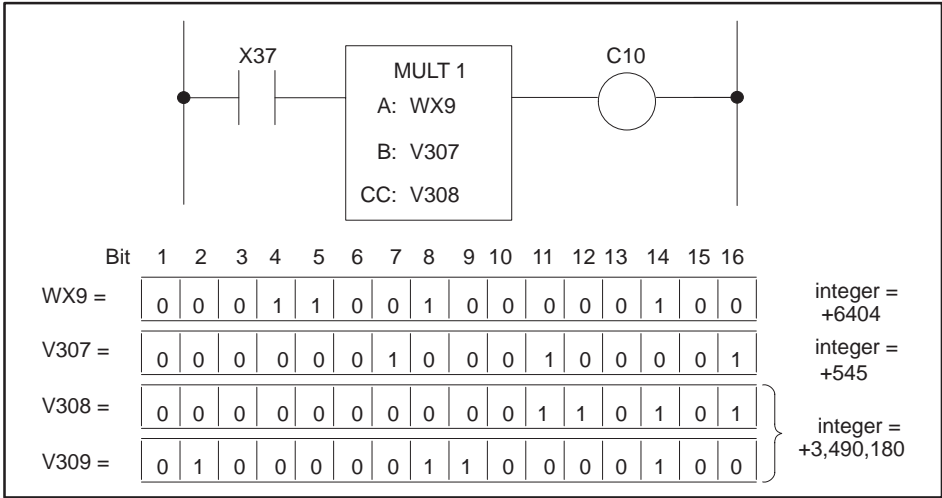


Figure 6-55 Multiplication Example

See Also

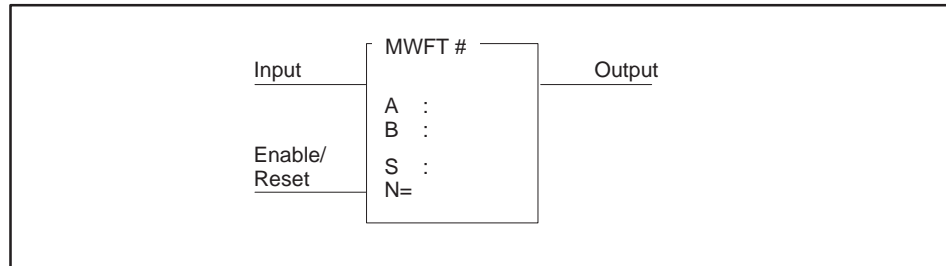
These RLL instructions can also be used for math operations.

ABSV	ADD	CMP	DIV	SQRT	SUB
Relational Contact					

### 6.39 Move Word from Table

**MWFT Description**

The Move Word from Table instruction (Figure 6-56) copies a word from a table to a V-Memory location. A table pointer designates the address of the next word in the table to be copied. One word is copied each scan.



Field	Valid Values	Function
#	Varies with controller model	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of table-move memory on page 4-6 discussed in Section 4.2.
A	V, W, (G, VMS, VMM, 575)	Specifies memory location of the table pointer. The value contained in pointer A is the memory location in the table of the next word to be copied.
B	V, W, (G, VMS, VMM, 575)	Memory location of the destination.
S	V	Starting address of the table.
N	1-256	Number of words to be copied.

Figure 6-56 MWFT Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**MWFT Operation**

The operation of the MWFT is described below and illustrated in Figure 6-57.

- When the Enable/Reset is off, the table starting address S loads into pointer A.
- When the Enable/Reset turns on, the box is enabled. When the Input also turns on, the following actions occur.

A word is copied from the table address specified by the value contained in pointer A to the memory location specified by B.

After the word is copied, table pointer A, that holds the address of the next word in the table to be copied, increments by 1.

If the Input and the Enable/Reset remain on, one word is copied every scan. As each word is copied, the table pointer increments until N words are copied.

- The output turns on when the last word is copied.
- When the instruction is reset, all table values remain unchanged, and destination address B contains the last word copied from the table.

If the Enable/Reset is off, the instruction does not execute, and there is no power flow at the box output.

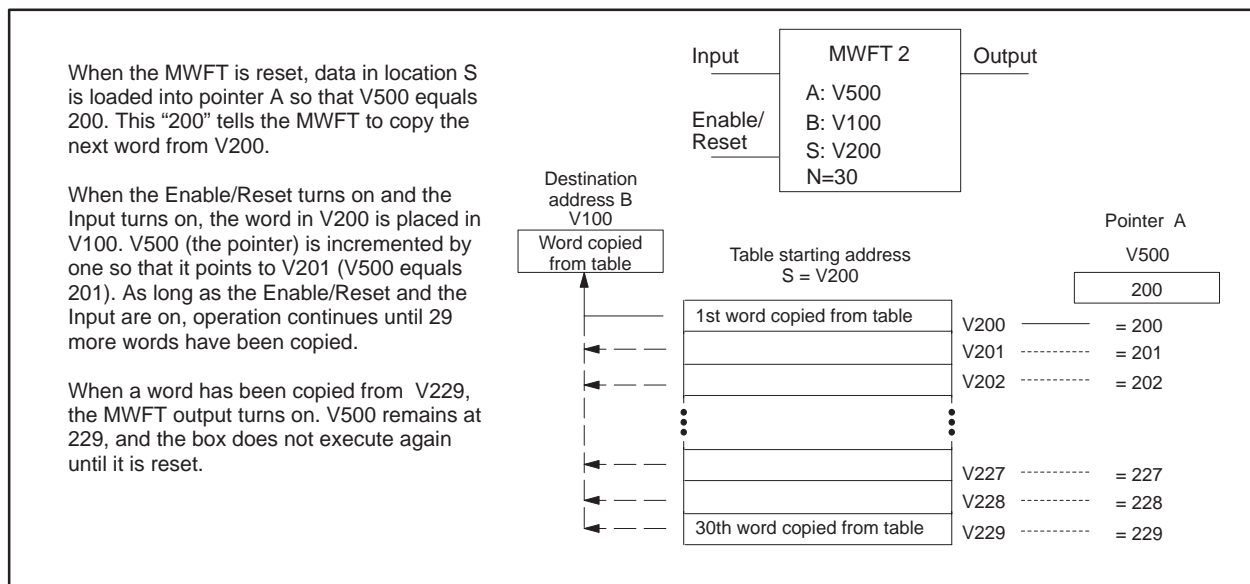


Figure 6-57 The MWFT Operation

See Also

These RLL instructions are also used for word moves.

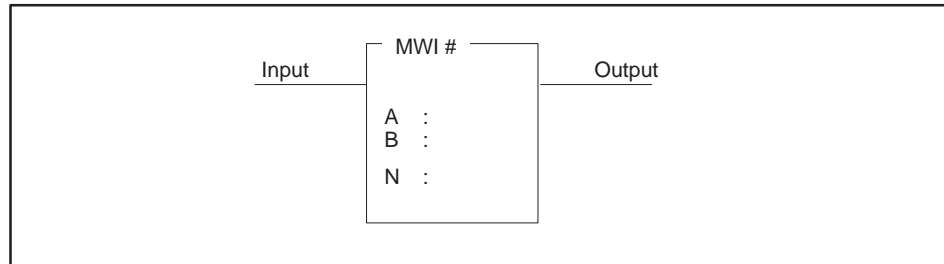
LDA	LDC	MIRW	MOVE	MOVW	MWI
MWIR	MWTT	SHRW			

Refer to Section E.10 for an application example of the MWFT.

## 6.40 Move Word with Index

### MWI Description

The Move Word with Index instruction (Figure 6-58) allows you to copy up to 256 words from one area of V-Memory to another area of V-Memory during a single scan.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	V, W, (G, VMS, VMM, 575) or constant (–32768 to +32767)	Memory location of word which gives the V-Memory index for the base of the source table. The addressed word can contain a value in the range 1 to 32767, corresponding to V1 through V32767, respectively.
B	V, W, (G, VMS, VMM, 575)	Memory location of word which gives the V-Memory index for the base of the source table. The addressed word can contain a value in the range 1 to 32767, corresponding to V1 through V32767, respectively.
N	V, W, (G, VMS, VMM, 575)	Memory location of word which gives the number of words to be moved. The addressed word can contain a value in the range 0 (Don't Move) through 256.

Figure 6-58 MWI Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

### MWI Operation

When the input is on, the MWI box executes. If the input remains on, the operation is executed on every scan. The operation of the MWI is described below and illustrated in Figure 6-59.

- The V-Memory table having a starting index specified in the word addressed by A is copied to the V-Memory table having a starting index specified in the word addressed by B.

- Up to 256 words can be copied as determined by the content of the word addressed by N.
- All words are copied into the destination table each scan.
- If the sum of the number of words to move and either the source (destination) table index exceeds the configured size (in words) of V-Memory, or if the number of words exceeds 256, the instruction does not execute. The output turns on when the instruction is executed.
- If either the source or the destination pointer plus table length exceeds V-Memory size, the instruction does not execute. The output is turned off, and bit 11 in STW01 is set.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

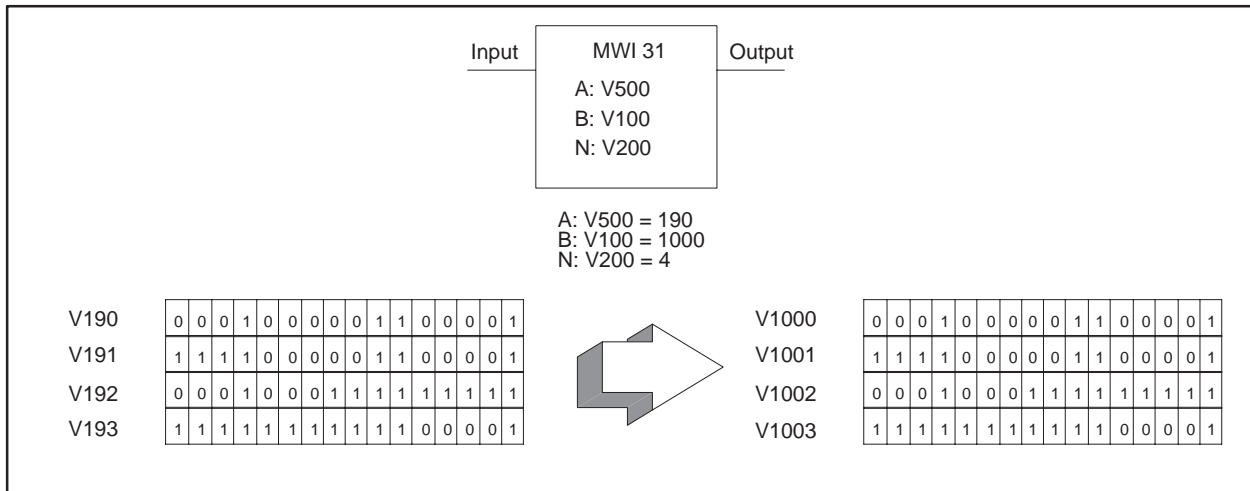


Figure 6-59 The MWI Operation

See Also

These RLL instructions are also used for word moves.

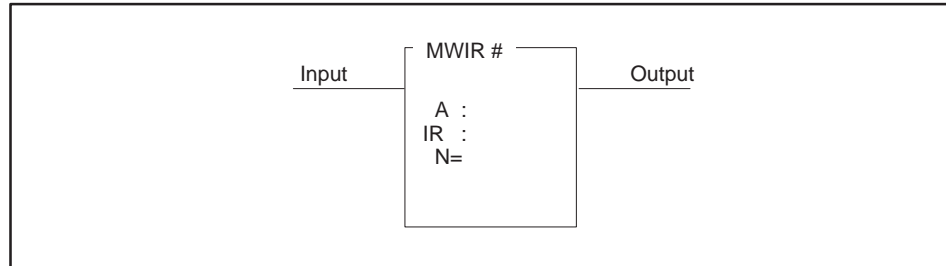
LDA	LDC	MIRW	MOVE	MOVW	MWFT
MWIR	MWTT	SHRW			



## 6.41 Move Word to Image Register

### MWIR Description

The Move Word to Image Register instruction (Figure 6-60) copies a specified number of bits from a word memory location to the discrete image register or into the control relay memory locations. All bits are copied in a single scan.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any readable word	Specifies memory location from which the bits are copied.
IR	Y, C, B	Starting address of the control relays or the discrete image register.
N	1–16	Number of bits to be copied.

Figure 6-60 MWIR Format

**NOTE:** If you plan to use this instruction in a subroutine (using B-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

### MWIR Operation

When the input is on, the MWIR box executes. If the input remains on, the operation executes on every scan. The operation of the MWIR box is described below and illustrated in Figure 6-61.

- Up to 16 bits (N) in the word memory location specified by A are copied, beginning with the least significant bit of the word.
- Bits are copied into the discrete image register or into the control relay memory locations, starting at the address designated by IR. The bits are copied during a single scan.
- The output turns on when the instruction is executed.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

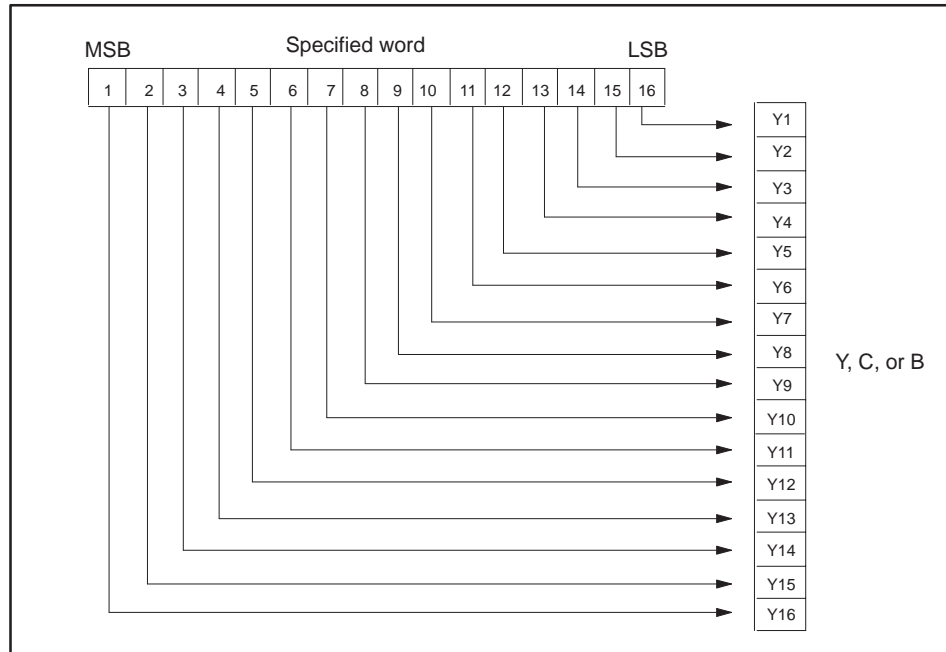


Figure 6-61 The MWIR Format

See Also

These RLL instructions are also used for word moves.

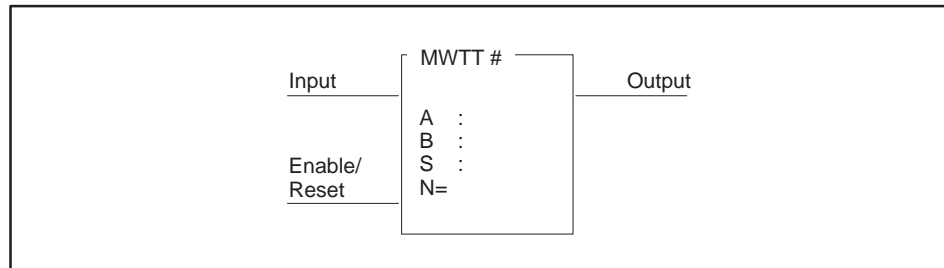
LDA	LDC	MIRW	MOVE	MOVW	MWFT
MWI	MWTT	SHRW			

Refer to Section E.8 for an application example of the MWIR.

## 6.42 Move Word To Table

### MWTT Description

The Move Word To Table instruction (Figure 6-62) copies a word from a source in memory to a destination within a table. A pointer designates the memory location in the table into which the next word is copied. One word is copied per scan.



Field	Valid Values	Function
#	Varies with controller model	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of table-move memory discussed on page 4-6 in Section 4.2.
A	V, W, (G, VMS, VMM, 575)	Specifies memory location of the word to be copied.
B	V, W, (G, VMS, VMM, 575)	Specifies memory location of the table pointer. The value contained in pointer B is the table memory location into which the next word is copied.
S	V	Starting address of the table.
N	1-256	Size of the table in words.

Figure 6-62 MWTT Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

### MWTT Operation

The operation of the MWTT is described below and shown in Figure 6-63.

- When the Enable/Reset is off, the table starting address S is loaded into pointer B.
- When the Enable/Reset turns on, the box is enabled. When the Input also turns on, the following actions occur.

A word is copied from the memory location specified by A to the table memory location specified by the value contained in pointer B.

Pointer B, which holds the destination memory location in the table for the next word, increments by 1.

If the Input remains on, one word is copied every scan. As each word is copied, the table pointer increments until N words are copied.

- The output turns on when the last word is copied.
- When the instruction is reset, all values in the table remain unchanged.

If the Enable/Reset is off, the instruction does not execute, and there is no power flow at the box output.

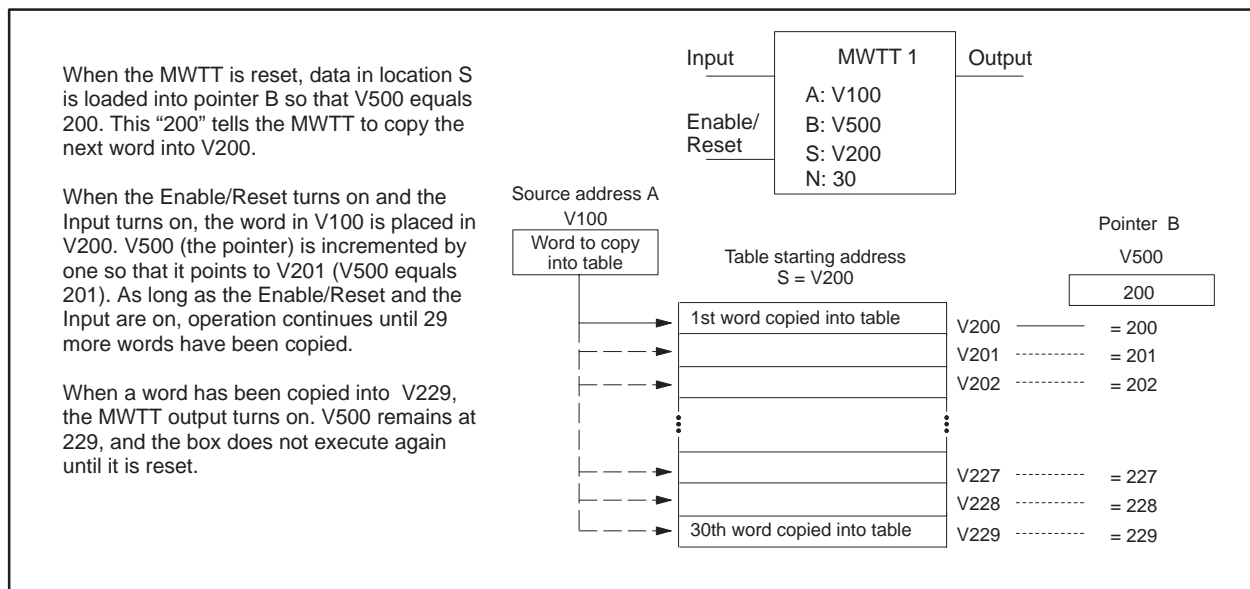


Figure 6-63 The MWTT Operation

See Also

These RLL instructions are also used for word moves.

LDA	LDC	MIRW	MOVE	MOVW	MWFT
MWI	MWIR	SHRW			

Refer to Section E.9 for an application example of the MWTT.

NOT

6.43 NOT

NOT Description

The NOT instruction (Figure 6-64) inverts the power flow.

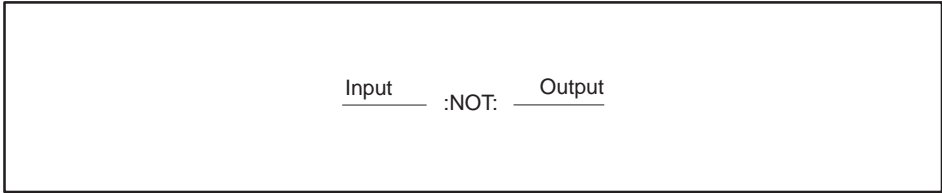


Figure 6-64 NOT Format

NOT Operation

The NOT changes the power flow to the state opposite its current state. Refer to Figure 6-65 for an example of how the NOT can simplify programming.

**NOTE:** Do not program a NOT in parallel with any rung that does not connect to the power rail.

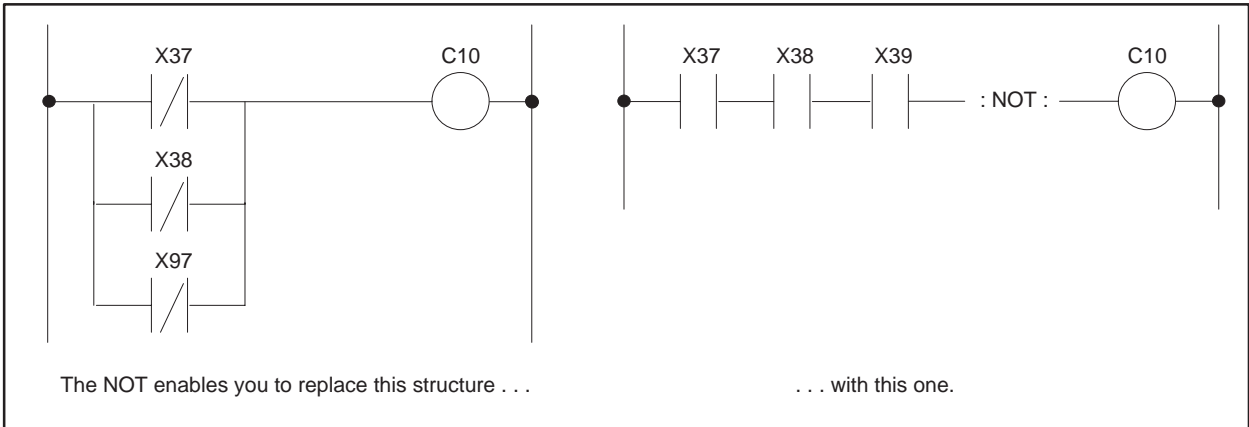


Figure 6-65 NOT Example

See Also

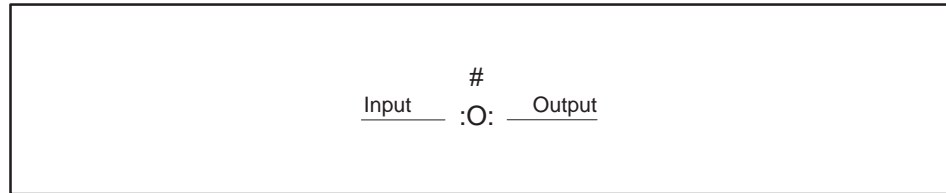
These RLL instructions are also used for electromechanical replacement.

Coils	Contacts	CTR	DCAT	DRUM	EDRUM
JMP	MCAT	MCR	MDRMD	MDRMW	SHRB
SKP/LBL	TMR	UDC			

## 6.44 One Shot

### One Shot Description

The One Shot instruction (Figure 6-66) turns on an output for a single scan.



Field	Valid Values	Function
#	Varies with controller model	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of One Shot memory discussed on page 4-7 in Section 4.2. Each One Shot instruction must have a unique instruction number.

Figure 6-66 One Shot Format

### One Shot Operation

The operation of the One Shot is described below.

- When the input transitions from off to on, the output turns on for exactly one scan.
- After the One Shot executes, its input must be off for at least one scan before the instruction executes again.

If the input is off, the instruction does not execute, and there is no power flow at the output.

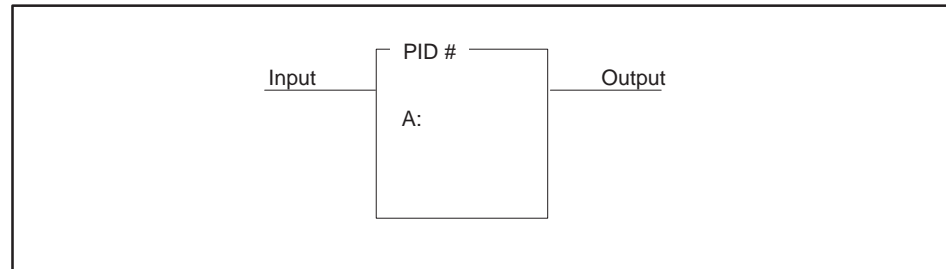
Refer to Section E.14 for an application example of the One Shot.

## 6.45 PID Loop

### PID Fast Loop Description

The PID instruction (Figure 6-67) performs the proportional – integral – derivative (PID) fast loop function.

**NOTE:** The PID instruction is not supported by TISOFT.



Field	Valid Values	Function
#	0–65535	Instruction reference number. Numbers for documentation purposes only; can be repeated.
A	Any readable word or constant	Specifies word that contains the fast loop number (valid fast loop number: 129 to 256). PID fast loop number if constant is used (valid fast loop number: 129 to 256).

Figure 6-67 PID Format

### PID Operation

The PID instruction can be used anywhere within the RLL program that a single-line input box instruction can be used. This instruction allows you to schedule a fast loop for immediate execution.

When power flow is on for the PID instruction, the referenced fast loop executes immediately to completion. The result of the fast loop is available to the next element of the current RLL rung. The fast loop algorithm uses the floating-point math coprocessor; consequently, it executes faster and with less overhead than a standard loop.

You can schedule a fast loop to execute every scan or cyclically by placing the PID instruction in a cyclic task. However, to ensure correct functioning of the fast loop (and SmarTune, if used,) schedule the fast loop to execute on a consistent time basis.

---

Fast loop programming is done by using the same programming table used for loops 1 through 128. Data entered in the SAMPLE RATE field, however, is not used since the sample rate is based on when you schedule the PID instruction to execute. Refer to Chapter 9, section 9.4 for information about programming loops.

---

**NOTE:** The ramp/soak feature is not supported by the fast loops.

---

When the input is turned on, the PID box will execute the loop algorithm for a particular fast loop (129–256) to completion as part of the RLL process and the box output will turn on. If the input remains on, the instruction executes on every scan. The following exceptions cause the program's execution to fail:

- If the fast loop is not configured, user program error 13 is logged in STW200 and there is no power flow at the box output.
- If the fast loop is disabled, user program error 14 is logged in STW200 and there is no power flow at the box output.

When the input is off, the instruction does not execute, and there is no power flow at the box output.

**See Also**

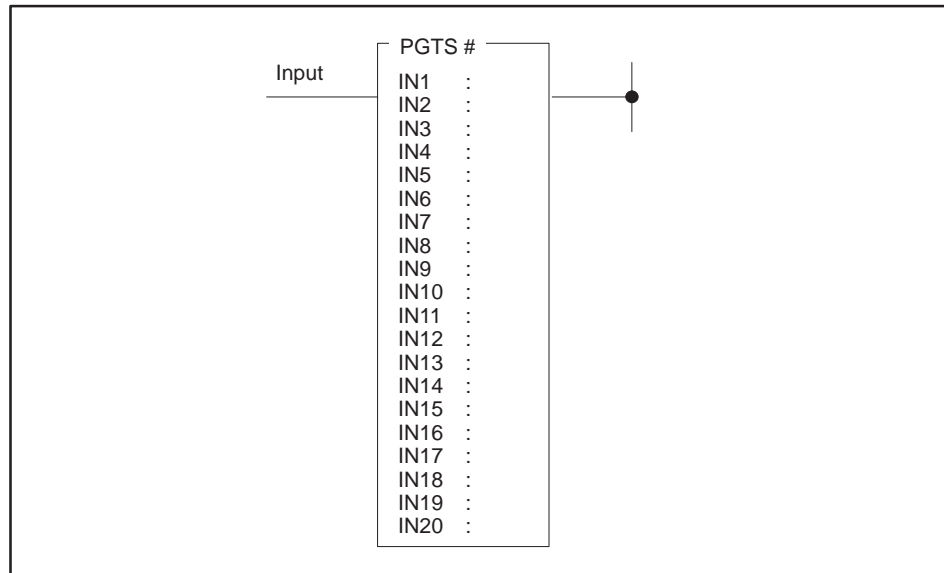
Refer to Chapter 9 for information on loop programming.



## 6.46 Parameterized Go To Subroutine

### PGTS Description

PGTS (Figure 6-68) operates similarly to the GTS instruction. Use PGTS to call a section of the RLL program that is preceded by a subroutine number and execute it. Unlike GTS, the PGTS allows you to pass parameters to a subroutine. These parameters allow you to write a generic subroutine using parameter identifiers (IN1–IN20) instead of specific memory locations. Several PGTS instructions (using different memory locations as parameters) can then call the same general subroutine.



Field	Valid Values	Function
#	1–32	Designates subroutine to call. You can pass parameters only to subroutines numbered 1–32.
IN	IN followed by any readable bit or word; IO followed by any writeable bit or word.	Designates address that contains data to be read by the subroutine. Change the field to show <b>IO</b> when you want the subroutine to write data to the address after it completes execution. When the field shows <b>IN</b> , the subroutine only reads data at the address. B and W locations are valid only when PGTS is used in a subroutine.

Figure 6-68 PGTS Format

**NOTE:** If subroutine parameters (W or B memory) are used as operands of instructions, note that parameter passing by PGTS and PGTSZ is by value. An operand that implies multiple memory locations will access multiple W or B locations and not multiple locations from their original memory area. Only explicitly passed parameters may be accessed with W or B operands. Refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

## PGTS Operation

PGTS operation is described below and shown in Figure 6-69.

- When the input turns on, the contents of each parameter are set equal to the contents of the memory location specified in the parameter field. Then the subroutine indicated by the PGTS number is called.
- When the subroutine returns control to the main RLL program, the contents of the memory location specified in each read/write (IO) parameter field is set equal to the contents of the parameter. The contents of memory locations designated IN are not changed.
- Contents of parameters are stored in PGTS discrete and word parameter areas (Section 4.2). When you use a parameter in the subroutine, refer to discrete points as  $B_n$  and words as  $W_n$ , where  $n$  = the number of the parameter.
- When you program a PGTS with TISOFT, the parameters must be entered consecutively. That is, do not skip any parameters.
- If you do not need to specify parameters, use the GTS instead (GTS uses less L-Memory).
- While you can still access any memory location from a subroutine, the PGTS allows you to create a generic subroutine that is called by multiple PGTS instructions, varying the parameters.

If the input is off, the instruction does not execute and the subroutine is not called for execution.

### **WARNING**

When you do a run-time edit, you must enter all the instructions required to define a subroutine before setting the controller to RUN mode. If you enter these instructions out of order, the controller changes from RUN to PROGRAM mode and freezes the outputs in their current status, which could cause unexpected operation of the controller.

Unexpected controller operation can result in death or serious injury to personnel, and/or equipment damage.

To ensure that instructions have been entered correctly, use the syntax check function to validate a program before placing the controller in RUN mode.

When you do a run-time edit using an earlier release of TISOFT, you must enter the instructions in this order: END, RTN, SBR, GTS or PGTS/PGTSZ.

Parameterized Go to Subroutine (continued)

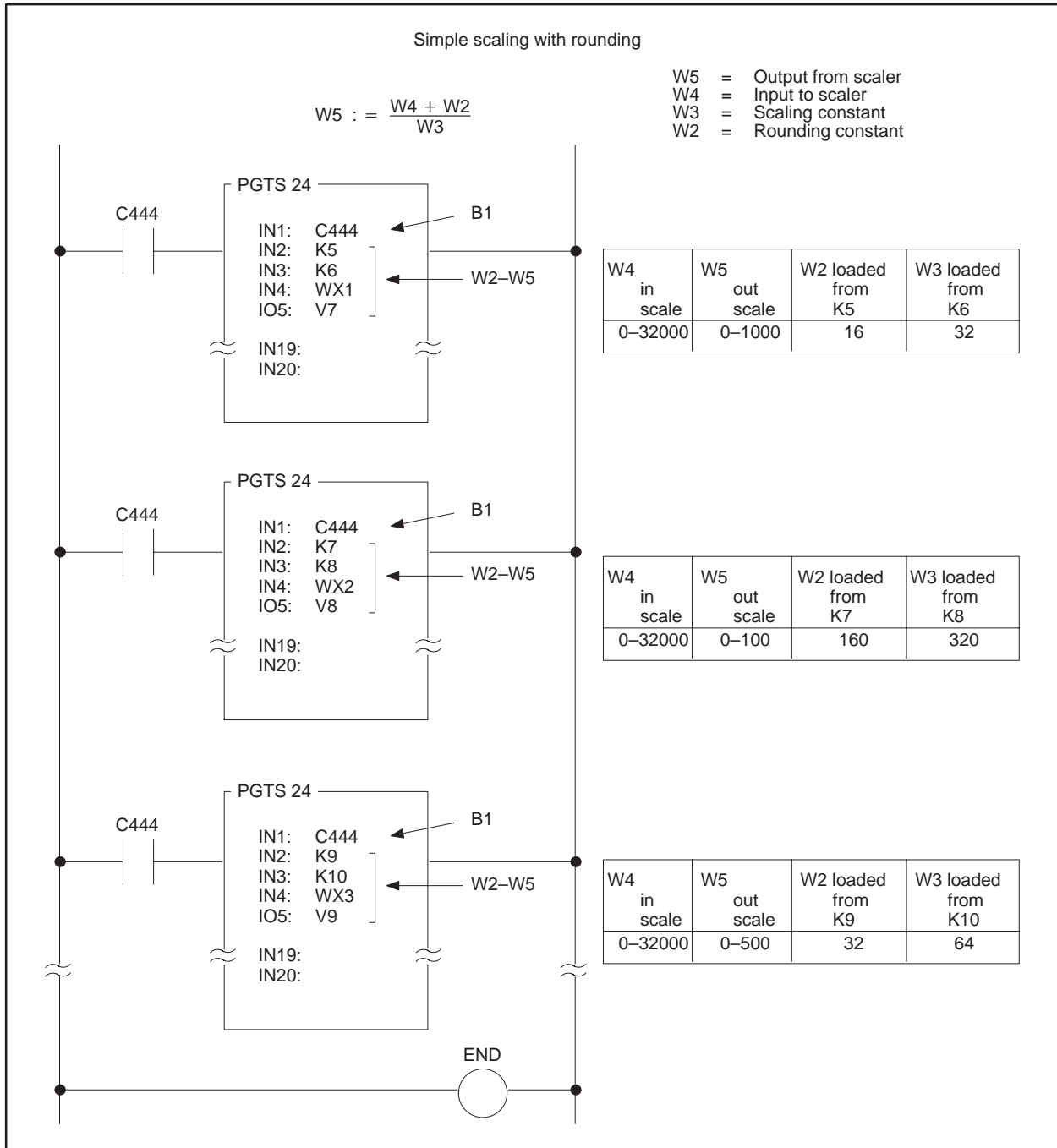


Figure 6-69 PGTS Instruction Example 2

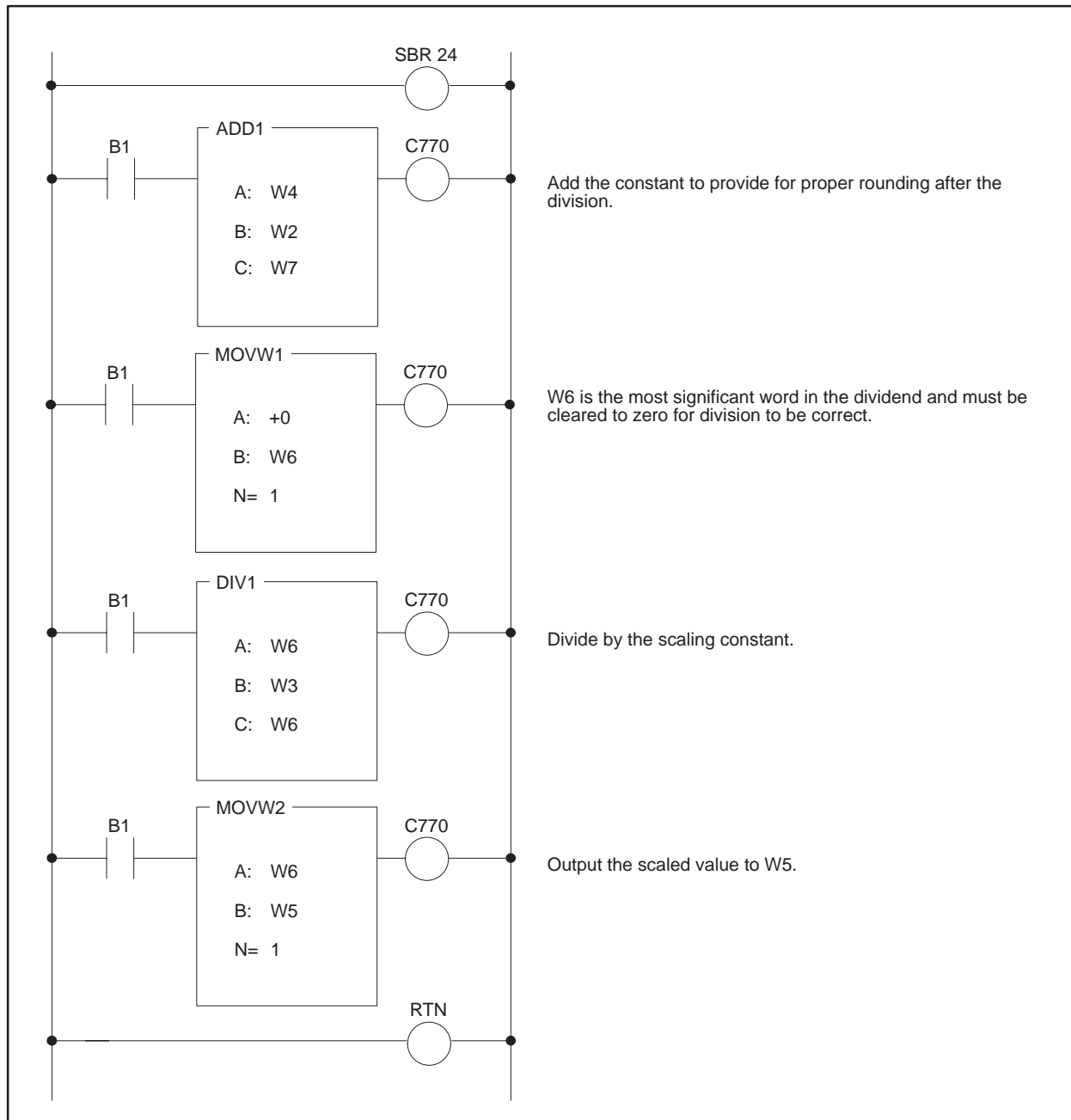


Figure 6-69 PGTS Instruction Example 2 (continued)

## Parameterized Go To Subroutine (continued)

---

---

**NOTE:** Avoid a direct reference in a subroutine to a memory location that is also identified as a parameter in the PGTS instruction. Otherwise, you can create a condition where the value of the parameter and the value in the memory location do not agree. Refer to the example in Figure 6-70.

---

If you use an instruction that copies long words into or from the subroutine, you need to allocate a parameter for each word of each long word that is copied.

For example, the product of a multiplication is stored as a long word. Two parameters are required to transfer the product from the subroutine to the main program. If you multiply the contents of V22 by the contents of V23 and store the product in V50 and V51, then both V50 and V51 must be listed as consecutive parameters.

### See Also

These RLL instructions are also used for subroutine operations.

GTS	PGTSZ	RTN	SBR	SFPGM	SFSUB	XSUB
-----	-------	-----	-----	-------	-------	------

If an IO parameter IO1, that specifies a non-parameter memory location Y1, is passed to a subroutine, and the subroutine references Y1 directly, then the values for IO1 and Y1 may not agree when the subroutine returns control back to the main program.

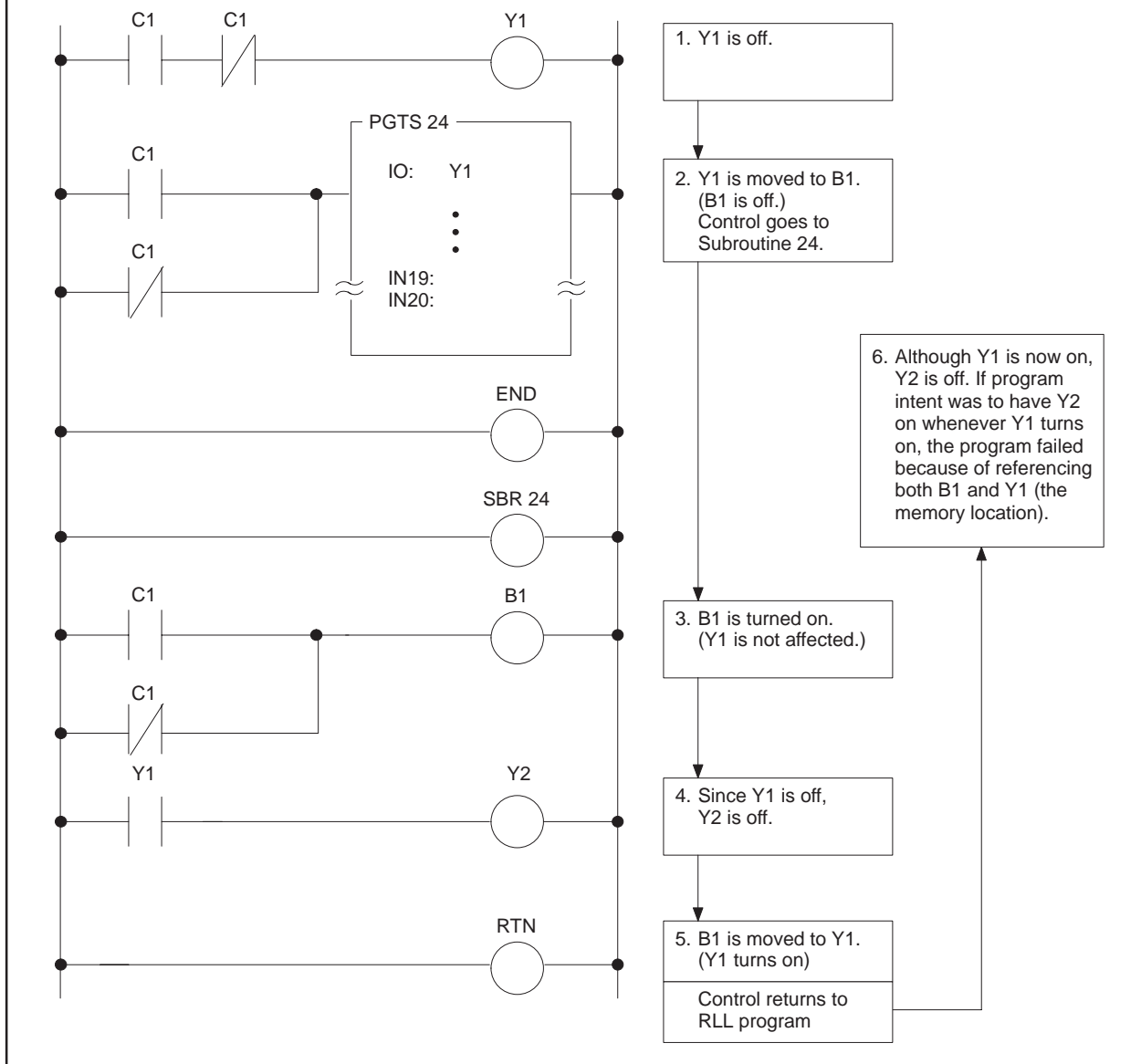
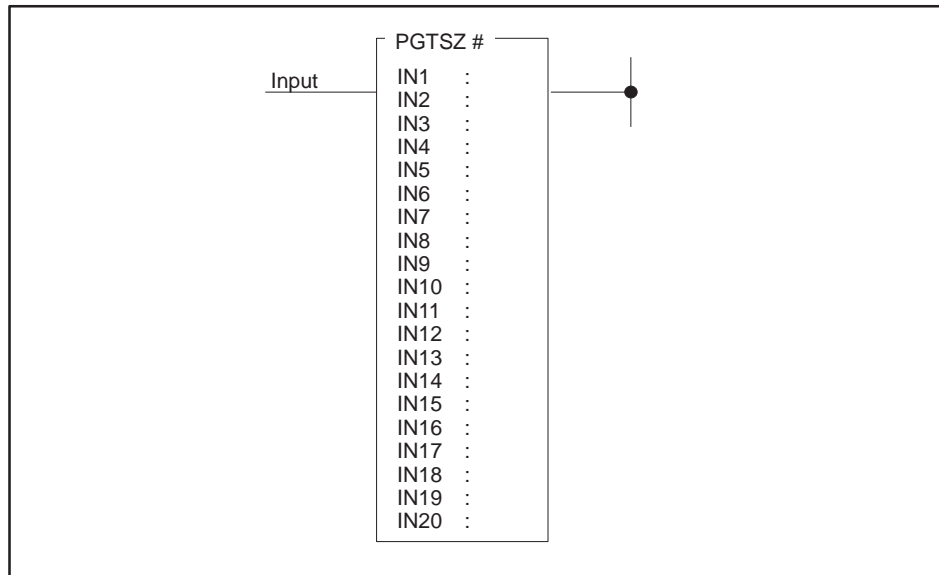


Figure 6-70 PGTS Instruction Example 1

## 6.47 Parameterized Go To Subroutine (Zero)

**PGTSZ Description** The PGTSZ instruction (Figure 6-71) operates similarly to the PGTS instruction. PGTSZ calls an RLL subroutine for execution and passes parameters to it. Unlike PGTS, the PGTSZ clears all discrete I/O parameters when the input to the PGTSZ is off.



Field	Valid Values	Function
#	1-32	Designates subroutine to call. You can pass parameters only to subroutines numbered 1-32.
IN	IN followed by any readable bit or word; IO followed by any writeable bit or word.	Designates address that contains data to be read by the subroutine. Change the field to show <b>IO</b> when you want the subroutine to write data to the address after it completes execution. When the field shows <b>IN</b> , the subroutine only reads data at the address. B and W locations valid only when PGTS is used in a subroutine.

Figure 6-71 PGTSZ Format

**NOTE:** If subroutine parameters (W or B memory) are used as operands of instructions, note that parameter passing by PGTS and PGTSZ is by value. An operand that implies multiple memory locations will access multiple W or B locations and not multiple locations from their original memory area. Only explicitly passed parameters may be accessed with W or B operands. Refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

---

**PGTSZ Operation**

When the input turns on, operation is identical to that of the PGTS, described in Section 6.46.

If the input is off, all discrete I/O parameters turn off, and the subroutine is not called for execution.

** WARNING**

When you do a run-time edit with TISOFT (Rel 4.2 or later), enter all the instructions required to define a subroutine (END, RTN, SBR, GTS or PGTS/PGTSZ) before setting the controller to RUN mode. Otherwise, the controller changes from RUN to PROGRAM mode and freezes outputs in their current status, which could cause unexpected operation of the controller.

Unexpected controller operation can result in death or serious injury to personnel, and/or equipment damage.

Use the TISOFT syntax check function to validate a program before placing the controller in RUN mode. When you do a run-time edit using an earlier release of TISOFT, you must enter the instructions in this order: END, RTN, SBR, GTS or PGTS/PGTSZ.

**See Also**

These RLL instructions are also used for subroutine operations.

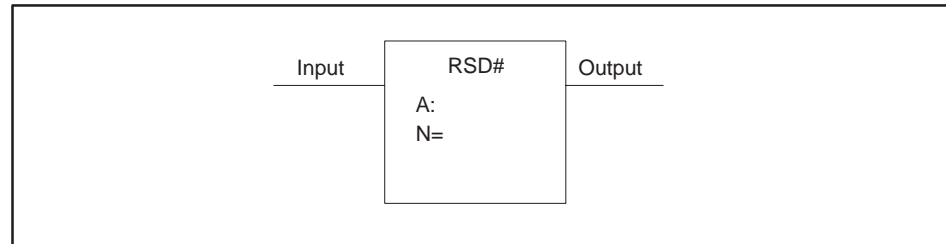
GTS	PGTS	RTN	SBR	SFPGM	SFSUB	XSUB
-----	------	-----	-----	-------	-------	------



## 6.48 Read Slave Diagnostic (RSD)

### RSD Description

The Read Slave Diagnostic instruction (Figure 6-72) transfers a PROFIBUS-DP slave's diagnostic buffer to user memory.



Field	Valid Values	Function
#	1 – 112	Instruction reference number. The number entered indicates the address of the PROFIBUS-DP slave whose diagnostic is to be read. Numbers can be repeated.
A	Any writeable word	Starting memory location for the destination.
N	1 – 256	Maximum number of words to be read. See Table 6-3.

Figure 6-72 RSD Instruction Format

The diagnostic buffer, whose address in user memory is specified by A, is formatted as shown in Table 6-3.

Table 6-3 RSD Buffer Format

Word	Byte	Content
A	0	Status as follows: 0 Transfer successful. 1 Transfer successful. A previous diagnostic was signaled and not read. 2 Transfer failed. The specified slave has not signaled a diagnostic.
	1	Length, in bytes, of actual diagnostic.
A+1...A+N-1	all	Diagnostic area

---

**NOTE:** The length (byte 1 of word A) indicates the actual diagnostic length, as signaled by the PROFIBUS-DP slave. If the size  $[(N-1)*2]$  of the destination buffer's diagnostic area is less than the actual diagnostic length, the diagnostic is truncated by the transfer.

---

## RSD Operation

When the input is on, the RSD box executes. If the input remains on, the operation executes on every scan. The operation of RSD is as follows:

- If the PROFIBUS-DP I/O subsystem is stopped or if the indicated slave has not signaled a diagnostic since the last execution of an RSD instruction for the slave, the destination buffer's status byte is set equal to 2 and the length is set equal to 0.
- If the slave has not signaled more than one diagnostic since the last execution of an RSD instruction for the slave, the destination buffer's status byte is set equal to 0, the length byte is set equal to the length of the last diagnostic signaled, and the value (possibly truncated) of the latest signaled diagnostic is copied to the diagnostic area.
- If the slave has signaled more than one diagnostic since the last execution of an RSD instruction for the slave, the destination buffer's status byte is set equal to 1, the length byte is set equal to the length of the last diagnostic signaled, and the value (possibly truncated) of the latest signaled diagnostic is copied to the diagnostic area.

If the input is off, the instruction does not execute and the output is off.

---

**NOTE:** Status words STW232 through STW238 indicate the PROFIBUS-DP slaves that have signaled a diagnostic that has not been read by an RSD instruction. Use a bit-of-word contact specifying the slave's status word bit as the input to the RSD instruction. Do this in order to execute the instruction whenever there is a diagnostic for the slave corresponding to the bit.

---

---

**NOTE:** The format of a slave's diagnostic buffer is dependent upon the PROFIBUS-DP slave type. See the user documentation for your slave(s).

---

## 6.49 Return from Subroutine

### RTN Description

The RTN instruction (Figure 6-73) ends execution of an RLL subroutine, and returns program execution to the rung following the GTS instruction.

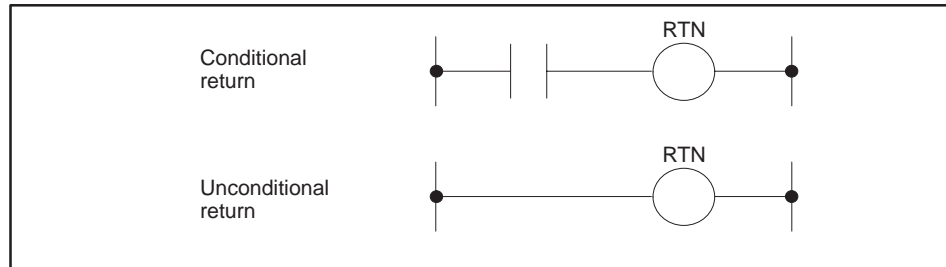


Figure 6-73 RTN Format

### RTN Operation

An RLL subroutine is executed until a RTN instruction is encountered. When an active RTN is reached in the subroutine, execution is returned to the first instruction following the GTS instruction in the RLL program. The RTN instruction can be either unconditional or conditional. The conditional RTN can be used within a subroutine to satisfy a condition that requires termination of the subroutine. The unconditional RTN must be used as the last instruction in a subroutine.

If the input is off to a conditional RTN instruction, program execution remains with the subroutine.

<b>⚠ WARNING</b>
<p>When you do a run-time edit with TISOFT (Rel 4.2 or later), enter all the instructions required to define a subroutine (END, RTN, SBR, GTS or PGTS/PGTSZ) before setting the controller to RUN mode. Otherwise, the controller changes from RUN to PROGRAM mode and freezes outputs in their current status, which could cause unexpected controller operation.</p> <p>Unexpected controller operation can result in death or serious injury to personnel, and/or equipment damage.</p> <p>Use the TISOFT syntax check function to validate a program before placing the controller in RUN mode. When you do a run-time edit using an earlier release of TISOFT, you must enter the instructions in this order: END, RTN, SBR, GTS or PGTS/PGTSZ.</p>

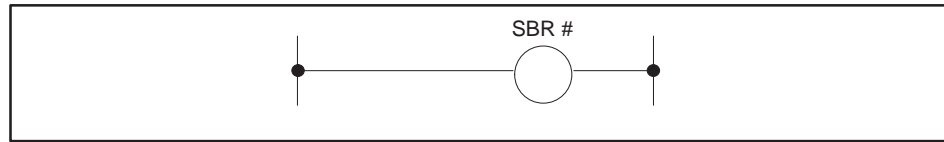
### See Also

These RLL instructions are also used for subroutine operations.

GTS	PGTS	PGTSZ	SBR	SFPGM	SFSUB	XSUB
-----	------	-------	-----	-------	-------	------

## 6.50 Subroutine

**SBR Description** Use the SBR instruction (Figure 6-74) before a set of RLL instructions (the RLL subroutine) to be executed only when they are called by the GTS, PGTS, or PGTSZ instructions.



Field	Valid Values	Function
#	1-255 if called by GTS. 1-32 if called by a PGTS or PGTSZ.	Instruction reference number. Numbers cannot be repeated within a program.

Figure 6-74 SBR Format

### SBR Operation

When the subroutine is called, it executes until either a conditional RTN with power flow or an unconditional RTN is encountered. When this occurs, RLL execution returns to the instruction following the calling (GTS, PGTS, PGTSZ) instruction.

Program subroutines according to the following guidelines.

- Place all subroutines at the end of the main RLL program.
- Separate the main RLL program from the subroutine(s) with an unconditional END instruction.
- A subroutine must be terminated by an unconditional RTN instruction, or a compile error is generated. An END within a subroutine also generates an error.

The unconditional RTN instruction separates a subroutine from a subsequent subroutine.

- You can nest subroutines to the 32nd level. A run-time non-fatal error occurs when this level is exceeded. (Bit 7 in STW1 is set, indicating a stack overflow.)
- When you pass parameters to the subroutine by calling the subroutine from a PGTS instruction, refer to discrete parameters as Bn, and word parameters as Wn, where n = the number of the parameter in the PGTS. See the example in Figure 6-75.

Subroutine (continued)

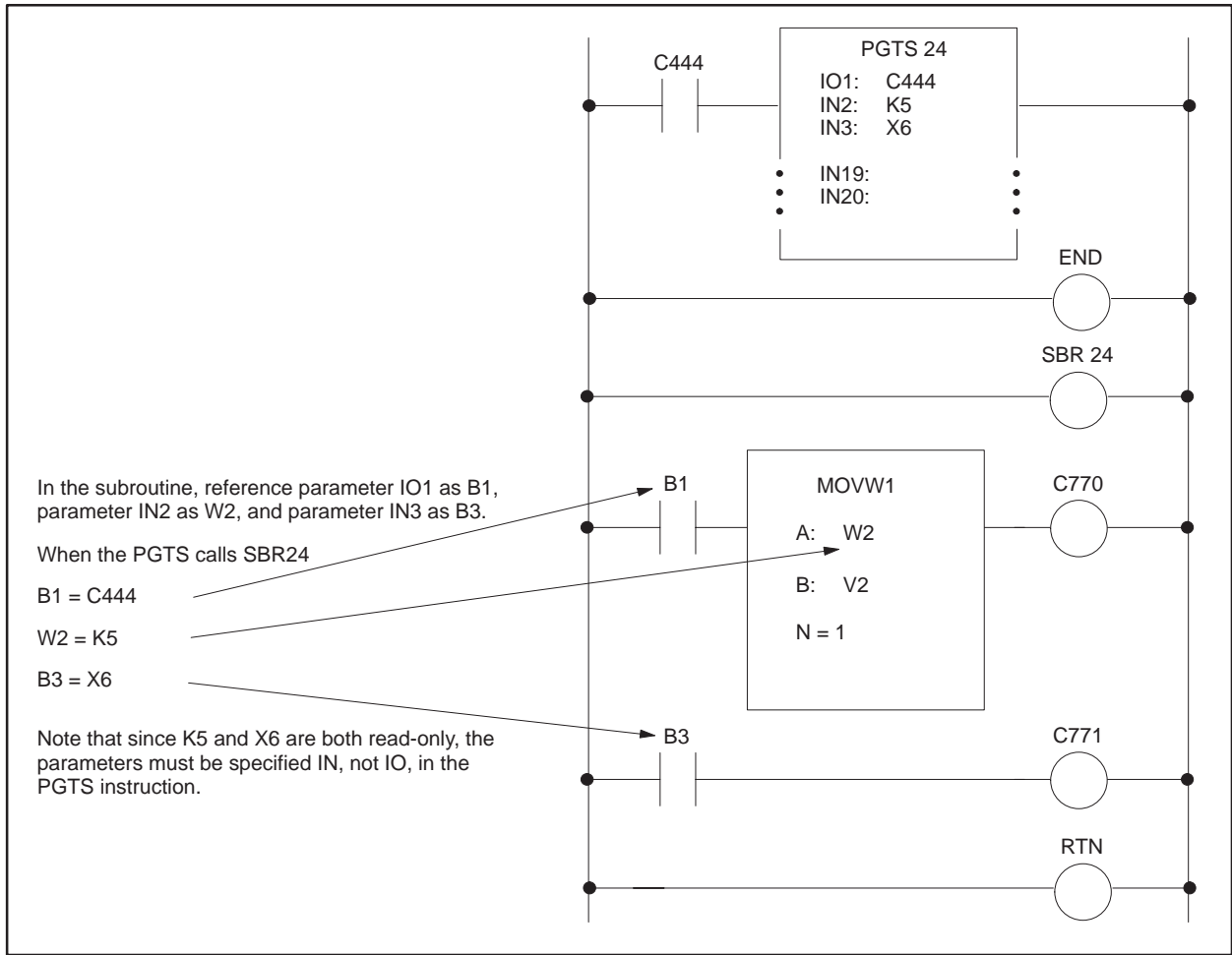


Figure 6-75 SBR Example

**⚠ WARNING**

When you do a run-time edit with TISOFT (Rel 4.2 or later), enter all the instructions required to define a subroutine (END, RTN, SBR, GTS or PGTS/PGTSZ) before setting the controller to RUN mode. Otherwise, the controller changes from RUN to PROGRAM mode and freezes outputs in their current status, which can cause unexpected controller operation.

Unexpected controller operation can result in death or serious injury to personnel, and/or damage to equipment.

Use the TISOFT syntax check function to validate a program before placing the controller in RUN mode. When you do a run-time edit using an earlier release of TISOFT, you must enter the instructions in this order: END, RTN, SBR, GTS or PGTS/PGTSZ.

Note these effects of subroutines on execution of MCRs, JMPs, and SKPs.

- All MCRs and JMPs in a subroutine remain active after a RTN if the instructions within the SBR do not turn them off before the RTN.
- MCRs and JMPs that are active at the time that the subroutine is called, remain active while the SBR is executing.
- A SKP/LBL pair must be defined within the same SBR or a compile error occurs.

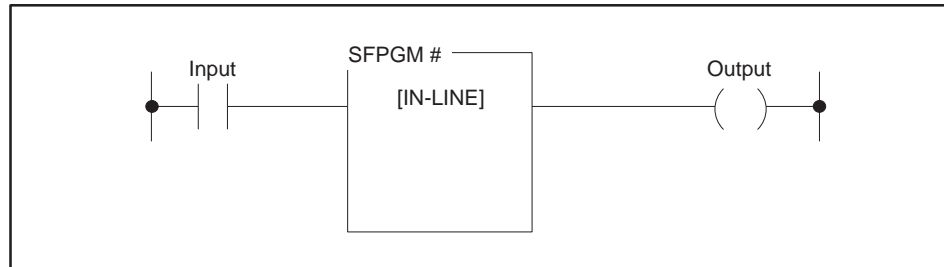
See Also

These RLL instructions are also used for subroutine operations.

GTS	PGTS	PGTSZ	RTN	SFPGM	SFSUB	XSUB
-----	------	-------	-----	-------	-------	------

## 6.51 Call an SF Program

**SFPGM Description** Use the SFPGM instruction (Figure 6-76) to call an SF program for execution.



Field	Valid Values	Function
#	1-1023	Number of the SF program to be called for execution.
IN-LINE*	—	If selected, the SF program executes immediately in-line to the RLL program and its result is available for use in the next rung of the current RLL scan. [SF program type must be priority or non-priority and compiled for in-line execution.]
*In-line execution is available only with controllers that support PowerMath. Refer to Section 7.2 for more information on in-line SFPGM execution.		

Figure 6-76 SFPGM Format

### SFPGM Operation

The RLL SFPGM instruction can be used anywhere within the RLL program that a single-line input box instruction can be used. When a priority/non-priority or cyclic SF program is called by the RLL SFPGM instruction with IN-LINE *not* selected, the SF program is placed in a queue for execution. Up to 32 SF programs of each type (for a total of 96 in three queues) can be queued at a given time. If a queue is full, the request for placement in the queue is made again on the next scan. This continues as long as the input to the RLL SFPGM instruction remains on.

**Priority/Non-Priority SF Programs** When power flow to the RLL SFPGM instruction transitions from off to on, the output from the instruction is examined. If the output is off and the SF program is not executing, the SF program is placed in the queue for execution.

- After the SF program executes, the output turns on.
- The SF program does not execute again until the input to the SFPGM instruction transitions from off to on.

If the controller changes from PROGRAM to RUN mode while the input to the RLL SFPGM instruction is on, the SF program is queued for execution.

**Cyclic Programs** When power flow to the SFPGM instruction transitions from off to on, the cyclic SF program is placed in the queue for execution.

- After the cyclic SF program executes one time, the output turns on. The SF program is automatically re-queued for execution, based on the programmed cycle time. This process continues as long as the input to the RLL SFPGM instruction is on.
- The output remains on until the input to the RLL SFPGM instruction turns off.
- A cyclic SF program is removed from the queue when it completes a scheduled cycle and the SFPGM instruction's input is off.

#### In-line SFPGM Execution

The SFPGM box instruction can be marked for in-line execution if the referenced SF program type is priority or non-priority and has been compiled. Cyclic SF programs cannot be marked for in-line execution.

When power flow is on for an in-line SFPGM, the compiled code for the SF program executes immediately as part of the RLL scan and the output turns on. The result of the box's execution is available to the next element of the current RLL rung. When the input is off, the instruction does not execute, and there is no power flow at the box output.

The following exceptions cause the program's execution to fail:

- If the SF program does not exist or if it has not been marked as compiled, user program error 8 is logged in STW200 and there is no power flow at the output.
- If the SF program is not enabled, user program error 9 is logged in STW200 and there is no power flow at the output.
- If the SF program's type is CYCLIC or RESTRICTED, user program error 10 is logged in STW200 and there is no power flow at the output.
- If an edit operation is in progress, user program error 11 is logged in STW200 and there is no power flow at the output.
- If the SFPGM instruction is being executed by an interrupt RLL task (555 specific), user program error 12 is logged in STW200 and there is no power flow at the output.

#### See Also

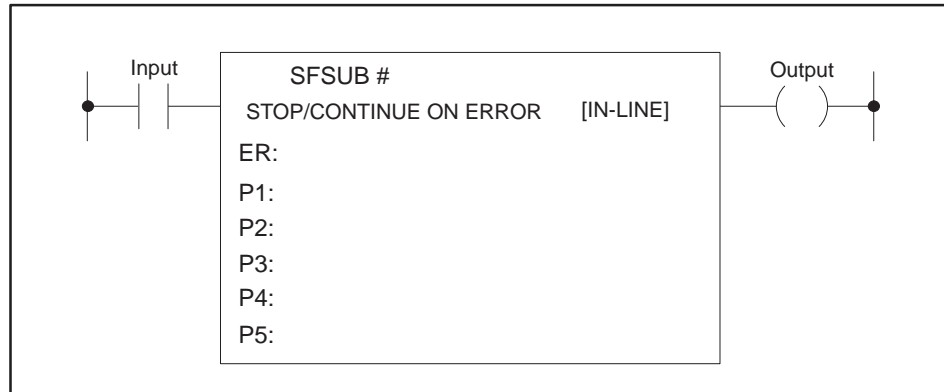
These RLL instructions are also used for subroutine operations.

GTS	PGTS	PGTSZ	RTN	SBR	SFSUB	XSUB
-----	------	-------	-----	-----	-------	------



## 6.52 Call SF Subroutines from RLL

**SFSUB Description** Use the SFSUB instruction, (shown in Figure 6-77) to call an SF subroutine for execution.



Field	Valid Values	Function
#	0 – 1023	If 1 – 1023, the number of the SF subroutine to be called for execution. If 0, then only the instruction parameters will be evaluated.
STOP/ CONTINUE ON ERROR		Select STOP ON ERROR if you want the SF Subroutine to terminate if an error is detected. Select CONTINUE ON ERROR if you want the SF Subroutine to continue, e.g., you want to handle errors within the subroutine.
IN-LINE*	—	If selected, the SF subroutine executes immediately in-line to the RLL program and its result is available for use in the next rung of the current RLL scan. [SF subroutine must be compiled for in-line execution.]
ER	C, Y, WY, V	Designates a single C or Y bit, or the first word of a 3 word area in WY or V Memory, where the error status will be written if an error occurs during parameter evaluation or during execution of the SF subroutine. Refer to Section 7.8, Reporting SF Program or SFSUB RLL Instruction Errors, for a description of the ER parameter.
Pn	Constant; any readable bit, word, or expression	Designates parameters to be evaluated and if # is 1 – 1023, it is passed to the SF subroutine. Up to five parameters may be specified; they must be specified in order; i.e., P entries must not be skipped.
*In-line execution is available only with controllers that support PowerMath. Refer to Section 7.2 for more information on in-line SFSUB execution.		

Figure 6-77 SFSUB Format

---

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

---

When the # is 0, only the instruction parameters are evaluated (this variety is called an SFSUB 0). You can use an SFSUB 0 to execute up to five expressions without calling an actual SF subroutine or program. The programming device may limit the length of the expression that can be placed into the P fields.

Multiple SFSUB instructions with the same value of # can be used in your program, since your application may require multiple accesses to the same SF subroutine but with different parameters for each access.

A variable in the P fields can be one of the following data types:

- Constant – Any integer or real number.
- Discrete or word element – An element is comprised of a data type and a number. A period following the element designates the element as an address of a real number. The absence of a period designates the element as an address of an integer.

Examples are V100, V252., C101, etc.

- Expression – An expression is a logical group of tokens evaluating to an address or a value, where a token is the smallest indivisible unit, e.g., an element address, operator, constant, parenthesis, etc. Refer to Section 7.9 for details on expressions.

Examples are V101.:=V65. + 14.2 and LSP1.:= V14. +K19.

### SFSUB Operation

The RLL SFSUB instruction can be used anywhere within the RLL program that a large box instruction, such as a drum, can be used. When power flow to the RLL SFSUB instruction transitions from off to on, the output from the RLL SFSUB instruction is examined to determine subsequent actions.

If the instruction is not currently executing, then the instruction is placed in one of the SFSUB queues for execution. There are two SFSUB execution queues, one to handle SFSUB 0 instructions and the other to handle all other SFSUB instructions.

---

When an SFSUB 0 instruction is pulled from its execution queue, the instruction parameters are evaluated and the instruction output turns on. When SFSUB instructions are pulled from the other execution queue, the instruction parameters are evaluated, statements in the corresponding SF subroutine are executed, and the instruction output turns on.

Upon completion of the SFSUB instruction, the instruction output remains on until the input turns off.

#### In-line SFSUB Execution

The SFSUB box instruction can be marked for in-line execution if the referenced SF subroutine has been marked as compiled.

When power flow is on for an in-line SFSUB, the SFSUB's compiled parameter evaluation code is executed and then, if the subroutine number is not 0, the compiled code for the subroutine is executed and the output turns on. The result of the box's execution is available to the next element of the current RLL scan. When the input is off, the instruction does not execute, and there is no power flow at the box output.

The following exceptions cause an in-line SFSUB instruction to fail prior to parameter evaluation:

- An edit operation is in progress. User program error 11 is logged in STW200 and there is no power flow at the output.
- The SFSUB statement is being executed by an interrupt RLL task. User program error 12 is logged in STW200 and there is no power flow at the output.

The following exceptions cause the SFSUB instruction to fail after parameter evaluation:

- The referenced SF subroutine does not exist.
- The referenced SF subroutine has not been compiled.
- The referenced SF subroutine is not enabled.

In each of these cases, the output turns on. These errors are logged in the SFSUB instruction's Error Status Address and the SF subroutine is not executed.

---

See Also

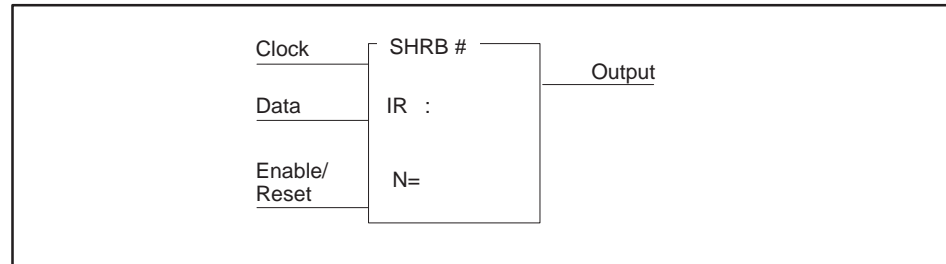
These RLL instructions are also used for subroutine operations.

GTS	PGTS	PGTSZ	RTN	SBR	SFPGM	XSUB
-----	------	-------	-----	-----	-------	------

## 6.53 Bit Shift Register

### SHRB Description

The Bit Shift Register instruction (Figure 6-78) creates a bit shift register using a specified number of control relays or points in the discrete image register. The shift register may be up to 1023 bits long.



Field	Valid Values	Function
#	Varies with controller model	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of memory discussed on page 4-8 in Section 4.2.
IR	Y, C, B	Lowest numbered control relay or location in the discrete image register into which the data is shifted.
N	1-1023	Size of the shift register (number of bits).

Figure 6-78 SHRB Format

**NOTE:** If you plan to use this instruction in a subroutine (using B-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**SHRB Operation**

The operation of the bit shift register follows.

- When the Enable/Reset turns on, the SHRB box is enabled.
- When the clock transitions from zero to one, the following actions occur.  
The last (highest numbered) bit of the shift register moves to the output.  
The data in the shift register shifts one address.  
The status of the Data input (0 or 1) moves into the lowest numbered point, as specified in the IR field.
- When the clock does not transition from zero to one, the last bit of the shift register moves to the output. The data does not shift.
- The Enable/Reset must be kept on as long as data are to be shifted into, and kept in, the SHRB. When the Enable/Reset loses power flow, the SHRB clears; i.e., all control relays or image register points comprising the SHRB clear to 0.
- If the Enable/Reset does not receive power flow, the instruction does not execute and the output does not turn on.

The example in Figure 6-79 shows the status of the shift register on two consecutive scans.



Scan	Data Input	Clock	Shift Register					Output
			Y1	Y2	Y3	Y4	Y5	
N	1 or 0	—	0	1	0	0	1	1
N + 1	1		1	0	1	0	0	1
N + 2	1 or 0	 or —	1	0	1	0	0	0

Figure 6-79 SHRB Example

**See Also**

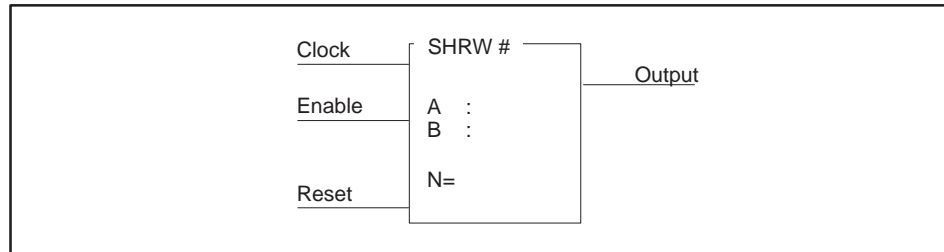
These RLL instructions are also used for electro-mechanical replacement.

Contacts	Coils	CTR	DCAT	DRUM	EDRUM
JMP	MCAT	MCR	MDRMD	MDRMW	NOT
SKP/LBL	TMR	UDC			

Refer to Section E.1 for an application example of the bit shift register.

## 6.54 Word Shift Register

**SHRW Description** The Word Shift Register instruction (Figure 6-80) copies words from a memory location into a shift register. The shift register is located in V-Memory and can be up to 1023 words long.



Field	Valid Values	Function
#	Varies with controller model	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of memory discussed on page 4-8 in Section 4.2.
A	Any readable word	Memory location of the word to be copied into the shift register.
B	V, W, (G, VMS, VMM, 575)	Starting address for the shift register.
N	1–1023	Size of the shift register (number of words).

Figure 6-80 SHRW Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**SHRW Operation** The operation of the SHRW is described below and shown in Figure 6-81.

- The Enable and Reset inputs must both be on for the SHRW box to execute.
- When the Clock transitions from off to on, the word currently in memory location A shifts into the shift register at the memory location specified by B. The shift occurs as follows.

Word  $B+(N-1)$  is discarded.

Word  $B+(N-2)$  is then copied to word  $B+(N-1)$ ; word  $B+(N-3)$  is copied to word  $B+(N-2)$ , etc.

Word B is copied to word  $B+1$ ; word A is copied to word B.

- After each shift is completed, the output turns on for one scan.
- If the Enable turns off, but the Reset remains on, all words currently in the SHRW are retained, but no words are shifted.
- If the Reset turns off, all words in the shift register clear to zero. The instruction does not execute, and there is no power flow at the box output.

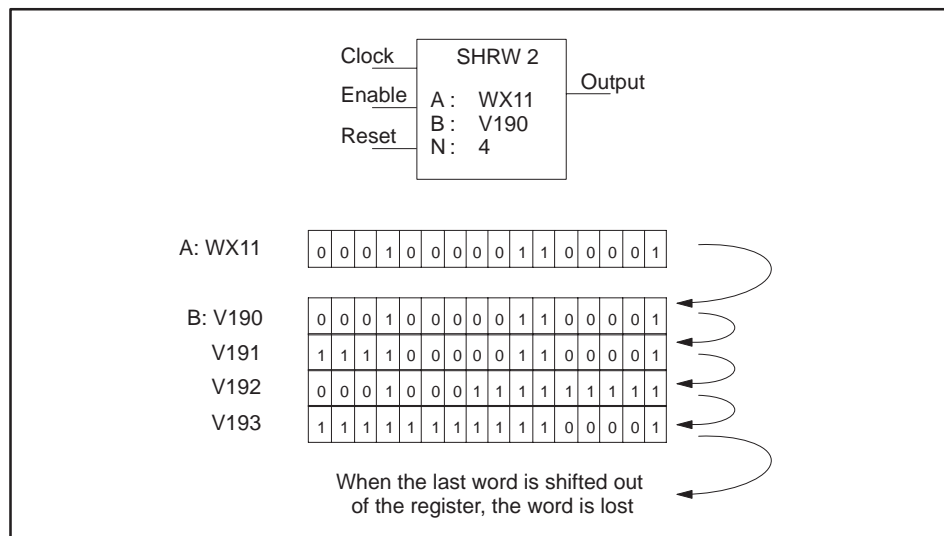


Figure 6-81 SHRW Operation

See Also

These RLL instructions are also used for word moves.

LDA	LDC	MIRW	MOVE	MOVW	MWFT
MWI	MWIR	MWTT			

Refer to Section E.2 for an application example of the SHRW.



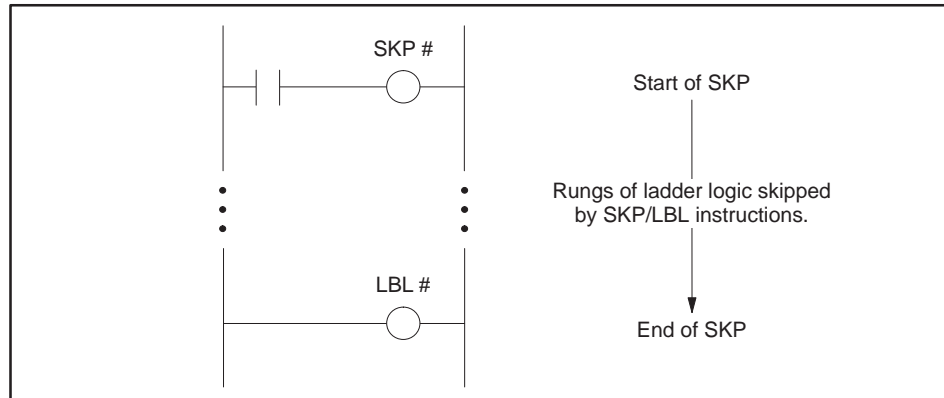
## 6.55 Skip / Label

---

### SKP / LBL Description

The SKP and LBL instructions (Figure 6-82) provide a means of enabling or disabling segments of a program during a scan. These instructions are often used when duplication of outputs is required, and those outputs are controlled by different logic. These instructions can be used to decrease scan time since the instructions between any active SKP and LBL instructions do not execute.

- SKP and LBL must be used together. The LBL must appear before the instruction that terminates the current program segment (TASK, END, or RTN).
- If you use an RLL subroutine (controllers 545, 555, and 575), you can use up to 255 SKP/LBL instructions within each subroutine and up to 255 SKP/LBL instructions for each TASK segment in the program.
- The reference numbers for the subroutine SKP/LBL instructions range from 1–255, and numbers cannot be duplicated within a given subroutine or TASK segment.
- The subroutine is distinct from the main RLL program, and reference numbers used in the subroutine can also be used in the main program. That is, a SKP23 in the main program does not interfere with a SKP23 in the subroutine.



Field	Valid Values	Function
#	1-255	Instruction reference number. Same number must be used for a SKP and its associated LBL. Numbers cannot be repeated, except for the 545, 555, and 575 that do allow numbers to be repeated.

Figure 6-82 SKP / LBL Format

## Skip / Label (continued)

---

- SKP / LBL Operation**    The operation for the skip and label instructions is described below.
- The SKP and the LBL instructions must be used together for the SKP to be executed.
    - For the 545, 555, and 575, a SKP without a LBL generates a compile error.
    - For other controllers, either instruction appearing without the other is ignored.
  - When the SKP receives power flow, all ladder logic between the SKP and its associated LBL is ignored by the controller. Outputs between the SKP and the LBL are frozen, i.e., their current status in the image register is unchanged.
  - All ladder logic within the SKP zone of control executes normally when the SKP does not have power flow.
  - For a SKP to LBL function located within the zone of control of an MCR or JMP, the SKP to LBL function overrides the MCR or JMP when the SKP has power flow.
  - The zone of control for a SKP is limited to the task segment or subroutine in which the SKP is used. That is, the matching LBL must be defined after the SKP and be located in the same task segment or subroutine as the SKP.
  - For a JMPE or MCRE contained within a SKP's zone of control, the program functions as if the JMPE or MCRE is located at the end of the program whenever the SKP is active.

### **WARNING**

**If you do not enter the LBL and SKP instructions in the correct order, the controller changes from RUN to PROGRAM mode and freezes outputs in their current status, which could cause unexpected controller operation.**

**Unexpected controller operation could result in death or serious injury to personnel, and/or equipment damage.**

**When you do a run-time edit with TISOFT (Rel 4.2 or later), enter the LBL instruction before setting the controller to RUN mode; also, use the TISOFT syntax check function to validate a program before placing the controller in RUN mode. When you do a run-time edit using an earlier release of TISOFT, you must enter the instructions in this order: LBL, then SKP.**

**NOTE:** When a SKP is active, timers between the SKP and its LBL do not run. Use care in the placement of timer instructions (TMR, DCAT, and MCAT) and drum instructions (DRUM, EDRUM, MDRMD, and MDRMW) if they are to continue operation while a SKP is active.

The operation of the SKP and LBL instructions is illustrated in Figure 6-83. In this example, SKP5 is located on rung A. When the SKP has power flow, the ladder logic within its zone of control (rungs B and C) does not execute.

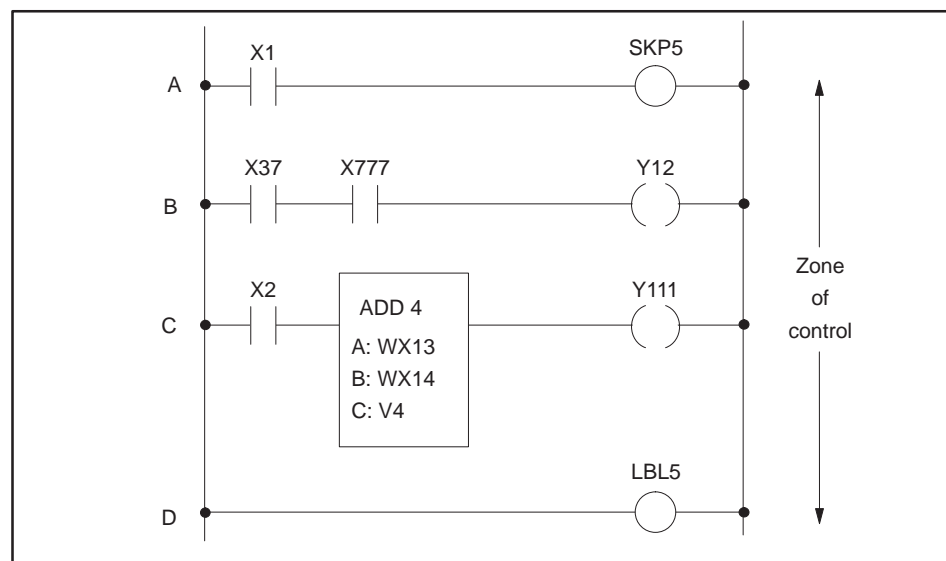


Figure 6-83 Example of SKP Zone of Control

See Also

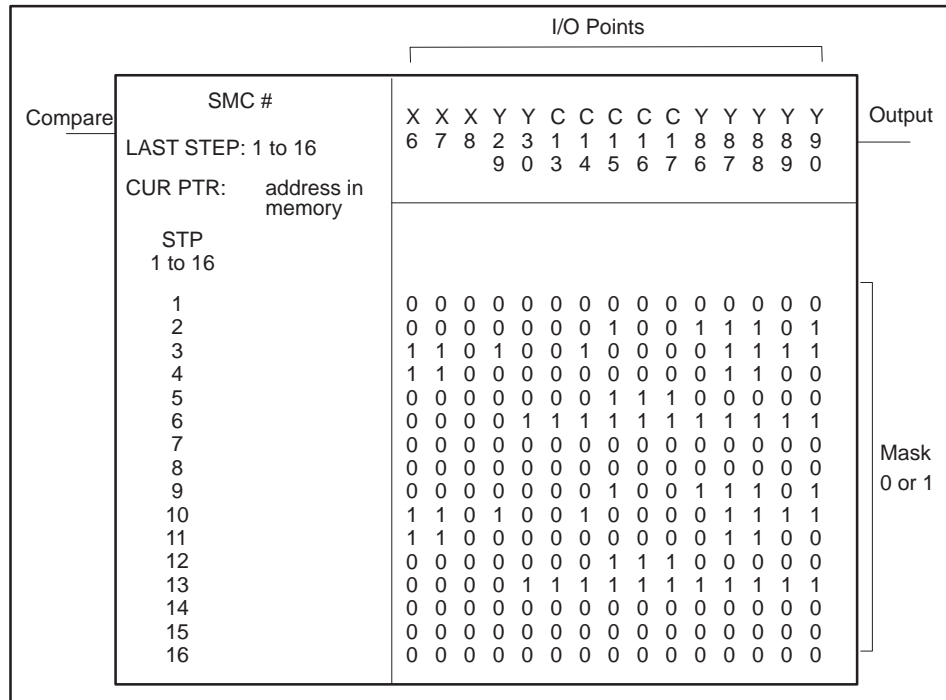
These RLL instructions are also used for electro-mechanical replacement.

Coils	Contacts	CTR	DCAT	DRUM	EDRUM
JMP	MCAT	MCR	MDRMD	MDRMW	NOT
SHRB	TMR	UDC			

## 6.56 Scan Matrix Compare

### SMC Description

The Scan Matrix Compare instruction (Figure 6-84) compares up to 16 predefined bit patterns to the current states of up to 15 discrete points. If a match is found, the step number that contains the matching bit pattern is entered into the memory location specified by the pointer, and the output is turned on.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
LAST STEP	1-16	Specifies last instruction step to be scanned for a match.
CUR PTR	V, G, W, VMS, VMM	Memory location that holds the step number where a match is found, or zero if no match is found.
I/O Points	X, Y, C, B, or blank	The discrete points to be compared to the step mask.

Figure 6-84 SMC Format

**SMC Operation**

The SMC operation is described below.

- The instruction executes when the Compare input is on.  
If the Compare input remains on, the SMC instruction checks all programmed steps on every scan.
- The status of up to 15 discrete points is checked against the predefined bit patterns.
- If a match is found, the step number of the matching mask is entered into the memory location specified by CUR PTR, and the output turns on.
- If no match is found, CUR PTR is cleared to 0, and the output turns off.

If the Compare input is off, the instruction does not execute, and there is no power flow at the box output. The CUR PTR retains its last value.

**See Also**

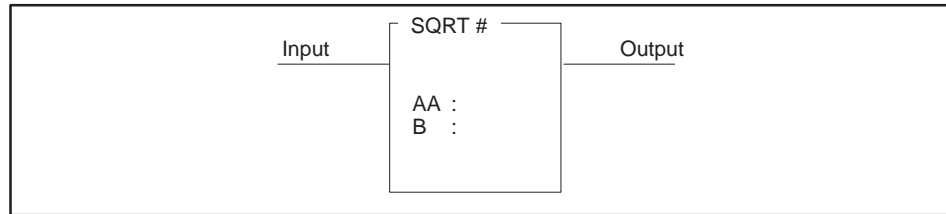
These RLL instructions are also used for bit manipulation.

BITC	BITS	BITP	IMC	WAND	WOR
WROT	WXOR	Bit-of-Word Contact/Coil			

6.57 Square Root

SQRT Description

The Square Root instruction (Figure 6-85) finds the integer square root of a 32-bit (long word) positive integer stored in memory locations AA and AA + 1. The result is stored in memory location B.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
AA	Any readable word	Specifies integer of which square root is taken. This is a long word. AA holds the 16 most significant bits, and AA + 1 holds the 16 least significant bits. Range: $0 \leq AA \leq (32,767)^2$
B	Any writeable word	Memory location for the result.

Figure 6-85 SQRT Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**NOTE:** The answer to the square root function can have large margins of error because this is integer math and the answer is truncated.

---

**SQR Operation**

When the input is on, the SQR box executes. If the input remains on, the operation is executed on every scan. The operation of the SQR follows:

$$B = \sqrt{AA}$$

- If the result of the square root is not an integer, SQR reports only the integer portion of the root. For example, although the square root of 99 is 9.95, the SQR function reports a square root of 9.
- The operation is valid if  $0 \leq AA \leq (32,767)^2$ .
- If the result is valid, the output turns on when the operation executes. Otherwise it turns off, and the contents of B do not change.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

**See Also**

These RLL instructions can also be used for math operations.

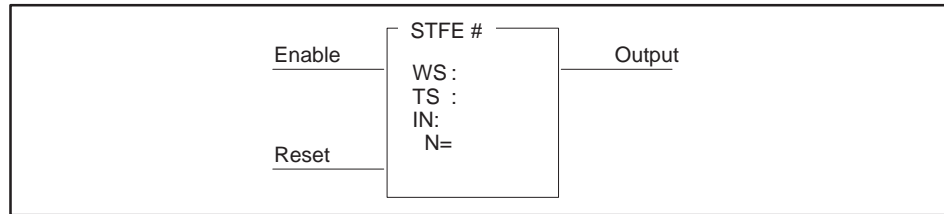
ABS	ADD	CMP	DIV	MULT	SUB
Relational Contact					



## 6.58 Search Table For Equal

### STFE Description

The Search Table For Equal instruction (Figure 6-86) locates the next occurrence of a word in a table that is equal to a source word. The position of the matching word is shown by an index.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
WS	Any readable word	Memory location of the source word.
TS	Any readable word	Starting address of the table.
IN	V, G, W, VMS, VMM	Specifies memory location where the index is stored. The index specifies the next word in the table to be compared with the source word.
N	1-256	Specifies length of the table.

Figure 6-86 STFE Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

### STFE Operation

The operation of the STFE is described below.

- You must turn off the Reset to initialize the index, setting it to -1.
- You must turn on the Reset before the STFE can operate.
- When the Enable turns on, the index increments by one and specifies the next word in the table to be compared with the source word. The value contained by the index ranges from 0 to N-1 while the STFE executes. N is the length of the table.
- The source word WS and the word in the table TS specified by the index are compared.

- If the two words are equal, the STFE output turns on for one scan and then turns off.

The index contains the position of the matching word in the table for the duration of this scan. The contents of the index must be used or saved during this scan since the STFE looks for the next match on the next scan as long as the Enable and Reset remain on.

- If the two words are not equal, the index increments by one and the next word in the table is compared to the source word.
- If no matches are found in the table, the output remains off. The index contains the position of the last word in the table.
- The entire table is searched during one scan until one match or no match is found.
- If the Enable turns off while the Reset is on, the index holds its current value. If the Reset turns off, the index resets to -1.
- After the entire table has been searched, i.e., the output is off and the index = N-1, the STFE must be reset (Reset turns off) in order to be executed again.

If the Reset is off, the instruction does not execute, and there is no power flow at the box output.

#### See Also

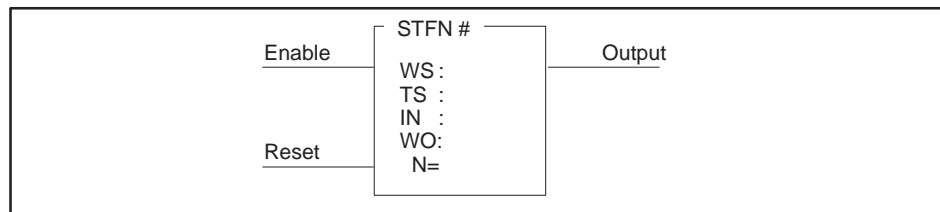
These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFN	TAND	TCPL	TOR
TTOW	TXOR	WTOT	WTTA	WTTO	WTTXO

## 6.59 Search Table For Not Equal

### STFN Description

The Search Table For Not Equal instruction (Figure 6-87) locates the next occurrence of a word in a table that is not equal to a source word. The position of the non-matching word is shown by an index, and the value of the non-matching word is copied into a specified memory location.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
WS	Any readable word	Memory location of the source word.
TS	Any readable word	Starting address of the table.
IN	V, G, W, VMS, VMM	Specifies memory location where the index is stored. The index specifies the next word in the table to be compared with the source word.
WO	Any writeable word	Memory location to which the non-matching word is written.
N	1–256	Specifies length of the table.

Figure 6-87 STFN Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

### STFN Operation

The operation of the STFN is described below.

- You must turn off the Reset to initialize the index, setting it to –1.
- You must turn on the Reset before the STFN can operate.
- When the Enable turns on, the index increments by one and specifies the next word in the table to be compared with the source word. The value contained by the index ranges from 0 to N–1 while the STFN executes. N is the length of the table.
- The source word WS and the word in the table TS specified by the index are compared.

- If the two words are not equal, the STFN output turns on for one scan and then turns off. The value of the non-matching word is copied into another memory location specified by WO.

The index contains the position of the non-matching word in the table for the duration of this scan. The contents of the index must be used or saved during this scan since the STFN looks for the next match on the next scan as long as the Enable and Reset remain on.

- If the two words are equal, the index increments by one and the next word in the table is compared to the source word.
- If no mismatches are found in the table, the output remains off. The index contains the position of the last word in the table.
- The entire table is searched during one scan until one mismatch or no mismatch is found.
- If the Enable turns off while the Reset is on, the index holds its current value. If the Reset does turn off, the index resets to -1.
- After the entire table has been searched, i.e., the output is off and the index = N-1, the STFN must be reset (Reset turns off) in order to be executed again.

If the Reset is off, the instruction is not executed, and there is no power flow at the box output.

See Also

These RLL instructions are also used for table operations.

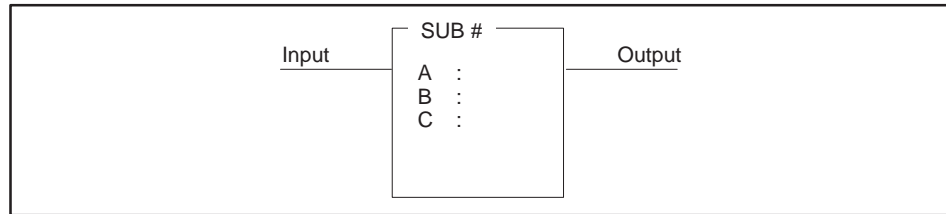
MIRFT	MIRTT	STFE	TAND	TCPL	TOR
TTOW	TXOR	WTOT	WTTA	WTTO	WTTXO

SUB

## 6.60 Subtract

### SUB Description

The Subtract instruction (Figure 6-88) subtracts a signed integer in memory location B from a signed integer in memory location A, and stores the result in memory location C.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
A	Any readable word	Memory location for the minuend (a word), the number from which a value is subtracted.
	or constant (-32768 to +32767)	Value of the minuend if a constant is used. A and B cannot both be constants.
B	Any readable word	Memory location for the subtrahend (a word), the number that is subtracted.
	or constant (-32768 to +32767)	Value of the subtrahend if a constant is used. A and B cannot both be constants.
C	Any writeable word	Memory location for the result (a word).

Figure 6-88 SUB Format

### SUB Operation

When the input is on, the SUB box executes. If the input remains on, the instruction executes on every scan. The operation executed is  $C = A - B$ .

If  $-32768 \leq \text{result} \leq 32767$ , then the output turns on. Otherwise, the output turns off, and the least significant (16 bits) of the result are stored in C.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

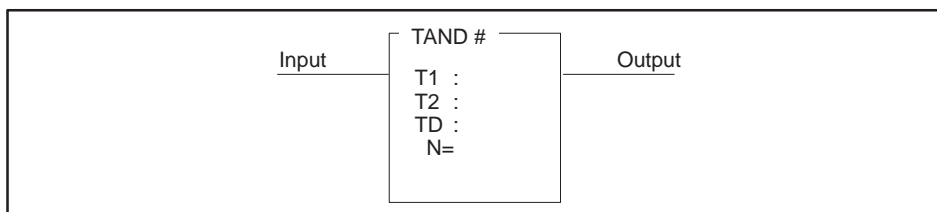
### See Also

These RLL instructions can also be used for math operations.

ABSV	ADD	CMP	DIV	MULT	SQRT
Relational Contact					

## 6.61 Table to Table AND

**TAND Description** The Table to Table AND instruction (Figure 6-89) ANDs the corresponding bits in two tables and places the results in a specified third table. If both bits are 1s, then the resultant bit is set to 1. Otherwise, the resultant bit is set to 0.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
T1	Any readable word	Starting address for the first table.
T2	Any readable word	Starting address for the second table.
TD	Any writeable word	Starting address for the destination table. TD can be the same as T1 or T2, or be different.
N	1–256	Specifies table length. All tables are N words long.

**Note:** If you plan to use this instruction in a subroutine, refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

Figure 6-89 TAND Format

**TAND Operation** The operation of the TAND follows.

- When the input turns on, a comparison is made between each bit of each word in the first (T1) and second (T2) tables.
- Each pair of bits is ANDed, and the resultant bit is placed in the third table (TD). If both bits are 1s, then the resultant bit is set to 1. Otherwise, the resultant bit is set to 0.
- The bits in all the words of the two tables are ANDed each scan.
- The output turns on when the instruction executes.

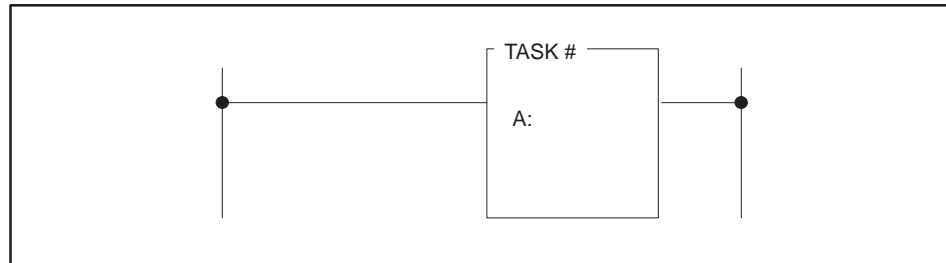
If the input is off, the instruction does not execute, and there is no power flow at the box output.

**See Also** These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFE	STFN	TCPL	TOR
TTOW	TXOR	WTOT	WTTA	WTTO	WTTXO

## 6.62 Start New RLL Task

**TASK Description** Use the TASK instruction (Figure 6-90) to delimit the main (I/O synchronous) RLL task and the cyclic RLL task.



Field	Valid Values	Function
#	1, 2, 8	Designates task. 1 = normal RLL task; 2 = cyclic RLL task; 8 = interrupt RLL task
A:	0–32767 or any readable word that contains 0–65535.	Specifies cycle time in milliseconds. All segments for a TASK2 are executed within the cycle time specified in the TASK instruction for the first TASK2 segment. Values specified in A for subsequent segments are ignored. For TASK2, a value of 0 indicates that default (10) is used. A must set to 0 for TASK1 and TASK8. The data file will not be displayed for TASK1 and TASK8 except during edit.

Figure 6-90 TASK Format

**TASK Operation** The operation of the TASK is described below.

- The TASK<sub>n</sub> instruction indicates that the RLL instructions that follow it comprise an RLL task segment, where n = 1 designates segments of the main RLL task, n = 2 designates segments of the cyclic RLL task, and n = 8 designates segments of the interrupt task. Refer to Figure 6-91a.

Task 1 is assumed when the first rung does not contain a TASK instruction. A task can consist of multiple segments, each preceded by a TASK instruction. The segments do not have to be contiguous (Figure 6-91b). Terminate an RLL task with another TASK instruction or with the END instruction.

- TASK2 is executed with a higher priority than TASK1. Therefore, normal RLL execution is interrupted by a cyclic RLL task.
- TASK8 is executed with a higher priority than TASK1 or TASK2. Therefore, both the normal RLL and the cyclic RLL are interrupted by a configured I/O interrupt.

- If you specify the cycle time A for a TASK2 task as a readable word, you can change the cycle time on a cycle-by-cycle basis. When A = 0, the default time of 10 ms is used.

## ! CAUTION

**Use caution** in determining the time requirements for a cyclic task.

**As the ratio of execution time to cycle time approaches 1:1, the risk increases that the main RLL task reports a scan watchdog Fatal Error, causing the controller to enter the Fatal Error mode, freeze analog outputs and turn off discrete outputs, which could lead to equipment failure.**

**You need to assess the time requirements for a cyclic task with care.**

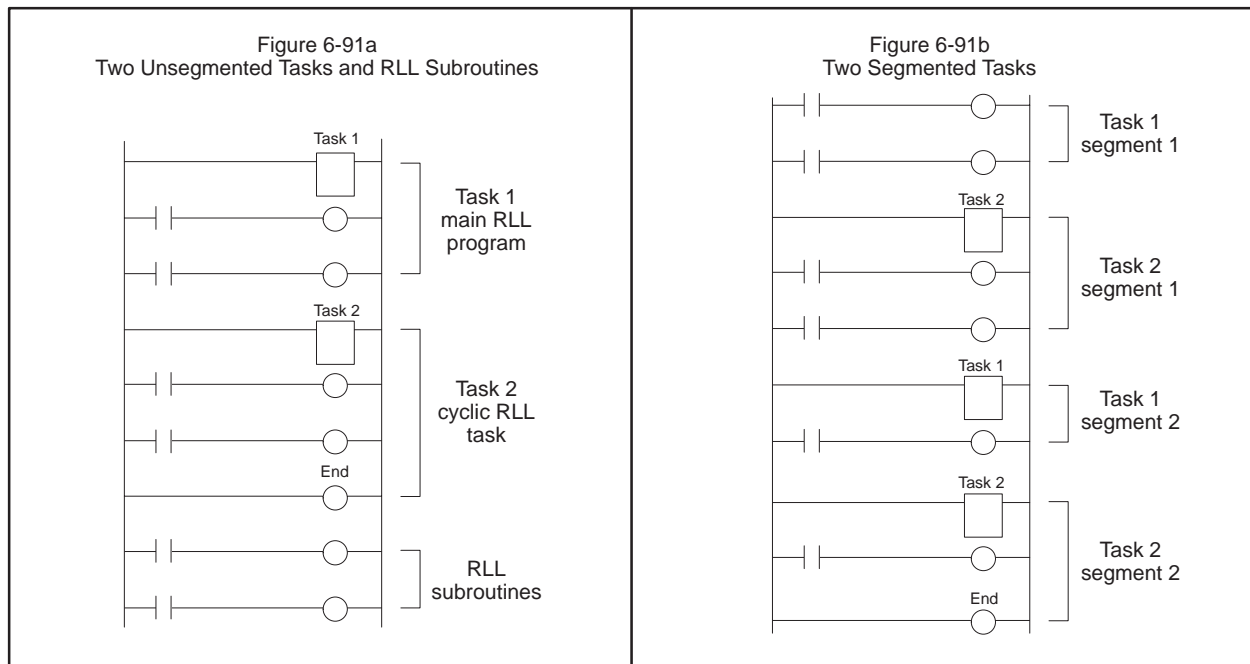


Figure 6-91 Examples of TASK Design



## Start New RLL Task (continued)

---

- When the normal RLL task fails to complete execution within the specified cycle time, bit 1 is set in STW219, and bit 14 is set in STW1 on the next TASK1 scan. When the cyclic RLL task fails to complete execution within the specified cycle time, bit 2 is set in STW219 on the next TASK2 scan. When a cyclic task overruns, the cycle on which the overrun is detected, is skipped. For example, a 3-ms task that overruns then executes at a 6-ms cycle rate.

You can display the peak execution time for a task using an operator interface and specifying TPET1 for TASK1 and TPET2 for TASK2.

- You can call any subroutine from a task and the normal subroutine nesting rules apply. Call a given subroutine from only one task. Subroutines are not re-entrant, and subroutine execution initiated by one task interferes with subroutine execution initiated by a second task.

### See Also

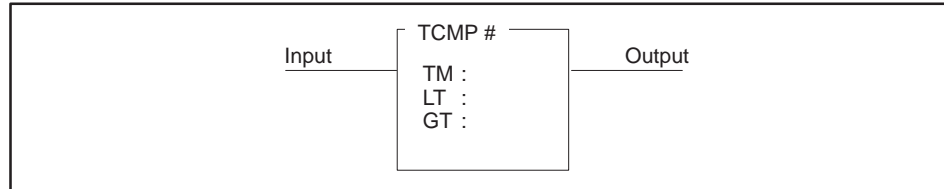
These RLL instructions can also be used for immediate I/O applications.

Immediate Contact/Coil	Immediate Set/Reset Coil	IORW
------------------------	--------------------------	------

Refer to Section 3.3 for more information about using TASK in a program.

## 6.63 Time Compare

**TCMP Description** The Time Compare instruction (Figure 6-92) compares current time in the real-time clock with values in the designated V-Memory locations.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
TM	V, G, W, VMS, VMM	Specifies the memory locations containing time to be compared to time in real-time clock. $V(TM) = \text{Hour} - \text{BCD}^* 0000-0023.$ $V(TM+1) = \text{Minute} - \text{BCD}^* 0000-0059.$ $V(TM+2) = \text{Second} - \text{BCD}^* 0000-0059.$ Enter the hexadecimal value of 00FF for any of the fields (hour, minute, second, etc.) that you want to exclude from the compare operation.
LT	Y, C, B, or blank	Bit turned on when time represented in TM locations < the real-time value in the clock.
GT	Y, C, B, or blank	Bit turned on when time represented in TM locations > the real-time value in the clock.
<b>Note:</b> If you plan to use this instruction in a subroutine, refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.		

\* BCD values are entered using the HEX data format.

Figure 6-92 TCMP Format

**TCMP Operation** When there is power flow to the input of the TCMP instruction, the current hours, minutes, and seconds in the real-time clock are compared to the values in the designated memory locations.

If a match occurs, the output of the instruction turns on. If the time represented by the memory locations is less than the real-time value in the clock, the bit designated by LT turns on. If the time represented by the memory locations is greater than the real-time value in the clock, the bit designated by GT turns on.

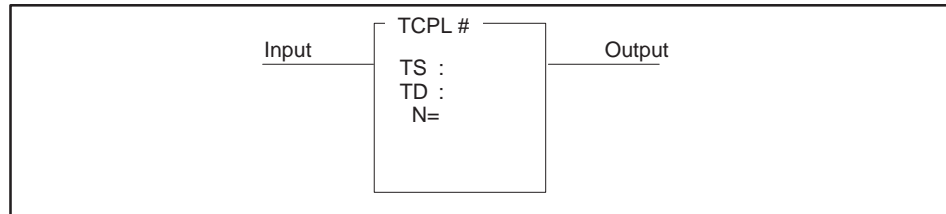
When the input is off, the comparison does not execute and there is no power flow at the box output.

**See Also** These RLL instructions can also be used for date/time functions.

DCMP	DSET	TSET
------	------	------

## 6.64 Table Complement

**TCPL Description** The Table Complement (Figure 6-93) inverts the status of each bit in a table and places the results in another specified table.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
TS	Any readable word	Starting address of the table containing the bits to be inverted.
TD	Any writeable word	Starting address of the destination table. TD can be the same as TS or be different.
N	1-256	Specifies length for both tables.

Figure 6-93 TCPL Format

**TCPL Operation** The operation of the TCPL is described below.

- When the input turns on, each bit in the source table specified by TS inverts and stores in the destination table specified by TD.  
0 inverted is 1; 1 inverted is 0.
- The bits in all the words of the table are inverted each scan.
- The output turns on when the instruction executes.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

**See Also** These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFE	STFN	TAND	TOR
TTOW	TXOR	WTOT	WTTA	WTTO	WTTXO

## 6.65 Text

---

### Text Box Description

The Text box allows you to place textual information, such as copyright, software version, or other text into your RLL program. The instruction forms a single network and takes no action. The Text Box's sole purpose is for documentation.

The text box (Figure 6-94) can hold up to five lines of 40 characters each. Characters allowed in the text box are: A through Z, 0 through 9, space, and printable special characters.

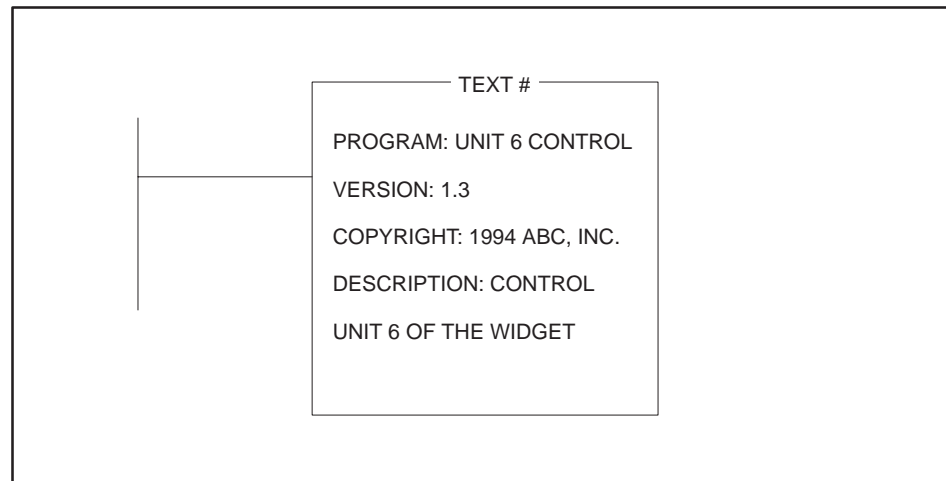


Figure 6-94 Text Box Format

## 6.66 Timer

TMR/TMRF  
Description

The Timer instruction (Figure 6-95) is used to time events. The timer output turns on after the timer times down, making this an “on delay” timer. A fast timer is denoted by the mnemonic TMRF; a slow timer is denoted by TMR.

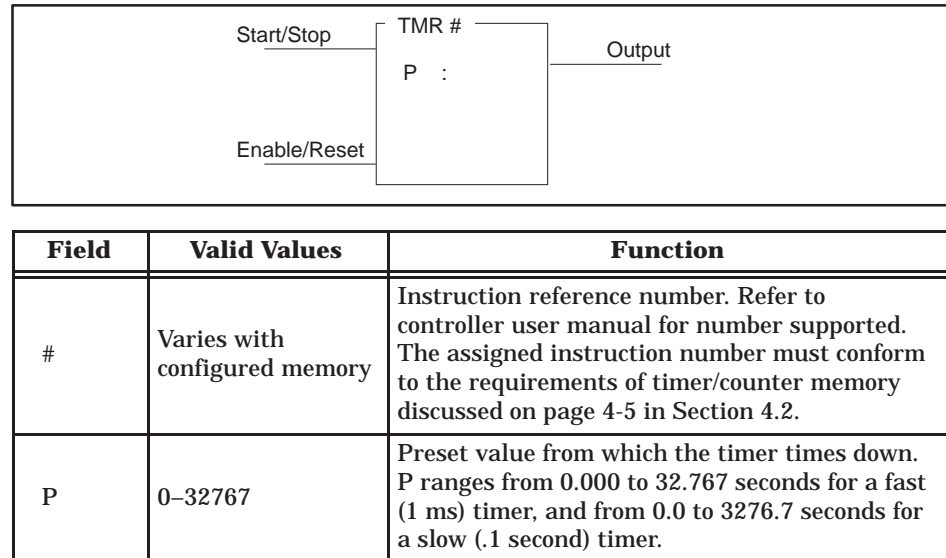


Figure 6-95 TMR/TMRF Format

TMR/TMRF  
Operation

The timer times down from the value specified in the preset, P. The preset is stored in TCP-Memory. The timer’s current time is stored in TCC-Memory.

- The Enable/Reset must be on for the timer to operate.
- When the Start/Stop input is on and the Enable/Reset is on, the timer begins to time down.
- Timing begins at the preset value P and continues down to zero.
- If the Start/Stop input turns off and the Enable/Reset input remains on, the timer stops but it saves the current value, TCC. If the Start/Stop input turns on again, the timer resumes timing.

TCC is also saved if the Enable/Reset input is on and a loss of power occurs, provided the controller battery backup is enabled.

- If the Enable/Reset input turns off, the timer resets to the preset time specified in P.
- The output turns on when the timer reaches zero, and it stays on until the timer resets; i.e., the Enable/Reset input turns off.

If the Enable/Reset does not receive power flow, the instruction does not execute and the output does not turn on.

#### Using the Timer Variables

You can use other RLL instructions to read from or write to the timer variables. You can also use an operator interface to read or write to the timer variables. While you are programming the timer, you are given the option of protecting the preset values from changes made with an operator interface.

#### See Also

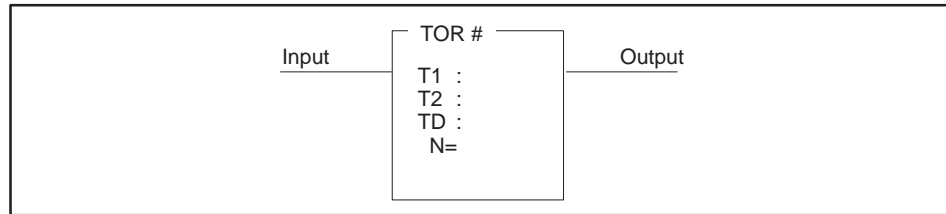
These RLL instructions are also used for electro-mechanical replacement.

Contacts	Coils	CTR	DCAT	DRUM	EDRUM
JMP	MCAT	MCR	MDRMD	MDRMW	NOT
SHRB	SKP/LBL	UDC			

Refer to Section E.3 for an application example of the timer.

### 6.67 Table to Table OR

**TOR Description** The Table to Table OR instruction (Figure 6-96) ORs the corresponding bits in two tables and places the results in a specified third table. If either bit is 1, then the resultant bit is set to 1. Otherwise, the resultant bit is set to 0.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
T1	Any readable word	Starting address for the first table.
T2	Any readable word	Starting address for the second table.
TD	Any writeable word	Starting address for the destination table. TD can be the same as T1 or T2, or be different.
N	1-256	Specifies table length. All tables are N words long.
<b>Note:</b> If you plan to use this instruction in a subroutine, refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.		

Figure 6-96 TOR Format

**TOR Operation** The operation of the TOR is described below.

- When the input turns on, a comparison is made between each bit of each word in the first (T1) and second (T2) tables.
- Each pair of bits is ORed, and the resultant bit is placed in the third table (TD). If either bit is 1, then the resultant bit is set to 1. Otherwise, the resultant bit is set to 0.
- The bits in all the words of the two tables are ORed each scan.
- The output is turned on when the instruction is executed.

If the input is off, the instruction is not executed, and there is no power flow at the box output.

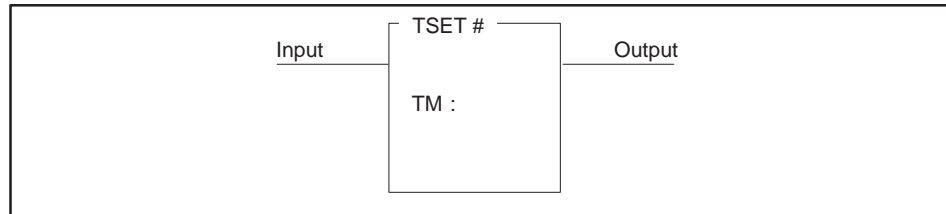
**See Also** These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFE	STFN	TAND	TCPL
TTOW	TXOR	WTOT	WTTA	WTT0	WTTX0

## 6.68 Time Set

### TSET Description

The Time Set instruction (Figure 6-97) sets the time portion of the real-time clock to the values contained in designated memory locations.



Field	Valid Values	Function
#	1 to number of one shots.	Instruction reference number. The TSET uses one shot memory. The assigned instruction number must conform to the requirements of one-shot memory discussed on page 4-7 in Section 4.2. Each TSET instruction must have a unique number.
TM	V, G, W, VMS, VMM	Designates the memory locations containing time to be written into the real-time clock.* V(TM) = Hours — BCD value 0000–0023. V(TM+1) = Minutes — BCD value 0000–0059. V(TM+2) = Seconds — BCD value 0000–0059.
<b>Note:</b> If you plan to use this instruction in a subroutine, refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.		

\* BCD values are entered using the HEX data format.

Figure 6-97 TSET Format

### TSET Operation

When the input to the TSET instruction transitions from off to on, the time portion of the real-time clock is set to the values contained within the three consecutive V-Memory locations designated by TM, and the output turns on for one scan.

**NOTE:** The time of day status words (STW141–144 and STW223–225) do not reflect the time change until the next RLL scan.

When the input is off, the operation does not execute, and there is no power flow at the box output.

### See Also

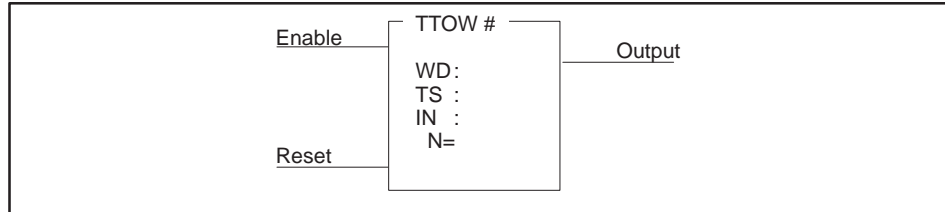
These RLL instructions can also be used for date/time functions.

DCMP	DSET	TCMP
------	------	------



6.69 Table to Word

**TTOW Description** The Table to Word instruction (Figure 6-98) copies a word in a table and places it in another memory location.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
WD	Any writeable word	Memory location for destination of the word.
TS	Any readable word	Starting address of source table.
IN	V, G, W, VMS, VMM	Specifies memory location where index is stored. The index indicates which word in the table is copied.
N	1-256	Length of table in words.

Figure 6-98 TTOW Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

## TTOW Operation

The operation of the TTOW is described below.

- The Reset must be on for the instruction to execute.
- When the Enable turns on, a copy is made of the specified word in the table TS.

The index (IN) indicates which word in the table is copied. The value contained by the index ranges from 0 to N-1, where N is the length of the table. If  $0 \leq IN < N$ , the word is copied. If  $N \leq IN$  or  $N < 0$ , the word is not copied.

- The word is placed in the memory location specified by WD. After the word is placed there, the value contained by the index increments by one.
- If both Enable and Reset remain on, one word is duplicated each scan.
- If the Enable turns off while the Reset is on, the index holds its current value and the word is not moved.

If the Reset turns off, the index resets to 0.

- The TTOW output remains on until the last word in the table is copied. It then turns off.
- The TTOW must be reset (Reset turns off) after the output turns off in order to execute again.

If the Reset is off, the instruction does not execute, and there is no power flow at the box output.

## See Also

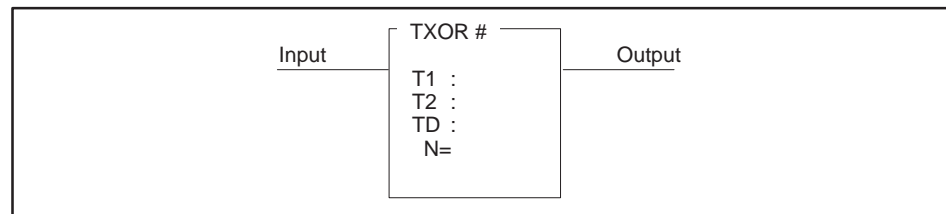
These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFE	STFN	TAND	TCPL
TOR	TXOR	WTOT	WTTA	WTO	WTTXO

## 6.70 Table to Table Exclusive OR

### TXOR Description

The Table to Table Exclusive OR instruction (Figure 6-99) executes an Exclusive OR on the corresponding bits in two tables and places the results in a specified third table. If the bits compared are the same, the resultant bit is set to a 0. If the bits compared are different, the resultant bit is set to 1.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
T1	Any readable word	Starting address of the first table.
T2	Any readable word	Starting address of the second table.
TD	Any writeable word	Starting address of the destination table. TD can be the same as T1 or T2, or can be different.
N	1-256	Table length. All tables are N words long.

Figure 6-99 TXOR Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

---

**TXOR Operation**

The operation of the TXOR is described below.

- When the input turns on, a comparison is made between each bit of each word in the first (T1) and second (T2) tables.
- An Exclusive OR is executed on each pair of bits, and the resultant bit is placed in the third table (TD). If the bits compared are either both 1s or both 0s, the resultant bit is set to a 0. If the bits compared are unlike (1 and 0), the resultant bit is set to 1.
- An Exclusive OR is executed on the bits in *all* the words of the two tables each scan.
- The output turns on when the instruction executes.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

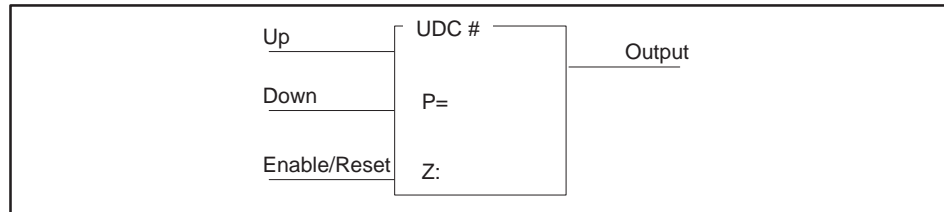
**See Also**

These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFE	STFN	TAND	TCPL
TOR	TTOW	WTOT	WTTA	WTTO	WTTXO

## 6.71 Up/Down Counter

**UDC Description** The Up-Down Counter instruction (Figure 6-100) counts the number of events (up or down) from 0 to 32,767.



Field	Valid Values	Function
#	Varies with controller model	Instruction reference number. Refer to controller user manual for number supported. The assigned instruction number must conform to the requirements of timer/counter memory discussed on page 4-5 in Section 4.2.
P	0-32767	Preset maximum value to which the UDC counts. The UDC does not count events beyond P.
Z	Y, C, B, or blank	Address of the coil to be turned on when the current count is equal to zero.

Figure 6-100 UDC Format

---

**UDC Operation**

When the counter counts up, it counts to the preset value specified in P, that is stored in TCP-Memory. The current count is stored in TCC-Memory.

- The Enable/Reset must be on for the counter to operate.
- When the Enable/Reset is on, the counter increments by one when the Up input transitions from off to on.
- When the Enable/Reset is on, the counter decrements by one when the Down input transitions from off to on. The UDC does not decrement to a number less than zero.
- TCC does not change if the Up and Down inputs both change from off to on during the same scan.
- If the Enable/Reset turns off, TCC resets to zero.
- The output specified in Z turns on whenever TCC equals zero. This output turns off when TCC does not equal zero.
- The box output turns on whenever TCC equals zero or TCP.
- After having counted to the preset value (TCP), the box does not require resetting in order to resume counting in the opposite direction. TCC does not ever exceed TCP.

If the Enable/Reset does not receive power flow, the instruction does not execute and the output does not turn on.

## Up/Down Counter (continued)

---

### Using the UDC Variables

Other RLL instructions can be used to read from or write to the UDC variables. You can also use an operator interface to read from or write to the UDC variables. While you are programming the UDC, you are given the option of protecting the preset values from changes made with an operator interface.

---

**NOTE:** If you use an operator interface to change TCP, the new TCP value is not changed in the original RLL program. If the RLL presets are ever downloaded the changes made with the operator interface are replaced by the original values in the RLL program.

---

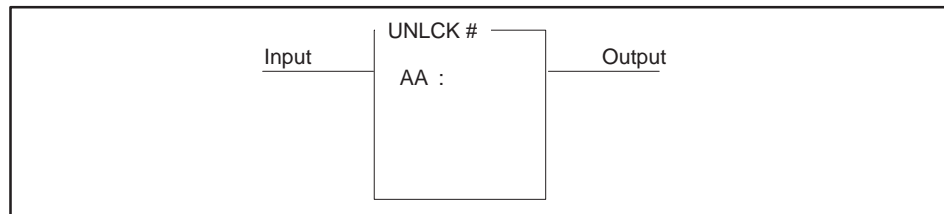
### See Also

These RLL instructions are also used for electromechanical replacement.

Contacts	Coils	CTR	DCAT	DRUM	EDRUM
JMP	MCAT	MCR	MDRMD	MDRMW	NOT
SHRB	SKP/LBL	TMR			

## 6.72 Unlock Memory

**UNLCK Description** The Unlock instruction (Figure 6-101), works with the LOCK instruction to provide a means whereby multiple applications in the 575 system coordinate access to shared resources, generally G-Memory data blocks.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers for documentation purposes only; can be repeated.
AA	G, VMS, VMM	Memory location (2 words) where lock structure is stored. Use same address for associated LOCK instruction.

Figure 6-101 UNLCK Format

**UNLCK Operation** Refer to Section 6.28 for a description of how UNLCK works with the LOCK instruction.

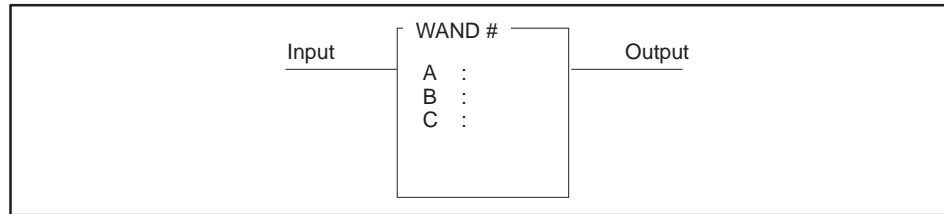
**See Also** This RLL instruction is also used to coordinate access to shared resources.

LOCK



## 6.73 Word AND

**WAND Description** The Word AND instruction (Figure 6-102) logically ANDs a word in memory location A with a word in memory location B, bit for bit. The result is stored in memory location C.



Field	Valid Values	Function
#	0–32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
A	Any readable word	Memory location of the first word in the AND operation.
B	Any readable word	Memory location of the second word in the AND operation.
	or constant (–32768 to +32767)	Value of the second word when a constant is used.
C	Any writeable word	Memory location where the result is stored.

Figure 6-102 WAND Format

**WAND Operation** When the input turns on, the instruction executes. If the input remains on, the instruction is executed on every scan.

- The word stored in the memory location specified by A is ANDed with the word stored in the memory location specified by B. The operation is done bit by bit, as illustrated in Figure 6-103.

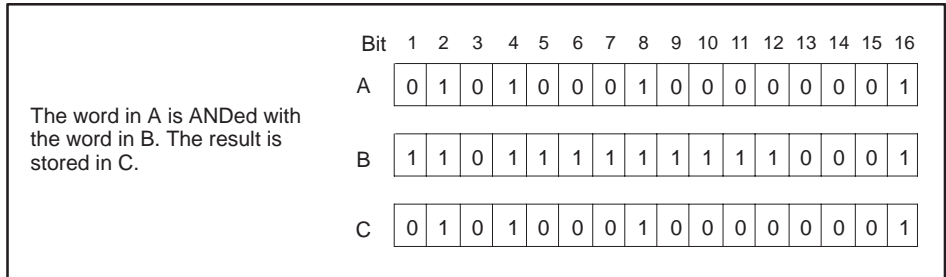
The words in A and B are not affected by the WAND instruction and retain their original values.

	A	B	C
	0	0	0
	0	1	0
	1	0	0
	1	1	1

For each bit location A and B,  
the result of an AND operation is given in C.

Figure 6-103 Result of ANDing Bits

- The result is stored in the memory location specified by C, as illustrated in Figure 6-104.



**Figure 6-104 Result of ANDing Two Words**

- If C is not zero, the output turns on when the instruction executes.
- If C is zero, the output turns off.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

See Also

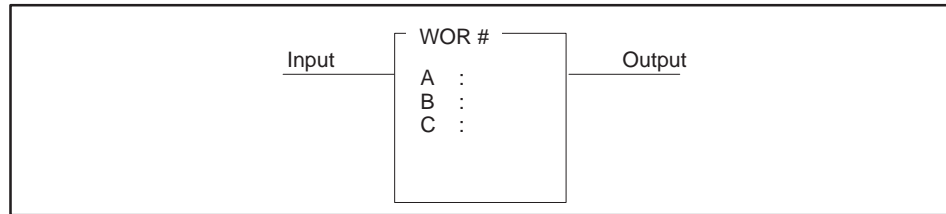
These RLL instructions are also used for bit manipulation.

BITC	BITS	BITP	IMC	SMC	WOR
WROT	WXOR	Bit-of-Word Contact/Coil			

6.74 Word OR

WOR Description

The Word OR instruction (Figure 6-105) logically ORs a word in memory location A with a word in memory location B. The result is stored in memory location C.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
A	Any readable word	Memory location of the first word in the OR operation.
B	Any readable word	Memory location of the second word in the OR operation.
	or constant (-32768 to +32767)	Value of the second word when a constant is used.
C	Any writeable word	Memory location where the result is stored.

Figure 6-105 WOR Format

WOR Operation

When the input is on, the WOR box executes. If the input remains on, the instruction executes on every scan.

- The word stored in the memory location specified by A is ORed with the word stored in the memory location specified by B. The operation is done bit by bit, as illustrated in Figure 6-106.

The words in A and B are not affected by the OR instruction and retain their original values.

For each bit location A and B, the result of an OR operation is given in C.	A	B	C
	0	0	0
	0	1	1
	1	0	1
	1	1	1

Figure 6-106 Result of ORing Bits

- The result is stored in the memory location specified by C, as illustrated in Figure 6-107.

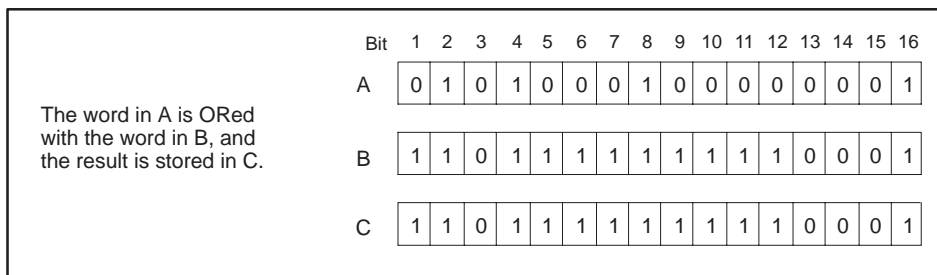


Figure 6-107 Result of ORing Two Words

- If C is not zero, the output turns on when the instruction executes.
- If C is zero, the output turns off.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

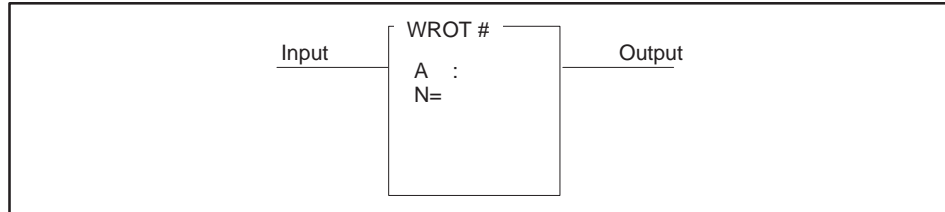
See Also

These RLL instructions are also used for bit manipulation.

BITC	BITS	BITP	IMC	SMC	WAND
WROT	WXOR	Bit-of-Word Contact/Coil			

## 6.75 Word Rotate

**WROT Description** The Word Rotate instruction (Figure 6-108) operates on the 4-bit segments of a word, rotating them to the right.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
A	Any writeable word	Memory location of the word to be rotated.
N	1-3	Number of times that the 4-bit segments are rotated.

Figure 6-108 WROT Format

**WROT Operation** When the input is turned on, the WROT box executes. If the input remains on, the instruction executes on every scan.

- Each 4-bit segment of the word specified in memory location A shift to the right as shown in Figure 6-109.

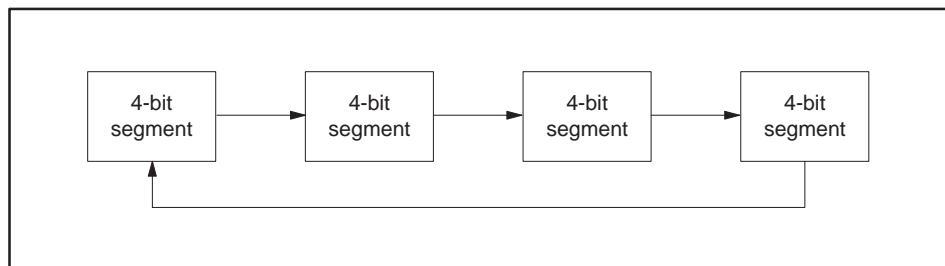


Figure 6-109 WROT Operation

- A segment can shift up to 3 positions as specified by N. See Figure 6-110.
- If A is not zero, the output turns on when the instruction executes.
- If A is zero, the output turns off.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

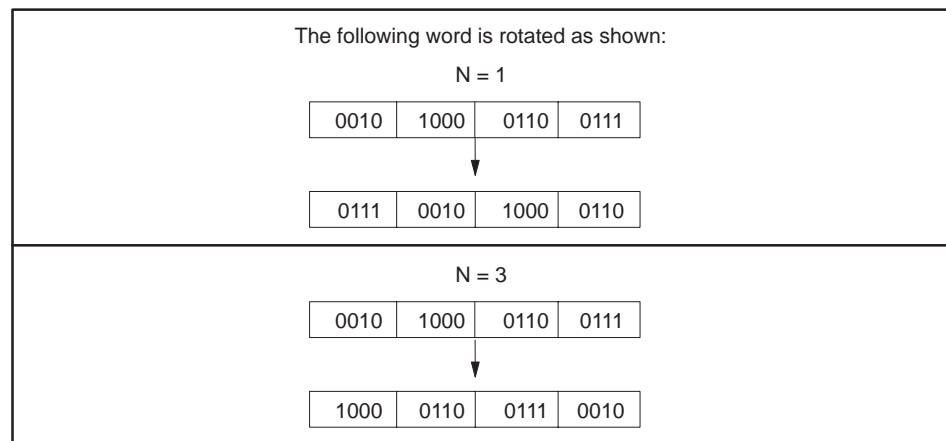


Figure 6-110 Result of a WROT Operation

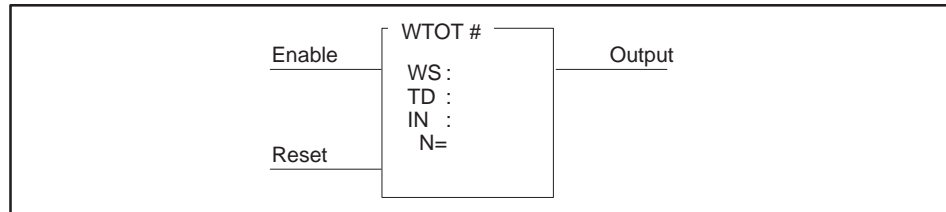
See Also

These RLL instructions are also used for bit manipulation.

BITC	BITS	BITP	IMC	SMC	WAND
WOR	WXOR	Bit-of-Word Contact/Coil			

6.76 Word To Table

**WTOT Description** The Word To Table instruction (Figure 6-111) places a copy of a word at a specified address within a table.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
WS	Any readable word	Memory location of the source word.
TD	Any writeable word	Starting address of the table.
IN	V, G, W, VMS, VMM	Specifies memory location where the index is stored. The index specifies where the word is placed in the table.
N	1-256	Specifies length of the table.

Figure 6-111 WTOT Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**WTOT Operation**

The operation of the WTOT is described below.

- The Reset must be on for the instruction to execute.
- When the Enable turns on, a copy of the source word WS is placed in the destination table TD.

The index (IN) indicates where the word is placed in the table. The value contained by the index ranges from 0 to N-1, where N is the length of the table. If  $0 \leq IN < N$ , the word is moved. If  $N \leq IN$  or  $N < 0$ , the word is not moved.

- After the word is placed into the table, the value contained by the index increments by one.
- If both Enable and Reset remain on, one word is moved each scan.
- If the Enable turns off while the Reset is on, the index holds its current value and the word is not moved.

If the Reset turns off, the index resets to 0.

- The WTOT output remains on until a word is placed in the last position in the table. It then turns off.
- The WTOT must be reset (Reset turns off) after the output turns off, in order to execute again.

If the Reset is off, the instruction does not execute, and there is no power flow at the box output.

**See Also**

These RLL instructions are also used for table operations.

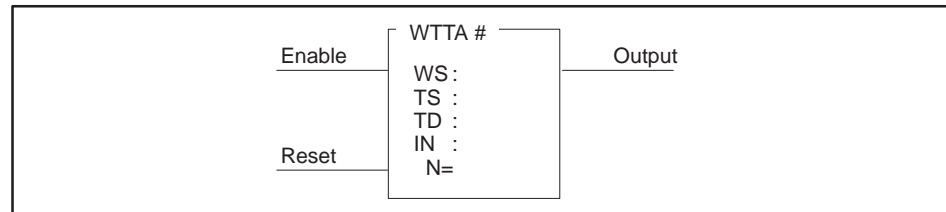
MIRFT	MIRTT	STFE	STFN	TAND	TCPL
TOR	TTOW	TXOR	WTTA	WTTO	WTTXO



## 6.77 Word To Table AND

### WTTA Description

The Word To Table AND instruction (Figure 6-112) ANDs each bit in a source word with the corresponding bit of a designated word in a table. The results are placed in a destination table. If both bits are 1s, a 1 is stored in the destination table. Otherwise, the resultant bit is set to 0.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
WS	Any readable word	Memory location of the source word.
TS	Any readable word	Starting address of the source table.
TD	Any writeable word	Starting address of the destination table. TD can be the same as TS or can be different.
IN	V, G, W, VMS, VMM	Specifies memory location where the index is stored. The index specifies that word in the table is ANDed.
N	1-256	Specifies length of the table.

Figure 6-112 WTTA Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**WTTA Operation**

The operation of the WTTA is described below.

- The Reset must be on for the instruction to execute.
- When the Enable turns on, each bit of the source word WS and of a specified word in the table TS is compared.

The index (IN) indicates which word in the table is ANDed. The value contained by the index ranges from 0 to N-1, where N is the length of the table. If  $0 \leq IN < N$ , the word is ANDed. If  $N \leq IN$  or  $N < 0$ , the word is not ANDed.

- Each pair of bits is ANDed, and the resultant bit is placed in the destination table TD. If both bits are 1s, the resultant bit is set to 1. Otherwise, the resultant bit is set to 0.

After a word in the table is compared, the value contained by the index increments by one.

- If both Enable and Reset remain on, the source word and a word in the table are ANDed each scan.
- If the Enable turns off while the Reset is on, the index holds its current value and the AND does not occur.

If the Reset turns off, the index resets to 0.

- The WTTA output remains on until the last word in the table has been ANDed with the source word. It then turns off.
- The WTTA must be reset (Reset turns off) after the output turns off in order to execute again.

If the Reset is off, the instruction does not execute, and there is no power flow at the box output.

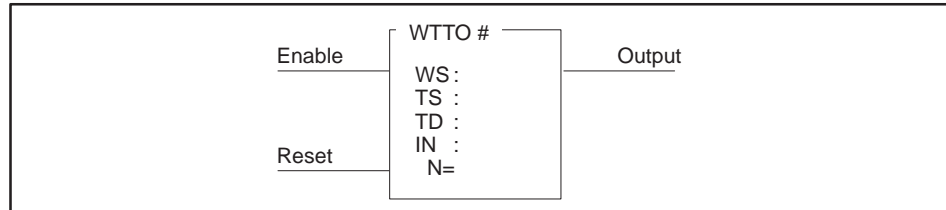
**See Also**

These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFE	STFN	TAND	TCPL
TOR	TTOW	TXOR	WTOT	WTTO	WTTXO

6.78 Word To Table OR

**WTTO Description** The Word To Table OR instruction (Figure 6-113) ORs each bit in a source word with the corresponding bit of a designated word in a table. The results are placed in a destination table. If either bit is 1, a 1 is stored in the destination table. Otherwise, the resultant bit is set to 0.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
WS	Any readable word	Memory location of the source word.
TS	Any readable word	Starting address of the source table.
TD	Any writeable word	Starting address of the destination table. TD can be the same as TS or can be different.
IN	V, G, W, VMS, VMM	Specifies memory location where the index is stored. The index specifies which word in the table is ORed.
N	1-256	Specifies length of the table.

Figure 6-113 WTTO Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**WTTO Operation**

The operation of the WTTO is described below.

- The Reset must be on for the instruction to execute.
- When the Enable turns on, each bit of the source word WS and of a specified word in the table TS is compared.

The index (IN) indicates which word in the table is ORed. The value contained by the index ranges from 0 to N-1, where N is the length of the table. If  $0 \leq IN < N$ , the word is ORed. If  $N \leq IN$  or  $N < 0$ , the word is not ORed.

- Each pair of bits is ORed, and the resultant bit is placed in the destination table TD. If either bit is 1, then the resultant bit is set to 1. Otherwise, the resultant bit is set to 0.

After a word in the table is compared, the value contained by the index increments by one.

- If both Enable and Reset remain on, the source word and a word in the table are ORed each scan.
- If the Enable turns off while the Reset is on, the index holds its current value and the OR does not occur.

If the Reset turns off, the index resets to 0.

- The WTTO output remains on until the last word in the table has been ORed with the source word. It then turns off.
- The WTTO must be reset (Reset turns off) after the output turns off in order to execute again.

If the Reset is off, the instruction does not execute, and there is no power flow at the box output.

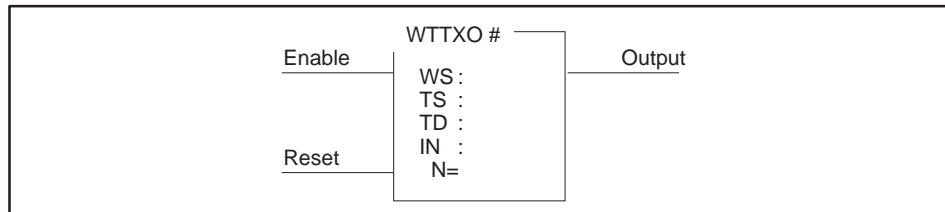
**See Also**

These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFE	STFN	TAND	TCPL
TOR	TTOW	TXOR	WTOT	WTTA	WTTXO

## 6.79 Word To Table Exclusive OR

**WTTXO Description** The Word To Table Exclusive OR (Figure 6-114) executes an Exclusive OR on each bit in a source word with the corresponding bit of a designated word in a table. The results are placed in a destination table. If the bits compared are the same, the resultant bit is set to a 0. Otherwise, the resultant bit is set to 1.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
WS	Any readable word	Memory location of the source word.
TS	Any readable word	Starting address of the source table.
TD	Any writeable word	Starting address of the destination table. TD can be the same as TS or can be different.
IN	V, G, W, VMS, VMM	Specifies memory location where the index is stored. The index specifies on which word in the table that the Exclusive OR is executed.
N	1-256	Specifies length of the table.

Figure 6-114 WTTXO Format

**NOTE:** If you plan to use this instruction in a subroutine (using W-memory operands), refer to page 4-10 for the discussion of how parameters are passed to ensure correct operation of the instruction.

**WTTXO Operation**

The operation of the WTTXO is described below.

- The Reset must be on for the instruction to execute.
- When the Enable turns on, each bit of the source word WS and of a specified word in the table TS is compared.

The index (IN) indicates the word in the table on which the Exclusive OR occurs. The value contained by the index ranges from 0 to N-1, where N is the length of the table. If  $0 \leq IN < N$ , the Exclusive OR takes place. If  $N \leq IN$  or  $N < 0$ , the Exclusive OR does not take place.

- An Exclusive OR is executed on each pair of bits, and the resultant bit is placed in the destination table TD. If the bits compared are the same, the resultant bit is set to a 0. If the bits compared are different, the resultant bit is set to 1.

After a word in the table is compared, the value contained by the index increments by one.

- If both Enable and Reset remain on, the Exclusive OR executes on the source word and a word in the table each scan.
- If the Enable turns off while the Reset is on, the index holds its current value and the Exclusive OR does not take place.

If the Reset turns off, the index resets to 0.

- The WTTXO output remains on until the last word in the table has been compared with the source word. It then turns off.
- The WTTXO must be reset (Reset turns off) after the output turns off in order to execute again.

If the Reset is off, the instruction does not execute, and there is no power flow at the box output.

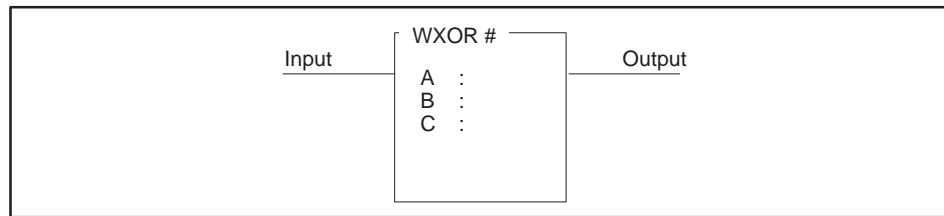
**See Also**

These RLL instructions are also used for table operations.

MIRFT	MIRTT	STFE	STFN	TAND	TCPL
TOR	TTOW	TXOR	WTOT	WTTA	WTTO

## 6.80 Word Exclusive OR

**WXOR Description** The Word Exclusive OR instruction (Figure 6-115) executes a logical Exclusive OR on a word in memory location A with a word in memory location B. The result is stored in memory location C.



Field	Valid Values	Function
#	0-32767	Instruction reference number. Numbers are for documentation purposes only; can be repeated.
A	Any readable word	Memory location of the first word in the Exclusive OR operation.
B	Any readable word	Memory location of the second word in the Exclusive OR operation.
	or constant (-32768 to +32767)	Value of second word when a constant is used.
C	Any writeable word	Memory location where the result is stored.

Figure 6-115 WXOR Format

**WXOR Operation** When the input is turned on, the WXOR box execute. If the input remains on, the instruction executes on every scan.

- An Exclusive OR operation executes on the word stored in the memory location specified by A with the word stored in the memory location specified by B. The operation is done bit by bit, as illustrated in Figure 6-116.
- The words in A and B are not affected by the WXOR instruction and retain their original values.

For each bit location A and B, the result of an Exclusive OR operation is given in C.	<table border="1" style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">A</th> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">B</th> <th style="border-bottom: 1px solid black;">C</th> </tr> </thead> <tbody> <tr> <td style="border-right: 1px solid black;">0</td> <td style="border-right: 1px solid black;">0</td> <td>0</td> </tr> <tr> <td style="border-right: 1px solid black;">0</td> <td style="border-right: 1px solid black;">1</td> <td>1</td> </tr> <tr> <td style="border-right: 1px solid black;">1</td> <td style="border-right: 1px solid black;">0</td> <td>1</td> </tr> <tr> <td style="border-right: 1px solid black;">1</td> <td style="border-right: 1px solid black;">1</td> <td>0</td> </tr> </tbody> </table>	A	B	C	0	0	0	0	1	1	1	0	1	1	1	0
A	B	C														
0	0	0														
0	1	1														
1	0	1														
1	1	0														

**Figure 6-116 Result of an Exclusive OR of Bits**

- The result is stored in the memory location specified by C, as illustrated in Figure 6-117.

An Exclusive OR operation is executed on the words in A and B and the result is stored in C.	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: right; padding-right: 5px;">Bit</td> <td style="padding-right: 5px;">1</td> <td style="padding-right: 5px;">2</td> <td style="padding-right: 5px;">3</td> <td style="padding-right: 5px;">4</td> <td style="padding-right: 5px;">5</td> <td style="padding-right: 5px;">6</td> <td style="padding-right: 5px;">7</td> <td style="padding-right: 5px;">8</td> <td style="padding-right: 5px;">9</td> <td style="padding-right: 5px;">10</td> <td style="padding-right: 5px;">11</td> <td style="padding-right: 5px;">12</td> <td style="padding-right: 5px;">13</td> <td style="padding-right: 5px;">14</td> <td style="padding-right: 5px;">15</td> <td style="padding-right: 5px;">16</td> </tr> <tr> <td style="padding-right: 5px;">A</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">1</td> </tr> <tr> <td style="padding-right: 5px;">B</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">1</td> </tr> <tr> <td style="padding-right: 5px;">C</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">1</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> <td style="border: 1px solid black; text-align: center;">0</td> </tr> </table>	Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	A	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1	B	1	1	0	1	1	1	1	1	1	1	1	1	0	0	0	1	C	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0	0
Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16																																																					
A	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1																																																					
B	1	1	0	1	1	1	1	1	1	1	1	1	0	0	0	1																																																					
C	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0	0																																																					

**Figure 6-117 Result of an Exclusive OR of Two Words**

- If C is not zero, the output turns on when the instruction executes.
- If C is zero, the output turns off.

If the input is off, the instruction does not execute, and there is no power flow at the box output.

See Also

These RLL instructions are also used for bit manipulation.

BITC	BITS	BITP	IMC	SMC	WAND
WOR	WROT	Bit-of-Word Contact/Coil			

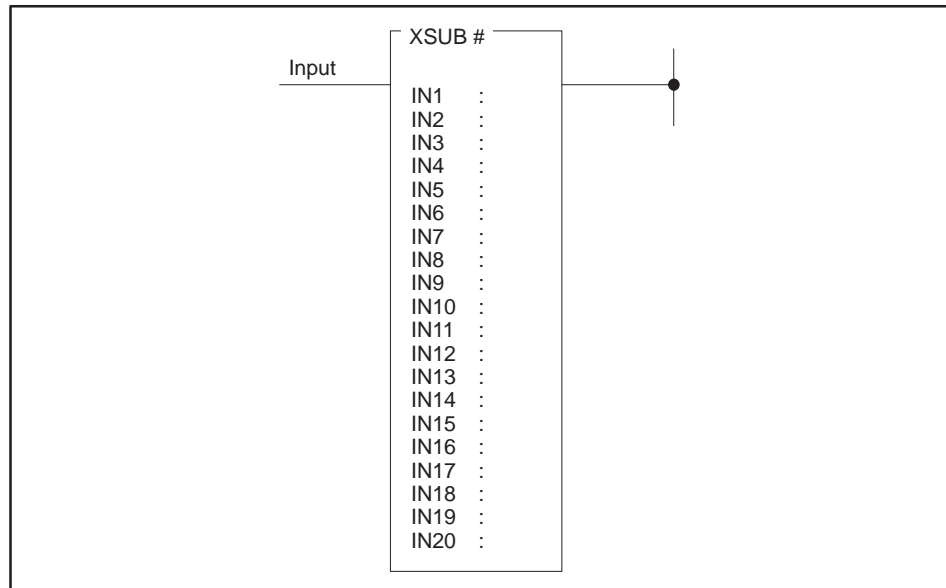
Refer to Section E.11 for an application example of the WXOR.



## 6.81 External Subroutine Call

### XSUB Description

The XSUB (Figure 6-118) allows you to pass parameters to a subroutine that is developed offline in a non-RLL programming language, such as C or Pascal, and then call the subroutine for execution. Refer to Appendix H for more information about designing and writing external subroutines.



Field	Valid Values	Function
#	1–32767	Designates subroutine to call.
IN/IO	IN followed by any readable bit or word. IO followed by any readable bit or word.	IN: Designates address that contains data to be read by the subroutine. IO: Designates an address to be passed to the subroutine.  B and W locations are valid only when XSUB is used in a subroutine.

Figure 6-118 XSUB Format

**NOTE:** The parameter fields (IN1–IN20) allow read-only addresses, e.g., K or WX, to be specified as I/O parameters. This allows you to pass the base address of a read-only array to the subroutine. It is recommended that you not design the subroutine to alter the contents of the read-only variable(s) since other instructions assume that they do not change.

**⚠ WARNING**

When you call an external subroutine, the built-in protection features of the controller are by-passed. Take care in testing the external subroutine before introducing it to a control environment.

Failure of the external subroutine may cause undetected corruption of controller memory and unpredictable operation by the controller, which could result in death or serious injury to personnel, and/or damage to equipment.

You must be careful in testing the external subroutine before introducing it to a control environment.

**XSUB Operation**

The operation of the XSUB instruction is described below. See Figure 6-119.

- Parameters must be numbered consecutively, i.e., you cannot skip parameter numbers.
- When the input is turned on:

The parameters are pushed on the user stack, in order, from the last parameter to the first parameter, and then the subroutine is called. This corresponds to the C language calling convention.

When a discrete data element (X, Y, C, B) is specified as an IN parameter, the discrete value is passed in the least significant bit of a long word. All other bits of the long word are unspecified (may be 0 or 1).

When a discrete data element is specified as an IO parameter, the address of the data element is passed. The actual value of the data element is contained in the least significant bit of the byte at this address. Other bits of this byte are unspecified.

When a word data element (V, K, etc.) is specified as an IN parameter, the value of the long word at this specified data element and the specified data element + 1 (e.g., V100 and V101) is passed. The addressed word is in the most significant half, and the next consecutive word is in the least significant half. Any readable data element is allowed.

When a word data element is specified as an IO parameter, the address of the data element is passed. The value of the parameter is contained at this address.

After all parameters have been pushed onto the stack, the subroutine is called. If the subroutine successfully executes (see Notes below) STW01 bit 11 turns off, and the controller continues the scan with the next network.

External Subroutine Call (continued)

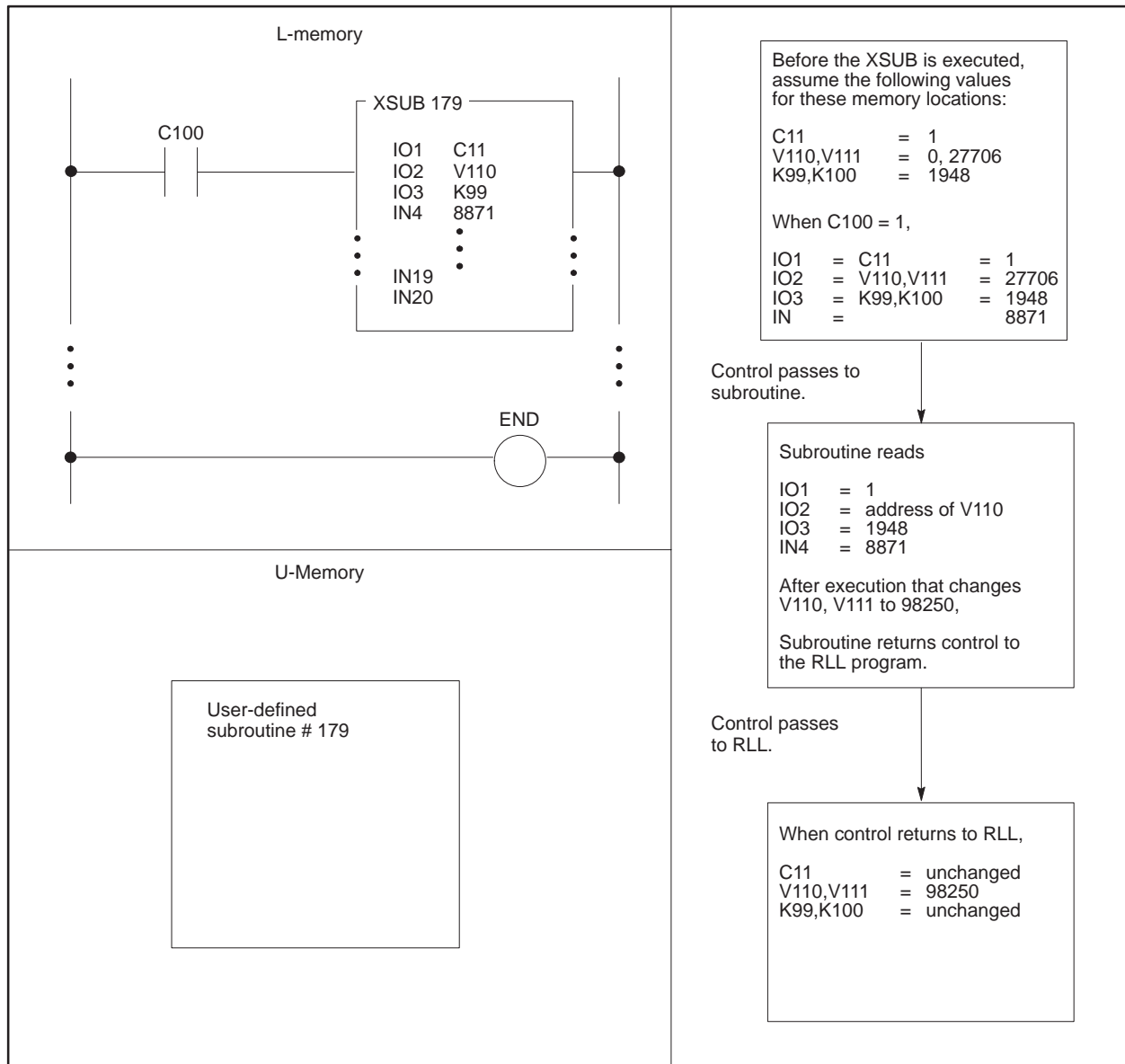


Figure 6-119 Example of the XSUB Instruction

**NOTE:** An XSUB in RLL with no defined external subroutine causes the user program error bit (6) and the instruction failed bit (11) to be set in STW01, with the reason set to 6 in STW200 (if this is the first error logged). The controller remains in RUN mode.

---

**NOTE:** For the 575, if an XSUB instruction attempts to access a non-existent VMEbus address a VMEbus error occurs. If this is the first VMEbus error, the offending VMEbus address is written to STW227-STW228 and the U-Memory offset of the offending instruction is written to STW229-STW230.

If you set the U-Memory header's E bit to 1 when you create your external subroutine(s), a VMEbus error will terminate the XSUB and continue RLL execution with the network following the XSUB instruction. In this case the user program error bit (6) and instruction failed bit (11) in STW01 are set to 1 and, if this is the first user program error encountered on the current RLL scan, the value 7 (VMEbus error) is written to STW200.

---

### **WARNING**

If you set the U-Memory header's E bit to 0 and a VMEbus error occurs during execution of an XSUB, the 575 controller will transition to the Fatal Error mode. The transition to Fatal Error mode freezes word outputs and clears discrete outputs, which could cause unexpected process operation.

Unexpected process operation could result in death or serious injury to personnel, and/or damage to equipment.

Avoid setting the U-Memory header's E bit to 0 when you create external subroutines.

- When the input is off, the instruction does not execute and the subroutine is not called. Bit 11 of STW01 turns off.

See Also

These RLL instructions are also used for subroutine operations.

GTS	PGTS	PGTSZ	RTN	SBR	SFPGM	SFSUB
-----	------	-------	-----	-----	-------	-------

# Chapter 7

## Special Function Programs

---

7.1	Defining Special Function Programs .....	7-2
7.2	Using PowerMath with Special Function Programming .....	7-4
7.3	SF Program Statements .....	7-10
7.4	Executing Special Function Programs .....	7-11
7.5	Executing Special Function Subroutines .....	7-14
7.6	Memory Usage by SF Programs .....	7-16
7.7	Entering SF Program Header with TISOFT .....	7-18
7.8	Reporting SF Program or SFSUB RLL Instruction Errors .....	7-20
7.9	Entering Special Function Programming Statements .....	7-22
7.10	Convert BCD to Binary .....	7-24
7.11	Convert Binary Inputs to BCD .....	7-25
7.12	Call Subroutine .....	7-26
7.13	Correlated Data Table .....	7-28
7.14	Exit on Error .....	7-30
7.15	Fall Through Shift Register—Input .....	7-31
7.16	Fall through Shift Register—Output .....	7-35
7.17	Go To/Label Function .....	7-39
7.18	IF/IIF/THEN/ELSE Functions .....	7-40
7.19	Integer Math Operations .....	7-42
7.20	Lead/Lag Operation .....	7-44
7.21	Real/Integer Math Operations .....	7-46
7.22	Pack Data .....	7-51
7.23	Pack Analog Alarm Data .....	7-56
7.24	Pack Loop Data .....	7-58
7.25	Pack Ramp/Soak Data .....	7-60
7.26	Pet Scan Watchdog .....	7-66
7.27	Printing .....	7-68
7.28	Return from SF Program/Subroutine .....	7-71
7.29	Scaling Values .....	7-72
7.30	Sequential Data Table .....	7-74
7.31	Synchronous Shift Register .....	7-76
7.32	Unscaling Values .....	7-78
7.33	Comment .....	7-80

## 7.1 Defining Special Function Programs

---

### Introduction

A special function program (SF program) consists of a set of instructions that can be called from loops, analog alarms, or from the RLL program, much like a GOSUB subroutine in a BASIC program or a procedure in a C-language program.

The higher-level, statement-driven programming language used in an SF program makes your programming task easier. You can derive solutions for complex programs that would require extensive RLL programming and consume large blocks of ladder memory. Operations such as mathematical calculations, if/then statements, unit and number format conversions, table transfers, data consolidation, etc., can be done with an SF program. Typically, these types of operations either cannot be done with the RLL instruction set, or they involve complex RLL programming.

An SF program can call a subroutine (SF subroutine) for execution. After completion, the SF subroutine returns control to the SF program that called it. The same programming statements used to write SF programs are used to write SF subroutines. An SF program cannot call other SF programs for execution, but SF subroutines can call other SF subroutines.

- Up to 1023 SF programs and 1023 SF subroutines can be defined on the 545–1104 and 545–1106 CPUs, all 555 CPUs, and all 575 CPUs.
- The 545–1103 and 545–1105 CPUs can support up to 64 SF programs and subroutines.

You must allocate a block of memory called Special Memory (S-Memory) before you can create SF programs. You do this with your programming unit when you configure controller memory. SF programs and SF subroutines are stored in S-Memory.

### Special Function Program Types

SF programs are categorized functionally by how they are called for execution. You designate the program type when you enter the program. The various SF program types are Priority, Non-priority, Cyclic, and Restricted.

---

**SF Programs Called from RLL**

Priority, non-priority, and cyclic SF programs are called from the RLL program by the RLL SFPGM instruction.

- A priority/non-priority SF program executes once after the input to the RLL SFPGM instruction transitions from off to on. The SF program does not execute again until the input to the RLL SFPGM instruction transitions from off to on again.

If the controller changes from PROGRAM to RUN mode while the input to the RLL SFPGM instruction is on, the SF program is queued for execution.

The difference between priority and non-priority SF programs is based on the amount of processor time allocated to executing the SF program. You allocate processor time to the two types of SF programs using the scan time tuning features (aux 19) of TISOFT.

- A cyclic SF program executes when the input to the RLL SFPGM instruction transitions from off to on. When the cyclic SF program has terminated, it is automatically re-queued for execution based on the programmed cycle time (0.5 second increments). This process continues as long as the input to the RLL SFPGM instruction is on. When the input turns off, the cyclic SF program is not re-queued for execution. (However, if it has been queued, it will execute one more time.)

You can adjust the cyclic SF processor time using AUX 19 of TISOFT.

**SF Programs Called from Loops/Analog Alarms**

Restricted SF programs are called by loops and analog alarms only. The processor program execution time dedicated to restricted SF programs is determined by the time allocated to loop and analog alarm processing. For the 545, 555, and 575, this processor time is user configurable.

## 7.2 Using PowerMath with Special Function Programming

---

### What is PowerMath?

PowerMath is supported in the 555–1105/–1106 and 575–2105/–2106 CPUs. PowerMath provides an easy-to-use computational environment which greatly extends the CPU's power by taking advantage of the new on-board floating-point co-processor and the built-in integer processor in the CPU's microprocessor. In addition, high-level Special Function programs can now be automatically compiled at run-time, reducing resource demand and thereby greatly increasing execution speed.

- SF programs and subroutines can now be marked as either “compiled” or “interpreted.” A compiled SF program or subroutine uses the CPU's integer and floating-point processors to execute the program directly, providing significant execution speed improvements over the default interpreted execution method.
- An SFPGM or SFSUB box instruction can be marked for “in-line” execution if the referenced SF program or subroutine has been compiled. When power flow is on for an in-line SFPGM or SFSUB box, it executes immediately as part of the RLL scan. The result of the box's execution is available to the next element of the current RLL rung.

In addition to these capabilities, PowerMath provides a number of extensions to the Special Function (SF) language. These extensions are available to compiled and interpreted SF programs and subroutines.

### 32-Bit Signed and 16-Bit Unsigned Integer Math

Special Function integer and floating-point expressions now support unsigned 16-bit integer and signed 32-bit (“long”) integer operands where expressions are allowed in PowerMath CPUs.

- An unsigned integer operand is identified with a “U” suffix, e.g., V105U or 32768U, and has a range of 0 to 65,535, inclusive. Unsigned integer variables occupy one word (16 bits) of controller memory.
- A long integer operand is identified with an “L” suffix, e.g., K15L or –200L, and has a range of –2,147,483,648 to 2,147,483,647, inclusive. Long integer variables occupy two consecutive words of controller memory. (In an interpreted SF program or subroutine, you should limit use of long integer operands to integer expressions. This will avoid loss of accuracy caused by the single-precision accumulator used for interpreted floating-point expressions.)
- Unsigned and long integer variables can reside in constant (K), user variable (V, G, VMS, or VMM), or temporary (T) memory, only.



Two new SF statements have been added to PowerMath CPUs:

- IIF (Integer IF) allows you to code IF-THEN-ELSE blocks using an integer expression for the conditional. An integer expression will execute faster than the equivalent floating-point expression. (Compiled integer expressions also use less memory than the equivalent floating-point expression.)
- PETWD (Pet Scan Watchdog) allows you to extend the scan watchdog limit while performing an in-line SFPGM or SFSUB from an RLL program.

Integer expressions (IMATH and IIF statements) now support the following additional operators:

- Logical AND
- Logical OR
- Comparison: “<”, “<=”, “=”, “>”, “>=”, and “<>”
- Absolute value (ABS)

Integer expressions (both interpreted and compiled) are now evaluated using a 32-bit accumulator<sup>1</sup>. This was necessary in order to support unsigned and long integer operands.

Compiled floating-point expressions are evaluated using a double-precision<sup>2</sup> floating-point accumulator. The double-precision accumulator allows a compiled floating-point expression to produce a true 32-bit integer result. It also improves the accuracy of all floating-point computations. (The SF interpreter continues to use a single-precision floating-point accumulator.)

The IMATH and MATH operators WAND, WOR, and WXOR now operate on a 32-bit integer accumulator. Because their names imply a 16-bit word operation, these operators have been renamed. WAND is now displayed as “&”, WOR as “|”, and WXOR as “^”.

---

<sup>1</sup>This change may cause a program which was expecting a “silent” integer truncation to log an arithmetic overflow error. Potential areas for consideration should this occur are shift operations and multiply operations.

<sup>2</sup>The data types of an operand of a floating-point expression can be either 16-bit signed integer, 16-bit unsigned integer, 32-bit signed integer, or single-precision floating-point operands. *Double-precision floating-point operands are not supported.* When an expression is used as a parameter to a CALL statement, the value passed to the called subroutine has the type of the variable on the left-hand side of the expression’s assignment (“.=”) operator, if present. If the expression does not assign its result to a variable, the expression’s value is passed as a single-precision floating-point value.

## Using PowerMath with Special Function Programming (continued)

---

### Why Choose Compiled Mode for an SF Program or Subroutine?

When compiled mode is selected, the SF program or subroutine is translated to the native instruction set of the CPU's microprocessor. The compiled code is then executed whenever the program or subroutine is scheduled for execution. The advantages of compiled execution are:

- Significant execution speed improvement. For example, a MATH statement that adds two floating-point values will execute in under 10  $\mu\text{s}$  when compiled versus more than 100  $\mu\text{s}$  when executed by the SF interpreter. Depending on the program's size and the placement of the target LABEL within the program, a GOTO statement may take 1 ms or more when executed by the interpreter. Compiled execution of a GOTO statement takes less than 1  $\mu\text{s}$  no matter where in the program the LABEL is located. This represents a 1,000x improvement.
- A compiled SF program or subroutine can be executed in-line to the user RLL program. This means that when the enable input to the SFPGM or SFSUB box instruction is on, the program or subroutine is executed immediately and its result is available for use in the next rung of the current RLL scan.

### Why Choose Interpreted Mode for an SF Program or Subroutine?

There are several reasons to choose interpreted mode for an SF program. The primary reasons are as follows:

- If the program has one or more statements which are not allowed in a compiled program (see page 7-7), or if it calls a subroutine which is not compiled, then it may not be compiled.
- A compiled program requires both S-Memory and Compiled Special (CS)-Memory, while an interpreted program requires only S-Memory. As a rule of thumb, the compiled code for an SF program requires twice as much CS-Memory as S-Memory. For example, an SF program that uses 1 Kbyte of S-Memory also uses 2 Kbytes of CS-Memory.
- A compiled SF program or subroutine can not be preempted by a second SF program or subroutine on the same execution queue. This may present a scheduling problem for a cyclic, loop or analog alarm queue. For example, if a compiled program is executing on a loop setpoint, a higher priority loop will not execute until the compiled program completes. This is not a problem if the program's execution time is small. However, if the program requires significant execution time, this could cause unnecessary loop overruns.
- The SF interpreter provides superior debug capability. For example, if a programming error causes an interpreted program to be in an infinite loop, you can disable the program to fix the problem. If the same program is compiled, a power cycle may be required. It is a good idea to debug the logic of a complex program or subroutine using the interpreter and then mark the program for compilation.

---

What Can Be  
Compiled?

Most SF programs and subroutines can be compiled. However, an SF program or subroutine which contains any of the following instructions *cannot* be compiled:

- The data compacting instructions: PACK, PACKLOOP, PACKRS, and PACKAA
- The shift register instructions: SSR, FTSR-IN and FTSR-OUT
- The PRINT instruction
- The BCD instructions: BCDBIN and BINBCD

Additionally, the CDT and SDT instructions, when used in a compiled SF program or subroutine, must specify a static table; that is, the table's base address must be a V, K, G, VMS, or VMM address and the table's size must be specified as a constant.

How Do  
SF Programs  
Execute?

Special Function programs execute in the following ways:

**SFPGM RLL instruction without the IN-LINE attribute** — The transition from OFF to ON of the SFPGM box input causes the referenced program to be placed in the appropriate execution queue (normal, priority, or cyclic). When the program reaches the top of the queue, it executes as follows:

- If the program is marked as compiled, its compiled code will execute to completion. If it is executing from a cyclic queue, a higher priority process on the queue will not execute until the program terminates.
- If the program is not marked as compiled, it will be executed by the SF interpreter. If it is executing from a cyclic queue and a higher priority process needs execution time, the higher priority process may interrupt the program's execution on any SF statement boundary.

**SFPGM RLL instruction with the IN-LINE attribute** — When the SFPGM box input is on, the program will execute to completion as part of the RLL process and the box output will remain on. The following exceptions cause the program's execution to fail:

- If the SF program does not exist or if it has not been compiled, user program error 8 is logged in STW200 and there is no power flow at the box output.
- If the SF program is not enabled, user program error 9 is logged in STW200 and there is no power flow at the box output.

## Using PowerMath with Special Function Programming (continued)

---

- If the SF program's type is CYCLIC or RESTRICTED, user program error 10 is logged in STW200 and there is no power flow at the box output.
- If an edit operation is in progress<sup>3</sup>, user program error 11 is logged in STW200 and there is no power flow at the box output.
- If the SFPGM statement is being executed by an interrupt RLL task (555 specific), user program error 12 is logged in STW200 and there is no power flow at the box output.

**Loop or analog alarm** — The program may be compiled or interpreted. If the program is compiled, it will execute to completion when the loop or analog alarm schedules it. If the program is interpreted, it will be initiated when the loop or analog alarm schedules it. While an interpreted SF program is in execution, a higher priority process on its queue may interrupt it on any SF statement boundary.

### How Do SF Subroutines Execute?

Special Function subroutines execute in the following ways:

**SFSUB RLL instruction without the IN-LINE attribute** — When power flow to the SFSUB box transitions from OFF to ON, the subroutine call is queued to the appropriate SF subroutine queue based on the subroutine number. When the subroutine call reaches the top of its queue, the SF interpreter evaluates the subroutine's parameters and then, if the subroutine number is not 0, executes the subroutine. If the subroutine has been compiled, its compiled code is executed.

**SFSUB RLL instruction with the IN-LINE attribute** — When power flow to the SFSUB box is on, the SFSUB's compiled parameter evaluation code is executed and then, if the subroutine number is not 0, the compiled code for the subroutine is executed. Power flow remains on.

The following exceptions cause an in-line SFSUB instruction to fail prior to parameter evaluation:

- An edit operation is in progress. User program error 11 is logged in STW200 and there is no power flow at the box output.
- The SFSUB statement is being executed by an interrupt RLL task. User program error 12 is logged in STW200 and there is no power flow at the box output.

---

<sup>3</sup>An edit operation is any change to L-memory (relay ladder logic), S-memory (SF programs and SF subroutines, loops, and analog alarms), U-memory (external subroutines), or system configuration (scan watchdog, I/O configuration, RAM/ROM program source, port lockout, etc.). Execution is inhibited while the change is in progress, that is, while the user is waiting for the operation's "enter" command to respond at the operator interface.

---

The following exceptions cause the SFSUB instruction to fail after parameter evaluation:

- The referenced SF subroutine does not exist.
- The referenced SF subroutine has not been compiled.
- The referenced SF subroutine is not enabled.

In each of these cases, power flow remains ON. These errors are logged in the SFSUB instruction's Error Status Address and the subroutine is not executed.

CALL Subroutine  
Statement  
Execution

**CALL Subroutine statement in an interpreted SF program or subroutine** — The subroutine parameters are evaluated by the SF interpreter. If the SF subroutine does not exist or if it is not enabled, an error is logged in the program's Error Status Address and the subroutine is not executed. Otherwise, if the subroutine has been compiled, its compiled code is executed to completion as part of the call statement, i.e., the subroutine can not be interrupted by a higher priority cyclic program, loop, or analog alarm executing from the same queue. If the subroutine has not been compiled, it is executed by the SF interpreter and can be interrupted (between statements) by a higher priority process in its queue.

**CALL Subroutine statement in a compiled SF program or subroutine** — The subroutine parameters are evaluated by the compiled SF program or subroutine. If the SF subroutine does not exist, has not been compiled, or is not enabled, an error is logged in the program's Error Status Address and the subroutine is not executed. Otherwise, the subroutine's compiled code is executed to completion, i.e., the subroutine cannot be interrupted by a higher priority cyclic program, loop or analog alarm executing from the same queue.

## 7.3 SF Program Statements

Table 7-1 lists programming statements that are used in SF programs and SF subroutines, and their functions. Compiled SF execution can be used by all controllers that support PowerMath. Certain statements cannot be used in compiled SF programs and subroutines, or can be used with restrictions.

Table 7-1 SF Program Statements

Operation Type	Statement	Function	Compile	Page
Data conversion	BCDBIN	Convert BCD To Binary	No	7-24
	BINBCD	Convert Binary Inputs To BCD	No	7-25
	SCALE	Scaling Values	Yes	7-72
	UNSCALE	Unscaling Values	Yes	7-78
Documentation	*	Comment	Yes	7-80
Math	IMATH	Integer Math Operations	Yes	7-42
	LEAD/LAG	Lead/Lag Operation	Yes	7-44
	MATH	Real/Integer Math Operations	Yes	7-46
Program flow	CALL	Call Subroutine	Yes	7-26
	EXIT	Exit On Error	Yes	7-30
	GOTO/LABEL	Go To/Label Function	Yes	7-39
	IF/THEN/ELSE/ENDIF	If/Then/Else Functions	Yes	7-40
	IIF/THEN/ELSE/ENDIF	Integer If/Then/Else Functions	Yes	7-40
	PETWD	Pet Scan Watchdog (w/ compiled SF only)	Only	7-66
	RETURN	Return from SF program/SF subroutine	Yes	7-71
Printing	PRINT	Print Functions	No	7-68
Table handling	CDT	Correlated Data Table	Yes*	7-28
	FTSR-IN	Fall Through Shift Register - In	No	7-31
	FTSR-OUT	Fall Through Shift Register - Out	No	7-35
	PACK	Pack Data	No	7-51
	PACKAA	Pack Analog Alarm Data	No	7-56
	PACKLOOP	Pack Loop Data	No	7-58
	PACKRS	Pack Ramp/Soak Table	No	7-60
	SDT	Sequential Data Table	Yes*	7-74
	SSR	Synchronous Shift Register	No	7-76

\*The CDT and SDT statements, when used in a compiled SF program or subroutine, must specify a static table; that is, the table's base address must be a V, K, G, VMS or VMM address and the table's size must be specified as a constant.

## 7.4 Executing Special Function Programs

When a priority/non-priority or cyclic SF program is called by the RLL SFPGM instruction, the SF program is placed in a queue for execution. Up to 32 SF programs of each type (for a total of 96 in three queues) can be queued at a given time. If a queue is full, the request for placement on the queue is made again on the next scan. This continues as long as the input to the RLL SFPGM instruction remains on.

The SFPGM instruction can be used anywhere within the RLL program that a single-line input box instruction can be used. Figure 7-1 shows the format of the RLL SFPGM instruction. The # is the number of the SF program to be called for execution.

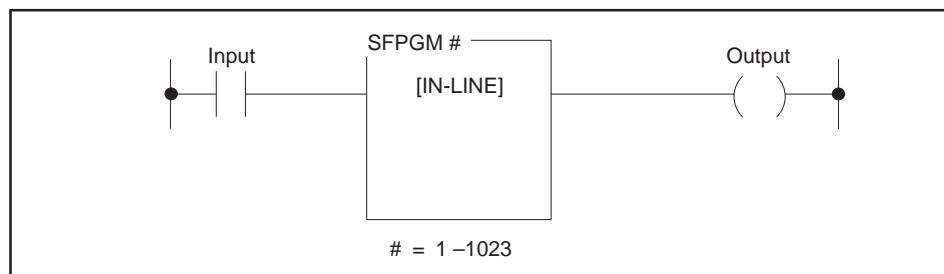



Figure 7-1 SFPGM Instruction Format

### Priority/non-priority SF Programs

When power flow to the RLL SFPGM instruction (when not marked as in-line) transitions from off to on, the output from the instruction is examined. If the output is off and the SF program is not executing, the SF program is placed in the queue for execution.

- After the SF program executes, the output turns on.
- The SF program does not execute again until the input to the SFPGM instruction transitions from off to on.

 <b>CAUTION</b>
<p>Following a transition from PROGRAM to RUN, and with the input on during the first execution of the RLL SFPGM instruction, the SF program is queued for execution.</p> <p>The SF program executes to completion only as long as the input remains on.</p> <p>Make sure the input to the SFPGM instruction is not turned off until after the SF program has executed to completion and the output has turned on.</p>

## Executing Special Function Programs (continued)

---

### In-Line Execution of Compiled SF Programs

With CPUs that support PowerMath, an SFPGM or SFSUB box instruction can be marked for “in-line” execution if the referenced SF program or SF subroutine has been compiled. When power flow is on for an in-line SFPGM or SFSUB box, it executes immediately as part of the RLL scan. The result of the box’s execution is available to the next element of the current RLL scan. Cyclic SF programs cannot be marked for in-line execution.

### Cyclic Programs

When power flow to the RLL SFPGM instruction transitions from off to on, the cyclic SF program is placed in the queue for execution.

- After the cyclic SF program executes one time, the output turns on. The SF program automatically re-queues for execution, based on the programmed cycle time. This process continues as long as the input to the RLL SFPGM instruction is on.
- The output remains on until the input to the RLL SFPGM instruction is turned off.
- A cyclic SF program is removed from the queue when it completes a scheduled cycle and the SFPGM instruction’s input is off.

### Restricted Programs Called by Loops

You can program a loop to call an SF program to do a calculation on any constant, variable, or I/O point. When you program a loop, you can schedule the SF program call to be made when the process variable, setpoint, or output is accessed.

**Calculation Scheduled on Setpoint** When the loop is in auto or cascade mode, the SF program is called at the sample rate and T2 (defined in Section 7.6) always equals 2. When the loop is in manual mode, the SF program is not called for execution.

**Calculation Scheduled on Process Variable** When the loop is in auto, cascade, or manual mode, the SF program executes every 2.0 seconds or at the sample rate, whichever is less. The SF program is called at least every 2 seconds to monitor/activate the PV alarms associated with the loop, even though loop calculations are not being performed.



---

In the case of a loop sample time greater than 2.0 seconds, the SF program is called at a 2.0 second-interval, with T2 = 3 indicating that the SF was called on PV. This allows for PV manipulation before PV alarming occurs in the loop. When it is time to perform the loop calculation, T2 equals 2 to indicate that the loop calculation is about to be performed. This allows for manipulation of PV and setpoint before the loop calculation executes. If the loop sample time is less than 2.0 seconds, T2 always equals 2.

---

**NOTE:** SF programs called on PV or SP execute after PV and SP are determined by the loop, but before any processing is performed, based on the values obtained. This allows SF programs to manipulate the PV or SP before the loop uses them for output adjustments.

---

**Calculation Scheduled on Output** When a loop with a sample time of less than 2.0 seconds calls an SF program, the SF program is actually called twice for every loop calculation.

- After PV and SP are determined, the SF program is called on SP (T2 = 2). This call allows for PV and SP manipulation before PV alarming and loop calculations are run. The loop calculation is then performed and the resultant output value is placed into the loop-output variable (LMN).
- Next, the SF program is then called on output (T2 = 5) to allow for manipulation of the loop output value in LMN before this value is written to the loop-output address.

If the sample time of the loop is greater than 2.0 seconds, the same applies except that the SF program is called at least every 2.0 seconds and T2 = 3 if it is not time to perform a loop calculation.

Refer to Section 7.6 for a description of T-Memory.

Restricted  
Programs Called  
by Analog Alarms

You can program an Analog Alarm to call an SF program to do a calculation on any constant, variable, or I/O point. The Analog Alarm is called at the SF program sample rate.

## 7.5 Executing Special Function Subroutines

---

### Calling SF Subroutines

An SF subroutine can be called for execution by an SF program or another SF subroutine through the CALL statement. See Section 7.12 for information about how the CALL statement operates.

Additionally, an SF subroutine can be called from RLL using the SFSUB RLL instruction. Refer to section 6.52 for information about how the SFSUB RLL instruction operates.

### Designing SF Subroutines

SF subroutines allow you to design modular programs. A calculation required in several places in the program may be placed in a subroutine and called by the routine number whenever it is needed. For example, consider a calculation such as:

$$y = 0.929783 * x + 2 * \left[ \frac{e^z + \ln(x)}{x^{0.25}} \right]^{0.5}$$

where  $y$  is the output and  $x$  and  $z$  are inputs. This calculation could be placed in an SF subroutine as follows:

#### SF Subroutine 0113

```
MATH P1: = 0.929783 * P2 + 2 * ((exp(P3) + ln(P2))/(P2 ** 0.25))** .5
```

where  $P1$  corresponds to the  $y$  output, and  $P2$  and  $P3$  correspond to the  $x$  and  $z$  inputs respectively. The SF subroutine 0113 would be called by a CALL statement as shown in the following example.

```
CALL SFSUB .. : 113   P1 ..... : V100.
      P2 ..... : T15   P3 ..... : V202.
      P4 ..... :      P5 ..... :
```

where  $V100.$  corresponds to the  $y$  output,  $T15$  corresponds to the  $x$  input, and  $V202.$  corresponds to the  $z$  input.

---

When you reference a parameter (P1, P2, etc.) in a SF subroutine you should not include the “.” suffix. A reference without this suffix, e.g., “P1”, instructs the controller to use the parameter according to the data type (integer or real) that was specified when the subroutine was called. For example, if parameter P1 is coded as “V100.” in the CALL statement, then a reference to P1 in the called subroutine would access the value at V100-V101 as a real number. If, on the other hand, P1 is coded as “V100” (without the “.” suffix) in the CALL statement, then the same reference to P1 in the called subroutine would access the value at V100 as an integer. In both cases the expected operation occurs.

If you reference a SF subroutine parameter using the “.” suffix, e.g. “P1.”, you are instructing the controller to ignore the parameter’s data type, as specified in the CALL statement, and to use the parameter as a real number. If in fact the CALL statement had coded P1 as V100 (a 16-bit integer) and the subroutine referenced parameter one as “P1.”, the subroutine would access the value at V100-V101 as a real number. (It would not convert V100 from integer to real and use the converted result.) In almost all cases this is not the expected operation.

Table 7-2 summarizes the effect of the “.” suffix when used on a parameter reference.

Table 7-2 Specifying Real or Integer Parameters

<b>Data Type Specified in CALL statement</b>	<b>Parameter Reference in SF Subroutine</b>	<b>Data Type Used in Calculation</b>
real (V100.)	Pn	real
integer (V100)	Pn	integer
real	Pn.	real
integer	Pn.	real, no conversion

## 7.6 Memory Usage by SF Programs

---

When an SF program is called, the operating system automatically allocates a block of temporary memory, T-Memory, to the program for the duration of that program. When the program terminates, the T-Memory allotted for that program clears.

T-Memory is 16 words long. Each word contains the following information.

- T1 — SF program Program Number.
- T2 — Code indicating how a program is called:
  - 1 = RLL program
  - 2 = SF program scheduled on a loop setpoint
  - 3 = SF program scheduled on a loop process variable
  - 4 = SF program on an analog alarm
  - 5 = SF program scheduled on a loop output
- T3 — If the SF program is called from a loop, then T3 contains the number of that loop from which the program was called. If the SF program is called from an analog alarm, T3 contains the number of that analog alarm. Otherwise, T3 contains 0.
- T4 and T5 — If the SF program is called from a loop, analog alarm, or is a cyclic SF program, T4 and T5 contain the cycle period in seconds stored as a real (32-bit) value. Otherwise, T4 and T5 contain 0.

- 
- T6 — If the SF program is called from a loop, analog alarm, or is a cyclic SF program, T6 contains 1 when the loop, analog alarm, or SF program has overrun. Otherwise, T6 contains 0.
  - T7 — If the SF program is called from a loop, analog alarm, or is a cyclic SF program, T7 is set to 1 if this is the first time the SF program is called. T7 is also set to 1 if this is the first time the loop executes after a commanded restart, or following a program-to-run transition, or following a mode change (i.e., manual to auto, auto to manual). Otherwise, T7 contains 0.
  - T8–T16 — No data is written to these words. You can use them any time during the program to store intermediate calculations.

You can use all 16 words in your SF program. You can read the information stored in T1–T7 by the controller; or if you prefer, store data into these locations as you can with T8–T16, writing over the information written by the controller.

## 7.7 Entering SF Program Header with TISOFT

The general steps for entering an SF program are listed below. Refer to your TISOFT user manual for detailed instructions.

- Select the SF program option (SFPGM-F8) from the menu on your programming device. The SF Program/Subroutine Directory is displayed.
- Select the SF program that you want to enter (Program 1, Program 2, etc.). Then press SHOW-F2 to access the SF program.
- The screen displays the program format. The program format consists of a header section and a program section, as illustrated in Figure 7-2.

```
TITLE: TANK 50                                SF PROGRAM: 1022
      CONTINUE ON ERROR (Y, N): NO
      ERROR STATUS ADDR: (Y, C, WY, V): V500
      PROGRAM TYPE (N, P, C, R): CYCLIC
      CYCLE TIME (SEC): 1.0
-----
00001 *      THE COMMENT STATEMENTS (*) EXPLAIN THE
00002 SCALE  BINARY INPUT ..... : WX10          SCALED RESULT . : V1
           LOW LIMIT ..... : 0.0              HIGH LIMIT ..... : 20.0
           20% OFFSET ..... : YES            BIPOLAR ..... : NO
00003 IF     V1 = 5
00004 PRINT  PORT ..... : 1                    MESSAGE: ..... :
           "TANK LEVEL IS LOW. PRESENT LEVEL IS" V1 "FT."
00005 MATH   LKC1. : = 3.0
00006 ELSE
00007 MATH   LKC1. : = 1.0
00008 ENDIF
**** END ****

S-MEMORY AVAILABLE: 1808                      DISBL
                                           555 NEWPGM
EXIT-F1  EDIT-F2                            FIND-F4 DELST-F5 INSST-F6 COMMNT-F7 DIS/CP-F8
```

Header

Program

Figure 7-2 Special Function Program Format

- 
- Press EDIT-F2 to enter SF program edit mode.
  - Enter a title for the program. The title is optional and can be left blank.
  - The **CONTINUE ON ERROR** field specifies if the program is to continue to run when an error occurs. Enter Y in this field to have the program continue when an error occurs. Enter N in this field to have program stop when an error occurs. See Section 7.8 for a discussion of error reports.
  - The **ERROR STATUS ADDRESS** field specifies how error conditions are handled. In order to have an error code written when a program error occurs, you must specify a V-Memory location or a word output (WY) in this field. If you enter a control relay or discrete output point in this field, then this point is set when an error occurs. Refer to Section 7.8 for a discussion of error reports.
  - The **PROGRAM TYPE** field specifies the program type. Enter N for a non-priority program, P for a priority program, C for a cyclic program, or R for a restricted program. Refer to Section 7.1 for a discussion of SF program types.
  - The **CYCLE TIME** field sets the periodicity of the program execution. For a cyclic program, enter the cycle time in seconds (0.5–6553.5). For example, a program with a cycle time of 5 seconds is executed every five seconds. Note that the controller rounds the value that you enter up to the next 0.5 second interval.
  - Save the header information, and then proceed to the program section.

## 7.8 Reporting SF Program or SFSUB RLL Instruction Errors

---

When you enter an SF program or an SFSUB RLL instruction, you have the option of specifying how to report errors. You assign an address in the ERROR STATUS ADDRESS field of the SF program header, (described in Section 7.7) or in the ER field of the SFSUB RLL instruction (described in Section 6.50). In this field, you can specify a control relay (C), a discrete output (Y), a V-Memory location, or a word output (WY).

### Reporting Errors with the SFEC Variable

The Special Function Error Code (SFEC) variable may be used to read from or write to the error code for an SF program or for an SFSUB RLL instruction. Each SF program or SFSUB RLL instruction contains one SFEC variable. All references to SFEC within an SF program or an SFSUB RLL instruction's parameters, or within any SF subroutine called by the SF program or SFSUB RLL instruction, refers to this single SFEC variable. (The programming system may require that you specify a number when you enter the SFEC variable name, e.g., SFEC1. The programmable controller ignores this number.)

When an SF program or an SFSUB RLL instruction is queued for execution, the SFEC for that SF program or SFSUB RLL instruction is cleared to zero. If an error occurs during execution, the error code associated with the error (refer to Appendix F) is written to SFEC. Errors can be detected by the operating system or they can be detected by the user program. If an error is detected by the user program, you indicate it to the system by an assignment to SFEC in a MATH or IMATH statement.

If you select NO in the CONTINUE ON ERROR field when you enter an SF Program, or, if you select STOP ON ERROR when you enter an SFSUB RLL instruction, assigning a non-zero value to the SFEC variable causes the SF program or SFSUB RLL instruction to terminate. (You can force termination of the SF program or SFSUB instruction by having your program or subroutine assign a non-zero value to SFEC.)

If you select YES in the CONTINUE ON ERROR field when you enter an SF Program, or CONTINUE ON ERROR when you enter an SFSUB RLL instruction, writing to the SFEC variable does not cause the SF program or SFSUB RLL instruction to terminate. In this case, your SF program or SF subroutine can examine the SFEC variable and take corrective active, as applicable. However, you are not able to force termination by writing to SFEC.

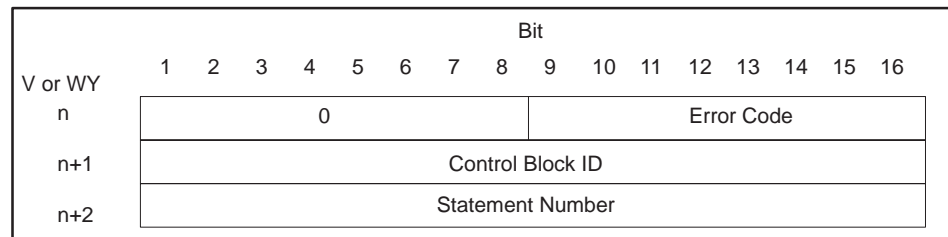
### Reporting Errors with Discrete Points

If you specify a control relay (C) or discrete output (Y) in the ERROR STATUS ADDRESS field when you enter an SF program, or in the ER field when you enter an SFSUB RLL instruction, this discrete point is set to one if an error occurs. No other report of the error is made; no error code is written.



**Reporting Errors  
with V or WY  
Memory**

If you specify a V-Memory location (Vn) or word output (WYn) in the ERROR STATUS ADDRESS field when you enter an SF program, or in the ER field when you enter an SFSUB RLL instruction, then three words of memory are reserved, as shown in Figure 7-3.



**Figure 7-3 Word Specification for SF Program Errors**

The error code is contained in the low-order eight bits of the first word (word n) in the group. Appendix F lists the error codes and their definitions.

The second word in the group (word n+1) is the control block ID. The controller assigns a control block for each loop, analog alarm, SF program and SF subroutine. The header in each control block stores information in the following format.

- Bits 1 and 2 (in word n+1) always contain zero.
- The next four bits (bits 3–6 in word n+1) indicate the control block type as follows.

0000: Loop Control Block

0001: Analog Alarm Control Block

0010: SF program Control Block

0011: SF subroutine Control Block or SFSUB RLL Instruction

0100 through 1111 are not used

- The next 10 bits (bits 7–16 in word n+1) are allocated for the Control Block Number or SFSUB RLL Instruction number.

The third word in the group (word n+2) contains the statement number of either the last SF statement to be executed correctly, or the statement number of the statement executing when the error occurred. (The Control Block ID indicates the SF program or SF subroutine that contains the statement.)

## 7.9 Entering Special Function Programming Statements

Each SF statement has one or more fields in which you enter data when you use the statement in an SF program. For each field, you enter a field type and a field descriptor, which are defined in Table 7-3.

Table 7-3 SF Statement Field Entry Definitions

Field Type		
Address	Element	Elements are comprised of a data type and a number. A period following the element designates the element as an address of a real number. No period designates the element as an address of an integer. Examples of elements are: V100 <i>or</i> V100. <i>or</i> LPVH1. <i>or</i> C102, etc.
	Address Expression	An address expression is a logical group of tokens evaluating to an address, where a token is the smallest indivisible unit, e.g., an element, operator, constant, parenthesis, etc. PowerMath CPUs also support 16-bit unsigned integers (identified with a U suffix, e.g., V105U) and 32-bit signed integers (identified with an L suffix, e.g., K15L) Examples of address expressions are V100(3) evaluates to the address V102 V100.(2) evaluates to the address V101. V102.(T16 + 10:) if T16 = 2, evaluates to the address V124. V105U, K15L (PowerMath CPUs only)
Value	Literal Constant	A literal constant is a real or integer number, such as 78, 3.468, 32980, etc.
	Value Expression	A value expression is a logical group of tokens evaluating to a value, where a token is the smallest indivisible unit, e.g., an element, operator, constant, parenthesis, etc. Examples of value expressions are V100:= LKD2. **3 (V100 + K774) (V102(3)) 32768U (PowerMath CPUs only) 600L (PowerMath CPUs only)
Field Descriptor		
Integer only	This field only accepts an integer value (e.g., 3761 <i>or</i> (V11 + 7)) or an address containing an integer value (e.g., V100 <i>or</i> WX88 <i>or</i> V100(2); PowerMath CPUs also support unsigned 16-bit integers and signed 32-bit integers (e.g., V105U, K15L).	
Real only	This field only accepts a real value (e.g., 33.421) or an address containing a real value, (e.g., V121. <i>or</i> V888. (13)).	
Integer/Real	This field accepts a real or integer value or an address containing a real or integer value.	
Writeable	This field only accepts a writeable address, (e.g., WY1000 <i>or</i> V23. <i>or</i> C55). Read-only addresses, (e.g., K551 <i>or</i> WX511 <i>or</i> X69) are not allowed.	
Optional	An entry in this field is optional and the field can be left blank.	
Bit	This field only accepts an address that contains a bit value (e.g., X17 <i>or</i> C200 <i>or</i> Y91).	

Figure 7-4 shows an example of the entries that are valid for the fields in the FTSR-IN statement.

FTSR-IN	Input . . . . . : A	Register start . . : B
	Register length . . : C	Status bit . . . . . : D
A =	Address	Integer
B =	Address	Integer, writeable
C =	Address or value	Integer
D =	Element	Bit, writeable
	└──────────┘	└──────────┘
	Field type	Field descriptors
Field A		
Field type	Enter either an element or an address expression.	
Field descriptors	The entry in field A must contain the address of an integer.	
Field B		
Field type	Enter either an element or an address expression.	
Field descriptors	The entry in field B must contain the address of an integer, and that address must be writeable.	
Field C		
Field type	Enter an element, address expression, literal constant, or a value expression.	
Field descriptors	The entry in field C must contain an integer (literal constant or value expression) or an address to an integer (element or address expression).	
Field D		
Field type	Enter only an element.	
Field descriptors	The entry in field D must contain the address of a bit and that address must be writeable.	

Figure 7-4 Example of Valid Entries for the FTSR-IN Statement

## 7.10 Convert BCD to Binary

**BCDBIN Description** The Convert BCD to Binary statement converts binary coded decimal (BCD) inputs to a binary representation of the equivalent integer. The BCDBIN format is shown in Figure 7-5.

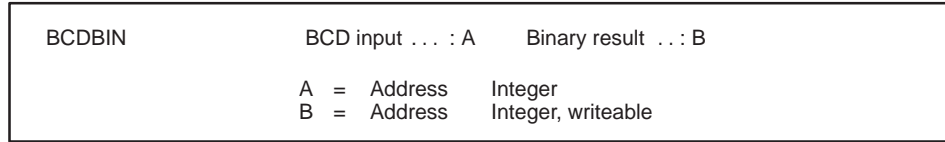


Figure 7-5 BCDBIN Format

- A is the memory location of the BCD word to be converted.
- B is the memory location of the integer value after conversion.

**BCDBIN Operation** The operation of BCDBIN is described below and illustrated in Figure 7-6.

- Each time the BCDBIN statement executes, the four digits of the BCD value located in the address specified by A are converted to the binary representation of the equivalent integer value.
- The result is stored in the address specified by B.

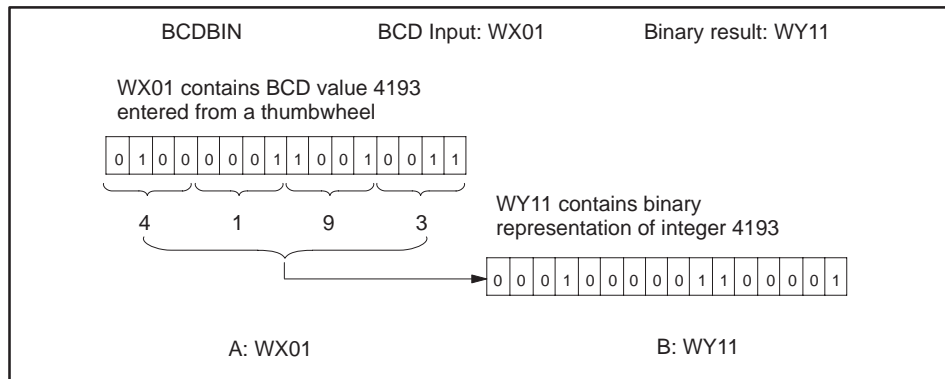


Figure 7-6 Example of BCDBIN Operation

## 7.11 Convert Binary Inputs to BCD

**BINBCD Description** The Convert Binary Inputs to BCD statement (Figure 7-7) converts the binary representation of an integer to the equivalent Binary Coded Decimal (BCD) value. Values up to 9999 are converted to equivalent BCD values.

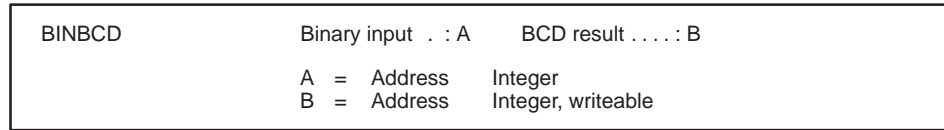


Figure 7-7 BINBCD Format

- A is the memory location of the integer to be converted.
- B is the memory location of the BCD word after conversion.

**BINBCD Operation** The operation of BINBCD is described below and illustrated in Figure 7-8.

- Each time the BINBCD statement executes, the integer located in the address specified by A is converted to BCD.
- An error occurs if the input value contained in A is less than zero or greater than 9999.
- The BCD value is stored in the address specified by B.

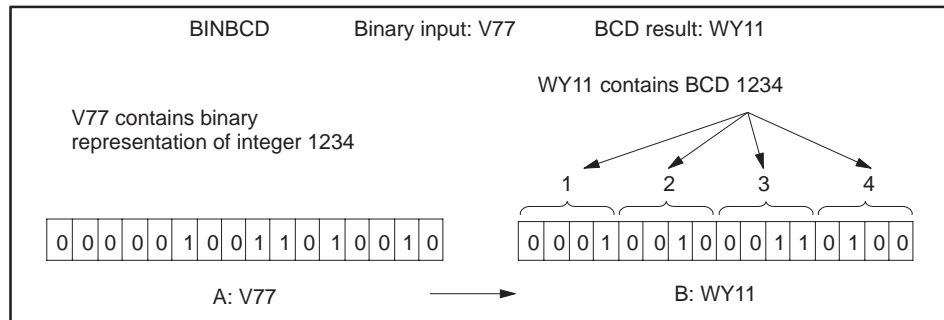


Figure 7-8 Example of BINBCD Operation

## 7.12 Call Subroutine

---

**CALL Description** The CALL statement calls an SF subroutine for execution. Up to five parameters may be passed to the subroutine by the CALL statement. The CALL format is shown in Figure 7-9.

CALL	SFSUB	A	P1 ...: B
	P2 .....	:C	P3 ...: D
	P4 .....	:E	P5 ...: F
	A =	Literal constant	Integer
	B-F =	Address <i>or</i> value	Integer/real, optional

Figure 7-9 CALL Format

- A is the number of the SF subroutine to be called and ranges from 1 to 1023.
- B-F are the fields in which constant values or variables are specified to be passed between the SF subroutine that is called, and the SF program or the SF subroutine that contains the CALL statement.

**CALL Operation** The operation of the CALL statement is described below.

- Up to five parameters may be specified in the P (B-F) fields to be passed to the SF subroutine.

The P fields are optional and can be left blank. If you have fewer than five entries for the P fields, enter them in order. That is, do not skip any of the P fields.

To specify a real value rather than an integer in a P field, place a period after the variable. For example, P1...: V100. passes a real number to P1; P2...: V102 passes an integer. Table 7-4 shows how data types are passed to an SF subroutine.

To specify a long (32-bit) signed integer, place an L after the variable. For example, V200L specifies a 32-bit (long) integer instead of a real value. (This integer type is supported by PowerMath CPUs only.)

To specify an unsigned 16-bit integer, place a U after the variable. For example, V202U specifies a 16-bit unsigned integer. (This integer type is supported by PowerMath CPUs only.)

- When the CALL statement executes, the following actions occur:

Control is transferred to the specified SF subroutine. Any parameters specified in the P fields are read by the SF subroutine.

Statements within the SF subroutine execute, and parameters in the P fields that are modified by the SF subroutine are updated. Then control transfers back to the SF program that called the SF subroutine.

Table 7-4 Specifying Real or Integer Parameters

Data Type Specified in CALL Statement	Data Type Specified in SF Subroutine*	P Data Type Used in SF subroutine*
real (	real (	real (
real (	integer	real (
integer	real (	real (
integer	integer	integer

\*See Section 7.5 for more information about specifying data types in SF subroutines.

### CAUTION

Subroutines may be nested to four levels. If the limit of four levels is exceeded, an error results.

This causes termination of the SF program and all subroutines prior to the one that exceeded the level.

CONTINUE ON ERROR does not override this condition. Ensure that you do not nest subroutines for more than four levels.

## 7.13 Correlated Data Table

### CDT Description

The Correlated Data Table statement compares an input value (the input) to a table of values (the input table), and locates a value in the input table that is greater than or equal to the input. The CDT then writes the value located in a second table (the output table), that is correlated with the value located in the input table, to an output address (the output). The CDT format is shown in Figure 7-10.

CDT	Input . . . . . : A	Output . . . . . : B
	Input table . . . : C	Output table . . . : D
	Table length . . : E	
	A = Address	Integer/real
	B = Address	Integer/real, writeable
	C = Address	Integer/real
	D = Address	Integer/real
	E = Address <i>or</i> value	Integer

Figure 7-10 CDT Format

- A is the input address.
- B is the address to which the output value is written.
- C is the starting address for the input table. When used in a compiled SF program or subroutine, you must specify a static table; that is, the table's base address must be a V, K, G, VMS, or VMM address.
- D is the starting address for the output table. When used in a compiled SF program or subroutine, you must specify a static table; that is, the table's base address must be a V, K, G, VMS, or VMM address.
- E is the length of each table and must be a value greater than zero. When used in a compiled SF program or subroutine, the table length must be specified as a value.



## CDT Operation

CDT statement operation is described here and illustrated in Figure 7-11.

- When the CDT is executed, the CDT compares the value of an input element specified in A to a pre-existing table of values having a starting address specified in C. The first value in the input table that is greater than or equal to the input is located.
- A value in a second pre-existing table (starting address specified in D) that correlates with the selected value in the input table is written to an output address specified in B.
- The input table must be in ascending order. That is, the lowest value is located in the lowest memory location and the highest value is located in the highest memory location.
- Table length E depends upon the memory location that you choose, and how much memory you allocated if the memory is user configurable.
- Both tables must have the same number of entries.

CDT	Input . . . . . : V1	Output . . . . . : V2
	Input table . . . : K64	Output table : K84
	Table length . . : 7	
	Input table	Output table
	K64 = 20	K84 = 48
	K65 = 28	K85 = 23
	K66 = 34	K86 = 62
	K67 = 39	K87 = 98
(Input value) V1=40	K68 = 43 →	K88 = 72
	K69 = 47	K89 = 65
	K70 = 50	K90 = 41
		(Output value) V2=72

Figure 7-11 CDT Statement Example

The input address V1 contains the value 40. The value in the input table that is greater than or equal to 40 is 43, contained in K68. The correlated value in the output table is in K88. The value written to the output address V2 is 72.

## 7.14 Exit on Error

---

**EXIT Description**      The EXIT statement allows you to terminate an SF program or SF subroutine and have an error code logged. The EXIT format is shown in Figure 7-12.

EXIT	Errcode .. : A
A = Literal constant	Integer, optional

**Figure 7-12 EXIT Format**

- A contains the value of the error code and can range from 0 to 255.

**EXIT Operation**      The operation of the EXIT statement is described below.

- When the SF program encounters the EXIT statement, program execution terminates. If an SF subroutine encounters the EXIT statement, the subroutine and calling SF program are terminated.
- If you use the EXIT statement in conjunction with an IF statement, you can terminate the program under specific conditions.

You can leave A blank and the current error code is written to the ERROR STATUS ADDRESS that you specify in the SF program header. If this address is a discrete point, it turns on.

You can define an error condition and assign it an error code 200–255 (codes 0–199 are reserved). When the EXIT statement executes, the program terminates and this error code is written to the ERROR STATUS ADDRESS. If this address is a discrete point, it turns on.

## 7.15 Fall Through Shift Register—Input

**FTSR-IN Description** The Fall Through Shift Register Input statement operates an asynchronous shift register. The shift register is essentially a table of 16-bit words. The FTSR-IN moves a word into the shift register each time the statement executes. The FTSR-IN is used in conjunction with the Fall Through Shift Register Output statement (FTSR-OUT) that moves words out of the shift register. The FTSR-IN format is shown in Figure 7-13.

FTSR-IN	Input . . . . . : A	Register start . . : B
	Register length . . : C	Status bit . . . . . : D
	A = Address	Integer
	B = Address	Integer, writeable
	C = Address <i>or</i> value	Integer
	D = Element	Bit, writeable

Figure 7-13 FTSR-IN Format

- A is the input address from which the words are moved.
- B is the starting address for the shift register. Four words (B through B + 3) are automatically reserved for the operation of the statement and make up the header of the shift register. The first word of your data is shifts into address B + 4.

---

**NOTE:** Do not write data to the header fields. The shift register does not operate correctly if any of these fields is modified by an external action. These fields may be redefined in future software releases.

---

- C is the length of the table. If a constant is used, it must be greater than zero. The total length of the shift register is C + header.
- D is the status bit and can be C or Y. The bit specified by D turns on when the register is full. The bit (D + 1) is automatically reserved as a second status bit. The bit specified by (D + 1) turns on when the register is empty.

## Fall Through Shift Register—Input (continued)

---

### FTSR-IN Operation

The operation of the FTSR-IN statement is described below.

- FTSR-IN is used in conjunction with an FTSR-OUT; you must use the same corresponding values for register start, register length, and status bit in the two FTSR statements.
- A is the input address from which the words are moved into the shift register.
- The starting address B determines the memory area in which the shift register is located. The first word of your data shifts into address B + 4.
- The four words (B through B + 3) are automatically reserved for the operation of the shift register.

(B) contains the Count, which equals the current number of entries in the shift register.

(B + 1) contains the Index, which acts like a pointer to indicate the next available location of the shift register into which a word can be shifted. When the Index equals zero, the next available location is (B + 4); when the Index equals one, the next available location is (B + 5), and so on.

(B + 2) contains the Length, which equals the maximum size of the shift register in words.

(B + 3) contains the Checkword. The checkword is used internally to indicate whether the FTSR is initialized.

- The register length C determines the size of the shift register. The register length depends upon the memory location that you choose and how much memory you have allocated (if the memory is user-configurable).
- The status bit specified by D is turned on to indicate that the register is full. The bit (D + 1) is automatically reserved as a second status bit and turns on whenever the shift register is empty.

Use the same status bits for the FTSR-IN that you use for the FTSR-OUT. FTSR-IN sets D when the register fills. FTSR-OUT clears this bit as the function executes. FTSR-OUT sets (D + 1) when the register is empty. FTSR-IN clears this bit.

- 
- If the shift register is empty, status bit D is off and (D + 1) is on.
  - When the FTSR-IN executes, the following actions occur.

The word currently in memory location A is shifted into the location specified by the Index.

The Count and the Index are each incremented by one.

Status bit (D + 1) turns off.

- Each time the FTSR-IN executes, another word moves into the next available location; the Index and the Count increment by one. When the Index equals the length, it resets to zero after the next execution by the FTSR-IN.
- When the shift register is full, another word cannot be shifted in until one is shifted out by the FTSR-OUT statement.
- When the shift register is full, status bit D turns on. If you attempt to shift in another word, an error generates. (Appendix F, error 87).
- You can use FTSR-OUT to remove words from the shift register before all locations are full. You can use FTSR-IN to shift more words into the shift register before all words are removed.

Figure 7-14 illustrates the operation of the FTSR-IN statement.

## Fall Through Shift Register—Input (continued)

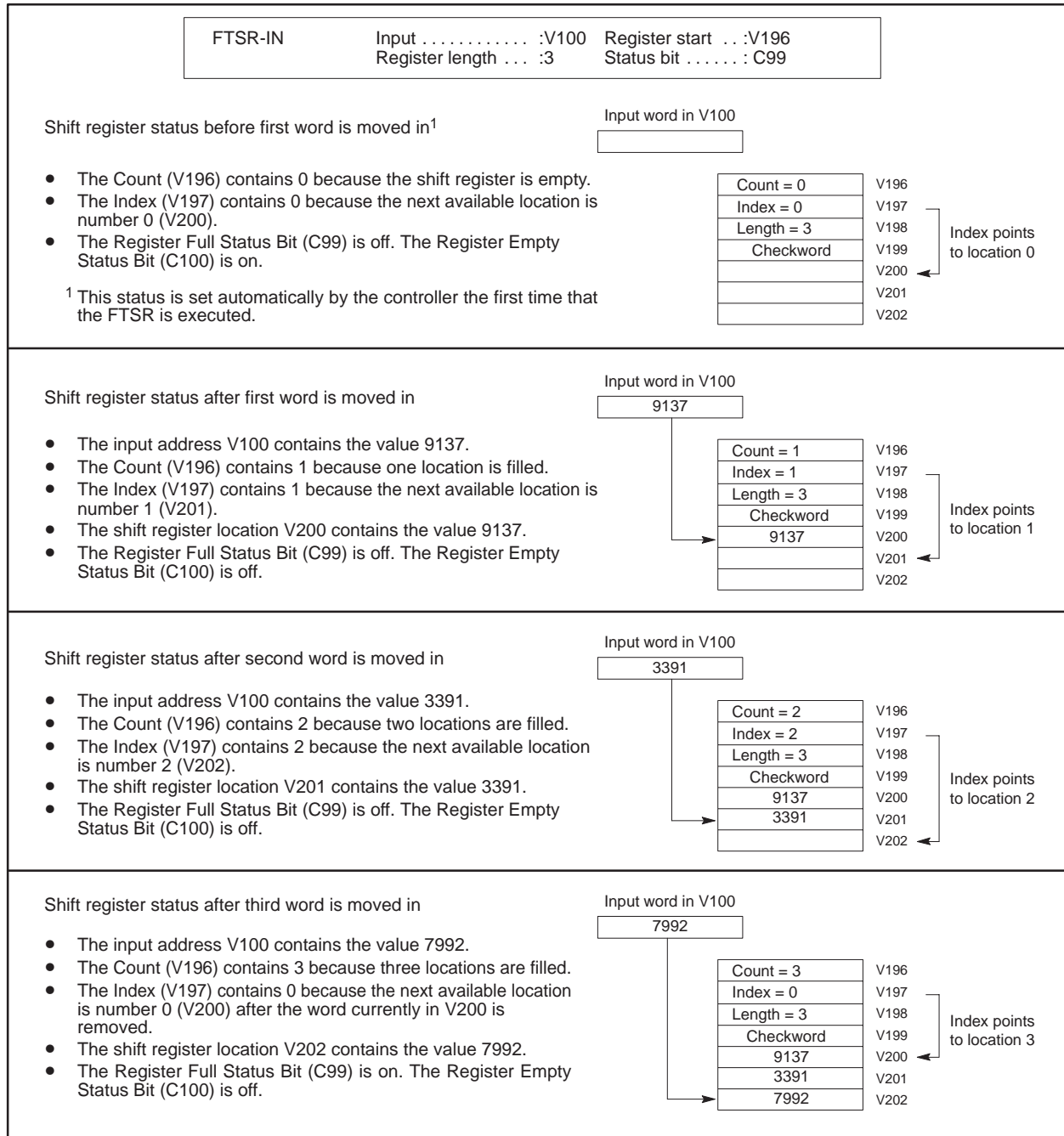


Figure 7-14 Example of FTSR-IN Operation

## 7.16 Fall through Shift Register—Output

---

### FTSR-OUT Description

The Fall Through Shift Register Output statement operates an asynchronous shift register. The shift register is essentially a table of 16-bit words. The FTSR-OUT moves data out of the shift register each time the statement executes. The FTSR-OUT is used in conjunction with the Fall Through Shift Register Input statement (FTSR-IN) that moves words into the shift register. Figure 7-15 shows the FTSR-OUT format.

FTSR-OUT	Register start . . . : A Register length . . : C	Output . . . : B Status bit . . : D
	A = Address	Integer, writeable
	B = Address	Integer, writeable
	C = Address <i>or</i> value	Integer
	D = Element	Bit, writeable

**Figure 7-15 FTSR-OUT Format**

- A is the starting address for the shift register. The four words (A through A + 3) are automatically reserved for the operation of the statement and make up the header of the shift register.

---

**NOTE:** Do not write data to the header fields. The shift register does not operate correctly if any of these fields is modified by an external action. These fields may be redefined in future software releases.

---

- B is the output address to which the words are moved.
- C is the length of the table. If a constant is used, it must be greater than 0.
- D is the status bit and can be C or Y. The bit specified by D is turned on when the register is full. The bit (D + 1) is automatically reserved as a second status bit. The bit specified by (D + 1) is turned on when the register is empty.

## Fall Through Shift Register—Output (continued)

---

### FTSR-OUT Operation

The operation of the FTSR-OUT statement is described below.

- FTSR-OUT is used in conjunction with a FTSR-IN; you must use the same corresponding values for register start, register length, and status bit in the two FTSR statements.
- Starting address A determines the memory area in which the shift register is located. The first word of user data is located in address A + 4.
- The four words (A through A + 3) are automatically reserved for the operation of the shift register.

(A) contains the Count, which equals the current number of entries in the shift register.

(A + 1) contains the Index, which acts like a pointer to indicate the next available location of the shift register into which a word can be shifted. When the Index equals zero, the next available location is (A + 4); when the Index equals one, the next available location is (A + 5), and so on.

(A + 2) contains the Length, which equals the maximum size of the shift register in words.

(A + 3) contains the Checkword. The checkword is used internally to indicate whether the FTSR has been initialized.

- B is the output address into which the words are moved.
- The register length C determines the size of the shift register. The register length depends upon the memory location that you choose and how much memory you allocated (if the memory is user configurable).
- D is the status bit and can be C or Y. The bit specified by D turns on to indicate that the register is full. The bit (D + 1) is automatically reserved as a second status bit and turns on whenever the shift register is empty.

Use the same status bits for the FTSR-OUT that you use for the FTSR-IN. FTSR-IN sets D when the register is full. FTSR-OUT clears this bit as the function executes. FTSR-OUT sets (D + 1) when the register is empty. FTSR-IN clears this bit.



- 
- If the shift register contains one or more words, the Count equals the number of current entries. The Index points to the next available location of the shift register into which a word can be moved. Status bit (D + 1) is off. Status bit D is on if the shift register is full.
  - When the FTSR-OUT executes, the following actions occur.

The oldest word in the shift register shifts into memory location B.

The Count decrements by one.

The Index is unchanged and continues to point to the next available location into which a word can be moved.
  - Each time the FTSR-OUT executes, another word moves out of the shift register and the Count is decremented by one. The Index remains unchanged.
  - After the shift register is empty, the Index and Count contain zero. Status bit D turns off and status bit (D + 1) turns on. If you attempt to shift a word out of an empty shift register, an error is generated (Appendix F, error 86).
  - You can use FTSR-OUT to remove words from the shift register before all locations are full. You can use FTSR-IN to shift more words into the shift register before all words are removed.

Figure 7-16 illustrates the operation of the FTSR-OUT statement.

## Fall Through Shift Register—Output (continued)

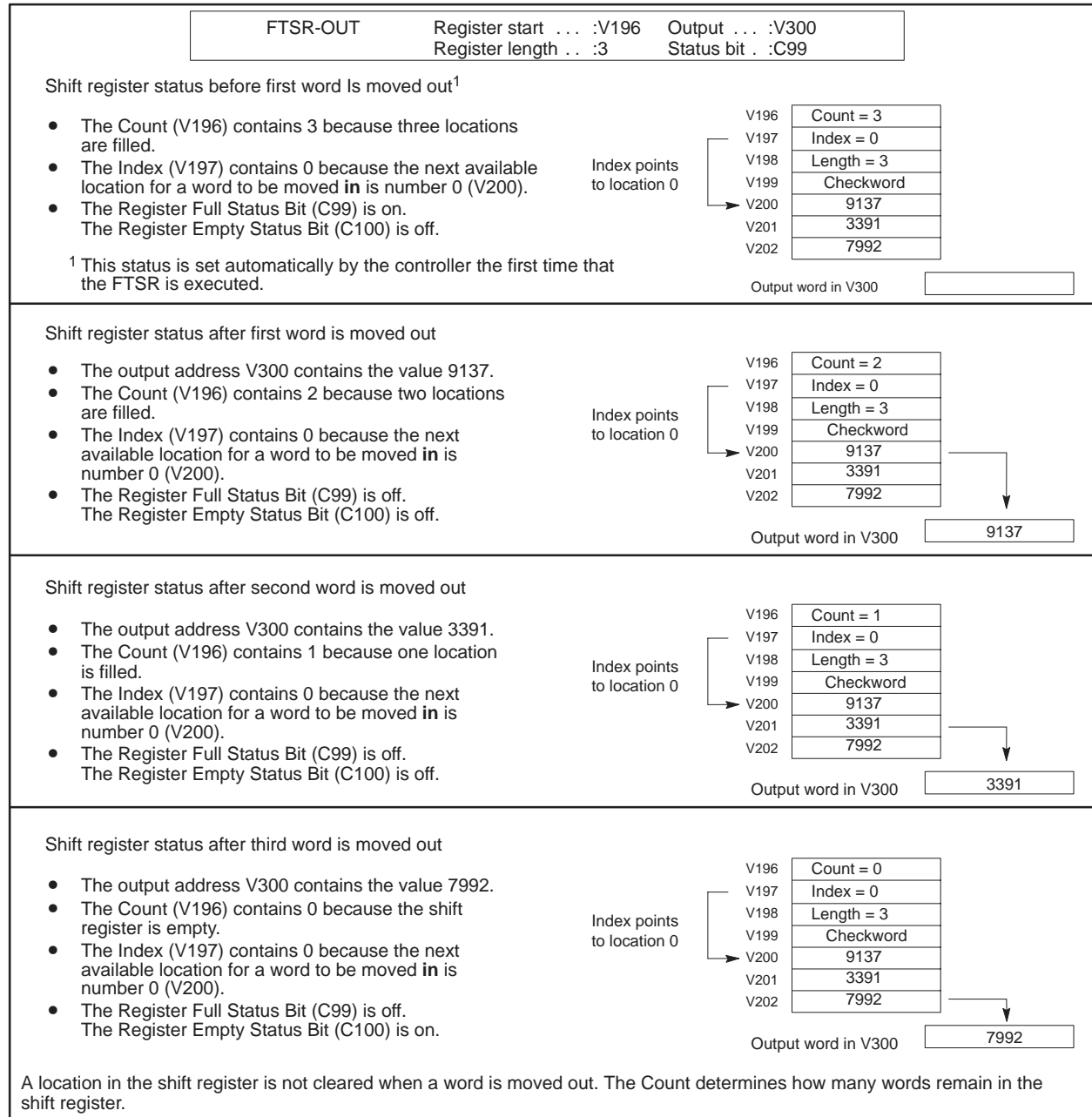


Figure 7-16 Example Of FTSR-OUT Operation

## 7.17 Go To/Label Function

The GOTO statement continues program execution at a specified LABEL statement. The GOTO and the LABEL statements are always used together. The format of the two statements is shown in Figure 7-17.

GOTO	LABEL . . . . . : A
<SF Statement>	
<SF Statement>	
<SF Statement>	
LABEL	LABEL . . . . . : A
A =	Literal constant      Integer

Figure 7-17 GOTO/LABEL Format

- The <SF statement> may be any of the SF program statements.
- A is the label and can range from 0 to 65535.

When the SF program encounters the GOTO, program execution continues at the LABEL specified by A.

Figure 7-18 illustrates the use of the GOTO/LABEL statement.

00005	MATH	V100 := V500
00006	IF	V100 < 1000
00007	GOTO	LABEL      37415
00008	ELSE	
00009	GOTO	LABEL      38000
00010	ENDIF	
00011	LABEL	LABEL      37415
00012	MATH	V100 := V465/K99

Figure 7-18 Example of GOTO/LABEL Statements

### CAUTION

**Do not repeat label definitions or leave a label undefined.**

**To do so may cause the controller to enter the FATAL ERROR mode, freeze analog outputs and turn off discrete outputs.**

**Ensure that all labels have a unique definition.**

## 7.18 IF/IIF/THEN/ELSE Functions

### IF/THEN/ELSE Description

The IF or IIF (Integer IF) statement is used for the conditional execution of statements and operates in conjunction with the ELSE and the ENDIF statements. When an IF statement is used, a THEN result is understood. The IF format is shown in Figure 7-19.

**NOTE:** Integer IF operations are available only in CPUs that support PowerMath.

IF (IIF)	Free format based on the following structure:
	IF <valid MATH (or IMATH) expression> .. <THEN understood>
	<SF statement>
	<SF statement>
	...
	<SF statement>
	ELSE
	<SF statement>
	<SF statement>
	...
	<SF statement>
	ENDIF
	(The <SF statement> may be any of the SF program statements.)

Figure 7-19 IF Format

### CAUTION

**Do not use an IF (or IIF) without an ENDIF.**

**To do so may cause the controller to enter the FATAL ERROR mode, freeze analog outputs and turn off discrete outputs.**

**Ensure that all IF statements are completed with an ENDIF statement.**

### IF Operation

Figure 7-20 illustrates the operation of the IF (or IIF) statement described below.

- Each time the IF executes, the condition defined within the statement is tested.
- If the <expression> is true (non-zero), statements in the THEN section execute; any statements in the ELSE section are skipped.
- If the <expression> is false (zero), statements in the THEN section are skipped; any statements in the ELSE section execute.

- The <expression> can be any MATH expression in IF statements or IMATH expression in IIF statements. See Table 7-7 for a list of the MATH functions. The use of the assignment operator ( := ) in an expression is optional.
- The IF statement operates in conjunction with the ENDIF statement and an optional ELSE statement.
- IIF (Integer IF) allows you to code IF-THEN-ELSE blocks using an integer-only expression for the conditional. An integer expression executes faster than the equivalent floating-point expression.
- The ENDIF indicates the end of an IF-THEN-ELSE structure.
- If there is no ELSE statement, the statements between the IF and the ENDIF are treated as THEN statements.
- If an ELSE statement is used, then any statements between IF and ELSE constitute by default a THEN section. An ELSE statement indicates the end of the THEN section and the beginning of the ELSE section in an IF-THEN-ELSE structure.
- Statements between ELSE and ENDIF constitute the ELSE section in the IF statement.
- IF, ELSE and ENDIF statements may be nested to any level.

0003	IF	V1. >= 5.35 AND V1. <= 7.65
0004	PRINT	PORT.....:1 MESSAGE.....: "TANK LEVEL IS LOW. PRESENT LEVEL IS" V1 "FT."
0005	MATH	LKC1. := 3.00
0006	ELSE	
0007	MATH	LKC1. := 1.00
0008	ENDIF	
0009	IIF	V1 = 5
0010	PRINT	PORT.....:1 MESSAGE.....: "TANK LEVEL IS LOW. PRESENT LEVEL IS" V1 "FT."
0011	MATH	LKC1. := 3.00
0012	ELSE	
0013	MATH	LKC1. := 1.00
0014	ENDIF	

Figure 7-20 Example of IF/THEN/ELSE Statements

## 7.19 Integer Math Operations

**IMATH Description** The Integer Math statement executes integer arithmetic computations. The IMATH format, based on the functions in Table 7-5, is shown in Figure 7-21.

**NOTE:** Non-PowerMath CPUs do not support the following operators: = <> < <= > >=, AND, OR, and the ABS intrinsic function.

Table 7-5 IMATH Operators

Operator	Description
NOT	Unary Not—The expression “NOT X” returns the one’s complement of X.
>>	Shift right (arithmetic) <sup>1</sup>
<<	Shift left (arithmetic) <sup>1</sup>
*	Multiplication
/	Integer division— Any remainder left over after the division is truncated.
MOD	Modulo arithmetic— The expression “X mod Y” returns the remainder of X after division by Y.
+	Addition
-	Subtraction/unary minus (negation)
=	Equal. The expression X = Y returns 1 if X equals Y, and zero if not. <sup>2</sup>
<>	Not equal. The expression X <> Y returns 1 if X is not equal to Y, and zero if so. <sup>2</sup>
<	Less than. The expression X < Y returns 1 if X is less than Y, and zero otherwise. <sup>2</sup>
<=	Less Than or Equal. The expression X <= Y returns 1 if X is less than or equal to Y, and zero otherwise. <sup>2</sup>
>	Greater Than. The expression X > Y returns 1 if X is greater than Y, and zero otherwise. <sup>2</sup>
>=	Greater Than or Equal. The expression X >= Y returns 1 if X is greater than or equal to Y, and zero otherwise. <sup>2</sup>
AND	Logical AND. The expression X AND Y returns 1 if both X and Y are non-zero, and zero otherwise. <sup>2</sup>
OR	Logical OR. The expression X OR Y returns 1 if either X or Y is non-zero, and zero otherwise. <sup>2</sup>
&	Bit-by-bit AND of two integers.
	Bit-by-bit OR of two integers.
^	Bit-by-bit exclusive OR of two integers.
:=	Assignment
ABS	Math intrinsic function Absolute Value <sup>2</sup>
<sup>1</sup> See page 7-50 for an application example.	
<sup>2</sup> Supported by PowerMath CPUs only.	

IMATH      A := B
A = Address      Integer, writeable
B = Address or value    Integer

Figure 7-21 IMATH Format

IMATH Operation

Figure 7-22 shows the operation of the IMATH statement described below.

- Each time the IMATH statement executes, the calculations within the statement are made.
- The IMATH computations are executed using the rules of precedence for arithmetic operations listed in Table 7-6.

Functions within a group are equivalent in precedence. Execution takes place from left to right. For example, in the operation (X \* Y / Z), X is multiplied by Y, and the result is divided by Z.

A subexpression enclosed in parentheses is evaluated before surrounding operators are applied, e.g., in (X+Y) \* Z, the sum of X+Y is multiplied by Z.

- Parentheses, constants, and subscripted variables are allowed in the expressions.
- You can use only integers in an IMATH statement. Mixed mode operation (integer and real numbers) is not supported.
- Denote a binary number by the prefix 0B (e.g.0B10111), a hexadecimal number by the prefix 0H (e.g. 0H7FFF). Add the suffix L to denote a long (32-bit) signed integer; add the suffix U to denote a 16-bit unsigned integer. (“L” and “U” integers are supported by PowerMath CPUs only.)
- The programming device checks to see if a statement is valid as you enter the statement and reports an error by placing the cursor in the field where the error occurs.

Table 7-6 Order of Precedence for IMATH Operators

Highest Precedence	Intrinsic function ABS <sup>1</sup> , NOT, Negation	NOT -
	Multiplication, Division, MOD	* / MOD
	Addition, Subtraction	+ -
	Shift left, Shift right	<< >>
	Relational Operators (= < <= >= <>) <sup>1</sup>	
	&, Logical AND <sup>1</sup>	
	, ^, Logical OR <sup>1</sup>	
Lowest Precedence	Assignment (:=)	:=
<sup>1</sup> Supported by PowerMath CPUs only.		

```
IMATH      V100(V5 + 2 * V7):= NOT(WX7 &(V99 ^ WX5))
```

Figure 7-22 IMATH Statement Example

## 7.20 Lead/Lag Operation

### LEAD/LAG Description

The LEAD/LAG statement (Figure 7-23) allows filtering to be done on an analog variable. This procedure calculates an output based on an input and the specified gain, lead, and lag values. The LEAD/LAG statement can only be used with cyclic processes, such as loops, analog alarms, and cyclic SF programs.

LEAD/LAG	Input . . . . . : A	Output . . . . . : B
	Lead time (Min) . . : C	Lag time (Min) . . . : D
	Gain (%) . . . . . : E	Old input . . . . . : F
	A = Address	Integer/real
	B = Address	Integer/real, writeable
	C = Address or value	Real
	D = Address or value	Real
	E = Address or value	Real
	F = Address	Integer/real, writeable

Figure 7-23 LEAD/LAG Format

- A specifies the location of the input value of the current sample period that is to be processed.
- B specifies the location of the output variable, the result of the LEAD/LAG operation.
- C specifies the lead time in minutes.
- D specifies the lag time in minutes.
- E (Gain) specifies the ratio of the change in output to the change in input at a steady state, as shown in the following equation. The constant must be greater than zero.

$$\text{Gain} = \frac{\Delta \text{ output}}{\Delta \text{ input}}$$

- F specifies the memory location of the input value from the previous sample period.
- For sample time, LEAD/LAG algorithm uses the sample time of the loop, analog alarm, or cyclic SF program from which it is called
- The first time it executes, LEAD/LAG is initialized and output equals input.



**LEAD/LAG  
Operation**

The LEAD/LAG algorithm uses the following equation.

$$Y_n = \left( \frac{T_{Lag}}{T_{Lag} + T_s} \right) Y_{n-1} + \text{Gain} \left( \frac{T_{Lead} + T_s}{T_{Lag} + T_s} \right) X_n - \text{Gain} \left( \frac{T_{Lead}}{T_{Lag} + T_s} \right) X_{n-1}$$

where  $Y_n$  = present output,  $Y_{n-1}$  = previous output,

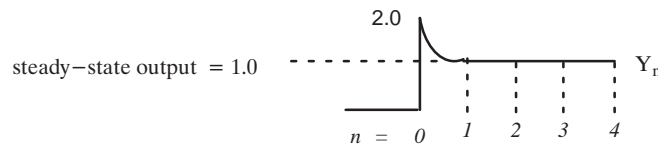
$X_n$  = present input,  $X_{n-1}$  = previous input

$T_s$  = sample time in minutes.

The output depends on the ratio of lead to lag as explained below. Assume the following values in each example:  $\Delta$  input and gain = 1.0

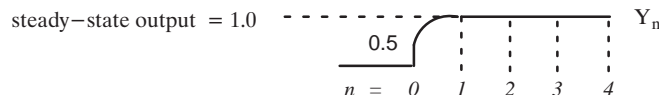
If  $T_{Lead} / T_{Lag}$  is greater than 1.0, then the initial response overshoots the steady-state output value.

$$\text{Initial output} = \Delta \text{input} * \text{Gain} \left( \frac{T_{Lead}}{T_{Lag}} \right) = 1.0 * 1.0 \left( \frac{2.0}{1.0} \right) = 2.0$$



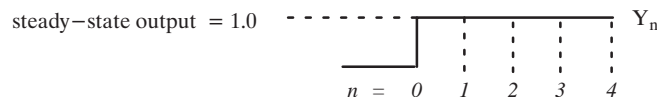
If  $T_{Lead} / T_{Lag}$  is less than 1.0, then the initial response undershoots the steady-state output value.

$$\text{Initial output} = \Delta \text{input} * \text{Gain} \left( \frac{T_{Lead}}{T_{Lag}} \right) = 1.0 * 1.0 \left( \frac{1.0}{2.0} \right) = 0.5$$



If  $T_{Lead} / T_{Lag}$  is equal to 1.0, then the initial response instantaneously reaches the steady-state output value.

$$\text{Initial output} = \Delta \text{input} * \text{Gain} \left( \frac{T_{Lead}}{T_{Lag}} \right) = 1.0 * 1.0 \left( \frac{1.0}{1.0} \right) = 1.0$$



## 7.21 Real/Integer Math Operations

### MATH Description

The MATH statement executes arithmetic computations involving both integers and real numbers. The MATH format, based on the operators in Table 7-7, is shown in Figure 7-24.

- Parentheses, constants, subscripted variables, and a set of intrinsic functions (listed in Table 7-8) are allowed in the expressions.
- Assignment operator ( := ) is required.

Table 7-7 MATH Operators

Operator	Description
**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction/Unary Minus (negation)
:=	Assignment
>>	Shift right (arithmetic). The sign bit is shifted into the vacated bits.
<<	Shift left (arithmetic). Zeros are shifted into the vacated bits.
=	Equal. The expression $X = Y$ returns 1 if X equals Y, and zero if not.
<>	Not equal. The expression $X <> Y$ returns 1 if X is not equal to Y, and zero if so.
<	Less Than. The expression $X < Y$ returns 1 if X is less than Y, and zero otherwise.
<=	Less Than or Equal. The expression $X <= Y$ returns 1 if X is less than or equal to Y, and zero otherwise.
>	Greater Than. The expression $X > Y$ returns 1 if X is greater than Y, and zero otherwise.
>=	Greater Than or Equal. The expression $X >= Y$ returns 1 if X is greater than or equal to Y, and zero otherwise.
MOD	Modulo arithmetic. The expression $X \text{ mod } Y$ returns the remainder of X after division by Y.
NOT	The expression NOT X returns 1 if X is equal to zero, and zero otherwise.
AND	Logical AND. The expression $X \text{ AND } Y$ returns 1 if both X and Y are non-zero, and zero otherwise.
OR	Logical OR. The expression $X \text{ OR } Y$ returns 1 if either X or Y is non-zero, and zero otherwise.
&	Bit-by-bit AND of two integers or real numbers.
	Bit-by-bit OR of two integers or real numbers.
^	Bit-by-bit exclusive OR of two integers or real numbers.

<p>MATH      A := B</p> <p>A = Address      Integer/real, writeable</p> <p>B = Address or value      Integer/real</p>
---

Figure 7-24 MATH Format

Table 7-8 MATH Intrinsic Functions

Function	Description
ABS	Absolute value
ARCCOS	Inverse Cosine in radians
ARCSIN	Inverse Sine in radians
ARCTAN	Inverse Tangent in radians
CEIL	CEIL(X) returns the smallest integer that is greater than or equal to X
COS	Cosine in radians
EXP	Exponential
FLOOR	FLOOR(X) returns the largest integer that is less than or equal to X
FRAC	FRAC(X) returns the fractional portion of X
LN	Natural (base e) Logarithm
LOG	Common (base 10) Logarithm
SIN	Sine in radians
TAN	Tangent in radians
ROUND	ROUND(X) returns the integer closest to X
SQRT	Square Root
TRUNC	TRUNC(X) returns the integer portion of X

### MATH Operation

The operation of MATH is described below and illustrated in Figure 7-25.

- Each time the MATH statement is executed, the calculations within the statement are made.
- The MATH computations are executed using the rules of precedence for arithmetic operations listed in Table 7-9. Functions within a group are equivalent in precedence. Execution takes place from left to right for all operators except exponentiation. For example, in the operation  $(X * Y / Z)$ , X is multiplied by Y, and the result is divided by Z.

A subexpression enclosed in parentheses is evaluated before surrounding operators are applied, e.g., in  $(X+Y) * Z$ , the sum of X+Y is multiplied by Z.

- When you read a discrete point in an SF program expression, a zero is returned if the discrete bit is off; a one is returned if the discrete bit is on. When you write to a discrete point in an SF program expression, the discrete bit turns off if the value is zero; the discrete bit turns on if the value is non-zero.
- You can use both integers and real numbers in a MATH statement. The controller executes this mixed-mode operation by converting all integers to real on input and rounding the resulting real to integer if the destination is an integer. (Refer to Section 2.3 for the supported range and precision of real numbers.)

## Real/Integer Math Operations (continued)

- Real variables are designated by a period following the memory address or variable name (V300. or LPV35.).
- Denote a binary number by the prefix 0B (e.g.: 0B10111), a hexadecimal number by the prefix 0H (e.g.: 0H7FFF). Add the suffix L to denote a long (32-bit) signed integer; add the suffix U to denote a 16-bit unsigned integer. (“L” and “U” integers are supported by PowerMath CPUs only.)
- The programming software checks a statement as you enter it, and, if necessary, reports an error by placing the cursor in the field containing the error.

Table 7-9 Order of Precedence for MATH Operators

Highest Precedence	Intrinsic Functions, NOT, Negation	NOT -
	Exponentiation <sup>1</sup>	**
	Multiplication, Division, MOD	* / MOD
	Addition, Subtraction	+ -
	Shift left, Shift right	<< >>
	Relational Operators (= < <= >= <>)	
	&, Logical AND	
Lowest Precedence	, ^, Logical OR	
	Assignment (:=)	

<sup>1</sup> Execution of exponentiation takes place from right to left. For example, in the operation (X \*\* Y \*\* Z), Y is raised to the power of Z; and then X is raised to the power determined by the result.

MATH	V75.:= 0.929783 * V77. + 2* SQRT ((EXP(V300.) + LN(V302.))/(V304.**0.25))
MATH	V100:= V901/(V45. + V46.)

Figure 7-25 MATH Statement Example

### Using Word Indexing

You can use two kinds of subscripted variables. Denote word indexing by the expression Z(n). Use word indexing to access the nth word from variable Z. Examples of word indexing follow:

$$\begin{array}{lll}
 V100(1) \equiv V100 & V100.(1) \equiv V100. & V100L(1) \equiv V100L \\
 V100(2) \equiv V101 & V100.(2) \equiv V101. & V100L(2) \equiv V101L \\
 V100(3) \equiv V102 & V100.(3) \equiv V102. & V100L(3) \equiv V102L
 \end{array}$$

### Using Element Indexing

Denote element indexing by the expression  $Z(n)$ . Use element indexing to access the  $n$ th element of an array  $Z$ ; the actual variable accessed depends upon the type of array. Examples of element indexing follow:

$$\begin{aligned} V100(:1) &\equiv V100 & V100.(1) &\equiv V100. & V100L(:1) &\equiv V100L \\ V100(:2) &\equiv V101 & V100.(2) &\equiv V102. & V100L(:2) &\equiv V102L \\ V100(:3) &\equiv V102 & V100.(3) &\equiv V104. & V100L(:3) &\equiv V104L \end{aligned}$$

### Indexing Loop and Analog Alarm Variables

For the loop and analog alarm variables, the two kinds of indexing are equivalent, as shown below:

$$\begin{aligned} LPV1(1) &\equiv LPV1(:1) \equiv LPV1 & LPV1.(1) &\equiv LPV1.(1) \equiv LPV1. \\ LPV1(2) &\equiv LPV1(:2) \equiv LPV2 & LPV1.(2) &\equiv LPV1.(2) \equiv LPV2. \\ LPV1(3) &\equiv LPV1(:3) \equiv LPV3 & LPV1.(3) &\equiv LPV1.(3) \equiv LPV3. \end{aligned}$$

### Using Multiple Subscripts

Since TISOFT does not use multiple subscripts, these expressions are not allowed:  $Z(n)(m)$ ,  $Z(n)(:m)$ ,  $Z(n)(:m)$ . Re-code the first two expressions as:

$$\begin{aligned} Z(n)(m) &\equiv Z(n + m - 1) \\ Z(n)(:m) &\equiv Z(n + m - 1) \end{aligned}$$

Re-code the third expression as:

$$\begin{aligned} Z(n)(:m) &\equiv Z(n + m - 1) \\ &\text{when } Z \text{ is an integer, or a loop or analog alarm variable.} \\ Z(n)(:m) &\equiv Z(n + 2*m - 2) \\ &\text{when } Z \text{ is a real number but not a loop or analog alarm variable.} \end{aligned}$$

A subscript may itself be an expression [as  $V100.(V5+2*V7)$ ] and may include real terms. All calculations are done according to the rules of real arithmetic. For example,  $V100.(12/6) \equiv V100.(2.0) \equiv V101$ .

## Real/Integer Math Operations (continued)

---

### MATH Examples

The following examples use some of the MATH functions.

- If  $X = 5.5$ , then  $\text{CEIL}(X) = 6$ . If  $X = -5.9$ , then  $\text{CEIL}(X) = -5$ .
- If  $X = 5.9$ , then  $\text{FLOOR}(X) = 5$ . If  $X = -5.9$ , then  $\text{FLOOR}(X) = -6$ .
- The shift right/left functions operate as follows. Assume that V300 contains 0000 0000 0000 1000, that equals 8.

V200 := V300 >> 1 places the following value into V200:  
0000 0000 0000 0100, that equals 4.

V200 := 8 >> 1 places the following value into V200:  
0000 0000 0000 0100, that equals 4.

V200 := V300 << 1 places the following value into V200:  
0000 0000 0001 0000, that equals 16.

V200 := 8 << 1 places the following value into V200:  
0000 0000 0001 0000, that equals 16.

If V400 contains 0000 0000 0000 0011, that equals 3, then  
V200 := V300 << V400 places the following value into V200:  
0000 0000 0100 0000, that equals 64.

For the shift right function, the sign bit is shifted into the vacated bits.  
If V677 contains 1000 1000 0000 0000, then V677 >> 3 places the  
following value into V677: 1111 0001 0000 0000.

For the shift left function, zeros are shifted into the vacated bits. If V677  
contains 0000 0001 0000 0000, then V677 << 3 places the following  
value into V677: 0000 1000 0000 0000.

## 7.22 Pack Data

### PACK Description

The Pack Data statement moves discrete and/or word data to or from a table. You can access the image register directly by using the PACK statement. PACK is primarily intended for use in consolidating data so that it can be efficiently transmitted to a host computer. The PACK format is shown in Figure 7-26.

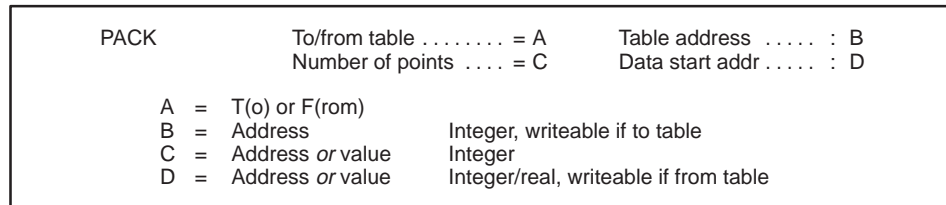


Figure 7-26 PACK Format

- A specifies whether you are writing data to or from the table.
- B specifies the address of the table, to or from which data are written or read.
- C is an integer number that specifies how many points or words are to be moved.
- For a TO table, D specifies the starting address of the points or words that are to be written to the table.  
For a FROM table, D specifies the starting address in memory into which data is to be read from the table.  
D + (C-1) must be within configured memory range.
- Fields C and D can be repeated for up to 20 writes/reads to and from the table (Figure 7-27).

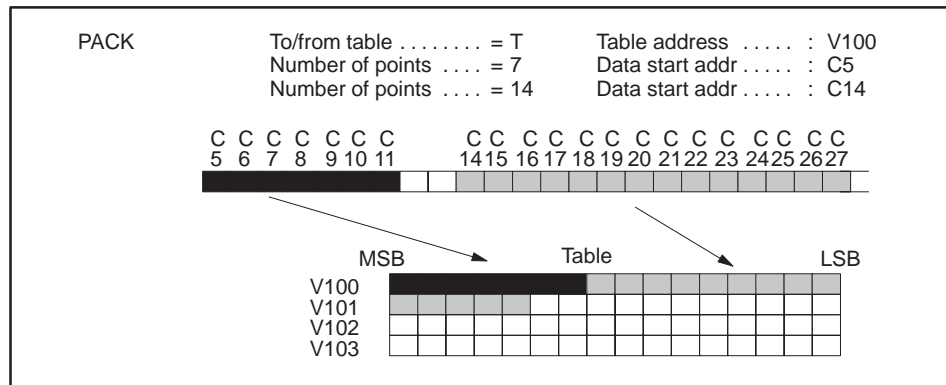


Figure 7-27 Example of PACKing Multiple Blocks of Bits Into Table

# PACK

## Pack Data (continued)

**PACK TO Operation**    The operation of the PACK TO statement is described below.

- For a TO Table, data are written into a table. This write operation begins with the data starting at the first Data Start Address and writes the specified number of points or words into the table, beginning with the first word of the table.

Bits are written sequentially as illustrated in Figure 7-28 below.

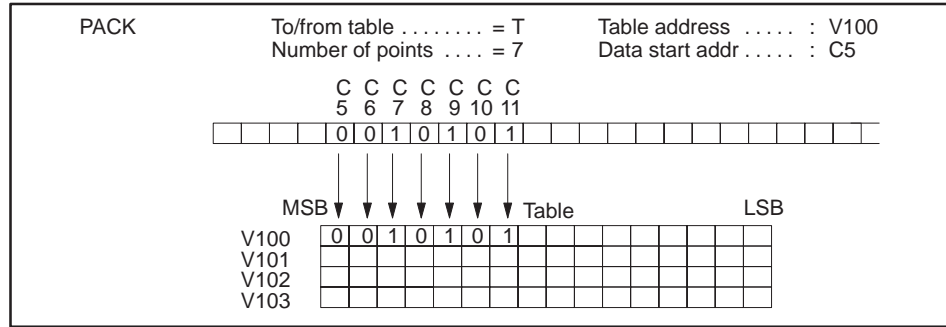


Figure 7-28 Example of PACKing Bits Into Table

You can specify multiple blocks of data to be written into the table. When the first word of the table is full, PACK begins to fill the second word.







Words are read sequentially from the table, as illustrated in Figure 7-33. You can also PACK multiple blocks of words.

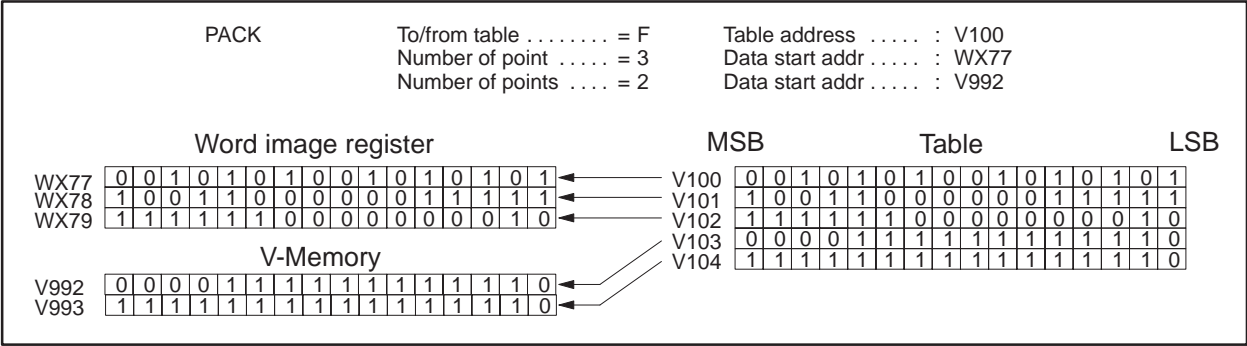


Figure 7-33 Example of PACKing Words from a Table

- You can PACK blocks of words and blocks of bits from a table with one PACK statement. See Figure 7-34. The data are packed according to these rules.

All discrete points designated in the Number of Points field are packed from the table.

Words are packed from the first available word in the table. That is, unused bits in the previous word of the table are not included as part of a word that is PACKed from the table.

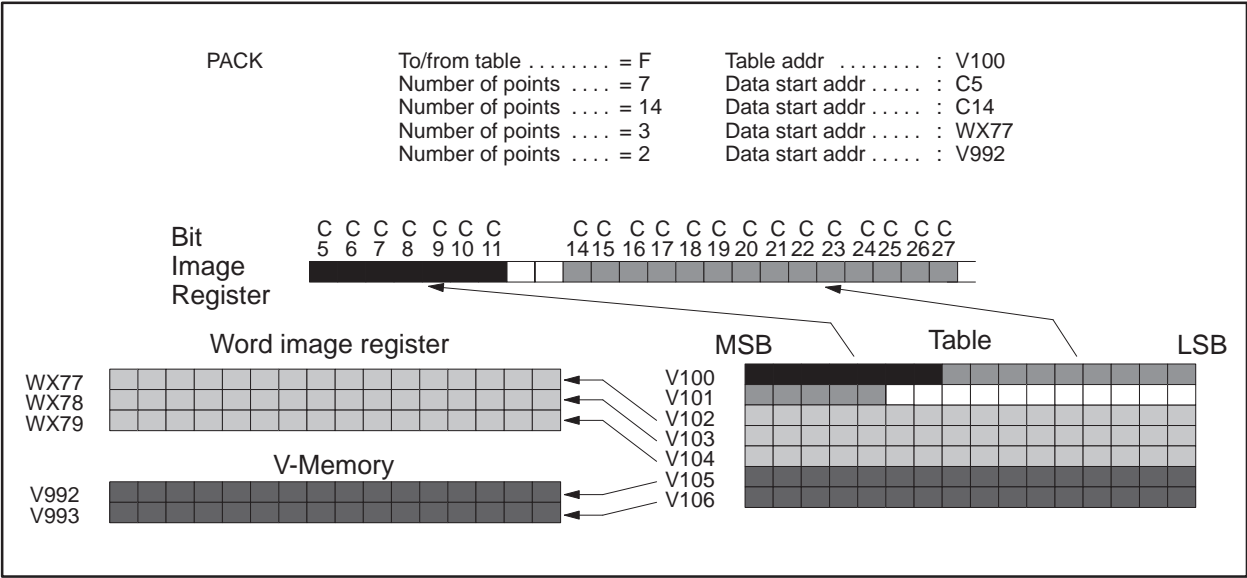


Figure 7-34 Example of PACKing Bits and Words from a Table

## 7.23 Pack Analog Alarm Data

### PACKAA Description

The Pack Analog Alarm Data statement moves analog alarm data to or from a table. PACKAA is primarily intended for use in consolidating analog alarm data to be accessed from an operator interface. The PACKAA format is shown in Figure 7-35.

PACKAA	To/from table . . . . . : A	Table address . . . . . : B
	Alarm number . . . . . : C	
	Parameters . . . . . : D	
A =	T(o) or F(rom)	
B =	Address	Integer, writeable if to table
C =	Address or value	Integer
D =	Element	Integer/real, writeable if from table, only analog alarm data types

Figure 7-35 PACKAA Format

- A specifies whether you are writing data to or from the table.
- B specifies the address of the table, to or from which data are moved.
- C specifies the number of the analog alarm to be accessed. C may range from 1 to the maximum number of alarms.
- D specifies the analog alarm variables. Up to eight variables can be designated. See Table 7-10 for a list of the analog alarm variables.

Table 7-10 Analog Alarm Variables

Mnemonic	Variable Name	Mnemonic	Variable Name
AACK	Acknowledge	APV*	Process Variable
AADB*	Deadband	APVH.	Process Variable High Limit
ACF	C-Flags (32 bits)	APVL.	Process Variable Low Limit
ACFH	Most Significant Word of C-Flags	ARCA.	Rate of Change Alarm Limit
ACFL	Least Significant Word of C-Flags	ASP*	Set Point
AERR*	Error	ASPH*	Set Point High Limit
AHA*	High Alarm Limit	ASPL*	Set Point Low Limit
AHHA*	High-High Alarm Limit	ATS.	Sample Rate
ALA*	Low Alarm Limit	AVF	V Flags
ALLA*	Low-Low Alarm Limit	AYDA*	Yellow Deviation Alarm Limit
AODA*	Orange Deviation Alarm Limit		
* Variables with an asterisk can be either a real number or an integer. Variables followed by a period are real numbers. Variables not followed by a period are integers. When you execute PACKAA using real numbers, two memory locations are allocated for each real number.			

**PACKAA Operation** The operation of the PACKAA statement is described below and illustrated in Figure 7-36 and Figure 7-37. When the PACKAA statement executes, the following actions occur.

- For a TO Table, the value of the analog alarm variable specified in D is written into the table at the address designated by B.

If additional variables are specified, the second variable is written to (B + 1), the third to (B + 2), and so on up to eight variables.

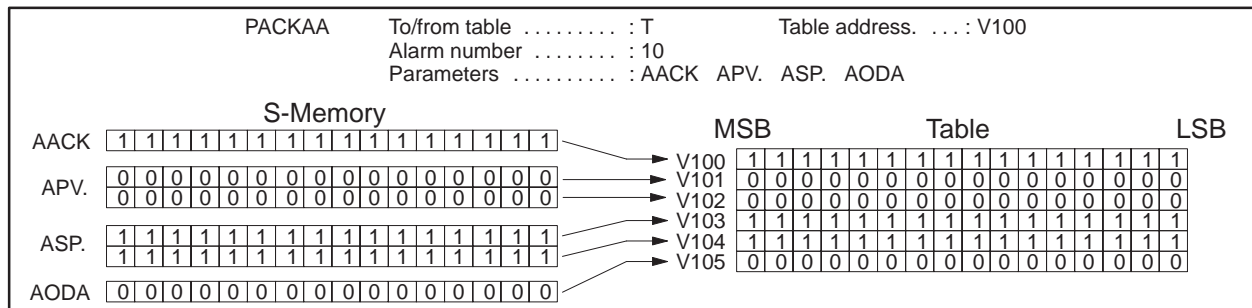


Figure 7-36 Example of PACKAA TO Table Operation

- For a FROM Table, PACKAA writes the word in the table starting address B into the specified analog alarm variable.

If additional variables are specified, the second word in the table is written to the second variable, and so on up to eight variables.

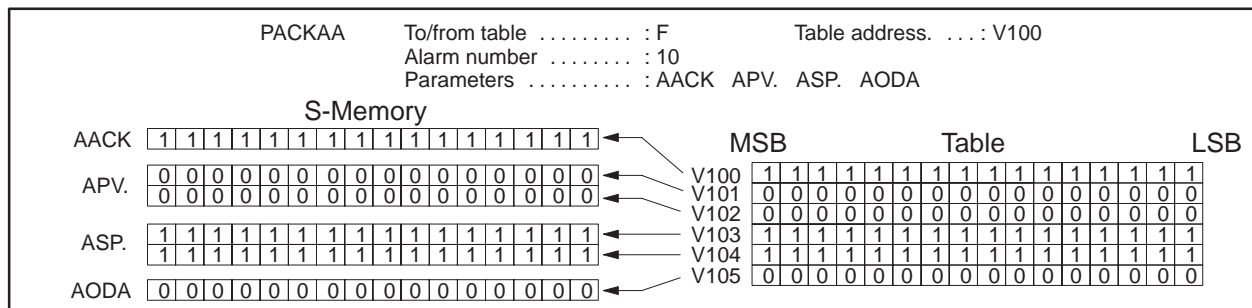


Figure 7-37 Example of PACKAA FROM Table Operation

## 7.24 Pack Loop Data

### PACKLOOP Description

The PACKLOOP statement (Figure 7-38) moves loop data to or from a table. PACKLOOP is primarily intended for use in consolidating loop data to be accessed from an operator interface.

PACKLOOP	To/from table . . . . . : A	Table address . . . . : B
	Loop number . . . . . : C	
	Parameters . . . . . : D	
A =	T(o) or F(rom)	
B =	Address	Integer, writeable if to table
C =	Address or value	Integer
D =	Element	Integer/real, writeable if from table, only loop data types

Figure 7-38 PACKLOOP Format

- A specifies whether you are writing data to or from the table.
- B specifies the address of the table, to or from which data are moved.
- C specifies the number of the loop to be accessed. The range for C is from 1 to the maximum number of loops.
- D specifies the loop variables. Up to eight variables can be designated. See Table 7-11 for a list of the loop variables.

### PACKLOOP Operation

The operation of the PACKLOOP statement is described below. PACKLOOP operates similarly to the PACKAA statement. See Figure 7-36 and Figure 7-37 for an example of how the PACKLOOP statement executes.

When the PACKLOOP statement executes the following actions occur.

- For a TO Table, the value of the loop variable specified in D is written into the table at the address designated by B.  
  
If additional variables are specified, the second variable is written to (B + 1), the third to (B + 2), and so on up to eight variables.
- For a FROM Table, PACKLOOP writes the word in the table starting address B into the specified loop variable.  
  
If additional variables are specified, the second word in the table is written to the second variable, and so on up to eight variables.

Table 7-11 Loop Variables

Mnemonic	Variable Name
LACK	Alarm Acknowledge
LADB*	Alarm Deadband
LCF	C-Flags (32 bits)
LCFH	Most Significant Word of C-Flags
LCFL	Least Significant Word of C-Flags
LERR*	Error
LHA*	High Alarm Limit
LHHA*	High-high Alarm Limit
LKC.	Gain
LKD.	Derivative Gain Limiting Coefficient
LLA*	Low Alarm Limit
LLLA*	Low-low Alarm Limit
LMN*	Output
LMX*	Bias
LODA*	Orange Deviation Alarm Limit
LPV*	Process Variable
LPVH.	Process Variable High Limit
LPVL.	Process Variable Low Limit
LRCA.	Rate of Change Alarm Limit
LRSF	Ramp/Soak Flags
LRSN	Ramp/Soak Step Number
LSP*	Set Point
LSPH*	Set Point High Limit
LSPL*	Set Point Low Limit
LTD.	Rate
LTI.	Reset
LTS.	Sample Rate
LVF	V-Flags
LYDA*	Yellow Deviation Limit
<p>* Variables with an asterisk can be either a real number or an integer. Variables followed by a period are real numbers. Variables not followed by a period are integers. When you execute PACKLOOP using real numbers, two memory locations are allocated for each real number.</p>	

## 7.25 Pack Ramp/Soak Data

**PACKRS Description** The Pack Ramp/Soak Data statement moves one or more elements (steps) of the ramp/soak profile for a given loop to or from a table. PACKRS is primarily intended to make the ramp/soak profiles accessible to an operator interface and to provide a method for dynamic ramp/soak profiling. The PACKRS format is shown in Figure 7-39.

<b>PACKRS</b>	TO/FROM TABLE . . . : A	TABLE ADDRESS . . . : B
	LOOP NUMBER . . . . : C	
	NO. OF STEPS . . . . : D	STARTING STEP . . . : E
<p>A = T(o) or F(rom)          B = Address . . . . . Integer, writeable if to table          C = Address or value . . . . Integer          D = Address or value . . . . Integer          E = Address or value . . . . Integer</p>		

Figure 7-39 PACKRS Format

- A specifies whether you are writing data to or from the table.
- B specifies the address of the table, to or from which data are moved.
- C specifies the loop number whose ramp/soak profile is involved in the pack operation.
- D specifies the number of ramp/soak steps to pack.
- E specifies the starting step in the ramp/soak profile at which the pack operation will begin.

**PACKRS Operation** The number of steps in a ramp/soak profile is established when it is programmed using TISOFT or your programming package. The PACKRS instruction cannot expand or shorten the ramp/soak profile for a given loop. This instruction can only read or modify existing steps in a preexisting profile.

PACKRS instructions that specify operations on non-existent profile steps are invalid, and the execution of this instruction terminates.



---

If TO Table is specified, this instruction copies the specified number of steps from the ramp/soak profile of a given loop, starting at the specified step number, to a table in memory whose starting address is indicated in the instruction.

If FROM Table is specified, this instruction copies the specified number of profile steps from a memory table into the ramp/soak profile for the indicated loop starting at the specified step number. The new step values overwrite the affected step values in the profile.

---

**NOTE:** Care should be taken when using the PACKRS instruction with a FROM Table specified. If the ramp/soak profile being modified is in progress when the PACKRS instruction executes, then your process could react erratically due to the sudden replacement of values in the profile steps. You can use one of the following methods to ensure that the profile update is done when the current profile is not in progress.

- In your program, check the state of the profile finished bit (bit 4) in LRSF for the corresponding loop. Do not execute the PACKRS statement unless the finished bit is set.
  - In your program, place the loop in the manual mode, execute the PACKRS to update the ramp/soak profile, then return the loop to automatic mode. (Remember, this causes the ramp/soak profile to be restarted at the initial step.)
-

Pack Ramp/Soak Data (continued)

When stored in a memory table, ramp/soak profile steps are six words long and have the following format:

- Word 1 (bit 1): Step type — 0 = ramp step, 1 = soak step (bit)
- Word 1 (bits 2–16) + Word 2: Address of status bit (special address format)
- Words 3/4: Setpoint, if ramp step, or Soak time, if soak step (REAL number)
- Words 5/6: Ramp rate, if ramp step, or Deadband, if soak step (REAL number)

The status bit address points to either an output point (Y) or a control relay (C). This address takes a short form for point numbers C1 – C512 and Y1 – Y1024. Higher point numbers use a long form of address. If all bits of the status bit address field are 0, then no status bit is selected for the step.

The short address form is shown in Figure 7-40.

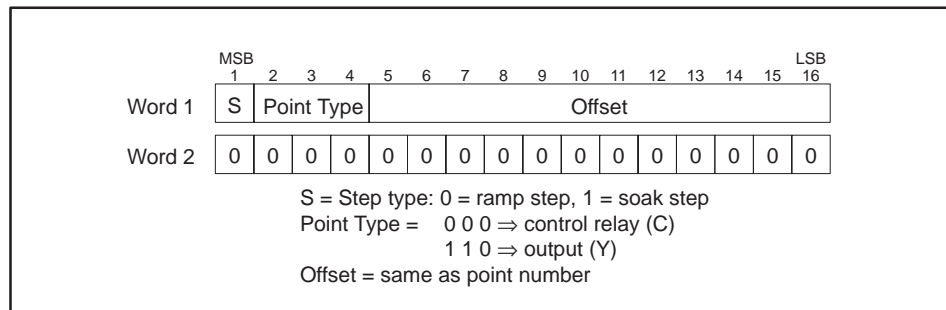


Figure 7-40 Address Format — Short Form

For example, the encoded address for Y23 using the short form is shown in Figure 7-41.

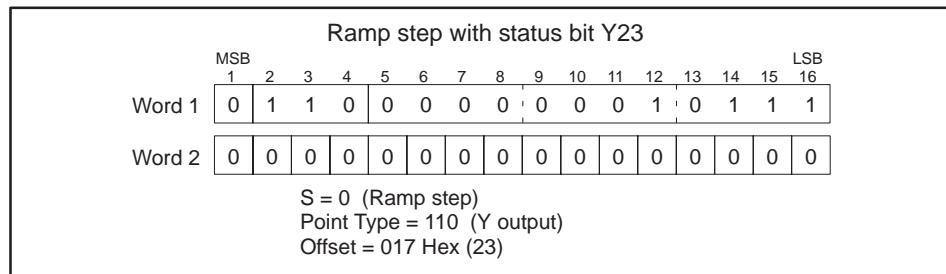


Figure 7-41 Short Form Address Example

The long address form is shown in Figure 7-42.

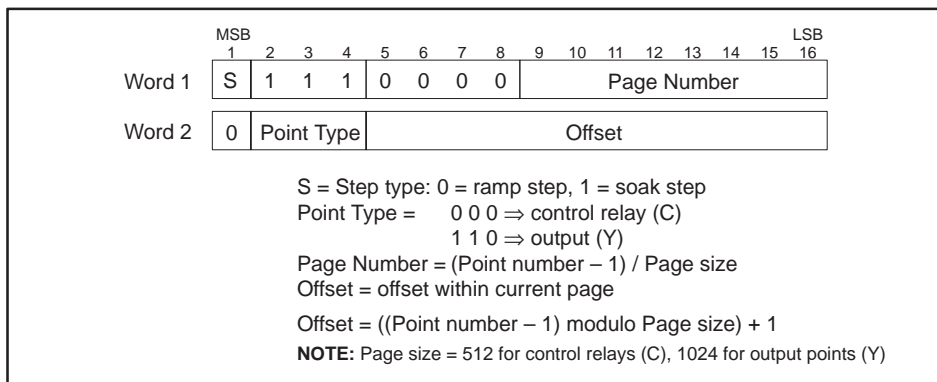


Figure 7-42 Address Format — Long Form

For example, the encoded address for C514 using the long form is shown in Figure 7-43.

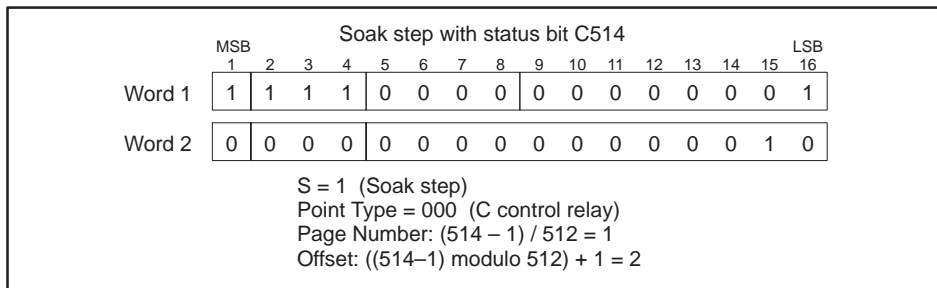


Figure 7-43 Long Form Address Example

Pack Ramp/Soak Data (continued)

Figure 7-44 shows an example of the PACKRS instruction moving values from a ramp/soak profile to a V-memory table.

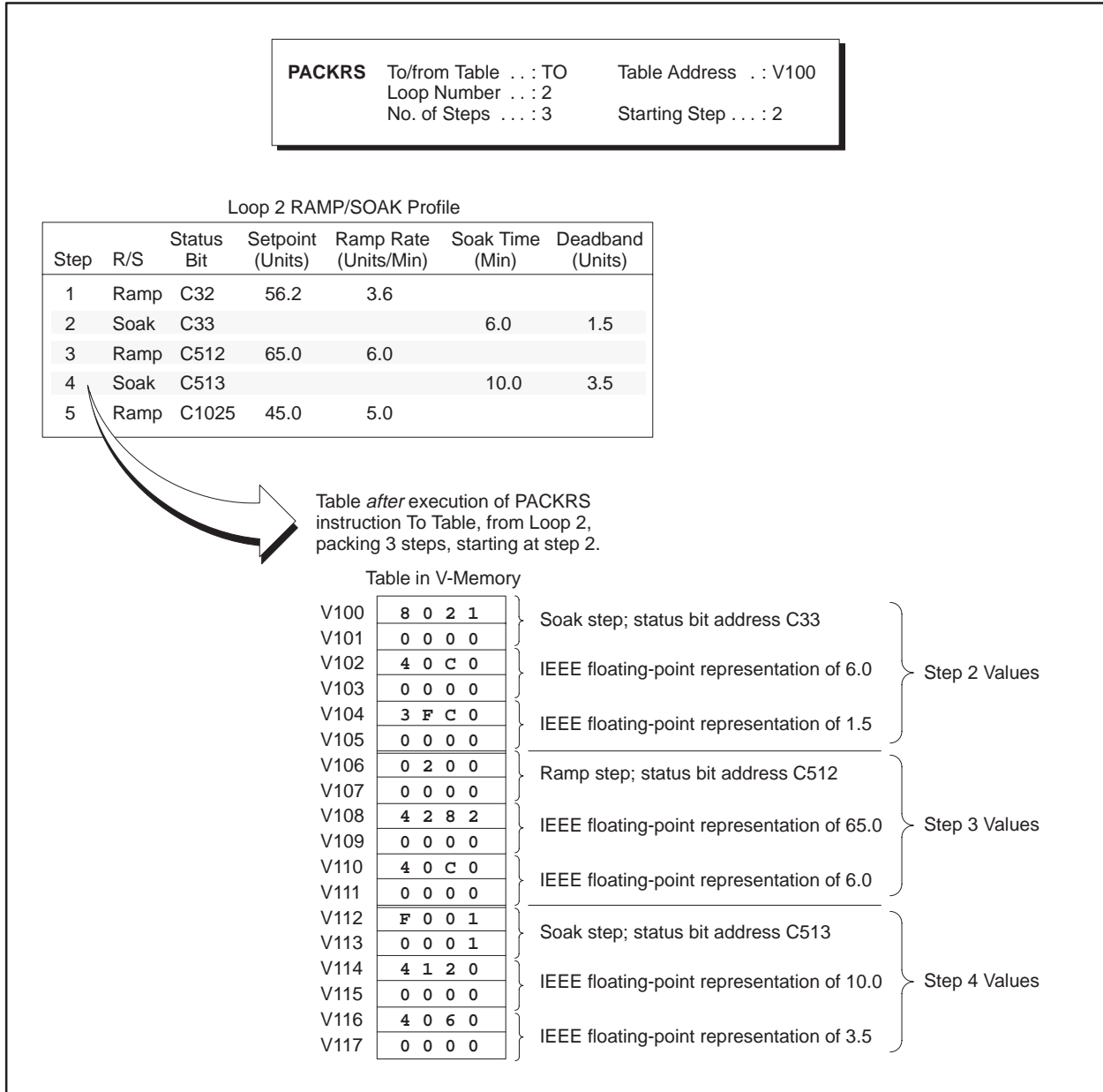


Figure 7-44 Example of PACKRS to a Table in V-Memory

Figure 7-45 shows an example of the PACKRS instruction moving values from a V-memory table to a Loop Ramp/Soak profile, changing two of the values in the profile, and leaving the remaining values unchanged.

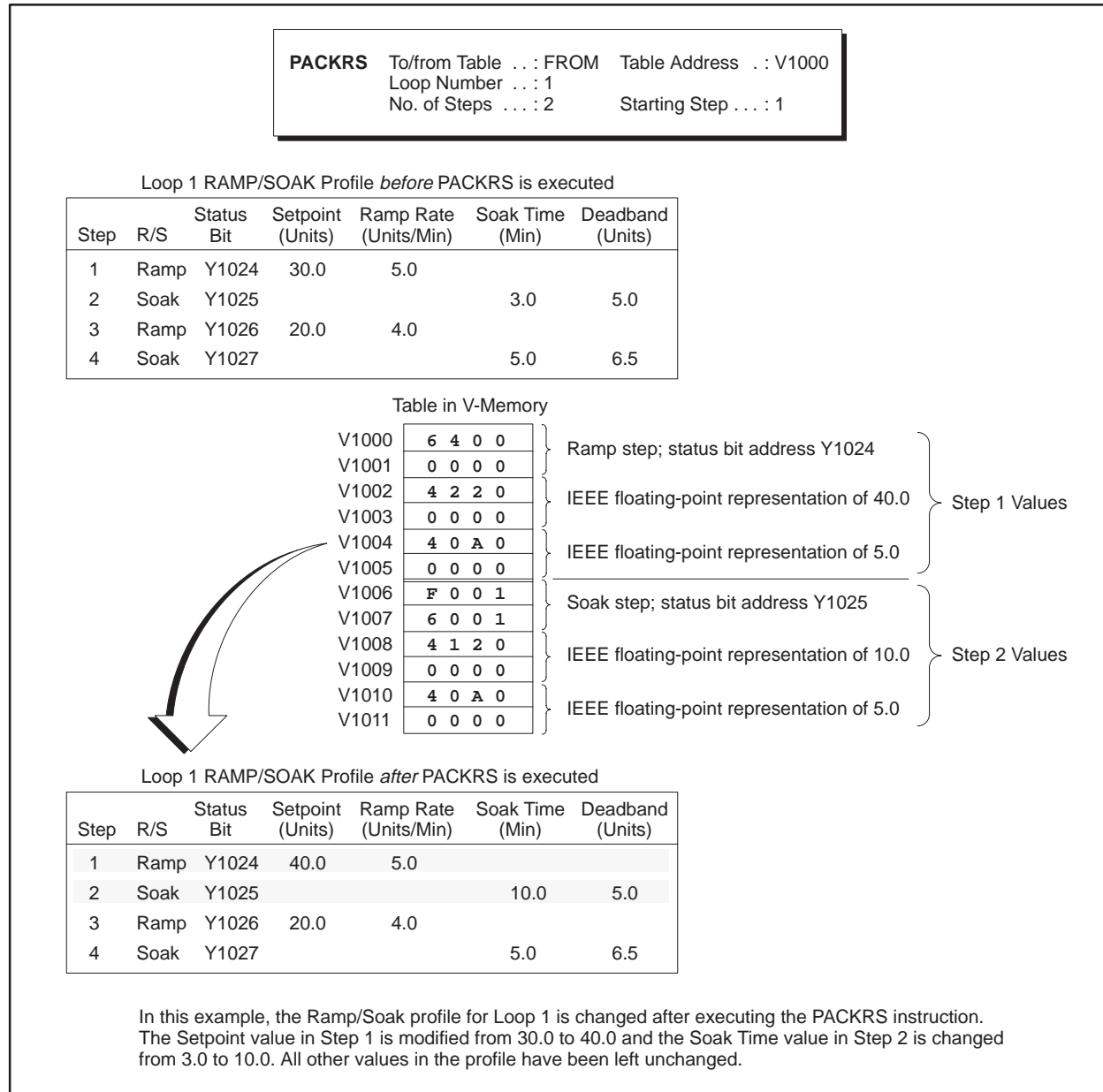


Figure 7-45 Example of PACKRS from a Table in V-Memory

## 7.26 Pet Scan Watchdog

---

**PETWD Description**      PETWD (Pet Scan Watchdog) allows you to extend the scan watchdog limit while performing an in-line SF program or SF subroutine from an RLL program. When the PETWD instruction executes, the scan watchdog timer is reset at that instance of time during the scan, therefore extending the scan watchdog limit beyond the configured scan watchdog limit.

A possible use would be in a large table lookup operation performed while the controlled process is at a steady-state condition.

The RETURN format has no subfields.

The PETWD instruction is intended to be used in the normal RLL task (TASK 1).

PETWD is available only for in-line compiled SF programs or subroutines in CPUs that support PowerMath.

### **WARNING**

The PETWD instruction allows you to place the PETWD instruction in an infinite loop, therefore preventing the scan watchdog limit from ever being reached.

If the PETWD instruction is in an infinite loop, the PLC would not issue a scan watchdog FATAL ERROR to shut the process down, therefore leaving your process uncontrolled. An uncontrolled process could result in death or serious injury to personnel, and/or damage to equipment.

Ensure that the PETWD instruction is not located in an infinite loop. To ensure that the PETWD instruction is not located in an infinite loop within an SF program or subroutine, place the PETWD instruction without a label at the beginning of the SF program or subroutine.



## 7.27 Printing

---

### PRINT Description

The Print statement sends a message to the ASCII communication ports. This statement can be used to print both text and the contents of integer and real variables. The PRINT format is shown in Figure 7-46.

PRINT	Port . . . . : A	Message . . . . :
	B:	
A = 1		
B = Address		Text enclosed in double quotes

Figure 7-46 PRINT Format

- A is the port number. You must enter a 1 in this field.
- B contains a free format message. The message begins on the line following the port and message designator fields. Element addresses and Expressions are separated by a space. No embedded space or the assignment operator (:=) in an expression is accepted.

### PRINT Operation

The operation of the PRINT statement is described below.

- When the PRINT statement executes, the message is sent to the port(s) specified.
- The maximum message length is 1019 characters, with characters counted in entries as follows:  
 Each text character = 1 character  
 Each variable entry = 6 characters  
 Each variable text entry = 6 characters  
 Carriage Return & Linefeed = 2 characters
- Text Entries contain ASCII text to be printed. Text entries are enclosed in quotation marks.

Example: PRINT PORT=1 MESSAGE:  
"END OF SHIFT REPORT"

- Variable Entries print the contents of variables in either integer or real format. Variables must be separated by spaces. Real numbers are indicated by following the address with a period (.). Integers are printed right-justified in a six character field with a floating minus sign. Real numbers are printed right-justified in a twelve character field using a FORTRAN G12.5 format.

Example: PRINT PORT=1 MESSAGE:  
"THE VALUES ARE" WX5 V104.



- 
- Time Entries are used to print out a variable in time format. The variable is printed out as hh:mm:ss. Time entries are indicated by following the address of the variable (EL or EXP) with :TIME.

Example: PRINT PORT=1 MESSAGE:  
"THE TIME IS NOW" STW141:TIME

- Date Entries are used to print out a variable in date format. The variable is printed out as yy/mm/dd. Date entries are indicated by following the address of the variable (EL or EXP) with :DATE.

Example: PRINT PORT=1 MESSAGE:  
"THE DATE IS NOW" STW141:DATE

- Variable Text Entries are used to print out text stored in either V or K memory. Variable Text Entries are indicated by following the address of the text (EL or EXP) to be printed with a percent sign (%) and the number of characters to be printed. If the number is coded as zero, PRINT assumes that the first word of the indicated variable contains the number of characters to print.

Example: PRINT PORT=1 MESSAGE:  
"BOILER" V250%16  
"DESCRIPTION" V102%0

"Boiler" V250%16 causes the 16 characters in V-Memory locations V250–V257 to be printed. Each word contains two 8-bit characters.

"Description" V102%0 causes the number of characters specified in V102 to be printed. If V102 contains 5, then the characters in V103–V105 are printed.

Variable Text Entries are a valuable tool for embedding control characters to be used by the device receiving the ASCII characters. The next page gives instructions about how to embed a control character in variable text.

## Printing (continued)

---

The form-feed indicator <FF> is entered as: "<FF>".

Follow these steps.

1. Enter the double quote character “
2. Enter the less than character <
3. Enter the F character F
4. Enter the F character F
5. Enter the greater than character >
6. Enter the double quote character ”

Example: PRINT      PORT=1 MESSAGE:  
                          “THERE IS A FORMFEED  
                          AFTER THIS <FF>”

To enter a <CR><LF> (Carriage return/Linefeed), follow these steps.

1. Enter the double quote character “
2. Press the carriage return key  or
3. Enter the double quote character ”

Example: PRINT      PORT=1 MESSAGE:  
                          “THERE IS A CARRIAGE RETURN  
                          LINEFEED AFTER THIS  
                          ”

To print the double quotes “”, precede it with another double quote “ as shown in the example below.

Example: PRINT      PORT=1 MESSAGE:  
                          “ “THIS QUOTED TEXT IS PRINTED INSIDE  
                          DOUBLE QUOTE CHARACTERS” ”

## 7.28 Return from SF Program/Subroutine

---

The Return statement is used to terminate an SF program or an SF subroutine. If invoked from an SF program, the program terminates. If invoked from an SF subroutine, control returns to the statement in the SF program following the SF subroutine call. The RETURN format has no subfields. If there is no RETURN statement, the program terminates after the last statement. The format of the RETURN statement is shown in Figure 7-47.

```
<SF Statement>  
<SF Statement>  
<SF Statement>  
RETURN
```

Figure 7-47 Example of the RETURN Statement

## 7.29 Scaling Values

---

**SCALE Description** The Scale statement uses as input an integer input and converts it to engineering units scaled between high and low limits. The SCALE format is shown in Figure 7-48.

SCALE	Binary input . . . . : A	Scaled result . . . : B
	Low limit . . . . . : C	High limit . . . . . : D
	20% offset . . . . . : E	Bipolar . . . . . : F
A =	Address	Integer
B =	Address	Integer/real, writeable
C =	Literal constant	Real (C ≤ D)
D =	Literal constant	Real (C ≤ D)
E =	Y(es) or N(o)	
F =	Y(es) or N(o)	

Figure 7-48 SCALE Format

- A is the memory location of the input.
- B is the memory location of the result after conversion.
- C is the lower limit to which the input can be scaled.
- D is the upper limit to which the input can be scaled.
- E indicates if the input is 20% offset (Yes) or 0% offset (No).
- F indicates if the input is bipolar (Yes) or not (No).

---

**NOTE:** You cannot choose both bipolar and 20% offset for an input (Fields E–F).

---

**SCALE Operation**

The operation of the Scale statement is described below and illustrated in Figure 7-49.

- Each time the SCALE statement executes, an integer located in A converts to an integer or real number in engineering units, scaled between high and low limits.

If the input is a variable that could range from -32000 to +32000, the variable is bipolar. Set option F to Y(es). If the input is a variable that could range from 0 to 32000, the variable is unipolar. Set option F to N(o).

If the input is a variable that has a 20% offset (ranges from 6400 to 32000), set option E to Y(es). If the input is a variable that has a 0% offset, set option E to N(o).

- The result is stored in the address specified by B.

The low and high limits specified in C and D determine the range of the converted number. Values of C and D may fall within the following limits.

$$\begin{aligned} \text{Range} = & \quad 5.42101070 * 10^{-20} \quad \text{to} \quad 9.22337177 * 10^{18} \\ & \quad - 9.22337177 * 10^{18} \quad \text{to} \quad - 2.71050535 * 10^{-20} \end{aligned}$$

- An error is logged if the input value is outside the low-limit to high-limit range; and the output is clamped to the nearer of either the low limit or the high limit.

You can use the SCALE statement to convert an input signal from an analog input module to a value in engineering units. For example, consider these conditions.

- The input is a 4–20 mA signal that is converted by the analog input module to a value between 6400 and 32000 (unipolar, 20% offset) and sent to WX33.
- You want the result of the SCALE statement to be a real number ranging between 0 and 100 and be placed in V100., as shown below.

The SCALE fields would contain these values.

SCALE	Binary input . . . . . : WX33	Scaled result . . . . . : V100.
	Low limit . . . . . : 0	High limit . . . . . : 100
	20% offset . . . . . : Y	Bipolar . . . . . : N

**Figure 7-49 SCALE Example**

## 7.30 Sequential Data Table

### SDT Description

The Sequential Data Table statement moves words one at a time from an existing table to a destination address. A pointer designates the address of the next word in the table to be moved. Each time the statement is executed, one word moves and replaces the word at the destination address. The SDT format is shown in Figure 7-50.

SDT	Input table . . . . . : A	Output . . . . . : B
	Table PTR . . . . . : C	Table length . . . . . : D
	Restart bit . . . . . : E	
	A = Address	Integer/real
	B = Address	Integer/real, writeable
	C = Address	Integer, writeable
	D = Address <i>or</i> value	Integer
	E = Element	Bit, writeable

Figure 7-50 SDT Format

- A is the starting address for the table. When used in a compiled SF program or subroutine, you must specify a static table; that is, the table's base address must be a V, K, G, VMS, or VMM address.
- B is the output address to which the words are moved. When used in a compiled SF program or subroutine, you must specify a static table; that is, the table's base address must be a V, K, G, VMS, or VMM address.
- C is the address of the pointer.
- D is the length of the table and must be a value greater than zero. When used in a compiled SF program or subroutine, the table length must be specified as a value.
- E is the address of the restart (status) bit and can be a C or Y.

### SDT Operation

The operation of the SDT statement is described below and illustrated in Figure 7-51.

- The SDT moves words from a pre-existing table.

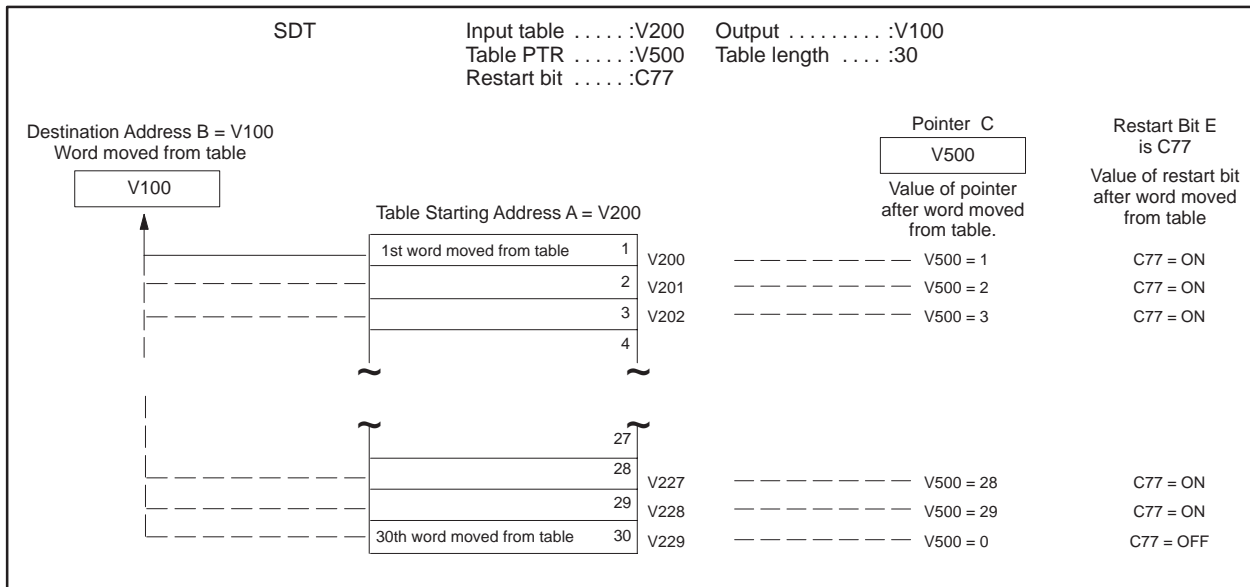
The size of the table depends upon the memory location that you choose and, if the memory is user-configurable, how much memory you allocated.

- Before the SDT is executed, pointer C contains zero. You must design your program to set the pointer to zero.

- Each time the SDT is executed, the following actions occur:  
 The table pointer is incremented by 1. Then the word in the table location specified by the pointer is moved to the destination address specified by B.  
 The process is repeated until the number of words specified in D has been moved.
- When the last word has been moved, the pointer is reset to zero.
- The restart bit E is on, except for the following conditions:  
 When the SDT resets the pointer, the restart bit turns off.  
 Prior to the first execution of the SDT, the bit could be either off or on depending upon prior usage.

The value of the pointer does not change when the SDT is not executing. All values in the table remain the same, and destination address B contains the value of the last word moved from the table.

You can use other logic to reset the pointer to zero, but the restart bit does not turn off.



**Figure 7-51 SDT Statement Example**

Before the SDT executes, the pointer V500 contains 0 (zero). When the statement executes, the pointer increments by 1, and the value in V200 is moved to V100. This process repeats each time the statement executes. After the last word is moved, the pointer resets to 0.

## 7.31 Synchronous Shift Register

---

### SSR Description

The Synchronous Shift Register statement builds a table that functions as synchronous shift register. The SSR format is shown in Figure 7-52.

SSR	Register start . . . : A	Status bit . . : B
	Register length . . : C	
A =	Address	Integer/real, writeable
B =	Element	Bit, writeable
C =	Address <i>or</i> value	Integer

Figure 7-52 SSR Format

- A is the starting address for the shift register.
- B is the status bit (C or Y) and is turned on when the register is empty.
- C is the length of the shift register. The maximum number of elements stored in the register is C. If a constant value is entered, it must be greater than zero.

### SSR Operation

The operation of the SSR statement is described below and illustrated in Figure 7-53.

- The starting address A designates the memory area in which the shift register is located.
- The register length C determines the size of the shift register. Size depends upon the memory location that you choose and how much memory you allocated (if the memory is user-configurable). The maximum number of elements stored in the register is C.
- The first position of the register, Register Start A, is empty until an element moves into A from another source.
- Each time the SSR executes, the element currently in memory location A shifts to A + 1. The element in A + 1 shifts to A + 2. Elements move down the shift register to A + 3, A + 4, etc., and A resets to zero.
- After the register is full, shifting in a new word causes the loss of the last word in the register at location [A + (C - 1)].
- The register is considered empty when it contains all zeros. The status bit B turns on when the register is empty.

---

**NOTE:** If the register contains the value -0.0, the register is not recognized as empty, and the status bit does not turn off.

---



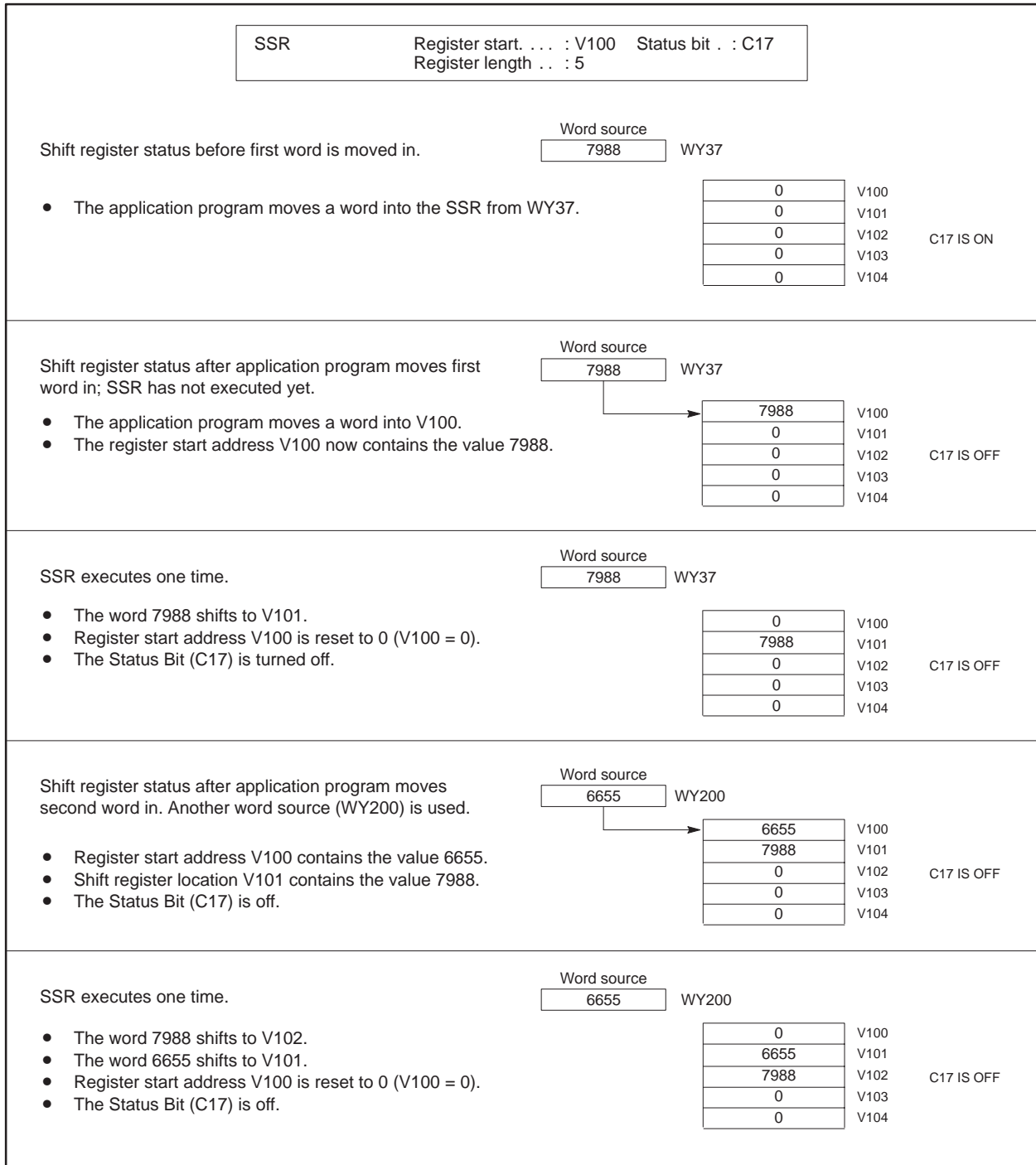


Figure 7-53 Example of SSR Operation

## 7.32 Unscaling Values

### UNSCALE Description

The Unscale statement takes as input a value in engineering units, scaled between high and low limits, and converts it to an integer. The UNSCALE format is shown in Figure 7-54.

UNSCALE	Scaled input . . . : A	Binary result . . . . : B
	Low limit . . . . . : C	High limit . . . . . : D
	20% offset . . . . . : E	Bipolar . . . . . : F
	A = Address	Integer/real
	B = Address	Integer, writeable
	C = Literal constant	Real (C ≤ D)
	D = Literal constant	Real (C ≤ D)
	E = Y(es) or N(o)	
	F = Y(es) or N(o)	

Figure 7-54 UNSCALE Format

- A is the memory location of the input.
- B is the memory location of the result after conversion.
- C is the lower limit of scaled input A.
- D is the upper limit of scaled input A.
- E indicates if the output is 20% offset (Yes) or 0% offset (No).
- F indicates if the output is bipolar (Yes) or not (No).

---

**NOTE:** You cannot choose both bipolar and 20% offset for an output (Fields E–F).

---

### UNSCALE Operation

The operation of the UNSCALE statement is described below and illustrated in Figure 7-55.

- Each time the UNSCALE statement executes, an integer or real number located in A is converted to a scaled integer.

The high and low limits of the value in A are specified in C and D. These limits can fall within the following range.

$$\begin{aligned} \text{Range} = & \quad 5.42101070 * 10^{-20} \quad \text{to} \quad 9.22337177 * 10^{18} \\ & \quad - 9.22337177 * 10^{18} \quad \text{to} \quad - 2.71050535 * 10^{-20} \end{aligned}$$

- The result is stored as an integer in the address specified by B.

If the output is a variable that has a 20% offset (ranges from 6400 to 32000), set option E to Y(es). If the output is a variable that has a 0% offset, set option E to N(o).

If the output is a variable that could range from -32000 to +32000, the variable is bipolar. Set option F to Y(es). If the output is a variable that could range from 0 to 32000, the variable is unipolar. Set option F to N(o).

- An error is logged if the scaled value of the input is outside the ranges given above, and the input is clamped to the nearer of either the low limit or the high limit.

You can use the UNSCALE statement to convert a value in engineering units to an output signal to an analog output module. For example, consider these conditions.

The value to be converted is at memory location V100. The value at V100 ranges between 0.0 and 100.0. You want the result of the UNSCALE statement to be an integer between 6400 and 32000 (unipolar, 20% offset) and to be sent to WY66.

The analog output module converts the UNSCALEed value at WY66 to a signal between 4 and 20 mA signal and sends the result to the field equipment.

The UNSCALE fields would contain these values.

UNSCALE	Scaled input ..... : V100	Binary output ..... : WY66
	Low limit ..... : 0.0	High limit ..... : 100.0
	20% offset ..... : Y	Bipolar ..... : N

Figure 7-55 UNSCALE Example



## 7.33 Comment

---

The Comment statement inserts a comment in a program for documentation purposes. The Comment statement is ignored during program execution. The COMMENT format is shown in Figure 7-56.

```
* This is an example of the free-form Comment statement.
```

Figure 7-56 Comment Format

- A comment statement can contain a maximum of 1021 characters.

# Programming Analog Alarms

---

<b>8.1</b>	<b>Overview</b> .....	<b>8-2</b>
<b>8.2</b>	<b>Analog Alarm Programming and Structure</b> .....	<b>8-4</b>
	Analog Alarm Numbers and Variable Names .....	8-4
	Programming Tables .....	8-4
	Analog Alarm C-Flags .....	8-5
<b>8.3</b>	<b>Specifying Analog Alarm V-Flag Address</b> .....	<b>8-6</b>
	Alarm V-Flag Address .....	8-6
<b>8.4</b>	<b>Specifying Analog Alarm Sample Rate</b> .....	<b>8-7</b>
	Sample Rate .....	8-7
<b>8.5</b>	<b>Specifying Analog Alarm Process Variable Parameters</b> .....	<b>8-8</b>
	Process Variable Address .....	8-8
	PV Range Low/High .....	8-8
	PV is Bipolar 20% Offset .....	8-8
	Square Root of PV .....	8-8
<b>8.6</b>	<b>Specifying Analog Alarm Deadband</b> .....	<b>8-9</b>
	Alarm Deadband .....	8-9
<b>8.7</b>	<b>Specifying Analog Alarm Process Variable Alarm Limits</b> .....	<b>8-10</b>
	PV Alarms: Low-low, Low, High, High-high .....	8-10
<b>8.8</b>	<b>Specifying Analog Alarm Setpoint Parameters</b> .....	<b>8-11</b>
	Remote Setpoint .....	8-11
	Clamp SP Limits .....	8-11
<b>8.9</b>	<b>Specifying Analog Alarm Special Function Call</b> .....	<b>8-12</b>
	Special Function .....	8-12
<b>8.10</b>	<b>Specifying Analog Alarm Setpoint Deviation Limits</b> .....	<b>8-13</b>
	Deviation Alarms: Yellow, Orange .....	8-13
<b>8.11</b>	<b>Specifying Other Analog Alarm Process Variable Alarms</b> .....	<b>8-14</b>
	Rate of Change Alarm .....	8-14
	Broken Transmitter Alarm .....	8-14

## 8.1 Overview

The analog alarm function allows you to monitor an analog input signal by setting standard alarms on a process variable (PV) and a target setpoint (SP). Eight alarms are available, as illustrated in Figure 8-1.

- High-high alarm point on the PV
- High alarm point on the PV
- Low alarm point on the PV
- Low-low alarm point on the PV
- Yellow deviation alarm point referenced to the SP
- Orange deviation alarm point referenced to the SP
- Rate of change alarm, for a PV changing too fast
- Broken transmitter, for a PV outside the designated valid range.

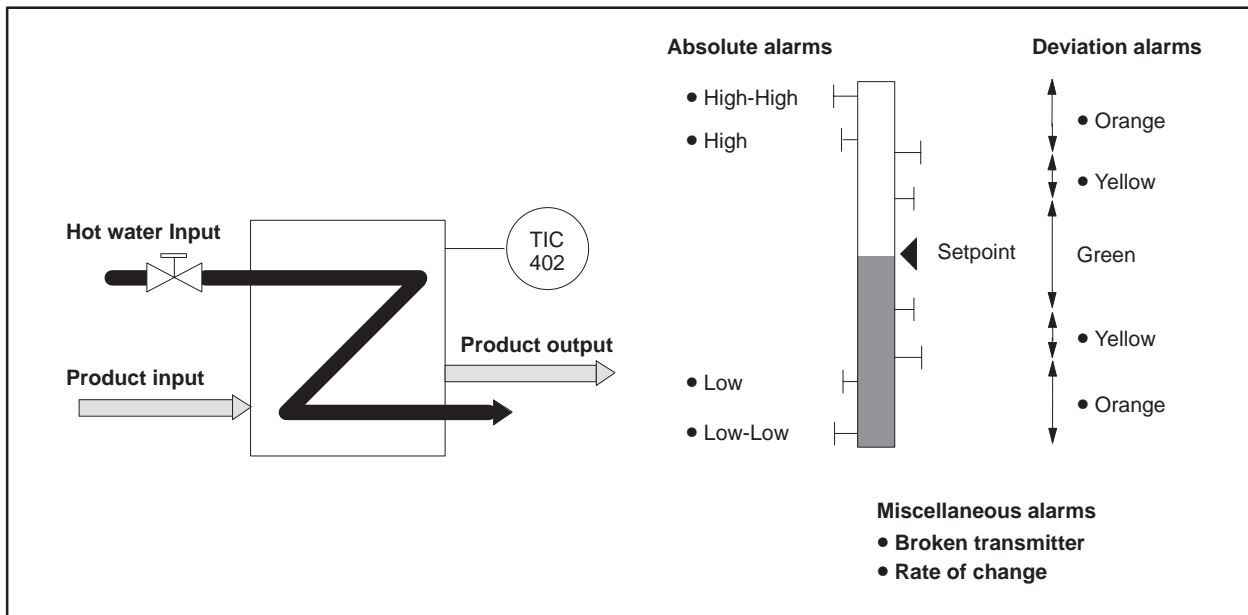


Figure 8-1 Example of Analog Alarm Application

---

The high-high, high, low, and low-low alarms are fixed absolute alarms and can correspond to warnings and shutdown limits for the process equipment itself. The yellow and orange deviation alarms move up and down with the target setpoint and can refer to specification tolerances around the target.

A PV alarm deadband is provided to minimize cycles in and out of alarm (chattering) that generate large numbers of messages when the PV hovers near one of the alarm limits.

An option is also available to call an SF program, discussed in Chapter 7, to initiate a special function calculation. This allows you to use the timing and scaling capabilities of the analog alarm algorithm in conjunction with SF program programming to provide a standard set of alarm checking capabilities on advanced custom-control algorithms written in SF programs.

## 8.2 Analog Alarm Programming and Structure

### Analog Alarm Numbers and Variable Names

Analog alarms are referenced by a user-assigned number from 1 to 128. The variables within each analog alarm are accessed by variable names assigned to each variable type. For example, the analog alarm setpoint is designated by ASP; to read the value of the setpoint for Analog Alarm 10, you would read ASP10. To read the value of the setpoint low limit for Analog Alarm 117, you would read ASPL117. Appendix A lists the analog alarm variable names.

### Programming Tables

When you program an analog alarm, you display the analog alarm programming table on your programming unit and enter the data in the appropriate fields. The general steps for entering analog alarm data follow. Refer to the TISOFT user manual for detailed instructions about programming analog alarms.

1. Select the ALARM option from the prompt line on your programming device.
2. Display the analog alarm that you want to program (#1, #2, etc.).
3. Enter the data for each field in the table.

The analog alarm programming table is shown in Figure 8-2. The page on which a field is described is also listed. All analog alarm parameters are stored in Special Memory (S-Memory) when you program the analog alarm. The size of S-Memory is user configurable. Refer to the TISOFT user manual for detailed instructions about configuring S-Memory.

ANALOG ALARM 128	TITLE: XXXXXXXX	MONITOR REMOTE SETPOINT:	
ALARM VFLAG ADDRESS:	Page 8-6	REMOTE SETPOINT:	Page 8-11
SAMPLE RATE (SECS):	Page 8-7	CLAMP SP LIMITS: LOW =	
		HIGH =	
PROCESS VARIABLE ADDRESS:	Page 8-8	SPECIAL FUNCTION:	Page 8-12
PV RANGE: LOW =	Page 8-8		
HIGH =			
PV IS BIPOLAR:	Page 8-8		
SQUARE ROOT OF PV:	Page 8-8		
20% OFFSET ON PV:	Page 8-8		
ALARM DEADBAND:	Page 8-9	MONITOR DEVIATION:	
MONITOR LOW-LOW/HI-HI:		DEVIATION ALARM: YELLOW =	Page 8-13
MONITOR LOW/HIGH:		ORANGE =	
PV ALARMS: LOW-LOW =	Page 8-10	MONITOR RATE OF CHANGE:	Page 8-14
LOW =		RATE OF CHANGE ALARM:	
HIGH =			
HIGH-HIGH =		MONITOR BROKEN TRANSMITTER:	Page 8-14

Figure 8-2 Analog Alarm Programming Table



**Analog Alarm  
C-Flags**

A set of flags (C-Flags) store the programming data that you enter into the Programming Tables for the analog alarms. The C-Flags correspond to individual bits making up the two words ACFH, that contains the most significant 16 bits, and ACFL, that contains the least significant 16 bits. Table 8-1 shows the designation for each bit in the C-Flag.

**Table 8-1 Analog Alarm C-Flags (ACFH and ACFL)**

<b>Variable</b>	<b>Word Bit</b>	<b>Flag Bit</b>	<b>Analog Alarm Function</b>
ACFH	1	1	0 = PV scale 0% offset 1 = PV scale 20% offset
	2	2	1 = Take square root of PV
	3	3	1 = Monitor HIGH/LOW alarms
	4	4	1 = Monitor HIGH-HIGH/LOW-LOW alarms
	5	5	1 = Monitor Deviation alarm
	6	6	1 = Monitor Rate-of-change alarm
	7	7	1 = Monitor Broken Transmitter Alarm
	8	8	0 = Local Setpoint 1 = Remote Setpoint
	9-16	9-16	Unused
ACFL	1-4	17-20	Unused
	5	21	0 = Process Variable is unipolar 1 = Process Variable is bipolar
	6	22	Unused
	7-16	23-32	Contains SF program number (if an SF program is scheduled to be called)

## 8.3 Specifying Analog Alarm V-Flag Address

---

### Alarm V-Flag Address

Enter an address: C, Y, V, or WY in the ALARM VFLAG ADDRESS field. If you select NONE, no data is written from the V-Flags in the analog alarm.

The V-Flags contain the operational data for an analog alarm. The V-Flags comprise the individual bits making up the 16-bit word AVF. The bits are defined in Table 8-2.

An entry in the ALARM VFLAG ADDRESS field causes analog alarm data to be written from the V-Flags to another address. The address can be either a bit (Y or C) that allocates 11 contiguous bits, or a word (WY or V) that allocates one word for V-Flag data.

Bits 1–2 are designated as control flags. If you create a V-Flag table in V-Memory, for example, the controller reads these two bits in the V-Memory address and writes over the corresponding bits in the AVF word. You can enable or disable the analog alarm by setting/clearing these control flags. You can read bits 3–12, but any changes that you make to them are overwritten by the controller.

Table 8-2 Analog Alarm V-Flags (AVF)

Bit	Analog Alarm Function
1	1 = Enable alarm
2	1 = Disable alarm
3	1 = PV is in high-high alarm
4	1 = PV is in high alarm
5	1 = PV is in low alarm
6	1 = PV is in low-low alarm
7	1 = PV is in yellow deviation alarm
8	1 = PV is in orange deviation alarm
9	1 = PV is in rate of change alarm
10	1 = Broken transmitter alarm
11	1 = Analog alarm is overrunning
12	1 = Alarm is enabled *
13–16	Unused
* If a word is selected for the analog alarm V-Flags, bit 12 is written. If a C or Y is selected, bit 12 is not used.	

---

**NOTE:** If you program an analog alarm and do not disable it, the controller begins to monitor the programmed variables as soon as you place the controller in RUN mode.

---

## 8.4 Specifying Analog Alarm Sample Rate

---

### Sample Rate

Enter a time in seconds in the SAMPLE RATE field.

The sample rate determines how often deviation alarm bits and associated math are evaluated. Sample rates are programmable in 0.1 second increments, with alarms checked at least once every two seconds. The sample rate can be any floating point number between 0.1 and  $1.6772 \times 10^6$  seconds.

## 8.5 Specifying Analog Alarm Process Variable Parameters

---

Process Variable Address	<p>Enter an address: V, WX or WY in the PROCESS VARIABLE ADDRESS field. (The 575 CPUs also allow G, VMM, or VMS memory address areas.)</p> <p>A process variable must be specified for each analog alarm. The process variable can be taken from the following.</p> <ul style="list-style-type: none"><li>• A word input or output module – The programming table uses a WX or WY address.</li><li>• A location in V-Memory – The programming table uses an address in V-Memory.</li></ul> <p>If you select NONE, the analog alarm does not read an address to obtain the process variable. In this case, you can use an SF program, to, for example, calculate a process variable. The result can be written to APV for processing by the analog alarm.</p>
PV Range Low/High	<p>Enter the low and high values of the process variable in the following fields: PV RANGE LOW and PV RANGE HIGH.</p> <p>You must specify the engineering values that correspond to the upper and lower ranges of the input span. If the span is 0 to 100%, the lower range is the engineering value corresponding to 0 volts. If the span is 20% to 100%, then the lower range is the engineering value corresponding to 1 volt. If the span is bipolar, the lower range is the engineering value corresponding to –5 or –10 volts.</p>
PV is Bipolar 20% Offset	<p>Select YES or NO to specify analog inputs as no offset, 20% offset, or bipolar in the following fields: PV IS BIPOLAR, and 20% OFFSET ON PV.</p> <p>The span of the analog inputs can be 0 to 5.0 volts, 0 to 10 volts, –10 to 10 volts, or –5 to 5 volts. The analog alarm processing feature provides a linear conversion over any of these process variable input spans. When you program the analog alarm, specify whether the process variable is to be no offset, 20% offset, or bipolar.</p> <p>A span of 0 to 5.0 volts (0 to 20.0 milliamps) is referred to as a span of 0 to 100%. A span of 1.0 to 5.0 volts (4.0 to 20.0 milliamps) is referred to as a span of 20% to 100% (20% offset on the process variable). Use bipolar with a span of –10 to 10 volts or –5 to 5 volts.</p>
Square Root of PV	<p>Select YES or NO for the square root option in the SQUARE ROOT OF PV field.</p> <p>Select YES if the input for the process variable is from a device (such as an orifice meter) that requires a square root calculation to determine the correct value to use.</p>

## 8.6 Specifying Analog Alarm Deadband

### Alarm Deadband

Enter a value in engineering units for the alarm deadband in the ALARM DEADBAND field.

When you specify an alarm deadband, the controller can provide hysteresis on all alarms except the rate of change alarm to prevent them from chattering when the process variable is near one of the alarm limits. The analog alarm does not exit the alarm condition until the process variable has come inside the alarm limit minus the deadband. This is shown graphically in Figure 8-3.

The range for the deadband (AADB) is  $0.0 \leq \text{AADB} \leq (\text{APVH} - \text{APVL})$ , where APVH and APVL are the process variable high and low limits, respectively. Typically, the deadband ranges from 0.2% to 5% of the span.

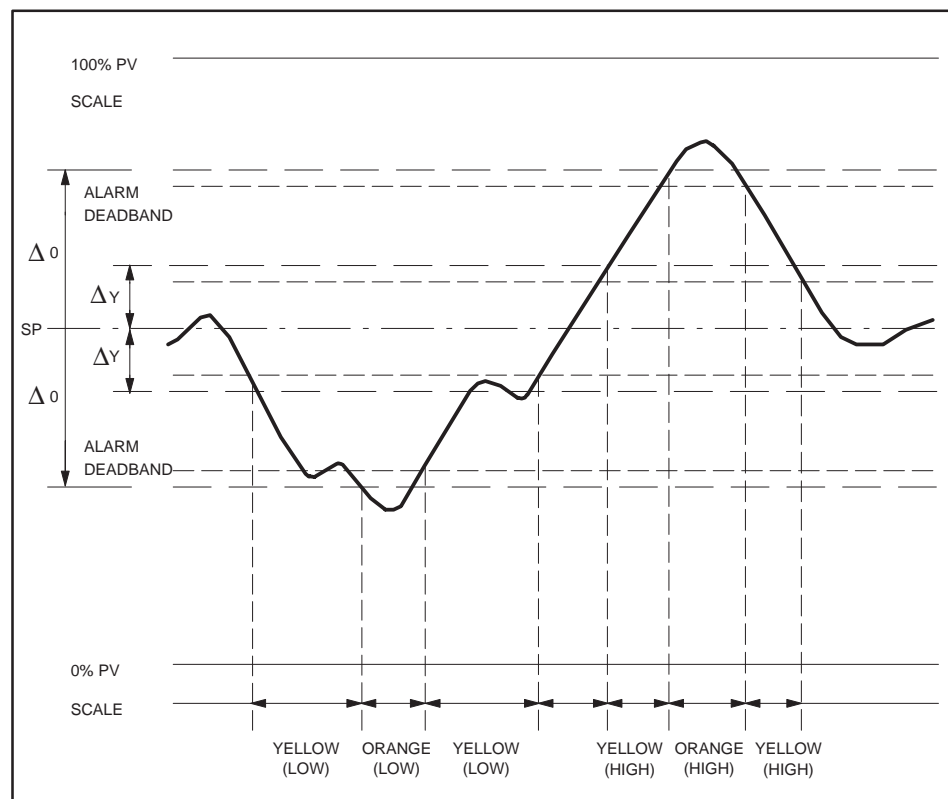


Figure 8-3 Example of Alarm Deadband For Analog Alarms

## 8.7 Specifying Analog Alarm Process Variable Alarm Limits

---

PV Alarms:  
Low-low, Low,  
High, High-high

Enter values in engineering units for the process variable alarm limits in the following fields: LOW, LOW-LOW, HIGH, And HIGH-HIGH. To have the controller monitor the alarm limits, select YES in the following fields: MONITOR LOW-LOW/HIGH-HIGH and MONITOR LOW/HIGH. Otherwise, select NO.

The high-high and low-low alarms can be entered as values requiring critical actions, while the high and low can be set at values requiring remedial measures. The range of possible values that can be used is given below.

- Low-low alarm — real number in engineering units; must be less than or equal to low alarm value and greater than or equal to low range of PV.
- Low alarm — real number in engineering units; must be less than or equal to high alarm value of PV.
- High alarm — real number in engineering units; must be less than or equal to high high alarm value of PV.
- High-high alarm — real number in engineering units; must be greater than or equal to high alarm value and less than or equal to high range of PV.

## 8.8 Specifying Analog Alarm Setpoint Parameters

---

- Remote Setpoint** Enter an address: V, K, WX, or WY, (or also G, VMM, or VMS in a 575), or a value, in the REMOTE SETPOINT field. Select NONE if there is no remote setpoint. To have the controller monitor the remote setpoint, select YES in the MONITOR REMOTE SETPOINT field. If you select NO, the analog alarm uses the current value in the analog alarm variable ASP.
- If you want to use a value external to the analog alarm for the setpoint, you specify the address for this value in the REMOTE SETPOINT field. For example, you can use data from a field transmitter for the setpoint by using a WX address for the transmitter input. Then specify this WX address in the REMOTE SETPOINT field.
- Clamp SP Limits** Enter values for the setpoint limits in the CLAMP SP LIMITS field. Select NONE if there are no limits, and zeroes are placed in the high and low fields.

## 8.9 Specifying Analog Alarm Special Function Call

---

**Special Function** Enter an SF program number in the SPECIAL FUNCTION field. Select NONE if no SF program is to be called for execution.

You can program the analog alarm to call an SF program to do a calculation on any constant, variable, or I/O point. This calculation occurs each time that the analog alarm processing is done, as required by the sample rate. The order of events follows.

When the analog alarm is processed, the process variable and the setpoint are read.

Before the analog alarm makes any comparisons between the process variable and the setpoint, the SF program is called for execution.

The SF program executes and writes results to the appropriate memory locations.

After the SF program terminates, the analog alarm continues processing.

---

**NOTE:** With PowerMath CPUs, an SF program can be compiled or interpreted. If the SF program is compiled, it executes to completion when the analog alarm calls it. If the SF program is interpreted, it is initiated when the alarm schedules it. While an interpreted SF program is executing, a higher priority process on its queue may interrupt it on any SF statement boundary.

---



## 8.10 Specifying Analog Alarm Setpoint Deviation Limits

---

**Deviation Alarms:** Enter values in engineering units for the setpoint deviation limits in the following fields: **YELLOW** and **ORANGE**. To have the controller monitor the deviation alarm limits, select **YES** in the **MONITOR DEVIATION** field. Otherwise, select **NO**.

**Yellow, Orange**

The deviation alarm bands are always centered around the target or setpoint; i.e., the deviation alarm test is actually on the control error.

There are two levels of deviation alarms.

- **Yellow Deviation** — This value indicates the maximum allowable error (SP – PV) that sets the yellow deviation alarm. The yellow deviation limit must be within the span of the process variable, and it must be less than or equal to the orange deviation alarm.
- **Orange Deviation** — This value indicates the maximum allowable error (SP – PV) that sets the orange deviation alarm. The orange deviation limit must be within the span of the process variable, and it must be greater than or equal to the yellow deviation alarm.

## 8.11 Specifying Other Analog Alarm Process Variable Alarms

---

**Rate of Change Alarm** Enter a value in engineering units for the rate of change alarm in the RATE OF CHANGE ALARM field. To have the controller monitor the rate of change, select YES in the MONITOR RATE OF CHANGE field. Otherwise, select NO.

If you program the controller to monitor the rate of change, an alarm occurs when the rate of change of the process variable exceeds the limit specified. This is a real number in engineering units/minute that is used to set the rate-of-change alarm flag.

**Broken Transmitter Alarm** To have the controller monitor for the broken transmitter condition, select YES in the MONITOR BROKEN TRANSMITTER field. Otherwise, select NO.

If you program the controller to monitor for the broken transmitter condition, an alarm occurs if the raw process variable is outside the valid range designated for the PV. The valid ranges follow.

- Bipolar: -32000 to 32000
- 0% Offset: 0 to 32000
- 20% Offset: 6400 to 32000

Figure 8-4 shows the process variable in broken transmitter alarm.

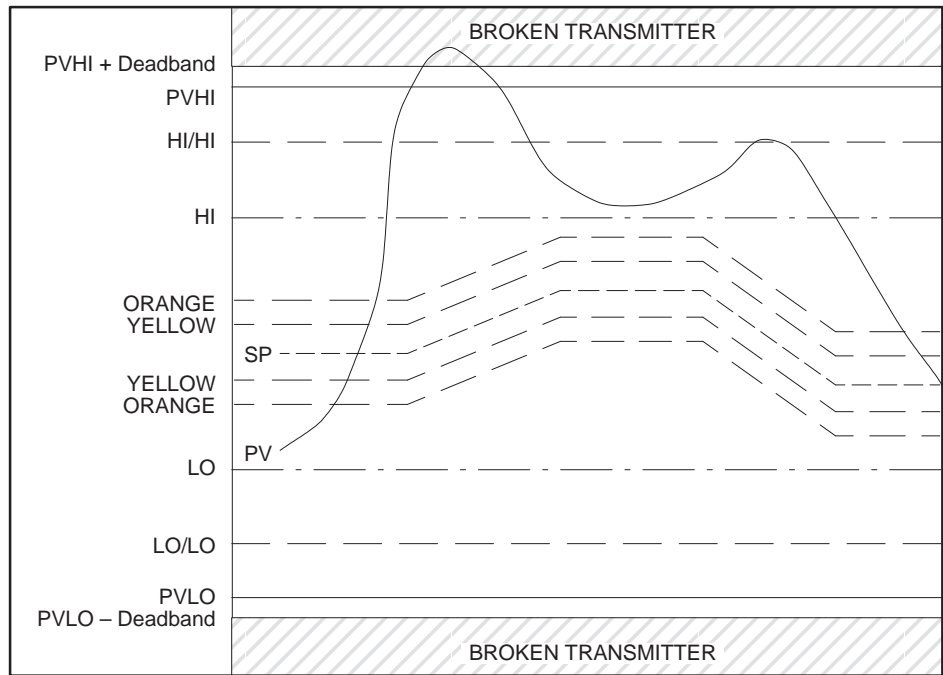


Figure 8-4 Example of Broken Transmitter Alarm

# Chapter 9

## Programming Loops

---

9.1	Overview .....	9-2
9.2	Using the PID Loop Function .....	9-4
9.3	Loop Algorithms .....	9-6
9.4	Programming Loops .....	9-8
9.5	Specifying Loop PID Algorithm .....	9-10
9.6	Specifying Loop V-Flag Address .....	9-11
9.7	Specifying Loop Sample Rate .....	9-12
9.8	Specifying Loop Process Variable Parameters .....	9-13
9.9	Specifying Loop Ramp/Soak Profile .....	9-14
9.10	Specifying Loop Output Parameters .....	9-18
9.11	Specifying Loop Alarm Deadband .....	9-19
9.12	Specifying Loop Process Variable Alarm Limits .....	9-20
9.13	Specifying Loop Setpoint Parameters .....	9-21
9.14	Specifying Loop Tuning Parameters .....	9-22
9.15	Specifying Loop Derivative Gain Limiting .....	9-25
9.16	Specifying Loop Special Function Call .....	9-26
9.17	Specifying Loop Locked Changes .....	9-28
9.18	Specifying Loop Error Operation .....	9-29
9.19	Specifying Reverse Acting Loops .....	9-30
9.20	Specifying Loop Setpoint Deviation Limits .....	9-31
9.21	Specifying Other Loop Process Variable Alarms .....	9-32
9.22	Using SmarTune Automatic Loop Tuning (555 CPUs Only) .....	9-34

## 9.1 Overview

Process and batch control capability is provided using the controller's proportional-integral-derivative (PID) loop functions, illustrated in Figure 9-1. When you program a loop, you can set the same eight alarm types used by analog alarms and described in Chapter 8.

- High-high alarm point on the process variable (PV)
- High alarm point on the PV
- Low alarm point on the PV
- Low-low alarm point on the PV
- Yellow deviation alarm point referenced to the setpoint (SP)
- Orange deviation alarm point referenced to the SP
- Rate of change alarm, for a PV changing too rapidly
- Broken transmitter, for a PV outside the designated valid range.

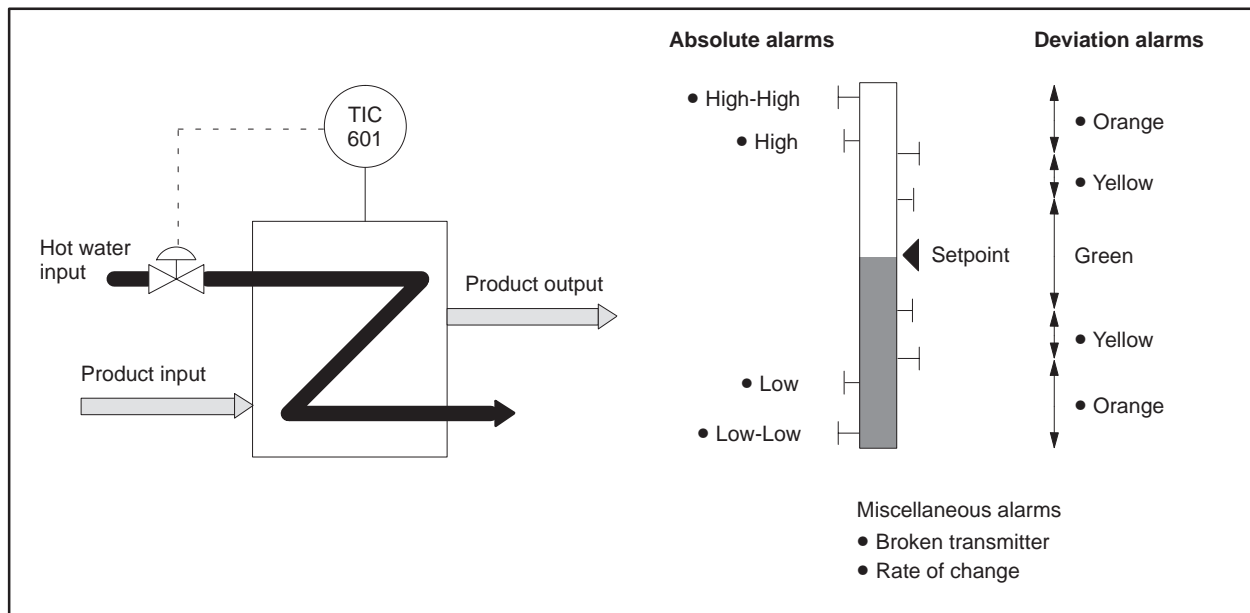


Figure 9-1 Example of Loop Control

---

The high-high, high, low, and low-low alarms are fixed absolute alarms and may correspond to warnings and shutdown limits for the process equipment itself. The yellow and orange deviation alarms move up and down with the setpoint and may refer to specification tolerances around the setpoint.

A PV alarm deadband is provided to minimize cycles in and out of alarm (chattering) that generate large numbers of messages when the PV hovers near one of the alarm limits.

An option is also available to call a Special Function Program (SF program, discussed in Chapter 7) to initiate a special function calculation. The SF program call can be scheduled on the PV, the SP, or the output.

## 9.2 Using the PID Loop Function

---

Loops operate in one of three states: Manual, Automatic, and Cascade. A fourth state — Loop Is Not Operating — is in effect when the controller is in Program mode.

Manual Mode	In Manual Mode, the loop output is not calculated by the controller but, instead, comes from the operator. While a loop is in Manual, the controller still monitors the Broken Transmitter, High-High, High, Low, Low-Low, and Rate-of-Change alarms. The Yellow and Orange deviation alarms are not monitored.
Auto Mode	In Auto Mode, the controller computes the loop output. The SP for the loop comes from either an operator interface, SF program, or from a Ramp/Soak Table. All alarms are monitored.
Cascade Mode	In Cascade Mode, the controller computes the loop output. The setpoint for the loop comes from a user-specified location called the remote setpoint. For truly cascaded loops, the remote setpoint is the output of another loop. The controller also allows the remote setpoint to be some other variable in the controller. Such loops are not truly cascaded, but the same term is used. All alarms are monitored.

For cascaded loops, the loop for which the output is used as the setpoint for another loop is called the outer loop. The loop that uses the output of another loop for its setpoint is called the inner loop. It is possible to cascade loops more than two levels deep.

If an inner loop of a cascade is placed in Auto or Manual, then all its outer loops must be placed in Manual to prevent reset windup. Similarly, an outer loop cannot be placed in Auto until all inner loops are in Cascade. The logic to handle opening and closing of cascades is built into the controller. Briefly, this is done as follows.

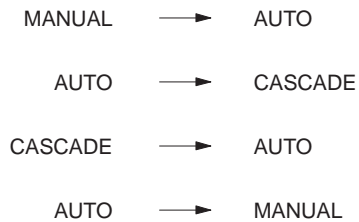
- If an inner loop is switched out of Cascade, then all of its outer loops are switched to Manual.
- A request to place an outer loop in Auto or Cascade is denied unless the inner loop is in Cascade.

If a loop is not truly cascaded, but is simply using a remote setpoint, changes to and from Cascade mode are allowed.

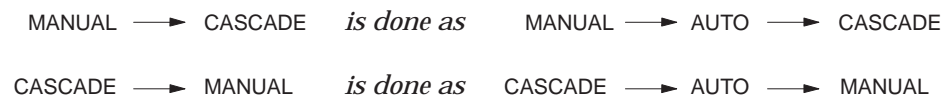
---

## Changing Loop Mode

The controller allows the loop mode to be changed by an SF program, ladder logic, or an operator interface device. While the loop can be requested to enter any mode from any other mode, the controller actually only performs the following mode transitions.



The other requests (Manual → Cascade and Cascade → Manual) are handled as transitions to Auto and then to the final mode as follows.





## 9.3 Loop Algorithms

The controller implements both the position and the velocity forms of the PID algorithm. For the position algorithm, the position of the device being controlled is computed based on the error. The velocity form of the PID algorithm computes the change in the device position based on the error.

### PID Position Algorithm

For the position form of the PID equation, the controller output  $M_n$  is computed as follows.

$$M_n = Kc \times e_n + Ki \sum_{j=1}^n e_j - Kr (PV_n - PV_{n-1}) + M_0$$

Variable	Definition	Loop Variable Mnemonic
Ts	Sample rate	LTS
Kc	Proportional gain	LKC
Ki	Coefficient of the integral term: $Kc \times (Ts / Ti)$	
Kr	Coefficient of the derivative term: $Kc \times (Td / Ts)$	
Ti	Reset or integral time	LTI
Td	Derivative time or rate	LTD
SP	Setpoint	LSP
PV <sub>n</sub>	Process Variable at nth sample	LPV
e <sub>n</sub>	Error at nth sample: $SP - PV_n$	
M <sub>0</sub>	Output at sample time 0	
M <sub>n</sub>	Output at sample time n	LMN

The controller combines the integral sum and the initial output into a single term called the bias (Mx). This results in the following equations that define bias and output at sample time.

$$Mx_n = Ki \sum_{j=1}^n e_j + M_0 \quad \text{Bias at sample time n}$$

$$M_n = Kc \times e_n - Kr (PV_n - PV_{n-1}) + Mx_n \quad \text{Output at sample time n}$$

The following is an example of the computation done by the controller during a single sample period for a loop. The rate portion of the algorithm is usually used for special cases and is set to 0 in this example.

Variable	Definition	Value
Ts	Sample rate	1 second
Kc	Proportional gain	.01
Ti	Reset or integral time	1 minute
Td	Derivative time or rate	0
SP	Setpoint	.5
PV <sub>n</sub>	Process Variable for this sample	.75
PV <sub>n-1</sub>	Process Variable for previous sample	.77
e <sub>n</sub>	Error for this sample: SP - PV	.5 - .75 = -.25
Mx <sub>n-1</sub>	Bias	.5
Ki	Coefficient of integral term: Kc × (Ts / Ti)	.01 × (1s / 60s) = .00017
Kr	Coefficient of derivative term: Kc × (Td / Ts)	.01 × (0s / 1s) = 0

$$\begin{aligned}
 \text{New Bias} &= M_{x_n} = K_i \times e_n + M_{x_{n-1}} \\
 &= M_{x_n} = (.00017 \times -.25) + .5 \\
 &= M_{x_n} = .4999 \\
 \text{New Output} &= M_n = K_c \times e_n - K_r \times (P V_n - P V_{n-1}) + M_{x_n} \\
 &= M_n = .01 \times -.25 - 0 \times (.75 - .77) + .4999 \\
 &= M_n = .49746
 \end{aligned}$$

The new bias is .4999 and the new output is 49.746%.

### PID Velocity Algorithm

The velocity form of the PID equation is obtained by subtracting the equation at time (n-1) from the equation at time (n).

$$\begin{aligned}
 \Delta M_n &= M_n - M_{n-1} \\
 &= K_c \left[ (e_n - e_{n-1}) + \frac{T_s}{T_i} \times e_n - \frac{T_d}{T_s} (P V_n - 2P V_{n-1} + P V_{n-2}) \right]
 \end{aligned}$$

Variable	Definition
M <sub>n</sub>	Loop output at the nth sample
Ti	Reset time
Kc	Proportional gain
Td	Rate Time
e <sub>n</sub>	Error (SP-PV) at the nth sample
Ts	Sample time
PV <sub>n</sub>	Process Variable at the nth sample

## 9.4 Programming Loops

### Loop Numbers and Variable Names

Loops are referenced by a user-assigned number. The variables within each loop are accessed by variable names assigned to each variable type. For example, the loop setpoint is designated by LSP; to read the value of the setpoint for Loop 10, you read LSP10. To read the value of the setpoint low limit for Loop 64, you read LSPL64. Appendix A lists the loop variable names.

### Programming Tables

When you program a loop, you display the loop programming table on your programming unit and enter the data in the appropriate fields. The general procedure for entering loop data are listed below. Refer to the TISOFT user manual for detailed instructions.

- Select the Loop option from the prompt line on your programming device.
- Display the loop that you want to program (# 1, # 2, etc.).
- Enter the data for each field in the table.

The loop programming table is shown in Figure 9-2. The page on which a field is described is also listed. All loop parameters are stored in Special Memory (S-Memory) when you program the loop. The size of S-Memory is user-configurable. Refer to the TISOFT user manual for detailed instructions about configuring S-Memory.

PID LOOP 12	TITLE: XXXXXXXX	REMOTE SETPOINT:	Page 9-21
POS/VEL PID ALGORITHM:	Page 9-10	CLAMP SP LIMITS: LOW =	Page 9-21
LOOP VFLAG ADDRESS:	Page 9-11	HIGH =	
SAMPLE RATE (SECS):	Page 9-12	LOOP GAIN:	
PROCESS VARIABLE ADDRESS:	Page 9-13	RESET (INTEGRAL TIME):	Page 9-22
PV RANGE: LOW =	Page 9-13	RATE (DERIVATIVE TIME):	
HIGH =		FREEZE BIAS:	Page 9-23
PV IS BIPOLAR:	Page 9-13	DERIVATIVE GAIN LIMITING:	Page 9-25
SQUARE ROOT OF PV:	Page 9-13	LIMITING COEFFICIENT:	
20% OFFSET ON PV:	Page 9-13	SPECIAL CALCULATION ON:	Page 9-26
LOOP OUTPUT ADDRESS:	Page 9-18	SPECIAL FUNCTION:	
OUTPUT IS BIPOLAR:	Page 9-18	LOCK SETPOINT:	Page 9-28
20% OFFSET ON OUTPUT:		LOCK AUTO/MANUAL:	
RAMP/SOAK PROGRAMMED:		LOCK CASCADE:	
RAMP/SOAK FOR SP:	Page 9-14	ERROR OPERATION:	Page 9-29
ALARM DEADBAND:	Page 9-19	REVERSE ACTING:	Page 9-30
MONITOR LOW-LOW/HI-HI:		MONITOR DEVIATION:	
MONITOR LOW/HIGH:		DEVIATION ALARM: YELLOW =	Page 9-31
PV ALARMS: LOW-LOW =	Page 9-20	ORANGE =	
LOW =		MONITOR RATE OF CHANGE:	Page 9-32
HIGH =		RATE OF CHANGE ALARM:	
HIGH-HIGH =		MONITOR BROKEN TRANSMITTER:	Page 9-32

Figure 9-2 Loop Programming Table

## Loop C-Flags

A set of flags (C-Flags) stores the programming data that you enter into the Programming Tables for the loops. The C-Flags correspond to individual bits making up the two words LCFH and LCFL. LCFH contains the most significant 16 bits, and LCFL contains the least significant 16 bits. Table 9-1 shows the designation for each bit in the C-Flag.

Table 9-1 Loop C-Flags (LCFH and LCFL)

Variable	Word Bit	Flag Bit	Loop Function
LCFH	1	1	0 = PV scale 0% offset 1 = PV scale 20% offset—only valid if PV is unipolar. See bit 21.
	2	2	1 = Take square root of PV
	3	3	1 = Monitor HIGH/LOW alarms
	4	4	1 = Monitor HIGH-HIGH/LOW-LOW alarms
	5	5	1 = Monitor yellow/orange deviation alarm
	6	6	1 = Monitor rate-of-change alarm
	7	7	1 = Monitor broken transmitter alarm
	8	8	PID algorithm type 0 = Position algorithm 1 = Velocity algorithm
	9	9	0 = Direct acting 1 = Reverse acting
	10	10	1 = Control based on error squared
	11	11	1 = Control based on error deadband
	12	12	1 = Auto-mode lock
	13	13	1 = Cascade-mode lock
	14	14	1 = Setpoint lock
	15	15	0 = Output scale 0% offset 1 = Output scale 20% offset—only valid if output is unipolar. See bit 20.
	LCFL	16	16 and 17
1		17	
2		18	1 = Freeze bias when output is out-of-range
3		19	1 = RAMP/SOAK on the setpoint
4		20	0 = Output is unipolar 1 = Output is bipolar
5		21	0 = PV is unipolar 1 = PV is bipolar
6		22	1 = Perform derivative gain limiting
7–16	23–32	Contains SF program number (if an SF program is scheduled to be called)	

## 9.5 Specifying Loop PID Algorithm

---

### Pos/Vel PID Algorithm

Select POS for the position algorithm or VEL for the velocity algorithm in the POS/VEL PID ALGORITHM field. See Section 9.3 for a discussion of the PID algorithm.

For the position algorithm, the position of the device being controlled is computed based on the error. The velocity form of the PID algorithm computes the change in the device position based on the error.

 <b>WARNING</b>
--

<p><b>Control devices can operate unpredictably causing damage to equipment.</b></p>
--

<p><b>Unpredictable operation can cause damage to equipment and/or death or serious injury to personnel.</b></p>
--

<p><b>Do not change the equation form (velocity to position, or vice versa) while the algorithm is executing.</b></p>
---

## 9.6 Specifying Loop V-Flag Address

### Loop V-Flag Address

Enter an address: C, Y, V, or WY (or also G, VMM, or VMS in a 575), in the LOOP VFLAG ADDRESS field. If you select NONE, no data is written from the V-Flags in the loop.

The V-Flags contain the operational data for a loop. The V-Flags correspond to individual bits making up the 16-bit word LVF. Bits are defined in Table 9-2.

An entry in the LOOP VFLAG ADDRESS field causes loop data to be written from the V-Flags to another address. The address can be either a bit (Y or C) that allocates 15 contiguous bits, or a word (WY or V) that allocates one word for V-Flag data.

The first three V-Flags are designated as control flags. If you create a V-Flag table in V-Memory, for example, the controller reads these three bits in the V-Memory address and writes over the corresponding bits in the LVF word. You can change the loop mode by setting/clearing these control flags. You can read bits 4–15, but any changes that you make to them are overwritten by the controller.

If you select NONE in the LOOP VFLAG ADDRESS field, no data is written from the loop V-Flags. You can still control the loop mode by using an SF program to change the control flag bits in LVF, or manually using TISOFT to write to LVF.

Table 9-2 Loop V-Flags (LVF)

Bit	Loop Function												
1	1 = Go to manual mode												
2	1 = Go to auto mode												
3	1 = Go to cascade mode												
4 & 5	<table border="0"> <tr> <td>4</td> <td>5</td> <td></td> </tr> <tr> <td>0</td> <td>0</td> <td>Loop is in manual mode</td> </tr> <tr> <td>1</td> <td>0</td> <td>Loop is in auto mode</td> </tr> <tr> <td>0</td> <td>1</td> <td>Loop is in cascade mode</td> </tr> </table>	4	5		0	0	Loop is in manual mode	1	0	Loop is in auto mode	0	1	Loop is in cascade mode
4	5												
0	0	Loop is in manual mode											
1	0	Loop is in auto mode											
0	1	Loop is in cascade mode											
6	<table border="0"> <tr> <td>0</td> <td>= Error is positive</td> </tr> <tr> <td>1</td> <td>= Error is negative</td> </tr> </table>	0	= Error is positive	1	= Error is negative								
0	= Error is positive												
1	= Error is negative												
7	1 = PV is in high-high alarm												
8	1 = PV is in high alarm												
9	1 = PV is in low alarm												
10	1 = PV is in low-low alarm												
11	1 = PV is in yellow deviation alarm												
12	1 = PV is in orange deviation alarm												
13	1 = PV is in rate-of-change alarm												
14	1 = Broken transmitter alarm												
15	1 = Loop is overrunning												
16	unused												

## 9.7 Specifying Loop Sample Rate

---

### Sample Rate

Enter a time in seconds in the SAMPLE RATE field.

The sample rate determines how often deviation alarm bits and associated math are evaluated. Sample rates are programmable in 0.1 second increments, with alarms checked at least once every two seconds. The sample rate can be any floating point number between 0.1 and  $1.6772 \times 10^6$  seconds.

## 9.8 Specifying Loop Process Variable Parameters

---

Process Variable Address	<p>Enter an address: V, WX or WY, (or also G, VMM, or VMS in a 575), or select NONE in the PROCESS VARIABLE ADDRESS field.</p> <p>A process variable must be specified for each loop. The variable may be taken from the following.</p> <ul style="list-style-type: none"><li>• A word input or output module — Use WX or WY address in the programming table.</li><li>• A location in V-Memory — Use an address in V-Memory in the programming table. When a special calculation is performed on a process variable, the result (called the computed variable) is stored in V-Memory where it is accessed by the loop.</li></ul> <p>If you select NONE, the loop does not read an address to obtain the process variable. In this case, you can use an SF program, for example, to calculate a process variable. The result can be written to LPV for processing by the loop.</p>
PV Range Low/high	<p>Enter the low and high values of the process variable in the following fields: PV RANGE LOW and PV RANGE HIGH.</p> <p>You must specify the engineering values that correspond to the upper and lower ranges of the input span. If the span is 0 to 100%, the lower range is the engineering value corresponding to 0 volts. If the span is 20% to 100%, then the lower range is the engineering value corresponding to 1 volt. If the span is bipolar, the lower range is the engineering value corresponding to -5 or -10 volts.</p>
PV is Bipolar 20% Offset	<p>Select YES or NO to specify analog inputs as no offset, 20% offset, or bipolar in the following fields: PV IS BIPOLAR, and 20% OFFSET ON PV.</p> <p>The span of the analog inputs may be either 0 to 5.0 volts, 0 to 10 volts, -10 to 10 volts, or -5 to 5 volts. The loop processing feature provides for a linear conversion over any of these process variable input spans.</p> <p>A span of 0 to 5.0 volts (0 to 20.0 milliamps) is referred to as a span of 0 to 100%. A span of 1.0 to 5.0 volts (4.0 to 20.0 milliamps) is referred to as a span of 20% to 100% (20% offset on the process variable). Use the bipolar option with a span of -10 to 10 volts or -5 to 5 volts.</p>
Square Root of PV	<p>Select YES or NO for the square root option in the SQUARE ROOT OF PV field.</p> <p>Select YES if the input for the process variable is from a device (such as an orifice meter) that requires a square root calculation to determine the correct value to use.</p>



## 9.9 Specifying Loop Ramp/Soak Profile

### Defining Ramp/Soak Operation

The ramp/soak feature allows you to define a variation for the process variable by specifying the time characteristics of the loop setpoint (Figure 9-3). The capability of varying the loop setpoint can be useful in a number of processes, such as heat treating and batch cooking.

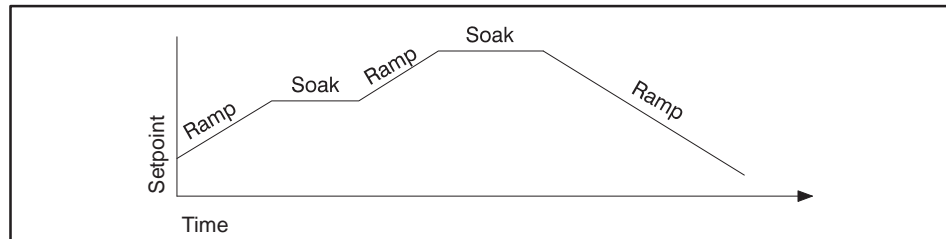


Figure 9-3 Example Ramp/Soak Cycle

You can use simple ramp operations to improve some process startup procedures. For example, the controllers do a bumpless transfer from manual to automatic mode. This transfer holds the process at the initial state when the mode change occurs. A two-step ramp/soak profile can then move the setpoint to a predefined value following the mode change, with minimal disturbance to the process.

### Defining Ramp/Soak Steps

Ramp/Soak is programmed as a set of time periods, or steps. A step can be one of three types: a ramp, a soak, or an end.

- The ramp step changes the loop setpoint linearly from its current value to a new value, at a specified rate of change.
- The soak step holds the setpoint constant for a specified period of time. You can guarantee a soak period by entering a deadband value. This form of soaking ensures that the process variable is within a specified deadband around the setpoint for a specified period of time.
- The end step terminates a ramp/soak profile. When the program reaches an end step, the loop remains in automatic mode and holds the setpoint constant.

You can program a status bit for each step of the ramp/soak. This bit is set to 1 when the loop is executing this step. It is reset when the loop leaves the step. This allows for easy tracking in the RLL program.

### Controlling the Ramp/Soak Operation

Ramp/Soak operation can be controlled by two methods: allowing the profile to be executed automatically, or by writing values to the variables that control ramp/soak.

---

**Automatic** Whenever the loop changes from manual to automatic mode, the loop begins to execute the ramp/soak profile at the initial step (Step 1). The loop continues to execute the profile until an end step is encountered in the profile. At this point, the loop remains in automatic mode, and the setpoint is held at the last value in the profile.

**Using Ramp/Soak Number** Each loop ramp/soak profile has a corresponding 16-bit variable, LRSN, that contains the current step. You can monitor LRSN with an SF program and also write a step number to it with an SF program. The ramp/soak profile changes to the step that is currently contained in LRSN. Note that the step number is zero-based. LRSN contains 0 when the profile is on step #1, 1 when the profile is on step #2, etc.

**Using the Ramp/Soak Flags** Each loop ramp/soak profile has a corresponding 16-bit variable, LRSF, that contains operational and status information for the profile.

When you program a ramp/soak profile, you may optionally specify a RAMP/SOAK FLAG ADDRESS. When you enter an address into this field, the controller writes the ramp/soak data from LRSF to this address. You can use TISOFT or APT or design your RLL program to write to the first three bits at the specified address. The controller reads these bits and then writes their status over the corresponding bits in LRSF. This enables you to change the ramp/soak operation by setting/clearing the three bits as needed. The controller ignores changes that you make in bits 4–16.

You can also monitor LRSF with an SF program and write changes to bits 1–3 with an SF program.

**Ramp/Soak for SP** Select YES or NO in the RAMP/SOAK FOR SP field to indicate whether a ramp/soak program for the loop is to be executed. The RAMP/SOAK PROGRAMMED field is a read-only field and contains YES or NO to indicate the creation of a ramp/soak program for the loop.

**Programming Ramp/Soak** To create a ramp/soak profile for a loop, exit the Loop Programming Table and select the Ramp/Soak Programming Table, shown in Figure 9-4.

## Specifying Loop Ramp/Soak Programming (continued)

RAMP/SOAK FLAG ADDRESS: XXXXXX		PID LOOP XX				
STEP	R/S	STATUS BIT	SETPOINT (UNITS)	RAMP RATE (UNITS/MIN)	SOAK TIME (MIN)	DEADBAND (UNITS)
1	S	XXXXXX			XXXXXXXXXXXXXX	XXXXXXXXXXXXXX
2	R	XXXXXX	XXXXXXXXXXXXXX	XXXXXXXXXXXXXX		
3	S	XXXXXX			XXXXXXXXXXXXXX	XXXXXXXXXXXXXX
4	R	XXXXXX	XXXXXXXXXXXXXX	XXXXXXXXXXXXXX		

EXIT-F1 UP-F2 DOWN-F3 EDIT-F4

Figure 9-4 Ramp/Soak Programming Table

The first field in the table contains the ramp/soak flag address. An entry in this field causes ramp/soak data to be written from the ramp/soak variable (LRSF) to another address, as described on Page 9-15. The address can be either a bit (Y or C) that allocates 5 contiguous bits, or a word (WY or V) that allocates one word for ramp/soak data. The format of the bits in a ramp/soak flag address correspond to the individual bits making up the 16-bit word LRSF. Bits are defined in Table 9-3.

Table 9-3 Loop Ramp/Soak Flags (LRSF)

Bit	Loop Function
1	1 = Restart at the first step. To restart, toggle bit off, on, then off again. The restart occurs on the trailing edge of a square wave.
2	1 = Hold at the current step. To hold, set bit on.
3	1 = Jog to next step. To jog, set bit on. Jog occurs on the rising edge of a square wave.
4	1 = Finish. Indicates ramp/soak is completed.
5	1 = Wait. This bit is set during a soak period when the PV is not within a specified deviation from the SP. The loop holds the soak timer when bit 5 is set.
6	1 = Hold in progress at current step.
7-8	Unused (always returned as 0).
9-16	1 = Contains step number loop is currently executing. Step number is zero-based. Step number contains 0 when the Ramp/Soak is on step #1, 1 when the Ramp/Soak is on step #2, etc.

Enter an address: C, Y, V, or WY (or also G, VMM, or VMS in a 575), in the RAMP/SOAK FLAG ADDRESS field. If you select NONE, no data is written from LRSF.

The rest of the ramp/soak program consists of entering data for each step: setpoint and ramp rate for ramp steps, and soak time and deadband for soak steps.

You can program a status bit (C or Y) for each step of the ramp/soak. This bit is set to 1 when the loop is executing this step. It is reset when the loop leaves the step.

Examples of ramp/soak profiles are shown in Figure 9-5.

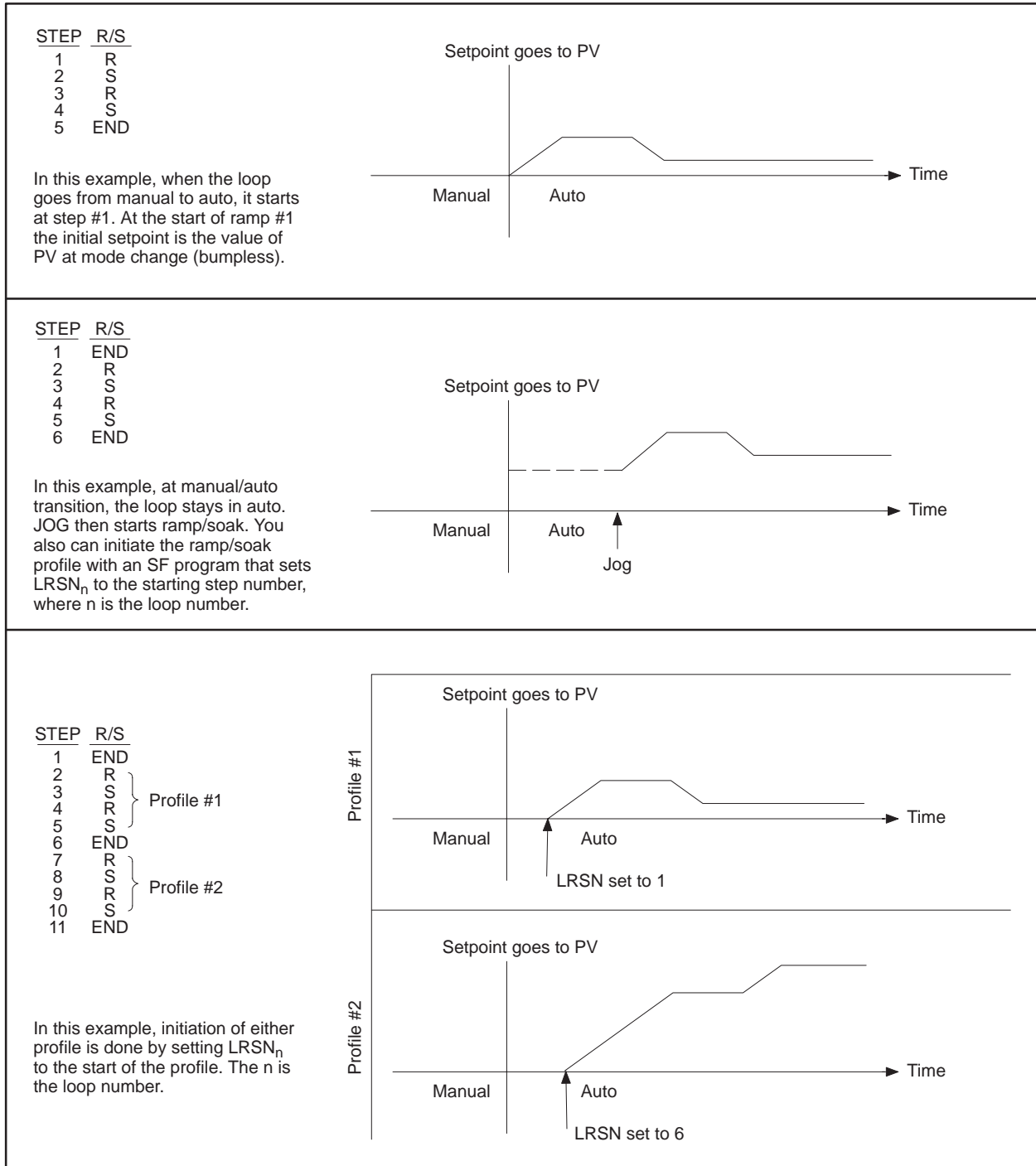


Figure 9-5 Ramp/Soak Table Examples

## 9.10 Specifying Loop Output Parameters

---

<b>Loop Output Address</b>	<p>Enter an address: WY or V (or also G, VMM, or VMS in a 575), in the LOOP OUTPUT ADDRESS field. Select NONE when you do not want the loop to write the output to an address.</p> <p>Use the LOOP OUTPUT ADDRESS field to specify the address into which the loop writes the value of the output. You can select NONE in situations, such as for cascaded loops, in which the outer loop does not require an output address.</p>
<b>Output is Bipolar</b>	<p>Select YES or NO in the OUTPUT IS BIPOLAR field. If you select YES, the output range is -32000 to +32000.</p>
<b>20% Offset on Output</b>	<p>Select YES or NO in the 20% OFFSET ON OUTPUT field. If you select YES, the output range is +6400 to +32000.</p> <p>If you select NO for both fields (no 20% offset and output is not bipolar) then the output range is 0-32000.</p>

## 9.11 Specifying Loop Alarm Deadband

### Alarm Deadband

Enter a value in engineering units for the alarm deadband in the ALARM DEADBAND field.

When you specify an alarm deadband, the controller can provide hysteresis on all alarms (except the rate of change alarm) to prevent them from chattering when the process variable is near one of the alarm limits. The loop does not exit the alarm condition until the process variable has come inside the alarm limit minus the deadband. This is shown graphically in Figure 9-6.

The range for the deadband (LADB) is  $0.0 \leq \text{LADB} \leq (\text{LPVH} - \text{LPVL})$ , where LPVH and LPVL are the process variable high and low limits, respectively. Typically, the deadband ranges from 0.2% to 5% of the span.

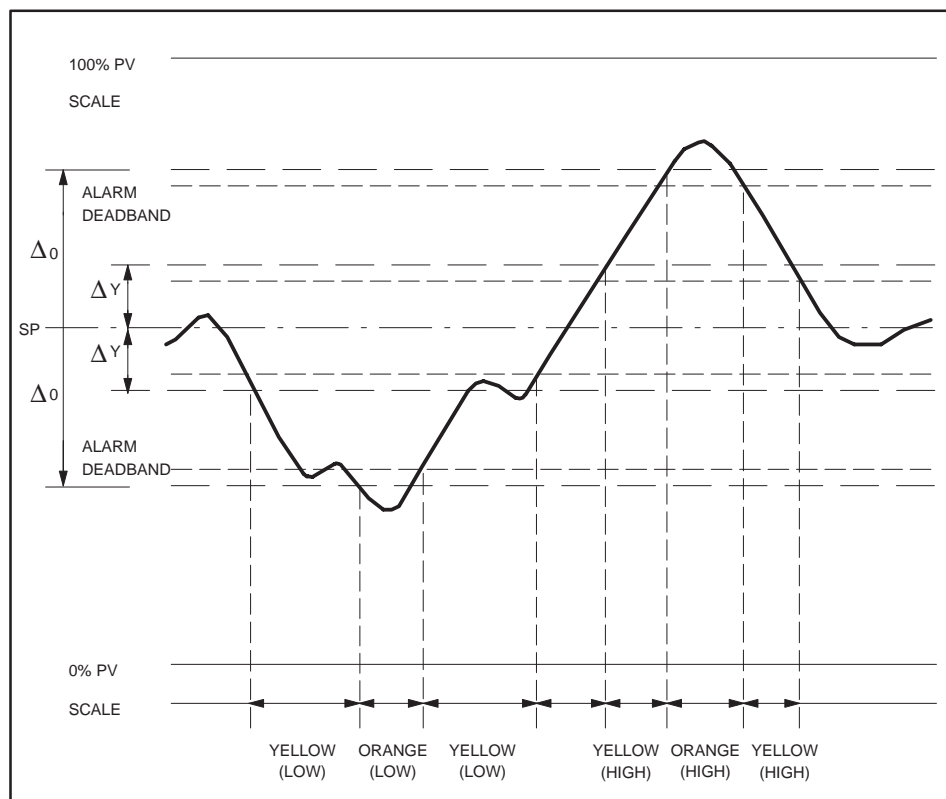


Figure 9-6 Example of Alarm Deadband For Loops

## 9.12 Specifying Loop Process Variable Alarm Limits

---

PV Alarms  
Low-low, Low-high,  
High-high

Enter values in engineering units for the process variable alarm limits in the following fields: LOW-LOW, LOW, HIGH, and HIGH-HIGH. To have the controller monitor the alarm limits, select YES in the following fields: MONITOR LOW-LOW/HIGH-HIGH and MONITOR LOW/HIGH. Otherwise, select NO.

The high-high and low-low alarms can be entered as values requiring critical actions, while the high and low can be set at values requiring remedial measures. The range of possible values that can be used is given below.

- Low-low alarm — real number in engineering units; must be less than or equal to low alarm value and greater than or equal to low range of PV.
- Low alarm — real number in engineering units; must be less than or equal to high alarm value of PV.
- High alarm — real number in engineering units; must be less than or equal to high high alarm value of PV.
- High-high alarm — real number in engineering units; must be greater than or equal to high alarm value and less than or equal to high range of PV.

## 9.13 Specifying Loop Setpoint Parameters

---

**Remote Setpoint** Enter an address: V, K, WX, WY, or LMN (or also G, VMM, or VMS in a 575), in the REMOTE SETPOINT field. Select NONE if there is no remote setpoint.

If you want to use a value external to the loop for the setpoint, you specify the address for this value in the REMOTE SETPOINT field. For example, you can use data from a field transmitter for the setpoint by using a WX address for the transmitter input. Then, specify this WX address in the REMOTE SETPOINT field.

If you want to use a remote setpoint for either cascading loops or performing a special function on the setpoint outside of a loop, you must specify the cascade mode.

If the controller is to control the mode of the inner loop in a cascade configuration, the remote setpoint for the inner loop must be specified as LMN<sub>n</sub> (the output of the outer loop n).

**Clamp SP Limits** Enter values for the setpoint limits in the CLAMP SP LIMITS field. Select NONE if there are no limits, and if zeroes are placed in the high and low fields.



## 9.14 Specifying Loop Tuning Parameters

---

Loop Gain, Reset,  
Rate

Enter values for the loop tuning constants in the following fields: LOOP GAIN, RESET (INTEGRAL TIME), and RATE (DERIVATIVE TIME).

It is not always necessary (or even desirable) to have full three-mode PID control of a loop. Parts of the PID equation can be eliminated by choosing appropriate values for the gain ( $K_c$ ), reset ( $T_i$ ), and rate ( $T_d$ ) thus, yielding a P, PI, PD, I, and even an ID or a D loop.

Removing Integral  
Action

The contribution to the output due to integral action can be eliminated by setting  $T_i = \text{infinity}$ . When this is done, you can manually control the bias term ( $M_x$ ) to eliminate any steady-state offset.

Removing  
Derivative Action

The contribution to the output due to derivative action can be eliminated by setting  $T_d = 0$ .

Removing  
Proportional Action

The contribution to the output due to the proportional term can be eliminated by setting  $K_c = 0$ . Since  $K_c$  is also normally a multiplier of the integral coefficient ( $K_i$ ) and the derivative coefficient ( $K_r$ ), the controller makes the computation of these values conditional on the value of  $K_c$  as follows.

$$\begin{aligned} K_i &= K_c \times (T_s/T_i) && \text{if } K_c \neq 0. \\ &= T_s/T_i && \text{if } K_c = 0. \text{ (for I or ID control)} \end{aligned}$$

$$\begin{aligned} K_r &= K_c \times (T_d/T_s) && \text{if } K_c \neq 0. \\ &= T_d/T_s && \text{if } K_c = 0. \text{ (for ID or D control)} \end{aligned}$$

The units and range of each of the tuning constants follow:

Coefficient	Unit	Range
Proportional Gain, $K_c$	%%	0.01–100.00
Reset (Integral Time)Time, $T_i$	minutes	$0 < T_i \leq \text{Infinity}$
Derivative Time, $T_d$	minutes	$0 \leq T_d < \text{Infinity}$

## Freeze Bias

Select YES in the FREEZE BIAS field to have the bias frozen when output goes out of range. Select NO to have the bias adjusted when output goes out of range.

If you select YES for the FREEZE BIAS option, the controller stops changing the bias  $M_x$  whenever the computed output  $\bar{M}$  goes outside the interval  $[0.0, 1.0]$ . When this option is selected, the computation of the new output  $M_n$  and bias  $M_x$  is done as follows.

$$\text{Calculated Bias} \quad \bar{M}_x = K_i \times e_n + M_{x_{n-1}}$$

$$\text{Calculated Output} \quad \bar{M} = K_c \times e_n - K_r (PV_n - PV_{(n-1)}) + \bar{M}_x$$

$$\begin{aligned} \text{New Output} \quad M_n &= 0.0 \quad \text{if } \bar{M} < 0.0 \\ &= \bar{M} \quad \text{if } 0.0 \leq \bar{M} \leq 1.0 \\ &= 1.0 \quad \text{if } \bar{M} > 1.0 \end{aligned}$$

$$\begin{aligned} \text{New Bias} \quad M_{x_n} &= \bar{M}_x \quad \text{if } 0.0 \leq \bar{M} \leq 1.0 \\ &= M_{x_{n-1}} \quad \text{otherwise} \end{aligned}$$

In this example, it is unlikely that the bias will go all the way to zero. When the PV does begin to come down, the loop begins to open the valve sooner than it would have if the bias had been allowed to go all the way to zero. This action has the effect of lessening the amount of overshoot.

Figure 9-7 illustrates the results of freezing the bias after a disturbance.

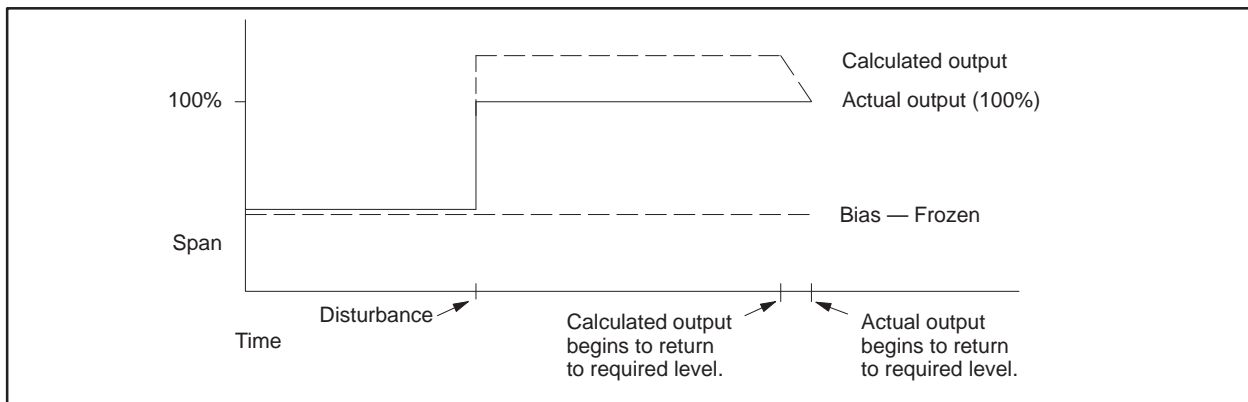


Figure 9-7 Loop Response to the Freeze Bias Option

## Specifying Loop Tuning Parameters (continued)

### Adjust Bias

If you select NO for the FREEZE BIAS option, the controller makes the computation of the bias term conditional on the computation of the output as follows.

$$\text{Calculated Bias} \quad \overline{Mx} = Ki \times e_n + Mx_{n-1}$$

$$\text{Calculated Output} \quad \overline{M} = Kc \times e_n - Kr (PV_n - PV_{(n-1)}) + \overline{Mx}$$

$$\begin{aligned} \text{New Output} \quad M_n &= 0.0 && \text{if } \overline{M} < 0.0 \\ &= \overline{M} && \text{if } 0.0 \leq \overline{M} \leq 1.0 \\ &= 1.0 && \text{if } \overline{M} > 1.0 \end{aligned}$$

$$\begin{aligned} \text{New Bias} \quad Mx_n &= \overline{Mx} && \text{if } 0.0 \leq \overline{M} \leq 1.0 \\ &= M_n - (Kc \times e_n - Kr (PV_n - PV_{n-1})) && \text{otherwise} \end{aligned}$$

With this method, the valve begins to close as soon as the process variable begins moving back toward the setpoint. If the loop is properly tuned, overshoot can be eliminated entirely, assuming that the setpoint is not changing. If the output goes out of range due to a setpoint change, then the loop probably oscillates because the bias term must stabilize again.

The choice of whether to use the default loop action or to freeze the bias depends on the application.

Figure 9-8 illustrates the results of adjusting the bias after a disturbance.

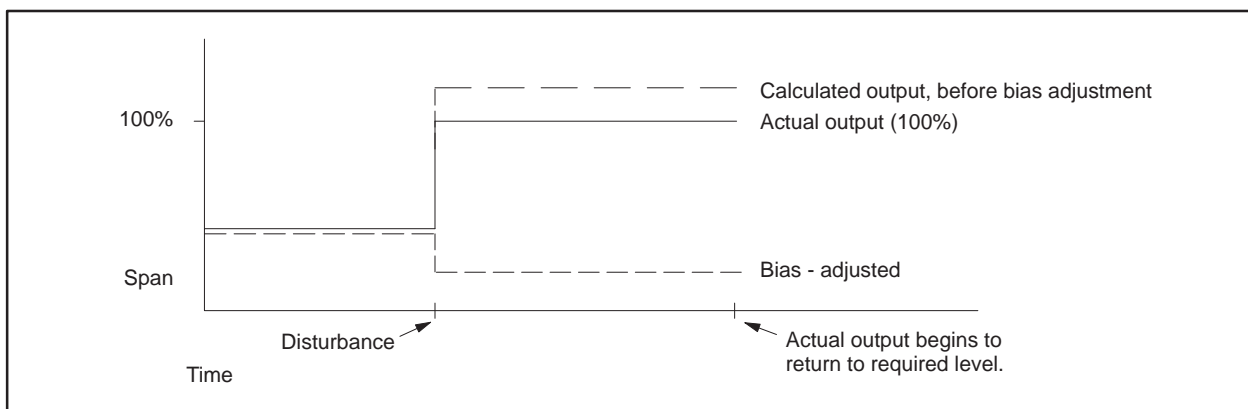


Figure 9-8 Loop Response to the Adjust Bias Option

## 9.15 Specifying Loop Derivative Gain Limiting

**Limiting Coefficient** Enter a value for the derivative gain limiting coefficient in the LIMITING COEFFICIENT field. Select YES or NO in the DERIVATIVE GAIN LIMITING field to have derivative gain limiting done. If you specify NO then derivative gain limiting is not done, even if a value is entered in the field. Typically, Kd should be in the range of 10 to 20.

In the standard PID algorithm, the algorithm responds excessively to process noise if the coefficient of the derivative term (Td/Ts) is significantly above the 10 to 20 range. This causes disturbances that lead to erratic behavior of the process.

To solve this problem, the controller allows you the option of selecting a derivative gain limiting coefficient (Kd). Using this coefficient enables the Process Variable to be filtered with a time constant that is proportional to the derivative time (Td). The PID equations with the derivative gain limiting coefficient follow.

- Position Algorithm.

$$Y_n = Y_{n-1} + \frac{T_s}{T_s + (T_d/K_d)} \times (PV_n - Y_{n-1})$$

$$\overline{Mx} = K_i \times e_n + Mx_{n-1}$$

$$\overline{M} = K_c \times e_n - K_r (Y_n - Y_{n-1}) + \overline{Mx}$$

- Velocity Algorithm.

$$Y_n = Y_{n-1} + \frac{T_s}{T_s + (T_d/K_d)} \times (PV_n - Y_{n-1})$$

$$\Delta M_n = K_c \times (e_n - e_{n-1}) + K_i \times e_n - K_r \times (Y_n - 2 \times Y_{n-1} + Y_{n-2})$$

Variable	Definition	Variable	Definition
M <sub>n</sub>	Loop output	Mx	Bias (Mx is the initial valve position)
K <sub>c</sub>	Proportional gain	T <sub>d</sub>	Rate time
e <sub>n</sub>	Error (SP - PV)	K <sub>i</sub>	Integral gain
T <sub>s</sub>	Sample time	K <sub>d</sub>	Derivative gain-limiting coefficient
PV <sub>n</sub>	Process variable		

## 9.16 Specifying Loop Special Function Call

---

### Special Calculation/ Special Function

Enter an SF program number in the SPECIAL FUNCTION field and select a variable (PROCESS VARIABLE, SETPOINT, or OUTPUT) in the SPECIAL CALCULATION ON field.

If you enter an SF program number in the SPECIAL FUNCTION field but select NONE for the SPECIAL CALCULATION ON field, the SF program is not called for execution.

You can program the loop to call an SF program to do a calculation on any constant, variable, or I/O point. You can schedule the SF program call to be made when the process variable, setpoint, or output is read.

---

**NOTE:** With PowerMath CPUs, an SF program can be compiled or interpreted. If the SF program is compiled, it executes to completion when the loop calls it. If the SF program is interpreted, it is initiated when the loop calls it. While an interpreted SF program is executing, a higher priority process on its queue may interrupt it on any SF statement boundary.

---

### Calculation Scheduled on Setpoint

When the loop is in AUTO or CASCADE mode, the SF program calls at the sample rate and T2 always equals 2. When the loop is in MANUAL mode, the SF program does not call for execution.

### Calculation Scheduled on Process Variable

When the loop is in AUTO, CASCADE, or MANUAL mode, the SF program either executes every 2.0 sec or at the sample rate, whichever is less. The SF program is called at least every 2 seconds to monitor/activate the PV alarms associated with the loop, even though loop calculations are not being done.

In the case of a loop sample time greater than 2.0 seconds, the SF program is called at a 2.0 second interval, and T2 equals 3, indicating that the SF was called on PV. This allows for PV manipulation before PV alarming occurs in the loop. When it is time to do the loop calculation, T2 equals 2 to indicate that the loop calculation is about to begin. This allows for manipulation of both PV and setpoint before the loop calculation is done. If the loop sample time is less than 2.0 seconds, T2 always equal 2.

---

**NOTE:** SF programs called on PV or SP are executed after PV and SP have been determined by the loop, but before any processing is done based on the values obtained. This allows SF programs to manipulate the PV or SP before the loop uses them for output adjustments.

---

---

Calculation  
Scheduled on  
Output

When a loop with a sample time of less than 2.0 seconds calls an SF program, the SF program is actually called twice for every loop calculation.

- After PV and SP are determined, the SF program is called on SP (T2 = 2). This call allows for PV and SP manipulation before PV alarming and loop calculations are run. The loop calculation is then performed and the resultant output value is placed in LMN.
- The SF program is then called on output (T2 = 5) to allow for manipulation of the loop output value in LMN before this value is written to the loop output address.

If the sample time of the loop is greater than 2.0 seconds, the same applies, except that the SF program is called at least every 2.0 seconds, and T2 = 3 if it is not time to do a loop calculation. (Refer to Section 7.6 for a description of T-Memory.)

## 9.17 Specifying Loop Locked Changes

---

Lock Setpoint,  
Auto/Manual,  
Cascade

Select YES or NO for the lock option in the following fields: LOCK SETPOINT, LOCK AUTO/MANUAL, LOCK CASCADE.

The loop programming table provides the option of locking setpoint, auto/manual, or cascade by answering YES in the fields for the option desired. Operator interface devices use the lock bits; these bits are not used by the controller.

## 9.18 Specifying Loop Error Operation

---

### Error Operation

Select SQUARED or DEADBAND in the ERROR OPERATION field. The Error Squared and the Error Deadband options are mutually exclusive. Select NONE if there is to be no calculation on the error value.

In calculating the control equation, the controller uses an error value equal to, or less, than 1.0 (% of PV span over 100). Therefore, selecting error squared gives a lower gain for a higher error. The control equation with error squared is based on signed error squared, instead of the error alone.

For example, an error of 0.5 squared sets the error term in the control equation to 0.25. Since this means the control equation is less responsive to the process variable, error squared is best used with PH control types of applications. When error squared control is selected, the error is calculated as follows.

$$\begin{aligned}\bar{e} &= SP - PV_n \\ e_n &= \bar{e} \times \text{abs}(\bar{e})\end{aligned}$$

Since  $e_n \leq \bar{e}$ , a loop using the error squared is less responsive than a loop using just the error. In fact, the smaller the error, the less responsive the loop.

### Error Deadband

To implement a high gain for large errors, and no gain for small errors, incorporate an error deadband. When error deadband is selected, the controller does not take any action on the output if the process variable is within the yellow deviation limits.

When error deadband control is selected, the error is calculated as:

$$\begin{aligned}\bar{e} &= SP - PV_n \\ e_n &= 0 \quad \text{if } \text{abs}(\bar{e}) < YDEV \\ &= \bar{e} - YDEV \quad \text{if } \bar{e} > YDEV \\ &= \bar{e} + YDEV \quad \text{if } \bar{e} < -YDEV\end{aligned}$$

YDEV is the yellow deviation alarm limit.

### No Error Calculation

If you select the NONE option, no calculation is done on the error value. The error is determined by the following equation.

$$e_n = SP - PV_n$$



## 9.19 Specifying Reverse Acting Loops

**Reverse Acting** Select YES for a reverse-acting loop in the REVERSE ACTING field. Select NO for a direct-acting loop.

The controller can give the gain output as positive or negative and the loop is defined as direct- or reverse-acting (Figure 9-9).

**Direct-Acting Loop** A direct-acting loop is defined to have a positive gain; i.e., a positive change in error ( $SP - PV$ ) results in a positive change in the output from the controller. The value of the output signal increases as the value of the error increases. Note that different manufacturers define forward- and reverse-acting controller responses in different ways.

**Reverse-Acting Loop** A reverse-acting loop is defined to have a negative gain; i.e., a positive change in error ( $SP - PV$ ) results in a negative change in the output from the controller. The value of the output signal decreases as the value of the error increases.

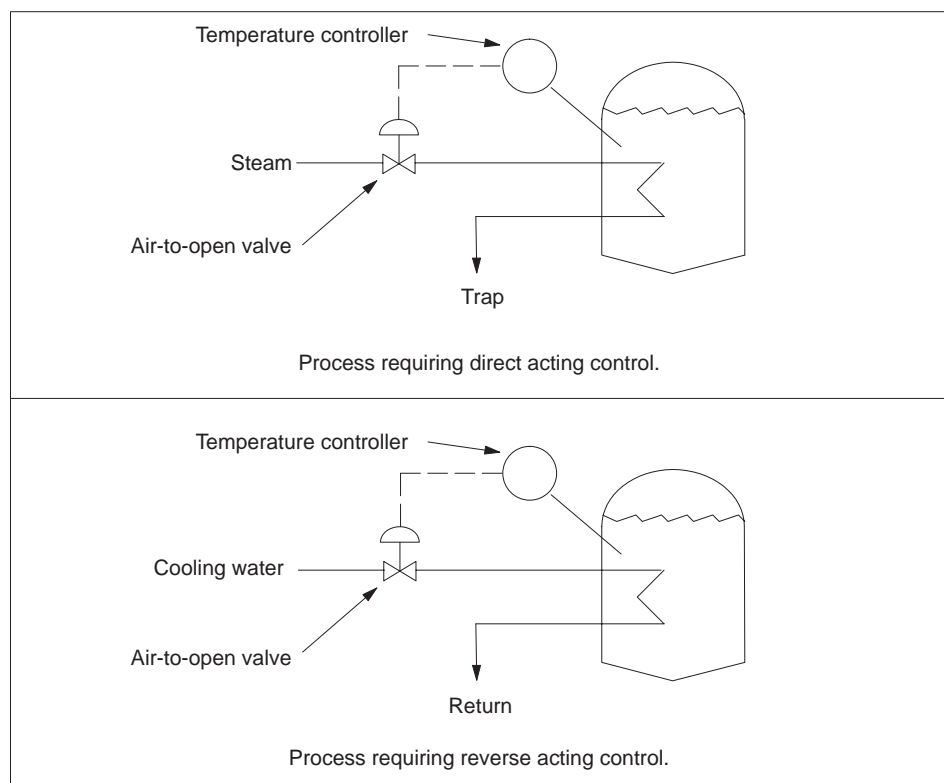


Figure 9-9 Examples of Direct- and Reverse-Acting Control

## 9.20 Specifying Loop Setpoint Deviation Limits

---

### Deviation Alarms Yellow, Orange

Enter values in engineering units for the setpoint deviation limits in the fields: YELLOW and ORANGE. To have the controller monitor the deviation alarm limits, select YES in the MONITOR DEVIATION field. Otherwise, select NO.

The deviation alarm bands are always centered around the setpoint; i.e., the deviation alarm test is actually on the control error. Therefore, they are only processed while the loop is in the auto or cascade mode.

There are two levels of deviation alarms.

- **Yellow Deviation** — This value indicates the maximum allowable error (SP – PV) that sets the yellow deviation alarm. The yellow deviation limit must be within the span of the process variable, and it must be less than or equal to the orange deviation alarm.
- **Orange Deviation** — This value indicates the maximum allowable error (SP – PV) that sets the orange deviation alarm. The orange deviation limit must be within the span of the process variable, and it must be greater than or equal to the yellow deviation alarm.

## 9.21 Specifying Other Loop Process Variable Alarms

---

**Rate of Change Alarm** Enter a value in engineering units for the rate of change alarm in the RATE OF CHANGE ALARM field. To have the controller monitor the rate of change, select YES in the MONITOR RATE OF CHANGE field. Otherwise, select NO.

If you program the controller to monitor the rate of change, an alarm occurs when the rate of change of the process variable exceeds the limit specified. This is a real number in engineering units/minute that is used to set the rate-of-change alarm flag.

**Broken Transmitter Alarm** To have the controller monitor for the broken transmitter condition, select YES in the MONITOR BROKEN TRANSMITTER field. Otherwise, select NO.

If you program the controller to monitor for the broken transmitter condition, an alarm occurs if the raw process variable is outside the valid range designated for the PV. The valid ranges follow.

- Bipolar : -32000 to 32000
- 0% Offset : 0 to 32000
- 20% Offset : 6400 to 32000

Figure 9-10 shows the process variable in broken transmitter alarm.

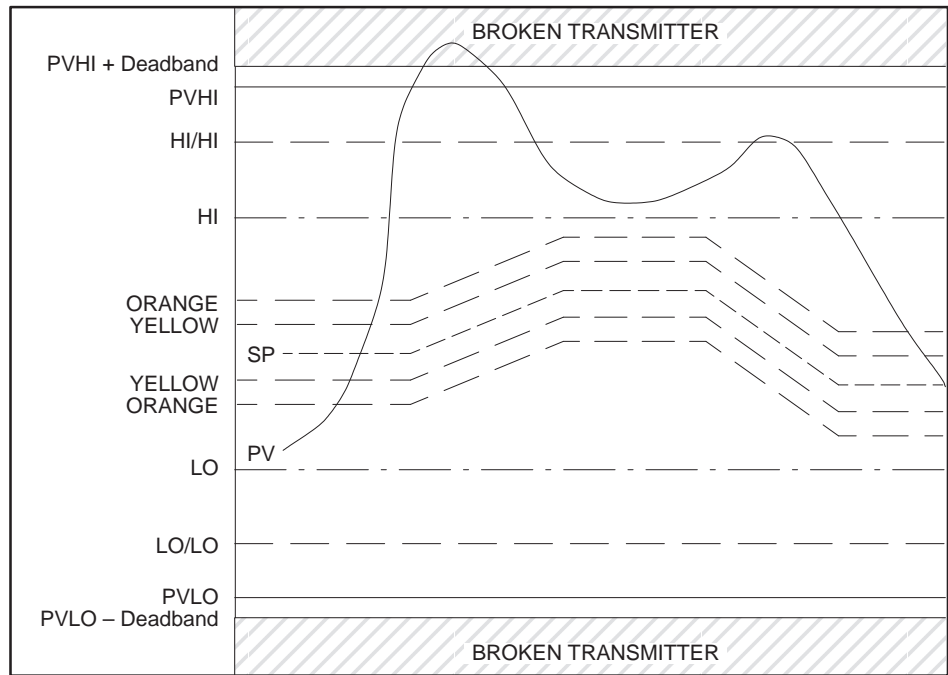


Figure 9-10 Example of Broken Transmitter Alarm

## 9.22 Using SmarTune Automatic Loop Tuning (555 CPUs Only)

---

### Overview of SmarTune

SmarTune™ is an automatic PID loop tuning process that is built into the new SIMATIC 555 CPUs with Release 5.0 or greater firmware. You must also use SoftShop 505 for Windows programming software, Release 2.2 or greater to use the SmarTune loop tuning feature. SmarTune is not supported by TISOFT. For complete information on how to configure SmarTune, refer to your SoftShop user manual.

SmarTune temporarily puts a loop into manual mode. It makes a selectable change to a loop's output ( $M_n$ ) to cause a process variable (PV) movement toward the center of the PV span. Resultant PV changes are sampled. After sampling criteria is met, sampled values are used to calculate theoretical optimum gain ( $K_c$ ), reset ( $T_i$ ), and rate ( $T_d$ ). Theoretical optimums are converted to pragmatic optimums by a heuristic and assigned. The loop is switched to its previous mode and its previous set point (SP) is re-assigned.

Only one SmarTune session is in progress at a time. Other requests are automatically queued. A SmarTune queue can hold all possible entries. Each entry is processed in the order requested. A session may be aborted at any time, whether in progress or queued.

A SmarTune configuration consists of 33 parameters for each loop, which are either value parameters or variable parameters. These parameters allow you to automate loop tuning as desired. For example, you can choose whether or not to automatically load the new tuning parameters directly into the referenced loop.

---

**NOTE:** SmarTune can only be used for position or temperature loops. It is not applicable to velocity loops.

---

The following section describes, in general terms, the PID autotune process for a temperature control loop.

### CAUTION

SmarTune should not be used if a process might experience harmful effects as a result of arbitrary  $M_n$  assignment. During a tuning session,  $M_n$  values are assigned in such a way as to determine the frequency response of a process.

The tuning process may result in process product that does not meet required standards. This product may need to be purged before and/or after a tuning session.

Ensure that your process is designed to handle the results of loop tuning.

---

## The Loop Tuning Process Equation

The PID algorithm consists of three components: the Proportional, the Integral (Reset), and the Derivative (Rate). Each component impacts the output to address the varied characteristics of the process variable. The PID expression is:

$$\text{Output} = P\_Gain * \text{Error} + I\_Gain * \int_0^t \text{Error} (dt) + D\_Gain * \text{Error} (d / dt)$$

where,

Error = Setpoint – Process Variable

P\_Gain = Proportional Gain

I\_Gain = Integral Gain

D\_Gain = Derivative Gain

## The Proportional Component

Temperature control with PID has two regions of operation, the proportional band, and the saturated region. The proportional band is the region above and below the setpoint where the controller output is less than 100%. The heat or cooling output is time proportioned as determined by the PID output. The proportional gain value determines the proportional band.

A typical proportional band might be around  $\pm 30^\circ\text{F}$  for a given machinery temperature control, as shown in Figure 9-11. For example, with a setpoint of  $300^\circ\text{F}$ , a proportional band of  $\pm 30^\circ\text{F}$  would equate to the region between  $270^\circ\text{F}$  and  $330^\circ\text{F}$ , where the controller would be in the proportional band. Outside of this region, on either side, is the saturated region where the controller output would be 100%, which equates to 100% heating or cooling.

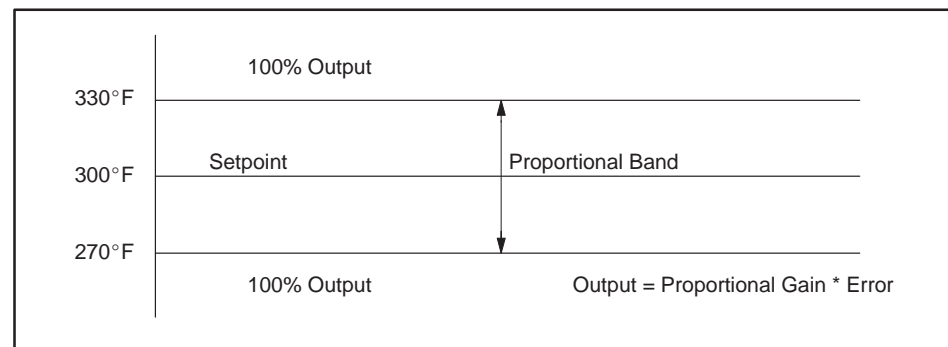


Figure 9-11 Proportional Band

A temperature controller using only the proportional component of the PID expression may experience a steady state error, as shown in Figure 9-12. This error is induced by thermal loading on a temperature zone. As the thermal loading on a temperature zone increases, the magnitude of the steady state error is increased.

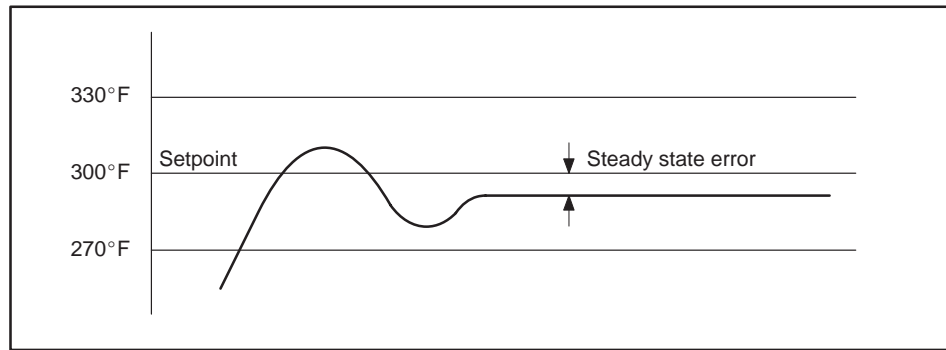


Figure 9-12 Steady State Error

Thermal loading is induced by energy losses to the surroundings, conduction through the machine, as well as the process. A proportional-only controller can resolve this error only to a limited degree.

### The Integral Component

The integral term of the PID expression provides a means to eliminate the error in the proportional band. This term is defined as the *Error* integrated over time. Thus, in the case of the steady state error, the output would be increased (or decreased depending on the sign of the *Error*) over time. The amount of the integral adjustment is determined by the magnitude of the *Error*, and the Integral gain. Excessive Integral gain would cause an oscillation about the setpoint. Likewise, minimal Integral gain would not reduce the *Error* in a timely manner and be ineffective.

---

## The Derivative Component

The Derivative term of the PID expression provides a mathematical means for limiting the rate of change of the process variable. As the rate of change becomes larger, the derivative term reduces the output, resulting in the reduction of the rate of change of the process variable. The Derivative gain defines the magnitude of the output reduction as a function of the rate of change of the process variable. Excessive Derivative gain would result in an undesirable output oscillation as the controller continues to eliminate the *Error*.

When the PID gains are set appropriately, the resulting process variable curve would take on the “ideal curve” appearance, as shown in Figure 9-13.

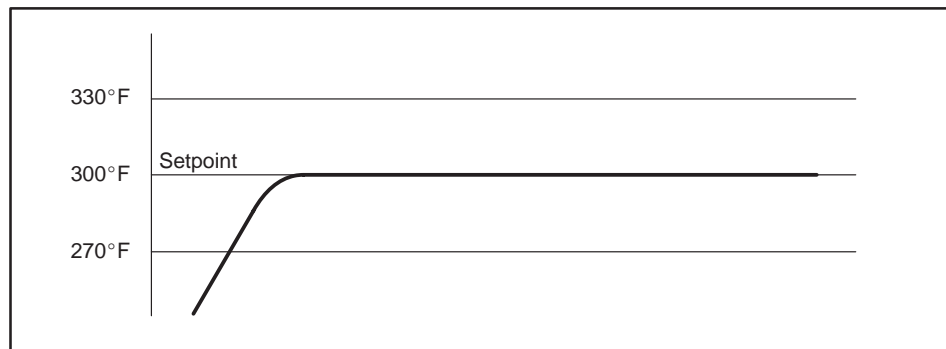


Figure 9-13 Ideal Process Variable Curve

Many factors affect the process variable curve. These factors may take the process beyond where the controller can create the ideal curve. It is the function of the PID SmarTune utility to determine the optimum PID gain values to achieve a response as close to the ideal curve as possible.

Essentially, the SmarTune utility creates a disturbance by initiating a step increase of the PID output. Process variable samples are collected as this increase in output precipitates a change in the process variable. When the sample period is complete, the data collection is analyzed for time lag, gradient, overshoot, steady state error, and oscillation. Using a frequency analysis method, the optimum PID gain values are determined. You can choose to accept the newly calculated gain values, or keep the present PID gain settings.



## Using SmarTune Automatic Loop Tuning (555 CPUs Only), (continued)

---

### Variable Parameters

The SmarTune variable parameters are listed and described in this section. Start Variable is the only variable that must be specified. It names a discrete variable used to activate a SmarTune session. The others may be null.

Variable parameters provide the coupling between a PLC program and SmarTune. If only Start Variable is specified, no program coupling is needed; a session begins when Start Variable becomes true and ends with a loop changing back to its previous mode and SP. Since coupling is done with variables, any program type may be used to monitor and control SmarTune (relay ladder logic, SFPGM, or SFSUB).

Table 9-4 lists the variable parameters used by SmarTune. The following paragraphs describe the parameters.

Table 9-4 Variable Parameters

Name	Type	Allowable Variable Types
Start Variable	discrete	X Y C WX WY V
Abort Variable	discrete	X Y C WX WY V
Ack Variable	discrete	Y C WY V
SmarTune Restart	discrete	X Y C WX WY V
Status Variable	word	WY V
PIN Variable	word	WX WY V
Previous Mode	word	V
Previous SP	real	V
Previous Output	word	V
Previous Gain	real	V
Previous Reset	real	V
Previous Rate	real	V
Calculated Gain	real	V
Calculated Reset	real	V
Calculated Rate	real	V

---

### Start Variable, Abort Variable, Ack Variable

These three discrete variables allow easy activation/deactivation via an RLL program, as shown in Figure 9-14.

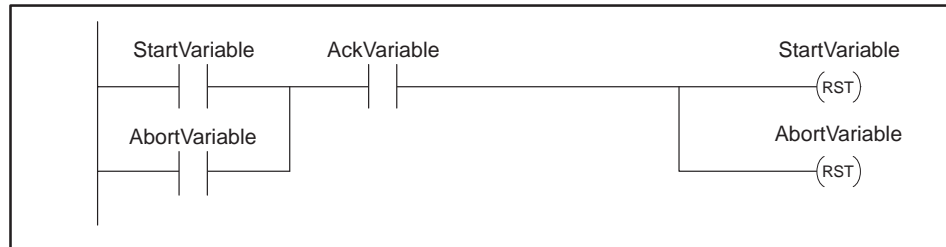


Figure 9-14 Example of Activation/Deactivation of Auto Tuning Process

These variables could just as easily be manipulated with IF, IMATH or MATH statements in an SFPGM or SFSUB. Allowed discrete variables include bits in a V-memory word.

- When Start Variable transitions from a false to a true, a SmarTune session is activated.
- When Abort Variable is true, a SmarTune session is deactivated.
- If both are true, a session is deactivated, and Start Variable must transition before a session will be activated.
- If a SmarTune session is already queued or in progress, Start Variable transitions are ignored.

Ack Variable acknowledges that SmarTune has detected that Start Variable or Abort Variable is true. It is used to synchronize Start Variable and Abort Variable program logic with SmarTune. If not used, Start Variable and AbortVariable should remain true for a relatively large amount of time. What constitutes a large amount of time depends on program size and time slice assignments. See the discussion for Activation Time Slice for further guidance.

### SmarTune Restart

If this discrete variable is true, then SmarTune is restarted completely. SmarTune will act as if a run-program-run transition occurred. If SmarTune Restart is specified in more than one configuration, all are tested for true and acted upon.

### **Status Variable**

This word variable reports on the current state of a session. Three bits are used in the word to allow easy use by an RLL program. Bit 2 is set when a SmarTune session is completed, with or without errors or warnings. If bit 3 is also set then an error was detected. Similarly, if bit 4 is set, then a warning condition occurred. If only bit 2 is set, then a SmarTune session completed with no errors or warnings. See Table 9-5 for a complete listing. Note that entries with x's represent ranges of values.

### **PIN Variable**

PIN Variable and PIN are provided to force a two-step procedure to be followed before a loop is tuned. To use this feature, PIN Variable and PIN must both be set. If PIN Variable is a null or PIN is zero, then SmarTune activation is a one-step procedure dependent only on Start Variable. If both are specified, then PIN Variable must equal PIN or a SmarTune session will not be started or queued.

### **Previous Mode**

If Previous Mode is used, it is set by SmarTune to a value which will switch a loop to its pre-session mode when written to a loop's LVF. This was conceived for use when Automatic Download has been configured as false, but may be used for other purposes. If Automatic Download is false, a loop is left in manual mode with its output set to Safe Output when a tuning session has completed. When AutomaticDownload is true, a loop is switched back to its previous mode and is assigned its previous SP on completion.

### **Previous SP, Previous Output, Previous Gain, Previous Reset, Previous Rate**

You can use these five parameters to record the prior SP,  $M_n$ , Kc, Ti, and Td of a loop before a SmarTune session starts. See Previous Mode for a short discussion on why they would be configured.

### **Calculated Gain, Calculated Reset, Calculated Rate**

You can use these three variables to record the tuning values calculated by SmarTune. See Previous Mode for a short discussion on why they would be configured.



## Using SmarTune Automatic Loop Tuning (555 CPUs Only), (continued)

Table 9-5 Status Code Bit Values (continued)

<b>rcew</b>	<b>fghi</b>	<b>jklm</b>	<b>nopq</b>	<b>Description</b>
0110	xxxx	xxxx	0001	PIN mismatch
0110	xxxx	xxxx	0010	Loop would not go to Manual Mode
0110	xxxx	xxxx	0011	Loop not completely under SmarTune control
0110	xxxx	xxxx	0100	SmarTune timeout (Maximum time exceeded)
0110	xxxx	xxxx	0101	Not enough free memory
0110	xxxx	xxxx	0110	Out of required system resources
0110	xxxx	xxxx	0111	PV greater than high stop
0110	xxxx	xxxx	1000	PV lower than low stop
0110	xxxx	xxxx	1001	PV change too small
0110	xxxx	xxxx	1010	Operation aborted
0110	xxxx	xx01	xxxx	Sample interval (LTS) range error (allowed range: 0.1 ms to 2 hours)
0110	xxxx	xx10	xxxx	PV (LPV) or output (LMN) range error (range < 0.00001)
0110	xxxx	xx11	xxxx	Sample size too small (probably would never happen) size < 33 (increase STEP or decrease NOISE)
0110	xxx0	01xx	xxxx	PV/output inconsistent 1; Noisy PV/output signal?
0110	xxx0	10xx	xxxx	PV/output inconsistent 2; Noisy PV/output signal?
0110	xxx0	11xx	xxxx	PV/output inconsistent 3; Noisy PV/output signal?
0110	xxx1	00xx	xxxx	PV/output inconsistent 4; Noisy PV/output signal?
0110	xxx1	01xx	xxxx	PV/output inconsistent 5; Noisy PV/output signal?

Table 9-6 lists the value parameters used by SmarTune, with the default values and the ranges possible for each.

Table 9-6 Value Parameters

Name	Default Value	Range
Max Time	30.0 minutes	0 to 71582 minutes (maximum is about 49 days)
Noise Band	0.005 of PV range	PV range (engineering units)
Step Change	0.07 of PV range	PV range (engineering units)
Wait Time	0.5 minutes	same as Max Time
PIN	0 (PIN not required)	0 to 32767
Automatic Download	TRUE	TRUE/FALSE
Calculate Derivative	FALSE	TRUE/FALSE
Safe Output	use Previous Output	Previous Output, 0 to 32000
High Stop	0.8 of PV range	PV range (engineering units)
Low Stop	0.2 of PV range	PV range (engineering units)
Largest Gain	8000000.0 %/%	real
Smallest Gain	0.0000008 %/%	real
Largest Reset	8000000.0 minutes	real
Smallest Reset	0.0000008 minutes	real
Largest Rate	8000000.0 minutes	real
Smallest Rate	0.0000008 minutes	real
Activation Time Slice	0	0: not configured here, 1 to 255 ms
Calculation Time Slice	0	0: not configured here, 1 to 255 ms

### Max Time

Max Time is a time in minutes. When a SmarTune session is started, a timer is set to this value. If that timer expires before the session has completed, the session is aborted with an error (see Table 9-5).

### Noise Band

When electrical signals are converted to values, they vary randomly by insignificant amounts. An insignificant amount is application dependent. Noise Band gives a value in engineering units denoting the boundary between a significant and an insignificant change. If a PV value differs from a prior value by a Noise Band or greater amount, then a PV change has occurred. Otherwise the PV is considered unchanged. Some errors and warnings in Table 9-5 could be caused by an incorrect Noise Band setting. A correct setting may be calculated from hardware specifications, or determined by experiment and observation, or both.

### Step Change

SmarTune works best with a PV change of about 7%. This change is accomplished by changing  $M_n$  proportional to the ratio between Step Change and PV span. Step Change is specified in engineering units of the PV. If a PV span is 0 to 60 degrees and Step Change is 5 degrees, then  $M_n$  would be changed by about 2667 ( $5/60 * 32000$ ). Due to round-off error, the actual value might be slightly different. This example is based on an  $M_n$  span of 0 to 32000. If a 20% offset on output is selected for a loop, an  $M_n$  change of about 2133 ( $M_n$  span of 25600) would be accomplished. See Table 9-5 for possible warnings and errors associated with Step Change.

### Wait Time

The SmarTune sample algorithm looks for a PV to change by Step Change or to quit changing. Wait Time is required to determine when a PV has quit changing. If a PV value does not change by a Noise Band amount within a Wait Time period, then it has stopped changing.

### PIN

PIN and PIN Variable are provided to force a two-step procedure to be followed before a loop is tuned. To use this feature, PIN and PIN Variable must both be set. If PIN is a zero or PIN Variable is a null, then SmarTune activation is a one-step procedure dependent only on Start Variable. If both are specified, PIN Variable must equal PIN or a SmarTune session will not be started or queued.

### Automatic Download

If Automatic Download is true, a loop tuning session is accomplished with minimum additional support. After tuning values are calculated, three actions are taken:

- Calculated  $K_c$ ,  $T_i$ , and  $T_d$  are written to a loop.
- The loop is changed to its prior mode.
- The loop's SP is assigned its prior value.

### Calculate Derivative

If Calculate Derivative is false, only  $K_c$  and  $T_i$  are calculated, and  $T_d$  is set to zero. If Calculate Derivative is true,  $K_c$ ,  $T_i$ , and  $T_d$  are calculated.

### Safe Output

Safe Output is an  $M_n$  value that will not cause any harm to a process. The default is to use the loop  $M_n$  value just prior to a tuning session start.

---

### **High Stop**

If a PV goes above High Stop,  $M_n$  is set to Safe Output and an error is declared (see Table 9-5).

### **Low Stop**

If a PV goes below Low Stop,  $M_n$  is set to Safe Output and an error is declared (see Table 9-5).

### **Largest Gain, Largest Reset, Largest Rate**

If a calculated value is larger than a configured value, then it is reduced to a configured value and a warning is declared (see Table 9-5).

### **Smallest Gain, Smallest Reset, Smallest Rate**

If a calculated value is smaller than a configured value, then it is increased to a configured value and a warning is declared (see Table 9-5).

### **Activation Time Slice, Calculation Time Slice**

These two values set how much impact SmarTune will have on PLC scan time. If zero in all configurations, a default will be used (2 milliseconds). Otherwise, in each category, the largest value specified will be used.

Activation Time Slice controls how responsive SmarTune is to tuning session requests. Increase this value if SmarTune is taking an excessive amount of time to start a tuning session. Remember that as this value is increased, PLC scan time will increase.

Calculation Time Slice determines how much real time it will take to calculate tuning parameters. It is possible a calculation might take 20 seconds or more of PLC time. If a PLC has a scan time of 10 milliseconds and Calculation Time Slice is 2 milliseconds, then a 20-second calculation would take about 120 seconds in real time:  $(10\text{ms} + 2\text{ms}) / 2\text{ms} * 20\text{s} = 120\text{s}$ . The above formula is an algebraic simplification of:  $X\text{s} / (2\text{ms} / 12\text{ms}) = 20\text{s}$  where X is real time in seconds. This value should be increased if a SmarTune session takes an excessive amount of time with a status of calculating (see Table 9-5 and Status Variable). Remember that as this value is increased, PLC scan time will increase while a SmarTune session is calculating.



*Appendix A*

# Memory and Variable Types

---

A.1	RLL Variable Access .....	A-2
A.2	SF Program Variable Access .....	A-3

## A.1 RLL Variable Access

Table A-1 lists variable types used in all of the 545, 555, and 575 controllers and which can be accessed by RLL instructions.

Table A-1 Controller Variable Types

Variable Type	RLL Access	Controller	Notes
Constant (K)	Read Only	All	
Control Relay (C)	Read/Write	All	
Drum (DSP, DCP, DSC, DCC)	Read/Write	All	<p>The time-driven drum (DRUM) uses the count preset stored in L-Memory when the DRUM is programmed. A new value for count preset written by RLL has no effect on DRUM operation.</p> <p>It is possible to read/write data to/from drum memory areas for an unprogrammed drum, using these memory locations like V-Memory. If you use TISOFT to display values in DSP or DSC memory, any value not in the range of 1–16 is displayed as 16. An APT program can display values that are greater than 16 for these variables.</p>
Global (G)	Read/Write	575 only	
Image Register (X, WX) (Y, WY)	Read Only Read/Write	All	
PGTS Discrete Parameter Area (B)	Read/Write	All	
PGTS Word Parameter Area (W)	Read/Write	All	
Status Word (STW)	Read Only	All	STW1 cannot be accessed by a multi-word move instruction, e.g., MOVE, MOVW. STW1 is a local variable that is only valid within a given RLL task. Do not do multiple-word move operations that begin with STW1.
Timer/Counter (TCP, TCC)	Read/Write	All	
Variable (V)	Read/Write	All	
VME (VMM, VMS)	Read/Write	575 only	

## A.2 SF Program Variable Access

Table A-2 lists the variable types defined by the 545, 555, and 575 controllers that can be used in SF programs.

Table A-2 Variable Names and Types Used in SF Programs

Name	Mnemonic	Units	Real Only	Integer Only	Read Only	See Note
Analog Alarm/Alarm Acknowledge Flags	AACK			✓		15
Analog Alarm Deadband	AADB	eu				1, 2, 8
Most Significant Word of Analog Alarm C-flags	ACFH			✓		1
Least Significant Word of Analog Alarm C-flags	ACFL			✓		1
Analog Alarm Error	AERR	eu			✓	3
Analog Alarm High Alarm Limit	AHA	eu				1, 2, 8
Analog Alarm High-High Alarm Limit	AHHA	eu				1, 2, 8
Analog Alarm Low Alarm Limit	ALA	eu				1, 2, 8
Analog Alarm Low-Low Alarm Limit	ALLA	eu				1, 2, 8
Analog Alarm Orange Deviation Alarm Limit	AODA	eu				1, 2, 8
Analog Alarm Process Variable	APV	eu				2
Analog Alarm Process Variable High Limit	APVH	eu	✓			1, 7
Analog Alarm Process Variable Low Limit	APVL	eu	✓			1, 7
Analog Alarm Rate of Change Alarm Limit	ARCA	eu/ min	✓			1, 7
Analog Alarm Setpoint	ASP	eu				2, 8
Analog Alarm SP High Limit	ASPH	eu				1, 2, 8
Analog Alarm SP Low Limit	ASPL	eu				1, 2, 8
Analog Alarm Sample Rate	ATS	sec	✓			1
Analog Alarm Flags	AVF			✓		9
Analog Alarm Yellow Deviation Alarm Limit	AYDA	eu				1, 2, 8
Alarm Peak Elapsed Time	APET	ms		✓	✓	16
Loop Alarm/Alarm Acknowledge Flags	LACK			✓		15
Loop Alarm Deadband	LADB	eu				1, 2, 8
Most Significant Word of Loop C-flags	LCFH			✓		1
Least Significant Word of Loop C-flags	LCFL			✓		1
Loop Error	LERR	eu			✓	3
Loop Alarm High Limit	LHA	eu				1, 2, 8
Loop Alarm High-High Limit	LHHA	eu				1, 2, 8
Loop Gain	LKC	%/%	✓			

## SF Program Variable Access (continued)

Table A-2 Variable Names and Types Used in SF Programs (continued)

Name	Mnemonic	Units	Real Only	Integer Only	Read Only	See Note
Loop Derivative Gain Limiting Coefficient	LKD		✓			
Loop Low Alarm Limit	LLA	eu				1, 2, 8
Loop Low-Low Alarm Limit	LLLA	eu				1, 2, 8
Loop Output	LMN	%				10
Loop bias	LMX	%				11
Loop Orange Deviation Limit	LODA	eu				1, 2, 8
Loop Process Variable	LPV	eu				2
Loop PV High Limit	LPVH	eu	✓			1, 7
Loop PV Low Limit	LPVL	eu	✓			1, 7
Loop Rate of Change Alarm Limit	LRCA	eu/ min	✓			1, 8
Loop Ramp/Soak Flags	LRSF			✓		9
Loop Ramp/Soak Step Number	LRSN			✓		14
Loop Setpoint	LSP	eu				2, 8
Loop Setpoint High Point	LSPH	eu				1, 2, 8
Loop Setpoint Low Limit	LSPL	eu				1, 2, 8
Loop Rate	LTD	min	✓			
Loop Reset	LTI	min	✓			
Loop Sample Rate	LTS	sec	✓			1
Loop V-flags	LVF			✓		9
Loop Yellow Deviation Alarm Limit	LYDA	eu				1, 2, 8
Loop Peak Elapsed Time	LPET	ms		✓	✓	16
SF Subroutine Parameters	P					5, 6
SF Error Code	SFEC			✓		4, 12
SF Program Peak Elapsed Time	PPET	ms		✓	✓	16
SF Subroutine Peak Elapsed Time	SPET	ms		✓	✓	16
Constant Memory	K				✓	
Temporary memory	T					4
RLL Tasks Peak Elapsed Time	TPET	ms		✓	✓	16

Table A-2 Variable Names and Types Used in SF Programs (continued)

Name	Mnemonic	Units	Real Only	Integer Only	Read Only	See Note
Discrete Input accessed from an SF Program	X			✓	✓	14
Discrete Output accessed from an SF Program	Y			✓		14
Control Relay accessed from an SF Program	C			✓		14
Drum Counter Preset	DCP			✓		
Drum Step Preset	DSP			✓		
Drum Count Current	DCC			✓		
Drum Step Current	DSC			✓		
Timer/Counter Preset	TCP			✓		
Timer/Counter Current	TCC			✓		
Variable Memory	V					
Word Input	WX				✓	
Word Output	WY					
Global Memory	G*					
VME Memory (A16 Addresses)	VMS*					
VME Memory (A24 Addresses)	VMM*					

\* These variables are supported only by the 575 controllers.

Unit Abbreviations	Meaning
eu	engineering units
ms	milliseconds
min	minutes
sec	seconds
%%	gain
%	percent

### NOTES to Table A-2:

1. Variable is read-only if S-memory is in ROM.
2. When accessed as an integer, the value returns as a scaled-integer number between 0 and 32000. When accessed as a real, the variable returns as a value in engineering units between the low-limit and the high-limit.
3. When accessed as an integer, the value returns as a scaled-integer number between -32000 and 32000. When accessed as a real, the variable returns as a value in engineering units between – span and + span.
4. This variable type may only be accessed in an SF program or SF subroutine.
5. This variable type may only be accessed in an SF subroutine.
6. The access restrictions are dependent on the type of variable passed to the subroutine.
7. If xPVL is changed to a value larger than xPVH, then xPVH is set to the new xPVL. Similarly, if xPVH is changed to a value smaller than xPVL, then xPVL is set to the new xPVH.
8. If xPVL or xPVH is modified and the current value of any of these variables is outside the new PV range, the value clamps to the nearest endpoint of the new PV range.
9. When written, only the control bits are actually modified. When read, only the status bits are returned, the control bits are always returned as zeros.
10. The value is dependent upon the PID algorithm in use as follows:  
  
Position: The value is a percent between 0.0 and 1.0 (if accessed as a real) , or 0 and 32000 (if accessed as an integer).  
  
Velocity: The value is a percent-of-change between -1.0 and 1.0 (if accessed as a real), or -32000 and 32000 (if accessed as an integer).
11. These variables are meaningless if the Velocity PID algorithm is being used.

- 
12. The value written to SFEC must range from 0–255. Unless “Error Continuation” is specified in the SF program, writing a non-zero value to SFEC terminates the program with the specified error code.
  13. LRSN is only effective if the loop is in Auto and ramp/soak for that loop is enabled. Error #49 is logged if the step is not programmed. If the step is programmed, the loop exits the current step and enters the specified step. Writing a value larger than the number of the last programmed ramp/soak step to LRSN completes the ramp/soak and sets the ramp/soak finish bit flag word.

LRSN is zero-based. LRSN contains 0 when the ramp/soak is on step #1, 1 when the ramp/soak is on step #2, etc.

14. When you read a discrete point in an SF program expression, a zero is returned if the discrete bit is off; a one is returned if the discrete bit is on. When you write to a discrete point in an SF program expression, the discrete bit is turned off if the value is zero; the discrete bit is turned on if the value is non-zero.
15. The bit format is shown in Table A-3 for the words AACK and LACK.

Bits 1–4 indicate which alarm is active.

Bits 9–12 indicate which alarms have not been acknowledged. You can acknowledge an alarm by using an operator interface to write a 1 to one of these bits.

Table A-3 Bit Format for Words AACK and LACK

Bit Number	Alarm
1	1 = PV is in broken transmitter alarm.
2	1 = PV is in rate-of-change alarm.
3	1 = PV is in high-high/low-low alarm.
4	1 = PV is in orange deviation alarm.
5–8	Bits 5–8 are not used.
9	1 = Broken transmitter alarm is unacknowledged.
10	1 = Rate-of-change alarm is unacknowledged.
11	1 = High-high/low-low alarm is unacknowledged.
12	1 = Orange deviation alarm is unacknowledged.
13–16	Bits 13–16 are not used.

16. PET variables apply to all of the 545, 555, and 575 controllers.

APET<sub>n</sub> contains the peak elapsed time for each analog alarm, which is the time from which the alarm is scheduled, until the process completes execution (n = 1–128).

LPET<sub>n</sub> contains the peak elapsed time for each loop, which is the time from which the loop is scheduled, until the process completes execution (n = 1–64).

PPET<sub>n</sub> contains the peak elapsed time for each SF program, which is the time from which the SF program is scheduled, until the process completes execution (n = 1–1023). PPET is only valid for SF programs that are queued from RLL.

SPET<sub>n</sub> contains the peak elapsed time for each SF subroutine, which is the time from which the SF subroutine is scheduled, until the process completes execution (n = 1–1023). SPET is only valid for SF subroutines that are queued from RLL.

TPET<sub>n</sub> contains the peak elapsed time for the execution of an RLL task, TPET1 for TASK1 and TPET2 for TASK2.



*Appendix B*

# RLL Memory Requirements

---

B.1 Memory Requirements ..... B-2

## B.1 Memory Requirements

This appendix gives the complete set of Relay Ladder Logic instructions used by the Series 505 controllers. Table B-1 lists each instruction, its mnemonic code, the range of reference numbers it may be assigned, and the minimum amount of L-memory it uses.

When calculating the actual amount of memory used by an instruction, add one word for each of the following cases:

- A box instruction reference number greater than 255.
- An image register ( X, Y, WX, WY ) point number greater than 1024.
- A control relay point number greater than 512.
- A TCP or TCC reference number greater than 128.

Table B-1 RLL Memory Requirements

Instruction		Mnemonic	Words L-Mem	Reference Number Range
Absolute Value		ABSV	3	1-32767 <sup>1</sup>
Add		ADD	4	1-32767 <sup>1</sup>
Bit Clear		BITC	3	1-32767 <sup>1</sup>
Bit Pick		BITP	3	1-32767 <sup>1</sup>
Bit Set		BITS	3	1-32767 <sup>1</sup>
Convert Binary To BCD		CBD	3	1-32767 <sup>1</sup>
Convert BCD To Binary		CDB	4	1-32767 <sup>1</sup>
Compare		CMP	5	1-32767 <sup>1</sup>
Coils	Y	$Y_n$ --( )--   --( / )--	1	Table 3-3
	Y, Immediate	$Y_n$ --( )--   --( / I )--	3	Table 3-3
	Set/Reset Y	$Y_n$ --(SET)--   --(RST)--	3	Table 3-3
	Set/Reset Y, Immediate	$Y_n$ --(SETI)--   --(RSTI)--	3	Table 3-3
	C	$C_n$ --( )--   --( / )--	1	Table 3-3
	Set/Reset C	$C_n$ --(SET)--   --(RST)--	3	Table 3-3
	Bit-of-Word	$V_{n,b}$ --( )--   --( / )--	3	n: Configurable b: 1 - 16
	Set/Reset Bit-of-Word	$V_{n,b}$ --(SET)--   --(RST)--	3	n: Configurable b: 1 - 16
<sup>1</sup> Numbers are for reference only.				

Table B-1 RLL Memory Requirements (continued)

Instruction		Mnemonic	Words L-Mem	Reference Number Range
Contacts	X, Y	$X_n$ $X_n$ --()--   --(/)--	1	Table 3-3
	X, Immediate	$X_n$ $X_n$ --(I)--   --(I)--	3	Table 3-3
	C	$C_n$ $C_n$ --()--   --(/)--	1	Table 3-3
	Bit-of-Word	$V_{n,b}$ $V_{n,b}$ --()--   --(/)--	3	n: Configurable b: 1 – 16
	Relational	$V_n$ $V_m$ --(<>)--	3	n: Configurable b: 1 – 16
Control Relay		C	1	Table 3-3
Counter		CTR	2	Configurable
Discrete Control Alarm Timer		DCAT	6	Configurable
Date Compare		DCMP	3	1–32767 <sup>1</sup>
Divide		DIV	4	1–32767 <sup>1</sup>
Drum		DRUM	50	Configurable
Date Set		DSET	3	1–32767 <sup>1</sup>
Event Drum		EDRUM	66	Configurable
End Unconditional		END	1	None
End Conditional		END(C)	1	None
Go To Subroutine		GTS	2	1–255
Indexed Matrix Compare		IMC	33	1–32767 <sup>1</sup>
Immediate I/O Read/Write		IORW	4	1–32767 <sup>1</sup>
Jump		JMP	1	1–8
End Jump		JMP(E)	1	1–8
End Jump Conditional		JMP(E)	2	1–8
Label		LBL	1	1–255
Load Address		LDA	5 <sup>3</sup>	1–32767 <sup>1</sup>
<sup>1</sup> Numbers are for reference only. <sup>2</sup> Varies with controller model. See documentation for specific controller for number supported. <sup>3</sup> Add 1 word for each index parameter.				

## Memory Requirements (continued)

Table B-1 RLL Memory Requirements (continued)

Instruction	Mnemonic	Words L-Memory	Reference Number Range
Load Data Constant	LDC	3	1-32767 <sup>1</sup>
Lock Memory (575 Only)	LOCK	4	1-32767 <sup>1</sup>
Motor Control Alarm Timer	MCAT	9	Configurable
Master Control Relay (MCR)	MCR	1	1-8
End MCR	MCR(E)	1	1-8
End MCR Conditional	MCR(E)	2	1-8
Maskable Event Drum Discrete	MDRMD	68	Configurable
Maskable Event Drum Word	MDRMW	54	Configurable
Move Image Register From Table	MIRFT	4	1-32767 <sup>1</sup>
Move Image Register To Table	MIRTT	4	1-32767 <sup>1</sup>
Move Discrete Image Register To Word	MIRW	4	1-32767 <sup>1</sup>
Move Element	MOVE	5 <sup>3</sup>	1-32767 <sup>1</sup>
Move Word	MOVW	4	1-32767 <sup>1</sup>
Multiply	MULT	4	1-32767 <sup>1</sup>
Move Word From Table	MWFT	5	Configurable
Move Word With Indirect Addressing	MWI	5	1-32767 <sup>1</sup>
Move Word To Discrete Image Register	MWIR	4	1-32767 <sup>1</sup>
Move Word To Table	MWTT	5	Configurable
NOT	:NOT:	2	None
One Shot	:O:	1	Configurable
Parameterized Go To Subroutine	PGTS	8 + 1/para.	1-32
Parameterized Go To Subroutine Zero	PGTSZ	8 + 1/para.	1-32
<sup>1</sup> Numbers are for reference only. <sup>2</sup> Varies with controller model. See documentation for specific controller for number supported. <sup>3</sup> Add 1 word for each index parameter.			

Table B-1 RLL Memory Requirements (continued)

<b>Instruction</b>	<b>Mnemonic</b>	<b>Words L-Memory</b>	<b>Reference Number Range</b>
PID Fast Loop	PID	3	129–256
Read Slave Diagnostic	RSD	4	1–112
Return (Conditional or Unconditional)	RTN	2	None
Subroutine	SBR	2	1–255
Queue SF Program	SFPGM	1	See Note 2
Queue SF Subroutine	SFSUB	5 <sup>3</sup>	0–1023 <sup>2</sup>
Bit Shift Register	SHRB	3	Configurable
Word Shift Register	SHRW	4	Configurable
Skip	SKP	1	1–255
Scan Matrix Compare	SMC	34	1–32767 <sup>1</sup>
Square Root	SQRT	3	1–32767 <sup>1</sup>
Table Search For Equal	STFE	6	1–32767 <sup>1</sup>
Table Search For Not Equal	STFN	7	1–32767 <sup>1</sup>
Subtract	SUB	4	1–32767 <sup>1</sup>
Table To Table AND	TAND	6	1–32767 <sup>1</sup>
Start New RLL Task	TASK	4	1–32767 <sup>1</sup>
Text	TEXT	$2 + (NC + NL)/2$ <sup>4</sup>	1–32767 <sup>1</sup>
Time Compare	TCMP	5	1–32767 <sup>1</sup>
Table Complement	TCPL	5	1–32767 <sup>1</sup>
Timer	TMR/ TMRF	2	Configurable
Table To Table OR	TOR	6	1–32767 <sup>1</sup>
Time Set	TSET	3	1–32767 <sup>1</sup>
Table To Word	TTOW	6	1–32767 <sup>1</sup>
Table To Table Exclusive OR	TXOR	6	1–32767 <sup>1</sup>
Up/Down Counter	UDC	3	Configurable
<sup>1</sup> Numbers are for reference only. <sup>2</sup> Varies with controller model. See documentation for specific controller for number supported. <sup>3</sup> With no parameters; words of L-memory varies according to expressions used in each parameter. <sup>4</sup> NC=number of characters of text; NL=number of lines of text.			

## Memory Requirements (continued)

---

Table B-1 RLL Memory Requirements (continued)

<b>Instruction</b>	<b>Mnemonic</b>	<b>Words L-Memory</b>	<b>Reference Number Range</b>
Unlock Memory (575 Only)	UNLCK	3	1-32767 <sup>1</sup>
Word AND	WAND	4	1-32767 <sup>1</sup>
Word OR	WOR	4	1-32767 <sup>1</sup>
Word Rotate	WROT	3	1-32767 <sup>1</sup>
Word To Table	WTOT	6	1-32767 <sup>1</sup>
Word To Table AND	WTTA	7	1-32767 <sup>1</sup>
Word To Table OR	WTTO	7	1-32767 <sup>1</sup>
Word To Table Exclusive OR	WTTXO	7	1-32767 <sup>1</sup>
Word Exclusive OR	WXOR	4	1-32767 <sup>1</sup>
External Subroutine Call	XSUB	8 + 1/par.	1-32767 <sup>1</sup>
<sup>1</sup> Numbers are for reference only.			

# Appendix C

## Controller Performance

---

<b>C.1</b>	<b>Calculating Performance</b> .....	<b>C-2</b>
	Calculating Normal Scan Time .....	C-2
	Calculating the Cyclic RLL Execution Time .....	C-4
	Total Scan Time Including Cyclic RLL .....	C-5
<b>C.2</b>	<b>Tuning the Timeline</b> .....	<b>C-8</b>
	Basic Strategy .....	C-8
	Using Peak Elapsed Time Words .....	C-8
	Using the Status Words .....	C-9
	Concepts to Remember When Tuning Timeline .....	C-10
<b>C.3</b>	<b>RLL Execution Times</b> .....	<b>C-12</b>
<b>C.4</b>	<b>SF Program Statement Execution Times</b> .....	<b>C-13</b>

---

**NOTE:** This section is to be used only as a reference guide for calculating controller performance characteristics. Figures given in tables of execution times may not apply to your controller release. For the current models of the listed controllers, consult the Release Notes included with your controller for up-to-date specifications for your firmware release.

---

## C.1 Calculating Performance

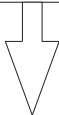
Use the information in this section to estimate a worst-case scan time for your application program. If a feature is not present, no time is added to the scan.

**Calculating Normal Scan Time** To calculate scan time for the normal scan, follow steps 1–7. Remember, the normal scan does not include any programmed cyclic RLL.

**1** **Normal I/O Update**

Note: 1000  $\mu$ s = 1 ms

Local base \_\_\_\_\_ ms  
 Remote bases \_\_\_\_\_ ms  
 DP I/O \_\_\_\_\_ ms  
 Total **1**  ms




Add the I/O update times for the local base, for each of the remote bases, and for the DP slaves.

- **Local Base** 545/555 575  
 For discrete inputs add ..... 2.0  $\mu$ s/point ..... \*  
 For discrete outputs add .... 2.5  $\mu$ s/point ..... \*  
 If any word modules are configured, add overhead ..... 50  $\mu$ s ..... N/A  
 For word inputs/outputs add ..... 3.6  $\mu$ s/word ..... \*
- **Remote Bases** 545/555 575  
 For the first remote base,  
 add ..... 5 ms ..... 5 ms  
 For each additional remote base, add ..... 1 ms ..... 1 ms  
 For word inputs/outputs,  
 add ..... 16  $\mu$ s/word ..... 16  $\mu$ s/word  
 If more than 128 word inputs/outputs on a base —  
 On each base that this is true, for every 128 words add .... 2 ms ..... 2 ms
- **DP I/O** 545/555 575  
 Information not available at time of publication. See Release Notes to obtain this information.

**2** **Non-Cyclic RLL Execution**

RLL Instructions \_\_\_\_\_ ms  
 Total **2**  ms



Add the execution times for the non-cyclic RLL instructions.

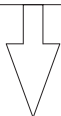
- For RLL instructions 545/555 575  
 (see the execution times in the Controller release notes), add .. \_\_\_\_\_ms ..... \_\_\_\_\_ms

\* Not available at time of publication.



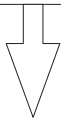
**3 Analog Timeslice**

Loops	_____ ms
Analog alarms	_____ ms
Cyclic SF Pgm	_____ ms
Priority SF Pgm	_____ ms
No-Prty SF Pgm	_____ ms
Normal Comm	
Port	_____ ms
Priority Comm	
Port	_____ ms
RBE	_____ ms
Total ③	<input type="text"/> ms



**4 SF Module Access**

Local base	_____ ms
Remote bases	_____ ms
Total ④	<input type="text"/> ms



Add the values you choose for each portion of the time-slice.

- **Loops:** See loop execution times (Figure C-1) ..... \_\_\_\_ms
- **Analog Alarms:** See analog alarm execution times (Figure C-1) ..... \_\_\_\_ms
- **Cyclic SF Programs:** See statement execution times (Table C-2) ..... \_\_\_\_ms
- **Priority SF Programs:** See statement execution times (Table C-2) ..... \_\_\_\_ms
- **Non-Priority SF Programs:** See statement execution times (Table C-2) ..... \_\_\_\_ms
- **Normal Communication** (processing service requests on the non-priority queue) ..... \_\_\_\_ms
- **Priority Communication** (processing service requests on the priority queue) ..... \_\_\_\_ms
- **RBE** (scan for PCS events) ..... \_\_\_\_ms

Add the SF module access times for each module in the local base and for each module in the remote bases.

- **Local Base** **545/555** **575**  
 SF modules require 0.1–4 ms for update.  
 For each low-activity module, e.g., ASCII, BASIC, DCP, add (typical) ..... 1.0 ms ..... N/A  
  
 For each high-activity module, e.g., NIM, PEERLINK, add (typical) ..... 2.5 ms ..... N/A
- **Remote Bases** **545/555** **575**  
 SF modules require 2–40 ms for update.  
 If any SF modules are installed, add overhead ..... 2 ms ..... 2 ms  
  
 For each low-activity module, e.g., ASCII, BASIC, DCP, add (typical) ..... 12 ms ..... 12 ms  
  
 For each high-activity module, e.g., NIM, PEERLINK, add (typical) ..... 25 ms ..... 25 ms

## Calculating Performance (continued)

**Communications Port Overhead**

⑤

Local ports	_____ ms
Remote ports	_____ ms
Total ⑤	<input type="text"/> ms

Add the overhead times for the local communication ports and for the remote communication ports.

- **Local Ports** For each 545/555/575 communication port used during normal operation add ..... 1 ms
- **Remote Ports** For each RBC communication port used during normal operation add ..... 2 ms

**CPU Overhead**

⑥

Overhead	<u>2</u> ms
Total ⑥	<input type="text"/> ms

Add the CPU overhead.

- For these controller models **545/555** **575**  
add ..... 2 ms ..... 2 ms

**Normal Scan Time**

⑦

Normal scan time	
①+②+③	
+④+⑤+⑥	<input type="text"/> ms

Add the values 1–6 for the normal scan time. .... \_\_\_\_ ms

This step completes the calculation for the normal controller scan. If you have programmed cyclic RLL, continue with steps 8–10.

### Calculating the Cyclic RLL Execution Time

To determine the execution time for the cyclic RLL portion of an application program, do the calculations in step 8.

**Cyclic RLL Execution**

⑧

Cyclic RLL execution	
Overhead	<u>0.16</u> ms
RLL Instructions	_____ ms
Total ⑧	<input type="text"/> ms

Add the overhead and execution times for the cyclic RLL boolean and box instructions.

- For these controller models **545/555** **575**  
add overhead of ..... 0.16 ms ..... 0.16 ms

For RLL instructions (see the execution times in the Controller release notes), add .. \_\_\_\_ms ..... \_\_\_\_ms

**Total Scan Time Including Cyclic RLL**

To determine the total scan time for an application program that has cyclic RLL, do the calculations in steps 9–10.

**Cyclic RLL Execution Frequency**

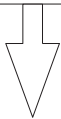
Calculate a preliminary number of times (frequency) that the cyclic RLL executes during the normal scan.

9

Frequency = 7 ÷ T

T = Cyclic RLL cycle time

Value 9  Times



**Total Scan Time**

The determination of the total scan time is an iterative process. After you obtain a value (Value 10) for the total scan time, substitute it for Value 7 in the cyclic RLL execution frequency calculation in Step 9, and then do step 10 again. Repeat this until the execution frequency for the cyclic RLL (Value 9) no longer changes.

10

Scan time total =

[ 9 × 8 ] + 7

Value 10  ms



Repeat steps 9–10, substituting

10 for 7 in step 9

until 9 no longer changes.

The calculation in step 10 is based on these values.

- Cyclic RLL frequency of execution ..... Value 9
- Cyclic RLL execution time ..... Value 8

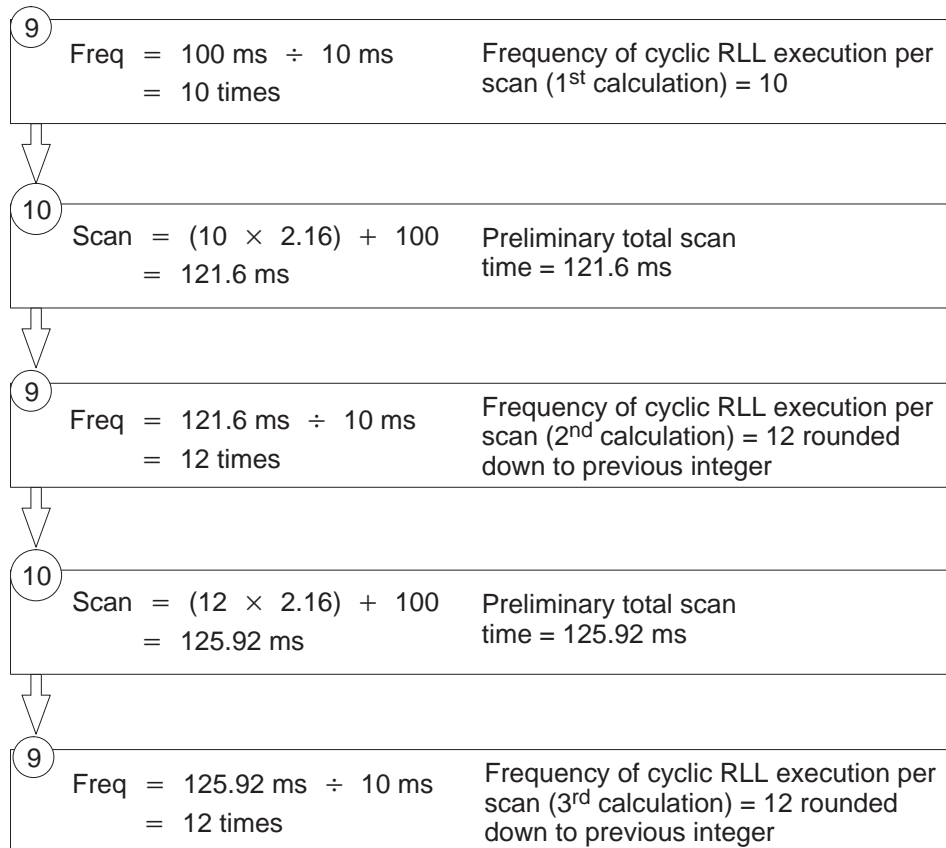
An example of the iterative process is shown in a sample calculation on page C-6.

## Calculating Performance (continued)

---

Consider this example, that has the following assumptions.

- Cyclic RLL cycle time is 10 ms
- Normal scan = 100 ms
- Cyclic RLL execution = 2.16 ms



The third iteration shows that the total scan time is approximately 126 ms, and the cyclic RLL executes 12 times per scan.

Figure C-1 shows the loop/analog execution time for the 545/575.

Loop Execution	
No alarms enabled .....	1.470 ms
All Alarms monitored .....	1.640 ms
All Alarms monitored .....	2.110 ms
One ramp/soak step added	
All Alarms monitored .....	2.110 ms
One ramp/soak step added	
20% Offset added	
All Alarms monitored .....	2.200 ms
One ramp/soak step added	
20% Offset added	
Square root of PV added	
All Alarms monitored .....	2.690 ms
One ramp/soak step added	
20% Offset added,	
Square root of PV	
Minimal Special Function Program added	
Analog Alarm Execution	
High, High-High, Low, Low-Low Alarms enabled .....	0.724 ms
All other options disabled	
High, High-High, Low, Low-Low Alarms enabled .....	0.740 ms
Deviation Alarms enabled	
No V-Flag address enabled	
No PV address enabled	
High, High-High, Low, Low-Low Alarms enabled .....	0.858 ms
Deviation Alarms enabled	
No V-Flag address enabled	
PV address enabled	
High, High-High, Low, Low-Low Alarms enabled .....	0.842 ms
No Deviation Alarms enabled	
No V-Flag address enabled	
PV address enabled	
High, High-High, Low, Low-Low Alarms enabled .....	0.922 ms
No Deviation Alarms enabled	
V-Flag address enabled	
PV address enabled	
High, High-High, Low, Low-Low Alarms enabled .....	1.250 ms
Deviation Alarms enabled	
V-Flag address enabled	
PV address enabled	
Remote SP enabled	

**Figure C-1 Loop/Analog Alarm Execution Time for the 545/575\***

\* Times for the 555 are one-half of the times specified in Figure C-1.

## C.2 Tuning the Timeline

---

### Basic Strategy

For most applications, you do not need to adjust the default timeslices for the timeline. After you have made your best predictions for analog process execution times (loops, analog alarms, SF programs, etc.), you may still want to make adjustments in the timeline, based on empirical data. You have the option of fine-tuning the sub-slices of the analog timeslice to ensure that these analog processes are executed as quickly as possible and do not overrun. The sections that follow describe some suggestions about how to approach the fine-tuning.

When you set the timeslices, you are also affecting the length of the overall controller scan. Shorter analog timeslices reduce the overall scan, and results in a faster I/O update. Typically, you want to reduce the analog portion of the scan as much as possible to reduce the overall scan time. However, do not allow too little time for the analog portion. Loops and analog alarms can begin to overrun, and the time for SF programs to execute after scheduling can be longer.

### Using Peak Elapsed Time Words

The controller stores the peak elapsed time for a process to execute in a PET variable. The peak elapsed time is the time from when a process is scheduled (placed in the queue) until the process completes execution. The controller updates these words each time the process is scheduled and executed.

- LPET<sub>n</sub> for loops (n = 1–64)
- APET<sub>n</sub> for analog alarms (n = 1–128)
- PPET<sub>n</sub> for SF Programs (n = 1–1023)
- SPET<sub>n</sub> for SF Subroutines (n = 1–1023)

You can determine if the loops, analog alarms, or cyclic SF programs are coming close to overrunning. If the value in the APET, the LPET, or the PPET approaches the sample time, you can increase the timeslice for the analog alarms or for the loops. Alternatively, you can decrease the other timeslices. This reduces the overall scan, allowing the analog alarms or loops to run more often in a given time. The time needed for the discrete portion of the scan limits how much you can reduce the overall scan.

If the PPET indicates that an SF program is taking significantly more time for execution than your calculation based on times in Table C-2, you can increase the timeslice appropriately. If the SF program is critical, move that SF program to the Priority queue.

PPET is only valid for an SF program queued from RLL (priority, non-priority, or cyclic SF programs). The time for executing an SF program called from a loop or analog alarm is included in LPET or APET, respectively.

SPET is only valid for an SF subroutine queued from RLL. The time for executing an SF subroutine called from an SF program is included in the PPET for the SF program. The time for executing an SF subroutine called from an SF program called from a loop or analog alarm is included in the appropriate LPET or APET.

**Using the Status Words**

Check the status of the following bits in Status Word 162 (STW162) to see if these analog processes are overrunning.

- Bit 3 Loops are overrunning.
- Bit 4 Analog Alarms are overrunning.
- Bit 5 Cyclic SF programs are overrunning.
- Bit 6 Non-priority SF program queue is full.
- Bit 7 Priority SF program queue is full. All priority and non-priority SF programs will be executed in turn.
- Bit 8 Cyclic SF program queue is full.

Check bit 14 in Status Word 1 (STW01) to see if the overall scan is overrunning. When the bit is true (= 1), the scan time required to execute the entire program is greater than the designated scan time.

The instantaneous discrete execution time (the time to execute the discrete portion of the scan) is reported in Status Word 192 (STW192). The instantaneous total scan time is reported in Status Word 10 (STW10).

Table C-1 summarizes the performance and overrun indicators.

**Table C-1 Performance and Overrun Indicators**

<b>Performance Overrun Indication</b>	<b>Status Word/AUX Function</b>
Discrete scan overrun indicator	STW01 and AUX 29
Previous discrete scan time	STW192 and AUX 19
Previous total scan time	STW10 and AUX 19
Peak discrete and total scan time	AUX 19
Cyclic process overrun indicators	STW162
Individual cyclic process overrun indicators	V-Flags and T6
SF queue full	STW162
Process peak elapsed time	LPET, APET, PPET, SPET
Scan watchdog	AUX14

## Tuning the Timeline (continued)

---

### Concepts to Remember When Tuning Timeline

**SF modules:** When you determine the base location for SF modules, consider the impact on the controller scan. Update time for an SF module is an order of magnitude faster when you install the module in the local base, versus a remote base, resulting in less extension of the controller scan.

If all SF modules cannot be installed in the local base, consider placing low-activity SF modules, such as the ASCII, BASIC, or DCP modules, in a remote base. Locate high-activity modules, such as the NIM or PEERLINK, in the local base.

---

**NOTE:** SF modules cannot be placed in the 575 local base.

---

**SF program execution time:** Your calculation of an SF program execution time based on the statement times (Table C-2) is the actual execution time required for the controller to run the SF program. The time from when the SF program program is placed in the queue until the point at which execution begins can vary. This wait depends upon the number of SF programs scheduled, how long they take for execution, how long the timeslice is, and the priority of other analog tasks scheduled for processing.

**Priority/non-priority SF program queues:** The two SF program queues provide a means of separating critical SF programs, (needing to run quickly) from less important SF programs. Keep the number of priority SF programs as small as possible, and if it is not essential that an SF program be executed very rapidly, assign it to the other queue.

You can increase the timeslice for the Priority SF programs to ensure that queued programs are executed as quickly as necessary.

**Cyclic SF program queue:** The controller allows you to queue up to 32 cyclic SF programs at once. If you create more than 32, only the first 32 that are queued are executed.

**Do not overload the controller:** Remember that the controller has a finite set of resources. Though the controller may support up to 128 loops, you cannot run them all at 0.1 second intervals without adversely affecting the execution of the other analog processes. You cannot run all allowed analog alarms at 0.1 second intervals for the same reason.



---

**RLL versus SF math:** The controller processes RLL math much faster than SF program math. When possible, use RLL for integer mathematical calculations for faster response time.

**Timeslice resolution:** Timeslices have a resolution of 1 ms. When you program a 4 ms timeslice, that timeslice is executed for four 1 ms clock pulses. The time from the beginning of the timeslice to the first pulse can vary from zero time to a full 1 ms pulse. Therefore, the actual time in a 4 ms timeslice is greater than 3 but less than or equal to 4 ms.

Each transition between timeslices takes approximately 200  $\mu$ s of overhead. This overhead is included in the time allotted to each timeslice and does not have an additional impact on the overall scan.

### C.3 RLL Execution Times

---

Execution times for RLL instructions are listed in release notes for your controller.

To calculate RLL program execution time, multiply the instruction execution time by the instruction frequency of occurrence for all instructions in your ladder logic program. Then sum these products. For example, if your program contains four ADD instructions, four contacts, and four coils with execution times of 10.30  $\mu\text{s}$ , 0.33  $\mu\text{s}$ , and 0.40  $\mu\text{s}$ , respectively, the program execution time is calculated as follows.

$$\begin{array}{rcl} 4 \text{ ADDs} & \times 10.30 \mu\text{s} & = 41.20 \mu\text{s} \\ 4 \text{ Contacts} & \times 0.33 \mu\text{s} & = 1.32 \mu\text{s} \\ 4 \text{ Coils} & \times 0.40 \mu\text{s} & = \underline{1.60 \mu\text{s}} \\ \text{RLL execution time} & & = 44.12 \mu\text{s} \end{array}$$

## C.4 SF Program Statement Execution Times

---

Execution times for the SF statements are listed in Table C-2 for the 545 and 575 controllers. All times are in microseconds.

---

**NOTE:** For the 555, execution times are 1/2 of the stated times. The execution times for the 545–1103 are approximately 1.2 times the stated times.

---

To calculate SF program execution time, multiply the statement execution time by the statement frequency of occurrence for all statements in your SF program. Then sum these products.

For example, if your program contains 1 SSR (table length = 3), 2 BINBCDs, 3 COMMENTS, then the program execution time for a 545 or 575 controller is calculated as follows.

$$\begin{array}{r r r r r}
 1 \text{ SSR} & \times & 250 + (140 \times 3) \text{ } \mu\text{s} & = & 670.0 \text{ } \mu\text{s} \\
 2 \text{ BINBCDs} & \times & 365 \text{ } \mu\text{s} & = & 730.0 \text{ } \mu\text{s} \\
 3 \text{ COMMENTS} & \times & 20.6 \text{ } \mu\text{s} & = & 61.8 \text{ } \mu\text{s} \\
 & & & & \underline{\underline{\hspace{1.5cm}}}
 \end{array}$$

$$\text{SFPGM Execution Time} = 1,461.8 \text{ } \mu\text{s}$$

---

**NOTE:** The calculation based on these statement execution times is the actual execution time required for the controller to run the SF program. The time from when the SF program is placed in the queue until the point at which execution begins can vary. This wait depends upon the number of SF programs scheduled, how long they take for execution, and the priority of other analog tasks scheduled for processing.

---

**Table C-2 SF Statement Execution Times for the 545/575**

SF Statement	Notes/Assumptions	Execution Time
Arrays	Accessing V102 using V100(3) Accessing V102 using V100(V1) where V1 = 3	add 50 $\mu$ sec to variable access add 150 $\mu$ sec to variable access
BCDBIN	input=V4, output=V5	297 $\mu$ sec
BINBCD	input=V5, output=V4	365 $\mu$ sec
CALL		$\approx 81 \text{ } \mu\text{sec} + (60 \text{ } \mu\text{sec} \times \# \text{ of parameters})$

## SF Program Statement Execution Times (continued)

**Table C-2 SF Statement Execution Times for the 545/575 (continued)**

<b>SF Statement</b>	<b>Notes/Assumptions</b>	<b>Execution Time</b>
CDT	input=V1, output=V2 in_table=V10, out_table=V20 length=x	best case: $\approx 689 \mu\text{sec}$ worst case: $\approx 689 \mu\text{sec} + (120 \mu\text{sec} \times (\text{length} - 1))$
COMMENT		20.6 $\mu\text{sec}$
Expressions	relational operators, e.g., >, >=, =, etc.	$\approx 70 \mu\text{sec}$
EXIT		41.0 $\mu\text{sec}$
FTSR-IN	input=V1, length=4, register start=V100 status=C50	625 $\mu\text{sec}$
FTSR-OUT	output=V3, length=4, register start=V100, status=C50	653 $\mu\text{sec}$
GOTO	GOTO Label 1	38.4 $\mu\text{sec} + (5.3 \mu\text{sec} \times \# \text{ of intervening statements between GOTO and LABEL})$
IF-THEN-ELSE	IF (expression) and the expression is true IF (expression) and the expression is false	95 $\mu\text{sec} + \text{time to evaluate expression}$ 95 $\mu\text{sec} + \text{time to evaluate expression} +$ $\approx 7 \mu\text{sec} \times \# \text{ of statements prior to ENDIF}$ or ELSE
ELSE or ENDIF		$\approx 20.5 \mu\text{sec}$
IMATH	Assume integer variables, when used	175 $\mu\text{sec}$ (assignment, e.g. V200 := 10) + 20 $\mu\text{sec}$ (per each operator, e.g. +, -, ) + 5 $\mu\text{sec}$ (per each constant operand + 100 $\mu\text{sec}$ (per each variable operand, e.g., V100)
LABEL	Label 1	$\approx 22 \mu\text{sec}$
LEAD/LAG		$\approx 1440 \mu\text{sec}$
MATH	Assume real variables	182 $\mu\text{sec}$ (assignment, e.g. V200 := 10.0) + [60 $\mu\text{sec}$ (for most operators, such as +, -, exp(**) $\approx 500 \mu\text{sec}$ )] + 7 $\mu\text{sec}$ (per each constant operand) + 100 $\mu\text{sec}$ (per each variable operand, e.g., V100)  Notes: 1) Intrinsic functions, such as ABS, FRAC, etc., average 315 $\mu\text{sec}$ of time for execution (max. $\approx 470 \mu\text{sec}$ for LOG. 2) Integers are converted to reals before computation is done. Add 25 $\mu\text{sec}$ for each integer $\rightarrow$ real, real $\rightarrow$ integer conversion that must occur.

**Table C-2 SF Statement Execution Times for the 545/575 (continued)**

SF Statement	Notes/Assumptions	Execution Time	
PACK		$\approx 110 \mu\text{sec} + \Sigma \text{ block time}$	<u>Discrete block time</u> $\approx 179 \mu\text{sec} + ((\# \text{points}-1) \times 87 \mu\text{sec}) + ((\# \text{points}-1) / 16) \times 220 \mu\text{sec}$  <u>Integer block time</u> $\approx 276 \mu\text{sec} + ((\# \text{points}-1) \times 170 \mu\text{sec})$  <u>Real block time</u> $\approx 413 \mu\text{sec} + ((\# \text{points}-1) \times 259 \mu\text{sec})$
PACK AA			$\approx 225 \mu\text{sec} + (\# \text{ of integer parameters} \times 152 \mu\text{sec}) + (\# \text{ of real parameters} \times 300 \mu\text{sec})$
PACKLOOP			$\approx 228 \mu\text{sec} + (\# \text{ of integer parameters} \times 374 \mu\text{sec}) + (\# \text{ of real parameters} \times 325 \mu\text{sec})$ for PACK_TO or $(\# \text{ of real parameters} \times 500 \mu\text{sec})$ for PACK_FROM
PRINT	Time to start print operation; the actual print time varies with the length of the print job, port baud rate, etc.	$\approx 165 \mu\text{sec}$	
RETURN		$\approx 60 \mu\text{sec}$	
SCALE	input=V1, output=V2 low=0, high=100, 20%=no, bipolar=no	$\approx 579 \mu\text{sec}$	
SDT	input table=V10, output=V1, pointer=V2, restart=C50, length=x	$\approx 604 \mu\text{sec}$	
SSR	using tablestart=V10, status bit=C10	$\approx 250 \mu\text{sec} + (140 \mu\text{sec} \times \text{table length})$	
UNSCALE	input=V2, output=V1, low=0, high=100 20%=no, bipolar=no	$\approx 582 \mu\text{sec}$	

# Loop and Analog Alarm Flag Formats

---

D.1	Loop Flags .....	D-2
D.2	Analog Alarm Flags .....	D-4

## D.1 Loop Flags

---

Table D-1, Table D-2, Table D-3, and Table D-4 give the formats for the C-Flags and V-Flags used by the 545, 555, and 575 controllers.

Table D-1 Loop V-Flags (LVF)

Bit	Loop Function
1	1 = Go to manual mode
2	1 = Go to auto mode
3	1 = Go to cascade mode
4 and 5	4 5 0 0 Loop is in manual mode 1 0 Loop is in auto mode 0 1 Loop is in cascade mode
6	0 = Error is positive 1 = Error is negative
7	1 = PV is in high-high alarm
8	1 = PV is in high alarm
9	1 = PV is in low alarm
10	1 = PV is in low-low alarm
11	1 = PV is in yellow deviation alarm
12	1 = PV is in orange deviation alarm
13	1 = PV is in rate-of-change alarm
14	1 = Broken transmitter alarm
16	unused

Table D-2 Loop C-Flags (LCFH and LCFL)

Variable	Word Bit	Flag Bit	Loop Function
LCFH	1	1	0 = PV scale 0% offset 1 = PV scale 20% offset—only valid if PV is unipolar. See bit 21.
	2	2	1 = Take square root of PV
	3	3	1 = Monitor high/low alarms
	4	4	1 = Monitor high-high/low-low alarms
	5	5	1 = Monitor yellow/orange deviation alarm
	6	6	1 = Monitor rate-of-change alarm
	7	7	1 = Monitor broken transmitter alarm
	8	8	PID algorithm type 0 = Position algorithm 1 = Velocity algorithm
	9	9	0 = Direct acting 1 = Reverse acting
	10	10	1 = Control based on error squared
	11	11	1 = Control based on error deadband
	12	12	1 = Auto-mode lock
	13	13	1 = Cascade-mode lock
	14	14	1 = Setpoint lock
	15	15	0 = Output scale 0% offset 1 = Output scale 20% offset—only valid if output is unipolar. See bit 20.
	LCFL	16	16 17 0 1 No special function 1 0 Special function on the process variable 0 1 Special function on the setpoint 1 1 Special function on the output
1		17	
2		18	1 = Freeze bias when output is out-of-range
3		19	1 = Ramp/Soak on the setpoint
4		20	0 = Output is unipolar 1 = Output is bipolar
5		21	0 = PV is unipolar 1 = PV is bipolar
6		22	1 = Perform derivative gain limiting
7–16	23–32	Contains SF program number (if an SF program is scheduled to be called)	



## D.2 Analog Alarm Flags

Table D-3 Analog Alarm V-Flags (AVF)

Bit	Analog Alarm Function
1	1 = Enable alarm
2	1 = Disable alarm
3	1 = PV is in high-high alarm
4	1 = PV is in high alarm
5	1 = PV is in low alarm
6	1 = PV is in low-low alarm
7	1 = PV is in yellow deviation alarm
8	1 = PV is in orange deviation alarm
9	1 = PV is in rate of change alarm
10	1 = Broken transmitter alarm
11	1 = Analog alarm is overrunning
12	1 = Alarm is enabled *
13-16	Unused
* If a word is selected for the analog alarm V-Flags, bit 12 is written. If a C or Y is selected, bit 12 is not used.	

Table D-4 Analog Alarm C-Flags (ACFH and ACFL)

Variable	Word Bit	Flag Bit	Analog Alarm Function
ACFH	1	1	0 = PV scale 0% offset 1 = PV scale 20% offset
	2	2	1 = Take square root of PV
	3	3	1 = Monitor high/low alarms
	4	4	1 = Monitor high-high/low-low alarms
	5	5	1 = Monitor Deviation alarm
	6	6	1 = Monitor Rate-of-change alarm
	7	7	1 = Monitor Broken Transmitter Alarm
	8	8	0 = Local Setpoint 1 = Remote Setpoint
	9-16	9-16	Unused
ACFL	1-4	17-20	Unused
	5	21	0 = Process Variable is unipolar 1 = Process Variable is bipolar
	6	22	Unused
	7-16	23-32	Contains SF program number (if an SF program is scheduled to be called)

# Appendix E

## Selected Application Examples

---

E.1	Using the SHRB .....	E-2
E.2	Using the SHRW .....	E-4
E.3	Using the TMR .....	E-6
E.4	Using the BITP .....	E-10
E.5	Using the DRUM .....	E-11
E.6	Using the EDRUM .....	E-13
E.7	Using the MIRW .....	E-17
E.8	Using the MWIR .....	E-20
E.9	Using the MWTT .....	E-24
E.10	Using the MWFT .....	E-26
E.11	Using the WXOR .....	E-28
E.12	Using the CBD .....	E-30
E.13	Using the CDB .....	E-32
E.14	Using the One Shot .....	E-33
E.15	Using the DCAT .....	E-34
E.16	Using Status Words .....	E-37

## E.1 Using the SHRB

### SHRB Application Example

An inspector tests a partially assembled piece and pushes a reject button when a defective piece is found. As the piece moves through the last 20 stations of final assembly, a reject lamp must light in each assembly station with the defective piece. Figure E-1 illustrates this application.

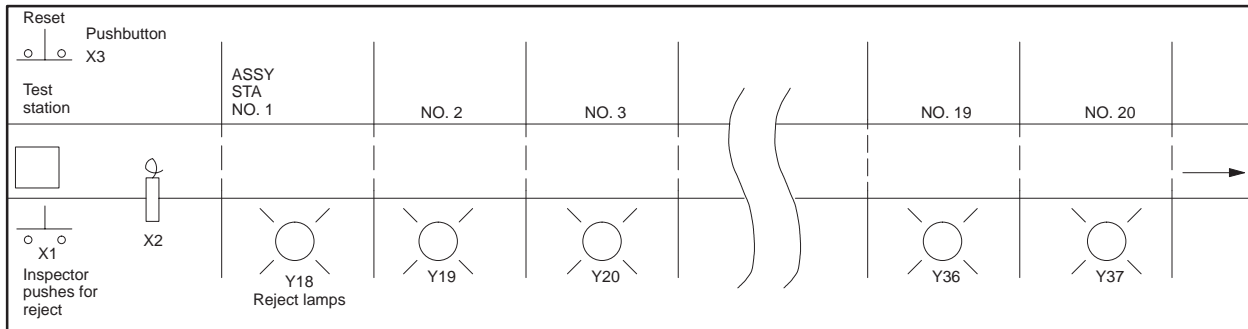


Figure E-1 SHRB Application Example

The following solution was devised.

- Pushbutton X1 is the reject button.
- Pushbutton X3 is the reset button.
- Outputs Y18 through Y37 control the status of assembly station reject lamps.
- Limit switch X2 cycles each time a piece is indexed.
- SHRB 1 shifts the status of the piece (lights the reject lamp) as indexed through the last 20 stations of final assembly.

### Explanation

The RLL solution is shown in Figure E-2.

- When the reject pushbutton X1 is pressed, coil C1 is latched on through contact C1.
- Coil C2 shows the status of Y37.
- When the piece is indexed through limit switch X2, the status of coil C1 is shifted into Y18.

- In Figure E-2, a shift register provides a 20-bit register for controlling the SHRB application. The 20-bit shift register, SHRB1 (shown in Figure E-3), controls the REJECT lamps at the 20 assembly stations.
- The reset pushbutton resets the 20-bit shift register to zero.
- In this application, the part must be inspected and, if found defective, the reject button must be pressed before limit switch X2 is cycled off-to-on by the passing box. This application assumes that X2 is off until a box strikes it.

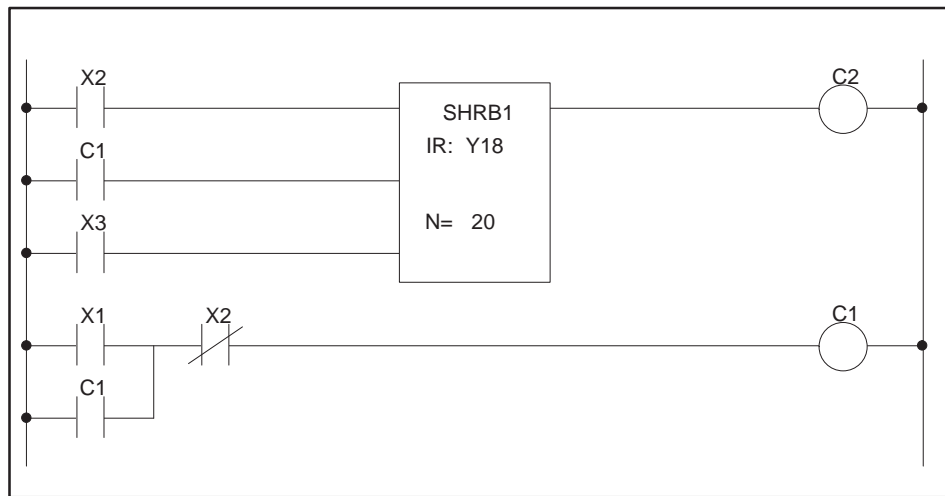


Figure E-2 RLL for SHRB Application Example

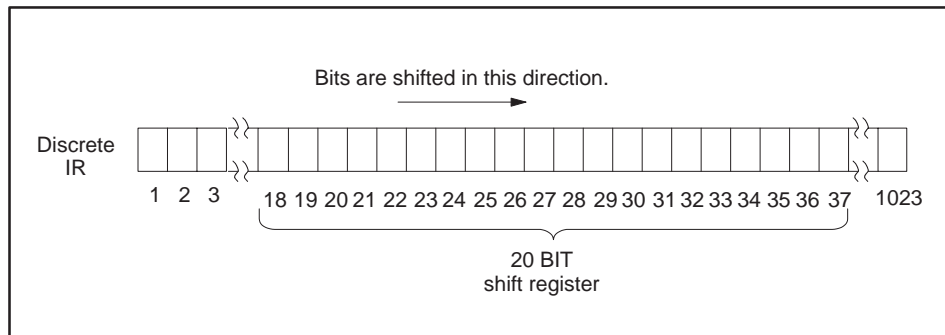


Figure E-3 20-Bit Shift Register in Discrete Image Register

## E.2 Using the SHRW

### SHRW Application Example

A paint line is to carry multiple parts (identified by part numbers), each of which must be painted a different color based on its part number. The part number is read by a photocell reader, and a limit switch sets up a load robot to load the part onto a carrier conveyor. The carrier conveyor is indexed through 12 stations, and the part number must accompany the part through each work station to actuate the desired functions. The part is removed from the carrier conveyor by an unload robot in station 12, and the main conveyor moves the part to the packing area. Figure E-4 illustrates this application.

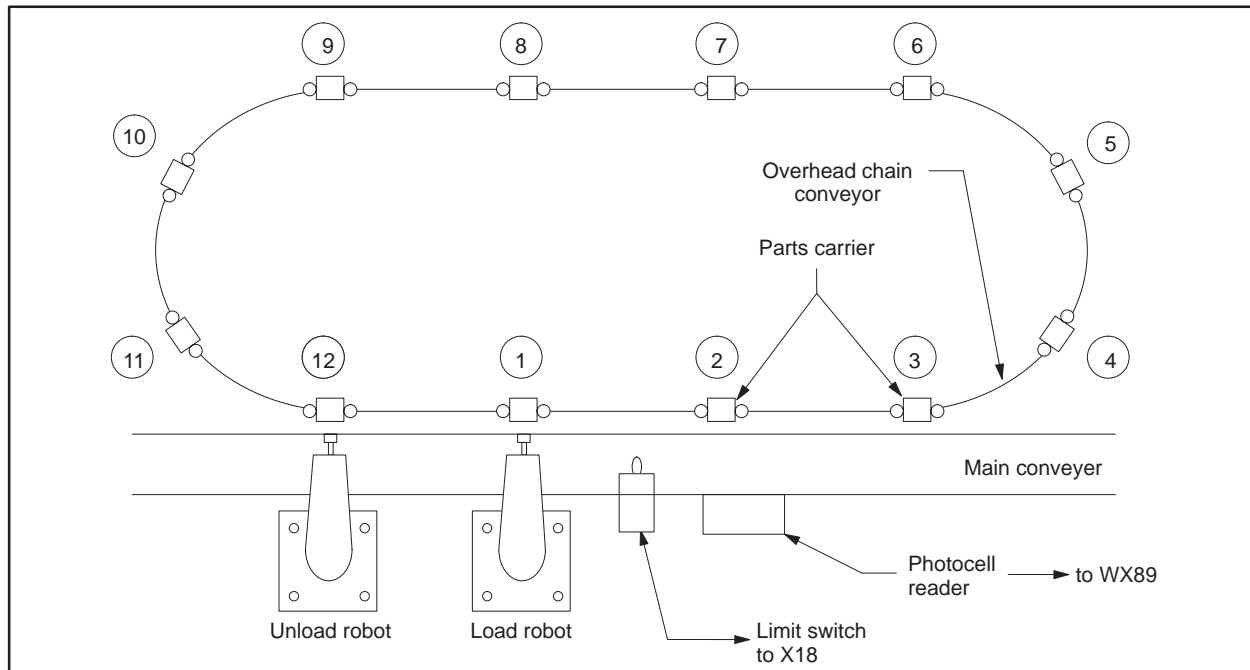


Figure E-4 SHRW Application Example

The following solution was devised.

- The photocell reader is connected to input #1 of a Word Input Module located in Slot 4 of Base 1 (WX89).
- A limit switch is connected to input #2 of a Discrete Input Module located in Slot 3 of Base 0 (X18).
- An SHRW shifts the number with the part as it is indexed through work stations.
- A CMP checks the part number in each station against a mask
- X11 is connected to a reset pushbutton.

Explanation

The RLL solution is shown in Figure E-5.

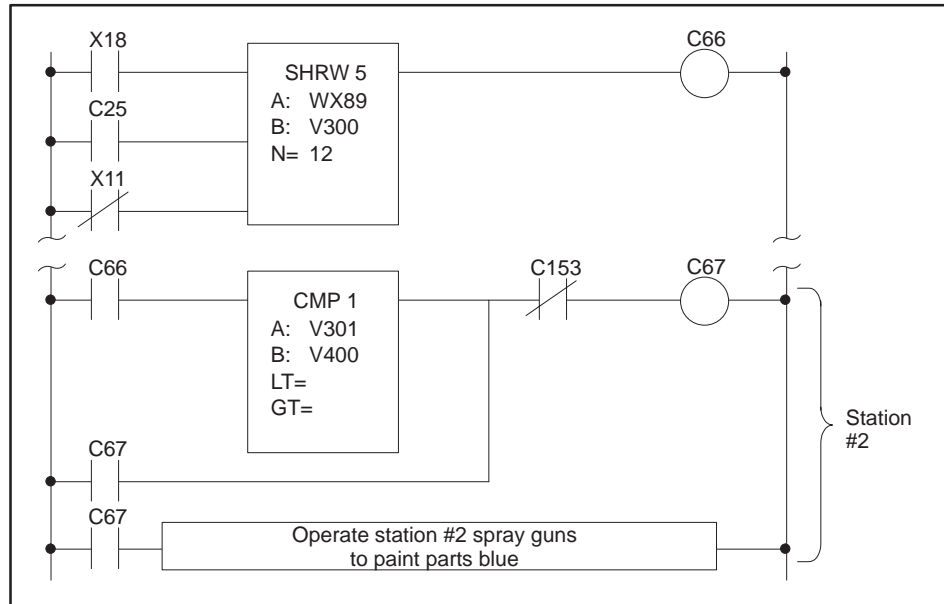


Figure E-5 RLL for SHRW Application Example

- The photocell reader (WX89) reads the number of a part moving along the main conveyor.
- Limit switch X18 turns on, allowing SHRW 5 to shift the part number (WX89) to V300, setting up the load robot to load the part onto the carrier conveyor at station 1. (The network to control the load robot is not shown.) C66 is energized for one scan.
- When the second part moves to limit switch X18, the sequence described above is repeated, the part number that was in memory location V300 is shifted to V301, and the part is indexed to station 2. CMP1 compares the station 2 mask (V400) with the part number in V301; coil C67 turns on if there is a compare (latched through contact C67) and initiates the network to paint the part blue.
- C153 resets the station 2 compare network when the work cycle is complete.
- A similar compare network is used to initiate the work cycle in the remaining stations, if required for that particular part number.

## E.3 Using the TMR

### TMR Application Example #1

A piece is to be indexed automatically into a drilling station. The piece is clamped and drilled in the station before being indexed out on a conveyor. If the automatic index and drilling cycle stops, a fault detection circuit must be actuated. Figure E-6 illustrates this application.

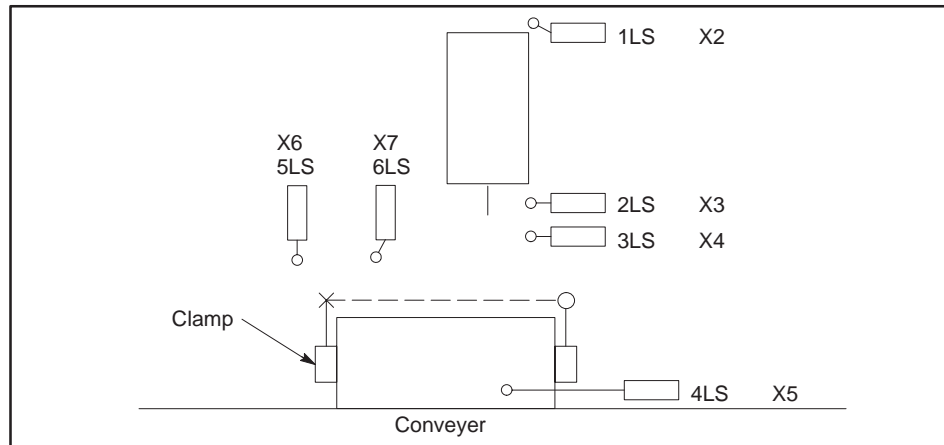


Figure E-6 TMR Application Example

The following solution was devised.

- Input X1 (1SSW) = Auto-Manual selector switch
- Input X2 (1LS) = drill in home position
- Input X3 (2LS) = drill advanced to piece
- Input X4 (3LS) = maximum drill depth reached
- Input X5 (4LS) = piece in position
- Input X6 (5LS) = piece clamped
- Input X7 (6LS) = piece unclamped

### Explanation #1

The RLL solution is shown in Figure E-7. Timers are used for dwell and cycle fault.

- When the Auto-Manual switch is in the auto mode (contact X1 is closed), the piece is unclamped (X7 closed) and the drill is in the home position (X2 closed). Coil Y9 turns on, allowing the conveyor to index a new piece into the drilling station.

- When the piece is in position (X5 closed), output Y10 operates a solenoid to clamp the piece.
- When the piece is clamped (X6 closed, X7 open), the index conveyer turns off (Y9 turns off), TMR2 starts timing, and output Y11 energizes a motor or solenoid to move the drill to the piece.
- When the drill reaches the piece (X3 closed, X2 open), drilling is started by output Y12.
- When the maximum drilling depth is reached (X4 on), the drill stops moving and the dwell timer TMR 1 starts timing.
- When TMR 1 times out, C1 turns on and output Y13 energizes a motor or solenoid to move the drill back to home position.
- TMR 2 times out if the drill machine fails to complete the index drill cycle.

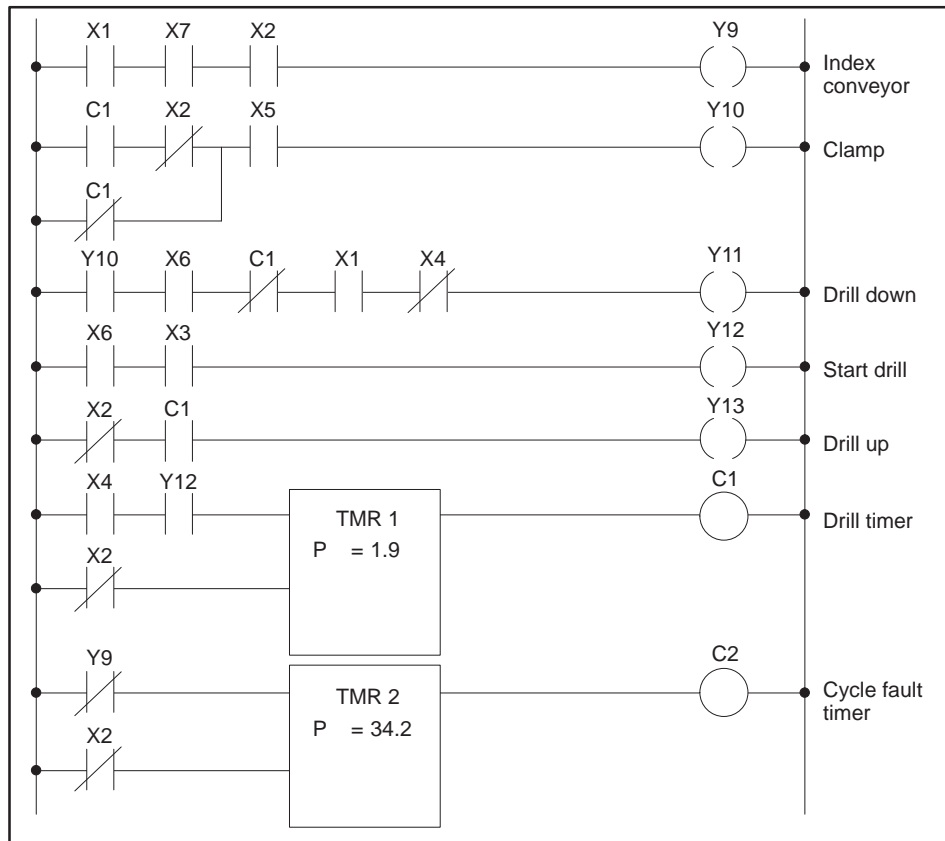


Figure E-7 RLL for TMR Application Example #1



## Using the TMR (continued)

### TMR Application Example #2

Figure E-8 is a timing diagram for the timer logic shown in Figure E-9.

- X24 is the enable and the reset switch.
- Y9 is the On Delay output.
- Y11 is a timed pulse that operates when Y9 is closed and X24 is open.

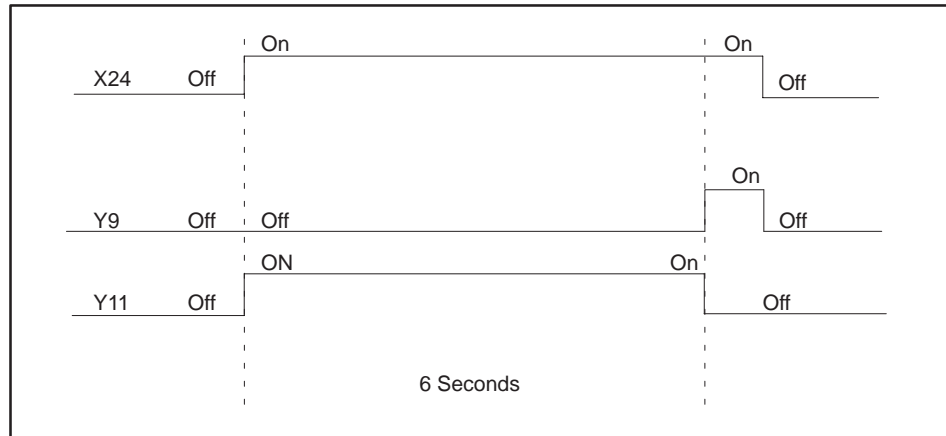


Figure E-8 Timing Diagram for TMR Application Example #2

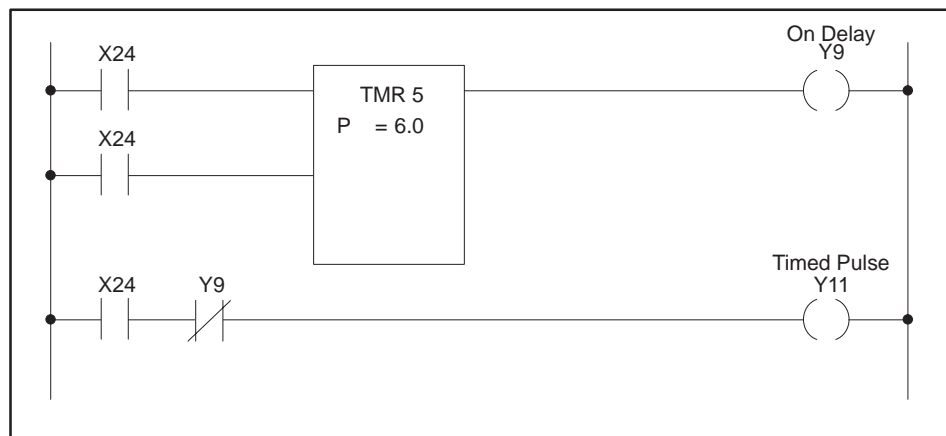


Figure E-9 RLL for TMR Application Example #2

Application #3

Figure E-10 is a timing diagram for the timer logic shown in Figure E-11.

- X24 is the enable and the reset switch.
- Y10 is the Off Delay output.
- Y11 is a timed pulse that operates when Y10 is closed and X24 is open.

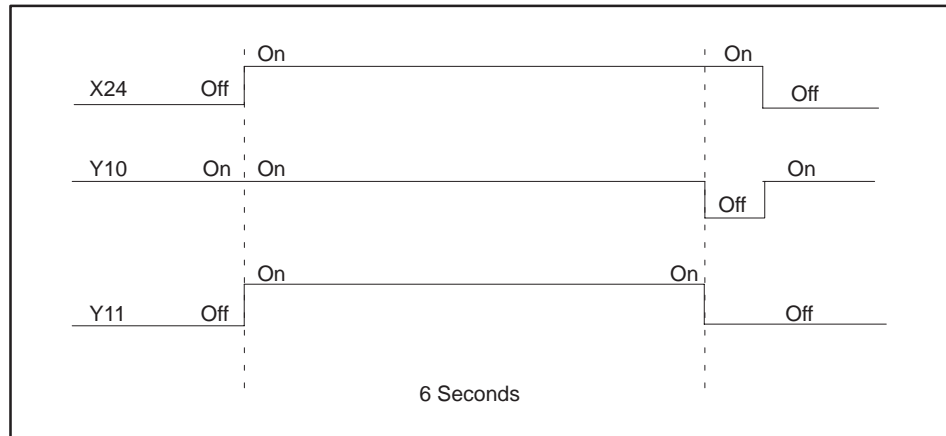


Figure E-10 Timing Diagram for TMR Application Example #3

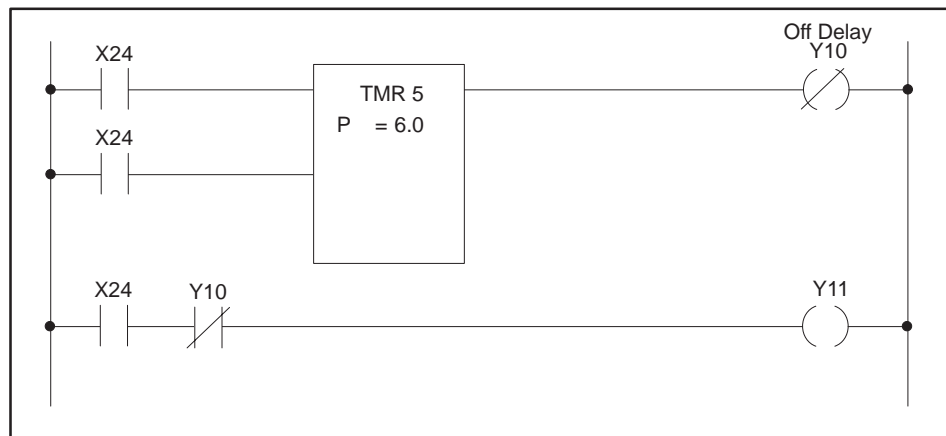


Figure E-11 RLL for TMR Application Example #3

## E.4 Using the BITP

### BITP Application Example

A panel indicator lamp is to warn of a low battery in the controller.

The following procedure was devised.

- X1 has power flow when the system is started.
- BITP1 checks bit 15 of STW1 for a 1 or a 0.
- X2 is a reset pushbutton.
- Output Y10 turns on a lamp.

### Explanation

Figure E-12 shows the RLL solution.

- When the system is started, contact X1 has power flow, enabling the BITP 1 instruction. Each scan, the BITP 1 checks the status of bit 15 in STW1.
- If bit 15 of STW1 is 1, coil Y10 energizes, lighting an indicator lamp.
- The lamp remains on until the controller battery is replaced and the reset button (X2) is pressed.

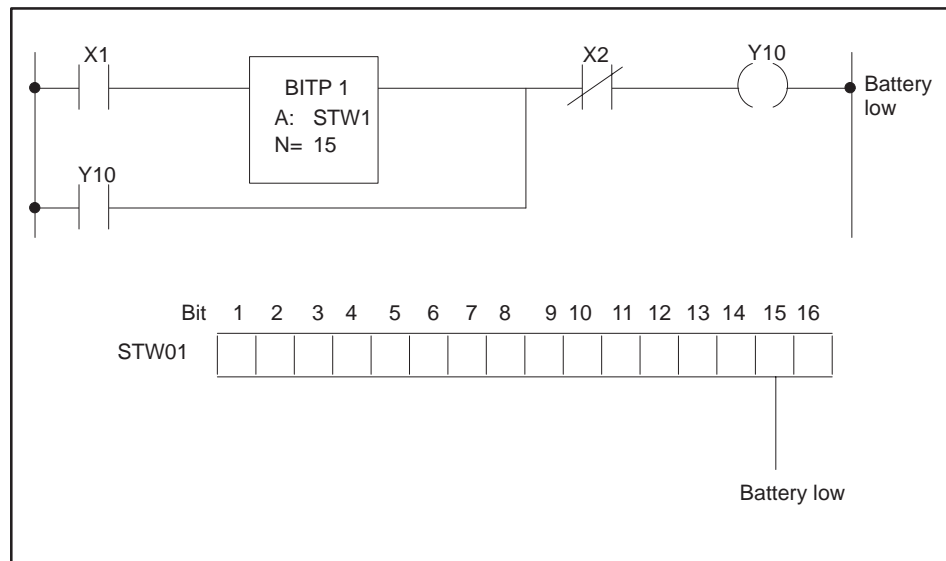


Figure E-12 RLL for BITP Application Example

## E.5 Using the DRUM

---

**DRUM Application Example** A time-based drum with two programmed modes controls the operation of a machine. Mode 1: the drum indexes through the programmed steps in the normal sequence. Mode 2: the starting drum step is increased for one drum cycle, as controlled by discrete inputs. The solution is listed below, and the RLL is shown in Figure E-13.

- Input contact X9 starts the drum.
- The drum controls output coils Y2 through Y8.
- Input contact X11 transfers a step value from inputs X12 through X16, to force the drum to a specified step.

**Explanation** When the controller is in RUN mode, DRUM 1 is in PRESET step 2; and Y2, Y3, Y7, and Y8 remain on until X9 is energized.

**Mode 1** When X9 is energized, and X11 is off, the drum remains in its current state (step 2) for 5 seconds.

- After 5 seconds, DRUM 1 indexes to step 3 and remains there for 6 seconds. Output coil Y4 energizes, and Y2, Y3, Y7, and Y8 remain on for the duration of this step.
- DRUM 1 continues to index through successive steps and remains in each step for the duration of the programmed CNT/STP times SEC/CNT. The output coils take on the states of the active step Mask.
- When step 16 is reached output coils Y2 through Y8 turn off. The drum remains in this step for 10 seconds, then Y1 turns on, resetting DRUM 1 to step 2; then the sequence continues.

**Mode 2** Each time X11 is energized, the drum is forced to a step number, that is determined by the states of inputs X12–X16. For example, if X16=0, X15=0, X14=1, X13=0, and X12=1, (00101) the drum is forced to step 5.

- When X11 is energized, O/S 1 turns on for one scan. This moves the drum step preset (DSP1) to memory location V1, the states of inputs X12–X16 to memory location V2, and turns on C1.
- With C1 energized, CMP 1 compares the step preset (in V1) to the step specified by the inputs (in V2). If the new step number in V2 is less than the value in V1, C5 turns on.
- With C5 energized, MOVW 2 loads the step number V1 to V2, thus defaulting to the step previously defined by DSP1. This limits the range of possible steps to a value between DSP1 and 16.
- MOVW 3 moves the step in location V2 to DSP1 and turns on C2. If the value loaded into DSP1 is not between 1 and 15, DSP1 defaults to 16.
- With C2 energized, the drum resets and then indexes to the value specified by DSP1.
- MOVW 4 loads the step preset from V1 back to DSP1.

# Using the DRUM (continued)

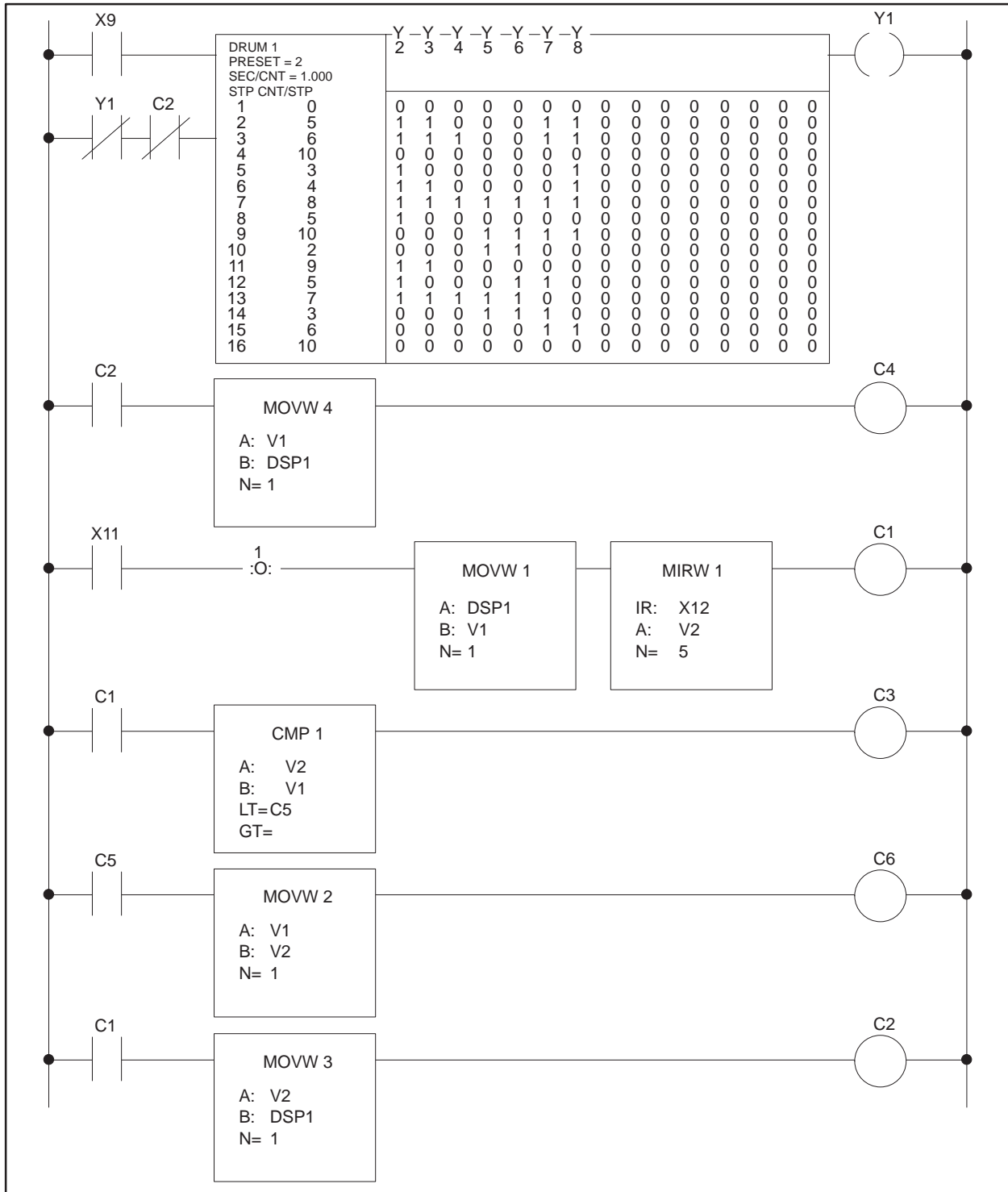


Figure E-13 RLL for DRUM Application Example

## E.6 Using the EDRUM

---

**EDRUM Application Example** A cam limit-switch on a rotating grinder is to be replaced by an event drum. The following solution was devised.

- An absolute encoder with a 10-bit Gray code output provides shaft position location from 0 (0 degrees) to 1024 (360 degrees) for the grinder table.
- An EDRUM is used to alter discrete outputs to control functions such as speed, pressure, and coolant at 15 pre-programmed shaft angles..
- The 15 angles are loaded in V-Memory locations V90 through V104.

**Explanation** Figure E-14, beginning on page E-15, illustrates the RLL solution.

---

**NOTE:** Gray code is binary code where only 1 bit changes as the counting number increases. For example: in Gray code, the integer 2 is represented as 0011, the integer 3 is represented as 0010, and the integer 4 is represented as 0110. Each number is different from the next by one digit.

---

- A 10-bit Gray-to-binary circuit converts the absolute shaft encoder input and stores the result in V603.
- Input X10 controls the operation of the grinder. When X10 is off, MWFT 1 is reset to the start of the angle table, SHRB 1 is cleared and EDRUM 1 is held at the preset step where all outputs are off.
- When X10 turns on, the scaling constants required to convert the 10-bit binary shaft position into degrees are loaded by LDC 1 and LDC 2. MULT 1 and DIV 1 perform the scaling and cause the current shaft position (in degrees) to be loaded into V606.

## Using the EDRUM (continued)

---

- One Shot 1 causes C1 to turn on for one scan. This allows MWFT 1 to load the first angle (V90) into V200
- Power flow through C1 also causes the C2 latch to be set. This allows a 1 to be the first data clocked into SHRB 1 when the correct starting angle (V90) is reached.
- CMP 1 compares the current shaft position loaded into V606 with the next angle in the table. When the values match C3 is turned on. This causes MWFT 1 to load the next value in the angle table in V200
- Each time C3 is turned on, SHRB 1 shifts one bit. The first time C3 turns on, the C2 latch is still set and a 1 is loaded. After that, only 0s are clocked until the SHRB is full.

As the 1 moves through the bit shift register, each move causes the next event in Event Drum 1 to occur. This causes the EDRUM to move to the next step and adjust to the states of outputs Y17 through Y31. These outputs control the speed, pressure, and coolant.

- The process repeats as long as X10 remains on. This indicates that a new part was loaded and that the grinder has returned to the correct start position at the end of each cycle.
- To set the grinder for a new part, alter the values in V90 – V104. The grinder can run multiple parts by using controller logic to change the locations to match the part indexed in the grinder.

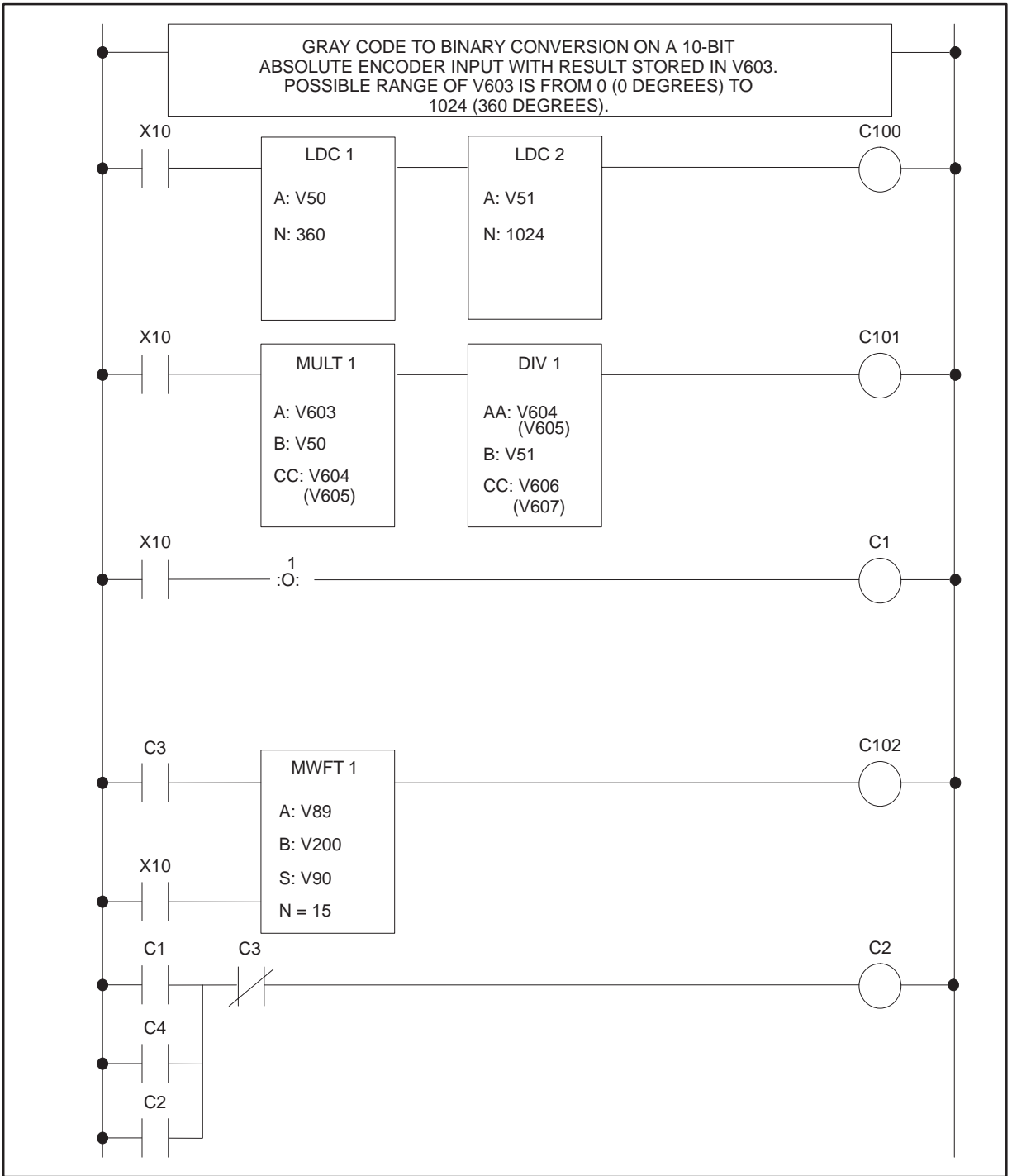


Figure E-14 RLL for EDRUM Application Example



# Using the EDRUM (continued)

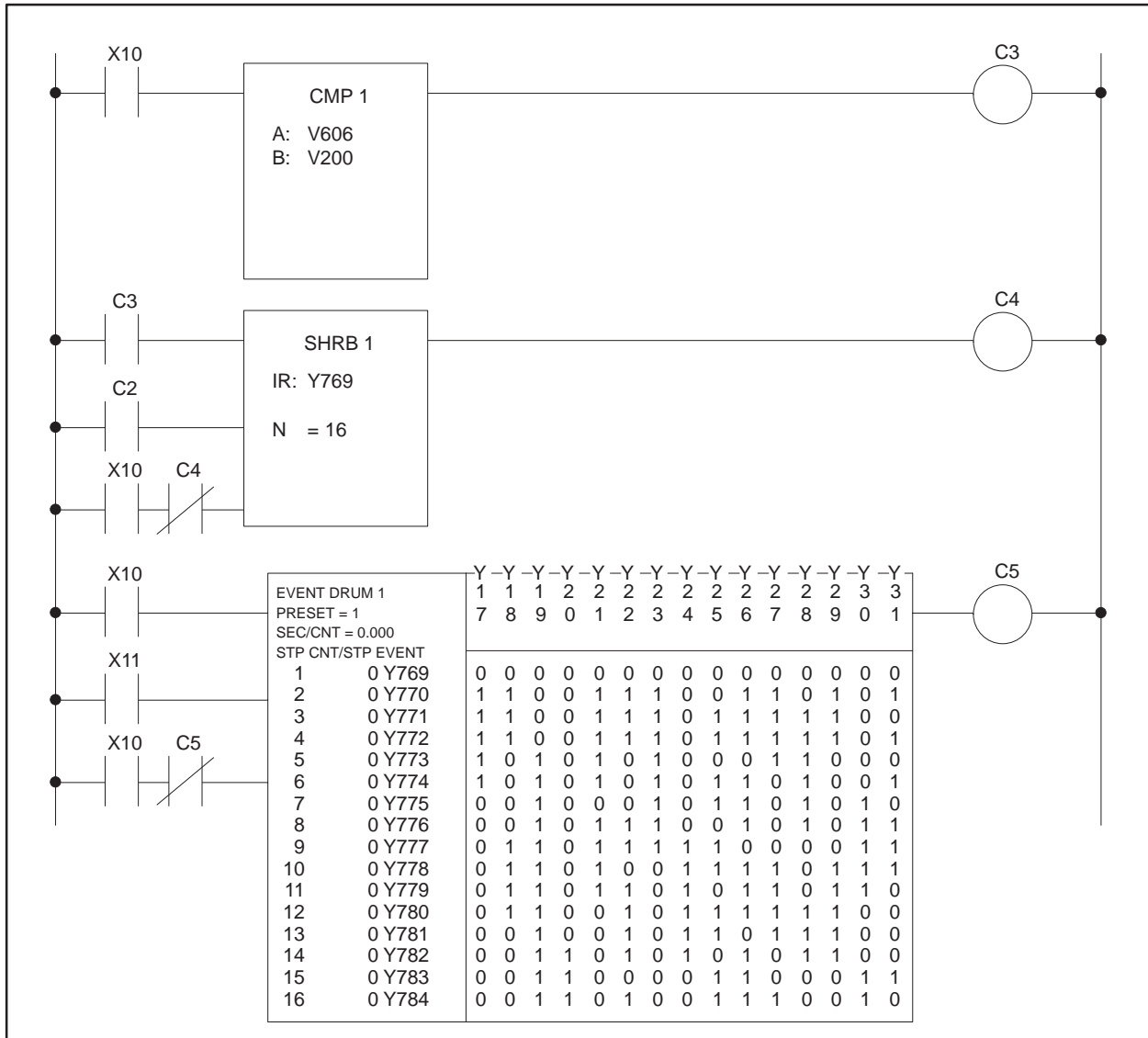


Figure E-14 RLL for EDRUM Application Example (continued)

## E.7 Using the MIRW

### Application

A ribbon-width measuring device tracks the edge of a product sheet moving along a conveyer. Two shaft encoders with a Gray code output provide sensors with position data. When both encoders are zero at the center of conveyer, the distance between the edge sensors is 16 inches (8 inches from the conveyer centerline). Three width calculations are required: 1) the width from the conveyer centerline to one edge; 2) the width from the centerline to the other edge, (these are for sheet-to-conveyer tracking information); and 3) the total width for product output calculations. Figure E-15 illustrates this application.

---

**NOTE:** Gray code is a binary code in which only 1 bit changes as the counting number increases. For example: in Gray code, the integer 2 is represented as 0011, the integer 3 is represented as 0010, and the integer 4 is represented as 0110. Each number is different from the next by one digit.

---

The following procedure provides a solution.

- The edge sensors track the sheet edge by providing a feedback signal to the appropriate drive motor (1M or 2M).
- Limit switches 1LS, 2LS, 3LS, and 4LS are over-travel limit detectors.
- The following values are loaded in V-Memory.

V900 = integer 24 (bit scaling)  
V901 = integer 800 (distance from centerline is 8.00 inches)  
V902 = integer 100 (scale encoder input to correct format prior to adding)

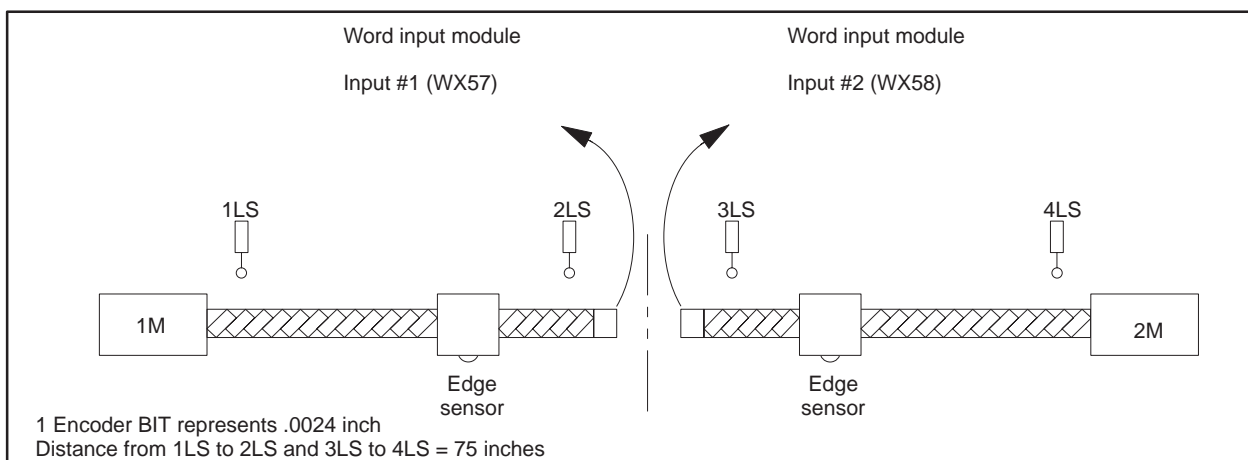


Figure E-15 MIRW Application Example

## Using the MIRW (continued)

---

### Explanation

Figure E-16 illustrates the RLL solution.

- When C27 has power flow, MWIR 3 loads the shaft encoder input into IR locations C124 through C138.
- The encoder Gray code is converted to binary logic, that is stored in IR locations Y140 through Y154.
- When C14 has power flow, MIRW 3 moves the status of Y540–Y554 into memory location V975.
- With C27 still on, MWIR 4 loads the second shaft encoder input into IR locations C156–C170.
- Gray code is converted to binary and stored in Y172–Y186.
- When C16 turns on, MIRW 4 moves the status of Y172–Y186 into memory location V976.
- When C15 and C17 turn on, MULT 10 multiplies the contents of V975 (encoder binary equivalent) by the scaling constant in V900 (integer 24), and stores the result in memory locations V977 and V978. MULT 11 multiplies V976 by V900, and stores the result in V979 and V980.
- When C18 turns on, DIV 8 and DIV 9 divide the scaled encoder values by 100.
- When C19 turns on, ADD 21 adds the scaled value (V981) for one side of the sheet to the fixed distance (V901) from the conveyer center line and stores the result in V985. V985 now contains the width of half the sheet, from the conveyer center line to one outside edge.
- ADD 22 adds V983 to V901 and stores the width of the other side of the sheet into memory location V986. The operator examines V985 and V986 to see whether the sheet is tracking to the left or right.
- ADD 23 adds the values in V985 and V986 and stores sheet width in V987. If  $WX57 = 31,680_{10}$  and  $WX58 = 29,990_{10}$

$$\frac{31680 \times 24}{100} + 800 = 8403 \text{ or } 84.03 \text{ inches}$$

$$\frac{29984 \times 24}{100} + 800 = 7996 \text{ or } 79.96 \text{ inches}$$

$$\text{Sheet Width} = 8403 + 7996 = 16,399 \text{ or } 163.99 \text{ inches}$$

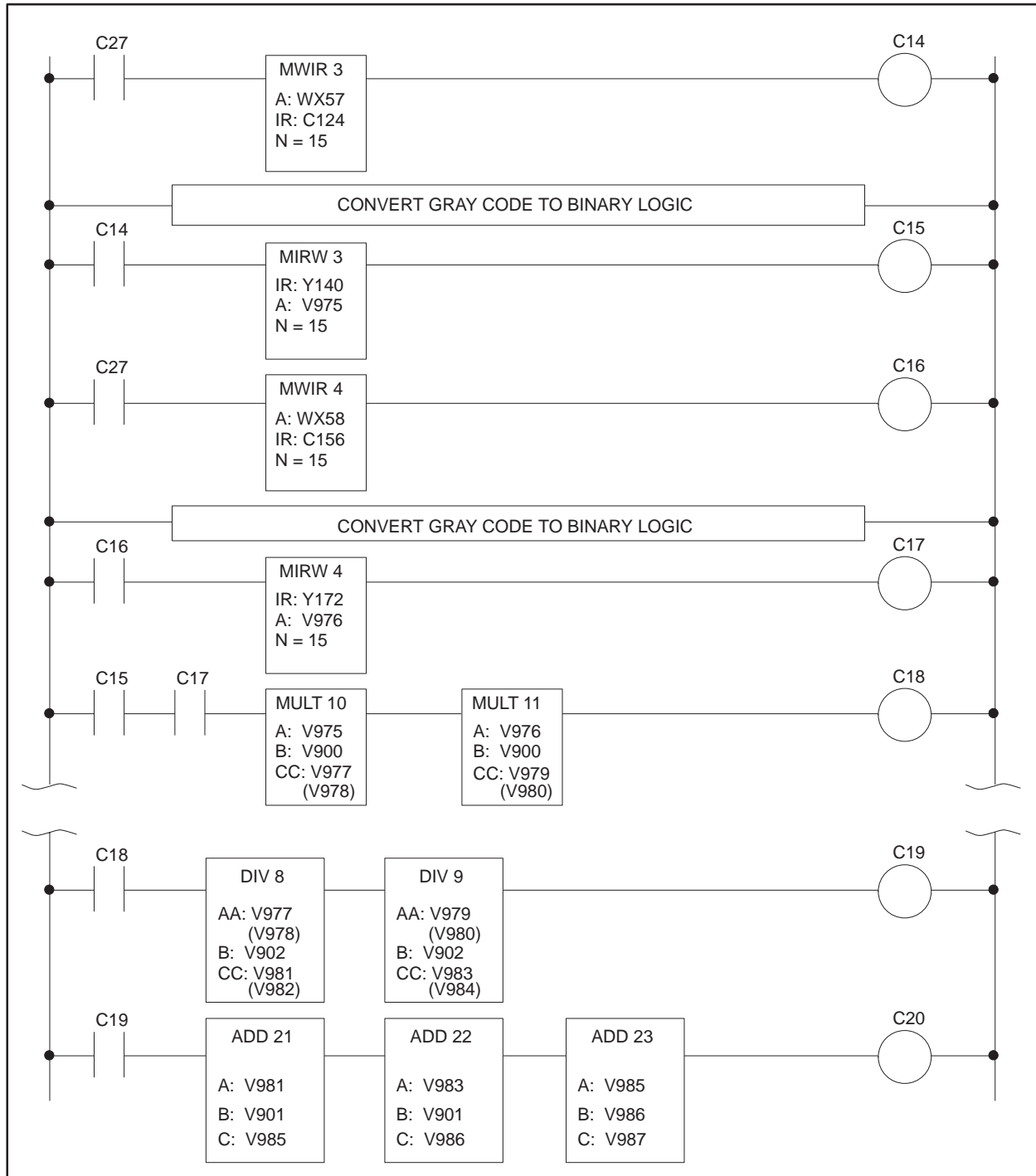


Figure E-16 RLL for MIRW Application Example

## E.8 Using the MWIR

---

**Application** A 15-bit Gray code encoder is used to input shaft position into the controller. The Gray code is to be converted to integer format for scaling and mathematical operations.

The following solution was devised.

- The MWIR converts from word format to bit format.
- Use Ladder logic to convert the bits from Gray code to integer.
- An MIRW converts the altered bits back to word format.

**Explanation** The RLL solution shown in Figure E-17 solves the application.

- If contact C27 has power flow, MWIR 3 moves the encoder input data from word IR WX57 to discrete IR locations C124–C138. (C124 is the LSB.)
- Bit 1 (MSB) of the Gray code is the same as the first bit of a binary number; therefore, Y154 and C138 are the same state (1 or 0).
- If bit 2 of the Gray code is 0, the second binary bit is the same as the first; if bit 2 of the Gray code is 1, the second binary bit is the inverse of the first binary bit. If C137 is open, Y153 follows the state of Y154. When C137 has power flow, Y153 is energized if Y154 is off; and Y153 is de-energized if Y154 is on
- The above step is repeated for each bit.
- MIRW 4 moves the converted word located in discrete IR Y140–Y154 to memory location V975. Y140 is the LSB. V975 now contains the binary equivalent of the Gray code encoder input.

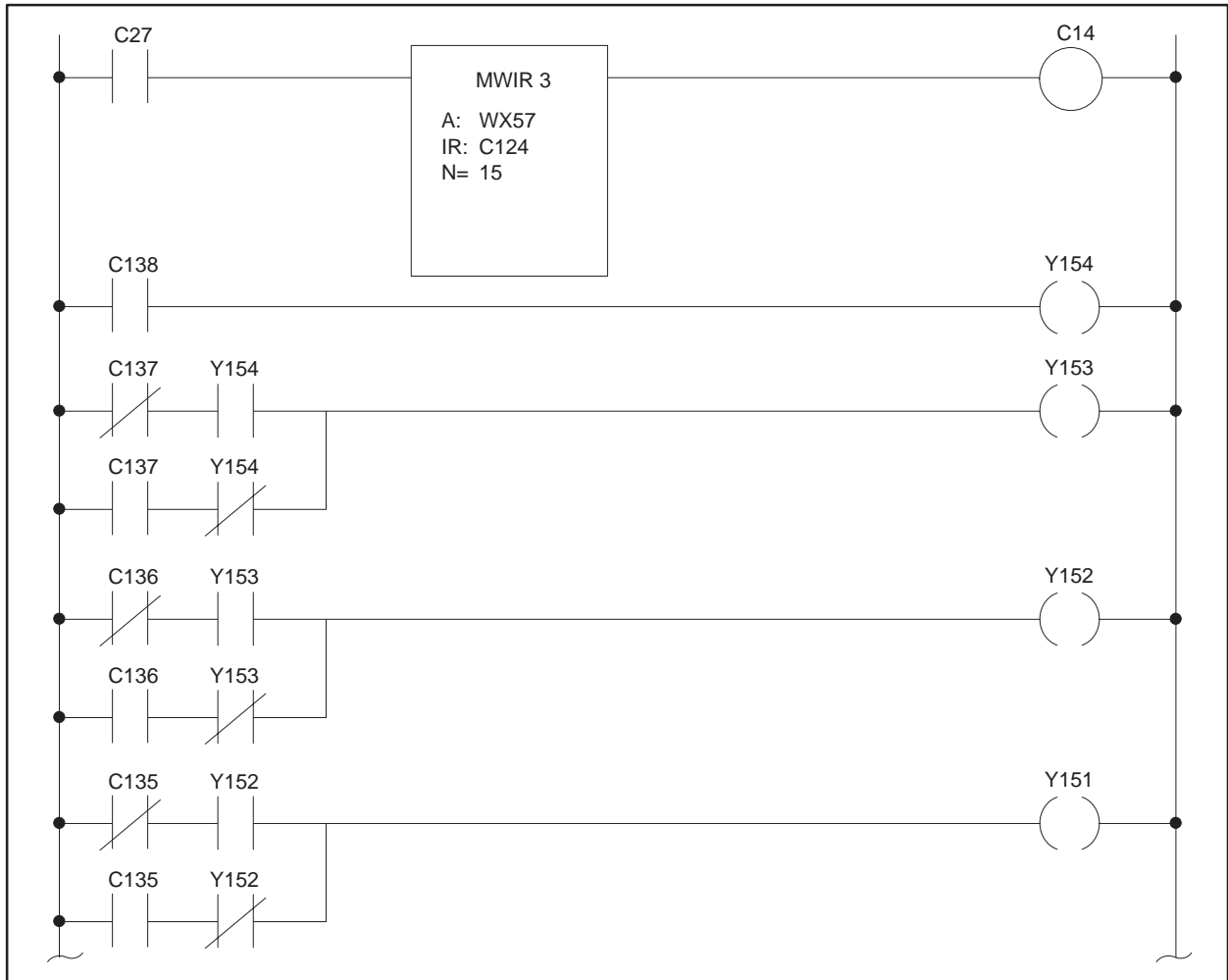


Figure E-17 RLL for MWIR Application Example  
(continued on next 2 pages)

## Using the MWIR (continued)

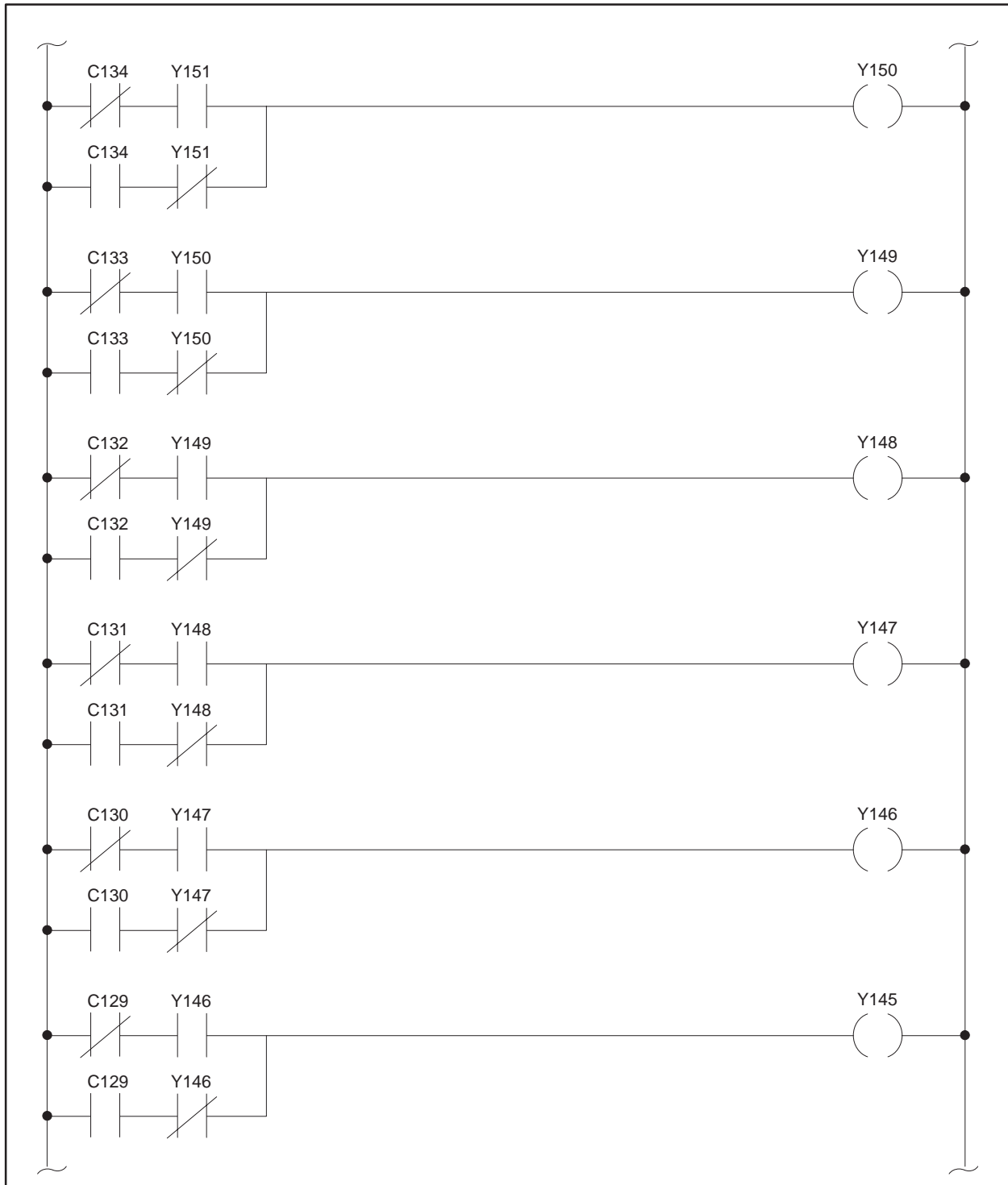


Figure E-17 RLL for MWIR Application Example (continued)

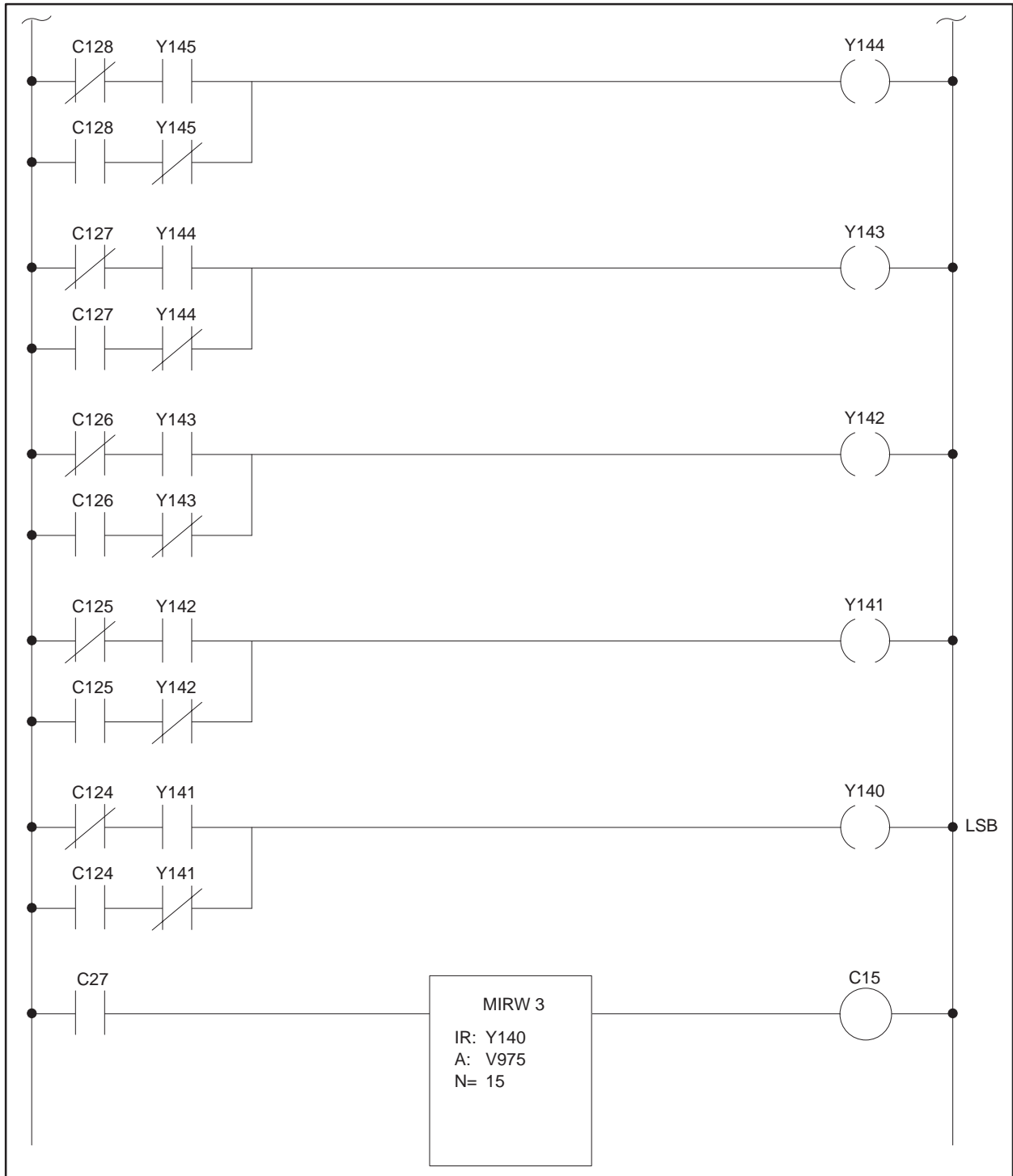


Figure E-17 RLL for MWIR Application Example (continued)



## E.9 Using the MWTT

---

### Application

A thermocouple temperature reading is to be logged every five minutes. The thermocouple input is linearized through a transmitter (shown in Figure E-18) and input to the controller through the first input of an Analog Input Module in Slot 3 of Base 10 (WX657). The temperature table is to be used for work shift history of trend recording.

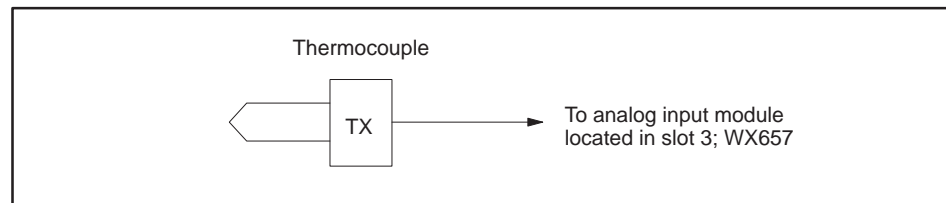


Figure E-18 MWTT Application Example

The following solution was devised.

- A one shot is turned on every five minutes by a timer.
- The one shot activates the logic to scale the thermocouple input, adds a low end offset temperature, and loads the result into a table with 150 locations.

### Explanation

The RLL solution shown in Figure E-19 solves the application.

- Every five minutes, C36 is turned on by a timing circuit (not shown) and C36 turns on one shot 5. One shot 5 activates MULT 38 for the first scan in which C36 is on, to multiply the analog input value (WX657) times a scaling constant loaded in memory location V117. The result is stored in locations V118 and V119.
- DIV 38 divides the scaled value in V118 and V119 by a constant loaded in V100. The quotient is stored in V120 and the remainder in V121.
- C37 turns on after DIV 38 executes, allowing ADD 38 to add the scaled temperature input (V120) to an offset temperature value that has been loaded into V101.
- C38 is energized after ADD 38 executes, allowing MWTT 7 to load the temperature value (located in V122) into the table at the pointer address in V123.
- When MWTT 7 is reset (contact X10 is off for one scan), the pointer address in V123 is reset to 700.
- When the pointer address in V123 reaches 849, C39 turns on, and no additional values are loaded into the table until MWTT 7 is reset.

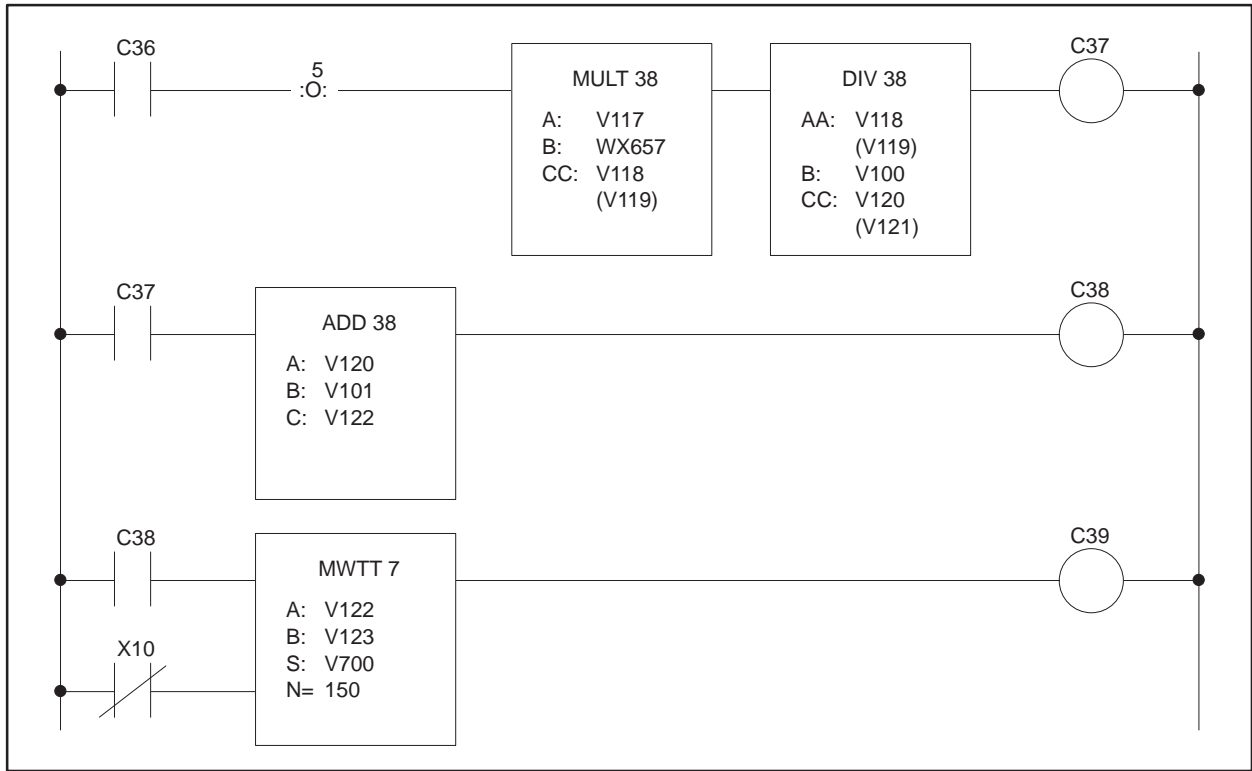


Figure E-19 RLL for MWTT Application Example

## E.10 Using the MWFT

---

Application	<p>The following example recovers the data (in locations V700–V849) stored in the MWTT application example program. The data points are plotted for a report. The plotter is connected to output word WY57. The data should change every second. Therefore, one second on the plot represents five minutes of the process.</p>
Explanation	<p>The RLL solution shown in Figure E-20 solves the application.</p> <ul style="list-style-type: none"><li>• When X1 is turned on, plotting begins. Once every second, TMR1 causes C1 to turn on for one scan.</li><li>• Each time C1 turns on, MWFT1 transfers a new word from the memory table to V101, beginning at V700. This continues once per second until all 150 words have been moved to V101, i.e., until V849 has been transferred.</li><li>• MOVW1 transfers the data in V101 to WY57, that is the plotter output word.</li><li>• Once started, X1 must be cycled off and then on to restart the plotting process.</li></ul>

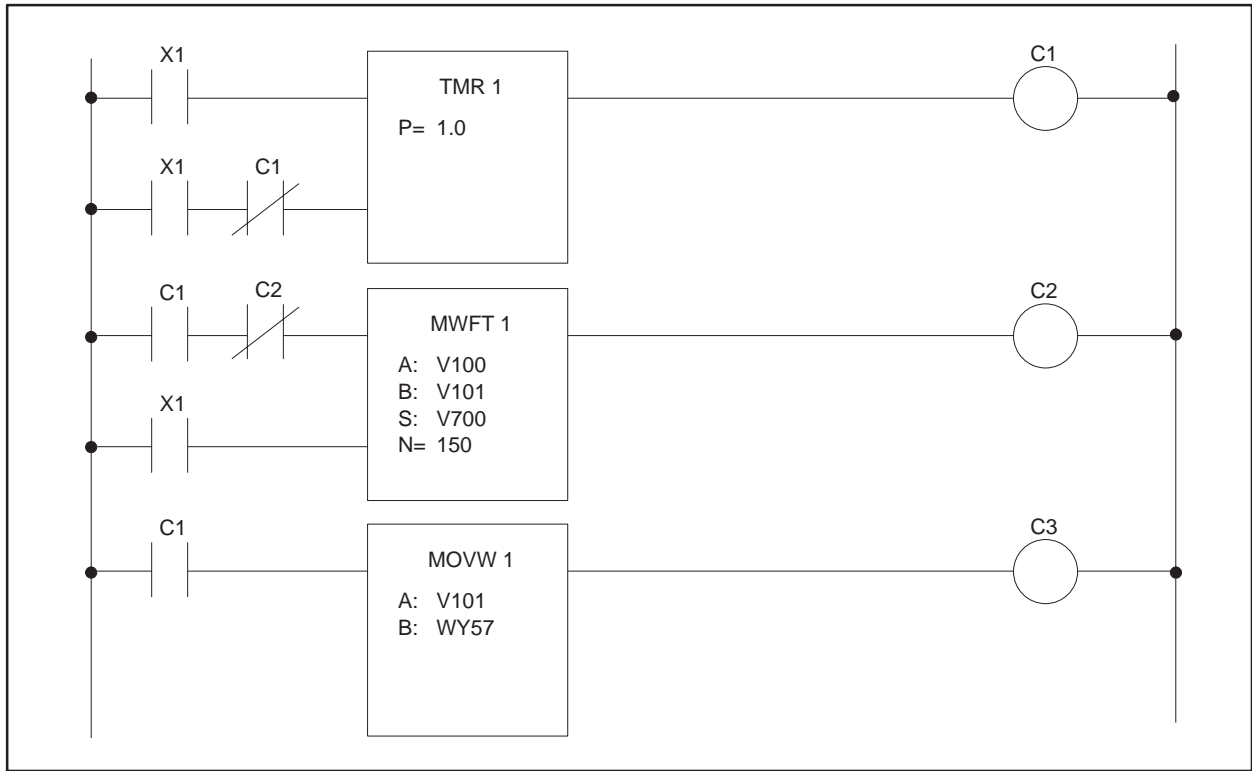


Figure E-20 RLL for MWFT Application Example

## E.11 Using the WXOR

**Application** At a critical point in a process, the status of 16 discrete inputs must be in a specific state to execute an operation. If any of the 16 inputs is not in the correct state, an alarm is sounded. There are 16 indicators that display which inputs are in the wrong state.

**Explanation** This application could be solved with contacts and coils without box functions. To save ladder logic and execution speed, use the RLL shown in Figure E-21.

- Before C1 has power flow, V1 is initialized to zero and V2 is loaded to contain the 16 critical states.
- When the critical process is ready to begin, C1 has power flow causing X1–X16 to be loaded into V3. An Exclusive OR is then executed on V3 and V2. V1 contains the result and contains a one in any bit location where V2 and V3 differ. If V2 and V3 are identical, then V1 contains all zeros and the WXOR 1 output C3 does not turn on.
- A difference between V3 and V2 causes C3 to come on. V1 is moved out to indicators Y41–Y56 to show which inputs are incorrect and alarm Y33 is latched on.
- Reset switch X17 can be turned on to reset alarm Y33 and to clear indicator panel Y41–Y56.

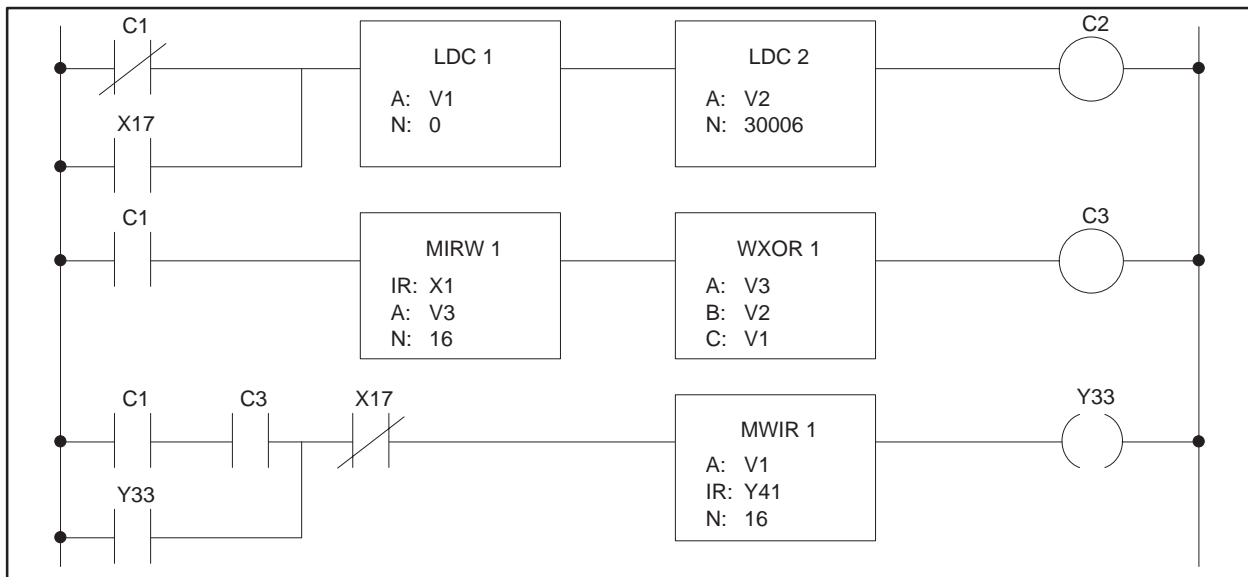


Figure E-21 RLL for WXOR Application Example

**Inputs are Correct**

Before C1 has power flow, the desired values for X1–X16 are loaded into V2, as shown below.

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
V2:	0	1	1	1	0	1	0	1	0	0	1	1	0	1	1	0	= 30,006
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	
	X16	X15	X14	X13	X12	X11	X10	X9	X8	X7	X6	X5	X4	X3	X2	X1	

- When C1 is on, the actual values of X1–X16 are loaded into V3:

X1 = OFF	X5 = ON	X9 = ON	X13 = ON
X2 = ON	X6 = ON	X10 = OFF	X14 = ON
X3 = ON	X7 = OFF	X11 = ON	X15 = ON
X4 = OFF	X8 = OFF	X12 = OFF	X16 = OFF

	X16	X15	X14	X13	X12	X11	X10	X9	X8	X7	X6	X5	X4	X3	X2	X1	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Actual Values In V3	0	1	1	1	0	1	0	1	0	0	1	1	0	1	1	0	30,006
Desired Values In V2	0	1	1	1	0	1	0	1	0	0	1	1	0	1	1	0	30,006
From The WXOR V1 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Since the WXOR 1 result is zero, C3 is not turned on, and MWIR 1 in the next rung is not executed. Alarm Y33 is not sounded.

**Inputs are Incorrect**

The inputs from the example above are used, except that inputs #5 and #12 are incorrect.

	X16	X15	X14	X13	X12	X11	X10	X9	X8	X7	X6	X5	X4	X3	X2	X1	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Actual Values In V3	0	1	1	1	1	1	0	1	0	0	1	0	0	1	1	0	32,038
Desired Values In V2	0	1	1	1	0	1	0	1	0	0	1	1	0	1	1	0	30,006
From The WXOR V1 =	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	2,064
	Y56	Y55	Y54	Y53	Y52	Y51	Y50	Y49	Y48	Y47	Y46	Y45	Y44	Y43	Y42	Y41	

- Since the WXOR result is not all zeros, C3 is turned on and the MWIR is executed. Y45 indicates that X5 is in the wrong state, and Y52 indicates that X12 is in the wrong state. Alarm Y33 stays on until reset by X17.

## E.12 Using the CBD

**Application** A 0-volt to +5-volt signal is monitored, and the voltage is read on a panel meter located at the controller. The 0-volt to +5-volt signal is the third input of an analog module located in slot 8 of Base 9. The following procedure provides a solution.

- MULT 36 and DIV 36 scale the analog input.
- CBD 16 converts the scaled integer value to a BCD value.
- MOVW 81 moves the BCD value to a word IR for output to a panel meter through a Word Output Module.

**Explanation** The RLL in Figure E-22 does the function that follows.

When X19 has power flow, the analog equivalent value located in the word IR WX635. . .

$$WX = \frac{\text{Input Voltage}}{5 \text{ Volts}} \times 32,000$$

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
WX635 =	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0

= Binary integer 28,896

. . . is multiplied by a scaling factor that previously has been loaded into memory location V123,

$$\frac{5.0 \text{ V}}{32000} \times 1 \times 10^7 = 1562$$

V123 =	0	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0
--------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

= Binary integer 1562

. . . and the result is stored in memory locations V124 and V125.

V124 =	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0
V125 =	1	0	1	1	0	1	1	0	1	1	0	0	0	0	0	0

} = Binary integer 45,135,552

The output of MULT 36 is energized, starting the DIV 36 operation. The value stored in memory locations V124 and V125 is divided by a scaling factor that previously has been loaded into memory location V100,

V100 =	0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0
--------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

= Binary integer 10,000

. . . and the result is stored in memory locations V126 and V127.

V126 = 

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

 = Binary integer 4513

V127 = 

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

 = Binary integer 5552

The output of DIV 36 energizes C73, starting the CBD 16 operation. The value stored in memory location V126 is converted to its BCD equivalent, and the result is stored in memory locations V128 and V129.

V128 = 

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 = 0

V129 = 

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

 = BCD 4513

4
5
1
3

The output of CBD 16 energizes, starting the MOVW 81 operation. The value stored in memory location V129 is moved to the output IR WY65. IR WY65 outputs the BCD number to a Word Output Module located in Slot 1 of Base 1. WY65 is the first output of this module. A reading of 4.513 volts is displayed on a digital panel meter where the decimal point is fixed internally to the panel meter.

From analog input or WX, V input = (Binary integer x 5 volts) ÷ 32,000

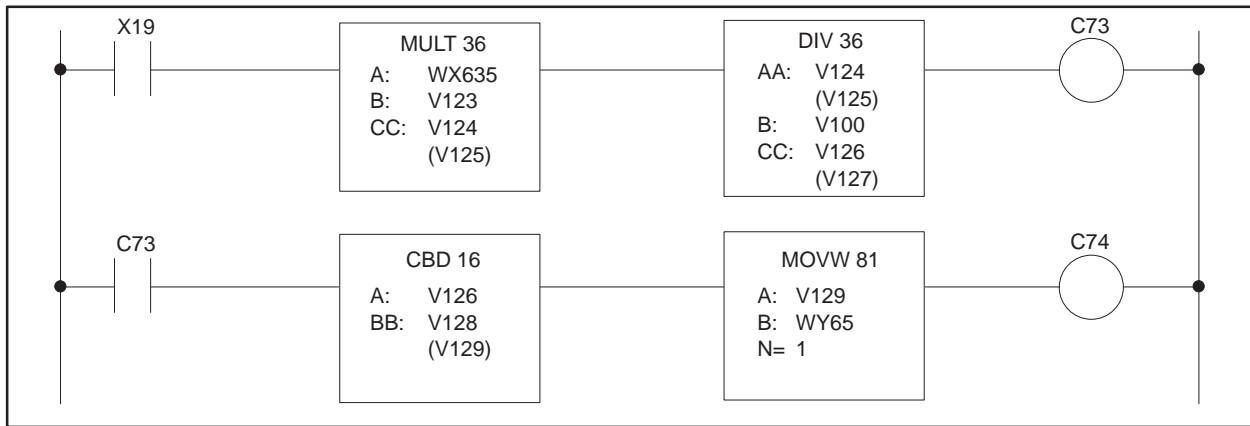


Figure E-22 RLL for CBD Application Example



## E.13 Using the CDB

### Application

BCD thumbwheels are input 2 of a Word Input Module located in Slot 3 of Base 6 (WX402). The thumbwheel input is to be converted to a binary integer equivalent for use in mathematics instructions.

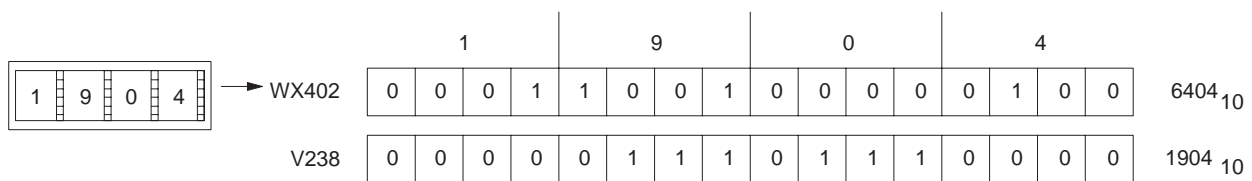
The following solution was devised.

- CDB 1 converts the word input from BCD to an integer.
- DIV 3 is a mathematics instruction in which the divisor is modified by a thumbwheel switch.

### Explanation

Figure E-23 shows the RLL for this operation.

- When contact C67 has power flow, CDB 1 converts the BCD value located in IR WX402 to an integer value, that is put in memory location V238.



- DIV 3 divides V635 and V636 by V238, and puts the quotient in V79.
- Coil C1 is energized when the instructions execute.

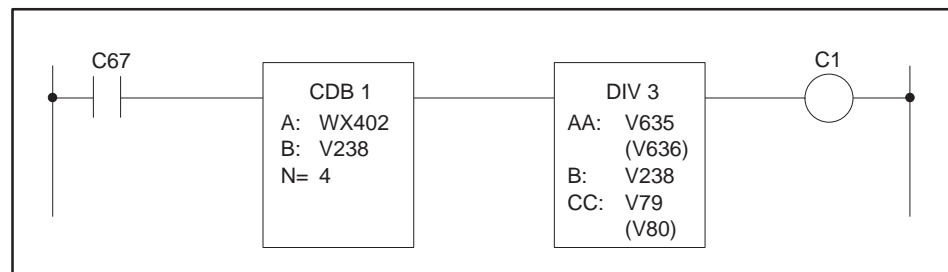


Figure E-23 RLL for CDB Application Example

## E.14 Using the One Shot

**Application** Each time a momentary pushbutton is pressed, an ADD executes once. The pushbutton address is X1.

The following solution was devised.

- A one shot preceding an ADD instruction solves this example.

**Explanation** Figure E-24 shows the RLL for this operation.

- When X1 is pressed, the output of one shot 20 is energized for one controller scan, and ADD 41 executes only during this controller scan.
- X1 must be turned off for at least one controller scan, and then turned on again, for the ADD 41 to execute again.

Values prior to network execution:

WX100= 70<sub>10</sub>

WX101= 51<sub>10</sub>

V74= 0<sub>10</sub>

Values after network execution:

WX100= 70<sub>10</sub>

WX101= 51<sub>10</sub>

V74= 121



Figure E-24 RLL for One Shot Application Example

If all the One Shot instruction numbers have been used, you can build one from RLL, as shown in Figure E-25.

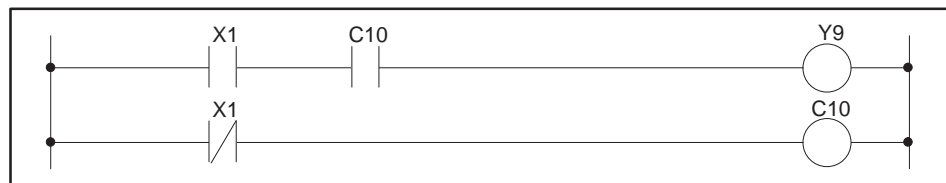


Figure E-25 Constructing a One Shot From RLL

## E.15 Using the DCAT

### Application

A remotely located pipeline valve is opened and closed by control logic. Because of the diameter of the pipeline, the valve requires 30 seconds to open or close. Feedback for the valve status informs maintenance personnel whether the valve is open, closed, traveling, failed to open, failed to close, or the sensor has failed. See Figure E-26.

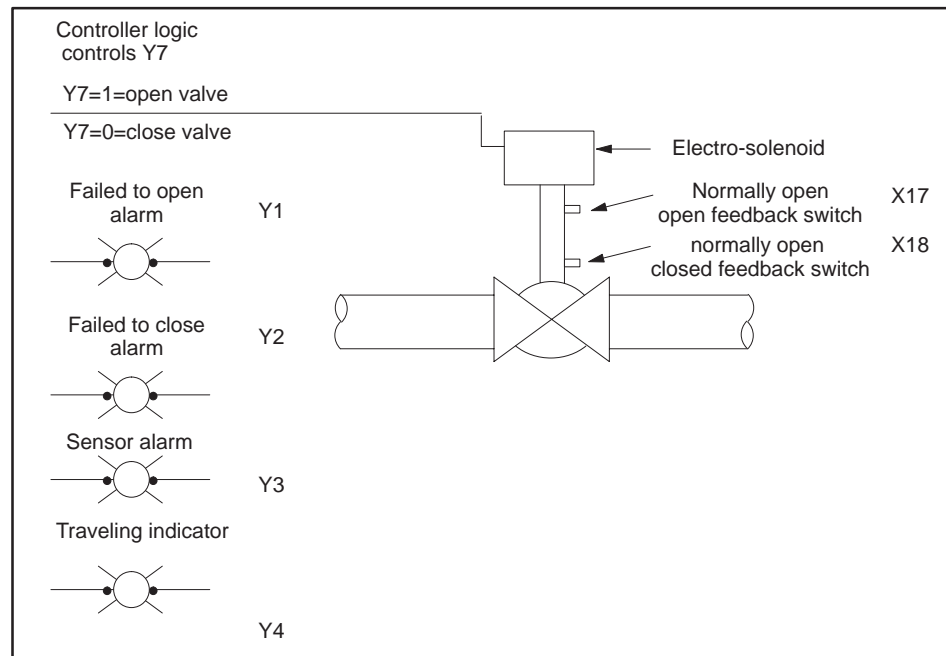


Figure E-26 DCAT Application Example

The following solution was devised.

- Control logic opens or closes the valve by sending power flow to electro-solenoid Y7.
- Limit Switch X17 is the normally open feedback switch that closes to indicate that the valve is open.
- Limit Switch X18 is the normally open feedback switch that closes to indicate that the valve is closed.
- While the valve transitions, the Y4 indicator (traveling) is on.
- If the valves fails to open, alarm Y1 turns on.
- If the valves fails to close, alarm Y2 turns on.
- If both feedback sensors are closed (for example, a sensor sticks), sensor failure alarm Y3 is turned on.

**Explanation**

The RLL solution is shown in Figure E-27. Valve control is accomplished by the events described below. Other program steps control the status of valve Y7 by turning Control Relay C40 off or on. Y7 follows the status of C40, unaffected by the DCAT instruction.

**Normal Operation**

Under normal conditions, the following events occur.

- If C40 goes to 1 (on), X17, X18, C5, and C6 are 0 (off) as long as it takes the valve to open.
- Subsequent program steps check the status of X17, X18, C5, and C6. If they are all off, the valve status is reported through indicator Y4 as traveling.
- Open feedback sensor X17 then closes and disables Open Alarm C5.
- If C40 goes to 0 (off) and commands the valve to close, closed feedback sensor X18 closes and disables the Close Alarm C6.

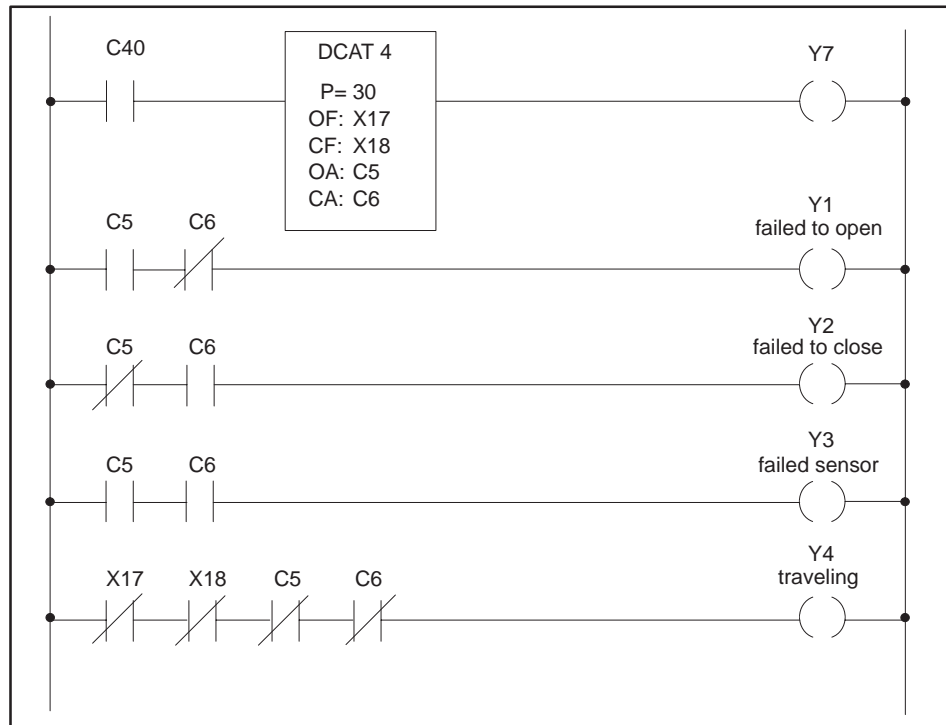


Figure E-27 RLL for DCAT Application Example

## Using the DCAT (continued)

---

- Valve Fails to Open**      If the valve fails to open, the following events occur.
- If C40 goes to 1 (on), commanding the valve to open, and open feedback does not turn on, the timer times out and energizes Open Alarm C5.
  - Subsequent RLL steps check the status of C5 and C6. If C5=1 and C6=0; the failed to open indicator Y1 turns on.
- Valve Fails to Close**      If the valve fails to close, the following events occur.
- If C40 goes to 0 (off), commanding the valve to close, and closed feedback does not turn on, the timer times out and energizes Closed Alarm C6.
  - Subsequent RLL steps check the status of C5 and C6. If C5=0 and C6=1, the failed to close indicator Y2 turns on.
- Sensor Fails**              If the sensor fails, the following events occur.
- At any time that X17 and X18 are both on, the DCAT turns on C5 and C6. Y4 reports a failure of the valve sensor system.

## E.16 Using Status Words

**Application** A procedure is required that logs off a failed I/O module and logs on a backup-module in the same base.

**NOTE:** Only self-diagnosing modules can indicate their own failure.

Y24 = Module Failure Alarm. Example module assignments:

- Module 1 in slot 1, Base 0 = WX1 to WX8 — STW11, Bit 16
- Module 2 in slot 2, Base 1 = WX9 to WX16 — STW12, Bit 15
- Module 3 in slot 3, Base 0 = Y17 to Y24

**Explanation** The RLL solution is shown in Figure E-28. The status of Input Module #1 is checked with a bit-of-word contact. If the contact turns on (bit 16 in STW11 = 1), the alarm Y24 turns on. The program then uses Y24 to determine which module should source V200 through V207.

For this method to function in an application, both input modules must be hardwired to the same field devices (i.e. WX1 through WX8 should be connected to the same devices, such as WX9 through WX16, respectively).

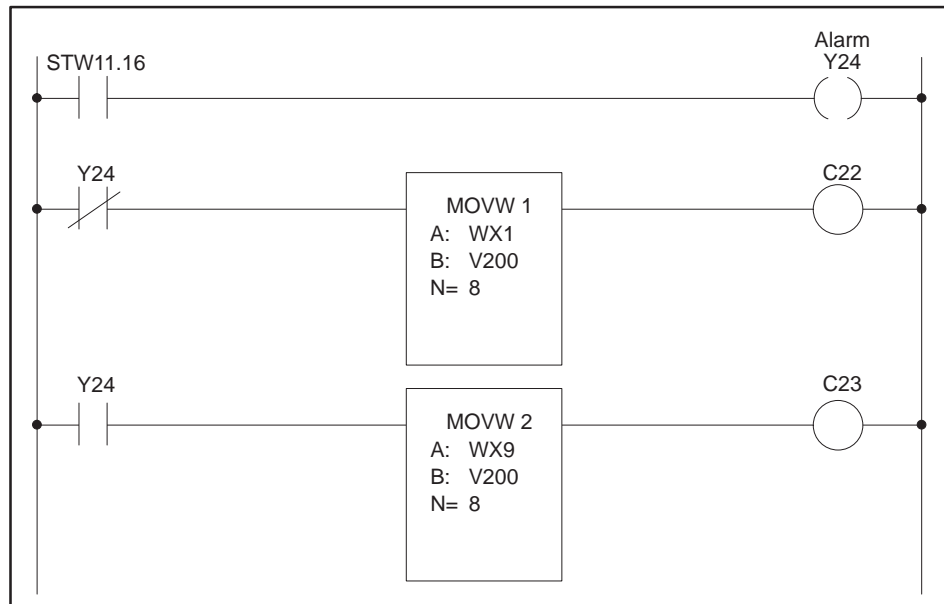


Figure E-28 RLL for Status Word Application Example

# Special Function Program Error Codes

Table F-1 Special Function Error Codes

Code		Meaning
Hex	Decimal	
02	02	Address out of range.
03	03	Requested data not found.
09	09	Incorrect amount of data sent with request.
11	17	Invalid data.
40	64	Operating system error detected.
42	66	Control block number out of range.
43	67	Control block does not exist or has not been compiled.
46	70	Offset out of range.
47	71	Arithmetic error detected while writing loop or analog alarm parameters.
48	72	Invalid SF program type.
49	73	Instruction number or ramp/soak step number out of range.
4A	74	Attempt to access an integer-only variable as a real.
4B	75	Attempt to access a real-only variable as an integer.
4E	78	Attempt to write a read-only variable (for example: X, WX, or STW).
4F	79	Invalid variable data type for this operation.
52	82	Invalid return value.
53	83	Attempt to execute a Cyclic Statement in a non-cyclic SF program.
54	84	Control block is disabled.
56	86	Attempt to perform an FTSR-OUT Statement on an empty FIFO.
57	87	Attempt to perform an FTSR-IN Statement on a full FIFO.
58	88	Stack overflow while evaluating a MATH, IF, or IMATH expression.
59	89	Maximum SFSUB nesting level exceeded. Subroutines may only be nested to a depth of 4.
5A	90	Arithmetic Overflow.
5B	91	Invalid operator in an IF, MATH, or IMATH expression.
5D	93	Attempt to divide by zero (IMATH statement).
5E	94	FIFO is incompatible with FTSR-IN/FTSR-OUT statement.
5F	95	FIFO is invalid.
60	96	Invalid Data Type code. This error is generally caused by an ill-formed MATH, IMATH, or IF expression.

# Appendix G

## Status Words

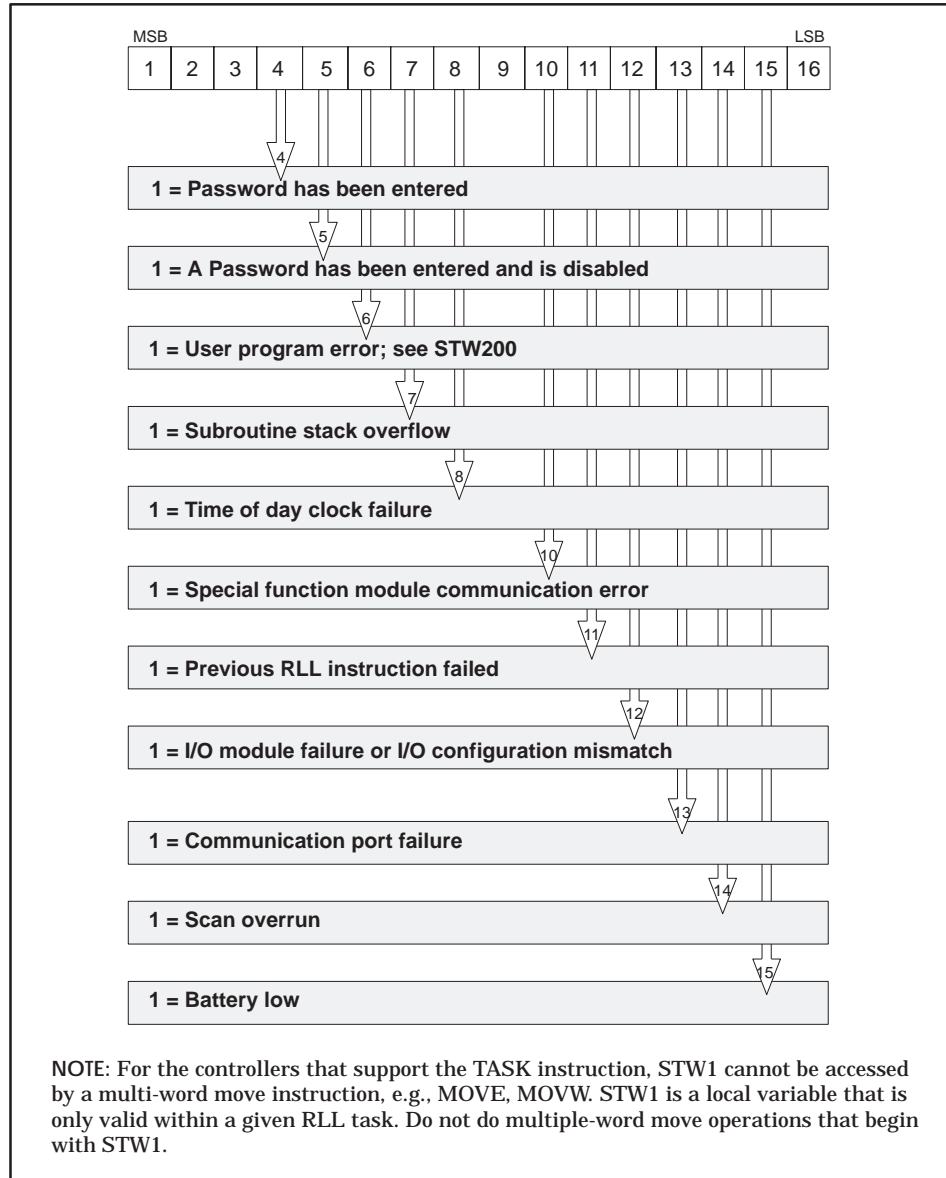
---

STW01: Non-fatal Errors .....	G-2
STW02: Base Controller Status .....	G-3
STW03 – STW09: PROFIBUS-DP Slave Status .....	G-4
STW10: Dynamic Scan Time .....	G-4
STW11 – STW138: I/O Module Status .....	G-5
STW139: Discrete Force Count .....	G-8
STW140: Word Force Count .....	G-8
STW141 – STW144: Date, Time, and Day of Week .....	G-8
STW145 – STW146: Receive and Timeout Errors .....	G-11
STW147: PROFIBUS-DP Slave Errors .....	G-11
STW148: PROFIBUS-DP Bus Communication Errors .....	G-11
STW149 – STW160: Reserved .....	G-11
STW161: Special Function Processor Fatal Errors .....	G-12
STW162: Special Function Processor Non-fatal Errors .....	G-13
STW163: RLL Subroutine Stack Overflow .....	G-14
STW164 – STW165: L-Memory Checksum C0 .....	G-14
STW166 – STW167: L-Memory Checksum C1 .....	G-14
STW168: Dual RBC Status .....	G-15
STW169 – STW175: Reserved .....	G-16
STW176: Dual Power Supply Status .....	G-16
STW177 – STW183: Reserved .....	G-17
STW184: Module Mismatch Indicator .....	G-17
STW185 – STW191: Reserved .....	G-17
STW192: Discrete Scan Execution Time .....	G-17
STW193 – STW199: Reserved .....	G-17
STW200: User Error Cause .....	G-18
STW201: First Scan Flags .....	G-19
STW202: Application Mode Flags (A – P) .....	G-20
STW203: Application Mode Flags (Q – Z) .....	G-21
STW204: Application Installed Flags (A – P) .....	G-22
STW205: Application Installed Flags (Q – Z) .....	G-23
STW206 – STW207: U-Memory Checksum C0 .....	G-24
STW208 – STW209: U-Memory Checksum C1 .....	G-24
STW210: Base Poll Enable Flags .....	G-25
STW211 – STW217: PROFIBUS-DP Slave Enable Flags .....	G-26
STW218: My Application ID .....	G-26
STW219: RLL Task Overrun .....	G-26
STW220: Interrupting Slots in Local Base .....	G-27
STW221: Module Interrupt Request Count .....	G-27
STW222: Spurious Interrupt Count .....	G-27
STW223 – STW225: Binary Time-of-Day .....	G-28
STW226: Time-of-Day Status .....	G-28
STW227 – STW228: Bus Error Access Address .....	G-30
STW229 – STW230: Bus Error Program Offset .....	G-30
STW231 PROFIBUS-DP I/O System Status .....	G-31
STW232 – STW238: PROFIBUS-DP Slave Diagnostic .....	G-31
STW239 – STW240: CS-Memory Checksum C0 .....	G-32
STW241 – STW242: CS-Memory Checksum C1 .....	G-32



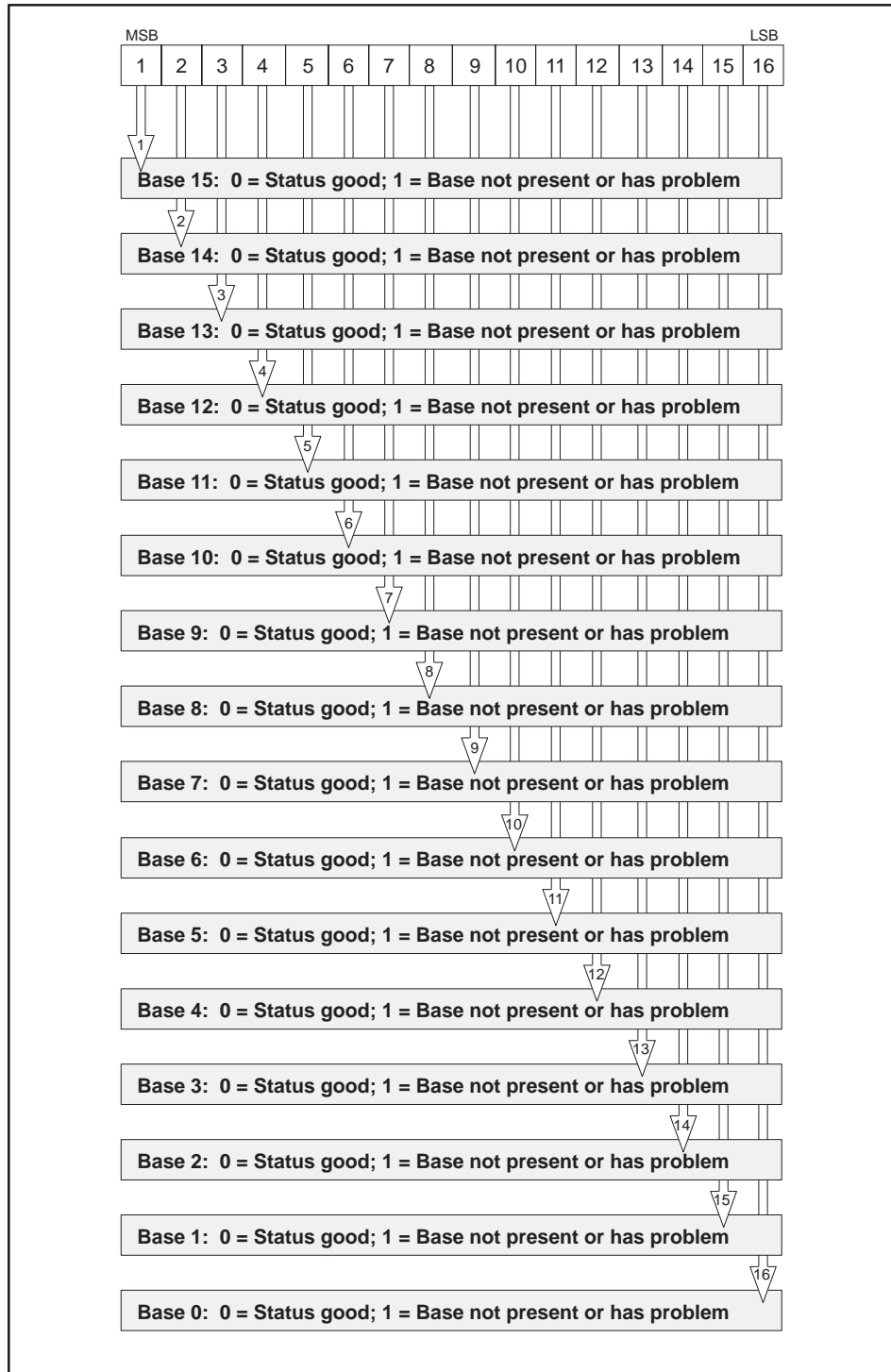
Each status word description explains the function or purpose of each bit within the word. If a bit is not used, it is not described; all unused bits are set to zero. If several bits perform a single function, they are described by a single definition. If a status word is reserved, it is noted accordingly.

**STW01:  
Non-fatal Errors**



**Applicable Controller**    ALL

STW02: Base  
Controller Status



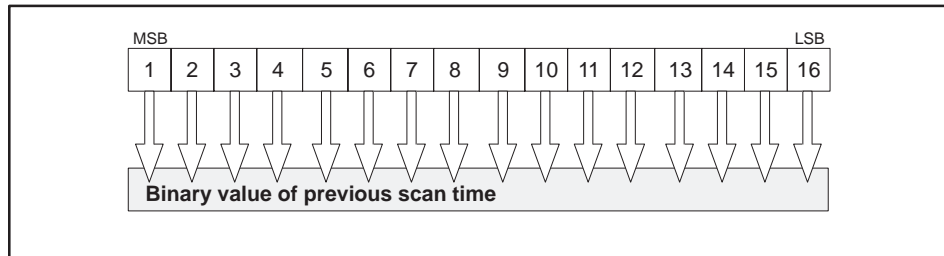
Applicable Controller ALL

**STW03 – STW09:  
PROFIBUS-DP Slave  
Status**

Bit	MSB															LSB
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
STW03	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
STW04	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
STW05	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
STW06	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
STW07	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
STW08	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
STW09	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97

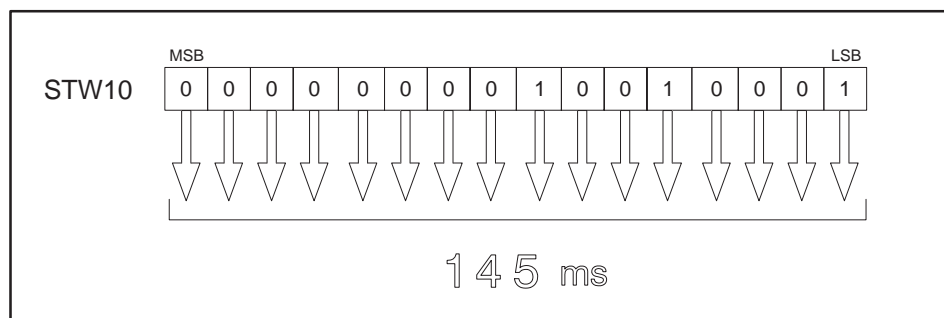
STW03 through STW09 give the status for PROFIBUS-DP slaves. The slave's bit, as indicated in the cells above, is a 1 if the slave is not present or is failed.

**STW10:  
Dynamic Scan  
Time**



**Applicable Controller**    ALL

Figure G-1 illustrates an example of STW10 containing a scan time of 145 ms.



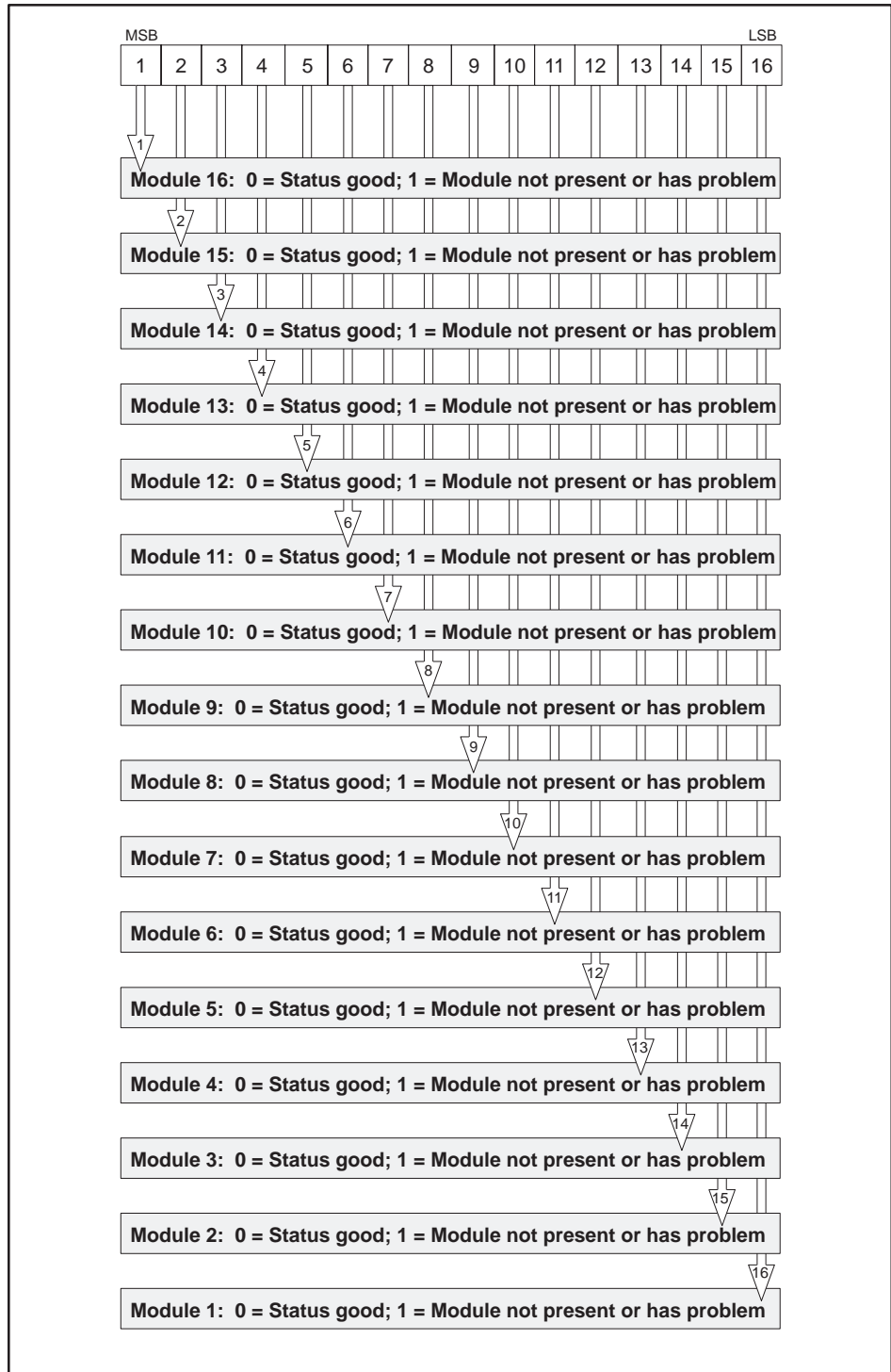
**Figure G-1** Example of Status Word Reporting Scan Time

**STW11 – STW138:  
I/O Module Status**

Status words STW11 through STW138 indicate the status of the individual I/O modules installed in the local base, an RBC in a remote base, or a PPX:505–6870 RBC on the PROFIBUS-DP I/O channel. Status word 11 applies to the local base, status words 12 – 26 apply to the 505 remote I/O channel, and status words 27 – 138 apply to the PROFIBUS-DP I/O channel. The illustration on page G-6 shows the content of these status words. Table G-1 lists the status words that correspond to the status of each base/slave.

**Table G-1 Status Words 11 Through 138**

<b>Status word</b>	<b>505 Modules Local/Remote</b>	<b>Status word</b>	<b>PROFIBUS-DP Module</b>	<b>Status word</b>	<b>PROFIBUS-DP Module</b>	<b>Status word</b>	<b>PROFIBUS-DP Module</b>
11	Local Base	27	Slave 1	43	Slave 17	59	Slave 33
12	Remote I/O Base 1	28	Slave 2	44	Slave 18	60	Slave 34
13	Remote I/O Base 2	29	Slave 3	45	Slave 19	61	Slave 35
14	Remote I/O Base 3	30	Slave 4	46	Slave 20	62	Slave 36
15	Remote I/O Base 4	31	Slave 5	47	Slave 21	63	Slave 37
16	Remote I/O Base 5	32	Slave 6	48	Slave 22	64	Slave 38
17	Remote I/O Base 6	33	Slave 7	49	Slave 23	65	Slave 39
18	Remote I/O Base 7	34	Slave 8	50	Slave 24	66	Slave 40
19	Remote I/O Base 8	35	Slave 9	51	Slave 25	67	Slave 41
20	Remote I/O Base 9	36	Slave 10	52	Slave 26	68	Slave 42
21	Remote I/O Base 10	37	Slave 11	53	Slave 27	69	Slave 43
22	Remote I/O Base 11	38	Slave 12	54	Slave 28	70	Slave 44
23	Remote I/O Base 12	39	Slave 13	55	Slave 29	71	Slave 45
24	Remote I/O Base 13	40	Slave 14	56	Slave 30	72	Slave 46
25	Remote I/O Base 14	41	Slave 15	57	Slave 31	73	Slave 47
26	Remote I/O Base 15	42	Slave 16	58	Slave 32	74	Slave 48
<b>Status word</b>	<b>PROFIBUS-DP Module</b>	<b>Status word</b>	<b>PROFIBUS-DP Module</b>	<b>Status word</b>	<b>PROFIBUS-DP Module</b>	<b>Status word</b>	<b>PROFIBUS-DP Module</b>
75	Slave 49	91	Slave 65	107	Slave 81	123	Slave 97
76	Slave 50	92	Slave 66	108	Slave 82	124	Slave 98
77	Slave 51	93	Slave 67	109	Slave 83	125	Slave 99
78	Slave 52	94	Slave 68	110	Slave 84	126	Slave 100
79	Slave 53	95	Slave 69	111	Slave 85	127	Slave 101
80	Slave 54	96	Slave 70	112	Slave 86	128	Slave 102
81	Slave 55	97	Slave 71	113	Slave 87	129	Slave 103
82	Slave 56	98	Slave 72	114	Slave 88	130	Slave 104
83	Slave 57	99	Slave 73	115	Slave 89	131	Slave 105
84	Slave 58	100	Slave 74	116	Slave 90	132	Slave 106
85	Slave 59	101	Slave 75	117	Slave 91	133	Slave 107
86	Slave 60	102	Slave 76	118	Slave 92	134	Slave 108
87	Slave 61	103	Slave 77	119	Slave 93	135	Slave 109
88	Slave 62	104	Slave 78	120	Slave 94	136	Slave 110
89	Slave 63	105	Slave 79	121	Slave 95	137	Slave 111
90	Slave 64	106	Slave 80	122	Slave 96	138	Slave 112



Applicable Controller ALL

---

STW11 – STW138:  
(continued)

The controller reports an I/O mismatch (an installed module that does not agree with the I/O configuration) as a failed I/O module. Although the module has not actually failed, you must enter correct I/O configuration data or install the proper module to correct the failure report.

In Figure G-2, the 1 in bit 10 indicates that slot seven in Base 4 contains a defective or incorrectly configured module (I/O mismatch). All other slots either contain correctly configured, working modules or are correctly configured as empty.

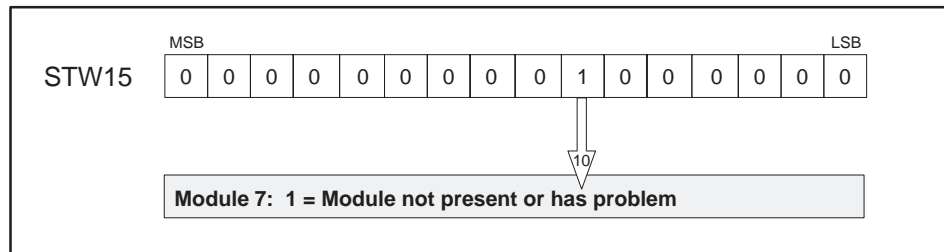


Figure G-2 Example of Status Word Reporting a Module Failure

---

**NOTE:** When a 505 remote I/O base loses communication with the controller, the appropriate bit in STW02 shows a 1. The bits in the status word (STW11–STW26) corresponding to the modules in that base show zeroes, even if modules on that base have failed or been incorrectly configured. That is, the modules of a failed base are not individually indicated as failed.

When a PPX:505–6870 RBC loses communication with the controller, the appropriate bit in STW02 shows a 1. The bits in the status word (STW27–STW138) corresponding to the slave module maintain their most recent value.

When you disable a base from the TISOFT I/O Configuration Screen, all bits in the status word (STW11–STW138) that corresponds to that base are set to zero.

---

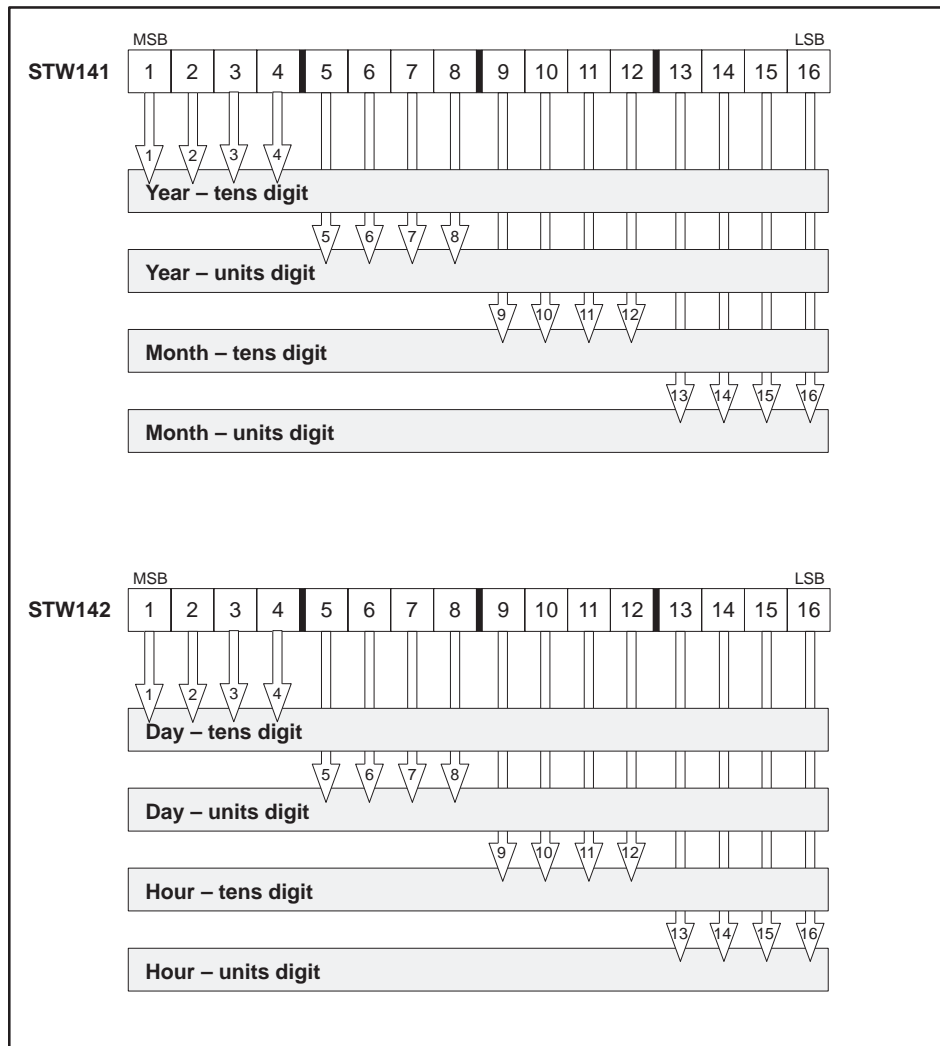
STW139: Discrete Force Count

Status word STW139 provides a count of discrete points (X/Y or C) that are currently forced.

STW140: Word Force Count

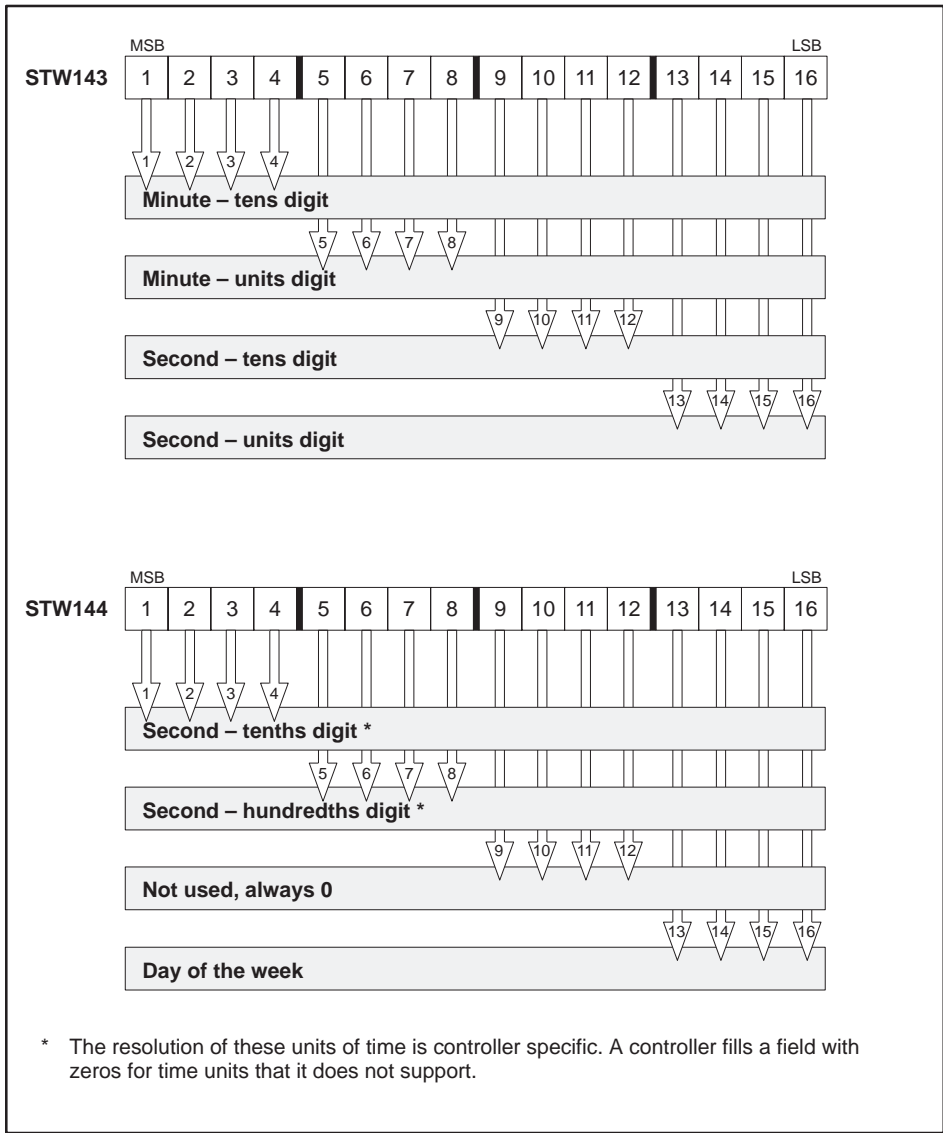
Status word STW140 provides a count of word points (WX/WY) that are currently forced.

STW141 – STW144:  
Date, Time, and  
Day of Week



See also the description of Time of Day Status for STW226 on G-33.

**NOTE:** The time of day is initialized to 1-Jan-1984 at 12:00 AM. (See also STW223-STW226.)



**Applicable Controller**    ALL



Figure G-3 illustrates clock information on the date: Monday, 5 October, 1992 at 6:39:51.76 P.M. Note that the 24-hour (military) format is used and Sunday is assumed to be day 1.

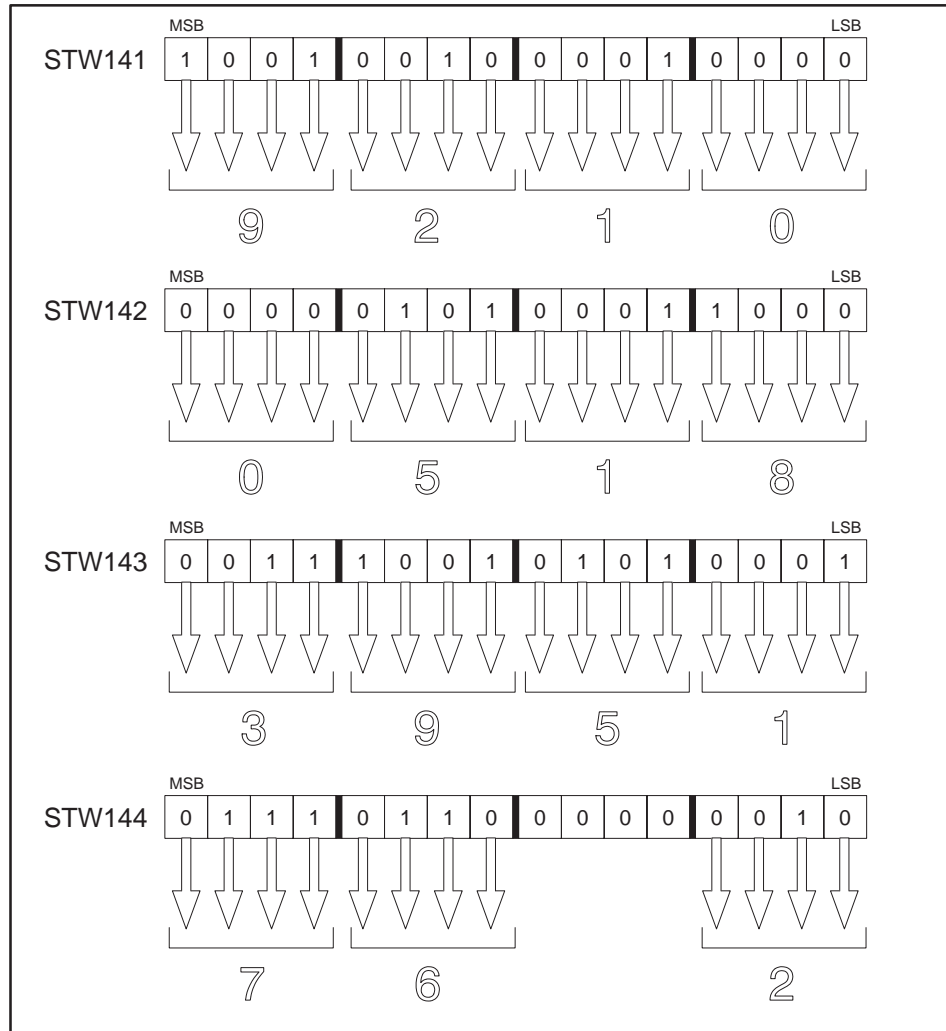


Figure G-3 Example of Status Words Reporting Time

---

STW145 – STW146:  
Receive and  
Timeout Errors

Status words STW145 and STW146 contain communication error counts for Remote I/O channel one. The channel records the number of receive errors and the number of timeout errors which have occurred since the most recent restart as shown in Table G-2. The counts are in binary.

Table G-2 Receive Errors and Timeout Errors for STW145 and STW146

<b>Receive Errors</b>		<b>Timeout Errors</b>	
Channel	Status Word	Channel	Status Word
1	STW145	1	STW146

---

**NOTE:** A typical system should have no more than one detected and corrected error over the I/O link per 20,000 scans. If this error rate is exceeded, it may indicate a possible wiring or noise problem. Three consecutive errors to an RBC causes the base to be logged off and the corresponding bit in STW2 to be set.

---

STW147:  
PROFIBUS-DP Slave  
Errors

Status word STW147 records the number of times, since the most recent restart, that PROFIBUS-DP slaves have failed to respond to a request from the Series 505 CPU.

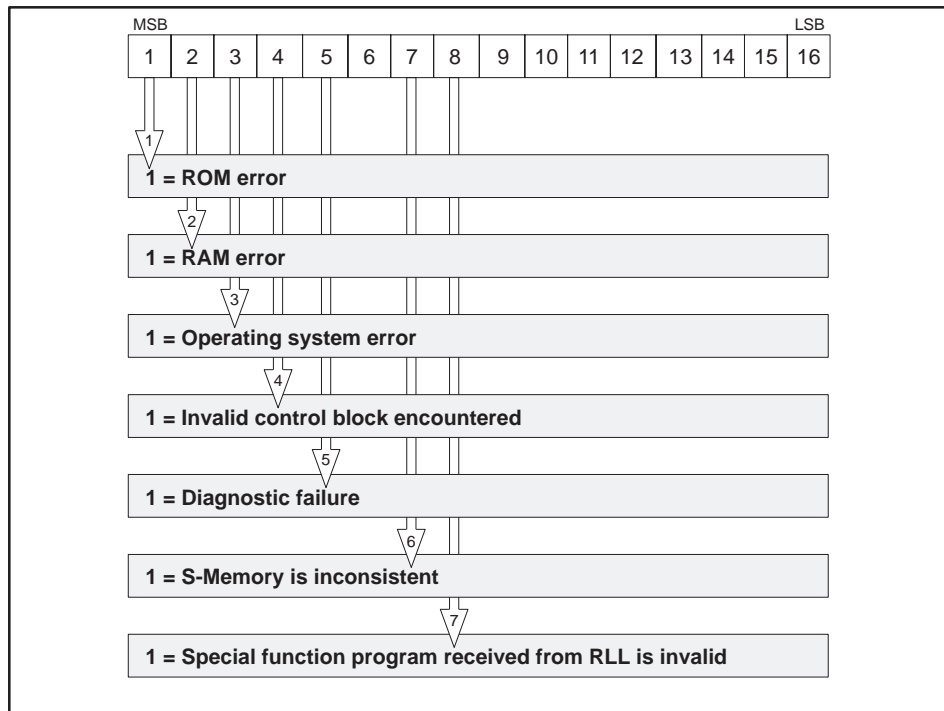
STW148:  
PROFIBUS-DP Bus  
Communication  
Errors

Status word STW148 indicates the number of times, since the most recent restart, that the PROFIBUS-DP I/O channel has experienced a loss of token, possibly due to a problem with the cable. Such errors generally cause total failure for the PROFIBUS-DP I/O link.

STW149 - STW160:  
Reserved

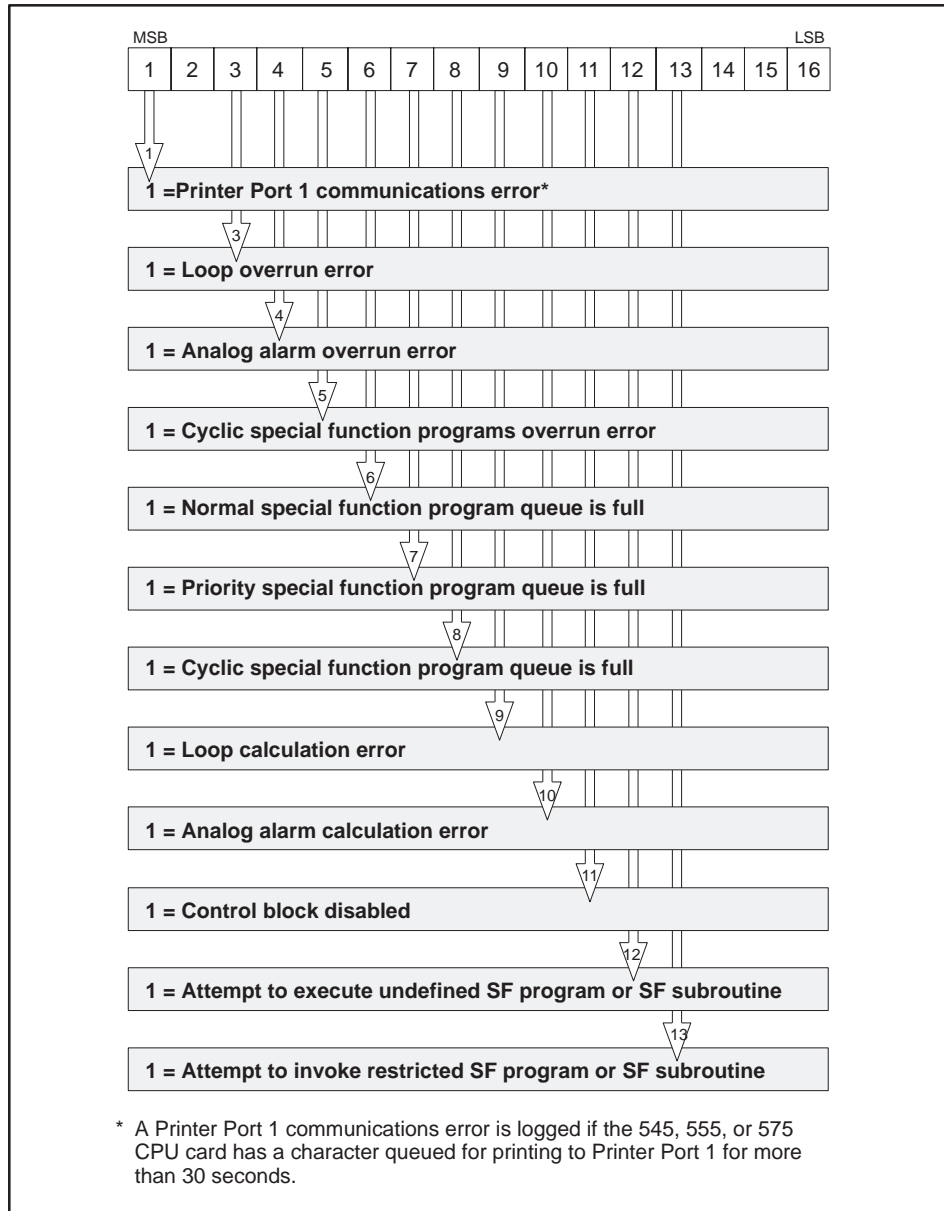
Reserved.

**STW161:**  
**Special Function**  
**Processor Fatal**  
**Errors**



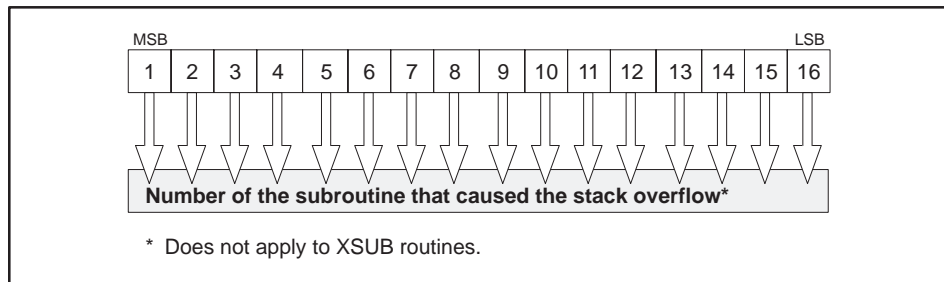
**Applicable Controller**    ALL

**STW162:  
Special Function  
Processor  
Non-fatal Errors**



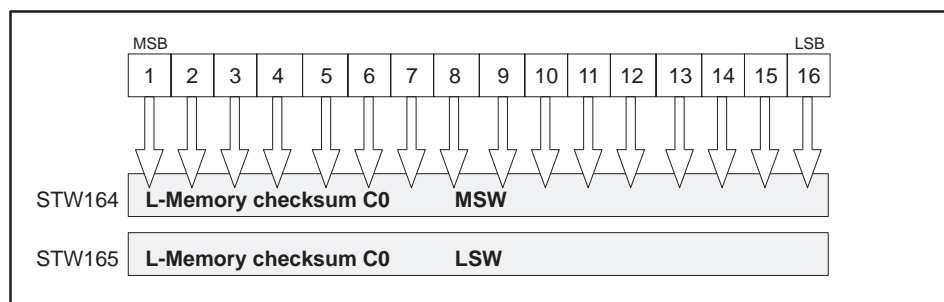
**Applicable Controller**    ALL

**STW163:**  
**RLL Subroutine**  
**Stack Overflow**



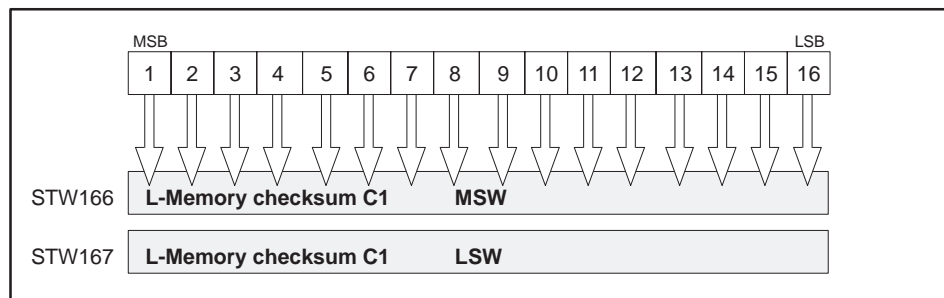
**Applicable Controller** ALL

**STW164 – STW165:**  
**L-Memory**  
**Checksum C0**



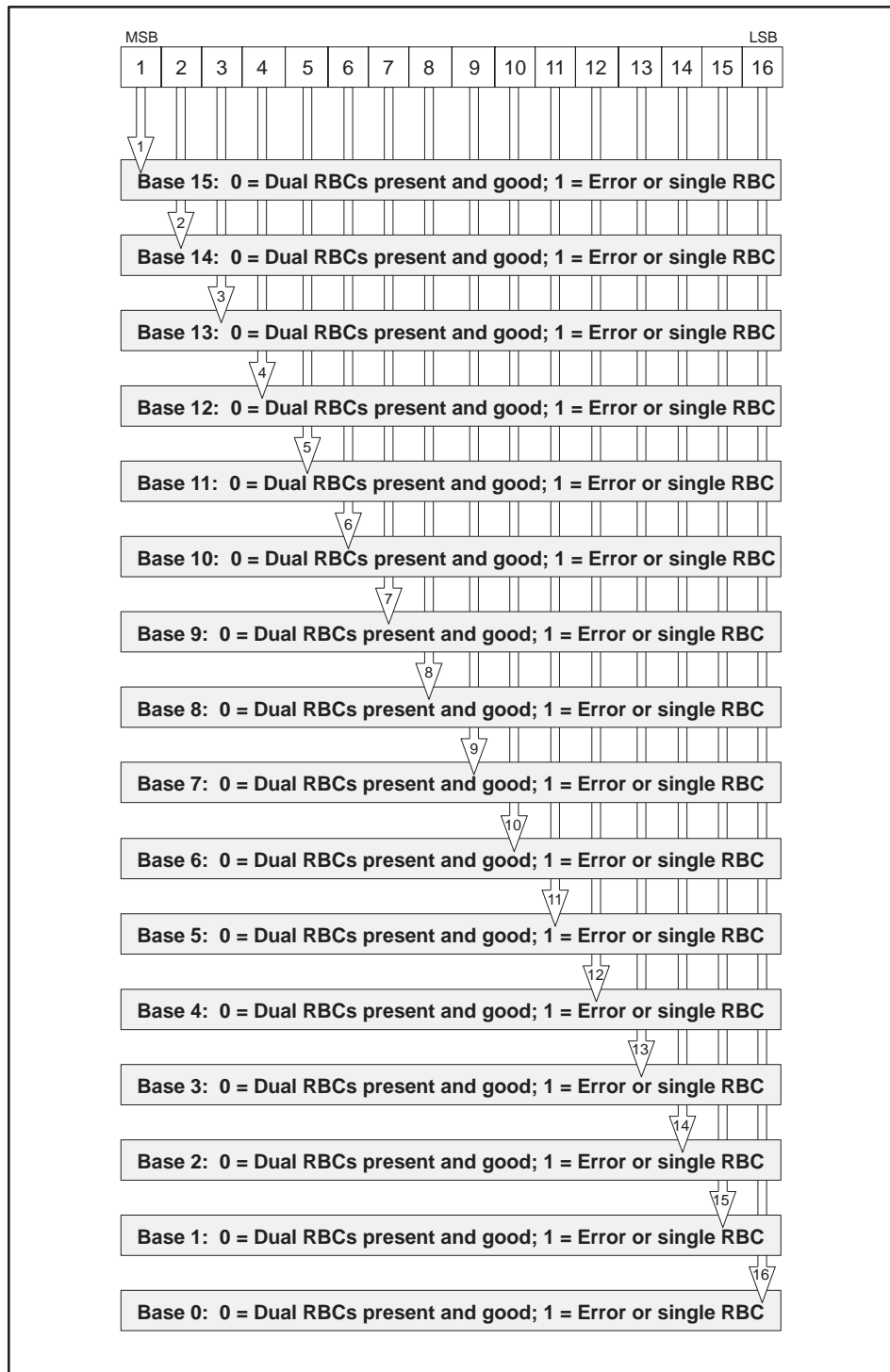
**Applicable Controller** ALL

**STW166 – STW167:**  
**L-Memory**  
**Checksum C1**



**Applicable Controller** ALL

STW168:  
Dual RBC Status

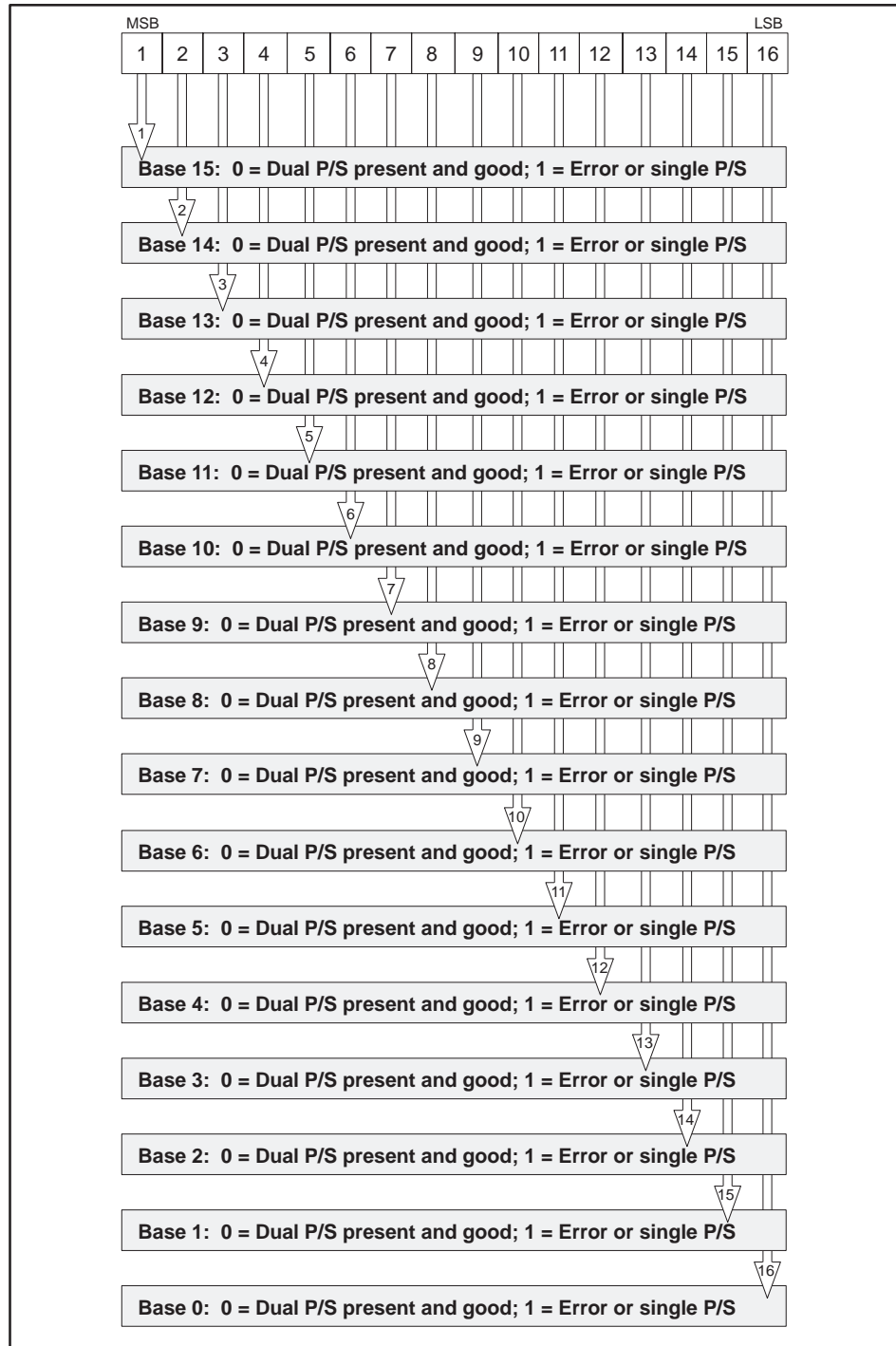


Applicable Controller ALL

STW169 – STW175:  
Reserved

Reserved.

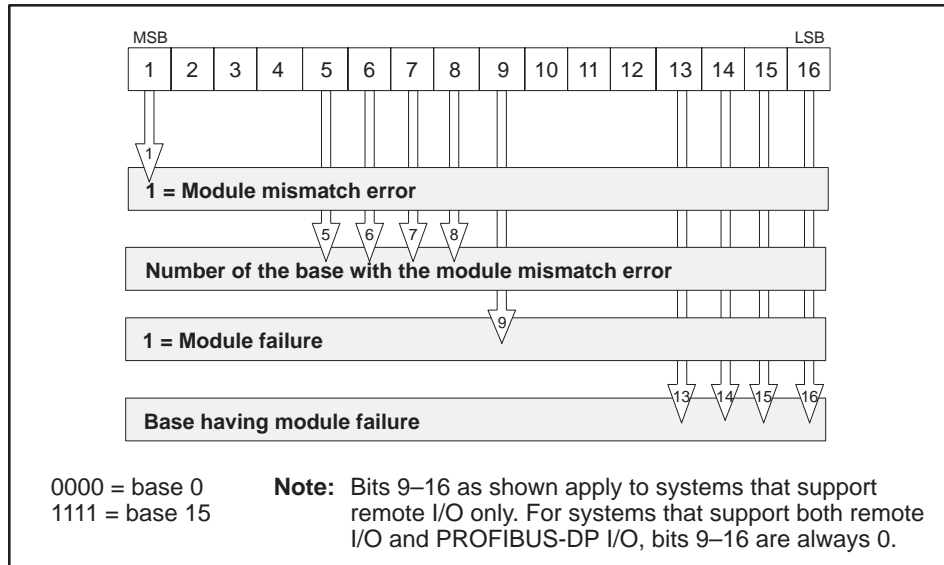
STW176:  
Dual Power Supply  
Status



Applicable Controller ALL

STW177 – STW183: Reserved.

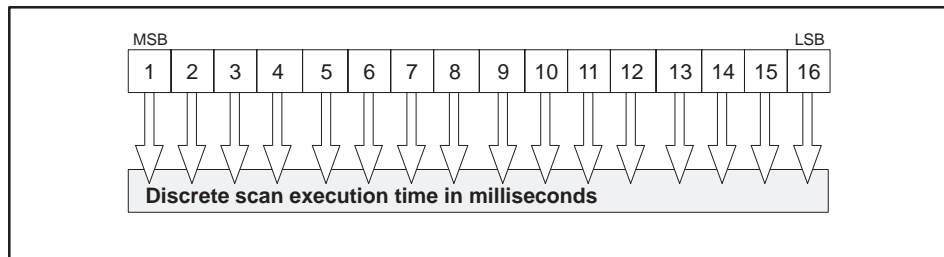
STW184:  
Module Mismatch  
Indicator



**Applicable Controller** ALL

STW185 – STW191: Reserved.

STW192:  
Discrete Scan  
Execution Time

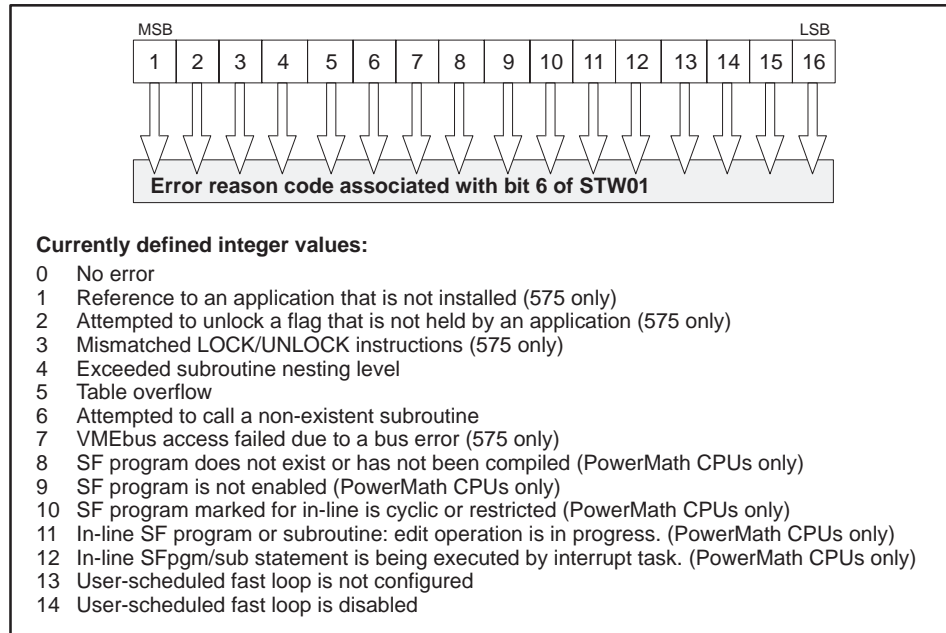


**Applicable Controller** ALL

STW193 – STW199: Reserved.



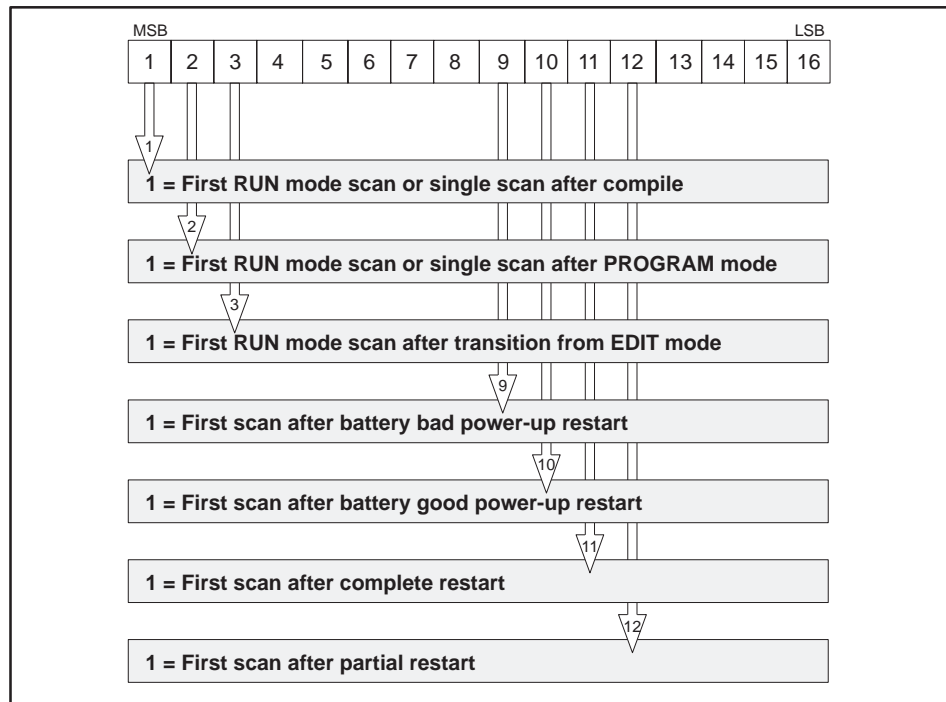
**STW200:**  
User Error Cause



**Applicable Controller** ALL (except as noted)

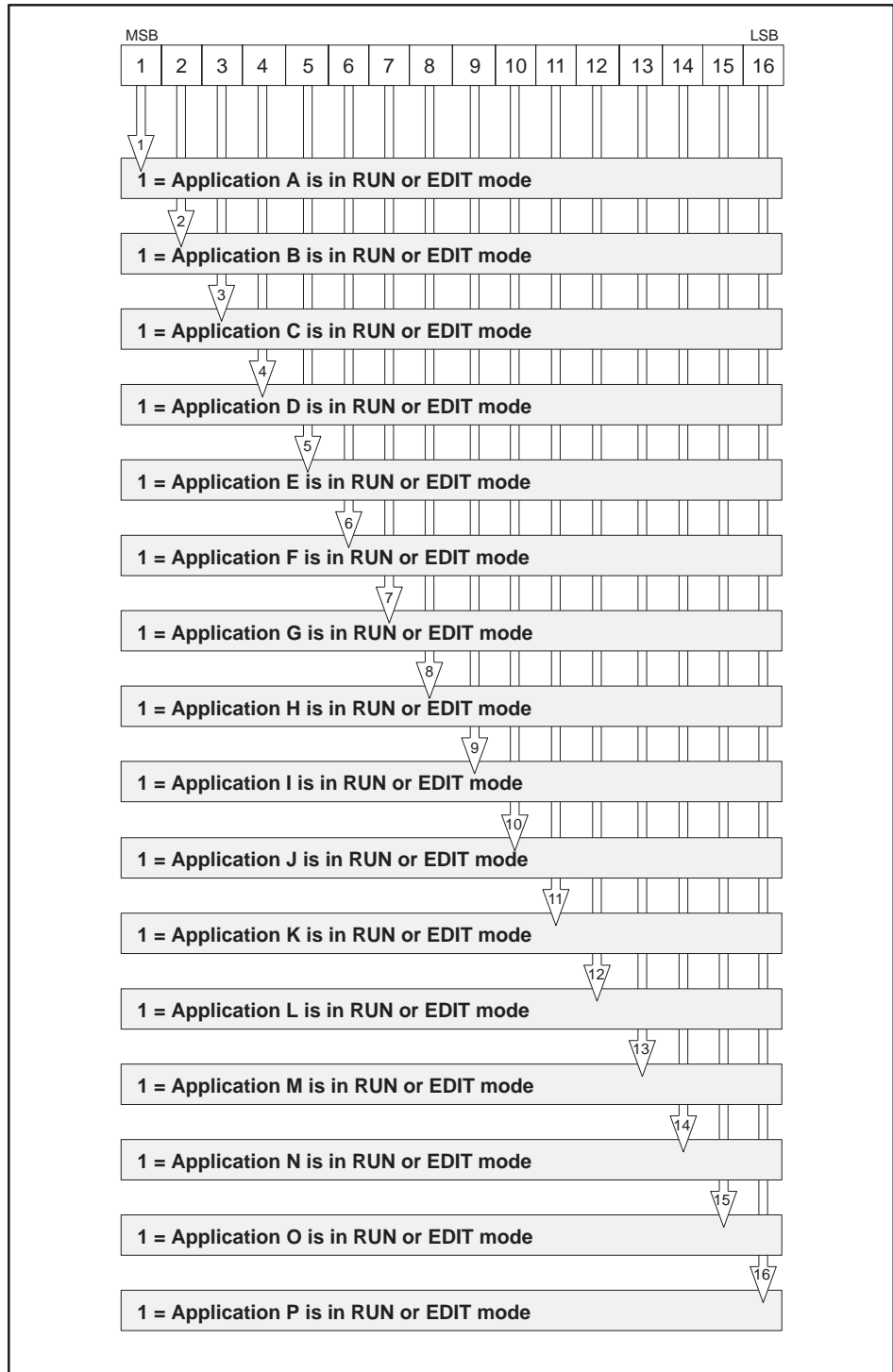
STW200 reports the first error that occurs in a given scan of the RLL program. After you correct the problem that causes the error, recompile and run the program again. If there is a second problem, the error code for this problem is recorded in STW200. Subsequent errors are recorded accordingly.

STW201:  
First Scan Flags



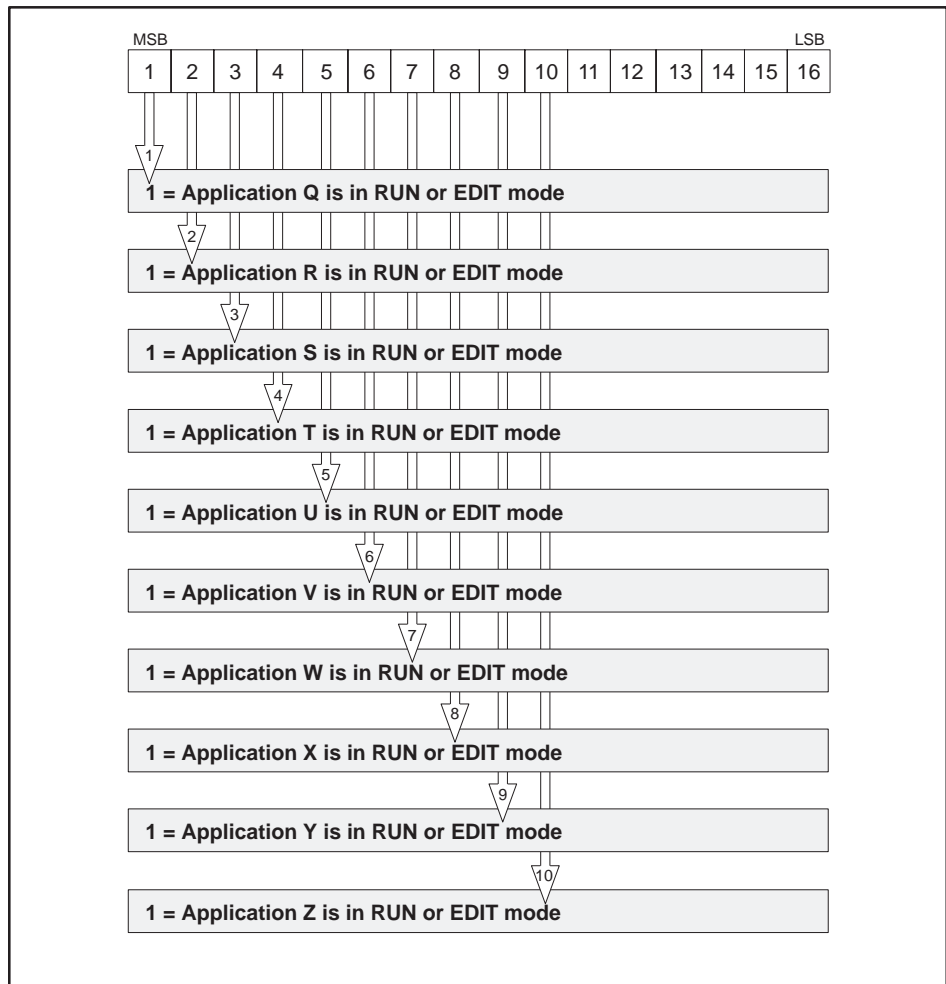
Applicable Controller ALL

STW202:  
Application Mode  
Flags (A – P)



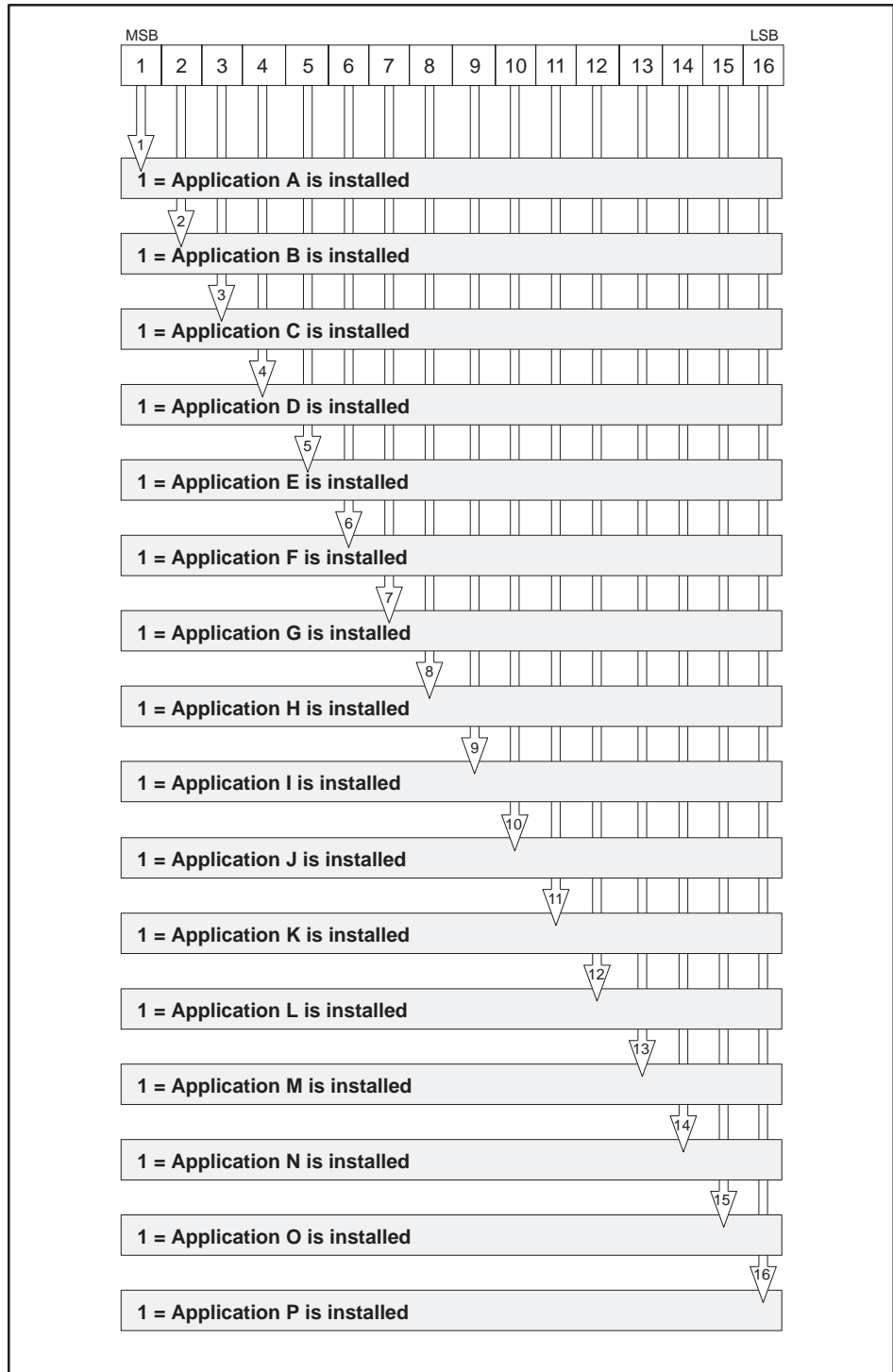
Applicable Controller 575

**STW203:**  
**Application Mode**  
**Flags (Q - Z)**



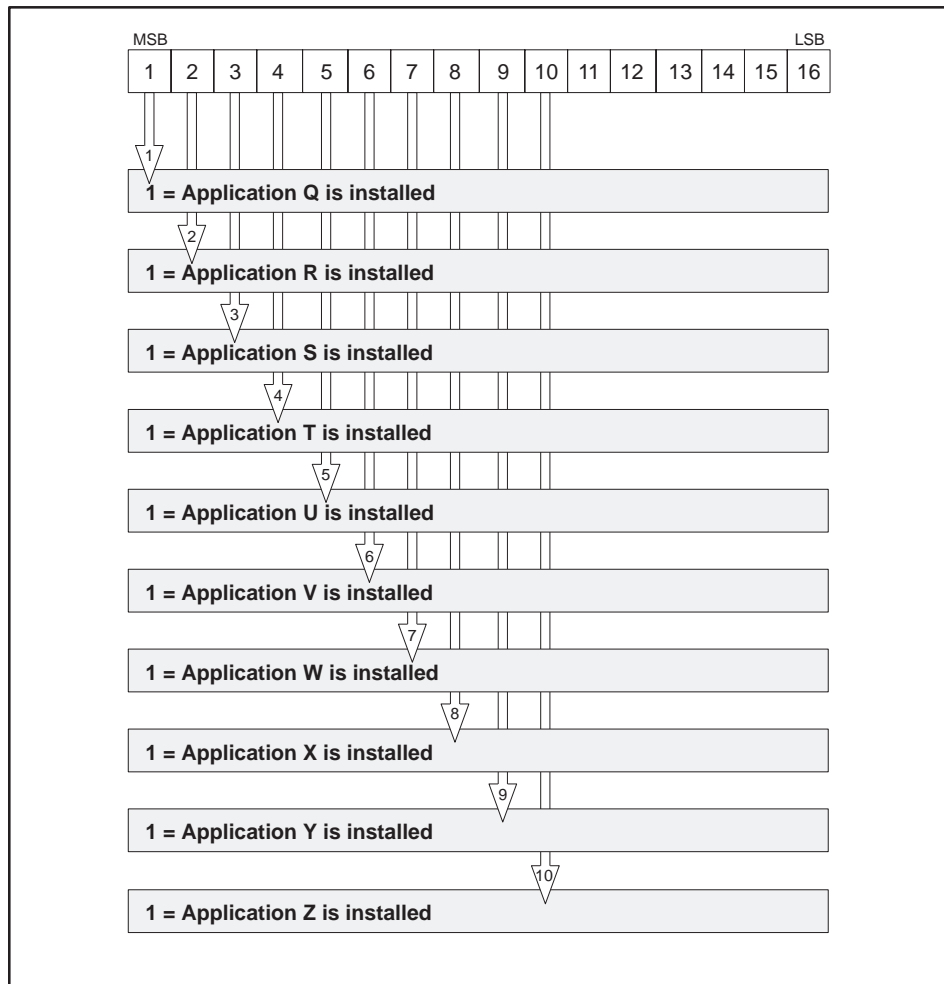
**Applicable Controller** 575

STW204:  
Application  
Installed Flags  
(A – P)



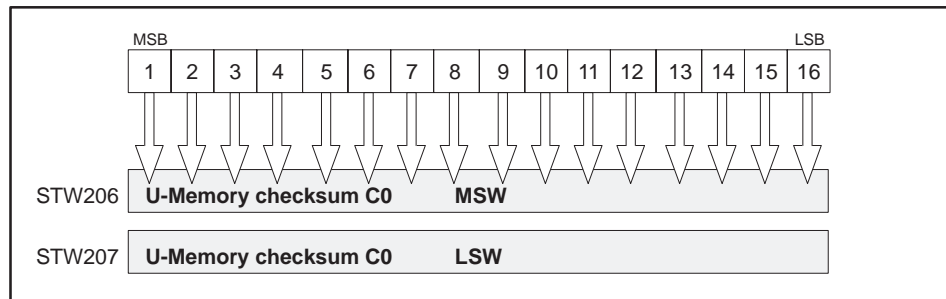
Applicable Controller 575

STW205:  
Application  
Installed Flags  
(Q – Z)



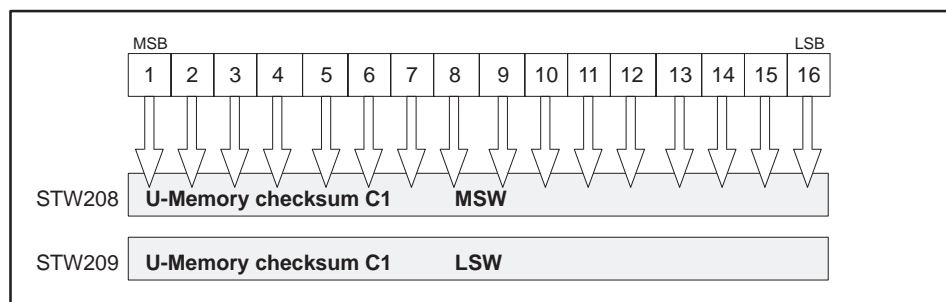
Applicable Controller 575

STW206 – STW207:  
U-Memory  
Checksum C0



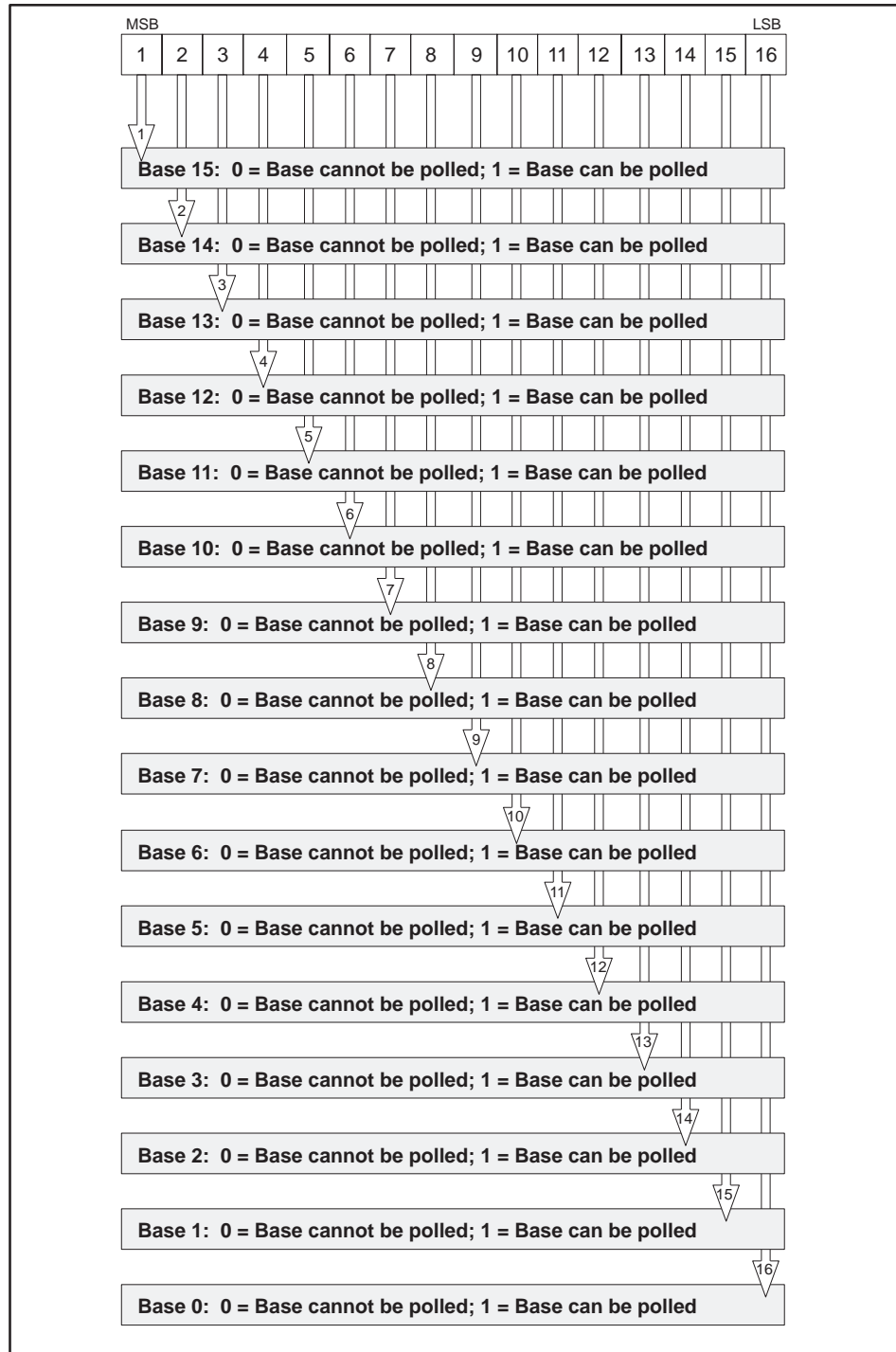
**Applicable Controller** ALL

STW208 – STW209:  
U-Memory  
Checksum C1



**Applicable Controller** ALL

**STW210:  
Base Poll Enable  
Flags**



**Applicable Controller**    ALL

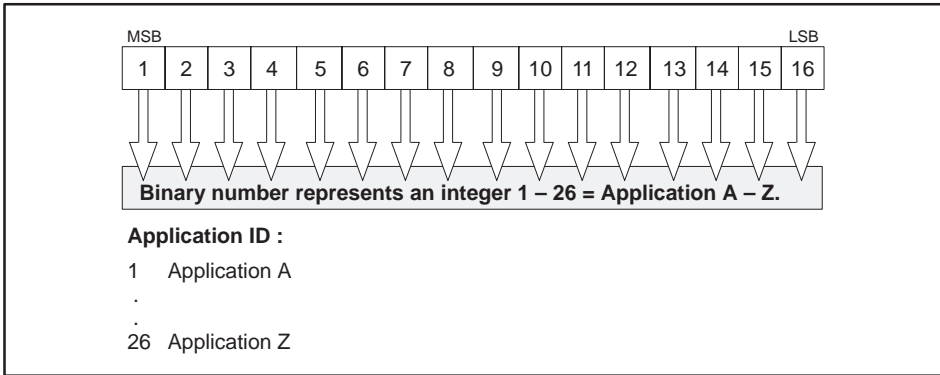


**STW211 – STW217:  
PROFIBUS-DP Slave  
Enable Flags**

Bit	MSB															LSB
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
STW211	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
STW212	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
STW213	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
STW214	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
STW215	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
STW216	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
STW217	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97

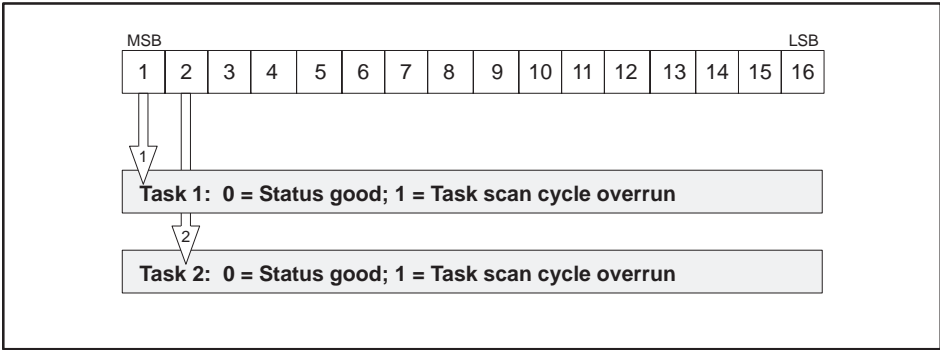
STW211 through STW217 indicate the enable status for PROFIBUS-DP slaves. The slave's bit, as indicated in the cells above, is a 1 if the slave is defined and enabled.

**STW218:  
My Application ID**



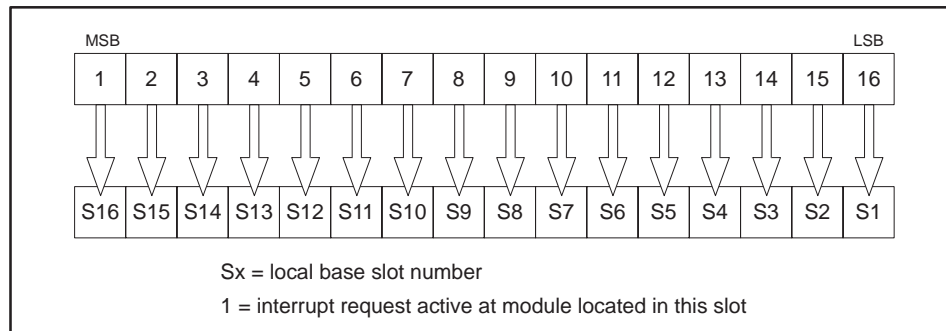
**Applicable Controller** 575

**STW219: RLL Task  
Overrun**



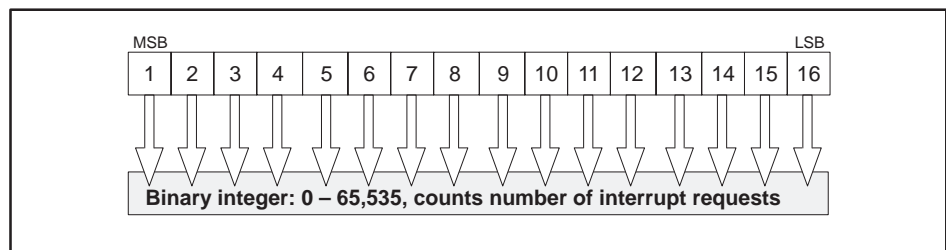
**Applicable Controller** ALL

**STW220:  
Interrupting Slots in  
Local Base**



**Applicable Controller**    545 and 555

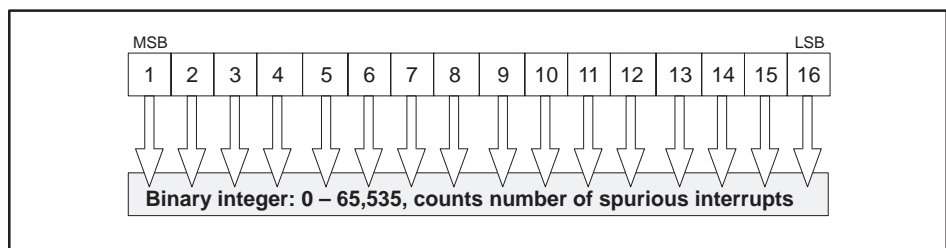
**STW221: Module  
Interrupt Request  
Count**



STW221 is a 16-bit integer (0 – 65,535) that is incremented each time an interrupt request is received from any interrupting module installed in the local base.

**Applicable Controller**    545 and 555

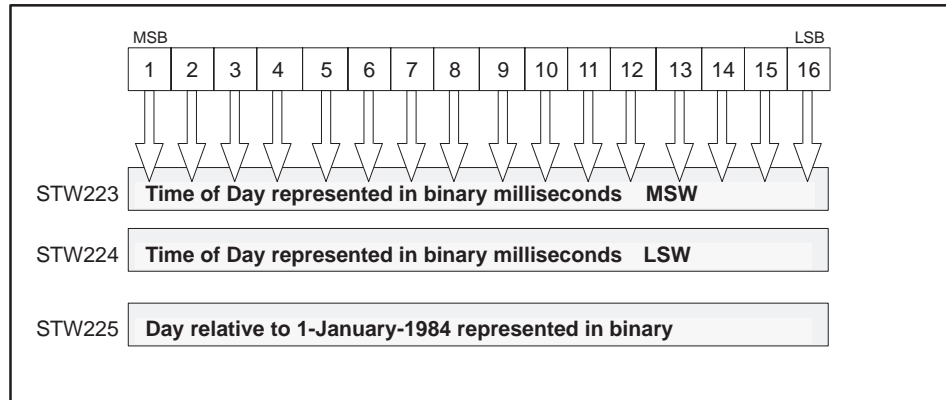
**STW222: Spurious  
Interrupt Count**



STW222 is a 16-bit integer (0 – 65,535) that is incremented each time a spurious interrupt occurs. A spurious interrupt is a VMEbus interrupt that is removed before the 575 can acknowledge it.

**Applicable Controller**    575

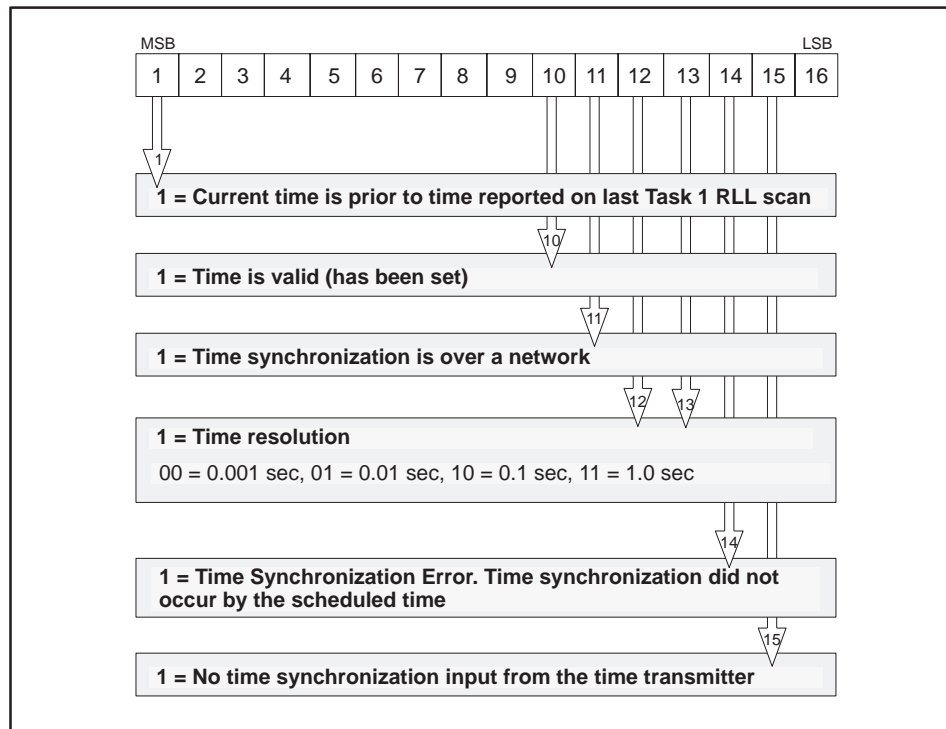
**STW223 – STW225:  
Binary Time-of-Day**



**Applicable Controller** ALL

STW223 and STW224 contain a 32-bit binary representation of the relative millisecond of the current day. STW225 contains a 16-bit binary representation of the current day relative to 1-January-1984, (day 0). See also the following description of Time-of-Day Status for STW226.

**STW226:  
Time-of-Day Status**



**Applicable Controller** ALL

---

STW226 contains a 16-bit representation of the Time-of-Day status.

If you use the time update feature of the SINEC H1 Communications Processor (PPX-505-CP1434TF), you should consider the following in specifying the communications processor's update time interval.

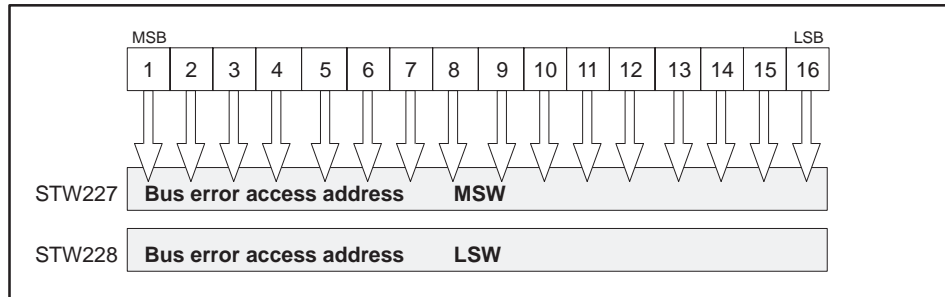
- Time updates from the communications processor result in the controller's time of day clock being written with the new time value. This results in a minor (<1 ms) scan time extension on the scan in which the update occurs.
- Between time updates, the time of day is reported based on the controller's time of day clock. This clock may drift (lose or gain time) relative to the SINEC H1 time source. Because of this time drift, the time reported on the controller scan following a time update from the communications processor may be before the time reported on the previous controller scan. Time status (STW226) bit 1 will indicate this occurrence.

---

**NOTE:** The programmable controller hides negative (to the past) clock changes due to time synchronization if the change is less than 50 ms. For such a change, the controller freezes the time of day until the updated time catches up to the controller's time when the update was received.

---

**STW227 – STW228:  
Bus Error Access  
Address**

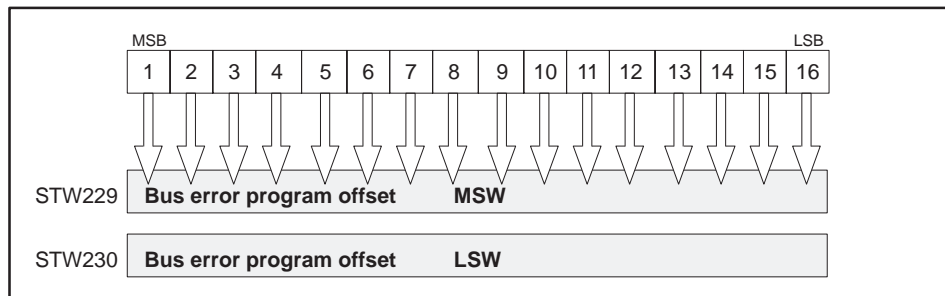


**Applicable Controller**    575

STW227 and STW228 contain a 32-bit binary representation of the VMEbus address of the first data access that was aborted due to a bus time out. Use them with STW1, STW200, and STW229-230 to diagnose user programming errors on a 575 system.

**NOTE:** For the 575, the most significant 8 bits of the VMEbus address are 01<sub>16</sub> for a normal (VMM) address space access, or F0<sub>16</sub> for a short (VMS) address space access. The remaining 24 bits of the address contain the address space offset.

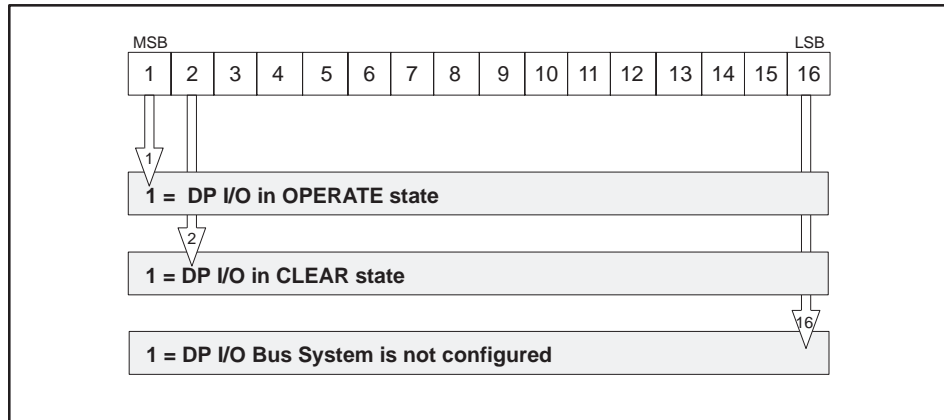
**STW229 – STW230:  
Bus Error Program  
Offset**



**Applicable Controller**    575

STW229 and STW230 contain a 32-bit binary representation the program offset. If a VMEbus access was aborted while executing an XSUB routine, these status words contain the U-Memory offset of the instruction that caused the aborted VMEbus access. Use them with STW1, STW200, and STW227-228 to diagnose user programming errors on a 575 system.

**STW231  
PROFIBUS-DP I/O  
System Status**

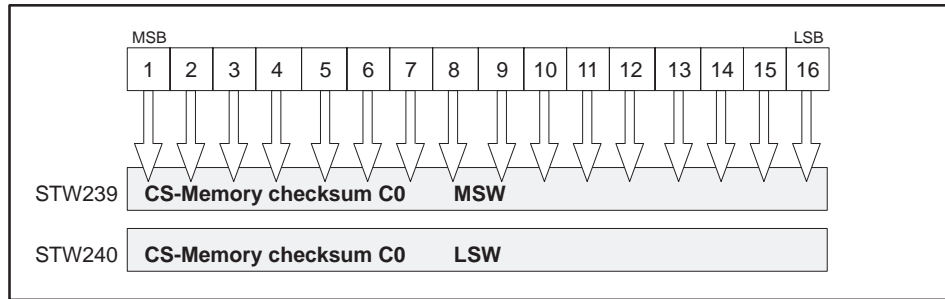


**STW232 – STW238:  
PROFIBUS-DP Slave  
Diagnostic**

Bit	MSB															LSB
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
STW232	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
STW233	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
STW234	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
STW235	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
STW236	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
STW237	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
STW238	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97

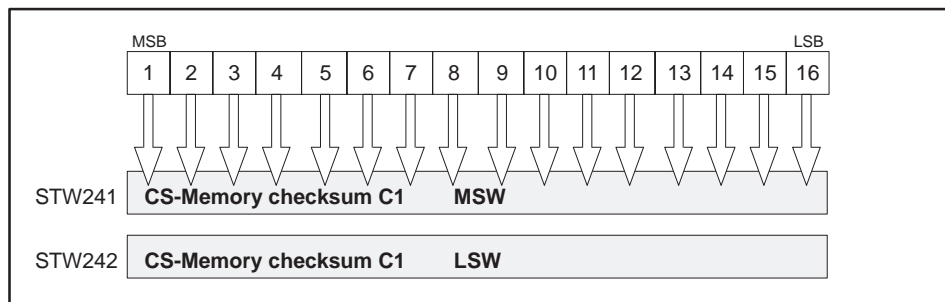
STW232 through STW238 indicate the PROFIBUS-DP slaves that have signaled a diagnostic that has not been read by an RSD instruction (see page 6-120). The slave's bit, as indicated in the cells above, is a 1 if a diagnostic has been signaled and not yet read.

STW239 – STW240:  
CS-Memory  
Checksum C0



**Applicable Controller**    PowerMath CPUs only

STW241 – STW242:  
CS-Memory  
Checksum C1



**Applicable Controller**    PowerMath CPUs only

# External Subroutine Development

---

<b>H.1</b>	<b>Designing the External Subroutine</b> .....	<b>H-2</b>
	Program Code Requirements .....	H-2
	Loading the Subroutine .....	H-3
<b>H.2</b>	<b>U-Memory Format</b> .....	<b>H-4</b>
	Header .....	H-4
	Code and Constant Data .....	H-5
	Modifiable Data .....	H-5
	User Stack .....	H-5
<b>H.3</b>	<b>Guidelines for Creating C Language Subroutines</b> .....	<b>H-6</b>
	Debugging the External Subroutine .....	H-6
	Static Data Initialization .....	H-7
	Accessing Discrete/Word Variables .....	H-10
	Floating Point Operations .....	H-11
	Unsupported C Language Features .....	H-11
<b>H.4</b>	<b>Developing an External Subroutine — Example</b> .....	<b>H-12</b>
	Example Header File .....	H-12
	Example Subroutine Source .....	H-14
	Preparing the Load Module .....	H-14
	Loading U-Memory .....	H-16
	Using the External Subroutines in RLL .....	H-16



### **WARNING**

When you call an external subroutine, the built-in protection features of the controller are by-passed. Use care when you test the external subroutine before introducing it to a control environment.

Failure to do so may cause undetected corruption of controller memory and unpredictable operation by the controller, which could result in death or serious injury to personnel, and/or damage to equipment.

You must take care in testing the external subroutine before introducing it to a control environment.

#### Program Code Requirements

Follow these rules when you develop an external subroutine.

- Use a compiler, such as Microtec® MCC68K, that allows generation of position independent code targeted as follows:

For the 545/555: Motorola® 68020.

For the 575: Motorola 68020 and optionally, the 68881 or the 68882 floating-point processor, if installed.

The object code must be position-independent, i.e., it must use PC-relative addresses for all references to programs and data contained in U-Memory.

- Link all subroutines into one downloadable load module in Motorola S-Record format. The resulting file must conform to the format specified in Section H.2.
- To help ensure that the subroutine interacts correctly and safely with the controller program, follow the guidelines in Section H.3 as you develop an external subroutine for your application.

---

## Loading the Subroutine

Follow these steps to prepare the external subroutine for use in the controller.

1. Compile/assemble the subroutines and header to create object modules.
2. Link the object modules for the header and subroutines to create the load module. The file name must have the extension `.rec`. The output must have the header at zero followed by the code and data constants, then the variables, and finally the stack.
3. Use TISOFT to configure U-Memory, if you have not already done so.
4. Use the TISOFT Convert S-Records option (AUX 40) to import the linked program into the TISOFT environment.
5. Use the TISOFT Load U-Memory option (AUX 43) to load the file created in step 4 into U-Memory.

An example application illustrating this process is given in Section H.4.

## H.2 U-Memory Format

---

External subroutines are stored in U-Memory. U-Memory consists of four logical segments, described below, and illustrated in Figure H-1.

### Header

The header contains the following data elements, that must be defined in the order specified.

**E/Version** This 16-bit word contains two data elements:

- Bit 1 (the MSB) specifies the error action taken in the event of a bus error while accessing the VMEbus in an XSUB routine on a 575. If bit 1 is a 0 and a VMEbus error occurs while processing an XSUB, the controller enters the Fatal Error mode. If bit 1 is a 1, and a VMEbus error occurs while processing an XSUB, the XSUB's execution terminates, bits 6 and 11 of STW1 are set, and STW200 is set to 7 (if this is the first error encountered on this RLL scan). The controller stays in RUN mode. This bit is ignored by 545 and 555.
- Bits 2 through 16 specify the header version number. It must have a binary value of 1 or the U-Memory load operation (TISOFT AUX function 43) fails.

**Num\_XSUBs** Specifies the number of external subroutines defined in the load module. This element is a 16-bit word.

**Data\_Start** Specifies the U-Memory offset for the beginning of the modifiable data area. This element is a 32-bit long word.

**Stack\_Start** Specifies the U-Memory offset to the lowest U-Memory location available for use as a run-time stack. The block of memory from this location to the end of U-Memory is available to the main RLL task (TASK1) during XSUB calls. This element is a 32-bit long word.

---

NOTE: For an XSUB called by the cyclic or interrupt RLL tasks (TASK2 or TASK8), the stack is allocated by the operating system and is relatively small ( approximately 500 bytes).

---

**Stack\_Size** Specifies the minimum number of bytes that must be available for use as the run-time stack area when an external subroutine is called by the main RLL task. This element is a 32-bit long word.

**Entry\_Points** This is a table containing  $n$  32-bit elements, where  $n$  equals the number of subroutines, as specified in Num\_XSUBs. Each element in this table specifies the U-Memory offset for the entry point of each of the subroutines 1 through  $n$ , respectively. A value of 0 indicates that the specified subroutine is not present.

Code and Constant Data

The code and constant data area immediately follow the header area. This area consists of position-independent, invariant machine code, and data constants.

Modifiable Data

The modifiable data area follows the code and constant data area and contains the static variables used by the subroutines.

User Stack

The user stack follows the modifiable data area in U-Memory. The size of the user stack depends upon the configured size of U-Memory and how much memory is used by the header, the code and constant data, and the modifiable data areas. The user stack starts at the last location of U-Memory and grows downwards, toward the address specified by Stack\_Start. Stack\_Size specifies the minimum size of this area.

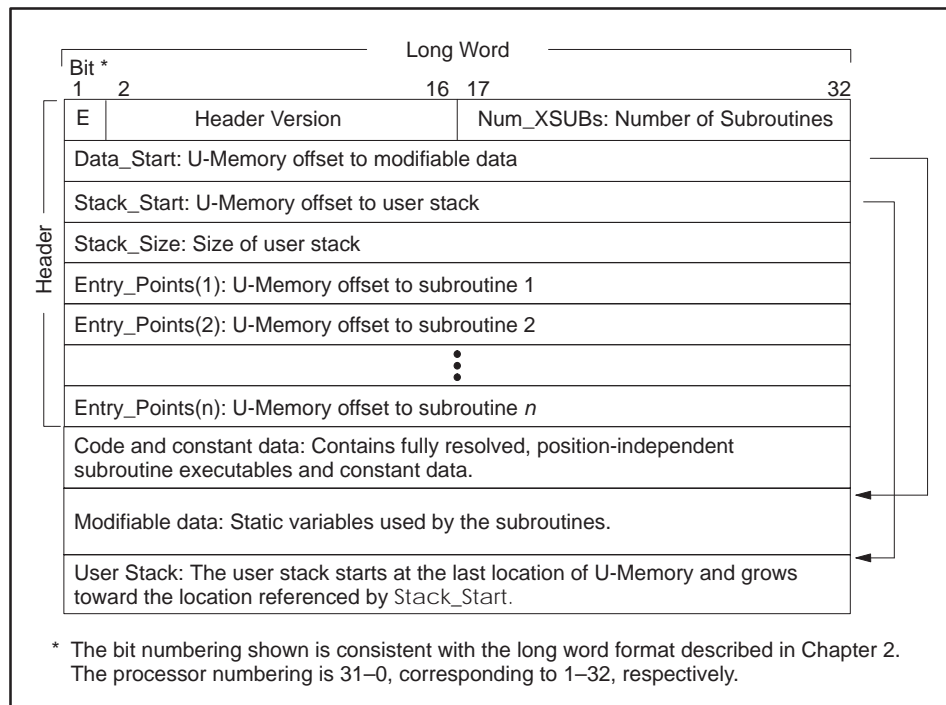


Figure H-1 Externally Developed Subroutine Code Format

**NOTE:** When U-Memory is loaded, the system verifies that sufficient U-Memory is configured to hold the header, code, data, and stack. The load is rejected unless there is enough memory. A subsequent attempt to reconfigure loaded U-Memory to a size less than the sum of header, code, data, and stack is also rejected.

## H.3 Guidelines for Creating C Language Subroutines

---

The guidelines in this section can answer some questions that may arise as you develop your code. These guidelines assume that you are using the Microtec MCC68K tool set. Version 4.2A of this compiler has been tested in a limited number of 545 and 575 applications and has been verified to generate code that reliably runs on these machines. MCC68K runs on IBM compatible personal computers, as well as a number of minicomputers and work stations. The MCC68K tool set is available from:

Microtec Research, Inc.  
2350 Mission College Blvd.  
Santa Clara, CA 95054  
Toll Free 800.950.5554

If you are using a different compiler, you need to make changes in these guidelines to fit that compiler's requirements.

### Debugging the External Subroutine

Facilities for debugging external subroutines on the controller are very limited. It is strongly recommended that you develop and test your external subroutines using a native compiler on your development computer. A number of C compilers are available commercially for this purpose, including Quick C® and Turbo C® for the MS-DOS® environment.

Before coding the external subroutine, be aware that compiler differences may exist between the native compiler on the development computer and the MCC68K compiler. A native compiler, designed for use on a general purpose system, e.g., the IBM PC/AT, usually has a larger set of runtime facilities than does a compiler like MCC68K, that is designed for embedded systems. If you use these facilities, they will not exist when you port your external subroutines to the controller.

After you have written and debugged your subroutines on the development computer, you must port the debugged subroutines to the controller. If you avoid architectural features of the development machine, and if you have not used runtime elements from the native compiler that are not present in MCC68K, then this is a straight forward procedure.

Before attempting to control an actual process, always check the subroutine in a test environment (on a controller that is not connected to a factory-floor process) to verify that the subroutine and controller program operate as expected.

---

## Static Data Initialization

In C, variables declared outside of functions or declared with the static attribute are initialized when the program starts, just before entering the main procedure. When you write external subroutines you do not have a main procedure and the normal initialization does not occur. Therefore, you need to assign one of your subroutines to perform the C initialization function. This subroutine must be called from the main RLL task whenever your application is (re)started, e.g., at power-up or a transition from PROGRAM to RUN mode.

Assembly subroutine `vinit.src` \* (Figure H-2) contains the necessary initialization routine for version 4.2A of MCC68K. Include the initialization subroutine as `XSUB1` in all U-Memory load modules. You should call `XSUB1` whenever your RLL performs its startup initialization. Subroutine `_vinit (XSUB1)` must be called before any static variables are referenced by your external subroutines.

\* The VINIT routine is based in part on INITCOPY.C, Copyright (C) 1990, Microtec Research, Inc.

## Guidelines for Creating C Language Subroutines (continued)

```
TTL      _VINIT -- Initialize Static Variables
OPT      CASE

*=====
* function _vinit -- Initialize Static Variables.
*
* * SYNOPSIS:
*
*     void _vinit (long int * code);
*
* where code is:
*
*     0  if initialization was successful.
*     1  if initialization failed due to invalid start code in ??INITDATA.
*     2  if initialization failed due to unknown flag byte in a copy
*         specification.
*
* * DESCRIPTION:
*
* This function may be called as an external subroutine in order to
* initialize all static variables declared by the U Memory load module.
*
* Subroutine _vinit zeroes all "zerovars" variables and sets all "vars"
* variables to their specified initial values. A "zerovars" variable
* is a variable declared using statements of the form:
*
*     [static] int var;
*
* A "vars" variable is a variable declared using statements of the form:
*
*     [static] int var = 5;
*
* External subroutine _vinit must be called before any other external
* subroutine. It should be called once whenever the RLL process is
* started or restarted.
*
* * ASSUMPTIONS:
*
* This subroutine assumes the Microtec C compiler has been used to create
* the objects comprising the load module and that the Microtec linker
* has been used with (at least) the "INITDATA vars" command. It also
* assumes that the U Memory header is the first element of section "const"
* and is located at U Memory offset 0.
*
* The structure of section ??INITDATA (created by the Microtec linker) is
* as follows. Byte 0 contains an 'S' indicating start of ??INITDATA.
* This byte is followed by zero or more copy specifications (see below).
* The last copy specification is followed by an 'E' indicating the end
* of the ??INITDATA section.
*
* A copy specification has four fields:
*
* flag      One byte containing a 'C', indicating start of copy specification;
* length    Four bytes containing the length (in bytes) of the <data> field;
* dest      Four bytes containing the starting U-Memory offset to which the
*           <data> field is to be copied;
* data      The values to be copied to <dest>. The length of this field is
*           specified by the <length> field.
*=====
```

Figure H-2 Initialization Routine Required for Microtec C

```

**** Initialize the 'zerovars' section to all 0.
*
SECTION code,,C int __vinit ()
XDEF __vinit {
__vinit LEA .startof.(zerovars)(PC),A0 ptr1 = address of zerovars;
MOVE.L #.sizeof.(zerovars),D0 length = size of zerovars;
BRA.S LOOP00S while (--length >= 0) {
CLR.B (A0)+ *ptr1 = 0;
LOOP00 SUBQ.L #1,D0 }
LOOP00S BHS LOOP00 .

**** Copy initial values from the ??INITDATA section (constructed by the
* linker due to the INITDATA command) to the appropriate destination
* address.
*
MOVE.L #.sizeof.(??INITDATA),D0 if (??INITDATA not empty)
SUBQ.L #2,D0 {
BLO.S ENDIF10 .
LEA .startof.(??INITDATA)(PC),A0 ptr1 = address of ??INITDATA
CMPI.B #'S',(A0)+ error if (*ptr1++ != 'S')
BNE.S ERROR1 .
LOOP20 MOVE.B (A0)+,D0 while ((t = *ptr1++) != 'E')
CMPI.B #'E',D0 {
BEQ.S ELOOP20 .
CMPI.B #'C',D0 error if (t != 'C')
BNE.S ERROR2 .
MOVE.L (A0)+,D0 length = *((long *) ptr1);
LEA .startof.(const)(PC),A1 ptr2 = address_of (header)
ADDA.L (A0)+,A1 + *((long *) ptr1);
BRA.S LOOP30S while (--length >= 0) {
LOOP30 MOVE.B (A0)+,(A1)+ *dest++ = *source++;
LOOP30S SUBQ.L #1,D0 }
BHS LOOP30 .
BRA LOOP20 }
ELOOP20 EQU * .
ENDIF10 EQU * }
MOVEQ #0,D0 code = no error;

**** Return the value of <code> to the user.
*
GOBAK MOVEA.L 4(SP),A0 return (code);
MOVE.L D0,(A0) .
RTS .

**** Error handlers:
*
ERROR1 EQU * error1:
MOVEQ #1,D0 code = no starting point;
BRA GOBAK return (code);

ERROR2 EQU * error2:
MOVEQ #2,D0 code = unknown flag byte;
BRA GOBAK return (code);
END ! };

```

Figure H-2 Initialization Routine Required for Microtec C (continued)



## Guidelines for Creating C Language Subroutines (continued)

---

### Accessing Discrete/Word Variables

As specified in Section 6.81, the calling conventions used by the XSUB instruction always pass 32-bit values or pointers to the external subroutine.

When passing a discrete value, e.g., IN X5, the on/off state of the parameter is in the least significant bit of the 32-bit value. Other bits are unspecified. The example in Figure H-3 shows one way to isolate the actual value of the discrete parameter.

```
void sub1 (long int D, ...)
{
    unsigned char D_value;
    D_value = D & 0x1;
    ...
}
```

Figure H-3 Example of Passing a Discrete Value

When passing a pointer to a discrete variable, e.g., IO X5, you must declare the data type of the parameter as an unsigned char pointer. The discrete value is in the least significant bit of the 8-bit value addressed by the pointer. Refer to the example in Figure H-4.

```
void sub2 (unsigned char *D, ...)
{
    if (*D & 0x1)
        ... handle case where parameter is on (true) ...
    else
        ... handle case where parameter is off (false)
}
```

Figure H-4 Example of Passing a Pointer

When passing a normal value, e.g., IN V103, the value is assumed to occupy a long word (V103 and V104). If only a word is required, you must include code to isolate this word from the most significant 16 bits of the value. See the example in Figure H-5.

```
void sub3 (long int V, ...)
{
    short int V_value;
    V_value = V >>16;
    ...
}
```

Figure H-5 Example of Passing Normal Values

When passing a pointer to a normal variable, e.g., IO V15, you control the data element type since you completely declare the data type in your C Language function.

 <b>CAUTION</b>
<p>For the 575 controller, word image register values can only be accessed as words or long words.</p> <p>If you access a word image register location as a byte (8 bits), the result is unspecified, and could cause damage to equipment,</p> <p>Ensure that you always use words or long words with the 575 controller.</p>

**NOTE:** The controller allows pointers to read-only variables (STW, K, X) to be passed to external subroutines. It is recommended that you not design the subroutine to alter the content of these variables since other instructions assume that the content does not change.

**Floating Point Operations**

The controller uses single precision floating-point math. The default type for floating-point constants and operations in the MCC68K compiler is double precision. On the 545, 555, and 575 without a math coprocessor, you may want to avoid the overhead associated with double precision math. Refer to your compiler's documentation for instructions for forcing single precision math.

**Unsupported C Language Features**

Do not use operating system-dependent language elements in external subroutines. This includes the C Language runtime routines listed below. Refer to your compiler's documentation for possible additional OS-dependent runtime.

clearerr	feof	fopen	fsacnf	lseek	puts	ungetc
close	ferror	fprintf	fwrite	open	read	
create	fflush	fputc	getc	printf	setbuf	
_exit	fgetc	fputs	getchar	putc	scanf	
fclose	fgets	fread	gets	putchar	sprintf	

## H.4 Developing an External Subroutine — Example

This section illustrates the creation of a U-Memory load module that defines two external subroutines: `long_add` (XSUB2) and `long_subtract` (XSUB3). The example was developed with the MS-DOS version of MCC68K.

### Example Header File

The `header.src` file (Figure H-6) defines the U-Memory header for the example application. When the header is linked with the initialization routine and the application-specific subroutine file, the header must be placed at location 0 of the load module. Additionally, all code and data constants must be loaded before any variables (`zerovars` and `vars`), which must be loaded before the stack section. See the sample link command file in Figure H-9.

```
TTL      HEADER  -- U-Memory header for sample application.

*=====
* HEADER.SRC -- U-Memory header for sample application.
*
* * DESCRIPTION:
*
* File HEADER.SRC contains the definition for the U Memory header required
* for the sample XSUB application. This file is written in the Microtec
* ASM68K assembly language. The object from this file must be loaded at
* relative address 0 of the U Memory load image.
*
*=====
OPT      CASE          Labels are case sensitive
SECTION  const,,R      Header must be first in <const> section
DC.W     1              Header version is 1 for rel 2.x
DC.W     NUM_SUB        Number of subroutine entry points
DC.L     .startof.(zerovars) Start of modifiable variables
DC.L     .startof.(stack) Lowest address for valid stack pointer
XREF     STACKSIZE      Size of stack (defined at link time)
DC.L     STACKSIZE
EP_TBL   EQU           *
XREF     __vinit         XSUB1 initializes static variables
DC.L     __vinit         .
XREF     _long_add       XSUB2 performs a long integer ADD
DC.L     _long_add       .
XREF     _long_subtract  XSUB3 performs a long integer SUB
DC.L     _long_subtract
NUM_SUB  EQU           (*-EP_TBL)/4  Computes number of entry points
END
```

Figure H-6 Example Assembly Language Header File

---

The header.src file contains pointers to the base of the `zerovars` and stack sections, and to external entry points `_vinit`, `_long_add`, and `_long_subtract`. Note that the subroutine entry point names are preceded with an underscore. This is a C Language requirement. During execution, these pointers are used by the controller's operating system as offsets relative to the start of U-Memory.

 **WARNING**

Other than the header, external subroutines should not define or use static pointers.

Use of invalid pointers is likely to cause unpredictable operation that could result in death or serious injury to personnel, and/or damage to equipment.

Pointers passed as parameters on a given subroutine call may be invalidated if you reconfigure user memory.

 **WARNING**

Do not change any portion of the U-Memory content loaded in front of the base address of `zerovars` after the U-Memory load. Otherwise, the controller enters the FATAL ERROR mode due to a U-Memory checksum violation, turns off discrete outputs and freezes analog outputs.

This could cause unpredictable operation of the controller that could result in death or serious injury to personnel, and/or damage to equipment.

Only properly trained personnel should work on programmable controller-based equipment.

## Developing an External Subroutine — Example (continued)

---

### Example Subroutine Source

Depending on the complexity of your application, the subroutine source may be a single file or several files. Figure H-7 shows file `xsubs.c`, which defines the application-specific subroutines comprising the example. The initialization routine is contained in file `vinit.src` (Figure H-2).

```
/*Procedure long_add:      Compute the sum of two long words */
/*                        and store the result in a third  */
/*                        long word.                        */
void long_add (long addend_1, long addend_2, long *sum)
{
    *sum = addend_1 + addend_2;
    return;
}
/*Procedure long_subtract: Subtract one long word from a   */
/*                        second long word and store the  */
/*                        result in a third long word.    */
void long_subtract
(long minuend, long subtrahend, long *difference)
{
    *difference = minuend - subtrahend;
    return;
}
```

Figure H-7 Example Subroutine Source File

### Preparing the Load Module

Figure H-8 shows the MS-DOS commands required to create a Motorola S-record load module for the example.

- The first two commands assemble `header.src` and `vinit.src`, producing object files `header.obj` and `vinit.obj`, respectively.
- The third command compiles `xsubs.c`, producing object file `xsubs.obj`. Compiler options force the compiler to generate PC-relative code (`-Mcp`) and data (`-Mdp`) references. These options are mandatory. They ensure that the resulting load module is position-independent. The `-c` option instructs `MCC68K` to create an object module without invoking the linker.
- The fourth command invokes the linker with command file `xsubs.cmd` and output file `xsubs.rec`. The `.rec` extension is required by `TISOFT`.

```
> asm68k header.src
> asm68k vinit.src
> mcc68k -Mcp -Mdp -c xsubs.c
> lnk68k -c xsubs.cmd -o xsubs.rec
```

Figure H-8 Example Commands for Preparing the Load Module

The content of the link command file depends on the complexity of your application. File `xsubs.cmd` shown in Figure H-9 is sufficient for the example application. Table H-1 lists the functions of the linker commands contained in this file.

```

CASE
FORMAT      S
LISTABS     NOPUBLICS,NOINTERNALS
ORDER       const,code,strings,literals,??INITDATA
ORDER       zerovars,vars,tags,stack
INITDATA    vars
PUBLIC      STACKSIZE=1024
BASE        0
LOAD        header.obj
LOAD        vinit.obj
LOAD        xsubs.obj
LOAD        c:\mcc68k\mcc68kpc.lib
END

```

Figure H-9 Example Link Command File

Table H-1 Linker Command Functions

Command	Description
CASE	Indicates that symbols are case sensitive.
FORMAT	Indicates that the linker output is to be in Motorola S-record format.
LISTABS	Tells the linker to omit symbol table information from the load module.
ORDER	Specifies the order in which sections are to be placed in the load image generated by the linker. The first ORDER statement lists all sections whose content do not change after U-Memory is loaded. Section <code>const</code> must be named first so that the U Memory header is at 0. This is followed by the names of other invariant sections produced by the compiler and linker. The second ORDER statement lists all sections whose content may change after U-Memory is loaded. These sections must be linked after all invariant sections. The <code>zerovars</code> section must be named first and the <code>stack</code> section must be named last.
INITDATA	Tells the linker to create a read only copy of initialized variables (section <code>vars</code> ) in section <code>??INITDATA</code> . Subroutine <code>_vinit</code> uses this copy to initialize the actual variables in section <code>vars</code> .
PUBLIC	Tells the linker to define variable <code>STACKSIZE</code> . The value on the right of the equal sign is placed in the U-Memory header's stack size data element.
BASE	Tells the linker to link relative to address 0.
LOAD	Tells the linker which modules are to be included in the load module. Name the header file ( <b>header.obj</b> ) first. You can load other modules in any order. File <b>C:\mcc68k\68020\mcc68kpc.lib</b> is the position independent run-time library for MCC68K.

## Developing an External Subroutine — Example (continued)

**Loading U-Memory** Use the TISOFT Convert S-Records option (AUX 40) to import xsubs.rec into the TISOFT file system; then use the TISOFT Load U-Memory option (AUX 43) to download to U-Memory.

---

**NOTE:** If you have not configured U-Memory, you must do so before TISOFT allows these functions.

---

### Using the External Subroutines in RLL

When you initialize the RLL program, you must also initialize the external subroutine variables. Figure H-10 illustrates a call to `_vinit` (XSUB1), which occurs once, whenever control relay C1 is off. Note that the `_vinit` call must specify a single IO parameter. This parameter is written with the return code from `_vinit`.

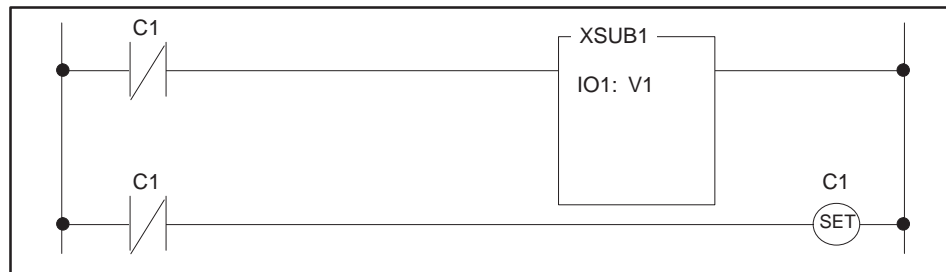


Figure H-10 Example Subroutine Call for Static Variable Initialization

Figure H-11 illustrates an RLL network that calls the `long_add` subroutine. There are three parameters in the XSUB2 box. These correspond to the three parameters in the `long_add` subroutine. The first parameter (IN1) corresponds to parameter `addend_1` in the definition of `long_add`. The second parameter (IN2) corresponds to `addend_2`, and the third parameter (IO3) corresponds to `sum`.

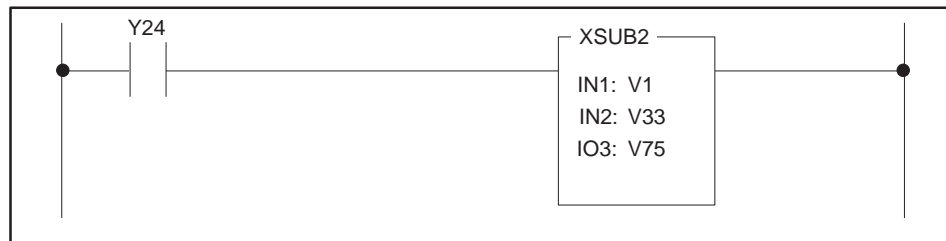


Figure H-11 Example Call to a Subroutine

---

There must be a one-to-one correspondence between parameters in the XSUB call (from top to bottom) and parameters in the subroutine definition (from left to right for C).

- Parameters one and two are IN parameters in the XSUB call. This is required since `long_add` expects `addend_1` and `addend_2` to be long integer values.
- Parameter three is an IO parameter in the XSUB call. This agrees with `long_add`'s definition of `sum` as a pointer, or address, parameter.

 **WARNING**

You must ensure agreement between the XSUB call and the XSUB's definition in the number and use of parameters.

If, for example, you were to specify IN for parameter three in the example XSUB2 call, the `long_add` subroutine would use the value of V75–76 as an address. The result, although unspecified, is likely to be a fatal error due to access to an undefined address or due to corruption of the controller execution environment.

This could cause unpredictable operation of the controller that could result in death or serious injury to personnel, and/or damage to equipment

Only properly trained personnel should work on programmable controller-based equipment.



# Interboard Communications for the 575

---

<b>I.1</b>	<b>Using Applications to Enable CPUs to Exchange Data</b> .....	<b>I-2</b>
	Applications .....	I-2
	Overview .....	I-4
	G-Memory Areas .....	I-4
	Required and Optional Applications .....	I-5
	Locking Mode Transitions for Two or More Applications .....	I-6
<b>I.2</b>	<b>Using Direct VMEbus Access to Communicate with Third-Party Boards</b> .....	<b>I-8</b>
	Accessing VMEbus Masters and Slaves Directly .....	I-8
<b>I.3</b>	<b>Coordinating Access to Shared Memory</b> .....	<b>I-10</b>
	Using Locks .....	I-10

## I.1 Using Applications to Enable CPUs to Exchange Data

### Applications

In the 575 system, an application is a user program that executes on a single 575 CPU (or another CPU that conforms to the SIMATIC 575 Interboard Communication Specification). An application on a 575 CPU consists of the following program elements.

- RLL programs
- SF programs
- Loops
- Analog alarms
- External subroutines

Each 575 application presents an area of memory (G-Memory) to the VMEbus. G-Memory allows CPUs and intelligent I/O to exchange data over the VMEbus backplane. For example, an application within a CPU can read data from another application and write data back to that application. See Figure I-1.

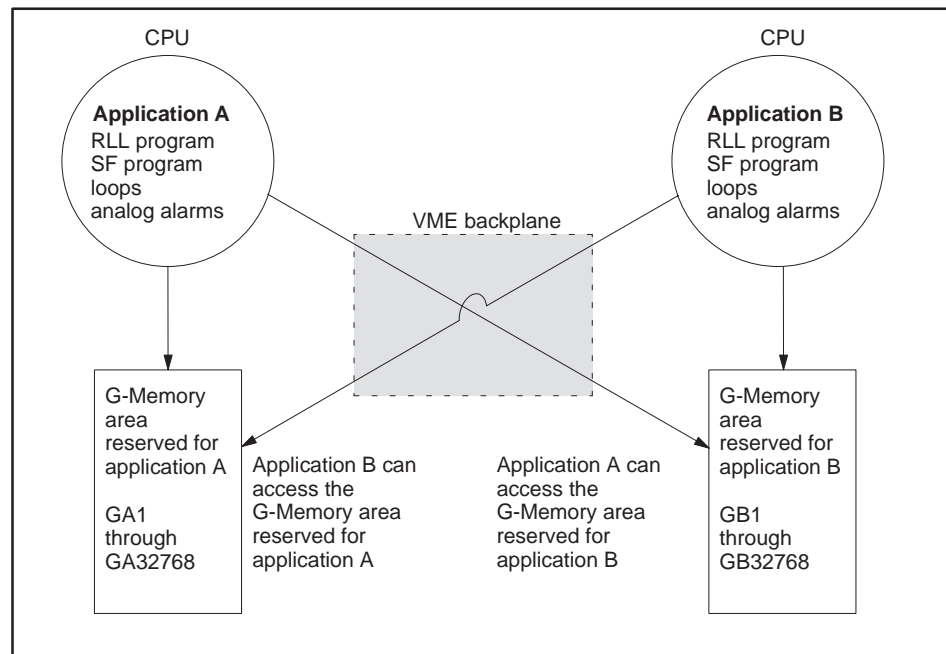


Figure I-1 Typical CPU Application

---

Boards can work together to control a process by communicating through their G-Memory areas. Using G-Memory, you can do the following tasks:

- Exchange data between applications without being aware of the physical addresses of the application's memory.
- Set locks so that one application can manipulate data in specific G-Memory locations without having those locations accessed by another application at the same time.
- Synchronize applications so that they initialize status words at the same time.
- Share data in memory instead of over the communication network. This practice allows data to be shared quickly and efficiently.

## Using Applications to Enable CPUs to Exchange Data (continued)

### Overview

Each application uses G-Memory to make data available to other applications. G-Memory resides on the VMEbus, and consists of up to 26 separate memory areas, totalling 32 Kwords. These areas are accessed by using the G-Memory addresses  $GA_n$  through  $GZ_n$  ( $n$  is a value from 1 to 32768). See Figure I-2.

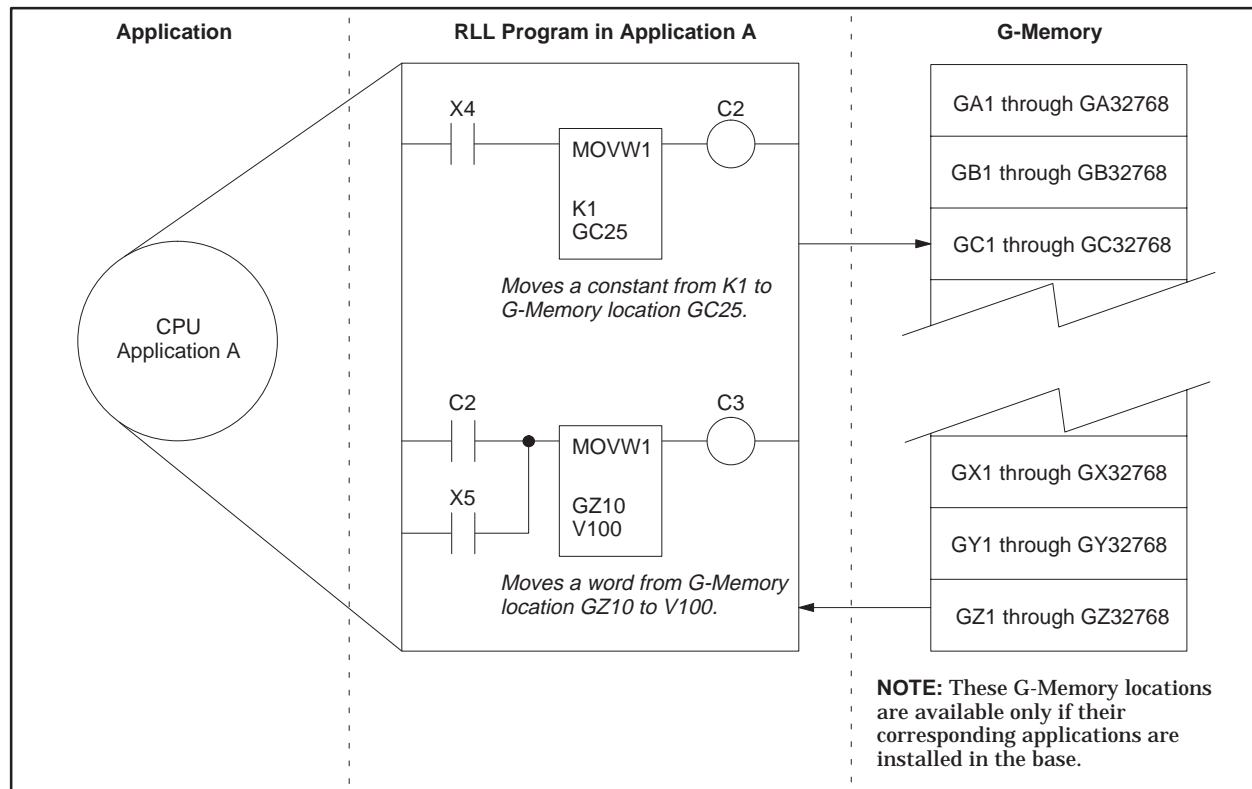


Figure I-2 Accessing G-Memory

### G-Memory Areas

Each 575 application is automatically assigned a G-Memory area. The memory area is determined by the relationship of the board to the first application inserted in the base.

Application A is assigned to the primary 575 CPU. G-Memory locations GA1 through GA32768 are automatically assigned to that board. If a board is the sixth application board installed in the base, G-Memory locations GF1 through GF32768 automatically belong to that board.

As long as the assignments for each of the installed applications are unique, any non-575 applications may be assigned to any of the unused application identifiers (B – Z).

---

## Required and Optional Applications

Applications depend on G-Memory to exchange data. Therefore, you must designate whether an application is required or optional. If an application is required, it must be installed before the application that refers to it can go to the RUN mode. If an application is optional, it does not have to be installed before the application that refers to it can operate.

In order for an application to reference another application's G-Memory, the second application must be installed as either required or optional.

If an optional application is installed:

- The application referring to that application can go to RUN mode.
- References to that application perform their specified function.

If an optional application is not installed:

- The application referring to that application can go to RUN mode.
- A reference to that application's G-Memory sets error bit 6 in STW1 if the reference actually is executed.
- If it is the first error logged, STW200 (uninstalled application) is logged as well.

To designate which applications are required or optional, use the REQAPP option from the TISOFT Memory Configuration menu.

## Using Applications to Enable CPUs to Exchange Data (continued)

### Locking Mode Transitions for Two or More Applications

You can configure CPUs so that they experience mode transitions (PROGRAM to RUN, etc.) simultaneously. When you request this function, the CPUs and their applications are mode-locked.

When you change an application from PROGRAM mode to RUN mode, all mode-locked applications change to RUN mode with it. The duration of the first scan is the same for all these applications, but the length of this scan does not affect the peak scan time. This feature allows you to initialize status words and memory so that applications do not inadvertently access corrupt data.

If you change a mode-locked application from RUN mode to PROGRAM mode, all mode-locked applications change to PROGRAM mode with it. See Figure I-3.

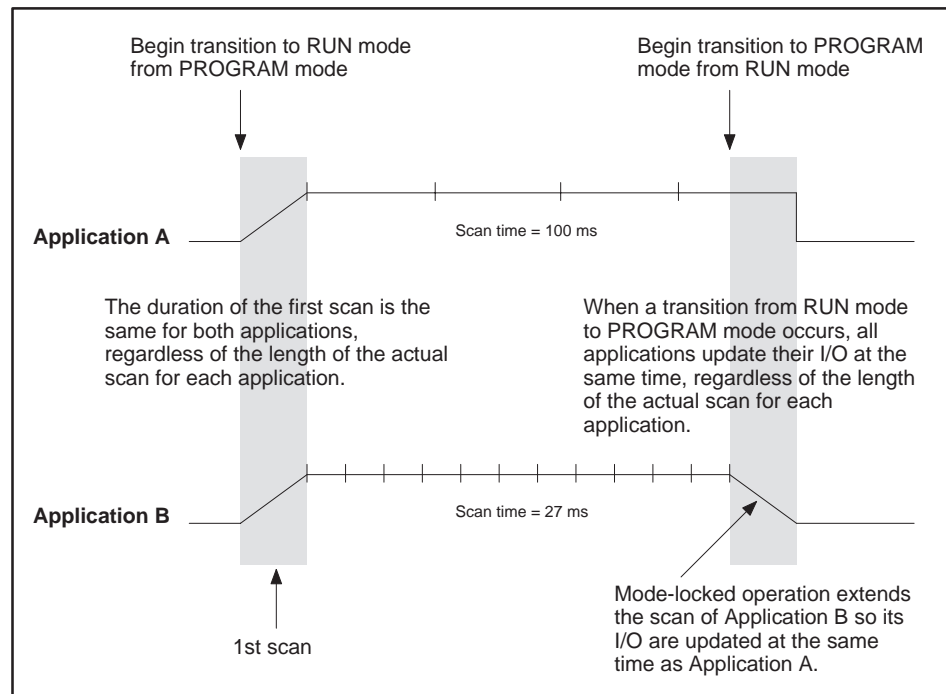


Figure I-3 Example of Mode-locked Applications

---

To mode-lock two or more applications, use TISOFT to bring up the Memory Configuration menu. Select REQAPP and follow the prompts.

You can perform run-time edits on mode-locked applications. If you perform a run-time edit to initialize values in a mode-locked application, the edit does not affect the other mode-locked applications, unless a compile error occurs, in which case all mode-locked applications go to PROGRAM mode.

If a fatal error occurs, the system transitions to the FAULT mode and shuts down all applications. A power-up restart synchronizes the first scan of all the 575 CPU applications, regardless of whether they are mode-locked. If you try to restart a different way, the system asks you whether or not you want mode-locked operation. You can specify which applications you want to mode-lock.

## I.2 Using Direct VMEbus Access to Communicate with Third-Party Boards

---

### Accessing VMEbus Masters and Slaves Directly

Most third-party boards that you can use in the 575 system do not conform to the SIMATIC 575 Interboard Communications Specification. This means that they do not provide application memory (G-Memory) as the means for interboard communication. To communicate with these boards, you must use direct VMEbus access.

You can access VMEbus slaves and masters directly from your 575 user program in the ways described below. All user-program access to VMEbus A16 address space (VMS memory type) is performed using VMEbus address modifier  $29_{16}$  (short, non-privileged access). User-program access to VMEbus A24 address space (VMM memory type) is performed using address modifier  $39_{16}$  (standard, non-privileged data access).



---

The 575 provides the following three ways to access VMEbus addresses directly from your user program.

- The Move Element (MOVE) RLL instruction allows you to move bytes, words, and long-words<sup>1</sup> to/from VMEbus memory. The MOVE instruction provides for recovery from VMEbus access errors. Access to an invalid VMEbus location results in an instruction error being reported to your RLL program, but the 575 does not enter FATAL ERROR mode.
- All word-oriented RLL instructions (e.g., ADD, SUB, MOVW, etc.) allow you to operate on VMEbus locations. They do not, however, provide for recovery from VMEbus access errors. If a VMEbus access error occurs, the 575 enters FATAL ERROR mode.
- You can create an external subroutine (XSUB) to perform the VMEbus access. You can pass the VMEbus address to the external subroutine to access the VMEbus by using an I/O parameter specifying VMS (A16) or VMM (A24) memory, or you can code the VMEbus address in your external subroutine. If you code the address in your external subroutine, you must offset the VMEbus address with the appropriate “address space selector” as follows:

575 CPU address F0xxxx<sub>16</sub> selects VMEbus A16 address xxxx<sub>16</sub>.

575 CPU address 01xxxxxx<sub>16</sub> selects VMEbus A24 address xxxxxx<sub>16</sub>.

Refer to Chapter 6 for a complete description of these instructions.

---

**NOTE:** The VMS and VMM variable types provide access to word (even-numbered) addresses. To specify an odd byte address in RLL, you must use the MOVE instruction, specifying an element size of byte and an odd index.

---

<sup>1</sup>Because the 575 is a D16 master, long-word (32-bit) VMEbus accesses are performed (physically) as two consecutive word (16-bit) accesses. This means that a long-word access may not be atomic. It is possible for the long word to be changed (if the board is accessed) after the first word is read and before the second word is read. It is also possible for the long word to be read (if the board is accessed) after the first word is written and before the second word is written.

## I.3 Coordinating Access to Shared Memory

---

### Using Locks

If you need to manipulate data stored in shared memory (G, VMM, or VMS) without interference from other applications, design your application to coordinate access to the shared memory using a lock. Figure I-4 and Figure I-5 illustrate how a lock operates. In these examples, 575 applications A and B are coordinating access to location GA110 using a lock that resides in location GA2 (and GA3).

A lock can be held in either an exclusive or a shared mode. When the lock is held in exclusive mode by an application, other applications are notified not to read from or write to the associated shared memory. When a lock is held in shared mode by an application, other applications are notified that they can read from the associated shared memory, but they should not write to it.

---

**NOTE:** Although a lock establishes something like a software contract, the 575 does not enforce this contract. If an application either fails to use the lock or ignores its state, access to the shared memory area can be corrupted.

---

On the 575, you can acquire a lock with the LOCK instruction and you can release it with the UNLCK instruction. For details on the lock data structure and the algorithms used to acquire and release a lock using a third-party board, refer to the *SIMATIC 575 Interboard Communication Specification* (PPX:575-8103-x).

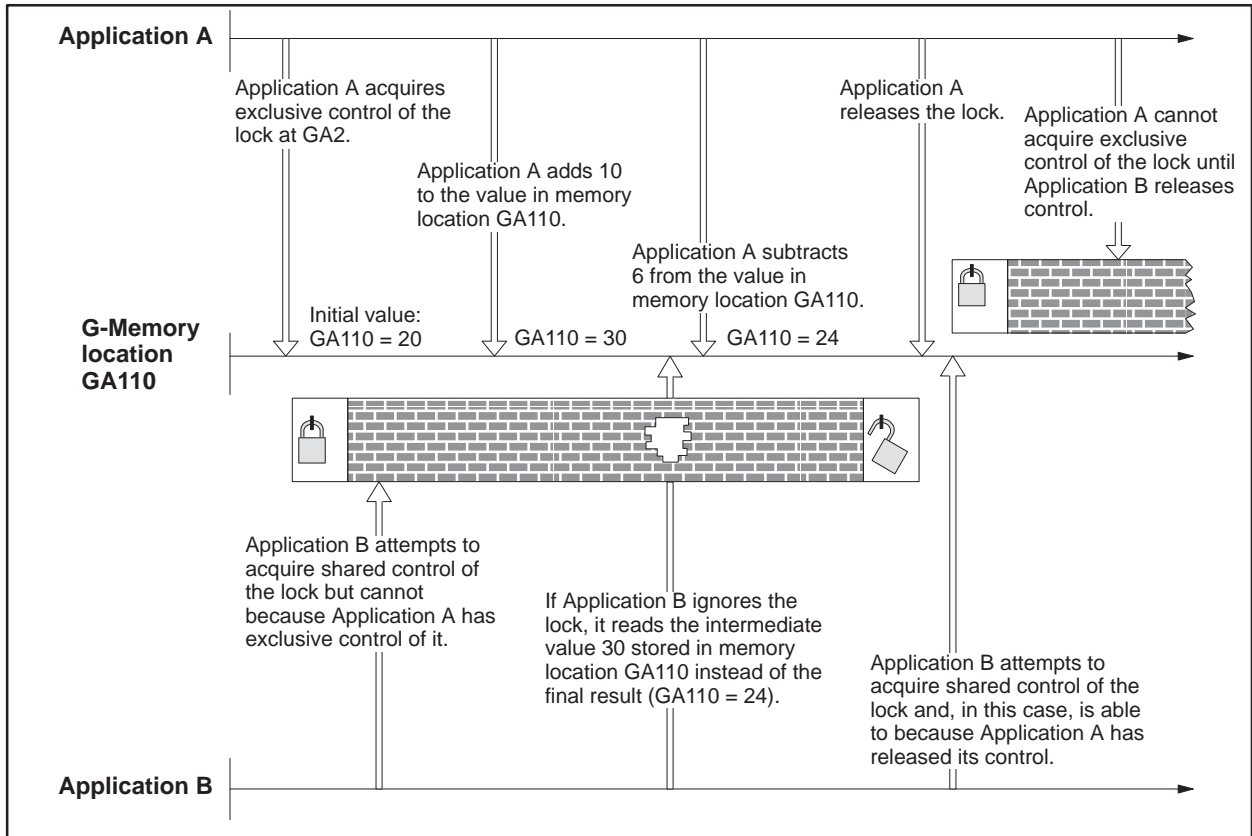


Figure I-4 Example of Locks and Their Uses

## Coordinating Access to Shared Memory (continued)

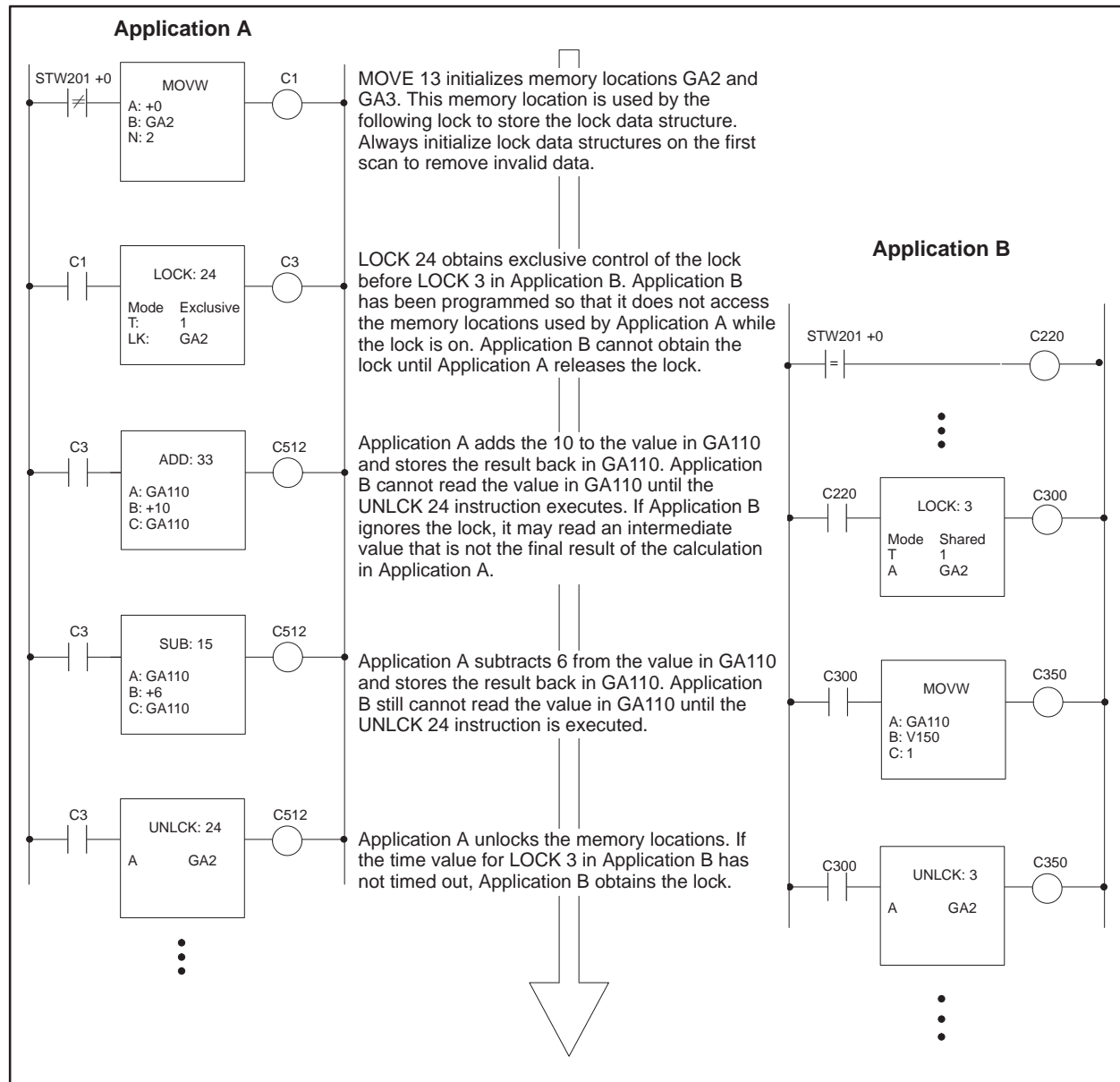


Figure I-5 RLL Example for Locks

## A

ABSV (compute absolute value), 6-11

ADD (addition), 6-12

Alarm deadband  
  analog alarm, 8-9  
  loop, 9-19

Algorithm, loop, 9-6, 9-10

Analog alarm  
  alarm deadband, 8-9  
  deviation alarms, 8-13  
  process variable alarms, 8-10  
  sample rate, 8-7  
  SF program call, 8-12  
  V-flags, 8-6

Analog tasks  
  545/555/575 controllers, 1-8-1-11  
  RBE, 1-9

Annex card, PROFIBUS-DP, 1-2

Application example  
  BITP (bit pick), E-10  
  CBD (convert binary to BCD), E-30-E-31  
  CDB (convert BCD to binary), E-32  
  DCAT (discrete control alarm timer),  
    E-34-E-36  
  DRUM (time-driven drum), E-11-E-12  
  EDRUM (time/event drum), E-13-E-16  
  MIRW (move image register to word),  
    E-17-E-19  
  MWFT (move word from table), E-26-E-27  
  MWIR (move word to image register),  
    E-20-E-23  
  MWTT (move word to table), E-24-E-25  
  One shot, E-33  
  SHRB (bit shift register), E-2-E-3  
  SHRW (word shift register), E-4-E-5  
  STW (status word), E-37  
  TMR (timer), E-6-E-9  
  WXOR (word exclusive OR), E-28-E-29

Application flags, status word, G-20, G-21

Application ID, status word, G-26

Application installed flags, status word, G-22,  
  G-23

Applications  
  and G-memory, I-4  
  in controllers, I-2-I-4  
  mode-locked, I-6-I-8  
  required and optional, I-5

APT, programming software, 5-32

Assistance, technical, xxxvi

Automatic loop tuning, 9-34-9-45

## B

Base poll enable flags, status word, G-25

BCD  
  conversions  
    CBD (convert binary to BCD), 6-16  
    CDB (convert BCD to binary), 6-18  
  defined, 2-6  
  format, 2-6

BCDBIN (SF program BCD conversion math),  
  7-24

Bias, loop  
  adjusting, 9-24  
  freezing, 9-23

BINBCD (SF program BCD conversion math),  
  7-25

Bit manipulations  
  BITC (bit clear), 6-13  
  BITS (bit set), 6-15  
  IMC (indexed matrix compare), 6-48  
  SMC (scan matrix compare), 6-140  
  WAND (word AND), 6-168  
  WOR (word OR), 6-170  
  WROT (word rotate), 6-172  
  WXOR (word exclusive OR), 6-182

Bit-of-word coil, 5-10, 6-22

## Index (continued)

---

Bit-of-word contact, 5-7  
BITC (bit clear), 6-13  
BITP (bit pick), application example, E-10  
BITS (bit set), 6-15  
Byte defined, 2-2

## C

C control relay, 5-7, 5-10  
CALL (SF program flow), 7-26  
CBD (convert binary to BCD), 6-16  
    application example, E-30–E-31  
CDB (convert BCD to binary), 6-18  
    application example, E-32  
CDT (SF program table handling), 7-28  
Clock data, 5-28  
Clock instructions  
    DCMP (date compare), 6-30  
    DSET (date set), 6-38  
    TCMP (time compare), 6-153  
    TSET (time set), 6-159  
CMP (compare), 6-20  
Coil  
    (normal), 5-9  
    (not-ed), 5-9  
COM PROFIBUS, 5-32  
Compiled mode SF execution, 7-6  
Compiled Special (CS) memory, 4-4  
Configuration, software  
    COM PROFIBUS, 5-32  
    SoftShop, 5-32  
    TISOFT, 5-32  
Constant (K) memory, 4-5  
Contact  
    (normal), 5-3  
    (not-ed), 5-5  
Control relay  
    forcing, 3-13  
    memory, 3-13, 4-4  
    non-retentive, 3-13, 3-14  
    retentive, 3-13, 3-14

Controller systems, overview, 1-2–1-5  
CS Memory, 4-4  
CS-Memory checksum, status word, G-32  
CTR (counter), 6-24  
Cyclic RLL  
    defined, 1-6, 5-18  
    IORW (immediate I/O read/write), 6-50  
    TASK (start new RLL task), 6-150  
    task overrun, status word, G-26  
Cyclic SF program, 6-126, 7-3, 7-12

## D

Data representation  
    BCD, 2-6  
    byte, 2-2  
    I/O point, 2-2  
    integer, 2-3, 2-4  
    long word, 2-2  
    real number, 2-5  
    word, 2-2  
Day of year, current status word, 5-30, 5-31,  
    G-28  
DCAT (discrete control alarm timer), 6-26  
    application example, E-34–E-36  
DCMP (date compare), 6-30  
Derivative gain limiting, loop, 9-25  
Deviation alarms  
    analog alarm, 8-13  
    loop, 9-31  
Direct-acting, loop, 9-30  
Discrete image register, 3-3, 4-4  
Discrete scan, controllers, 1-6  
Discrete scan execution time, status word, G-17  
DIV (division), 6-32  
DRUM (time-driven drum), 6-34  
    application example, E-11–E-12  
Drum memory  
    DCC, 4-9  
    DCP, 4-9  
    DSC, 4-9  
    DSP, 4-9

---

DSET (date set), 6-38  
Dual power supply status, status word, G-16  
Dual RBC status, status word, G-15

## E

E bit, U-memory header, 6-187  
E/Version word, H-4  
Editing during run-time, 5-33–5-40  
EDRUM (event-driven drum), application example, E-13–E-16  
EDRUM (time/event drum), 6-40  
Electromechanical replacement  
  bit-of-word coil, 5-10, 6-22  
  bit-of-word contact, 5-7  
  C control relay, 5-7, 5-10  
  CTR (counter), 6-24  
  DCAT (discrete control alarm timer), 6-26  
  DRUM (time-driven drum), 6-34  
  EDRUM (time/event drum), 6-40  
  immediate X contact, 5-8  
  immediate Y coil, 5-10  
  JMP (jump), 6-52  
  LBL (label), 6-136  
  MCAT (motor control alarm timer), 6-63  
  MCR (master control relay), 6-68  
  MDRMD (maskable event drum, discrete), 6-72  
  MDRMW (maskable event drum, word), 6-76  
  relational contact, 5-8, 6-23  
  reset coil, 5-11, 6-22  
  reset coil bit-of-word, 5-11  
  reset coil immediate, 5-11  
  set coil, 5-11, 6-22  
  set coil bit-of-word, 5-11  
  set coil immediate, 5-11  
  SHRB (bit shift register), 6-132  
  SKP (skip), 6-136  
  TMR (timer), 6-156  
  X contact, 5-7  
  Y coil, 5-10  
  Y contact, 5-7

END (unconditional end), 6-44  
ENDC (conditional end), 6-45  
Error operation, loop, 9-29  
Errors, SF program, 7-20, F-1  
EXIT (SF program flow), 7-30  
External subroutine  
  code requirements, H-2  
  guidelines, H-6  
    accessing word/discrete variables, H-10  
    debugging, H-6  
    floating point operations, H-11  
    static data initialization, H-7  
  header elements, H-4  
  header file example, H-12  
  link command file example, H-15  
  loading procedure, H-3  
  RLL XSUB call example, H-16  
  source file example, H-14

## F

Fast loop (PID instruction), 6-110  
First scan flags, status word, G-19  
Forcing function, 3-3, 3-5, 3-13  
Format  
  BCD, 2-6  
  integer, 2-3, 2-4  
  real number, 2-5  
FTSR-IN (SF program table handling), 7-31  
FTSR-OUT (SF program table handling), 7-35

## G

G-memory, 4-11  
  access to, I-10  
  addresses, I-4  
  description of, I-2  
  LOCK and UNLCK, I-11  
  locks, I-10–I-14  
  locks and RLL, I-11  
Gain, loop, 9-22  
Global (G) memory, 4-11  
GOTO (SF program flow), 7-39  
GTS (go to subroutine), 6-46

## Index (continued)

---

### H

Header, U-Memory, H-4  
Hotline, for technical assistance, xxxvi

### I

I/O  
  base status, status word, G-3  
  channel  
    PROFIBUS-DP, 1-4  
    Series 505 remote, 1-2  
  expansion, 1-2  
  forcing, 3-3, 3-5  
  immediate, 3-8  
  local, 1-2  
  module status, status word, G-6  
  modules supporting immediate I/O, 3-10  
  point defined, 2-2  
  point numbers, 1-5  
IF (SF program flow), 7-40–7-41  
IIF (SF program flow), 7-40–7-41  
Image register  
  discrete, 3-3, 4-4  
  immediate update, 3-8  
  normal update, 3-2  
  word, 3-5, 4-4  
IMATH (SF program integer math), 7-42  
IMC (indexed matrix compare), 6-48  
Immediate I/O  
  defined, 3-8  
  instructions  
    IORW (immediate I/O read/write), 6-50  
    TASK (start new RLL task), 6-150  
Immediate X contact, 5-8  
Immediate Y coil, 5-10  
In-line SF program execution, 7-12  
Integer, format, 2-3, 2-4  
Integer defined, 2-3, 2-4  
Interpreted mode SF execution, 7-6  
Interrupt I/O operation, 3-11–3-12  
  configuring interrupt module, 3-11–3-12  
  defined, 1-6, 3-11  
  performance characteristics, 5-26

RLL program, 5-22–5-24  
troubleshooting, 5-27  
using IORW instruction, 5-23–5-24  
using status word 220, 5-23–5-36  
using status word 221, 5-27  
using Task 8, 5-22–5-24

Interrupt request count, status word, G-27  
Interrupting slots in local base, status word, G-27

IORW (immediate I/O read/write), 3-9, 6-50

### J

JMP (jump) instruction, 6-52

### K

K-Memory, 4-5

### L

L-Memory, 4-4  
L-Memory checksum, status word, G-14  
LABEL (SF program flow), 7-39  
Ladder (L) memory, 4-4  
LBL (label) RLL instruction, 6-136  
LDA (load address) instruction, 6-54  
LDC (load data constant), 6-59  
LEAD/LAG (SF program signal processing math), 7-44  
LMN, remote setpoint, 9-21  
LOCK (lock memory) instruction, 6-60  
Locks  
  G-memory, I-10–I-14  
  in RLL, I-11–I-14  
Long word defined, 2-2  
Loop  
  alarm deadband, 9-19  
  algorithm, 9-6, 9-10  
  broken transmitter alarm, 9-5  
  derivative gain limiting, 9-25  
  deviation alarms, 9-31



---

Loop (continued)  
  direct-acting, 9-30  
  error operation, 9-29  
  locking mode, 9-28  
  locking setpoint, 9-28  
  mode, locking, 9-28  
  operational states, 9-28  
  output  
    20% offset, 9-18  
    address, 9-18  
    bipolar, 9-18  
  overview, 9-2  
  process variable alarms, 9-20  
  ramp/soak, 9-14  
  reverse-acting, 9-30  
  sample rate, 9-12  
  setpoint, locking, 9-28  
  SF program call, 9-26  
  V-flags, 9-11

## M

MATH (SF program real/integer math), 7-46

### Math operations

  ABSV (compute absolute value), 6-11  
  ADD (addition), 6-12  
  CMP (compare), 6-20  
  DIV (division), 6-32  
  MULT (multiplication), 6-98  
  SQRT (square root), 6-142  
  SUB (subtraction), 6-148

MCAT (motor control alarm timer), 6-63

MCR (master control relay) instruction, 6-68

MDRMD (maskable event drum, discrete), 6-72

MDRMW (maskable event drum, word), 6-76

Memory, G. *See* G-memory

### Memory types

  Compiled special (CS), 4-4  
  constant (K), 4-5  
  control relay, 4-4  
  drum, 4-9  
  global (G), 4-11  
  image register, 4-4  
  ladder (L), 4-4  
  one shot, 4-7  
  readable memory (defined), 4-3  
  shift register, 4-8

  special (S), 4-4  
  status word, 4-5  
  table move, 4-6  
  temporary (T), 4-4, 7-16  
  timer/counter, 4-5  
  user subroutine (U), 4-11  
  variable (V), 4-4  
  VME, 4-11  
  writeable memory (defined), 4-3

MIRFT (move image register from table), 6-82

MIRTT (move image register to table), 6-84

MIRW (move image register to word), 6-86  
  application example, E-17–E-19

Mode, loop, locking, 9-28

### Mode-locked applications

  examples, I-6  
  run-time edits, I-7  
  two or more, I-6

Module (I/O), modules supporting immediate I/O, 3-10

Module mismatch indicator, status word, G-17

MOVE (move element), 6-88

MOVW (move word), 6-96

MULT (multiplication), 6-98

MWFT (move word from table), 6-100  
  application example, E-26–E-27

MWI (move word with indirect addressing), 6-102

MWIR (move word to image register), 6-104  
  application example, E-20–E-23

MWTT (move word to table), 6-106  
  application example, E-24–E-25

## N

Non-priority SF program, 6-126, 7-3, 7-11  
  timeslice, 1-9

NOT instruction, 6-108

## O

One shot, 6-109  
  application example, E-33

## Index (continued)

---

One shot memory, 4-7

Output, loop  
  20% offset, 9-18  
  address, 9-18  
  bipolar, 9-18

## P

PACK (SF program table handling), 7-51

PACKAA (analog alarm handling), 7-56

PACKLOOP (move loop data), 7-58

PACKRS (pack ramp/soak data), 7-60

Parameter area  
  PGTS discrete, 4-10  
  PGTS word, 4-10

Password protection, 5-39

PETWD (pet scan watchdog), 7-66

PGTS  
  (parameterized go to subroutine), 6-112  
  discrete parameter area, 4-10  
  word parameter area, 4-10

PGTSZ (parameterized go to subroutine zero),  
  6-118

PID (call fast loop), 6-110

Position algorithm, loop, 9-6, 9-10

Power flow, 5-2

PowerMath, using with SF programming,  
  7-4-7-9

Priority SF program, 6-126, 7-3, 7-11  
  timeslice, 1-10

Process variable alarms  
  analog alarm, 8-10  
  loop, 9-20

Programmable controller status, status word,  
  G-2

Programming software  
  APT, 5-32  
  SoftShop, 5-32  
  TISOFT, 5-32

## R

Ramp/soak, 9-14

Rate, loop, 9-22

RBE, 1-9  
  event detection, 1-10

Readable memory, 4-3

Real number  
  defined, 2-5  
  format, 2-5

Relational contact, 5-8, 6-23

Reset, loop, 9-22

Reset coil, 5-11, 6-22

Reset coil bit-of-word, 5-11

Reset coil immediate, 5-11

Restricted SF program, 7-3, 7-12

RETURN (SF program flow), 7-71

Reverse-acting, loop, 9-30

RLL

  box instruction, 5-12  
  coil, 5-8  
  contact, 5-3  
  rung structure, 5-12  
  scan principles, 5-13  
  subroutine stack overflow, status word, G-14

RLL instructions

  ABSV (compute absolute value), 6-11  
  ADD (addition), 6-12  
  bit-of-word coil, 5-10, 6-22  
  bit-of-word contact, 5-7  
  BITC (bit clear), 6-13  
  BITS (bit set), 6-15  
  C control relay, 5-7, 5-10  
  CBD (convert binary to BCD), 6-16  
  CDB (convert BCD to binary), 6-18  
  CMP (compare), 6-20  
  CTR (counter), 6-24  
  DCAT (discrete control alarm timer), 6-26  
  DCMP (date compare), 6-30  
  DIV (division), 6-32  
  DRUM (time-driven drum), 6-34  
  DSET (date set), 6-38  
  EDRUM (time/event drum), 6-40  
  END (unconditional end), 6-44  
  ENDC (conditional end), 6-45  
  GTS (go to subroutine), 6-46

---

RLL instructions (continued)

- IMC (indexed matrix compare), 6-48
- immediate X contact, 5-8
- immediate Y coil, 5-10
- IORW (immediate I/O read/write), 6-50
- JMP (jump), 6-52
- LBL (label), 6-136
- LDA (load address), 6-54
- LDC (load data constant), 6-59
- LOCK (lock memory), 6-60
- MCAT (motor control alarm timer), 6-63
- MCR (master control relay), 6-68
- MDRMD (maskable event drum, discrete), 6-72
- MDRMW (maskable event drum, word), 6-76
- MIRFT (move image register from table), 6-82
- MIRTT (move image register to table), 6-84
- MIRW (move image register to word), 6-86
- MOVE (move element), 6-88
- MOVW (move word), 6-96
- MULT (multiplication), 6-98
- MWFT (move word from table), 6-100
- MWI (move word with indirect addressing), 6-102
- MWIR (move word to image register), 6-104
- MWTT (move word to table), 6-106
- NOT, 6-108
- One shot, 6-109
- PGTS (parameterized go to subroutine), 6-112
- PGTSZ (parameterized go to subroutine zero), 6-118
- PID fast loop, 6-110
- relational contact, 5-8, 6-23
- reset coil, 5-11, 6-22
- reset coil bit-of-word, 5-11
- reset coil immediate, 5-11
- RSD (return slave diagnostic), 6-120
- RTN (return from subroutine), 6-122
- SBR (subroutine), 6-123
- set coil, 5-11, 6-22
- set coil bit-of-word, 5-11
- set coil immediate, 5-11
- SF program called from RLL, 7-11
- SF subroutine (call SF subroutine from RLL), 7-14
- SFPGM (SF program call), 6-126
- SFSUB (SF subroutine call), 6-128
- SHRB (bit shift register), 6-132
- SHRW (word shift register), 6-134
- SKP (skip), 6-136
- SMC (scan matrix compare), 6-140
- SQRT (square root), 6-142
- STFE (search table for equal), 6-144
- STFN (search table for not equal), 6-146
- SUB (subtraction), 6-148
- TAND (table to table AND), 6-149
- TASK (start new RLL task), 6-150
- TCMP (time compare), 6-153
- TCPL (table complement), 6-154
- TEXT, 6-155
- TMR (timer), 6-156
- TOR (table to table OR), 6-158
- TSET (time set), 6-159
- TTOW (table to word), 6-160
- TXOR (table to table exclusive OR), 6-162
- UNLK (unlock memory), 6-167
- WAND (word AND), 6-168
- WOR (word OR), 6-170
- WROT (word rotate), 6-172
- WTOT (word to table), 6-174
- WTTA (word to table AND), 6-176
- WTTO (word to table OR), 6-178
- WTTXO (word to table exclusive OR), 6-180
- WXOR (word exclusive OR), 6-182
- X contact, 5-7
- XSUB (external subroutine call), 6-184
- Y coil, 5-10
- Y contact, 5-7

RLL theory

- box instruction, 5-12
- coil, 5-8
  - normal, 5-9
  - not-ed, 5-9
- concept, 5-2
- contact, 5-3
  - normal, 5-3
  - not-ed, 5-5
- cyclic RLL, 5-18
- immediate I/O, 3-8
- power flow, 5-2
- rung structure, 5-12
- scan principles, 5-13
- subroutines, 5-16

RSD (read slave diagnostic), 6-120

RTN (return from subroutine), 6-122

Run-time editing, 5-33–5-40

## Index (continued)

---

### S

- S-Memory, 4-4
- Sample rate
  - analog alarm, 8-7
  - loop, 9-12
- SBR (subroutine), 6-123
- SCALE (SF program data conversion math), 7-72
- Scan operations, 1-6
  - setting, 1-10
- Scan time, status word, G-4
- SDT (SF program table handling), 7-74
- Set coil, 5-11
  - bit-of-word, 5-11
  - defined, 5-11, 6-22
  - immediate, 5-11
- SF processor non-fatal errors, status word, G-13
- SF program
  - called from analog alarm, 8-12
  - called from loop, 9-26
  - called from RLL, 6-126, 7-11
  - defined, 5-16, 7-2
  - element (defined), 7-22
  - errors, 7-20, F-1
  - expression (defined), 7-22
  - types
    - cyclic, 6-126, 7-3, 7-12
    - non-priority, 6-126, 7-3, 7-11
    - priority, 6-126, 7-3, 7-11
    - restricted, 7-3, 7-12
- SF program statements
  - BCDBIN (BCD conversion math), 7-24
  - BINBCD (BCD conversion math), 7-25
  - CALL (program flow), 7-26
  - CDT (table handling), 7-28
  - EXIT (program flow), 7-30
  - FTSR-IN (table handling), 7-31
  - FTSR-OUT (table handling), 7-35
  - GOTO (program flow), 7-39
  - IF (program flow), 7-40–7-41
  - IIF (program flow), 7-40–7-41
  - IMATH (integer math), 7-42
  - LABEL (program flow), 7-39
  - LEAD/LAG (signal processing math), 7-44
  - MATH (real/integer math), 7-46
  - PACK (table handling), 7-51
  - PACKAA (analog alarm handling), 7-56
  - PACKLOOP (move loop data), 7-58
  - PACKRS (pack ramp/soak data), 7-60
  - PETWD (pet scan watchdog), 7-66
  - RETURN (program flow), 7-71
  - SCALE (data conversion math), 7-72
  - SDT (table handling), 7-74
  - SSR (table handling), 7-76
- SF subroutine
  - (call SF subroutine from RLL), 7-14
  - timeslice, 1-9
- SFPGM (SF program call from RLL), 6-126
- SFSUB (SF subroutine call from RLL), 6-128
- Shift register memory, 4-8
- SHRB (bit shift register), 6-132
  - application example, E-2–E-3
- SHRW (word shift register), 6-134
  - application example, E-4–E-5
- SKP (skip) instruction, 6-136
- Slaves, supported on PROFIBUS channel,
  - configuring with COM PROFIBUS, 5-32
- SmarTune, automatic loop tuning, 9-34–9-45
- SMC (scan matrix compare), 6-140
- SoftShop programming software, xxxiv, 5-32
- Special (S) memory, 4-4
- Spurious interrupt count, status word, G-27
- SQRT (square root), 6-142
- SSR (SF program table handling), 7-76
- Status word
  - application example, E-37
  - application flags, G-20, G-21
  - application ID, G-26
  - application installed flags, G-22, G-23
  - base poll enable flags, G-25
  - CS-Memory checksum, G-32
  - cyclic RLL task overrun, G-26
  - discrete scan execution time, G-17
  - dual power supply status, G-16
  - dual RBC status, G-15
  - first scan flags, G-19
  - I/O base status, G-3
  - I/O module status, G-6
  - interrupt request count, G-27
  - interrupting slots in local base, G-27
  - L-Memory checksum, G-14

---

Status word (continued)  
memory, 4-5  
module mismatch indicator, G-17  
programmable controller status, G-2  
receive errors, timeout errors, G-11  
RLL subroutine stack overflow, G-14  
scan time, G-4  
SF processor non-fatal errors, G-13  
spurious interrupt count, G-27  
time data, 5-28, G-9  
U-Memory checksum, G-24  
user error cause, G-18

STFE (search table for equal), 6-144

STFN (search table for not equal), 6-146

SUB (subtraction), 6-148

Subroutine, external  
accessing word/discrete variables, H-10  
coding requirements, H-2  
debugging, H-6  
floating point operations, H-11  
guidelines, H-6  
header elements, H-4  
header file example, H-12  
link command file example, H-15  
loading procedure, H-3  
RLL XSUB call example, H-16  
source file example, H-14  
static data initialization, H-7

Subroutine instructions  
GTS (go to subroutine), 6-46  
PGTS (parameterized go to subroutine), 6-112  
PGTSZ (parameterized go to subroutine zero), 6-118  
RSD (return slave diagnostic), 6-120  
RTN (return from subroutine), 6-122  
SBR (subroutine), 6-123  
XSUB (external subroutine call), 6-184

Subroutines, 5-16

Subscripting variables, SF program math, 7-49

## T

T-Memory, 4-4, 7-16

Table move memory, 4-6

Table operations  
MIRFT (move image register from table), 6-82  
MIRTT (move image register to table), 6-84

STFE (search table for equal), 6-144  
STFN (search table for not equal), 6-146  
TAND (table to table AND), 6-149  
TCPL (table complement), 6-154  
TOR (table to table OR), 6-158  
TTOW (table to word), 6-160  
TXOR (table to table exclusive OR), 6-162  
WTOT (word to table), 6-174  
WTTA (word to table AND), 6-176  
WTTO (word to table OR), 6-178  
WTTXO (word to table exclusive OR), 6-180

TAND (table to table AND), 6-149

Task, RLL program segments, 5-18

TASK (start new RLL task), 5-18, 6-150

TCMP (time compare), 6-153

TCPL (table complement), 6-154

Technical assistance, xxxvi

Temporary (T) memory, 4-4, 7-16

TEXT, Text Box documentation, 6-155

Text box, 6-155

Time data, 5-28  
status word, G-9

Time of day, binary status word for, 5-30, 5-31, G-28

Time slice, analog task processing, 1-8

Timer/counter memory, 4-5

TISOFT, programming software, 5-32

TMR (timer), 6-156  
application example, E-6–E-9

TOR (table to table OR), 6-158

Transmitter alarm, broken, loop, 9-5

TSET (time set), 6-159

TTOW (table to word), 6-160

Tuning loops, 9-22

TXOR (table to table exclusive OR), 6-162

## U

U-Memory, 4-11  
external subroutine and, H-4  
header, H-4

## Index (continued)

---

U-Memory checksum, status word, G-24  
UDC (up-down counter), 6-164  
UNLK (unlock memory) instruction, 6-167  
User error cause, status word, G-18  
User subroutine (U) memory, 4-11

## V

V-flags  
  analog alarm, 8-6  
  loop, 9-11  
V-Memory, 4-4  
Variable (V) memory, 4-4  
Variable subscripting, SF program math, 7-49  
Velocity algorithm, loop, 9-7, 9-10  
VMEbus, address, accessing non-existent, 6-89, 6-187  
VMEbus error, 4-11, 6-89, 6-187

## W

WAND (word AND), 6-168  
WOR (word OR), 6-170  
Word defined, 2-2  
Word image register, 3-5, 4-4  
Word moves  
  LDA (load address), 6-54

LDC (load data constant), 6-59  
MIRW (move image register to word), 6-86  
MOVE (move element), 6-88  
MOVW (move word), 6-96  
MWFT (move word from table), 6-100  
MWI (move word with indirect addressing), 6-102  
MWIR (move word to image register), 6-104  
MWTT (move word to table), 6-106  
SHRW (word shift register), 6-134

Writeable memory, 4-3  
WROT (word rotate), 6-172  
WTOT (word to table), 6-174  
WTTA (word to table AND), 6-176  
WTTO (word to table OR), 6-178  
WTTXO (word to table exclusive OR), 6-180  
WXOR (word exclusive OR), 6-182  
  application example, E-28–E-29

## X

X contact, 5-7  
XSUB (external subroutine call), 6-184

## Y

Y coil, 5-10  
Y contact, 5-7

# Customer Response

---

We would like to know what you think about our user manuals so that we can serve you better.  
How would you rate the quality of our manuals?

	Excellent	Good	Fair	Poor
Accuracy	_____	_____	_____	_____
Organization	_____	_____	_____	_____
Clarity	_____	_____	_____	_____
Completeness	_____	_____	_____	_____
Graphics	_____	_____	_____	_____
Examples	_____	_____	_____	_____
Overall design	_____	_____	_____	_____
Size	_____	_____	_____	_____
Index	_____	_____	_____	_____

Would you be interested in giving us more detailed comments about our manuals?

**Yes!** Please send me a questionnaire.

**No.** Thanks anyway.

Your Name: \_\_\_\_\_

Title: \_\_\_\_\_

Telephone Number: (\_\_\_\_) \_\_\_\_\_

Company Name: \_\_\_\_\_

Company Address: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

**Manual Name:** SIMATIC 545/555/575 Programming Reference User Manual

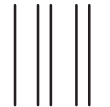
**Edition:** Second

**Manual Assembly Number:** 2806090-0002

**Date:** 08/98

**Order Number:** PPX:505-8204-2

FOLD



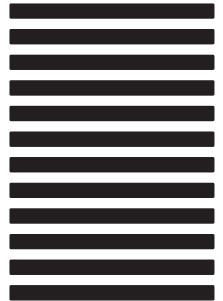
NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO.3 JOHNSON CITY, TN

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Technical Communications M/S 519  
SIEMENS ENERGY & AUTOMATION INC.  
3000 BILL GARLAND RD  
P O BOX 1255  
JOHNSON CITY TN 37605-1255



FOLD



SIMATIC and SINEC are trademarks of Siemens AG.

SoftShop, PowerMath, SmarTune, PCS, Series 505, Series 500, APT, PEERLINK, and TISOFT are trademarks of Siemens Energy & Automation, Inc.

IBM and AT are registered trademarks and XT is a trademark of International Business Machines Corporation.

DEC is a registered trademark and VAX is a trademark of Digital Equipment Corporation.

Microtec is a registered trademark of Microtec Research, Inc.

Motorola is a registered trademark of Motorola, Inc.

MS-DOS, Windows, and Quick C are registered trademarks of Microsoft Corporation.

Turbo C is a registered trademark of Borland International, Inc.