EcoStruxure Machine Expert - Basic

Generic Functions Library Guide

Original instructions

EIO0000003289.04
09/2025

www.se.com

Legal Information

The information provided in this document contains general descriptions, technical characteristics and/or recommendations related to products/solutions.

This document is not intended as a substitute for a detailed study or operational and site-specific development or schematic plan. It is not to be used for determining suitability or reliability of the products/solutions for specific user applications. It is the duty of any such user to perform or have any professional expert of its choice (integrator, specifier or the like) perform the appropriate and comprehensive risk analysis, evaluation and testing of the products/solutions with respect to the relevant specific application or use thereof.

The Schneider Electric brand and any trademarks of Schneider Electric SE and its subsidiaries referred to in this document are the property of Schneider Electric SE or its subsidiaries. All other brands may be trademarks of their respective owner.

This document and its content are protected under applicable copyright laws and provided for informative use only. No part of this document may be reproduced or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), for any purpose, without the prior written permission of Schneider Electric.

Schneider Electric does not grant any right or license for commercial use of the document or its content, except for a non-exclusive and personal license to consult it on an "as is" basis.

Schneider Electric reserves the right to make changes or updates with respect to or in the content of this document or the format thereof, at any time without notice.

To the extent permitted by applicable law, no responsibility or liability is assumed by Schneider Electric and its subsidiaries for any errors or omissions in the informational content of this document, as well as any non-intended use or misuse of the content thereof.

Table of Contents

Safety Information 7
- Before You Begin 7
- Start-up and Test 8
- Operation and Adjustments 9

About the Document 10

Introduction 14
- How to Use the Source Code Examples 14
- Operation Blocks 15
- Comparison Blocks 18

Language Objects 19
- Objects 19
- Memory Bit Objects 20
- I/O Objects 21
- Word Objects 23
- Floating Point and Double Word Objects 27
- Structured Objects 31
- Indexed Objects 33
- Function Block Objects 35

Instructions 37
- Boolean Processing 37
  - Boolean Instructions 37
  - Load Operators (*LD, LDN, LDR, LDF*) 39
  - Assignment Operators (*ST, STN, R, S*) 41
  - Logical AND Operators (*AND, ANDN, ANDR, ANDF*) 42
  - Logical OR Operators (*OR, ORN, ORR, ORF*) 43
  - Exclusive OR Operators (*XOR, XORN, XORR, XORF*) 44
  - NOT Operator (*N*) 46
  - Rising and Falling Functions (*RISING*, *FALLING*) 47
  - Comparison Instructions 49
- Numerical Processing 50
  - Introduction to Numerical Operations 50
  - Assignment Instructions 50
  - Bit Strings Assignment 51
  - Words Assignment 52
  - Arithmetic Operators on Integers 53
  - Logic Instructions 55
  - Shift Instructions 56
  - BCD/Binary Conversion Instructions 58
  - Single/Double Word Conversion Instructions 59
- Program 60
  - END Instructions 60
  - NOP Instructions 61
  - Jump Instructions 61
  - Conditional Elements 62
  - Loop Elements 64
  - Subroutine Instructions 65
- Floating Point 66

Arithmetic Instructions on Floating Point Objects 67
Trigonometric Instructions 69
Angle Conversion Instructions 70
Integer/Floating Conversion Instructions 71
ASCII 72
ROUND Instructions 72
ASCII to Integer Conversion Instructions 74
Integer to ASCII Conversion Instructions 75
ASCII to Float Conversion Instructions 76
Float to ASCII Conversion Instructions 78
ASCII to Double Word Conversion Instructions 79
Double Word to ASCII Conversion Instructions 81
Stack Operators 82
Stack Instructions (MPS, MRD, MPP) 82
Instructions on Object Tables 83
Word, Double Word, and Floating Point Tables Assignment 84
Table Summing Functions 85
Table Comparison Functions 86
Table Search Functions 87
Table Search Functions for Maximum and Minimum Values 88
Number of Occurrences of a Value in a Table 89
Table Rotate Shift Functions 90
Table Sort Functions 91
Floating Point Table Interpolation (LKUP) Functions 92
MEAN Functions of the Values of a Floating Point Table 95
SWAP Function of the Values of a Word Table 96
Instructions on I/O Objects 96
Read Immediate Digital Embedded Input (READ_IMM_IN) 96
Write Immediate Digital Embedded Output (WRITE_IMM_OUT) 98
Read Immediate Function Block Parameter (READ_IMM) 99
Write Immediate Function Block Parameter (WRITE_IMM) 100
I/O Objects 101
Fast Counter (%FC) 101
High Speed Counter (%HSC) 101
Pulse (%PLS) 101
Pulse Width Modulation (%PWM) 101
Network Objects 102
Network Objects 102
Software Objects 103
Using Function Blocks 103
Function Block Programming Principles 103
Adding a Function Block 104
Configuring a Function Block 106
Timer (%TM) 106
Description 106
Configuration 107
TON: On-Delay Timer 109
TOF: Off-Delay Timer 112
TP: Pulse Timer 116
Programming Example 119

Counter (%C) 119
Description 120
Configuration 121
Programming Example 122
Message (%MSG) and Exchange (EXCH) 124
Overview 124
Description 126
Configuration 129
Programming Example 132
ASCII Examples 133
Modbus Standard Requests and Examples 135
LIFO/FIFO Register (%R) 141
Description 141
Configuration 142
LIFO Register Operation 144
FIFO Register Operation 145
Programming Example 146
Drums (%DR) 146
Description 147
Configuration 148
Programming Example 149
Shift Bit Register (%SBR) 152
Description 152
Configuration 153
Programming Example 154
Step Counter (%SC) 155
Description 155
Configuration 156
Programming Example 157
Schedule Blocks (%SCH) 158
Description 158
Programming and Configuring 161
Real Time Clock (%RTC) 163
Description 163
Configuration 165
PID Function 166
Data Logging 167
Grafcet Steps 169
PTO Objects 169
Motion Task Table (%MT) 169
Pulse Train Output (%PTO) 170
Drive Objects 170
Communication Objects 171
Read Data from a Remote Device (%READ_VAR) 171
Description 171
Function Configuration 175
Programming Example 177
Write Data to a Modbus Device (%WRITE_VAR) 177
Description 177
Function Configuration 179
Programming Example 181

Read and Write Data on a Modbus Device (%WRITE_READ_VAR) 182
Description 182
Function Configuration 184
Programming Example 186
Communication on an ASCII Link (%SEND_RECV_MSG) 187
Description 187
Function Configuration 189
Programming Example 191
Send Receive SMS (%SEND_RECV_SMS) 192
Description 192
Function Configuration 198
Timing Diagrams for Communication Object Function Blocks 204
Examples of Timing Diagrams 204
User-Defined Elements 206
User-Defined Functions 206
User-Defined Function Blocks 206
Clock Functions 207
Clock Functions 207
Time and Date Stamping 207
Setting Date and Time 209
Glossary 213
Index 215

Safety Information

Important Information

Read these instructions carefully, and look at the equipment to become familiar with the device before trying to install, operate, service, or maintain it. The following special messages may appear throughout this documentation or on the equipment to warn of potential hazards or to call attention to information that clarifies or simplifies a procedure.



The addition of this symbol to a “Danger” or “Warning” safety label indicates that an electrical hazard exists which will result in personal injury if the instructions are not followed.



This is the safety alert symbol. It is used to alert you to potential personal injury hazards. Obey all safety messages that follow this symbol to avoid possible injury or death.

⚠ DANGER

DANGER indicates a hazardous situation which, if not avoided, **will result in** death or serious injury.

⚠ WARNING

WARNING indicates a hazardous situation which, if not avoided, **could result in** death or serious injury.

⚠ CAUTION

CAUTION indicates a hazardous situation which, if not avoided, **could result** in minor or moderate injury.

NOTICE

NOTICE is used to address practices not related to physical injury.

Please Note

Electrical equipment should be installed, operated, serviced, and maintained only by qualified personnel. No responsibility is assumed by Schneider Electric for any consequences arising out of the use of this material.

A qualified person is one who has skills and knowledge related to the construction and operation of electrical equipment and its installation, and has received safety training to recognize and avoid the hazards involved.

Before You Begin

Do not use this product on machinery lacking effective point-of-operation guarding. Lack of effective point-of-operation guarding on a machine can result in serious injury to the operator of that machine.

⚠WARNING

UNGUARDED EQUIPMENT

- Do not use this software and related automation equipment on equipment which does not have point-of-operation protection.
- Do not reach into machinery during operation.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

This automation equipment and related software is used to control a variety of industrial processes. The type or model of automation equipment suitable for each application will vary depending on factors such as the control function required, degree of protection required, production methods, unusual conditions, government regulations, etc. In some applications, more than one processor may be required, as when backup redundancy is needed.

Only you, the user, machine builder or system integrator can be aware of all the conditions and factors present during setup, operation, and maintenance of the machine and, therefore, can determine the automation equipment and the related safeties and interlocks which can be properly used. When selecting automation and control equipment and related software for a particular application, you should refer to the applicable local and national standards and regulations. The National Safety Council's Accident Prevention Manual (nationally recognized in the United States of America) also provides much useful information.

In some applications, such as packaging machinery, additional operator protection such as point-of-operation guarding must be provided. This is necessary if the operator's hands and other parts of the body are free to enter the pinch points or other hazardous areas and serious injury can occur. Software products alone cannot protect an operator from injury. For this reason the software cannot be substituted for or take the place of point-of-operation protection.

Ensure that appropriate safeties and mechanical/electrical interlocks related to point-of-operation protection have been installed and are operational before placing the equipment into service. All interlocks and safeties related to point-of-operation protection must be coordinated with the related automation equipment and software programming.

NOTE: Coordination of safeties and mechanical/electrical interlocks for point-of-operation protection is outside the scope of the Function Block Library, System User Guide, or other implementation referenced in this documentation.

Start-up and Test

Before using electrical control and automation equipment for regular operation after installation, the system should be given a start-up test by qualified personnel to verify correct operation of the equipment. It is important that arrangements for such a check are made and that enough time is allowed to perform complete and satisfactory testing.

⚠WARNING

EQUIPMENT OPERATION HAZARD

- Verify that all installation and set up procedures have been completed.
- Before operational tests are performed, remove all blocks or other temporary holding means used for shipment from all component devices.
- Remove tools, meters, and debris from equipment.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

Follow all start-up tests recommended in the equipment documentation. Store all equipment documentation for future references.

Software testing must be done in both simulated and real environments.

Verify that the completed system is free from all short circuits and temporary grounds that are not installed according to local regulations (according to the National Electrical Code in the U.S.A, for instance). If high-potential voltage testing is necessary, follow recommendations in equipment documentation to prevent accidental equipment damage.

Before energizing equipment:

- Remove tools, meters, and debris from equipment.
- Close the equipment enclosure door.
- Remove all temporary grounds from incoming power lines.
- Perform all start-up tests recommended by the manufacturer.

Operation and Adjustments

The following precautions are from the NEMA Standards Publication ICS 7.1-1995:

(In case of divergence or contradiction between any translation and the English original, the original text in the English language will prevail.)

- Regardless of the care exercised in the design and manufacture of equipment or in the selection and ratings of components, there are hazards that can be encountered if such equipment is improperly operated.
- It is sometimes possible to misadjust the equipment and thus produce unsatisfactory or unsafe operation. Always use the manufacturer’s instructions as a guide for functional adjustments. Personnel who have access to these adjustments should be familiar with the equipment manufacturer’s instructions and the machinery used with the electrical equipment.
- Only those operational adjustments required by the operator should be accessible to the operator. Access to other controls should be restricted to prevent unauthorized changes in operating characteristics.

About the Document

Document Scope

This guide describes how to use Function Blocks and Instructions in programs you create with EcoStruxure Machine Expert - Basic software. The descriptions apply to the logic controllers supported by EcoStruxure Machine Expert - Basic.

Validity Note

This document has been updated for the release of EcoStruxure™ Machine Expert - Basic V1.4.

Product Related Information

⚠WARNING

LOSS OF CONTROL

- Perform a Failure Mode and Effects Analysis (FMEA), or equivalent risk analysis, of your application, and apply preventive and detective controls before implementation.
- Provide a fallback state for undesired control events or sequences.
- Provide separate or redundant control paths wherever required.
- Supply appropriate parameters, particularly for limits.
- Review the implications of transmission delays and take actions to mitigate them.
- Review the implications of communication link interruptions and take actions to mitigate them.
- Provide independent paths for control functions (for example, emergency stop, over-limit conditions, and error conditions) according to your risk assessment, and applicable codes and regulations.
- Apply local accident prevention and safety regulations and guidelines.[1]
- Test each implementation of a system for proper operation before placing it into service.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

[1] For additional information, refer to NEMA ICS 1.1 (latest edition), *Safety Guidelines for the Application, Installation, and Maintenance of Solid State Control* and to NEMA ICS 7.1 (latest edition), *Safety Standards for Construction and Guide for Selection, Installation and Operation of Adjustable-Speed Drive Systems* or their equivalent governing your particular location.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Only use software approved by Schneider Electric for use with this equipment.
- Update your application program every time you change the physical hardware configuration.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

General Cybersecurity Information

In recent years, the growing number of networked machines and production plants has seen a corresponding increase in the potential for cyber threats, such as unauthorized access, data breaches, and operational disruptions. You must, therefore, consider all possible cybersecurity measures to help protect assets and systems against such threats.

To help keep your Schneider Electric products secure and protected, it is in your best interest to implement the cybersecurity best practices as described in the Cybersecurity Best Practices document.

Schneider Electric provides additional information and assistance:

- Subscribe to the Schneider Electric security newsletter.
- Visit the Cybersecurity Support Portal web page to:
  - Find Security Notifications.
  - Report vulnerabilities and incidents.
- Visit the Schneider Electric Cybersecurity and Data Protection Posture web page to:
  - Access the cybersecurity posture.
  - Learn more about cybersecurity in the cybersecurity academy.
  - Explore the cybersecurity services from Schneider Electric.

Available Languages of the Document

The document is available in these languages:

- English (EIO0000003289)
- French (EIO0000003290)
- German (EIO0000003291)
- Spanish (EIO0000003292)
- Italian (EIO0000003293)
- Chinese (EIO0000003294)
- Portuguese (EIO0000003295)
- Turkish (EIO0000003296)

Related Documents

Title of Documentation	Reference Number
EcoStruxure Machine Expert - Basic - Operating Guide	EIO0000003281 (ENG) EIO0000003282 (FRA) EIO0000003283 (GER) EIO0000003284 (SPA) EIO0000003285 (ITA) EIO0000003286 (CHS) EIO0000003287 (POR) EIO0000003288 (TUR)
Modicon M221 Logic Controller Advanced Functions - Library Guide	EIO0000003305 (ENG) EIO0000003306 (FRE) EIO0000003307 (GER) EIO0000003308 (SPA) EIO0000003309 (ITA) EIO0000003310 (CHS) EIO0000003311 (POR) EIO0000003312 (TUR)
Modicon M221 Logic Controller - Programming Guide	EIO0000003297 (ENG) EIO0000003298 (FRE) EIO0000003299 (GER) EIO0000003300 (SPA) EIO0000003301 (ITA) EIO0000003302 (CHS) EIO0000003303 (POR) EIO0000003304 (TUR)
Modicon M221 Logic Controller - Hardware Guide	EIO0000003313 (ENG) EIO0000003314 (FRA) EIO0000003315 (GER) EIO0000003316 (SPA) EIO0000003317 (ITA) EIO0000003318 (CHS) EIO0000003319 (POR) EIO0000003320 (TUR)

To find documents online, visit the Schneider Electric download center (www.se.com/ww/en/download/).

Information on Non-Inclusive or Insensitive Terminology

As a responsible, inclusive company, Schneider Electric is constantly updating its communications and products that contain non-inclusive or insensitive terminology. However, despite these efforts, our content may still contain terms that are deemed inappropriate by some customers.

Terminology Derived from Standards

The technical terms, terminology, symbols and the corresponding descriptions in the information contained herein, or that appear in or on the products themselves, are generally derived from the terms or definitions of international standards.

In the area of functional safety systems, drives and general automation, this may include, but is not limited to, terms such as *safety*, *safety function*, *safe state*, *fault*, *fault reset*, *malfunction*, *failure*, *error*, *error message*, *dangerous*, etc.

Among others, these standards include:

Standard	Description
IEC 61131-2:2007	Programmable controllers, part 2: Equipment requirements and tests.
ISO 13849-1:2023	Safety of machinery: Safety related parts of control systems. General principles for design.
EN 61496-1:2020	Safety of machinery: Electro-sensitive protective equipment. Part 1: General requirements and tests.
ISO 12100:2010	Safety of machinery - General principles for design - Risk assessment and risk reduction
EN 60204-1:2006	Safety of machinery - Electrical equipment of machines - Part 1: General requirements
ISO 14119:2013	Safety of machinery - Interlocking devices associated with guards - Principles for design and selection
ISO 13850:2015	Safety of machinery - Emergency stop - Principles for design
IEC 62061:2021	Safety of machinery - Functional safety of safety-related electrical, electronic, and electronic programmable control systems
IEC 61508-1:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: General requirements.
IEC 61508-2:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: Requirements for electrical/electronic/programmable electronic safety-related systems.
IEC 61508-3:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: Software requirements.
IEC 61784-3:2021	Industrial communication networks - Profiles - Part 3: Functional safety fieldbuses - General rules and profile definitions.
2006/42/EC	Machinery Directive
2014/30/EU	Electromagnetic Compatibility Directive
2014/35/EU	Low Voltage Directive

In addition, terms used in the present document may tangentially be used as they are derived from other standards such as:

Standard	Description
IEC 60034 series	Rotating electrical machines
IEC 61800 series	Adjustable speed electrical power drive systems
IEC 61158 series	Digital data communications for measurement and control – Fieldbus for use in industrial control systems

Finally, the term *zone of operation* may be used in conjunction with the description of specific hazards, and is defined as it is for a *hazard zone* or *danger zone* in the *Machinery Directive* (*2006/42/EC*) and *ISO 12100:2010*.

NOTE: The aforementioned standards may or may not apply to the specific products cited in the present documentation. For more information concerning the individual standards applicable to the products described herein, see the characteristics tables for those product references.

Introduction

Overview

This chapter provides you information about how to use the source code examples and the blocks that are required to run many of the examples of operations and assignment instructions given in this document.

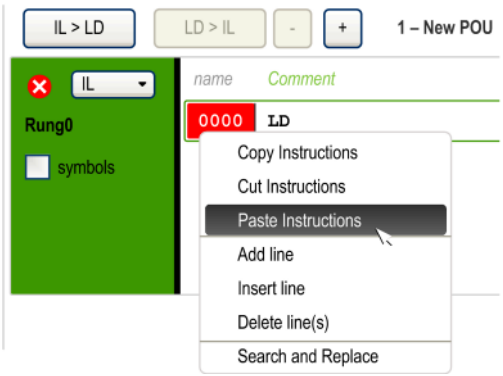
How to Use the Source Code Examples

Overview

Except where explicitly mentioned, the source code examples contained in this document are valid for both the Ladder Diagram and Instruction List programming languages. A complete example may require more than one rung.

Reversibility Procedure

To obtain the equivalent Ladder Diagram source code:

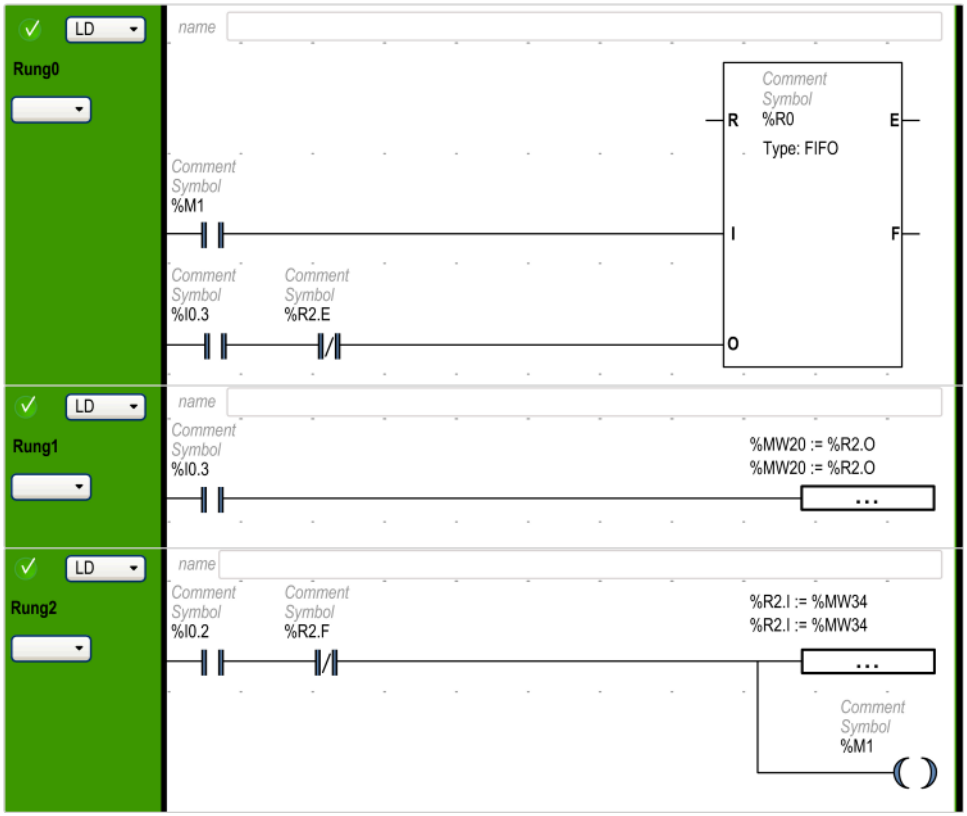
Step	Action
1	Select and copy (**Ctrl+C**) the source code for the first rung of the sample program shown in this manual.
2	In EcoStruxure Machine Expert - Basic, create a new rung by clicking  on the toolbar.
3	In this rung, click the **LD > IL** button to display Instruction List source code.
4	Select the line number **0000**, then right-click and choose **Paste Instructions** to paste the source code into the rung:  **NOTE:** Remember to delete the **LD** instruction from the last line of the rung if you have pasted the instructions by inserting the line(s) before the default LD operator.
5	Click the **IL > LD** button to display the Ladder Diagram source code.
6	Repeat the previous steps for any additional rungs in the sample program.

Example

Instruction List program:

Rung	Source Code
0	BLK %R0 LD %M1 I LD %I0.3 ANDN %R2.E O END_BLK
1	LD %I0.3 [%MW20:=%R2.O]
2	LD %I0.2 ANDN %R2.F [%R2.I:=%MW34] ST %M1

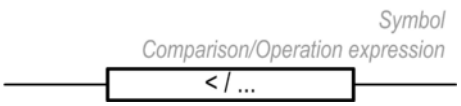
Corresponding Ladder Diagram:



Operation Blocks

Inserting IL Operations and Assignment Instructions in Ladder Diagrams

You can use the **Operation Block** graphical symbol to insert Instruction List operations and assignment instructions into Ladder Diagram rungs:

 Operation block at the end of a rung.	 Operation block anywhere in a rung.

The **Operation Block** graphical symbol can be inserted in any position in a Ladder Diagram rung except the first column, as it cannot be used as the first contact in a rung.

If more than one **Operation Block** graphical symbol is used in a Ladder Diagram rung, they must be placed in series. **Operation Block** instructions cannot be used in parallel.

NOTE: If the application is configured with a functional level of at least **Level 5.0**:

- You can use up to five operands and three levels of parentheses in a operation block. The operands must be of the same object types: words with words, float with float, etc.
- A minimum of 20 memory words (`%MW`) must be available to use multiple operands in the master task. If using multiple operands in a periodic task as well, another 20 memory words must be available.

To insert an operation block graphical symbol in a Ladder Diagram rung:

Step	Action
1	Click the **Operation Block**  button on the toolbar.
2	Click anywhere the rung to insert the **Operation Block**.
3	Click the **Selection mode**  button on the toolbar.
4	Double-click the **operation expression** line. The Smart Coding, page 16 button  appears at the end of the line. Click this button for help selecting a function and with the syntax of the instruction.
5	Type a valid Instruction List operation or assignment instruction and press ENTER. For example: `%MF10 := ((SIN( %MF12 + 60.0 ) + COS( %MF13 )) + %MF10 ) + 1.2` You can modify the expression in online mode. Refer to Online Mode Modifications (see EcoStruxure Machine Expert - Basic, Generic Functions Library Guide).

NOTE: Multiple operand expressions cannot be used in event tasks.

OPER Instruction Syntax

The `OPER` instruction corresponds to an operation block placed anywhere in a rung.

The equivalent *OPER* instruction can be used directly in Instruction List rungs.

OPER [*expression*] where *expression* is a valid expression, containing up to five operands and three levels of parentheses. For example:

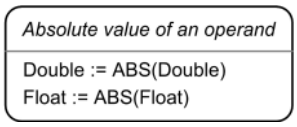
```
OPER [ %MF10 := ((SIN( %MF12 + 60.0 ) + COS( %MF13 )) + %MF10 ) + 1.2]
```

Smart Coding Tooltips in Ladder Diagrams

To help you selecting functions, EcoStruxure Machine Expert - Basic displays tooltips while you type function names in operation blocks.

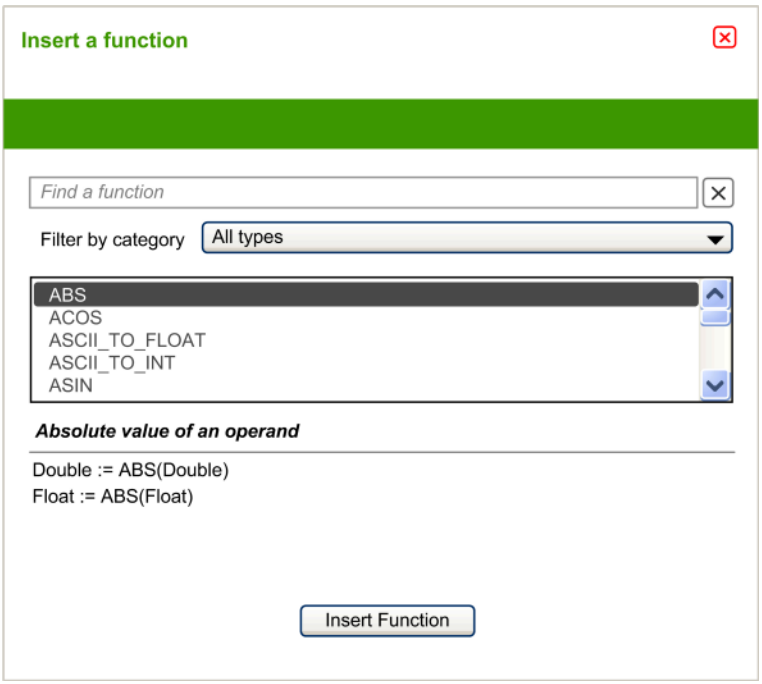
Two types of tooltip are available:

- A list of function names, dynamically updated with the function names that begin with the typed characters. For example, typing “AS” displays *ASCII_TO_FLOAT*, *ASCII_TO_INT*, and *ASIN*.
- Help with the syntax of a function, displayed when you type an opening parenthesis. For example, typing “ABS(“ displays:



Using the Smart Coding Assistant

The Smart Coding Assistant appears when you click the Smart Coding button  in the operation expression line:



Proceed as follows:

Step	Action
1	Optionally, filter the list by category of function: • All types • ASCII • Floating point • I/O objects • Floating Point • Numerical Processing • Table • PID • User-defined function
2	Select a function to add to the expression.
3	Click **Insert Function**.

Getting Help with Syntax

If the syntax of the Instruction List operation or assignment instruction is incorrect, the border of the **operation expression** box turns red. For assistance, either:

- Move the mouse over the **operation expression** line, or
- Select **Tools > Program Messages**.

Comparison Blocks

Inserting IL Comparison Expressions in Ladder Diagrams

You can use the **Comparison Block** graphical symbol to insert Instruction List comparison expressions into Ladder Diagram rungs:



The operands must be of the same object types: words with words, float with float, etc.

Proceed as follows:

Step	Action
1	Click the **Comparison Block**  button on the toolbar.
2	Click anywhere in the rung to insert the **Comparison Block**.
3	Double-click the **Comparison expression** line.
4	Type a valid Instruction List comparison operation and press ENTER. You can modify the expression in online mode. Refer to Online Mode Modifications (see EcoStruxure Machine Expert - Basic, Operating Guide).

NOTE: If the application is configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 6.0**:

- You can use up to five operands and three level of parentheses in a comparison block.
- A minimum of 20 memory words (`%MW`) must be available to use multiple operands in the master task. If using multiple operands in a periodic task as well, another 20 memory words must be available.

NOTE: Multiple operand expressions cannot be used in event tasks.

Getting Help with Syntax

If the syntax of the Instruction List comparison operation is incorrect, the border of the **Comparison expression** box turns red. For assistance, either:

- Move the mouse over the **Comparison expression** line, or
- Select **Tools > Program Messages**.

Language Objects

Objects

Overview

In EcoStruxure Machine Expert - Basic, the term *object* is used to represent an area of logic controller memory reserved for use by an application. Objects can be:

- Simple software variables, such as memory bits and words
- Addresses of digital or analog inputs and outputs
- Controller-internal variables, such as system words and system bits
- Predefined system functions or function blocks, such as timers and counters.

Controller memory is either pre-allocated for certain object types, or automatically allocated when an application is downloaded to the logic controller.

Objects can only be addressed by a program once memory has been allocated. Objects are addressed using the prefix *%*. For example, *%MW12* is the address of a memory word, *%Q0.3* is the address of an embedded digital output, and *%TM0* is the address of a *Timer* function block.

Memory Bit Objects

Introduction

Memory bit objects are bit-type software variables that can be used as operands and tested by Boolean instructions.

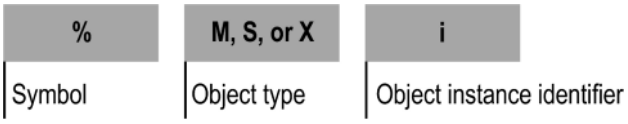
Examples of bit objects:

- Memory bits
- System bits
- Step bits
- Bits extracted from words

The range of valid objects is from 0 to the maximum reference used in your application (see the *programming guide* of your logic controller).

Syntax

Use this format to address memory, system, and step bit objects:



This table describes the elements in the addressing format:

Group	Item	Description
Symbol	%	The percent symbol precedes a software variable.
Object type	M	Memory bits store intermediary values while a program is running.
	S	System bits provide status and control information for the controller.
	X	Step bits provide status of Grafcet step activities.
Object instance identifier	i	The identifier of the object representing their sequential instance in memory. The maximum number of objects depends on the number of objects configured to the limits of available memory. For the maximum amount of available memory, see the *programming guide* of your logic controller.

For information on addressing of I/O bits, refer to I/O objects, page 21.

For information on addressing of bit extracted from word, refer to Extracting Bit from Word Object, page 27.

Description

This table lists and describes memory, system, and step bits objects that are used as operands in Boolean instructions:

Type	Description	Address or Value	Write Access[1]
Immediate values	0 or 1 (False or True)	0 or 1	–
Memory	Memory bits are internal memory areas used to store binary values. **NOTE:** Unused I/O objects cannot be used as memory bits.	`%Mi`	Yes
System	System bits `%S0` to `%S127` allow you to monitor the correct operation of the controller and the correct running of the application program, as well as control certain system-level features.	`%Si`	Depends on i
Grafcet steps	Bits *%X1* to *%Xi* are associated with Grafcet steps. Step bit *Xi* is set to 1 when the corresponding step is active, and set to 0 when the step is deactivated.	`%Xi`	Yes
[1] Written by the program or by using an animation table.			

Example

This table shows some examples of bit object addressing:

Bit Object	Description
`%M25`	Memory bit number 25
`%S20`	System bit number 20
`%X4`	Grafcet step number 4

I/O Objects

Introduction

I/O objects include both bits and words. Each physical input and output is mapped to these objects in internal memory. I/O bit objects can be used as operands and tested by Boolean instructions. I/O word objects can be used in most non-Boolean instructions such as functions and instructions containing arithmetic operators.

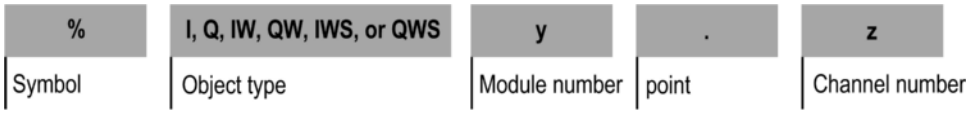
Examples of I/O objects:

- Digital inputs
- Digital outputs
- Analog inputs
- Analog outputs
- Communication inputs and outputs

The range of valid objects is from 0 to the maximum configured and supported for your controller (see the Hardware Guide and Programming Guide for your logic controller).

Syntax

This figure shows the input/output address format:



This table describes the components of the addressing format:

Component	Item	Value	Description
Symbol	`%`	–	The percent symbol precedes an internal address.
Object type	`I`	–	Digital input (bit object)
	`Q`	–	Digital output (bit object)
	`IW`	–	Analog input value (word object)
	`QW`	–	Analog output value (word object)
	`IWS`	–	Analog input channel status (word object)
	`QWS`	–	Analog output channel status (word object)
Module number	`y`	0	Embedded I/O channel on the logic controller, or on a cartridge inserted in the logic controller.
		1...*m* [1]	I/O channel on an expansion module directly connected to the controller.
		m+1...*n* [2]	I/O channel on an expansion module connected using the TM3 Transmitter/Receiver modules.
Channel number	`z`	0...31	I/O channel number on the logic controller or expansion module. The number of available channels depends on the logic controller or expansion module reference.
		*p*0*q* [3]	I/O channel on a cartridge inserted in the logic controller. The number of available channels depends on the cartridge reference.

[1] *m* is the number of local modules configured (maximum 7).

[2] *n* is the number of remote modules configured (maximum n+7). The maximum position number is 14.

[3] *p* is the number of the cartridge in the controller. *q* is the channel number on the cartridge.

Description

This table lists and describes the I/O objects that are used as operands in instructions:

Type	Address or Value	Write Access [1]	Description
Input bits	%Iy.z [2]	No [3]	These bits are the logical images of the electrical states of the physical digital I/O. They are stored in data memory and updated between each scan of the program logic.
Output bits	%Qy.z [2]	Yes	
Input word	%IWy.z [2]	No	These word objects contain the analog value of the corresponding channel.
Output word	%QWy.z [2]	Yes	
Input word status	%IWSy.z [2]	No	These word objects contain the status of the corresponding analog channel.
Output word status	%QWSy.z [2]	No	

[1] Written by the program or by using an animation table.

[2] y is the module number and z is the channel number. Refer to addressing syntax of I/Os, page 22 for descriptions of y and z.

[3] Although you cannot write to input bits, they can be forced.

Examples

This table shows some examples of I/O addressing:

I/O Object	Description
%I0.5	Digital input channel number 5 on the controller (embedded I/O are module number 0).
%Q3.4	Digital output channel number 4 on the expansion module at address 3 (expansion module I/O).
%IW0.1	Analog input 1 on the controller (embedded I/O).
%QW2.1	Analog output 1 on the expansion module at address 2 (expansion module I/O).
%IWS0.101	Analog input channel status of input channel 1 on the first cartridge in the logic controller).
%QWS1.1	Analog output channel status of output channel 1 on the expansion module at address 1 (expansion module I/O).

Word Objects

Introduction

Word objects addressed in the form of 16-bit words are stored in data memory and can contain an integer value from -32768 to 32767 (except for the **Fast Counter** function block which is from 0 to 65535).

Examples of word objects:

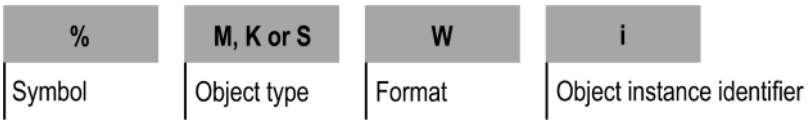
- Immediate values
- Memory words (`%MWi`)
- Constant words (`%KWi`)
- I/O exchange words (`%IWi`, `%QWi`, `%IWSi`, `%QWSi`)
- System words (`%SWi`)
- Function blocks (configuration and/or runtime data)

The range of valid objects is from 0 to the maximum reference used in your application (see the Programming Guide of your logic controller).

For example, if the maximum reference in your application for memory words is `%MW9`, then `%MW0` through `%MW9` are allocated space. `%MW10` in this example is not valid and cannot be accessed either internally or externally.

Syntax

Use this format to address memory, constant, and system words:

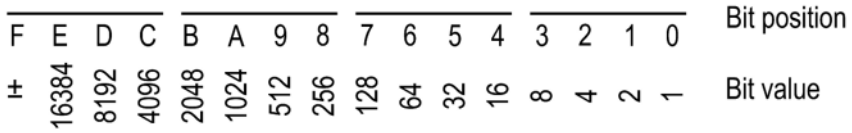


This table describes the elements in the addressing format:

Group	Item	Description
Symbol	%	The percent symbol precedes an internal address.
Object type	M	Memory words store values while a program is running.
	K	Constant words store constant values or alphanumeric messages. Their content can only be written to or modified using EcoStruxure Machine Expert - Basic.
	S	System words provide status and control information for the logic controller.
Format	W	16-bit word.
Object instance identifier	i	The identifier of the object representing their sequential instance in memory. The maximum number of objects depends on the number of objects configured to the limits of available memory. For the maximum amount of available memory, see the *programming guide* of your logic controller.

Format

The contents of the words or values are stored in user memory in 16-bit binary code (two's complement format) using the following convention:



In signed binary notation, by convention, bit 15 is allocated to the sign of the coded value:

- Bit 15 is set to 0: the content of the word is a positive value.
- Bit 15 is set to 1: the content of the word is a negative value (negative values are expressed in two's complement logic).

Words and immediate values (see the Exception List, page 25 for unsigned integers) can be entered or retrieved in the following format:

- Decimal

  Min.: -32768, Max.: 32767 (1579, for example)
- Hexadecimal

  Min.: 16#0000, Max.: 16#FFFF (for example, 16#A536)

  Alternate syntax: #A536
- ASCII format rules as follows:
  - The function reads the most significant byte first.
  - Any ASCII character that is not in the interval [0 - 9] ([16#30 - 16#39]) is considered to be an end character, except for a minus sign '-' (16#2D) when it is placed as the first character.
  - In case of overflow (>32767 or <-32768), the system bit `%S18` (arithmetic overflow or detected error) is set to 1 and 32767 or -32768 value is returned.
  - If the first character of the operand is an "end" character, the value 0 is returned and the bit `%S18` is set to 1.

  For example, "HELLO":
  - `%MW0:="HE"`
  - `%MW1:="LL"`
  - `%MW2:="O"`

Exception List

This table lists the value range of the objects that are unsigned integers:

Object	Value
%SW	0...65535
%FC.V and *%FC.P*	0...65535
%FC.VD and *%FC.PD*	0...4294967295
%HSC.V, *%HSC.P*, *%HSC.S0*, *%HSC.S1*, and *%HSC.C*	0...65535
%HSC.DV, *%HSC.PD*, *%HSC.S0D*, *%HSC.S1D*, and *%HSC.CD*	0...4294967295
%HSC.T	100...1000
%PWM.P	0...32767
%PWM.R	0...100
%PLS.P	0...32767
%PLS.N	0...32767
%PLS.ND	0...2147483647

Other than the objects in the exception list, the data has the following value ranges:

- Words: -32768...32767
- Double words: -2147483648...2147483647

Description

This table describes the word objects:

Words	Description	Address or Value	Write Access[1]
Immediate values	These are integer values that are in the same format as the 16-bit words, which enable values to be assigned to these words.	-	No
	Base 10 (decimal)	-32768 to 32767	
	Base 16 (hexadecimal)	16#0000 to 16#FFFF	
Memory	Used as "working" words to store values during operation in data memory.	`%MWi`	Yes
Constants	Store constants or alphanumeric messages. Their content can be written or modified using EcoStruxure Machine Expert - Basic during configuration and in online mode (see EcoStruxure Machine Expert - Basic, Operating Guide).	`%KWi`	Yes, during configuration and in online mode using Constant word properties (see EcoStruxure Machine Expert - Basic, Operating Guide).
System	These 16-bit words have several functions: • Provide access to data coming directly from the controller by reading `%SWi` words. • Perform operations on the application (for example, adjusting schedule blocks).	`%SWi`	Depends on i
Function blocks	These words correspond to current parameters or values of function blocks.	*%TM2.P*, *%Ci.P*, and so on.	Yes
[1] Written by the program or by using an animation table.			

The maximum number of objects available is determined by the logic controller. Refer to the *programming guide* of your logic controller for maximum number of objects.

Example

This table shows some examples of word object addressing:

Word Object	Description
`%MW15`	Memory word number 15
`%KW26`	Constant word number 26
`%SW30`	System word number 30

Extracting Bits from Word Objects

This table describes how to extract 1 of the 16 bits from the following word objects:

Word Object	Address or Value	Write Access[1]
Memory	%MWi:Xk	Yes
System	%SWi:Xk	Depends on i
Constant	%KWi:Xk	No
Input value	%IWy.z:Xk[2]	No
Output value	%QWy.z:Xk[2]	Yes
Input status	%IWSy.z:Xk[2]	No
Output status	%QWSy.z:Xk[2]	Yes

[1] Written by the program or by using an animation table.

[2] For information on I/O word objects, refer to Addressing I/O objects (see EcoStruxure Machine Expert - Basic, Operating Guide).

Xk indicates the bit number that has to be extracted from the word object. For example, *%MW0.X3*; bit stored at the third sequential position of the memory word *%MW0* will be extracted.

Floating Point and Double Word Objects

Introduction

A floating point object is a real number; that is, a number with a fractional part (for example: 3.4E+38, 2.3, or 1.0).

A double word consists of 4 bytes stored in data memory and containing a two's complement value from -2147483648 to +2147483647.

Floating point and double word operations are not supported by all logic controllers.

For compatibility, refer to the *Programming Guide* of your logic controller.

Floating Point Format and Value

The floating format used is the standard IEEE STD 734-1985 (equivalent to IEC 559). The length of the words is 32 bits, which corresponds to single decimal point floating numbers.

This table shows the format of a floating point value:

Bit 31	Bits {30...23}	Bits {22...0}
Sign of the exponent	Exponent	Significand

Representation precision is from 2...24 to display floating point numbers; it is not necessary to display more than 6 digits after the decimal point.

NOTE: The value 1285 is interpreted as a whole value; in order for it to be recognized as a floating point value, it must be written thus: 1285.0

Limit Range of Arithmetic Functions on Floating Point Objects

This table describes the limit range of arithmetic functions on floating point objects:

Arithmetic Function		**Limit Range and Invalid Operations**	
Type	**Syntax**	**NaN (Not a Number)**	**Infinity**
Square root of an operand	*SQRT(x)*	x < 0	x > *SQRT(3.402824E +38)* is the maximum number that can be obtained
Power of an integer by a real *EXPT(%MF,%MW)*	*EXPT(y, x)* (where: `x^y = %MW^%MF`)	x < 0 and y = fractional number	*X ^ Y* > 3.402824E+38
Base 10 logarithm	*LOG(x)*	x < 0	Calculation possible until maximum value of x is obtained (3.402824E+38)
Natural logarithm	*LN(x)*	x < 0	No maximum value of x. *LN(3.402824E+38)* is the maximum number that can be obtained
Natural exponential	*EXP(x)*	No limit in the real range	x > 88.72283 If x < -103.973 the result is 0

Validity Check

When the result is not within the valid range, the system bit `%S18` is set to 1.

The status word `%SW17` indicates the cause of an error detected in a floating operation.

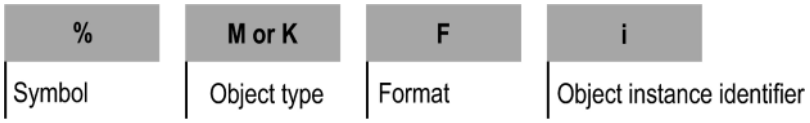
Different bits of the word `%SW17`:

`%SW17:X0`	Invalid operation, result is not a number (NaN)
`%SW17:X1`	Reserved
`%SW17:X2`	Division by 0, result is invalid (-Infinity or +Infinity)
`%SW17:X3`	Result greater in absolute value than +3.402824E+38, result is invalid (-Infinity or +Infinity)
`%SW17:X4 to X15`	Reserved

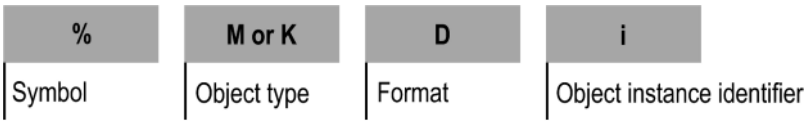
This word is reset to 0 by the system following a cold start, and can also be reset by the program for reusage purposes.

Syntax

Use this format to address memory and constant floating point objects:



Use this format to address memory and constant double word objects:



This table describes the elements in the addressing format:

Group	Item	Description
Symbol	%	The percent symbol precedes an internal address.
Object type	M	Memory objects are used to store intermediary values while a program is running.
	K	Constants are used to store constant values or alphanumeric messages (only for double words).
Format	F	32-bit floating point object.
	D	32-bit double word object.
Object instance identifier	i	The identifier representing instance (sequential position) of an object in memory. Refer to the *programming guide* of your logic controller for maximum number of objects.

Description of Floating Point and Double Word Objects

This table describes floating point and double word objects:

Type of Object	Description	Address	Write Access
Immediate values	Integers (double word) or decimal (floating point) numbers with identical format to 32-bit objects.	-	No
Memory floating point	Objects used to store values during operation in data memory.	%MFi	Yes
Memory double word		%MDi	Yes
Floating constant value	Used to store constants.	%KFi	Yes, during configuration and in online mode using Constant word properties (see EcoStruxure Machine Expert - Basic, Operating Guide)
Double constant		%KDi	Yes, during configuration and in online mode using Constant word properties (see EcoStruxure Machine Expert - Basic, Operating Guide)

NOTE: The maximum number of objects is determined by the logic controller; refer to the Programming Guide for your hardware platform for details.

Example

This table shows some examples of floating point and double word objects addressing:

Object	Description
%MF15	Memory floating point object number 15
%KF26	Constant floating point object number 26
%MD15	Memory double word number 15
%KD26	Constant double word number 26

Possibility of Overlap Between Objects

Single, double length and floating words are stored in the data space in one memory zone. Thus, the floating word %MFi and the double word %MDi correspond to the single length words %MWi and %MWi+1; the word %MWi containing the least significant bits and the word %MWi+1 the most significant bits of the word %MFi.

This table shows how floating and double memory words overlap:

Floating and Double	Odd Address	Memory Words
%MF0 / %MD0		%MW0
	%MF1 / %MD1	%MW1
%MF2 / %MD2		%MW2
	%MF3 / %MD3	%MW3
%MF4 / %MD4		%MW4
	...	%MW5
...		...
	%MFi / %MDi	%MWi
%MFi+1 / %MDi+1		%MWi+1

This table shows how floating and double constants overlap:

Floating and Double	Odd Address	Memory Words
%KF0 / %KD0		%KW0
	%KF1 / %KD1	%KW1
%KF2 / %KD2		%KW2
	%KF3 / %KD3	%KW3
%KF4 / %KD4		%KW4
	...	%KW5
...		...
	%KFi / %KDi	%KWi
%KFi+1 / %KDi+1		%KWi+1

Example:

%MF0 corresponds to %MW0 and %MW1. %KF43 corresponds to %KW43 and %KW44.

Structured Objects

Introduction

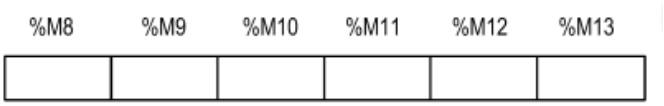
Structured objects are combinations of adjacent objects. EcoStruxure Machine Expert - Basic supports the following types of structured objects:

- Bit strings
- Tables of words
- Tables of double words
- Tables of floating words

Bit Strings

Bit strings are a series of adjacent object bits of the same type and of a defined length (L). Bit strings are referenced starting on byte boundaries.

Example: Bit string `%M8:6`



NOTE: `%M8:6` is valid (8 is a multiple of 8) while `%M10:16` is invalid (10 is not a multiple of 8).

Bit strings can be used with the Assignment instruction, page 51.

Available Types of Bits

Available types of bits for bit strings:

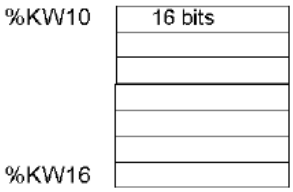
Type	Address	Write Access
Digital input bits	`%I0.0:L` or `%I1.0:L`(1)	No
Digital output bits	`%Q0.0:L` or `%Q1.0:L`(1)	Yes
System bits	`%Si:L` with i multiple of 8	Depending on i
Grafcet step bits	`%Xi:L` with i multiple of 8	Yes (by program)
Memory bits	`%Mi:L` with i multiple of 8	Yes
(1) Only I/O bits 0 to 16 can be read in a bit string. For logic controllers with 24 or 32 I/O channels, bits over 16 cannot be read in a bit string. **L** Represents the length of the structured objects (bit strings, table of words, table of double words, and table of floating words).		

The number of bits is determined by the logic controller; refer to the Programming Guide for your hardware platform for details.

Tables of Words

Word tables are a series of adjacent words of the same type and of a defined length (L, maximum value is 255).

Example: Word table `%KW10:7`



Word tables can be used with the Assignment instruction, page 84.

Available Types of Words

Available types of words for word tables:

Type	Address	Write Access
Memory words	`%MWi:L`(1)	Yes
Constant words	`%KWi:L`(1)	No
System words	`%SWi:L`(1)	Depending on i
Analog outputs	`%QWi.j:L`(2)	Yes
Analog inputs	`%IWi.j:L`(2)	No
EtherNet/IP input assemblies	`%QWEi:L`(2)	Yes
EtherNet/IP output assemblies	`%IWEi:L`(2)	No
Modbus mapping input registers	`%IWMi:L`(2)	No
Modbus mapping output registers	`%QWMi:L`(2)	Yes
IOScanner input registers	`%IWNa.b.c:L`(2)	No
IOScanner output registers	`%QWNa.b.c:L`(2)	Yes

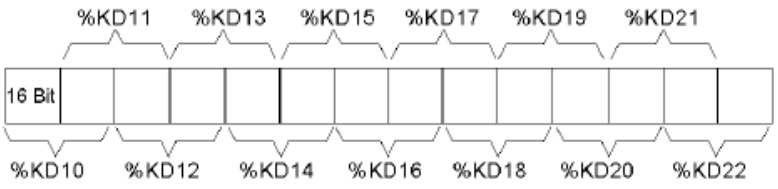
(1) The number of words L is determined by the logic controller; refer to the Programming Guide for your hardware platform for details.

(2) The number of words L depends on the configuration of each object. Refer to I/O Objects, page 21 or Network Objects, page 102.

Tables of Double Words

Double word tables are a series of adjacent words of the same type and of a defined length (L, maximum value is 255).

Example: Double word table `%KD10:7`



Double word tables can be used with the Assignment instruction, page 41.

Available Types of Double Words

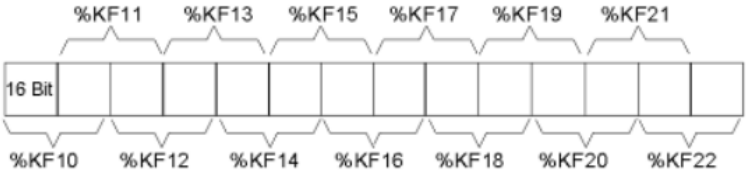
Available types of words for double word tables:

Type	Address	Write Access
Memory words	%MDi:L	Yes
Constant words	%KDi:L	No

Tables of Floating Words

Floating word tables are a series of adjacent words of the same type and of a defined length (L, maximum value is 255).

Example: Floating point table %KF10:7



Floating point tables can be used with the Assignment instruction, page 41.

Types of Floating Words Available

Available types of words for floating word tables:

Type	Address	Write Access
Memory words	%MFi:L	Yes
Constant words	%KFi:L	No

Indexed Objects

Introduction

An indexed object is a single word, double word, floating point or memory bit object with an indexed object address. There are 2 types of object addressing:

- Direct addressing
- Indexed addressing

Direct Addressing

A direct address of an object is set and defined when a program is written.

Example: %M26 is a memory bit with the direct address 26.

Indexed Addressing

An indexed address of an object provides a method of modifying the address of an object by adding an index to the direct address of the object. The content of the index is added to the direct address of the object. The index is defined by a memory word %MWi.

Example: %MW108[%MW2] is a word with an address consisting of the direct address 108 plus the contents of word %MW2.

If word %MW2 has a value of 12, writing to %MW108[%MW2] is equivalent to writing to %MW120 (108 plus 12).

Objects Available for Indexed Addressing

This table describes the available types of objects for indexed addressing:

Type	Address	Write Access
Memory words	%MWi[MWj]	Yes
	%MWi[%PARAMj][1]	Yes
Constant words	%KWi[%MWj]	No
Memory double words	%MDi[MWj]	Yes
Double constant words	%KDi[%MWj]	No
Memory floating points	%MFi[MWj]	Yes
Constant floating points	%KFi[%MWj]	No
Memory bits	%Mi[%MWj]	Yes

i Object instance identifier that represents instance (sequential position) of an object in memory. Refer to the programming guide of your logic controller for maximum number of objects.

j Object instance identifier of the index object whose content has to be added to the direct address of some other object.

[1] Valid only in the **RUNG** of a User-Defined Function (UDF)/User-Defined Function Block (UDFB) where %PARAM*j* is defined.

Indexed objects can be used with the Assignment instruction, page 50 and in Comparison instructions, page 49.

This type of addressing enables series of objects of the same type (such as memory words and constants) to be scanned in succession, by modifying the content of the index object in the program.

Index Overflow System Bit %S20

An overflow of the index occurs when the address of an indexed object exceeds the limits of the memory zone containing the same type of object. In summary:

- The object address plus the content of the index is less than 0.
- The object address plus the content of the index is greater than the largest word directly referenced in the application.

In the event of an index overflow, system bit %S20 is set to 1 and the object is assigned an index value of 0.

NOTE: Monitor for overflow. Your program must read %S20 for possible processing. You should then confirm that it is reset to 0.

%S20 (initial status = 0):

- On index overflow: set to 1 by the controller.
- Acknowledgment of overflow: manually set to 0 in the program after modifying the index.

Some of the detected mathematical errors could have significant impact on the execution of your application. Monitor for these potential errors, and program instructions to appropriately control the execution of your application should one or more of these detected errors occur. The impact of any of these detected errors depends upon configuration, equipment used, and the program instructions executed prior to and after detection of the potential error or errors.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Write programming instructions to test the validity of operands intended to be used in mathematical operations.
- Avoid using operands of different data types in mathematical operations.
- Always monitor the system bits assigned to indicate invalid mathematical results.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

Function Block Objects

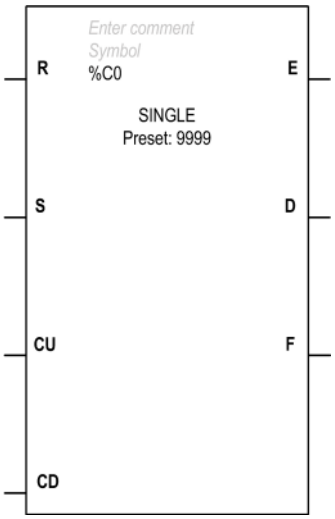
Introduction

A function block is a reusable object that accepts one or more input values and returns one or more output values. A function block is called through an instance (a copy of a function block with its own dedicated name and variables). Each function block instance has a persistent state (outputs and internal variables) from one call to another.

NOTE: The function blocks (*%FC*, *%HSC*, *%PLS*, and *%PWM*) and Status Alarm drive their inputs and outputs (*%I0.x* and *%Q0.x*, affected in the configuration) directly with no relation with the controller cycle. The image bits (*%I0.x* and *%Q0.x*) are not updated by the controller. So, these inputs and outputs bits cannot be used directly in the user program, and an animation table using these inputs/outputs cannot show the current states of these inputs/outputs.

Example

This illustration shows a Counter function block:



Bit Objects

Bit objects correspond to the function block outputs. These bits can be accessed by Boolean test instructions using either of the following methods:

- Directly (for example, `LD E`) if they are wired to the block in reversible programming, page 103.
- By specifying the block type (for example, `LD %Ci.E`).

Inputs can be accessed in the form of instructions.

Word Objects

Word objects correspond to specified parameters and values as follows:

- Block configuration parameters: some parameters are accessible by the program (for example, pre-selection parameters), and some are inaccessible by the program (for example, time base).
- Current values: for example, `%Ci.V`, the current count value.

Double Word Objects

Double-word objects increase the computational capability of your logic controller while executing system functions, such as fast counters (*%FC*), high speed counters (*%HSC*) and pulse generators (*%PLS, %PWM*).

To address the 32-bit double word objects used with function blocks, append the character D to the original syntax of the standard word objects.

This example shows how to address the current value of a fast counter in standard format and in double word format:

- `%FCi.V` is the current value of the fast counter in standard format.
- `%FCi.VD` is the current value of the fast counter in double word format.

Instructions

Boolean Processing

Aim of This Section

This section provides an introduction to Boolean processing instructions.

Boolean Instructions

Introduction

Boolean instructions can be compared to Ladder Diagram language elements. These instructions are summarized in this table:

Item	Operator	Instruction Example	Description
Test elements	The load (*LD*) instruction is equivalent to the first open contact connected to a power rail of a ladder diagram. Logical *AND* and *OR* instructions are equivalent to open contacts after the first contact connected to the power rail of a ladder diagram.	`LD %I0.0`	Contact is closed when bit `%I0.0` is at state 1.
Action elements	The store (*ST*) instruction is equivalent to a coil.	`ST %Q0.0`	The associated bit object takes a logical value of the bit accumulator (result of previous logic).

The Boolean result of the test elements is applied to the action elements as shown by the following instructions:

Rung	Instruction

0	`LD %I0.0`
	`AND %I0.1`
	`ST %Q0.0`

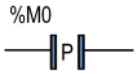
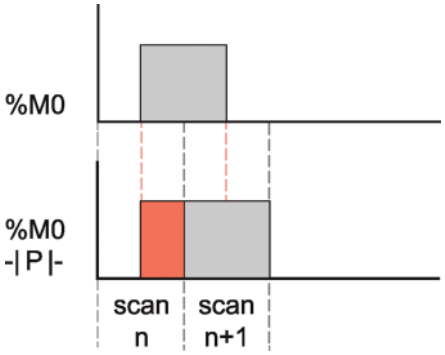
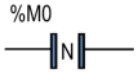
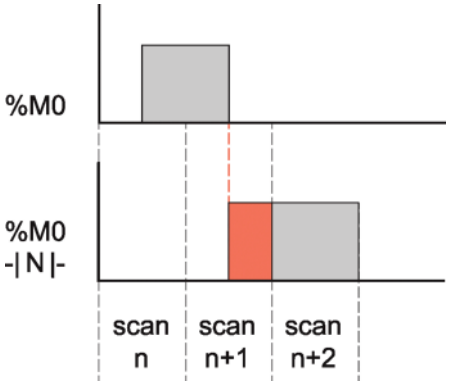
NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Testing Controller Inputs

Boolean test instructions can be used to detect rising or falling edges on the controller inputs. An edge is detected when the state of an input has changed between “scan n-1” and the current “scan n”. This edge remains detected during the current scan.

Edge Detection

This table summarizes the instructions and timing for detecting rising and falling edges:

Edge	IL Instruction	Ladder Diagram	Timing Diagram
Rising Edge Contact	`LDR %M0`		
Falling Edge Contact	`LDF %M0`		
 The objects are only updated at the beginning of the next master cycle following edge detection. A state change of a memory bit (e.g. %M0) is indicated one scan delayed.			

NOTE: Rising and falling edge contacts can only be used on input bits (%I) and memory bits (%M) objects.

Edge detection can also be performed using the *RISING*  and *FALLING*  functions, page 47. To use them, the application must be configured with a Functional Level of at least **Level 6.0**.

One of the differences between the rising edge/falling edge contacts and *RISING*/*FALLING* functions is the scan applied on the program:

- The *LDR*/*LDF* instructions indicate rising/falling edges with a delay of one scan.
- The *RISING*/*FALLING* functions indicate rising/falling edges in the same cycle in which they occur.

Rising Edge Detection

The Load Rising Edge (*LDR*) instruction is equivalent to a Rising Edge detection contact. The Rising Edge detects a change of the input value from 0 to 1.

A positive transition sensing contact is used to detect a Rising Edge as seen in this example:

Rung	Instruction
0	`LDR  %I0.0`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Falling Edge Detection

The Load Falling Edge (*LDF*) instruction is equivalent to a Falling Edge detection contact. The Falling Edge detects a change of the controlling input from 1 to 0.

A negative transition sensing contact is used to detect a Falling Edge as seen in this example:

Rung	Instruction
0	`LDF %I0.0`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Load Operators (*LD, LDN, LDR, LDF*)

Introduction

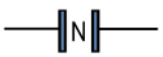
Load operators *LD*, *LDN*, *LDR*, and *LDF* correspond respectively to open, close, rising edge, and falling edge contacts. *LDR* and *LDF* are used only with logic controller inputs and memory words.

Syntax

This table lists the types of load operators with Ladder Diagram equivalents and operands:

Operators	Ladder Diagram Equivalent	Operands

LD		`0/1`
		`%I, %Q, %M, %S, %X, %BLK.x`
		`%IW:Xk, %QW:Xk, %IWS:Xk, %QWS:Xk, %MW:Xk, %SW:Xk, %KW:Xk`
		`%Mi[%MWj]`[1]

LDN		
LDR		`%I, %M`
LDF		
[1] Functional level >= 10.0		

Coding Examples

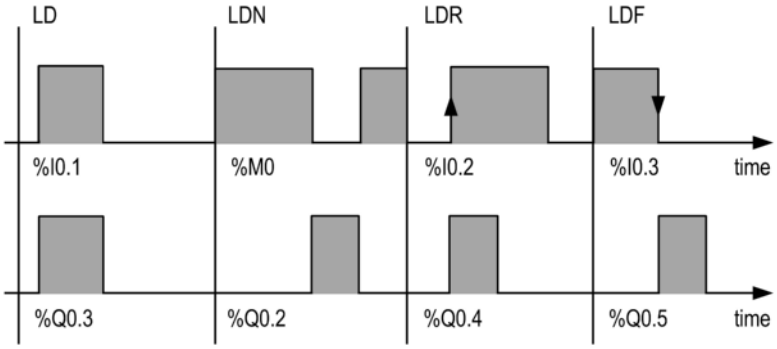
Examples of Load instructions:

Rung	Instruction
0	`LD %I0.1` `ST %Q0.3`
1	`LDN %M0` `ST %Q0.2`
2	`LDR %I0.1` `ST %Q0.4`
3	`LDF %I0.3` `ST %Q0.5`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

The following diagram illustrates the timing, and the effect on the output, of the code from the coding example(s):



NOTE: The memory bit (*%M*) bit edge detection is performed between master task scans.

Assignment Operators (*ST, STN, R, S*)

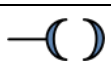
Introduction

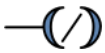
The Assignment operators *ST*, *STN*, *S*, and *R* correspond respectively to the direct, inverse, set, and reset coils.

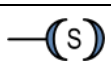
Syntax

This table lists the types of Assignment operators with Ladder Diagram equivalents and operands:

Operators	Ladder Diagram Equivalent	Operands

ST		`%Q, %M, %BLK.x`
		`%QW:Xk, %MW:Xk, %S(1), %SW:Xk(1)`
		`%Mi[%MWj](2)`

STN		

S		`%Q, %M, %S, %X, %BLK.x`
		`%QW:Xk, %MW:Xk, %SW:Xk(1)`
		`%Mi[%MWj](2)`

R		
(1) %S or %SW:Xk are on non-read-only system objects. **(2)** Functional level >= 10.0		

Coding Examples

Examples of Assignment instructions:

Rung	Instruction

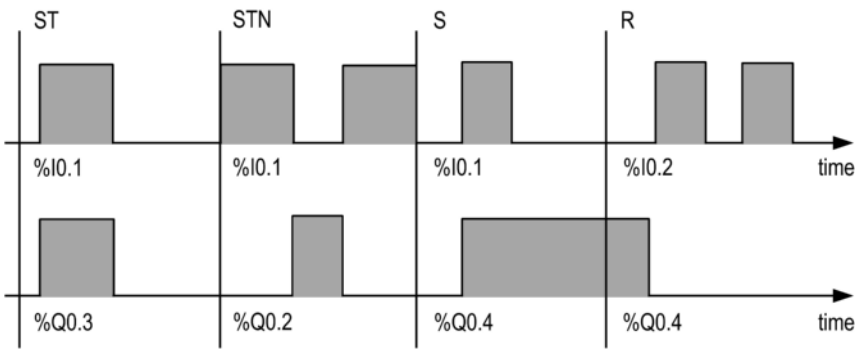
0	`LD %I0.1`
	`ST %Q0.3`
	`STN %Q0.2`
	`S %Q0.4`

1	`LD %I0.2` `R %Q0.4`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

The following diagram illustrates the timing, and the effect on the output, of the code from the coding example(s):



Logical AND Operators (*AND, ANDN, ANDR, ANDF*)

Introduction

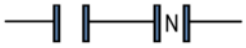
The AND operators perform a logical *AND* operation between the operand (or its inverse, rising edge or falling edge) and the Boolean result of the preceding instruction.

Syntax

This table lists the types of *AND* operators with Ladder Diagram equivalents and operands:

Operators	Ladder Diagram Equivalent	Operands

AND		`0/1`
		`%I, %Q, %M, %S, %X, %BLK.x`
		`%IW:Xk, %QW:Xk, %IWS:Xk, %QWS:Xk, %MW:Xk, %SW:Xk, %KW:Xk`
		`%Mi[%MWj]`(1)

ANDN		
ANDR		`%I, %M`
ANDF		
(1) Functional level >= 10.0		

Coding Examples

Examples of logical *AND* instructions:

Rung	Instruction

0	`LD %I0.1`
	`AND %M1`
	`ST %Q0.3`

1	`LD %M0`
	`ANDN %I0.0`
	`ST %Q0.2`

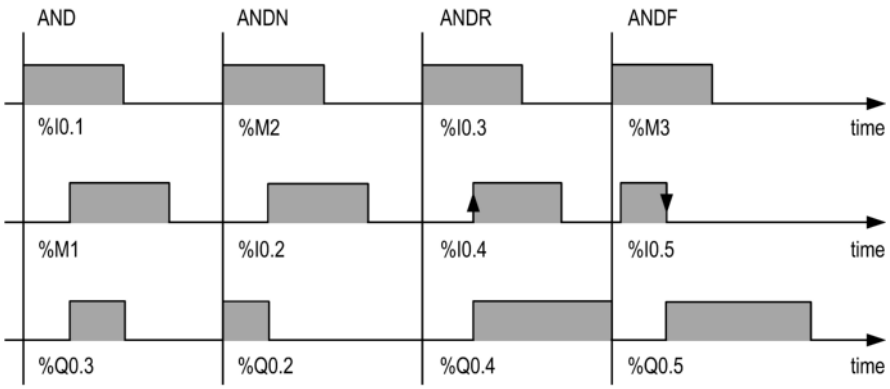
2	`LD %I0.3`
	`ANDR %I0.4`
	`S %Q0.4`

3	`LD %M3`
	`ANDF %I0.5`
	`S %Q0.5`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

The following diagram illustrates the timing, and the effect on the output, of the code from the coding example(s):



Logical OR Operators (*OR, ORN, ORR, ORF*)

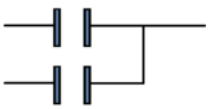
Introduction

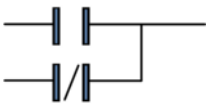
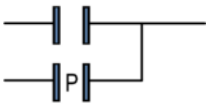
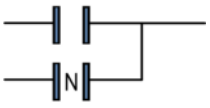
The *OR* operators perform a logical *OR* operation between the operand (or its inverse, rising edge or falling edge) and the Boolean result of the preceding instruction.

Syntax

This table lists the types of *OR* operators with Ladder Diagram equivalents and operands:

Operators	Ladder Diagram Equivalent	Operands

OR		`0/1`
		`%I, %Q, %M, %S, %X, %BLK.x`
		`%IW:Xk, %QW:Xk, %IWS:Xk, %QWS:Xk, %MW:Xk, %SW:Xk, %KW:Xk`
		`%Mi[%MWj]`(1)

ORN		
ORR		`%I, %M`
ORF		

(1) Functional level >= 10.0

Coding Examples

Examples of logical *OR* instructions:

Rung	Instruction

0	`LD %I0.1`
	`OR %M1`
	`ST %Q0.0`

1	`LD %I0.2`
	`ORN %M2`
	`ST %Q0.1`

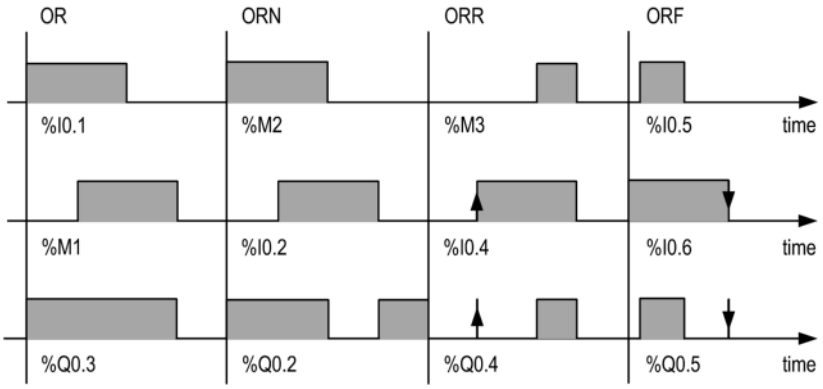
2	`LD %M0`
	`ORR %I0.3`
	`S %Q0.5`

3	`LDF %I0.5`
	`ORF %I0.6`
	`S %Q0.0`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

The following diagram illustrates the timing, and the effect on the output, of the code from the coding example(s):



Exclusive OR Operators (*XOR, XORN, XORR, XORF*)

Introduction

The *XOR* operator performs an exclusive OR operation between the operand and the Boolean result of the operator instruction.

The *XORN* operator performs an exclusive OR operation between the inverse of the operand and the Boolean result of the preceding instruction.

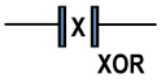
The *XORR* operator performs an exclusive OR operation between the rising edge of the operand and the Boolean result of the preceding instruction.

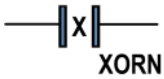
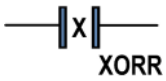
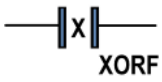
The *XORF* operator performs an exclusive OR operation between the falling edge of the operand and the Boolean result of the preceding instruction.

Syntax

This table lists the types of *XOR* operators and operands:

Operators	Ladder Diagram Equivalent	Operands

XOR		`%I, %Q, %M, %S, %X, %BLK.x`
		
		`%IW:Xk, %QW:Xk, %IWS:Xk, %QWS:Xk, %MW:Xk, %SW:Xk, %KW:Xk`

XORN		
XORR		`%I, %M`
XORF		

Coding Examples

Using the *XOR* instruction:

Rung	Instruction

0	`LD %I0.1`
	`XOR %M1`
	`ST %Q0.3`

Equivalent logical instructions of the *XOR* operator:

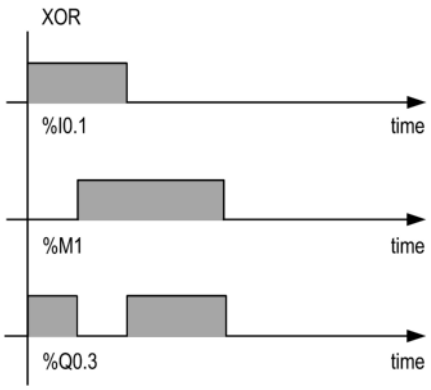
Rung	Instruction

0	`LD %I0.1`
	`ANDN %M1`
	`OR( %M1`
	`ANDN %I0.1`
	`)`
	`ST %Q0.3`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

The following diagram illustrates the timing, and the effect on the output, of the code from the coding example(s):

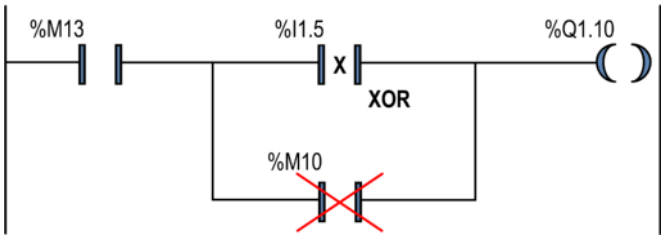


Special Cases

Do not insert:

- *XOR* contacts in the first position of a rung.
- *XOR* contacts in parallel with other Ladder Diagram elements (see the following example).

As shown in this example, inserting an element in parallel with the *XOR* contact will generate a validation error:



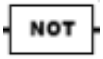
NOT Operator (*N*)

Introduction

The *NOT* (*N*) operator has an implicit operand; that being, the result stored in the boolean accumulator. The NOT negates the value of the accumulator.

Syntax

This table shows the *N* operator:

Operator	Ladder Diagram Equivalent	Operands
N		Not applicable.

Coding Examples

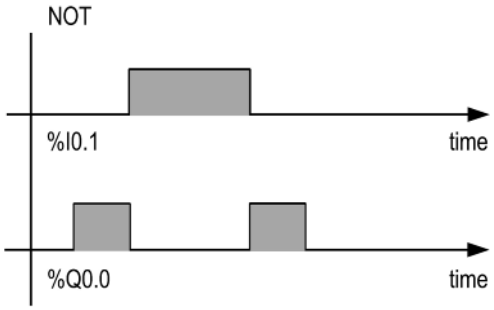
Example of *NOT* instruction:

Rung	Instruction
0	LD %I0.1 N ST %Q0.0

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

The following diagram illustrates the timing, and the effect on the output, of the code from the coding example(s):



Rising and Falling Functions (*RISING*, *FALLING*)

Introduction

The `RISING` and a `FALLING` functions evaluate respectively a rising and a falling edge of the expression that immediately precedes it.

These functions have an implicit operand; that is, the result of the preceding expression, which is stored in the boolean accumulator.

You cannot place these functions in the first column of a Ladder language rung or branch, and they cannot be the first instruction in an IL rung.

NOTE: To use the *RISING* and *FALLING* functions, the application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 6.0**.

Syntax

Function	Ladder Diagram Equivalent	Operands
`RISINGn` [1]		Not applicable.
`FALLINGn` [1]		Not applicable.

[1] n is an integer incremented each time a rising or a falling edge is inserted.

This integer is calculated automatically if you:

- Define no index.
- Enter an incorrect index.
- Delete the index.
- Change the index.

Coding Examples

Function	Rung	Instruction
RISING	0	LD %M0 RISING0 ST %Q0.0
FALLING	1	LD %I0.1 FALLING0 ST %Q0.7

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

You can insert up to:

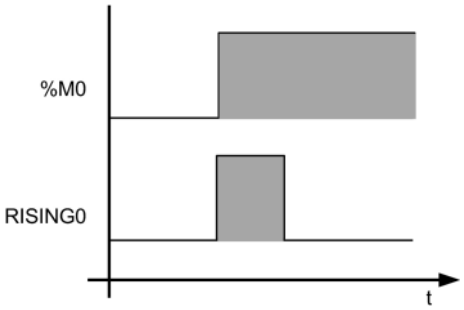
- 32 instructions of each type if the functional level is less than **10.1**
- 255 instructions of each type if the functional level is greater or equal to **10.1**

You cannot use the RISING and FALLING functions immediately after any of the following instructions:

- *AND*
- *ANDN*
- *OR*
- *ORN*

Timing Diagram

The following diagram illustrates the timing of the example above for one master task scan time:



Comparison Instructions

Introduction

Comparison operators are used to compare up to 5 operands with 3 levels of parentheses.

This table lists the types of Comparison operators:

Operator	Function
>	Test if Op1 is greater than Op2
>=	Test if Op1 is greater than or equal to Op2
<	Test if Op1 is less than Op2
<=	Test if Op1 is less than or equal to Op2
=	Test if Op1 is equal to Op2
<>	Test if Op1 is different from Op2

Syntax

The following describes Instruction List syntax. You can insert Instruction List comparison expressions, page 18 in Ladder Diagram rungs using a **Comparison Block** graphical element.

Syntax for Comparison instructions:

Operator	Syntax
>, >=, <, <=, =, <>	LD [Op1 operator Op2] AND [Op1 operator Op2] OR [Op1 operator Op2]

This table gives details of operands:

Type	Op1	Op2
Words	`%MWi, %KWi, %IW, %QWi, %SWi, %BLK.x`	Immediate value, `%MWi, %KWi, %IW, %QW, %IWSi, %QWSi, %SWi, %BLK.x, %MWi[%MWi], %KWi[%MWi]`
Double words	`%MDi, %KDi`	Immediate value, `%MDi, %KDi, %MDi[%MWi], %KD[%MWi]`
Floating point words	`%MFi, %KFi`	Immediate floating point value, `%MFi, %KFi, %MFi[%MWi], %KFi[%MWi]`

NOTE: Comparison instructions can be used within parentheses.

Coding Examples

The comparison is executed inside square brackets following instructions *LD*, *AND*, and *OR*. The result is 1 when the comparison requested is true.

Examples of Comparison instructions:

Rung	Instruction
0	LD %I0.2 AND [%MW10>100] ST %Q0.3
1	LD %M0 AND [%MW20<%KW35] ST %Q0.4
2	LD %I0.2 OR [%MF30>=%MF40] ST %Q0.5

An example of using a Comparison instruction within parentheses:

Rung	Instruction
0	LD %M0 AND([%MF20>10.0] OR %I0.0) ST %Q0.1

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Numerical Processing

Aim of This Section

This section provides an introduction to Numerical Processing.

Introduction to Numerical Operations

At a Glance

Numerical instructions generally apply to 16-bit words and to 32-bit double words. They are written between square brackets. If the result of the preceding logical operation was true (Boolean accumulator = 1), the numerical instruction is executed. If the result of the preceding logical operation was false (Boolean accumulator = 0), the numerical instruction is not executed and the operand remains unchanged.

Assignment Instructions

Introduction

Assignment instructions are used to load Op2 (operand 2) into Op1 (operand 1).

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for Assignment instructions:

Operator	Syntax
:=	[Op1 := Op2] Op1 takes the value of Op2

Assignment operations can be performed on:

- Bit strings
- Words
- Double words
- Floating word
- Word tables
- Double word tables
- Floating word tables
- Pulse train output objects

Bit Strings Assignment

Introduction

Operations can be performed on the following bit strings:

- Bit string to bit string (Example 1)
- Bit string to word (Example 2) or double word (indexed)
- Word or double word (indexed) to bit string (Example 3)
- Immediate value to bit string

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for bit string assignments:

Operator	Syntax
:=	[Op1 := Op2] Op1 takes the value of Op2

This table gives details for the operands:

Type	Op1	Op2
Word, double word	%MWi,%QWi, %SWi %MWi[%MWi], %MDi, %MDi[%MWi] %Mi:L, %Qi:L, %Si:L, %Xi:L %TMi.P, %Ci.P, %Ri.I, %Ri.O, %FCi.P, %PLSi.P, %PWMi.P %Ci.PD, %FCi.PD	Immediate value, %MWi, %KWi, %IW, %QWi, %IWSi, %QWSi, %SWi,%BLK.x, %MWi[%MWi], %KWi[%MWi], %MDi[%MWi], %KDi[%MWi], %Mi:L,%Qi:L, %Si:L, %Xi:L, %Ii:L %TMi.P, %Ci.P, %Ri.I, %Ri.O, %FCi.P, %PLSi.P, %PWMi.P %Ci.PD, %FCi.PD

NOTE: The abbreviation %BLK.x (for example, %C0.P) is used to describe any function block word.

Structure

Examples of bit string assignments:

Rung	Instruction
0	LD 1 [%Q0.0:8:=%M64:8]
1	LD %I0.2 [%MW100:=%M0:16]
2	LDR %I0.3 [%MW104:16:=%KW0]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Usage rules:

- For bit string to word assignment: the bits in the string are transferred to the word starting on the right (first bit in the string to bit 0 in the word); and the word bits which are not involved in the transfer (length ≤16) are set to 0.
- For word to bit string assignment: The word bits are transferred from the right (word bit 0 to the first bit in the string).

Words Assignment

Introduction

Assignment operations can be performed on the following words and double words:

- Word (indexed) to word (2, for example) (indexed or not)
- Double word (indexed) to double word (indexed or not)
- Immediate whole value to word (Example 3) or double word (indexed or not)
- Bit string to word or double word
- Floating point (indexed or not) to floating point (indexed or not)
- Word or double word to bit string
- Immediate floating point value to floating point (indexed or not)

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for word assignments:

Operator	Syntax
:=	[Op1 := Op2] Op1 takes the value of Op2

This table gives details of operands:

Type	Op1	Op2
Word, double word, bit string	%BLK.x, %MWi, %MWi[%PARAMj], %QWi, %SWi %MWi[%MWj], %MDi, %MDi[%MWj], %Mi:L, %Qi:L, %Si:L, %Xi:L	Immediate value, %MWi, %MWi[%PARAMj], %KWi, %IW, %QWi, %IWSi, QWSi, %SWi, %MWi[%MWj], %KWi[%MWi], %MDi, %MDi[%MWj], %KDi, %KDi[%MWj], %Mi:L, %Qi:L, %Si:L, %Xi:L, %Ii:L
Floating point	%MFi, %MFi[%MWj]	Immediate floating point value, %MFi, %MFi[%MWj], %KFi, %KFi[%MWj]

NOTE: The abbreviation `%BLK.x` (for example, `R3.I`) is used to describe any function block word. For bit strings `%Mi:L`, `%Si:L`, and `%Xi:L`, the base address of the first of the bit string must be a multiple of 8 (0, 8, 16, ..., 96, ...).

Structure

Examples of word assignments:

Rung	Instruction
0	`LD 1` `[%SW112:=%MW100]`
1	`LD %I0.2` `[%MW0[%MW10]:=%KW0[%MW20]]`
2	`LD %I0.3` `[%MW10:=100]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Arithmetic Operators on Integers

Introduction

Arithmetic operators are used to perform arithmetic operations between 2 integer operands or on 1 integer operand.

This table lists the types of Arithmetic operators:

Operator	Function
+	Add 2 operands
-	Subtract 2 operands
*	Multiply 2 operands
/	Divide 2 operands
REM	Remainder of division of the 2 operands
SQRT	Square root of an operand
INC	Increment an operand
DEC	Decrement an operand
ABS	Absolute value of an operand

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for Arithmetic instructions:

Operator	Syntax
+,-,,/,REM*	[Op1 := Op2 operator Op3]
INC, DEC	[operator Op1]
SQRT (1)	[Op1 := SQRT(Op2)]
ABS (1)	[Op1 := ABS(Op2)]

This table gives details of operands:

Type	Op1	Op2 and Op3 [1]
Words	%MWi, %QWi, %SWi, %BLK.x [2], %MWi[%MWj [3]], %MWi[%PARAMj] [4]	Immediate value, %MWi, %KWi, %IWi, %QWi, %IWSi, %QWSi, %SWi, %BLK.x [2] %MWi[%MWj [3]], %MWi[%PARAMj] [4]
Double words	%MDi, %BLK.x	Immediate value, %MDi, %KDi, %BLK.x [2]

[1] With this operator, Op2 cannot be an immediate value. The ABS function can only be used with double words (%MD and %KD) and floating points (%MF and %KF). So, OP1 and OP2 must be double words or floating points.

[2] %BLK.x represents all block objects.

[3] Functional level >= 5.0.

[4] Functional level >= 10.0.

Structure

Examples of Arithmetic instructions:

Rung	Instruction
0	LD %M0 [%MW0:=%MW10+10]
1	LD %I0.2 [%MW0:=SQRT(%MW10)]
2	LDR %I0.3 [%MW10:=32767]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Special Cases

Addition

- Overflow during word operation

  If the result exceeds the capacity of the result word, bit %S18 (overflow) is set to 1 and the result is not significant (see rung 1 of application example, page 55). The user program manages bit %S18.

  NOTE: For double words, the limits are -2147483648 and 2147483647.

Multiplication

- Overflow during operation

  If the result exceeds the capacity of the result word, bit %S18 (overflow) is set to 1 and the result is not significant.

Division / remainder

- Division by 0

  If the divider is 0, the division is impossible and system bit %S18 is set to 1. The result is then incorrect.

- Overflow during operation

  If the division quotient exceeds the capacity of the result word, bit %S18 is set to 1.

Square root extraction

- Overflow during operation

  Square root extraction is only performed on positive values. Thus, the result is positive. If the square root operand is negative, system bit %S18 is set to 1 and the result is incorrect.

Some of the detected mathematical errors could have significant impact on the execution of your application. Monitor for these potential errors, and program instructions to appropriately control the execution of your application should one or more of these detected errors occur. The impact of any of these detected errors depends upon configuration, equipment used, and the program instructions executed prior to and after detection of the potential error or errors.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Write programming instructions to test the validity of operands intended to be used in mathematical operations.
- Avoid using operands of different data types in mathematical operations.
- Always monitor the system bits assigned to indicate invalid mathematical results.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

NOTE: The user program must manage system bits `%S17` and `%S18`. These are set to 1 by the controller and must be reset by the program so that they can be reused (see previous page for example).

Application Example

Overflow during addition:

Rung	Instruction
0	`LD %M0` `[%MW0:=%MW1+%MW2]`
1	`LDN %S18` `[%MW10:=%MW0]`
2	`LD %S18` `[%MW10 :=32767]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

If `%MW1` =23241 and `%MW2`=21853, the result would be (45094), which cannot be expressed in 1 signed 16-bit word. Therefore, bit `%S18` is set to 1 and the value in *%MW0* (-20442) is incorrect. In this example when the result is greater than 32767, its value is fixed at 32767.

Logic Instructions

Introduction

The Logic operators can be used to perform a logical operation between 2 word operands or, in the case of logical *NOT*, on 1 word operand.

This table lists the types of Logic instructions:

Instruction	Function
AND	*AND* (bit-wise) between 2 operands
OR	Logic *OR* (bit-wise) between 2 operands
XOR	Exclusive *OR* (bit-wise) between 2 operands
NOT	Logic complement (bit-wise) of an operand

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for Logic instructions:

Operator	Syntax	Op1	Op2 and Op3
AND, OR, XOR	[Op1 := Op2 operator Op3]	%MWi, %QWi, %SWi, %BLK.x	Immediate value (1), %MWi, %KWi, %IWi, %QWi, %IWSi, %QWSi, %SWi, %BLK.x
NOT	[Op1 := NOT(Op2)]		
(1) With *NOT*, Op2 cannot be an immediate value.			

Structure

Examples of Logic instructions:

Rung	Instruction
0	LD %M0 [%MW0:=%MW10 AND 16#00FF]
1	LD 1 [%MW0:=%KW5 OR %MW10]
2	LD %I0.3 [%MW102:=NOT(%MW100)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Application Example

Logical *AND* instruction:

```
[%MW15:=%MW32 AND %MW12]
```

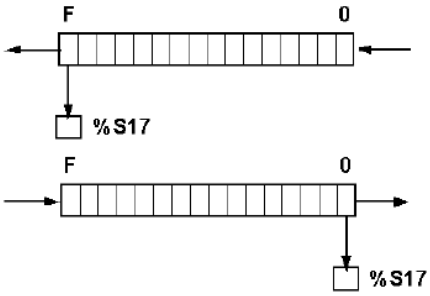
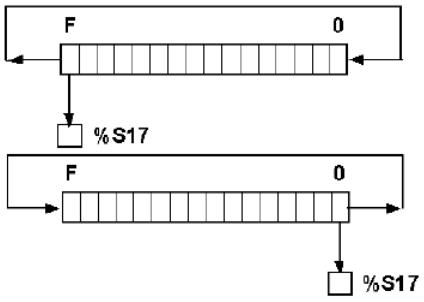
When %MW32 = 0001 1011 (binary) (27 (decimal)) and %MW12 = 0011 0110 (binary) (54 (decimal)) then the result will be %MW15 = 0001 0010 (binary) (18 (decimal)).

Shift Instructions

Introduction

Shift instructions move bits of an operand a specified number of positions to the right or to the left.

This table lists the types of Shift instructions:

Instruction	Function	
Logic shift		
`SHL(op2,n)`	Logic shift of *n* positions to the left.	
`SHR(op2,n)`	Logic shift of *n* positions to the right.	
Rotate shift		
`ROL(op2,n)`	Rotate shift of *n* positions to the left.	
`ROR(op2,n)`	Rotate shift of *n* positions to the right.	
Where *n* is an integer immediate value for: • word: 1...16 inclusive • double word: 1...32 inclusive.		

NOTE: System bit `%S17` indicates the value of the last ejected bit.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for Shift instructions:

Operator	Syntax
SHL, *SHR*	[Op1 := operator (Op2,n)]
ROL, *ROR*	
Where *n* is an integer immediate value for: • word: 1...16 inclusive • double word: 1...32 inclusive.	

This table gives details of operands:

Types	Op1	Op2
Words	`%MWi, %QWi, %SWi` `%BLK.x`	`%MWi, %KWi, %IWi, %QWi, %IWSi, %QWSi, %SWi, %BLK.x`
Double words	`%MDi` `%BLK.x`	`%MDi, %KDi` `%BLK.x`

Structure

Examples of Shift instructions:

Rung	Instruction
0	LDR %I0.1 [%MW0:=SHL(%MW10,5)]
1	LDR %I0.2 [%MW10:=ROR(%KW9,8)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

BCD/Binary Conversion Instructions

Introduction

Conversion instructions perform conversion between different representations of numbers.

This table lists the types of BCD/Binary Conversion instructions:

Instruction	Function
BTI	BCD to Binary conversion
ITB	Binary to BCD conversion

Review of BCD Code

Binary Coded Decimal (BCD) represents a decimal digit (0 to 9) by coding 4 binary bits. A 16-bit word object can thus contain a number expressed in 4 digits (0000 - 9999), and a 32-bit double word object can therefore contain an eight-figure number.

During conversion, system bit %S18 is set to 1 if the value is not BCD. This bit must be tested and reset to 0 by the program.

BCD representation of decimal numbers:

Deci-mal	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Examples:

- Word %MW5 expresses the BCD value 2450 which corresponds to the binary value: 0010 0100 0101 0000
- Word %MW12 expresses the decimal value 2450 which corresponds to the binary value: 0000 1001 1001 0010

Word %MW5 is converted to word %MW12 by using instruction *BTI*.

Word %MW12 is converted to word %MW5 by using instruction *ITB*.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for BCD/Binary Conversion instructions:

Operator	Syntax
BTI, *ITB*	[Op1 := operator (Op2)]

This table gives details of operands:

Types	Op1	Op2
Words	%MWi, %QWi, %SWi %BLK.x	%MWi, %KWi, %IWi, %QWi, %IWSi, %QWSi, %SWi, %BLK.x
Double word	%MDi %BLK.x	%MDi, %KDi %BLK.x

Structure

Examples of BCD/Binary Conversion instructions:

Rung	Instruction
0	LD %M0 [%MW0:=BTI(%MW10)]
1	LD %I0.2 [%MW10:=ITB(%KW9)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Application Example

The *BTI* instruction is used to process a setpoint value at controller inputs via BCD encoded thumb wheels.

The *ITB* instruction is used to display numerical values (for example, the result of a calculation, the current value of a function block) on BCD coded displays.

Single/Double Word Conversion Instructions

Introduction

This table describes instructions used to perform conversions between single and double words:

Instruction	Function
LW	LSB of double word extracted to a word.
HW	MSB of double word extracted to a word.
CONCATW	Concatenates 2 words into a double word.
DWORD	Converts a 16-bit word into a 32-bit double word.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for single/double word Conversion instructions:

Operator	Syntax	Op1	Op2	Op3
LW, HW	Op1 = operator (Op2)	`%MWi`	`%MDi, %KDi, %BLK.x`	[-]
CON-CATW	Op1 = operator (Op2, Op3))	`%MDi, %BLK.x`	`%MWi, %KWi,` immediate value	`%MWi, %KWi,` immediate value
DWORD	Op1 = operator (Op2)	`%MDi, %BLK.x`	`%MWi, %KWi`	[-]

Structure

Examples of single/double word Conversion instructions:

Rung	Instruction
0	`LD %M0` `[%MW0:=HW(%MD10)]`
1	`LD %I0.2` `[%MD10:=DWORD(%KW9)]`
2	`LD %I0.3` `[%MD11:=CONCATW(%MW10,%MW5)]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Program

Aim of This Section

This section provides an introduction to program instructions.

END Instructions

Introduction

The *END* instructions define the end of the execution of a program scan.

END, ENDC, and ENDCN

Four different *END* instructions are available:

- *END*: unconditional end of program
- *ENDC*: end of program if Boolean result of preceding test instruction is 1
- *ENDCN*: end of program if Boolean result of preceding test instruction is 0
- *ENDT*: end of transition rung in a Grafcet (SFC) program (only valid in a transition rung).

By default (normal mode) when the end of program is activated, the outputs are updated and the next scan is started.

If scanning is periodic, when the end of period is reached the outputs are updated and the next scan is started.

The *END* instruction ends the current level:

- If it is activated in a subroutine, a user-defined function or a user-defined function block, it terminates it and returns to the calling program.
- If it is activated in a master, periodic or event task, it terminates the present task.

Examples

Example of an unconditional *END* instruction:

Rung	Instruction
0	`LD %M1` `ST %Q0.1`
1	`LD %M2` `ST %Q0.2`
2	`END`

Example of a conditional *END* instruction:

Rung	Instruction
0	`LD %I0.0` `ST %Q0.0`
1	`LD %I0.1` `ST %Q0.1`
2	`LD %I0.2` `ENDC`
3	`LD %I0.3` `ST %Q0.2`
4	`END`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

NOP Instructions

Introduction

The *NOP* instructions do not perform any operation. Use them to "reserve" lines in a program so that you can insert instructions later without modifying the line numbers.

Jump Instructions

Introduction

Jump instructions cause the execution of a program to be interrupted immediately and to be continued from the line after the program line containing label `%Li` (i = maximum module number).

JMP, JMPC, and JMPCN

3 different Jump instructions are available:

- *JMP*: unconditional program jump
- *JMPC*: program jump if Boolean result of preceding logic is 1
- *JMPCN*: program jump if Boolean result of preceding logic is 0

Examples

Examples of Jump instructions:

Rung	Instruction
0	LD %M15 JMPC %L8
1	LD [%MW24<%MW12] ST %Q0.3 JMPC %L12
2	%L8: LD %M12 AND %M13 ST %M12 JMPC %L12
3	LD %M11 S %Q0.0
4	%L12: LD %I0.0 ST %Q0.4

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Guidelines

- Jump instructions are not allowed between parentheses, and must not be placed between the instructions AND, OR, and a close parenthesis instruction ")".
- The label can only be placed before an LD, LDN, LDR, LDF, or BLK instruction.
- The label number of label `%Li` must be defined only once in a program.
- The program jump is performed to a line of programming which is downstream or upstream. When the jump is upstream, pay attention to the program scan time. Extended scan time can trigger the watchdog timer.

Conditional Elements

Description

Conditional elements allow you to code conditions in your program. The elements may only be programmed while in offline mode.

This feature contains the three following elements:

- `IF`
- `ELSE`
- `ENDIF`

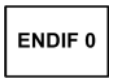
You can insert one element per rung.

NOTE: The application must be configured with a functional level of at least **Level 6.0**.

Maximum Number of Elements

The maximum number of elements is up to 128 less the number of %L declared in your application.

Element Description

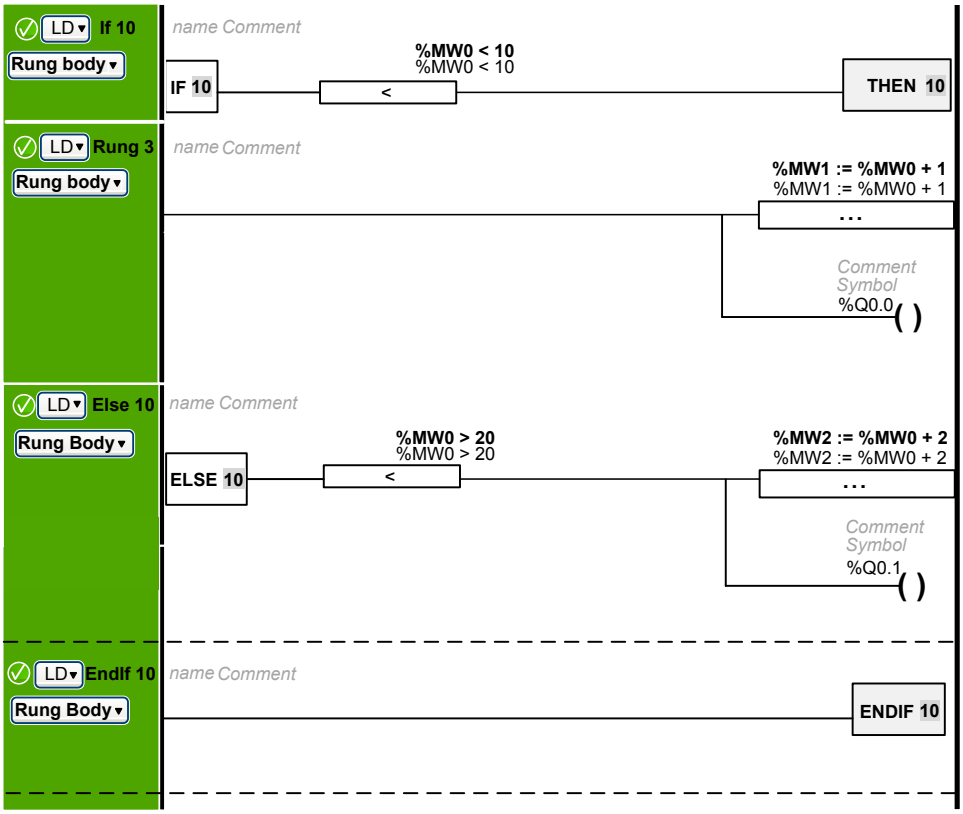
Element	IL Instruction	Ladder Diagram	Description
IF	IF0...THEN0		Must be at the beginning of a rung. The THEN element is automatically added. You cannot modify this element.
ELSE	ELSE0		Only if an IF element is defined in a preceding rung. Must be at the beginning of a rung.
ENDIF	ENDIF0		Must be at the end of a rung. You cannot insert other elements in the same rung.

Configuring a Condition Instruction

Step	Action
1	In an empty rung, insert the IF element by clicking  →  →IF. Insert the condition to be evaluated between IF and THEN elements.
2	Optionally you can change the index. If you modify the index of one element, you must assign the same index to the other elements. The elements with the same index must be in the same POU.
3	Configure your program. When the condition between IF and THEN is TRUE, the subsequent rungs of your program between THEN and the next conditional statement (ELSE or ENDIF) are evaluated. Otherwise, the subsequent rungs are skipped and not evaluated.
4	Optionally insert the ELSE element by clicking  →  →ELSE. When the condition between IF and THEN is FALSE, the portion of the program between ELSE and ENDIF is evaluated. This includes program statements in the same rung as the ELSE element. Otherwise, this portion of the program is skipped and not evaluated.
5	In the last rung, insert the ENDIF element by clicking  →  →ENDIF.

In the example below:

- When the IF condition is TRUE, that is, %MW0 is less than 10:
  1. Rung 3 is evaluated.
  2. The ELSE10 rung is skipped. %MW2 and %Q0.1 retain their last values.
- When the IF condition is FALSE, that is, %MW0 is greater than or equal to 10:
  1. Rung 3 is skipped. %MW1 and %Q0.0 retain their last values.
  2. The ELSE10 rung is evaluated, that is, the operations on %MW2 and %Q0.1 are executed when %MW0 is greater than 20.



Loop Elements

Description

Loop elements allow you to code a sequence of instructions in your program. The elements may only be programmed while in offline mode.

This feature contains the two following elements:

- FOR
- ENDFOR

You can insert one element per rung.

NOTE: The application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 6.0**.

Maximum Number of Elements

The maximum number of elements is up to 128 less the number of %L declared in your application.

Element Description

Element	IL Instruction	Ladder Diagram	Description
FOR	FOR0		Must be at the beginning of a rung. Click ... to configure.
ENDFOR	ENDFOR0		Must be at the end of a rung. You cannot insert other elements in the same rung.

Configuring a Loop

Step	Action
1	In an empty rung, insert the FOR element by clicking → LOOP →FOR.
2	Optionally you can change the index. If you modify the index of one element, you must assign the same index to the other element. The elements with the same index must be in the same POU.
3	Click ... to configure the FOR element. **Result**: The **FOR Assistant** window appears.
4	In the last rung, insert the ENDFOR element by clicking → LOOP →ENDFOR.

Configuring the FOR Element

Label	Description
Loop counter	Type a %MWx variable.
Initial value	Type a %MWx variable or a value between -32768...32767.
Sign	• **<** • **<=** • **=** • **>=** • **>** • **<>**
End value	Type a %MWx variable or a value between -32768...32767.
Incrementation step	Type a %MWx variable or a value between -32768...32767.

Subroutine Instructions

Introduction

The Subroutine instructions cause a program to perform a subroutine and then return to the main program at the point from which the subroutine was called.

Procedure

A subroutine is created in a Free POU. Refer to Free POUs (see EcoStruxure Machine Expert - Basic, Operating Guide) for information on creating a Free POU and subroutine, and defining the subroutine number. Also, refer to Managing POUs (see EcoStruxure Machine Expert - Basic, Operating Guide) for more information on managing POUs with task and rungs.

Calling a subroutine in 3 steps:

1	The *SRn* instruction calls the subroutine referenced by a Free POU SRn if the result of the preceding boolean instruction is 1.
2	The subroutine is referenced by a Free POU SRn, where n is the number of the subroutine.
3	The subroutine instruction must be written as a Free POU independent of the main program.

For more information about subroutines, refer to Creating Periodic Task (see EcoStruxure Machine Expert - Basic, Operating Guide).

Examples

Example of instructions containing a Subroutine:

Rung	Instruction
0	LD %M15 AND %M5 ST %Q0.0
1	LD [%MW24>%MW12] SR1
2	LD %I0.4 AND %M13 ST %Q0.1 END

Example of a Subroutine instruction (*SR1*):

Rung	Instruction
0 (SR1)	LD %I0.0 ST %Q0.0

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Guidelines

- A subroutine cannot call up another subroutine. Attempting to call a subroutine within a Free POU will generate a compiler error.
- Subroutine instructions are not allowed between parentheses, and must not be placed between the instructions *AND*, *OR*, and a close parenthesis instruction ")".
- Care should be taken when an Assignment instruction is directly follows a subroutine call in IL. This is because the subroutine may change the content of the Boolean accumulator. Therefore upon return, it could have a different value than before the call.

Floating Point

Aim of This Section

This section describes the advanced instructions of floating point.

Arithmetic Instructions on Floating Point Objects

Introduction

These instructions are used to perform an arithmetic operation between 2 floating point operands or on 1 floating point operand:

Instruction	Purpose
+	Addition of 2 operands
-	Subtraction of 2 operands
*	Multiplication of 2 operands
/	Division of 2 operands
LOG	Base 10 logarithm
LN	Natural logarithm
SQRT	Square root of an operand
ABS	Absolute value of an operand
TRUNC	Whole part of a floating point value
EXP	Natural exponential
EXPT	Power of an integer by a real

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Operators and syntax of arithmetic instructions on floating point:

Operators	Syntax
+, - *, /	Op1 := Op2 operator Op3
SQRT, ABS, TRUNC, LOG, EXP, LN	Op1 := operator (Op2)
EXPT	Op1 := operator (Op2,Op3)

Operands of arithmetic instructions on floating point:

Operators	Op1	Op2	Op3
+, - *, /	`%MFi`	`%MFi`, `%KFi`, immediate value	`%MFi`, `%KFi`, immediate value
SQRT, ABS, LOG, EXP, LN	`%MFi`	`%MFi`, `%KFi`	[-]
TRUNC	`%MFi`, `%MDi`	`%MFi`, `%KFi`	[-]
EXPT	`%MFi`	`%MFi`, `%KFi`	`%MWi`, `%KWi`, immediate value
NOTE: EcoStruxure Machine Expert - Basic prevents the use of function with a `%MWi` as Op1.			

Structure

Example of arithmetic instruction:

Rung	Instruction
0	LD %M0 [%MF0:=%MF10+129.7]
1	LD %I0.2 [%MF1:=SQRT(%MF10)]
2	LDR %I0.3 [%MF2:=ABS(%MF20)]
3	LDR %I0.4 [%MF3:=TRUNC(%MF2)]
4	LD %M1 [%MF4:=LOG(%MF10)]
5	LD %I0.5 [%MF5:=LN(%MF20)]
6	LD %I0.0 [%MF6:=EXP(%MF30)]
7	LD %I0.1 [%MF7:=EXPT(%MF40,%MW52)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Rules of Use

- Operations on floating point and integer values cannot be directly mixed. Conversion operations, page 70 convert into one or other of these formats.
- The system bit %S18 is managed in the same way as integer operations, page 70, the word %SW17 indicates the cause of the detected error.
- When the operand of the function is an invalid number (for example, logarithm of a negative number), it produces an indeterminate or infinite result and changes bit %S18 to 1. The word %SW17 indicates the cause of the detected error.

  NOTE: For the TRUNC instruction, the system bit %S17 is not affected.

Application Examples for TRUNC Instruction with %MDi

This table shows examples of TRUNC instruction when%MDi is used to store the result:

Example	Result
TRUNC(3.5)	3
TRUNC(324.18765)	324
TRUNC(927.8904)	927
TRUNC(-7.7)	-7
TRUNC(45.678E+20)	2 147 483 647 (maximum signed double word) [1] %S18 is set to 1
TRUNC(-94.56E+13)	- 2 147 483 648 (minimum signed double word) [1] %S18 is set to 1
[1] This example applies to the *TRUNC* instruction when used with %MDi. (When used with %MFi, the *TRUNC* instruction has no overflow and therefore has no maximum/minimum limits.)	

Trigonometric Instructions

Introduction

These instructions enable the user to perform trigonometric operations:

SIN	sine of an angle expressed in radian	*ASIN*	arc sine (result within $-\frac{\pi}{2}$ and $\frac{\pi}{2}$)
COS	cosine of an angle expressed in radian	*ACOS*	arc cosine (result within 0 and π)
TAN	tangent of an angle expressed in radian	*ATAN*	arc tangent (result within $-\frac{\pi}{2}$ and $\frac{\pi}{2}$)

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Operators, operands, and syntax of instructions for trigonometric operations:

Operators	Syntax	Op1	Op2
SIN, COS, TAN, ASIN, ACOS, ATAN	Op1:=operator(Op2)	`%MFi`	`%MFi, %KFi`

Structure

Example of Trigonometric instructions:

Rung	Instruction
0	`LD %M0` `[%MF0:=SIN(%MF10)]`
1	`LD %I0.0` `[%MF1:=TAN(%MF20)]`
2	`LD %I0.3` `[%MF2:=ATAN(%MF30)]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Rules of Use

- When the operand of the function is an invalid number (for example, the arc cosine of a number greater than 1), it produces an indeterminate or infinite result and changes bit `%S18` to 1. The word `%SW17` indicates the cause of the detected error.
- The functions *SIN/COS/TAN* allow as a parameter an angle between -4096π and $+4096\pi$ but their precision decreases progressively for angles outside the range -2π and $+2\pi$ because of the imprecision introduced by the modulo 2π carried out on the parameter before any operation.

Angle Conversion Instructions

Introduction

These instructions are used to carry out conversion operations:

DEG_TO_RAD	Conversion of degrees into radian, the result is the value of the angle between 0 and 2π
RAD_TO_DEG	Conversion of an angle expressed in radian, the result is the value of the angle 0...360 degrees

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Operators, operands, and syntax of conversion instructions

Operators	Syntax	Op1	Op2
DEG_TO_RAD RAD_TO_DEG	Op1:=operator(Op2)	`%MFi`	`%MFi, %KFi`

Structure

Example of conversion instructions:

Rung	Instruction
0	`LD %M0` `[%MF0:=DEG_TO_RAD(%MF10)]`
1	`LD %M2` `[%MF2:=RAD_TO_DEG(%MF20)]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Rules of Use

The angle to be converted must be between -737280.0 and +737280.0 (for *DEG_TO_RAD* conversions) or between -4096π and $+4096\pi$ (for *RAD_TO_DEG* conversions).

For values outside these ranges, the displayed result will be + NaN, the `%S18` and `%SW17:X0` bits being set to 1.

Integer/Floating Conversion Instructions

Introduction

4 conversion instructions are offered:

INT_TO_REAL	Conversion of an integer word to floating
DINT_TO_REAL	Conversion of a double word (integer) to floating
REAL_TO_INT	Conversation of a floating to integer word (the result is the nearest algebraic value)
REAL_TO_DINT	Conversation of a floating to double integer word (the result is the nearest algebraic value)

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Operators and syntax (conversion of an integer word to floating):

Operators	**Syntax**
INT_TO_REAL	Op1=INT_TO_REAL(Op2)

Operands (conversion of an integer word to floating):

Op1	**Op2**
`%MFi`	`%MWi,%KWi`

Example: integer word conversion to floating: 147 to 1.47e+02

Operators and syntax (double conversion of integer word to floating):

Operators	**Syntax**
DINT_TO_REAL	Op1=DINT_TO_REAL(Op2)

Operands (double conversion of integer word to floating):

Op1	**Op2**
`%MFi`	`%MDi,%KDi`

Example: integer double word conversion to floating: 68905000 to 6.8905e+07

Operators and syntax (floating conversion to integer word or integer double word):

Operators	**Syntax**
REAL_TO_INT	Op1=operator(Op2)
REAL_TO_DINT	

Operators (floating conversion to integer word or integer double word):

Type	**Op1**	**Op2**
Words	`%MWi, %QWi.j, %QWEi, %QWMi`	`%MFi, %KFi`
Double words	`%MDi`	`%MFi, %KFi`

Example:

- Floating conversion to integer word: 5978.6 to 5979
- Floating conversion to integer double word: -1235978.6 to -1235979

NOTE: If during a real to integer (or real to integer double word) conversion the floating value is outside the limits of the word (or double word), bit `%S18` is set to 1.

Structure

Example of integer/ floating conversion instruction:

Rung	Instruction
0	`LD 1` `[%MF0:=INT_TO_REAL(%MW10)]`
1	`LD %I0.8` `[%MD2:=REAL_TO_DINT(%MF9)]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Precision of Rounding

Standard IEEE 754 defines 4 rounding modes for floating operations.

The mode employed by the instructions above is the "rounded to the nearest" mode:

"if the nearest representable values are at an equal distance from the theoretical result, the value given will be the value whose low significance bit is equal to 0".

That is to say, the value will be rounded either up or down, but to the even number.

For example:

- Rounding of the value 10.5 to 10.
- Rounding of the value 11.5 to 12.

ASCII

Aim of This Section

This section describes the advanced instructions of ASCII.

ROUND Instructions

Introduction

The *ROUND* instruction rounds a floating point representation that is stored in an ASCII string.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the *ROUND* instruction, use the syntax: `Op1 := ROUND( Op2,Op3 ).`

For example:
`[%MW0:7:=ROUND(%MW8,4)]`

Parameters

This table describes the ROUND function parameters:

Parameters	Description
Op1	`%MW` in which result is stored
Op2	`%MW` containing the floating point to be rounded
Op3	Number of significant digits required in rounding Integer from 1 to 8

Rules of Use

The *ROUND* instruction rules are as follows:

- The operand is rounded down.
- The end character of the operand string is used as an end character for the result string.
- The end character can be any ASCII character that is not in the interval [0 - 9] ([16#30 - 16#39]), except for:
  - dot '.' (16#2E),
  - minus '-' (16#2D),
  - plus '+' (16#2B),
  - Exp 'e' or 'E' (16#65 or 16#45).
- The result and operand should not be longer than 13 bytes: Maximum size of an ASCII string is 13 bytes.
- The scientific notation is not authorized.

Special Cases

The software verifies the syntax. The following examples would result in syntax errors:

Incorrect syntax	Correct syntax
`%MW10:= ROUND(%MW1,4)` missing ":7" in result	`%MW10:7 := ROUND(%MW1,4)`
`%MW10:13:= ROUND(%MW1,4)` %MW10:n where n ≠ 7 is incorrect	`%MW10:7 := ROUND(%MW1,4)`

Application Example

This table shows examples of *ROUND* instruction:

Example	Result
`ROUND("987654321", 5)`	"987650000"
`ROUND("-11.1", 8)`	"-11.1"
`ROUND("NAN")`	"NAN"

ASCII to Integer Conversion Instructions

Introduction

The ASCII to Integer conversion instructions convert an ASCII string into an Integer value.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the ASCII to Integer conversion instructions, use this syntax: `Op1 := ASCII_TO_INT( Op2 )`

For example:
`[%MW0:=ASCII_TO_INT(%MW8)]`

The instruction reads up to four word objects from Op2, converts them to integer format, and stores the result in Op1.

Parameters

This table describes the ASCII to Integer conversion function parameters:

Parameters	Description
Op1	`%MW` in which result is stored
Op2	`%MW` or `%KW`

Rules of Use

The ASCII to Integer instructions rules are as follows:

- Op2 must be between -32768 to 32767.
- The function reads the most significant byte first.
- Leading spaces are ignored.
- An ASCII character that is not in the range [0 - 9] ([16#30 - 16#39]) is considered to be an end character, except for a minus sign '-' (16#2D) when it is placed as the first character.
- In case of overflow (>32767 or <-32768), the system bit %S18 (arithmetic overflow or detected error) is set to 1 and the value 32767 or -32768 is returned.
- If the first character of the operand is a "separator" character, the value 0 is returned and the bit `%S18` is set to 1.

  NOTE: The separator characters are '+' and '-', the letter 'e' or 'E', or '.' (the decimal separator).
- Scientific notation is not valid.

Application Example

Consider that the following ASCII data has been stored in %MW10 to %MW13:

Parameter	Hexadecimal Value	ASCII Representation
%MW10	16#3932	9, 2
%MW11	16#3133	1, 3
%MW12	16#2038	' ', 8
%MW13	16#387A	8, 'z'

This table shows examples of ASCII to Integer conversion:

Example	Result
%MW20 := ASCII_TO_INT(%MW10)	%MW20 = 29318
%MW20 := ASCII_TO_INT(%MW12)	%MW20 = 8
%MW20 := ASCII_TO_INT(%MW13)	%MW20 = 0 and %S18 is set to 1

Integer to ASCII Conversion Instructions

Introduction

The Integer to ASCII conversion instructions convert an Integer into an ASCII string value.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the Integer to ASCII conversion instructions, use this syntax: `Op1 := INT_TO_ASCII( Op2 )`

For example:

```
[%MW0:4:=INT_TO_ASCII(%MW8)]
```

Parameters

This table describes the Integer to ASCII conversion function parameters:

Parameters	Description
Op1	%MW in which result is stored
Op2	%MW, %KW, %SW, %IW, %QW or any WORD (Immediate values are not accepted)

Rules of Use

The Integer to ASCII conversion rules are as follows:

- Op2 must be between -32768 to 32767.
- The function writes the most significant byte first.
- The end character selected in the Application Behavior window (see EcoStruxure Machine Expert - Basic, Operating Guide) is added. The default value is ‘CR’ (carriage return, ASCII 13).
- The function automatically determines how many `%MW` variables should be filled with ASCII values (from 1 to 4).

Syntax Errors

The software verifies the syntax. The following examples would result in syntax errors:

Incorrect syntax	Correct syntax
`%MW10 := INT_TO_ASCII(%MW1)` missing ":4" in result	`%MW10:4 := INT_TO_ASCII(%MW1)`
`%MW10:n := INT_TO_ASCII(%MW1)` `%MW10`:n where n ≠ 4 is incorrect	`%MW10:4 := INT_TO_ASCII(%MW1)`

Application Example

For the instruction `MW10:4 := INT_TO_ASCII(%MW1)` using the end character ‘CR’:

If ...	**Then...**	
Integer Value	**Hexadecimal Value**	**ASCII Representation**
`%MW1` = 123	`%MW10` = 16#3231	2, 1
	`%MW11` = 16#0D33	‘CR’, 3
`%MW1` = 45	`%MW10` = 16#3534	5, 4
	`%MW11` = 16#000D	‘CR’
`%MW1` = 7	`%MW10` = 16#0D37	‘CR’, 7
`%MW1` = -12369	`%MW10` = 16#312D	1, ‘-’
	`%MW11` = 16#3332	3, 2
	`%MW10` = 16#3936	9, 6
	`%MW11` = 16#000D	‘CR’

ASCII to Float Conversion Instructions

Introduction

The ASCII to Float conversion instructions convert an ASCII string into a floating point value.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the ASCII to Float conversion instructions, use this syntax: `Op1 := ASCII_TO_FLOAT( Op2 ).`

For example:
`[%MF0:=ASCII_TO_FLOAT(%MW8)]`

Parameters

This table describes the ASCII to Float conversion function parameters:

Parameters	Description
Op1	`%MF`
Op2	`%MW` or `%KW`

Rules of Use

ASCII to Float conversion rules are as follows:

- The function reads the most significant byte first.
- Any ASCII character that is not in the interval [0 - 9] ([16#30 - 16#39]) is considered to be an "end" character, except for:
  - dot '.' (16#2E),
  - minus '-' (16#2D),
  - plus '+' (16#2B),
  - Exp 'e' or 'E' (16#65 or 16#45).
- ASCII string format can be scientific notation (i.e. "-2.34567e+13") or decimal notation (that is, 9826.3457)
- In case of overflow (calculation result is >3.402824E+38 or <-3.402824E+38):
  - The system bit `%S18` (arithmetic overflow or detected error) is set to 1,
  - `%SW17`:X3 is set to 1,
  - Value +/- 1.#INF (+ or - infinite value) is returned.
- If the calculation result is between -1.175494E-38 and 1.175494E-38, then the result is rounded to 0.0.
- If the operand is not a number:
  - Value NaN is returned,
  - The bit `%SW17`:X0 is set to 1.

Application Example

Consider that the following ASCII data has been stored in `%MW10` to `%MW14`:

Parameter	Hexadecimal Value	ASCII Representation
`%MW10`	16#382D	8, '-'
`%MW11`	16#322E	2, '.'
`%MW12`	16#3536	5, 6
`%MW13`	16#2B65	'+', 'e'
`%MW14`	16#2032	' ', 2

This table shows examples of ASCII to Float conversion:

Example	Result
%MF20 := ASCII_TO_FLOAT(%MW10)	%MF20 = -826.5
%MF20 := ASCII_TO_FLOAT(%MW11)	%MF20 = 26.5
%MF20 := ASCII_TO_FLOAT(%MW12)	%MF20 = 6500.0
%MF20 := ASCII_TO_FLOAT(%MW13)	%MF20 = NaN
%MF20 := ASCII_TO_FLOAT(%MW14)	%MF20 = 2.0

Float to ASCII Conversion Instructions

Introduction

The Float to ASCII conversion instructions convert a floating point value into an ASCII string value.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the Float to ASCII conversion instructions, use this syntax: `Op1 := FLOAT_TO_ASCII(Op2).`

For example:
`[%MW0:7:=FLOAT_TO_ASCII(%MF8)]`

Parameters

This table describes the Float to ASCII conversion function parameters:

Parameter	Description
Op1	%MW
Op2	%MF or %KF

Rules of Use

The Float to ASCII conversion rules are as follows:

- The function writes the most significant byte first.
- The representation is made using conventional scientific notation.
- "Infinite" or "Not a number" results return the string "NAN".
- The end character selected in the Application Behavior window (see EcoStruxure Machine Expert - Basic, Operating Guide) is added. The default value is ‘CR’ (carriage return, ASCII 13).
- The function automatically determines how many %MW variables should be filled with ASCII values.
- Conversion precision is 6 figures.

Syntax Errors

The software verifies the syntax. The following examples would result in syntax errors:

Incorrect Syntax	Correct Syntax
`%MW10 := FLOAT_TO_ASCII(%MF1)` missing ":7" in result	`%MW10:7 := FLOAT_TO_ASCII(%MF1)`
`%MW10:n := FLOAT_TO_ASCII(%MF1)` `%MW10:n` where `n` ≠ 7 is incorrect	`%MW10:7 := FLOAT_TO_ASCII(%MF1)`

Application Example

For the instruction `%MW10:7 := FLOAT_TO_ASCII(%MF1)`:

Number to Convert	Result
1234567800	1.234568e+09
0.000000921	9.210000e-07
9.87654321	9.876543e+00
1234	1.234000e+03

ASCII to Double Word Conversion Instructions

Introduction

The ASCII to double word conversion instruction converts an ASCII string to a double word value.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the ASCII to double word conversion instruction, use this syntax: `Op1 := ASCII_TO_DINT( Op2 )`

For example:
`[%MD4 := ASCII_TO_DINT(%MW5)]`

Parameters

This table describes the ASCII to double word conversion instruction parameters:

Parameters	Description
Op1	`%MDx`
Op2	`%MWy` or `%KWy`

NOTE: It is not necessary to define Op1 and Op2 in an animation table.

Rules of Use

The ASCII to Integer instructions rules are as follows:

- Op2 must be between –2147483648 to 2147483647.
- The function reads the most significant byte first.
- Leading spaces are ignored.
- An ASCII character that is not in the range [0 - 9] ([16#30 - 16#39]) is considered to be an end character, except for a minus sign '-' (16#2D) when it is placed as the first character.
- In case of overflow (> 2147483647 or < –2147483648), the system bit `%S18` (arithmetic overflow or detected error) is set to 1 and the value 2147483647 or –2147483648 is returned.
- If the first character of the operand is a "separator" character, the value 0 is returned and the bit `%S18` is set to 1.

  NOTE: The separator characters are '+' and '-', the letter 'e' or 'E', or '.' (the decimal separator).
- Scientific notation is not valid.

Application Example

Consider that the following ASCII data has been stored in `%MW11` to `%MW13`:

Parameter	Hexadecimal Value	ASCII Representation
`%MW8`	16#3431	4, 1
`%MW9`	16#3532	5, 2
`%MW10`	16#3239	2, 9
`%MW11`	16#3133	1, 3
`%MW12`	16#2038	‘ ‘, 8
`%MW13`	16#387A	8, ‘z‘

This table shows examples of ASCII to Double word conversion:

Example	Result
`%MD10 := ASCII_TO_DINT(%MW8)`	`%MD10` = 142592318
`%MD10 := ASCII_TO_DINT(%MW12)`	`%MD10` = 8
`%MD10 := ASCII_TO_DINT(%MW13)`	`%MD10` = 0 and `%S18` is set to 1

Double Word to ASCII Conversion Instructions

Introduction

The double word to ASCII conversion instruction converts a double word value to an ASCII string value.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the double word to ASCII conversion instruction, use this syntax: `Op1 := DINT_TO_ASCII( Op2 )`

For example:
`[%MW4:6 := DINT_TO_ASCII(%MD5)]`

Parameters

This table describes the double word to ASCII conversion function parameters:

Parameters	Description
Op1	`%MWx:6`
Op2	`%MD` or `%KD`

NOTE: It is not necessary to define Op1 and Op2 in an animation table.

Rules of Use

The Integer to ASCII conversion rules are as follows:

- Op2 must be between –2147483648 to 2147483647.
- The function writes the most significant byte first.
- The end character selected in the Application Behavior window (see EcoStruxure Machine Expert - Basic, Operating Guide) is added. The default value is ‘CR’ (carriage return, ASCII 13).
- The function automatically determines how many `%MW` variables should be filled with ASCII values (from 1 to 6).

Syntax Errors

The software verifies the syntax. The following examples would result in syntax errors:

Incorrect Syntax	Correct Syntax
`%MW2 := DINT_TO_ASCII (%MD1)` missing “:6” in result	`%MW2:6 := DINT_TO_ASCII (%MD1)`
`%MW2:n := DINT_TO_ASCII (%KD7)` `%MW2`:n where n ≠ 6 is incorrect	`%MW2:6 := DINT_TO_ASCII (%KD7)`

Application Example

For the instruction `%MW0:6 := DINT_TO_ASCII(%MD10)` using the line end character 'CR':

If...	Then...	
Integer Value	**Hexadecimal Value**	**ASCII Representation**
`%MD10` = 1236589	`%MW0` = 16#3231	2, 1
	`%MW1` = 16#3633	6, 3
	`%MW2` = 16#3835	8, 5
	`%MW3` = 16#0D37	'CR', 9
`%MD10` = 45	`%MW0` = 16#3534	5, 4
	`%MW1` = 16#000D	'CR'
`%MD10` = -1236945	`%MW0` = 16#3145	1, '-'
	`%MW1` = 16#3332	3, 2
	`%MW2` = 16#3936	9, 6
	`%MW3` = 16#3534	5, 4
	`%MW4` = 16#000D	'CR'

Stack Operators

Aim of This Section

This section describes the advanced instructions of stack.

Stack Instructions (MPS, MRD, MPP)

Introduction

The stack instructions process routing to coils. The *MPS*, *MRD*, and *MPP* instructions use a temporary storage area called the stack which can store up to 32 Boolean expressions.

NOTE: These instructions cannot be used within an expression between parentheses.

Syntax

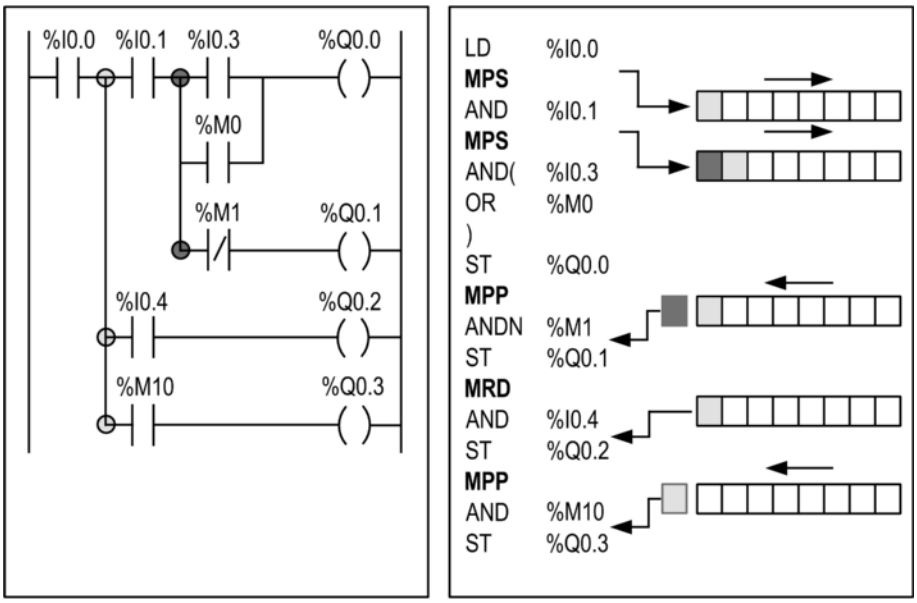
This table describes the 3 stack instructions:

Instruction	Description	Function
MPS	Memory Push onto stack	Stores the result of the last logical instruction (contents of the accumulator) onto the top of the stack (a push) and shifts the other values to the bottom of the stack.
MRD	Memory Read from stack	Reads the top of the stack into the accumulator.
MPP	Memory Pop from stack	Copies the value at the top of the stack into the accumulator (a pop) and shifts the other values towards the top of the stack.

NOTE: For each *MPS* (push) instruction, a matching *MPP* (pop) instruction must appear within the same rung.

Operation

This diagram displays how stack instructions operate:



Application Example

Example of using stack instructions:

Rung	Instruction
0	LD %I0.0 AND %M1 MPS AND %I0.1 ST %Q0.0 MRD AND %I0.2 ST %Q0.1 MRD AND %I0.3 ST %Q0.2 MPP AND %I0.4 ST %Q0.3

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Instructions on Object Tables

Aim of This Section

This section describes instructions to manage Object Tables of:

- Double words
- Floating point objects

Word, Double Word, and Floating Point Tables Assignment

Introduction

Assignment operations can be performed on the following object tables:

- Immediate whole value to word table (see rung 0 of structure example, page 84) or double word table
- Word to word table (see rung 1 of structure example, page 84)
- Word table to word table (see rung 2 of structure example, page 84)

  Table length (L) should be the same for both tables.
- Double word to double word table
- Double word table to double word table

  Table length (L) should be the same for both tables.
- Immediate floating point value to floating point table
- Floating point to floating point table
- Floating point table to floating point table

  Table length (L) should be the same for both tables.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for word, double word, and floating point table assignments:

Operator	Syntax
:=	[Op1 := Op2] Op1 takes the value of Op2

This table gives details of operands:

Type	Op1	Op2
Word table	`%MWi:L`, `%SWi:L`, `%QWi.j:L`, `%QWEi:L`, `%QWMi:L`, `%QWNi.j.k:L`	`%MWi:L`, `%SWi:L`, immediate whole value, `%MWi`, `%KWi`, `%IW`, `%QW`, `%SWi`, `%BLK.x`, `%QWi.j:L`, `%QWEi:L`, `%QWMi:L`, `%QWNi.j.k:L`, `%IWi.j:L`, `%IWEi:L`, `%IWMi:L`, `%IWNi.j.k:L`
Double word tables	`%MDi:L`	Immediate whole value, `%MDi`, `%KDi`,`%MDi:L`, `%KDi:L`
Floating word tables	`%MFi:L`	Immediate floating point value, `%MFi`, `%KFi`, `%MFi:L`, `%KFi:L`
L Length of the table (maximum 255).		

NOTE: The abbreviation `%BLK.x` (for example, `R3.I`) is used to describe any function block word.

Structure

Examples of word table assignments:

Rung	Instruction
0	`LD 1` `[%MW0:10:=100]`
1	`LD %I0.0` `[%MW0:10:=%MW11]`
2	`LDR %I0.3` `[%MW10:20:=%KW20:20]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Table Summing Functions

Introduction

The *SUM_ARR* function adds together all the elements of an object table:

- If the table is made up of double words, the result is given in the form of a double word,
- If the table is made up of floating words, the result is given in the form of a floating word.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of table summing instruction:

```
Res:=SUM_ARR(Tab)
```

Parameters of table summing instruction:

Type	Result (Res)	Table (Tab)
Double word tables	%MDi	%MDi:L,%KDi:L
Floating word tables	%MFi	%MFi:L,%KFi:L
L Length of the table (maximum 255).		

NOTE: When the result is not within the valid double word format range according to the table operand, the system bit %S18 is set to 1.

Structure

Example of summing function:

Rung	Instruction
0	LD %I0.2 [%MD5:=SUM_ARR(%MD3:1)]
1	LD 1 [%MD5:=SUM_ARR(%KD5:2)]
2	LD 1 [%MF2:=SUM_ARR(%MF8:5)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Application Example

```
%MD4:=SUM_ARR(%MD30:4)
```

Where %MD30=10, %MD32=20, %MD34=30, %MD36=40

So %MD4:=10+20+30+40

Table Comparison Functions

Introduction

The *EQUAL_ARR* function carries out a comparison of 2 tables, element by element.

If a difference is shown, the rank of the first dissimilar elements is returned in the form of a word, otherwise the returned value is equal to -1.

The comparison is carried out on the entire table.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of table comparison instruction:

```
Res:=EQUAL_ARR(Tab1,Tab2)
```

Parameters of table comparison instructions:

Type	Result (Res)	Tables (Tab1 and Tab2)
Double word tables	%MWi	%MDi:L,%KDi:L
Floating word tables	%MWi	%MFi:L,%KFi:L
L Length of the table (maximum 255).		

NOTE: The tables must be of the same length and type.

Structure

Example of table comparison function:

Rung	Instruction
0	LD %I0.2 [%MW5:=EQUAL_ARR(%MD20:7,%KD0:7)]
1	LD 1 [%MW0:=EQUAL_ARR(%MD20:7,%KD0:7)]
2	LD 1 [%MF2:=SUM_ARR(%MF8:5)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Application Example

```
%MW5:=EQUAL_ARR(%MD30:4,%KD0:4)
```

Comparison of 2 tables:

Rank	Word Table	Constant Word Tables	Difference
0	%MD30=10	%KD0=10	=
1	%MD32=20	%KD2=20	=
2	%MD34=30	%KD4=60	Different
3	%MD36=40	%KD6=40	=

The value of the word %MW5 is 2 (first different rank).

Table Search Functions

Introduction

There are 3 search functions:

- *FIND_EQR*: searches for the position in a double or floating word table of the first element which is equal to a given value
- *FIND_GTR*: searches for the position in a double or floating word table of the first element which is greater than a given value
- *FIND_LTR*: searches for the position in a double or floating word table of the first element which is less than a given value

The result of these instructions is equal to the rank of the first element which is found or at -1 if the search is unsuccessful.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of table search instructions:

Function	Syntax
FIND_EQR	Res:=Function(Tab,Val)
FIND_GTR	
FIND_LTR	

Parameters of floating word and double word table search instructions:

Type	Result (Res)	Table (Tab)	Value (Val)
Floating word tables	%MWi	%MFi:L,%KFi:L	%MFi,%KFi
Double word tables	%MWi	%MDi:L,%KDi:L	%MDi,%KDi
L Length of the table (maximum 255).			

Structure

Example of table search function:

Rung	Instruction
0	LD %I0.2 [%MW5:=FIND_EQR(%MD20:7,%KD0)]
1	LD %I0.3 [%MW6:=FIND_GTR(%MD20:7,%KD0)]
2	LD 1 [%MW7:=FIND_LTR(%MF40:5,%KF4)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Application Example

```
%MW5:=FIND_EQR(%MD30:4,%KD0)
```

Search for the position of the first double word = `%KD0=30` in the table:

Rank	Word Table	Result
0	`%MD30=10`	-
1	`%MD32=20`	-
2	`%MD34=30`	Value (Val), rank
3	`%MD36=40`	-

Table Search Functions for Maximum and Minimum Values

Introduction

There are 2 search functions:

- *MAX_ARR*: search for the maximum value in a double word and floating word table
- *MIN_ARR*: search for the minimum value in a double word and floating word table

The result of these instructions is equal to the maximum value (or minimum) found in the table.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of table search instructions for maximum and minimum values:

Function	Syntax
MAX_ARR	Result:=Function(Tab)
MIN_ARR	

Parameters of table search instructions for maximum and minimum values:

Type	Result (Res)	Table (Tab)
Double word tables	`%MDi`	`%MDn:L,%KDn:L`
Floating word tables	`%MFi`	`%MFn:L,%KFn:L`
i is the object instance identifier for the memory variable. *n* is the memory index of the table that indicates the base address for the search. *L* is the number of positions to be considered on a search including the base address index (maximum value of L is 255).		

NOTE: L counts only the addresses that are not overlapped during the search. For more information, refer to Possibility of Overlap Between Objects, page 30.

Structure

Example of table search function:

Rung	Instruction
0	LD %I0.2 [%MD0:=MIN_ARR(%MD20:7)]
1	LD 1 [%MF8:=MIN_ARR(%MF40:5)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Number of Occurrences of a Value in a Table

Introduction

This function *OCCUR_ARR* searches in a double word or floating word table for the number of elements equal to a given value.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of table search instructions for maximum and minimum values:

Function	Syntax
OCCUR_ARR	Res:=Function(Tab,Val)

Parameters of table search instructions for maximum and minimum values:

Type	Result (Res)	Table (Tab)	Value (Val)
Double word tables	%MWi	%MDi:L,%KDi:L	%MDi,%KDi
Floating word tables	%MWi	%MFi:L,%KFi:L	%MFi,%KFi
L Length of the table (maximum 255).			

Structure

Example of number of occurrences:

Rung	Instruction
0	LD %I0.3 [%MW5:=OCCUR_ARR(%MF20:7,%KF0)]
1	LD %I0.2 [%MW5:=OCCUR_ARR(%MD20:7,%MD1)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

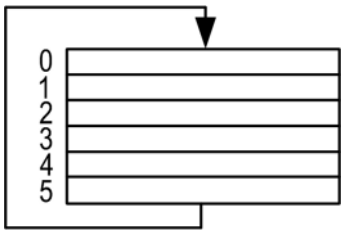
Table Rotate Shift Functions

Introduction

There are 2 shift functions:

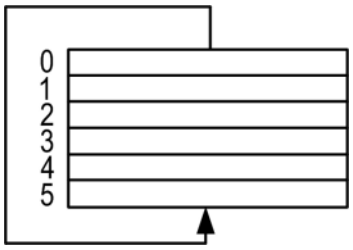
- *ROL_ARR*: performs a rotate shift of n positions from top to bottom of the elements in a floating word table

Illustration of the *ROL_ARR* functions



- *ROR_ARR*: performs a rotate shift of n positions from bottom to top of the elements in a floating word table

Illustration of the *ROR_ARR* functions



Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of rotate shift instructions in floating word or double word tables *ROL_ARR* and *ROR_ARR*:

Function	Syntax
ROL_ARR	Function(n,Tab)
ROR_ARR	

Parameters of rotate shift instructions for floating word tables: **ROL_ARR** and **ROR_ARR**:

Type	Number of positions (n)	Table (Tab)
Floating word tables	`%MWi, %KWi, immediate value`	`%MFi:L`
Double word tables	`%MWi, %KWi, immediate value`	`%MDi:L`
L Length of the table (maximum 255).		

NOTE: If the value of n is negative or null, no shift is performed.

Structure

Example of table rotate shift function:

Rung	Instruction
0	`LD %I0.2` `[ROL_ARR(%KW0,%MD20:7)]`
1	`LD %I0.3` `[ROR_ARR(2,%MD20:7)]`
2	`LD %I0.4` `[ROR_ARR(2,%MF40:5)]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Table Sort Functions

Introduction

The sort function *SORT_ARR* performs sorts in ascending or descending order of the elements of a double word or floating word table and stores the result in the same table.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of table sort functions:

Function	Syntax
SORT_ARR	Function(direction,Tab)

The "direction" parameter gives the order of the sort:

- Direction > 0: the sort is done in ascending order.
- Direction < 0: the sort is done in descending order.
- Direction = 0: no sort is performed

The result (sorted table) is returned in the Tab parameter (table to sort).

Parameters of table sort functions:

Type	Sort Direction	Table (Tab)
Double word tables	`%MWi, %KWi, immediate value`	`%MDi:L`
Floating word tables	`%MWi, %KWi, immediate value`	`%MFi:L`
L Length of the table (maximum 255).		

Structure

Example of table sort function:

Rung	Instruction
0	LD %I0.1 [SORT_ARR(%MW20,%MF0:6)]
1	LD %I0.2 [SORT_ARR(%MW20,%MF0:6)]
2	LD %I0.3 [SORT_ARR(0,%MF40:8)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Floating Point Table Interpolation (LKUP) Functions

Introduction

The *LKUP* function is used to interpolate a set of X versus Y floating point data for a given X value.

Review of Linear Interpolation

The LKUP function makes use of the linear interpolation rule, as defined in this equation:

$$Y = Y_i + \left[\frac{(Y_{i+1} - Y_i)}{(X_{i+1} - X_i)} . (X - X_i)\right]$$ *(Equation 1)*

for $X_i \leq X \leq X_{i+1}$, where $i = 1 \ldots (m-1)$;

assuming X_i values are ranked in ascending order:
$X_1 \leq X_2 \leq \ldots X \ldots \leq X_{m-1} \leq X_m$.

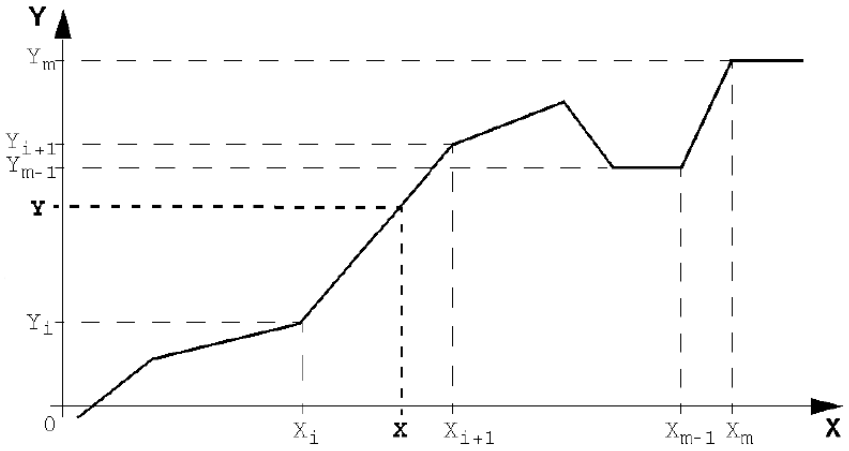
NOTE: If any two consecutive Xi values are equal (X_i=X_{i+1}=X), equation (1) yields an invalid exception. In this case, to cope with this exception the following algorithm is used in place of equation (1):

$$Y = \left[\frac{(Y_{i+1} - Y_i)}{2}\right]$$ *(Equation 2)*

for $X_i = X_{i+1} = X$, where $i = 1 \ldots (m-1)$.

Graphical Representation

This graph illustrates the linear interpolation rule described above:



Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

The *LKUP* function uses three operands, two of which are function attributes, as described in this table:

Syntax	**Op1** **Output Variable**	**Op2** **User-defined (X) value**	**Op3** **User-defined (X_i,Y_i) Variable Array**
[Op1 := LKUP(Op2,Op3)]	`%MWi`	`%MF0`	Integer value, `%MWi`, or `%KWi`

Definition of Op1

Op1 is the memory word that contains the output variable of the interpolation function.

Depending on the value of Op1, you can know whether the interpolation was successful or not, and what prevented success, as outlined in this table:

Op1 (`%MWi`)	**Description**
0	Successful interpolation
1	Interpolation error detected: Incorrect array, $X_m < X_{m-1}$
2	Interpolation error detected: Op2 out of range, $X < X_1$
4	Interpolation error detected: Op2 out of range, $X > X_m$
8	Invalid size of data array: • Op3 is set as odd number, or • Op3 < 6.

NOTE: Op1 does not contain the computed interpolation value (Y). For a given (X) value, the result of the interpolation (Y) is contained in `%MF2` of the Op3 array, page 94.

Definition of Op2

Op2 is the floating point variable (%MF0 of the Op3 floating point array) that contains the user-defined (X) value for which to compute the interpolated (Y) value.

Valid range for Op2: $X_1 \leq Op2 \leq X_m$.

Definition of Op3

Op3 sets the size (Op3 / 2) of the floating-point array where the (X_i,Y_i) data pairs are stored.

X_i and Y_i data are stored in floating point objects with even indexes; starting at %MF4 (note that %MF0 and %MF2 floating point objects are reserved for the user set-point X and the interpolated value Y, respectively).

Given an array of (m) data pairs (X_i,Y_i), the upper index (u) of the floating point array (%MFu) is set by using these relationships:

- $Op3 = 2 \cdot m$ *(Equation 3)*
- $u = 2 \cdot (Op3 - 1)$ *(Equation 4)*

The floating point array Op3 (%MFi) has a structure similar to that of this example (where Op3=8):

(X)		(X_1)		(X_2)		(X_3)	
%MF0		%MF4		%MF8		%MF12	
	%MF2		%MF6		%MF10		%MF14
	(Y)		(Y_1)		(Y_2)		(Y_3)
							(Op3=8)

NOTE: As a result of the above array of floating-point structure, Op3 must meet both of the following requirements; or otherwise this will cause an error in the *LKUP* function:

- Op3 is an even number, and
- Op3 ≥ 6 (for there must be at least two data points to allow linear interpolation).

Structure

Interpolation operations are performed as follows:

Rung	Instruction
0	LD %I0.2 [%MW20:=LKUP(%MF0,%KW1)]
1	LD %I0.3 [%MW22:=LKUP(%MF0,10)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Application Example

Use of a *LKUP* interpolation function:

```
[%MW20:=LKUP(%MF0,10)]
```

In this example:

- %MW20 is Op1 (the output variable).
- %MF0 is the user-defined (X) value which corresponding (Y) value must be computed by linear interpolation.
- %MF2 stores the computed value (Y) resulting from the linear interpolation.
- 10 is Op3 (as given by *equation 3* above). It sets the size of the floating point array. The highest ranking item %MFu, where u=18 is given by *equation 4,* above.

There are four pairs of data points stored in Op3 array [%MF4,...%MF18]:

- %MF4 contains X_1,%MF6 contains Y_1.
- %MF8 contains X_2,%MF10 contains Y_2.
- %MF12 contains X_3,%MF14 contains Y_3.
- %MF16 contains X_4,%MF18 contains Y_4.

MEAN Functions of the Values of a Floating Point Table

Introduction

The *MEAN* function is used to calculate the mean average from a given number of values in a floating point table.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of the floating point table means calculation function:

Function	Syntax
MEAN	Result=Function(Op1)

Parameters of the calculation function for a given number L (maximum 255) of values from a floating point table:

Op1	Result (Res)
%MFi:L, %KFi:L	%MFi

Structure

Example of mean function:

Rung	Instruction
0	LD %I3.2 [%MF0:=MEAN(%MF10:5)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

SWAP Function of the Values of a Word Table

Introduction

The *SWAP* function is used to swap the low and high bytes of the values in a word table and store the result in a word table.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the SWAP function, use the syntax:

```
Op1 := SWAP(Op2)
```

Where:

Operand	Type
Op1	%MWx:L, %QWx.y:L, %QWEx:L, %QWMx:L, %QWNx.y.z:L
Op2	%MWx:L, %KWx:L, %QWx.y:L, %IWx.y:L, %QWEx:L, %IWEx:L, %QWMx:L, %IWMx:L, %QWNx.y.z:L, %IWNx.y.z:L
L Length of the table (maximum 255).	

Structure

Example of SWAP function:

Rung	Instruction
0	%MW0:3 := "1234" [%MW0:3 := SWAP(%MW0:3)]

For this example, the result is:

%MW0="12" and %MW1="34"

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Instructions on I/O Objects

Aim of This Section

This section describes the instructions on I/O objects.

Read Immediate Digital Embedded Input (READ_IMM_IN)

Introduction

The *READ_IMM_IN* instruction reads an embedded digital input (input integrated in the logic controller) during the execution of a task and immediately updates the input image. This therefore avoids having to wait for the next task cycle to update the input image.

NOTE: This instruction is only valid for embedded digital inputs.

NOTE: When using this instruction, evaluate the relative performance (turn on and turn off delays) of regular inputs and fast inputs. Refer to Digital Inputs in the programming guide for your logic controller.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the *READ_IMM_IN* instruction, use this syntax:

```
Op1 := READ_IMM_IN(Op2)
```

Where:

Operand	Type	Description
Op1	%MWi	Stores the function return code (see the table below).
Op2	Immediate value (integer) %MWi %KWi	Defines the input index (%I0.x).
i Object instance identifier for the memory variable.		

Function Return Code

This table describes the function return codes:

Code	Description
0	No error detected.
1	Input declared is greater than maximum input allowed.
2	Input declared is forced.

Example

```
%MW0 := READ_IMM_IN(2)
```

Upon execution of this operation block the current value of the input *%I0.2* is read and the input image is immediately updated. The function return code is stored in the *%MW0* memory word.

Structure

Example of *READ_IMM_IN* instruction:

Rung	Instruction
0	LD %M0 [%MW0:=READ_IMM_IN(%MW5)]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Write Immediate Digital Embedded Output (WRITE_IMM_OUT)

Introduction

The *WRITE_IMM_OUT* instruction physically writes to an embedded digital output (output integrated into the logic controller). The value is immediately read from the output image. This therefore avoids having to wait for the next task cycle to write to the embedded output.

NOTE: This function is only valid for embedded digital outputs.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the *WRITE_IMM_OUT* instruction, use this syntax:

```
Op1 := WRITE_IMM_OUT(Op2)
```

Where:

Operand	Type	Description
Op1	`%MWi`	Stores the function return code (see the table below).

Op2	Immediate value (integer)
	`%MWi`
	`%KWi`
		Defines the output index (`%Q0.x`).

i is the object instance identifier for the memory variable.		

Function Return Code

This table describes the function return codes:

Code	Description
0	No error detected.
3	Output declared is greater than maximum output allowed.
4	Output declared is forced.
5	Output declared is used as dedicated hardware output.
6	Output declared is used as alarm output.

Example

`%MW0 := WRITE_IMM_OUT(%MW5)` (with `%MW5` = 2)

At execution of this operation block the output image `%Q0.2` is written physically on the embedded digital output. The function return code is stored in the *%MW0* memory word.

Structure

Example of *WRITE_IMM_OUT* instruction:

Rung	Instruction
0	`LD %M0` `[%MW0:= WRITE_IMM_OUT(%MW4)]`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Read Immediate Function Block Parameter (READ_IMM)

Introduction

The *READ_IMM* instruction reads a function block parameter during the execution of a task and updates the input image during the same cycle.

This function is available only for certain function block parameters. The *READ_IMM* instruction reads directly from the *HSC.V* and *HSC.P* registers in High Speed Counter (*%HSC*) function blocks. For more details, refer to High Speed Counter (%HSC) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the *READ_IMM* instruction, use this syntax:

```
READ_IMM(Op1)
```

Where:

Operand	Type	Description
Op1	*%HSCx.P*, *%HSCx.PD*, *%HSCx.V*, *%HSCx.VD*	This instruction reads a function block parameter given in Op1 and updates the value in the I/O image and the corresponding register.
x is the object instance identifier for the function block.		

Example

The following code is an example of using the *READ_IMM* instruction:

Rung	Instruction
0	`LD %M0` `[READ_IMM(%HSC0.P)]`

NOTE: Refer to reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

NOTE: The application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 3.3** to use the READ_IMM instruction.

Write Immediate Function Block Parameter (WRITE_IMM)

Introduction

The *WRITE_IMM* instruction writes a function block parameter during the execution of a task and updates the output image during the same cycle.

This function is available only for certain function block parameters. The *WRITE_IMM* instruction writes directly to the *HSC.V* and *HSC.P* registers in High Speed Counter (*%HSC*) function blocks. For more details, refer to High Speed Counter (%HSC) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions, page 15 in Ladder Diagram rungs using an **Operation Block** graphical element.

For the *WRITE_IMM* instruction, use this syntax:

```
WRITE_IMM(Op1)
```

Where:

Operand	Type	Description
Op1	*%HSCx.P*, *%HSCx.PD*, *%HSCx.V*, *%HSCx.VD*	This instruction writes a function block parameter given in Op1 and updates the value in the I/O image.
x is the object instance identifier for the function block.		

Example

The following code is an example of using the *WRITE_IMM* instruction:

Rung	Instruction
0	`LD %M1` `[WRITE_IMM(%HSC0.V)]`

NOTE: Refer to reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

NOTE: The application must be configured with a functional level of at least **Level 3.3** to use the WRITE_IMM instruction.

I/O Objects

Fast Counter (%FC)

Overview

Refer to Fast Counter (%FC) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

High Speed Counter (%HSC)

Overview

Refer to High Speed Counter (%HSC) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

Pulse (%PLS)

Overview

Refer to Pulse (%PLS) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

Pulse Width Modulation (%PWM)

Overview

Refer to Pulse Width Modulation (%PWM) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

Network Objects

Network Objects

Presentation

Network objects are used to communicate via EtherNet/IP, Modbus TCP, or Modbus Serial IOScanner.

There are two types of network object for EtherNet/IP communication:

- *%QWE*: Input Assembly
- *%IWE*: Output Assembly

There are two types of network object for Modbus TCP communication:

- *%QWM*: Input registers
- *%IWM*: Output registers

The following types of network object are used for the Modbus Serial IOScanner:

- *%IN*: Digital inputs (IOScanner)
- *%QN*: Digital outputs (IOScanner)
- *%IWN*: Input registers (IOScanner)
- *%QWN*: Output registers (IOScanner)
- *%IWNS*: IOScanner Network Diagnostic Codes

  NOTE: References to input and output are from the point of view of the EtherNet/IP master or Modbus TCP client.

For more information on how to configure network objects, refer to the programming guide for your logic controller.

Software Objects

Using Function Blocks

Function Block Programming Principles

Overview

A function block is a reusable object that accepts one or more input values and returns one or more output values.

You can insert up to five function blocks into each Ladder Diagram rung.

The function block parameters are not available if:

- your controller does not support the function block,
- the function block is not configured.

Ladder Diagram Programs

To use a function block in a Ladder Diagram rung:

Step	Action
1	Insert, page 104 the function block into a rung.
2	Wire the inputs and outputs as necessary.
3	Configure, page 106 the function block by specifying values for its parameters.

Instruction List Programs

To add a function block to an Instruction List program, you can use one of the following methods:

- Function block instructions (for example, `BLK %TM2`): This reversible method of programming enables operations to be performed on the block in a single place in the program.
- Specific instructions (for example, `CU %Ci`). This non-reversible method enables operations to be performed on function block inputs in several places in the program. For example:

Line	Instruction
1000	`CU %C1`
1074	`CD %C1`
1209	`R %C1`

Use the instructions *BLK*, *OUT_BLK*, and *END_BLK* for reversible programming of function blocks:

- *BLK:* Indicates the beginning of the block.
- *OUT_BLK:* Is used to wire directly the block outputs.
- *END_BLK:* Indicates the end of the block.

  NOTE: Test and input instructions on the relevant block can only be placed between the *BLK* and *OUT_BLK* instructions (or between *BLK* and *END_BLK* when *OUT_BLK* is not programmed).

  NOTE: *BLK* is an application keyword. A user-declared variable or parameter should not start/end with this keyword as it can be detected as a program validation error.

Example with Output Wiring

This example shows a Counter function block in a program with wired outputs:

Rung	Instruction
0	BLK %C8 LDF %I0.1 R LD %I0.1 AND %M0 CU OUT_BLK LD D AND %M1 ST %Q0.0 END_BLK

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Example Without Output Wiring

This example shows reversible programming of a *Counter* function block without wired outputs:

Rung	Instruction
0	BLK %C8 LDF %I0.1 R LD %I0.2 AND %M0 CU END_BLK
1	LD %C8.D AND %M1 ST %Q0.4

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Adding a Function Block

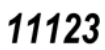
Inserting a Function Block Into a Ladder Diagram Program

Follow this procedure:

Step	Action
1	Create a new Ladder Diagram rung in the programming workspace of EcoStruxure Machine Expert - Basic. Refer to the EcoStruxure Machine Expert - Basic Operating Guide for details.
2	Click the **Function** button on the graphical toolbar at the top of the programming workspace. **Result**: A list of all available function block objects is displayed (see the table below).
3	Select the function block.
4	Move the function block to the required position in the rung; then click to insert it. You can insert up to five function blocks in series or parallel.

Available Function Block Objects

This table presents the available function block objects:

Function Block Object	Description
	Timer
	LIFO/FIFO Register
	Shift Bit Register
	Step Counter
	Counter
	Fast Counter
	High Speed Counter
	Drum
	RTC (Real-Time Clock)
	Pulse
	Pulse Width Modulation
	Message
	Data Logging
	Pulse Train Output **NOTE:** For a complete list of PTO objects, refer to PTO Function Blocks in the Modicon M221 Logic Controller Advanced Functions, Library Guide.
DRV	Drive objects **NOTE:** For a complete list of drive objects, refer to Drive (% DRV) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.
	Communication function blocks **NOTE:** For a complete list of communication function blocks, refer to Communication Objects, page 171.
UDFB	User-defined function block

Configuring a Function Block

Configuring a Function Block in a Ladder Diagram Program

Follow this procedure:

Step	Action
1	Optionally, click **Address** within the function block. A default address appears in the text box, for example "%TM0" for a *Timer* function block. To change the default address, delete the final digit of the address (the instance identifier). A list of all available addresses appears. Select the address to use to identify this instance of the function block. The properties of the function block appear in the center of the function block object and in the **Properties** table in the bottom half of the programming workspace. At any other time, double-click anywhere within the function block to display the properties.
2	Optionally, click **Comment** within the function block, type a short description of the function block. For example, **Pulse Timer**.
3	Optionally, click **Symbol** within the function block and begin typing the name of the symbol to associate with this function block. A list of all existing symbols with names beginning with the character or characters you type appears; click the symbol to use. To create a new symbol for this function block, type the name of the symbol to create, and select the object to associate with the symbol. See the EcoStruxure Machine Expert - Basic Operating Guide (see EcoStruxure Machine Expert - Basic, Operating Guide) for details on using symbols.
4	Click the function block. **Result:** The **Configuration** tooltip appears.
5	Configure the available parameters of each function block, as described in the “Parameters” topic of individual function block descriptions. You can modify the values of the objects in online mode. Refer to Online Modifications (see EcoStruxure Machine Expert - Basic, Operating Guide).

NOTE: You can also display the **Properties** table by double-clicking on the function block in a rung.

Timer (%TM)

Using Timer Function Blocks

This section provides descriptions and programming guidelines for using *Timer* function blocks.

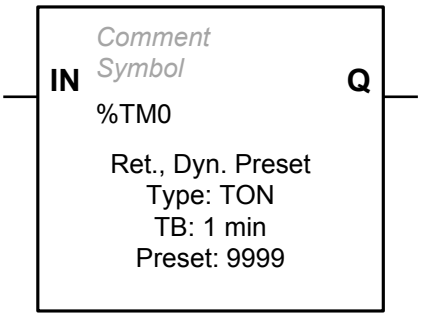
Description

Introduction

A *Timer* function block  is used to specify a period of time before doing something, for example, triggering an event.

Illustration

This illustration is the *Timer* function block.



Inputs

The *Timer* function block has the following input:

Label	Description	Value
IN	Input address (or instruction)	Starts the *Timer* when a rising edge (TON or TP types) or falling edge (TOF type) is detected.

Outputs

The *Timer* function block has the following output:

Label	Description	Value
Q	Output address (`%TMi.Q`)	Associated bit *%TMi.Q* is set to 1 (depending on the *Timer* type) when the *Timer* expires.

Configuration

Parameters

To configure parameters, follow the Configuring a Function Block procedure, page 106 and read the description of Memory Allocation Modes in the EcoStruxure Machine Expert - Basic Operating Guide.

The *Timer* function block has the following parameters:

Parameter	Description	Value	Editable in Online Mode
Used	Address used	If selected, this address is in use in the program.	No
Address	*Timer* object address (*%TM*i)	A program can contain only a limited number of *Timer* objects. Refer to the Programming Guide of the related platform for the maximum number of timers.	No
Symbol	Symbol	The symbol associated with this object. Refer to the EcoStruxure Machine Expert - Basic Operating Guide, Defining and Using Symbols (see EcoStruxure Machine Expert - Basic, Operating Guide) for details.	No
Type	*Timer* type	One of the following: • TON, page 109: Timer on-Delay (default) • TOF, page 112: Timer off-Delay • TP, page 116: Pulse timer (monostable)	Yes[1]
Retentive	TRUE/FALSE	When the **Retentive** checkbox is not selected (default), the value is reset when a falling edge of the **IN** parameter is detected. Counting restarts from 0. When the **Retentive** checkbox is selected, the timer retains its value when a falling edge of the **IN** parameter is detected before the **Preset** value is reached. Counting restarts from this value. **NOTE:** The application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 3.3** to use the **Retentive** parameter.	Yes[1]
Dynamic Preset	TRUE/FALSE	When the **Dynamic Preset** checkbox is not selected (default) and the preset value is modified (by programming instruction or using EcoStruxure Machine Expert - Basic), the modified preset value only takes effect on the next activation of the timer. When the **Dynamic Preset** checkbox is selected and the preset value is modified (by programming instruction or using EcoStruxure Machine Expert - Basic), the modified preset value takes effect immediately. When the **Dynamic Preset** checkbox is selected and a new preset value is received while the timer is running: • If the Present value (`TMi.V`) is greater than or equal to the new preset value, the timer behaves as if the preset value has been reached. • If the Present value (`TMi.V`) is less than the new preset value, the timer continues to run until it reaches the new preset value. **NOTE:** Your application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 12.0** to use the **Dynamic Preset** parameter.	Yes[1]
Time Base	Time base	The base time unit of the timer. The smaller the timer base unit, the greater the acuity of the timer: • **1 ms** (supported in *%TM0*...*%TM5*) • **10 ms** • **100 ms** • **1 sec** • **1 min** (default)	Yes[1]
Preset	Preset value	0...9999. Default value is 9999. Timer Period = **Preset** x Time **Base** Timer Delay = **Preset** x Time **Base** This configured preset value can be read, tested, and modified using the associated object `%TMi.P`.	Yes[1]
Comment	Comment	A comment can be associated with this object.	No
[1] The timer is immediately reset to 0 following any change to the parameter value when in online mode.			

Objects

The *Timer* function block has the following objects:

Object	Description	Value
%TMi.P	Preset value	See description in Parameters table above.
%TMi.V	Present value	Word that increments from 0 to the preset value %TMi.P when the timer is running. The value can be read and tested, but not written to, by the program. Its value can be modified in an animation table.
%TMi.Q	*Timer* output	See description in Outputs table above.

TON: On-Delay Timer

Introduction

The TON (*On-Delay Timer*) type of timer is used to control on-delay actions. This delay is programmable using the software.

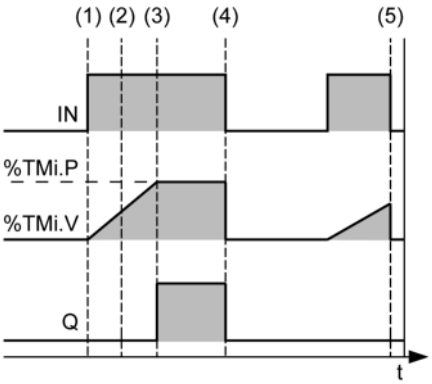
Special Cases

The following table contains a list of special cases for programming the Timer function block.

Special case	Description
Effect of a cold restart (%S0=1)	Sets the %TMi.V value to 0. Sets output %TMi.Q to 0. The %TMi.P preset value is reset to the value defined during configuration.
Effect of a warm restart (%S1=1)	Has no effect on the %TMi.V timer value and %TMi.P preset value of the timer. The timer value does not change during a power outage.
Effect of a controller stop	Stopping the controller does not freeze the value.
Effect of a program jump	Jumping over the timer block does not freeze the timer. The timer continues to increment until it reaches the preset value (%TMi.P). At that point, the Done bit (%TMi.Q) assigned to output Q of the timer block changes state. However, the associated output wired directly to the block output is not activated and not scanned by the controller.
Testing by bit %TMi.Q (Done bit)	Test bit %TMi.Q only once in the program.
Effect of modifying the preset %TMi.P	The effect of modifying the preset value by using an instruction, or by adjusting the value with an animation table, depends on the configuration of the **Dynamic Preset** parameter. Refer to Dynamic Preset parameter for details.

Timing Diagram

This diagram illustrates the operation of the TON type *Timer*.



(1) The *Timer* starts on the rising edge of the IN input.

(2) The value `%TMi.V` increases from 0 to `%TMi.P` in increments of 1 unit for each pulse of the time base parameter *TB*.

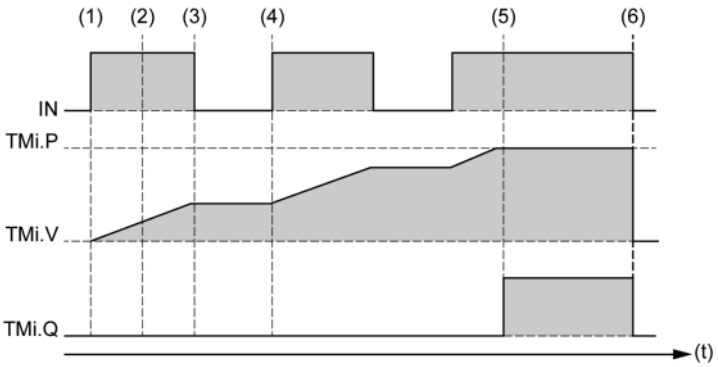
(3) The `%TMi.Q` output bit is set to 1 when the value has reached the preset value `%TMi.P`.

(4) The `%TMi.Q` output bit remains at 1 while the *IN* input is at 1.

(5) When a falling edge is detected at the *IN* input, the *Timer* is stopped, even if the *Timer* has not reached `%TMi.P`, the `%TMi.V` value is set to 0.

Timing Diagram with Retentive Checkbox Selected

This diagram illustrates the operation of the TON type *Timer* when the **Retentive** checkbox is selected.



(1) The *Timer* starts on the rising edge of the IN input.

(2) The value `%TMi.V` increases from 0 to `%TMi.P` in increments of 1 unit for each pulse of the time base parameter *TB*.

(3) On the falling edge of the *IN* input, the *Timer* is stopped and remains unchanged awaiting the next rising edge of the *IN* input.

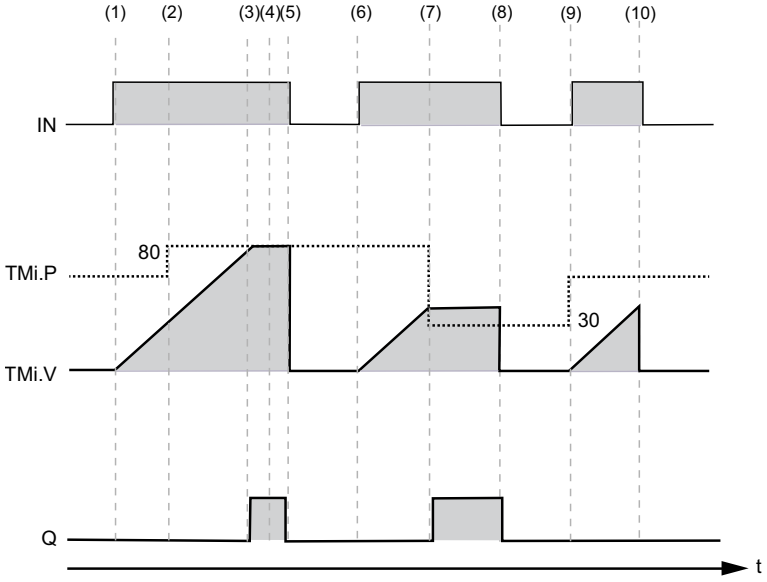
(4) On the rising edge of the *IN* input, the *Timer* starts again from the value it stopped at.

(5) The `%TMi.Q` output bit is set to 1 when the value reaches the preset value `%TMi.P`.

(6) When a falling edge is detected at the *IN* input, if the *Timer* has reached the preset value `%TMi.P`, the `%TMi.V` value is set to 0.

Timing Diagram with Dynamic Preset Checkbox Selected

This diagram illustrates the operation of the TON type *Timer* when the **Dynamic Preset** checkbox is selected.



(1) The *Timer* starts on the rising edge of the *IN* input.

(2) The value of `%TMi.P` is changed to 80. Since `%TMi.V` is less than the new `%TMi.P` value, the `%TMi.Q` output bit value is unchanged and `%TMi.V` continues to increment with each pulse of the *Time Base* parameter *TB*.

(3) The `%TMi.Q` output bit is set to 1 when `%TMi.V` reaches the new preset value `%TMi.P`.

(4) The `%TMI.Q` output bit remains at 1 while the *IN* input is at 1.

(5) On a falling edge of the *IN* input, the *Timer* is stopped. `%TMi.V` is set to 0.

(6) The *Timer* starts on the rising edge of the *IN* input.

(7) The value of `%TMi.P` is changed to 30. The new `%TMi.P` value will be taken into account at the next *Time Base* parameter value update. Since `%TMi.V` is greater than the new `%TMi.P` value, the `%TMi.Q` output bit is set to 1.

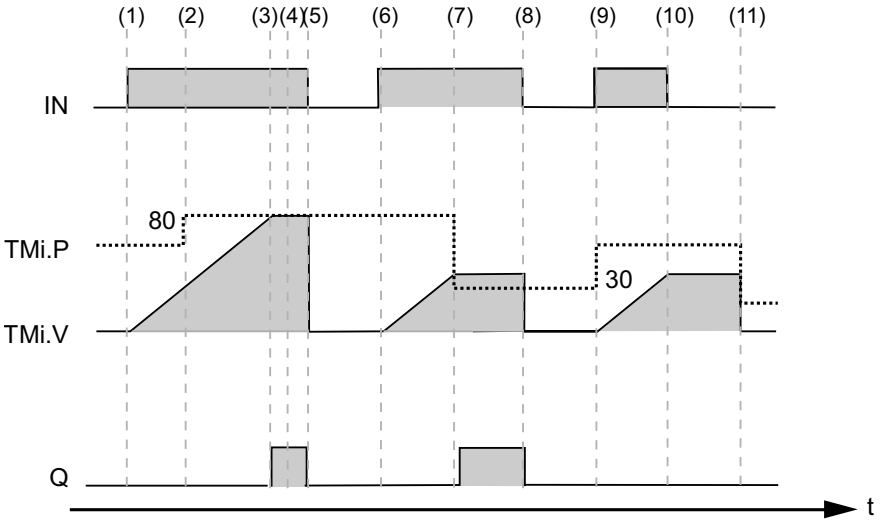
(8) When a falling edge is detected at the *IN* input, the *Timer* is stopped. `%TMi.V` is set to 0.

(9) The *Timer* starts on the rising edge of the *IN* input.

(10) When a falling edge is detected at the *IN* input, the *Timer* is stopped, even if the value of `%TMi.P` has not been reached. The `%TMi.V` value is set to 0.

Timing Diagram with Dynamic Preset and Retentive Checkboxes Selected

This diagram illustrates the operation of the TON type *Timer* when both the **Dynamic Preset** and **Retentive** checkboxes are selected.



(1) The *Timer* starts on the rising edge of the *IN* input.

(2) The value of `%TMi.P` is changed to 80. Since `%TMi.V` is less than the new `%TMi.P` value, the `%TMi.Q` value is unchanged and `%TMi.V` continues to increment with each pulse of the *Time Base* parameter TB.

(3) The `%TMi.Q` output bit is set to 1 when `%TMi.V` reaches the new preset value `%TMi.P`.

(4) The `%TMI.Q` output bit remains at 1 while the *IN* input remains at 1.

(5) On a falling edge of the *IN* input, the *Timer* is stopped. `%TMi.V` is set to 0.

(6) The *Timer* starts on the rising edge of the *IN* input.

(7) The value of `%TMi.P` is changed to 30. The new `%TMi.P` value will be taken into account at the next *Time Base* parameter value update. Since `%TMi.V` is greater than the new `%TMi.P` value, the `%TMi.Q` output bit is set to 1.

(8) When a falling edge is detected at the *IN* input, the *Timer* is stopped. `%TMi.V` is set to 0.

(9) The *Timer* starts on the rising edge of the *IN* input.

(10) When a falling edge is detected at the *IN* input, the *Timer* is stopped. `%TMi.V` remains unchanged as the *Retentive* parameter is enabled.

(11) If `%TMi.P` is set to a value less than `%TMi.V` when the timer is disabled (that is, the *IN* input is set to 0), the `%TMi.V` value is set to 0.

TOF: Off-Delay Timer

Introduction

Use the TOF (*Off-Delay Timer*) type of *Timer* to control off-delay actions. This delay is programmable using the software.

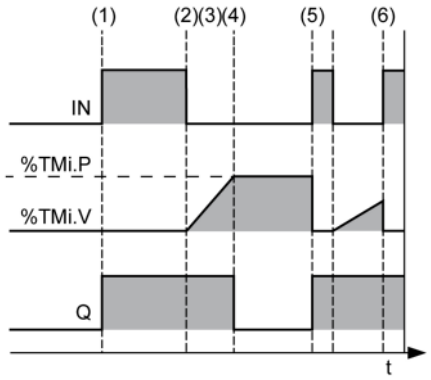
Special Cases

The following table contains a list of special cases for programming the Timer function block.

Special case	Description
Effect of a cold restart (%S0=1)	Sets the %TMi.V value to 0. Sets output %TMi.Q to 0. The %TMi.P preset value is reset to the value defined during configuration.
Effect of a warm restart (%S1=1)	Has no effect on the %TMi.V timer value and %TMi.P preset value of the timer. The timer value does not change during a power outage.
Effect of a controller stop	Stopping the controller does not freeze the value.
Effect of a program jump	Jumping over the timer block does not freeze the timer. The timer continues to increment until it reaches the preset value (%TMi.P). At that point, the Done bit (%TMi.Q) assigned to output Q of the timer block changes state. However, the associated output wired directly to the block output is not activated and not scanned by the controller.
Testing by bit %TMi.Q (Done bit)	Test bit %TMi.Q only once in the program.
Effect of modifying the preset %TMi.P	The effect of modifying the preset value by using an instruction, or by adjusting the value with an animation table, depends on the configuration of the **Dynamic Preset** parameter. Refer to Dynamic Preset parameter for details.

Timing Diagram

This diagram illustrates the operation of the TOF type *Timer*.



(1) At a rising edge of input *IN*, %TMi.Q is set to 1.

(2) The *Timer* starts on the falling edge of input *IN*.

(3) The value %TMi.V increases to the preset value %TMi.P in increments of 1 unit for each pulse of the time base parameter TB.

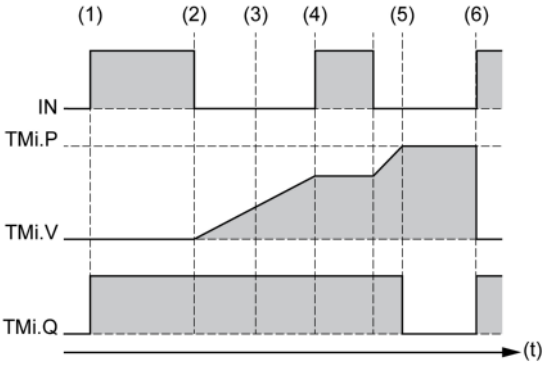
(4) The %TMi.Q output bit is reset to 0 when the value reaches the preset value %TMi.P.

(5) At a rising edge of input *IN*, %TMi.V is set to 0.

(6) At a rising edge of input *IN*, %TMi.V is set to 0, even if the preset value is not reached.

Timing Diagram with Retentive Checkbox Selected

This diagram illustrates the operation of the TOF type *Timer* when the **Retentive** checkbox is selected.



(1) At a rising edge of input *IN*, `%TMi.Q` is set to 1.

(2) The *Timer* starts on the falling edge of input *IN*.

(3) The value `%TMi.V` increases to the preset value `%TMi.P` in increments of 1 unit for each pulse of the time base parameter *TB*.

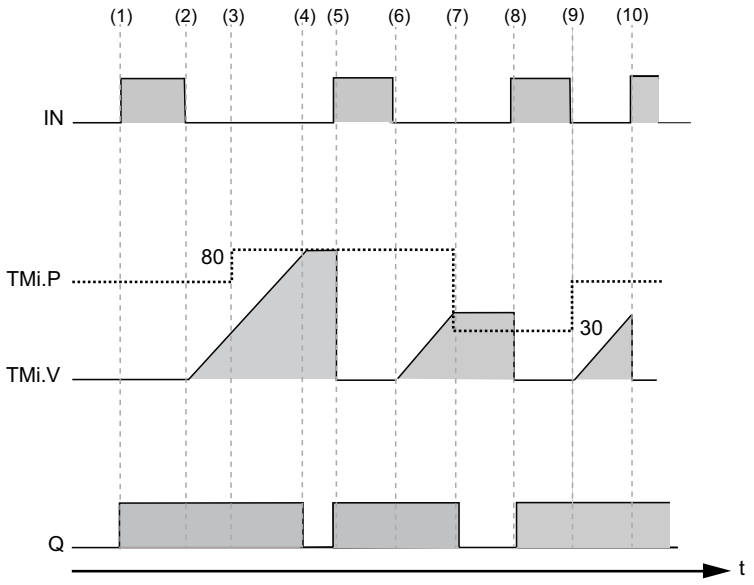
(4) On a rising edge of the *IN* input, the Timer is stopped and remains unchanged awaiting the next falling edge of the *IN* input.

(5) The `%TMi.Q` output bit is reset to 0 when the value reaches the preset value `%TMi.P`

(6) At a rising edge of input *IN*, `%TMi.V` is set to 0 and `%TMi.Q` is set to 1.

Timing Diagram with Dynamic Preset Checkbox Selected

This diagram illustrates the operation of the TOF type *Timer* when the **Dynamic Preset** checkbox is selected.



(1) At a rising edge of the *IN* input, `%TMi.Q` is set to 1.

(2) The *Timer* starts on the falling edge of input *IN*.

(3) The value of `%TMi.P` is changed to 80. Since `%TMi.V` is less than the new `%TMi.P` value, the `%TMi.Q` output bit value is unchanged and `%TMi.V` continues to increment with each pulse of the *Time Base* parameter `TB`.

(4) The `%TMi.Q` output bit is reset to 0 when `%TMi.V` reaches the preset value `%TMi.P`.

(5) On a rising edge of the *IN* input, `%TMi.V` is set to 0

(6) The *Timer* starts on the falling edge of the *IN* input.

(7) The value of `%TMi.P` is changed to 30. The new `%TMi.P` value will be taken into account at the next *Time Base* parameter value update. The new `%TMi.P` value will be taken into account at the next *Time Base* parameter value update. Since `%TMi.V` is greater than the new `%TMi.P` value, the `%TMi.Q` output bit is reset to 0.

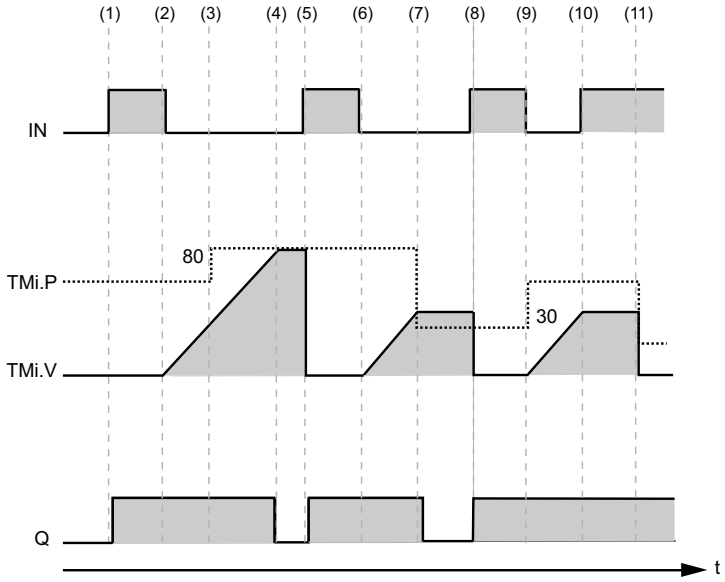
(8) At a rising edge of input *IN*, `%TMi.V` is set to 0.

(9) The *Timer* starts on the falling edge of the *IN* input.

(10) At a rising edge of the *IN* input, `%TMi.V` is set to 0, even if the preset value `%TMi.P` has not been reached.

Timing Diagram with Dynamic Preset and Retentive Checkboxes Selected

This diagram illustrates the operation of the TOF type *Timer* when both the **Dynamic Preset** and **Retentive** checkboxes are selected.



(1) On a rising edge of the *IN* input, `%TMi.Q` is set to 1.

(2) The Timer starts on a falling edge of the *IN* input.

(3) The value of `%TMi.P` is changed to 80. Since `%TMi.V` is less than the new `%TMi.P` value, the `%TMi.Q` output bit value is unchanged and `%TMi.V` continues to increment with each pulse of the *Time Base* parameter `TB`.

(4) The `%TMi.Q` output bit is reset to 0 when the `%TMi.V` value reaches the new preset value `%TMi.P`.

(5) On detecting a rising edge of the *IN* input, `%TMi.V` is set to 0.

(6) The *Timer* starts on the falling edge of the *IN* input.

(7) The value of `%TMi.P` is changed to 30. The new `%TMi.P` value will be taken into account at the next *Time Base* parameter value update. Since `%TMi.V` is greater than the new `%TMi.P` value, the`%TMi.Q` output bit is reset to 0.

(8) On detecting a rising edge of the *IN* input, `%TMi.V` is set to 0.

(9) The *Timer* starts on the falling edge of the *IN* input.

(10) At a rising edge of the *IN* input, the *Timer* is stopped. `%TMi.V` remains unchanged as the *Retentive* parameter is enabled.

(11) If `%TMi.P` is set to a value less than `%TMi.V` when the timer is disabled (that is, the *IN* input is set to 1) `%TMi.V` is reset to 0.

TP: Pulse Timer

Introduction

The TP (Pulse Timer) type of *Timer* is used to create pulses of a precise duration. This delay is programmable using the software.

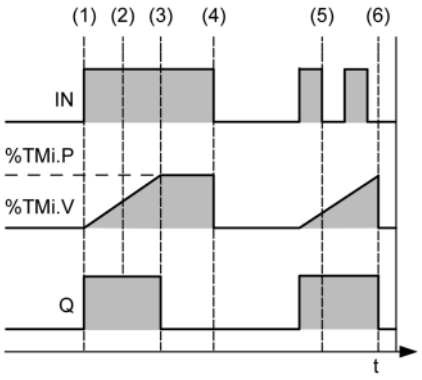
Special Cases

The following table contains a list of special cases for programming the Timer function block.

Special case	Description
Effect of a cold restart (%S0=1)	Sets the %TMi.V value to 0. Sets output %TMi.Q to 0. The %TMi.P preset value is reset to the value defined during configuration.
Effect of a warm restart (%S1=1)	Has no effect on the timer %TMi.V value and %TMi.P preset value of the timer. The timer value does not change during a power outage.
Effect of a controller stop	Stopping the controller does not freeze the value.
Effect of a program jump	Jumping over the timer block does not freeze the timer. The timer continues to increment until it reaches the preset value (%TMi.P). At that point, the Done bit (%TMi.Q) assigned to output Q of the timer block changes state. However, the associated output wired directly to the block output is not activated and not scanned by the controller.
Testing by bit %TMi.Q (Done bit)	Test bit %TMi.Q only once in the program.
Effect of modifying the preset %TMi.P	The effect of modifying the preset value by using an instruction, or by adjusting the value with an animation table, depends on the configuration of the **Dynamic Preset** parameter. Refer to Dynamic Preset parameter for details.

Timing Diagram

This diagram illustrates the operation of the TP type *Timer*.



(1) The *Timer* starts on the rising edge of the *IN* input. The current value %TMi.V is set to 0 if the *Timer* has not already started and %TMi.Q is set to 1 when the *Timer* starts.

(2) The current value %TMi.V of the *Timer* increases from 0 to the preset value %TMi.P in increments of one unit per pulse of the time base parameter *TB*.

(3) The %TMi.Q output bit is set to 0 when the current value has reached the preset value %TMi.P.

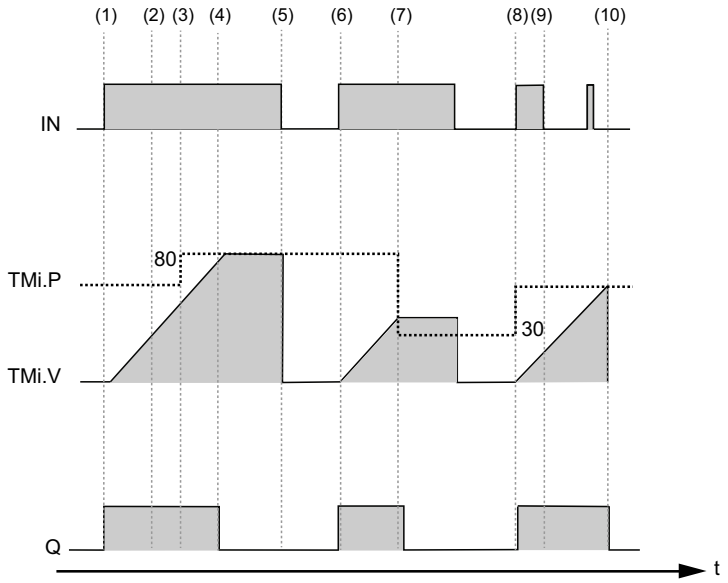
(4) The current value %TMi.V is set to 0 when %TMi.V equals %TMi.P and input *IN* returns to 0.

(5) This *Timer* cannot be reset.

(6) When %TMi.V equals %TMi.P and input *IN* is 0, then %TMi.Q is set to 0.

Timing Diagram with Dynamic Preset Checkbox Selected

This diagram illustrates the operation of the TP type *Timer* when the **Dynamic Preset** checkbox is selected.



(1) The timer starts on a rising edge of the *IN* input. `%TMi.V` is set to 0 if the timer has not already started. The `%TMi.Q` output bit value is set to 1 when the timer starts.

(2) The current value `%TMi.V` of the *Timer* increases from 0 to the preset value `%TMi.P` in increments of one unit per pulse of the *Time Base* parameter `TB`.

(3) The value of `%TMi.P` is changed to 80. Since `%TMi.V` is less than the new `%TMi.P` value, the `%TMi.Q` value is unchanged and `%TMi.V` continues to increment with each pulse of the *Time Base* parameter `TB`.

(4) The `%TMi.Q` output bit is reset to 0 when `%TMi.V` reaches the value of the preset parameter `%TMi.P`.

(5) The current value `%TMi.V` is set to 0 when `%TMi.V` equals `%TMi.P` and the *IN* input is reset to 0.

(6) The timer starts on a rising edge of the *IN* input. The current value `%TMi.V` is set to 0 if the timer has not already started and `%TMi.Q` is set to 1 when the timer starts.

(7) The value of `%TMi.P` is changed to 30. The new `%TMi.P` value will be taken into account at the next *Time Base* parameter value update. Since `%TMi.V` is greater than the new `%TMi.P` value, the `%TMi.Q` output bit is reset to 0.

(8) The *Timer* starts on a rising edge of the *IN* input. The current value `%TMi.V` is set to 0 if the timer has not already started and `%TMi.Q` is set to 1 when the timer starts.

(9) This type of timer cannot be reset.

(10) When `%TMi.V` equals `%TMi.P` and the input *IN* is 0, `%TMi.Q` is set to 0.

Programming Example

Introduction

Timer function blocks have the following operating modes:

- TON (Timer On-Delay), page 109: used to specify a period of time between a specified input being activated and an output sensor being switched on.
- TOF (Timer Off-Delay), page 112: used to specify a period of time between an output associated with a sensor no longer being detected and the corresponding output being switched off.
- TP (Timer - Pulse), page 116: used to create a pulse of a precise duration.

The delays or pulse periods of *Timers* are programmable and can be configured from within the software.

Programming

This example is a *Timer* function block with reversible instructions:

Rung	Reversible Instruction
0	BLK %TM0 LD %M0 IN OUT_BLK LD Q ST %Q0.0 END_BLK
1	LD [%TM0.V<400] ST %Q0.1
2	LD [%TM0.V>=400] ST %Q0.2

This example is the same *Timer* function block with non-reversible instructions:

Rung	Non-Reversible Instruction
0	LD %M0 IN %TM0
1	LD %TM0.Q ST %Q0.0
2	LD [%TM0.V<400] ST %Q0.1
3	LD [%TM0.V>=400] ST %Q0.2

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Counter (%C)

Using Counter Function Blocks

This section provides descriptions and programming guidelines for using *Counter* function blocks.

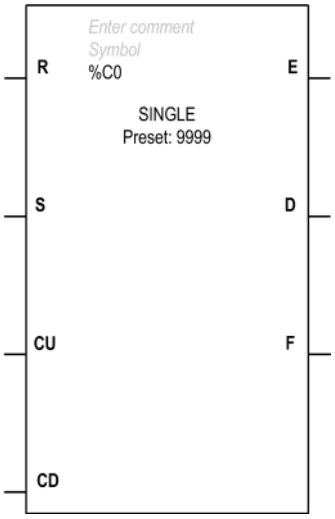
Description

Introduction

The *Counter* function block 123 provides up and down counting of events. These 2 operations can be done concurrently.

Illustration

This illustration presents the *Counter* function block.



Inputs

The *Counter* function block has the following inputs:

Label	Description	Value
R	Reset input (or instruction)	Sets the counter (`%Ci.V`) to 0 when the reset input (**R**) is set to 1.
S	Set input (or instruction)	Sets the counter (`%Ci.V`) to the preset value (`%Ci.P`) when the set input (**S**) is set to 1.
CU	Count up	Increments the counter value (`%Ci.V`) by 1 on a rising edge at count up input (**CU**).
CD	Count down	Decrements the counter value (`%Ci.V`) by 1 on a rising edge at count down input (**CD**).

Outputs

The **Counter** function block has the following outputs:

Label	Description	Value
E	Down count overflow	The associated bit `%Ci.E` (counter empty) is set to 1 when the counter reaches 0 value. In case of following decrement, the counter value passes to 9999.
D	Preset output reached	The associated bit `%Ci.D` (count done) is set to 1 when `%Ci.V = %Ci.P`.
F	Up count overflow	The associated bit `%Ci.F=1` (counter full), when `%Ci.V` changes from 9999 to 0 (set to 1 when `%Ci.V` reaches 0, and reset to 0 if the *Counter* continues to count up).

Configuration

Parameters

To configure parameters, follow the Configuring a Function Block procedure, page 106 and read the description of Memory Allocation Modes in the EcoStruxure Machine Expert - Basic Operating Guide.

The *Counter* function block has the following parameters:

Parameter	Description	Value	Editable in online mode?
Used	Address used	If selected, this address is currently in use in a program.	No
Address	*Counter* object address	A program can contain only a limited number of counter objects. Refer to the *Programming Guide* of your controller for the maximum number of counters.	No
Symbol	Symbol	The symbol associated with this object. Refer to the EcoStruxure Machine Expert - Basic Operating Guide, Defining and Using Symbols (see EcoStruxure Machine Expert - Basic, Operating Guide) for details.	No
Preset	Preset value	Values accepted by preset value [0 − 9999]. Default value is 9999. This configured value can be read, tested, and modified using the associated object %Ci.P.	Yes
Comment	Comment	A comment can be associated with this object.	No

Objects

The *Counter* function block has the following objects:

Object	Description	Value
`%Ci.V`	Current value of the *Counter*	This word is incremented or decremented according to inputs (or instructions) **CU** and **CD** (see Inputs table, page 120). Can be only read. It can be modified in an animation table.
`%Ci.P`	Preset value	See Parameters table, page 121. It can be modified in an animation table.
`%Ci.E`	Empty	See Outputs table, page 120. It can be modified in an animation table.
`%Ci.D`	Done	See Outputs table, page 120. It can be modified in an animation table.
`%Ci.F`	Full	See Outputs table, page 120. It can be modified in an animation table.

Operations

This table describes the main stages of *Counter* function block operations:

Operation	Action	Result
Reset	Input R is set to state 1(or the *R* instruction is activated).	The current value %Ci.V is forced to 0. Outputs %Ci.E, %Ci.D,and %Ci.F are at 0. The reset input has priority.
Set	If input S is set to 1 (or the *S* instruction is activated) and the reset input is at 0 (or the *R* instruction is inactive).	The current value %Ci.V takes the %Ci.P value and the %Ci.D output is set to 1.
Counting	A rising edge appears at the Count up input CU (or instruction CU is activated).	The %Ci.V current value is incremented by one unit.
	The %Ci.V current value is equal to the %Ci.P preset value.	The "preset reached" output bit %Ci.D switches to 1.
	The %Ci.V current value changes from 9999 to 0.	The output bit %Ci.F (up-counting overflow) switches to 1.
	If the *Counter* continues to count up.	The output bit %Ci.F (up-counting overflow) is reset to 0.
Count down	A rising edge appears at the down-counting input CD (or instruction CD is activated).	The current value %Ci.V is decremented by 1 unit.
	The current value %Ci.V changes from 0 to 9999.	The output bit %Ci.E (down-counting overflow) switches to 1.
	If the *Counter* continues to count down.	The output bit %Ci.F (down-counting overflow) is reset to 0.

Special Cases

This table shows a list of special operating/configuration cases for *Counter* function block:

Special Case	Description
Effect of a cold restart (%S0=1) or **INIT**	• The current value %Ci.V is set to 0. • Output bits %Ci.E, %Ci.D, and %Ci.F are set to 0. • The preset value is initialized with the value defined during configuration.
Effect of a warm restart (%S1=1) of a controller stop	Has no effect on the current value of the *Counter* (%Ci.V).
Effect of modifying the preset %Ci.P	Modifying the preset value via an instruction or by adjusting it takes effect when the block is processed by the application (activation of one of the inputs).

NOTE: Effect of **INIT** is the same as %S0=1.

Programming Example

Introduction

The following example is a counter that provides a count of up to 5000 items. Each pulse on input %I0.2 (when memory bit %M0 is set to 1) increments the Counter function block *%C8* up to its final preset value (bit %C8.D=1). The counter is reset by input %I0.1.

Programming

This example is a *Counter* function block with reversible instructions:

Rung	Reversible Instruction

0	`BLK  %C8`
	`LD   %I0.1`
	`R`
	`LD   %I0.2`
	`AND  %M0`
	`CU`
	`END_BLK`

1	`LD   %C8.D` `ST   %Q0.0`

This example is the same *Counter* function block with non-reversible instructions:

Rung	Non-Reversible Instruction
0	`LD  %I0.1` `R   %C8`

1	`LD  %I0.2`
	`AND %M0`
	`CU  %C8`

2	`LD  %C8.D` `ST  %Q0.0`

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

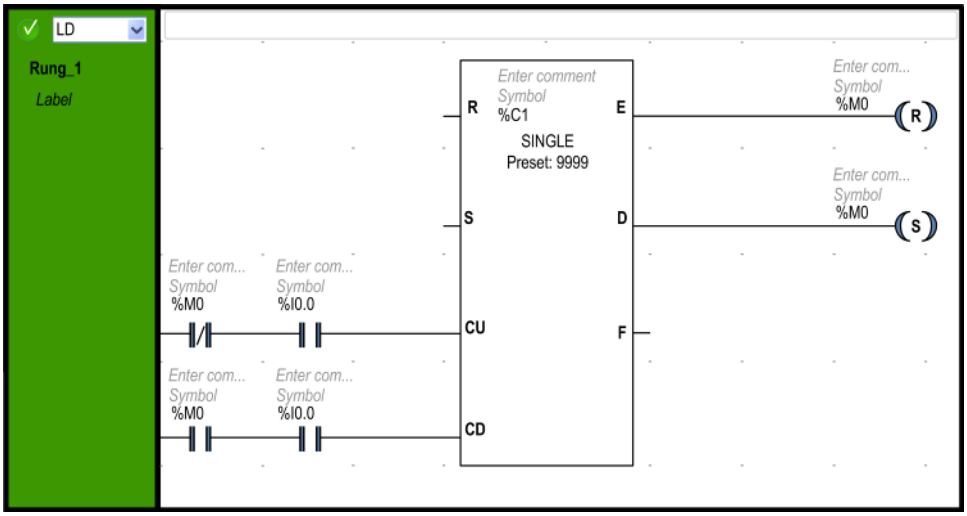
Configuration

The parameters must be entered during configuration:

Preset value (`%Ci.P`): set to 5000 in this example.

Example of an Up/Down Counter

This illustration is an example of a *Counter* function block.



In this example, `%M0` is the increment (`%M0` = False) and the decrement (`%M0` = True) order. The counter counts the Front edge of `%I0.0`. If `%M0` is False, at each Front Edge on `%I0.0`, `%C1.V` is incremented until it reaches the preset `%C1.P` value, and the Done indicator `%C1.D` switches to TRUE. The `%C1.D` output sets `%M0` and switches the instruction into decrement order. Then at each Front Edge on `%I0.0`, `%C1.V` is decremented until it reaches 0. The Empty indicator (`%C1.E`) switches on and resets `%M0` (Increment order).

Message (%MSG) and Exchange (EXCH)

Using Message Function Blocks

This section provides descriptions and programming guidelines for using *Message* function blocks.

Overview

Introduction

A logic controller can be configured to communicate in Modbus protocol or can send and/or receive messages in character mode (ASCII).

EcoStruxure Machine Expert - Basic provides the following functions for these communications:

- **Exchange** (*EXCH*) instruction to transmit/receive messages.
- *Message* function block (`%MSG`) to control the data exchanges.

The logic controller uses the protocol configured for the specified port when processing an **Exchange** instruction. Each communication port can be assigned a different protocol. The communication ports are accessed by appending the port number suffix to the **Exchange** instruction (*EXCH1*, *EXCH2*) or *Message* function block (`%MSG1`, `%MSG2`).

The logic controllers implement Modbus TCP messaging over the Ethernet network by using the *EXCH3* instruction and `%MSG3` function block.

This table shows the **Exchange** instruction and *Message* function block used to access the communication ports of the controller:

Communication Port	Port Number Suffix	Exchange Instruction	Message Function Block
SL1	1	*EXCH1*	`%MSG1`
SL2	2	*EXCH2*	`%MSG2`
Ethernet	3	*EXCH3*	`%MSG3`

Exchange Instruction

The **Exchange** instruction allows a logic controller to send and/or receive information to/from ASCII or Modbus devices. You define a table of words (`%MWi:L`) containing control information and the data to be sent and/or received. Refer to Configuring the transmission table, page 130. A message exchange is performed using the **Exchange** instruction.

Syntax

The following is the format for the **Exchange** instruction:
`[EXCHx %MWi:L]`

Where: x = port number; L = total number of words of the word table.

The logic controller must finish the exchange from the first **Exchange** instruction before a second **Exchange** instruction can be started. The *Message* function block must be used when sending several messages.

ASCII Protocol

ASCII protocol provides the logic controller a character mode protocol to transmit and/or receive data with a device. This protocol is supported using the **Exchange** instruction and controlled using the *Message* function block.

3 types of communications are possible with the ASCII protocol:

- Transmission only
- Transmission/Reception
- Reception only

Modbus Protocol

In case of serial link, the Modbus protocol is a master-slave protocol that allows for one, and only one, master to request responses from slaves, or to act based on the request. On Ethernet support, several Master (client) can exchange with one slave (server). Each slave must have a unique address. The master can address individual slaves, or can initiate a broadcast message to the slaves. Slaves return a message (response) to queries that are addressed to them individually. Responses are not returned to broadcast queries from the master.

Modbus master mode allows the controller to send a Modbus query to a slave, and to wait for the response. The Modbus master mode is only supported via the **Exchange** instruction. Both Modbus ASCII and RTU are supported in Modbus master mode.

Modbus slave mode allows the controller to respond to standard Modbus queries from a Modbus master.

For detailed information about Modbus protocol, refer to the document *Modbus application protocol* which is available at http://www.modbus.org.

Modbus Slave

The Modbus protocol supports 2 Data link layer of the OSI Model formats: ASCII and RTU. Each is defined by the Physical Layer implementation, with ASCII using 7 data bits, and RTU using 8 data bits.

When using Modbus ASCII mode, each byte in the message is sent as 2 ASCII characters. The Modbus ASCII frame begins with a start character (':'), and ends with 2 end characters (CR and LF). The end of frame character defaults to 0x0A (LF). The check value for the Modbus ASCII frame is a two's complement of the frame, excluding the start and end characters.

Modbus RTU mode does not reformat the message prior to transmitting; however, it uses a different checksum calculation mode, specified as a CRC.

The Modbus Data Link Layer has the following limitations:

- Address 1-247
- Bits: 128 bits on request
- Words: 125 words of 16 bits on request

Description

Introduction

The *Message* function block manages data exchanges and has three functions:

- Communications error checking:

  Error checking verifies the size of each **Exchange** table, and verifies the validity of the exchange related to the configuration.

- Coordination of multiple messages:

  To help ensure coordination when sending multiple messages, the *Message* function block provides the information required to determine when a previous message is complete.

- Transmission of priority messages:

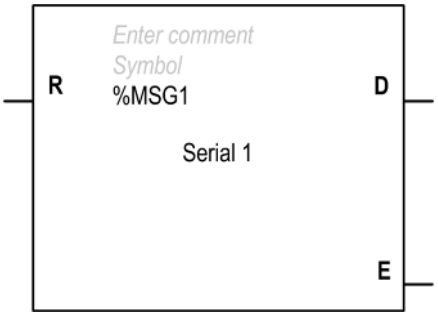
  The *Message* function block allows the on-going message transmission to be stopped in order to allow the immediate sending of an urgent message.

The programming of the *Message* function block is optional.

When errors are detected, codes are written to the system words `%SW63`, `%SW64`, and `%SW65` for the exchange blocks `EXCH1`, `EXCH2` and `EXCH3`, respectively. For more information, refer to the *Programming Guide* of your controller.

Illustration

This illustration presents the *Message* function block:



Inputs

The *Message* function block has the following input:

Label	Description	Value
R	Reset input (% MSGx.R)	Set to 1 to reinitialize communication: • Communication done (%MSGx.D) output is set to 1 • Communication Error Detected (%MSG.E) output is set to 0 • An error is set on any active Communication function block (% READ_VAR, %WRITE_VAR, etc.). • Active TCP connections to other Modicon M221 Logic Controllers are closed. **NOTE:** Only one *Message* function block, *EXCH* instruction, or Communication function block can be active at a time on a communication port during a master task cycle. If you attempt to use several communication function blocks, *MSG*, or *EXCH* instructions concurrently on the same communication port, the function blocks return an error code.Therefore, verify that no active exchange (%MSGx.D is TRUE) is in progress on a communication port before starting a Communication function block, *Message* function block, or *EXCH* instruction. Further, verify that the IOScanner is not also active on the communication port. **NOTE:** The IOScanner does not update the outputs of the *% MSG* function block. Therefore, the *%MSG.D* bit is irrelevant to the function of the IOScanner.

Outputs

The *Message* function block has the following outputs:

Label	Description	Value
D	Communication Done (`%MSGx.D`)	State 1: • End of transmission (if transmission) • End of reception (end character received) • Error • Reset the block State 0: request in progress.
E	Communication Error Detected (`%MSGx.E`)	State 1: • Undefined command • Table incorrectly configured • Incorrect character received (speed, parity, and so on) • Reception table full (not updated) State 0: message length correct, link established. Refer to the table below for the error codes written to the system words when communication error is detected.

Communication Error Codes

This table describes the error codes written to the system words when communication error is detected:

System word	Function	Description
%SW63	EXCH1 block error code	EXCH1 error code: 0 - operation was successful 1 - number of bytes to be transmitted exceeds the limit (> 255) 2 - insufficient transmission table 3 - insufficient word table 4 - receive table overflowed 5 - time-out elapsed 6 - transmission 7 - incorrect command within table 8 - selected port not configured/available 9 - reception error: This error code reflects an incorrect or corrupted reception frame. It can be due to an incorrect configuration in the physical parameters (for example, parity, data bits, baudrate, and so on) or an unreliable physical connection causing signal degradation. 10 - cannot use %KW if receiving 11 - transmission offset larger than transmission table 12 - reception offset larger than reception table 13 - controller stopped EXCH processing
%SW64	EXCH2 block error code	EXCH2 error code: See %SW63.
%SW65	EXCH3 block error code	1-4, 6-13: See %SW63. (Note that error code 5 is invalid and replaced by the Ethernet-specific error codes 109 and 122 described below.) The following are Ethernet-specific error codes: 101 - incorrect IP address 102 - no TCP connection 103 - no socket available (all connection channels are busy) 104 - network is down 105 - network cannot be reached 106 - network dropped connection on reset 107 - connection aborted by peer device 108 - connection reset by peer device 109 - connection time-out elapsed 110 - rejection on connection attempt 111 - host is down 120 - incorrect index (remote device is not indexed in configuration table) 121 - system error (MAC, chip, duplicate IP) 122 - receiving process timed-out after data was sent 123 - Ethernet initialization in progress

Configuration

Detected Error

If an error is detected when using an **Exchange** instruction, bits %MSGx.D and %MSGx.E are set to 1, system words %SW63...%SW65 contain the error code for the corresponding communication port. Refer to the System Words chapter of your logic controller Programming Guide.

Operations

This table describes the main stages of *Message* function block operations:

Operation	Action	Result
Reset	Input R is set to state 1 (or the *R* instruction is activated).	• Any messages that are being transmitted are stopped. • The communication error output is reset to 0. • The Done bit is set to 1. A new message can now be sent.
Communication done	Output D is set to state 1.	The logic controller is ready to send another message. Use of the %MSGx.D bit to help avoid losing messages when multiple messages are sent.
Communication Detected Error	The communication error output is set to 1: • Either because of a communications programming error or a message transmission error. • If the number of bytes defined in the data block associated with the **Exchange** instruction (word 1, least significant byte) is greater than 128 (+80 in hexadecimal by FA). • If an error is detected in sending a Modbus message to a Modbus device. In this case, you should verify the wiring, and that the destination device supports Modbus communication.	

Special Cases

This table contains a list of special cases for the *Message* operation:

Special Case	Description
Effect of a cold restart (%S0=1) or **INIT**	Forces a reinitialization of the communication.
Effect of a warm restart (%S1=1)	Has no effect.
Effect of a controller stop	If a message transmission is in progress, the controller stops its transfer and reinitializes the outputs %MSGx.D and %MSGx.E.

NOTE: Effect of **INIT** is the same as %S0=1.

Limitations

Note the following limitations:

- Port 2 (for ASCII protocol) availability and type (see %SW7) are verified only at power-up or reset
- Port 2 (for Modbus protocol) presence and configuration (RS-485) are verified at power-up or reset
- **Any message processing on port 1 is aborted** when EcoStruxure Machine Expert - Basic is connected
- **Exchange** instructions abort active Modbus slave processing
- Processing of **Exchange** instructions is not retried in the event of a detected error
- Reset input (R) can be used to abort **Exchange** instruction reception processing
- **Exchange** instructions are configured with a time-out in case of Modbus protocol.
- Multiple messages are controlled via %MSGx.D

Configuring the Transmission/Reception Table

The maximum size of the transmitted and/or received frames is:

- 250 bytes for Modbus protocol.
- 255 bytes for ASCII protocol.

The word table associated with the **Exchange** instruction is composed of the control, transmission, and reception tables:

<table>
<tr><th rowspan="2"></th><th colspan="2">Most Significant Byte</th><th colspan="2">Least Significant Byte</th></tr>
<tr><th>Modbus</th><th>ASCII</th><th>Modbus</th><th>ASCII</th></tr>
<tr><td rowspan="2">Control table</td><td colspan="2">Command</td><td colspan="2">Length (transmission/reception)</td></tr>
<tr><td>Rx offset</td><td>Reserved (0)</td><td>Tx offset</td><td>Reserved (0)</td></tr>
<tr><td rowspan="4">Transmission table</td><td colspan="2">Transmitted byte 1</td><td colspan="2">Transmitted byte 2</td></tr>
<tr><td colspan="2" rowspan="2">...</td><td colspan="2">...</td></tr>
<tr><td colspan="2">Transmitted byte n</td></tr>
<tr><td colspan="2">Transmitted byte n+1</td><td colspan="2"></td></tr>
<tr><td rowspan="4">Reception table</td><td colspan="2">Received byte 1</td><td colspan="2">Received byte 2</td></tr>
<tr><td colspan="2" rowspan="2">...</td><td colspan="2">...</td></tr>
<tr><td colspan="2">Received byte p</td></tr>
<tr><td colspan="2">Received byte p+1</td><td colspan="2"></td></tr>
</table>

NOTE: In addition to queries to individual slaves, the Modbus master controller can initiate a broadcast query to all slaves. The **Command** byte in case of a broadcast query must be set to 00, while the slave address must be set to 0.

Control Table for ASCII Protocol

The **Length** byte contains the length of the transmission table in bytes (255 maximum), which is overwritten by the number of characters received at the end of the reception, if reception is requested.

The **Command** byte must contain one of the following:

- 0: Transmission only
- 1: Send/receive
- 2: Reception Only

Control Table for Modbus Protocol

The **Length** byte contains the length of the transmission table in bytes (250 maximum), which is overwritten by the number of characters received at the end of the reception, if reception is requested.

This parameter is the length in bytes of the transmission table. If the **Tx offset** parameter is equal to 0, this parameter will be equal to the length of the transmission frame. If the **Tx offset** parameter is not equal to 0, one byte of the transmission table (indicated by the offset value) will not be transmitted and this parameter is equal to the frame length itself plus 1.

The **Command** byte in case of Modbus RTU request (except for broadcast) must be equal to 1 (**Tx** and **Rx**). For broadcast, it must be 0.

The **Tx offset** byte contains the rank (1 for the first byte, 2 for the second byte, and so on) within the transmission table of the byte to ignore when transmitting the bytes. This is used to handle issues associated with byte/word values within the Modbus protocol. For example, if this byte contains 3, the third byte would be ignored, making the fourth byte in the table the third byte to be transmitted.

The **Rx offset** byte contains the rank (1 for the first byte, 2 for the second byte, and so on) within the reception table to add when transmitting the packet. This is used to handle issues associated with byte/word values within the Modbus protocol. For example, if this byte contains 3, the third byte within the table would be filled with a 0, and the third byte which was received would be entered into the fourth location in the table.

Transmission/Reception Tables for ASCII Protocol

When in transmit-only mode, the control and transmission tables of type %MW are filled prior to executing the **Exchange** (EXCH) instruction. No space is required for the reception of characters in transmit-only mode. Once all bytes are transmitted, `%MSGx.D` is set to 1, and a new **Exchange** (EXCH) instruction can be executed.

When in transmit/receive mode, the control and transmission tables are filled in prior to executing the **Exchange** (EXCH) instruction, and must be of type `%MW`. Space for up to 256 reception bytes is required at the end of the transmission table. Once all bytes are transmitted, the logic controller switches to reception mode and waits to receive any bytes.

When in reception-only mode, the control table is filled in prior to executing the **Exchange** instruction, and must be of type `%MW`. Space for up to 256 reception bytes is required at the end of the control table. The logic controller immediately enters reception mode and waits to receive any bytes.

Reception ends when end of frame bytes used have been received, or the reception table is full. In this case, a detected error code (receive table overflowed) appears in the system words `%SW63` and `%SW64`. If a non-zero timeout is configured, reception ends when the timeout is completed. If a zero timeout value is selected, there is no reception timeout. Therefore, to stop reception, `%MSGx.R` input must be activated.

Transmission/Reception Tables for Modbus Protocol

When using either mode (Modbus ASCII or Modbus RTU), the transmission table is filled with the request prior to executing the **Exchange** (EXCH) instruction. At execution time, the logic controller determines what the data link layer is, and performs the conversions necessary to process the transmission and response.

Start, end, and check characters are not stored in the Transmission/Reception tables.

Once all bytes are transmitted, the logic controller switches to reception mode and waits to receive any bytes.

Reception is completed in one of several ways:

- timeout on a character or frame has been detected,
- end of frame characters received in ASCII mode,
- the reception table is full.

Transmitted byte x entries contain Modbus protocol (RTU encoding) data that is to be transmitted. If the communications port is configured for Modbus ASCII, the correct framing characters are appended to the transmission. The first byte contains the device address (specific or broadcast), the second byte contains the function code, and the rest contain the information associated with that function code.

> **NOTE:** This is a typical application, but does not define all the possibilities. No validation of the data being transmitted will be performed.

Received bytes x entries contain Modbus protocol (RTU encoding) data that is to be received. If the communications port is configured for Modbus ASCII, the correct framing characters are removed from the response. The first byte contains the device address, the second byte contains the function code (or response code), and the rest contain the information associated with that function code.

> **NOTE:** This is a typical application, but does not define all the possibilities. No validation of the data being received is performed, except for checksum verification.

Programming Example

Introduction

The following are examples of programming a *Message* function block.

Programming a Transmission of Several Successive Messages

Execution of the **Exchange** instruction activates a *Message* function block in the application program. The message is transmitted if the *Message* function block is not already active (`%MSGx.D = 1`). If several messages are sent in the same cycle, only the first message is transmitted using the same port.

Example of a transmission of 2 messages in succession on port 1:

Rung	Reversible Instruction	Comment
0	LD %M142 [%MW2:=16#0106] [%MW3:=0] [%MW4:=16#0106] [%MW5:=4] [%MW6:=7]	Write on a slave, at address 1: value 7 on the register 4. [%MW2:=16#0106]: Command code: 01 hex, transmission length: 06 hex [%MW3:=0]: No reception or transmission offset [%MW4:=16#0106]: Slave address: 01 hex, function code: 06 hex (Write Single Register) [%MW5:=4]: Register address [%MW6:=7]: Value to write
1	LD %MSG1.D AND %M0 [EXCH1 %MW2:8] R %M0	%MSG2.D: Detects whether the port is busy or not and thereby manages coordination of multiple messages.
2	LDR %I0.0 AND %MSG1.D [EXCH1 %MW2:8] S %M0	–

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Programming a Reinitialization Exchange

An exchange is canceled by activating the input (or instruction) R. This input initializes communication and resets output %MSGx.E to 0 and output %MSGx.D to 1. It is possible to reinitialize an exchange if an error is detected.

Example of reinitializing an exchange:

Rung	Reversible Instruction	Comment
0	BLK %MSG1 LD %M0 R END_BLK	–

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

ASCII Examples

Application Writing

Example of ASCII application:

Rung	Instruction	Comment
0	LD 1 [%MW10:=16#0104] [%MW11:=16#0000] [%MW12:=16#4F4B]	[%MW10:=16#0104]: Command code: 01 hex, transmission length: 04 hex [%MW11:=16#0000]: 0000: Null [%MW12:=16#4F4B]: Ok **NOTE:** The table has 8 elements.
1	LD 1 AND %MSG2.D [EXCH2 %MW10:8]	
2	LD %MSG2.E ST %Q0.0 END	

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Use EcoStruxure Machine Expert - Basic to create a program with 3 rungs:

- First, initialize the control and transmission tables to use for the **Exchange** instruction. In this example, a command is set up to both send and receive data. The amount of data to send is set to 4 bytes, as defined in the application, followed by the end of frame character defined in the configuration. Start and end characters do not display in an animation table, only data characters. In all cases, those characters are automatically transmitted or verified at reception (by %SW63 and %SW64), when used.

  NOTE: The end characters defined in the configuration are sent automatically in the end of the frame. For example, if you have configured the first end character to 10 and the second end character to 13, 16#0A0D (ASCII codes, 0A = LF and 0D = CR) is sent in the end of the frame.

- Next, verify the status bit associated with %MSG2 and issue the EXCH2 instruction only if the port is ready. For the EXCH2 instruction, a value of 8 words is specified. There are 2 control words (%MW10 and %MW11), 2 words to be used for transmit information (%MW12 and %MW13), and 4 words to receive data (%MW14 through %MW17).
- Finally, the detected error status of the %MSG2 is verified and stored on the first output bit on the local base controller I/O. Additional error handling using %SW64 could also be added to make this more accurate.

Animation Table Initialization

Example of initializing an animation table in online mode:

Address	Value	Format
%MW10	0104	Hexadecimal
%MW11	0000	Hexadecimal
%MW12	4F4B	Hexadecimal
%MW13	0A0D	Hexadecimal
%MW14	AL	ASCII
%MW15	OH	ASCII
%MW16	A	ASCII

To display the possible formats, right-click on the **Values** box in an animation table.

The final step is to download this application to the controller and run it. Initialize an animation table to animate and display the %MW10 through %MW16 words. This information is exchanged with a logic controller and displayed in an animation table.

Modbus Standard Requests and Examples

Modbus Master: Read N Bits

This table represents requests 01 and 02 (01 for output or memory bit, 02 for input bit):

	Table Index	**Most Significant Byte**	**Least Significant Byte**
Control table	0	01 (Transmission/reception)	06 (Transmission length)(1)
	1	03 (Reception offset)	00 (Transmission offset)
Transmission table	2	Slave@(1...247)	01 or 02 (Request code)
	3	Address of the first bit to read in the slave	
	4	N_1 = Number of bits to read	
Reception table (after response)	5	Slave@(1...247)	01 or 02 (Response code)
	6	00 (byte added by Rx offset action)	N_2 = Number of data bytes to read = [1+(N_1-1)/8], where the result is the integer part of the division.
	7	Value of the first bit (value 00 or 01) expanded into a byte	Value of the second bit (if N_2>1) expanded into a byte
	8	Value of the third bit (if N_1>1) expanded into a byte	–
	...	...	...
	(N_2/2)+6 (if N_2 is even) (N_2/2+1)+6 (if N_2 is odd)	Value of the N_2^{th} bit (if N_1>1) expanded into a byte	–
(1) This byte also receives the length of the string transmitted after response.			

Modbus Master: Read N Words

This table represents requests 03 and 04 (03 for output or memory word, 04 for input word):

	Table Index	**Most Significant Byte**	**Least Significant Byte**
Control table	0	01 (Transmission/reception)	06 (Transmission length)(1)
	1	03 (Reception offset)	00 (Transmission offset)
Transmission table	2	Slave@(1...247)	03 or 04 (Request code)
	3	Address of the first word to read	
	4	N = Number of words to read	
Reception table (after response)	5	Slave@(1...247)	03 or 04 (Response code)
	6	00 (byte added by Rx offset action)	2*N (number of bytes read)
	7	First word read	
	8	Second word read (if N>1)	
	...	...	
	N+6	Word N read (if N>2)	
(1) This byte also receives the length of the string transmitted after response.			

NOTE: The Reception offset of 3 adds a byte (value = 0) at the third position in the reception table. This helps ensure a good positioning of the number of bytes read and of the read words' values in this table.

Modbus Master: Write Bit

This table represents request 05 (write a single bit: output or memory):

	Table Index	**Most Significant Byte**	**Least Significant Byte**
Control table	0	01 (Transmission/reception)	06 (Transmission length)(1)
	1	00 (Reception offset)	00 (Transmission offset)
Transmission table	2	Slave@(1...247) or 0 in case of broadcast	05 (Request code)
	3	Value to write for MSB of the index word 4; whether 0xFF or 0x00(2).	
	4	Bit value to write in the slave (`16#0000` = False and `16#FF00` = True)	
Reception table (after response)	5	Slave@(1...247)	05 (Response code)
	6	Address of the bit written	
	7	Value written	
(1) This byte also receives the length of the string transmitted after response. **(2)** For a bit to write 1, the associated word in the transmission table must contain the value FF00h, and 0 for the bit to write 0.			

NOTE:

- This request does not need the use of offset.
- The response frame is the same as the request frame here (in a normal case).

Modbus Master: Write Word

This table represents request 06 (write a single word: output or memory):

	Table Index	**Most Significant Byte**	**Least Significant Byte**
Control table	0	01 (Transmission/reception)	06 (Transmission length)(1)
	1	00 (Reception offset)	00 (Transmission offset)
Transmission table	2	Slave@(1...247) or 0 in case of broadcast	06 (Request code)
	3	Address of the word to write	
	4	Word value to write	
Reception table (after response)	5	Slave@(1...247)	06 (Response code)
	6	Address of the word written	
	7	Value written	
(1) This byte also receives the length of the string transmitted after response.			

NOTE:

- This request does not need the use of offset.
- The response frame is the same as the request frame here (in a normal case).

Modbus Master: Write of N Bits

This table represents request 15 (write N bits: output or memory):

	Table Index	Most Significant Byte	Least Significant Byte
Control table	0	01 (Transmission/reception)	8 + number of bytes (transmission)
	1	00 (Reception offset)	07 (Transmission offset)
Transmission table	2	Slave@(1...247) or 0 in case of broadcast	15 (Request code)
	3	Address of the first bit to write	
	4	N_1 = Number of bits to write	
	5	00 (byte not sent, offset effect)	N_2 = Number of data bytes to write = [1+(N_1-1)/8], where the result is the integer part of the division.
	6	Value of the first byte	Value of the second byte
	7	Value of the third byte	Value of the fourth byte
	...	...	...
	(N_2/2)+5 (if N_2 is even) (N_2/2+1)+5 (if N_2 is odd)	Value of the N_2^{th} byte	
Reception table (after response)	–	Slave@(1...247)	15 (Response code)
	–	Address of the first bit written	
	–	Number of bits written (= N_1)	

NOTE: The Transmission offset = 7 suppresses the seventh byte in the sent frame. This also allows a correct correspondence of words' values in the transmission table.

Modbus Master: Write of N Words

This table represents request 16:

	Table Index	Most Significant Byte	Least Significant Byte
Control table	0	01 (Transmission/reception)	8 + (2*N) (Transmission length)
	1	00 (Reception offset)	07 (Transmission offset)
Transmission table	2	Slave@(1...247) or 0 in case of broadcast	16 (Request code)
	3	Address of the first word to write	
	4	N = Number of words to write	
	5	00 (byte not sent, offset effect)	2*N = Number of bytes to write
	6	First word value to write	
	7	Second value to write	
	...	...	
	N+5	N values to write	
Reception table (after response)	N+6	Slave@(1...247)	16 (Response code)
	N+7	Address of the first word written	
	N+8	Number of words written (= N)	

NOTE: The Transmission offset = 7 suppresses the seventh byte in the sent frame. This also allows a correct correspondence of words' values in the transmission table.

Modbus Request: Read Device Identification

This table represents request 43 (read device identification):

Rung	Instruction	Comment
0	LD 1 [%MW800: =16#0105] [%MW801: =16#0000] [%MW802: =16#032B] [%MW803: =16#0E01] [%MW804: =16#0000]	[%MW800:=16#0105]: Standard Modbus header [%MW801:=16#0000]: No transmission and reception offset [%MW802:=16#032B]: Slave address, function code [%MW803:=16#0E01]: MEI type, read device ID code [%MW804:=16#0000]: Object ID, unused

Modbus Request: Diagnostic

This table represents request 8 (diagnostic):

Rung	Instruction	Comment
0	LD 1 [%MW1000: =16#0106] [%MW1001: =16#0000] [%MW1002: =16#0308] [%MW1003: =16#0000] [%MW1004: =16#1234]	[%MW1000:=16#0106]: Standard Modbus header [%MW1001:=16#0000]: No transmission and reception offset [%MW1002:=16#0308]: Slave address, function code [%MW1003:=16#0000]: Subfunction code [%MW1004:=16#1234]: Any data The Slave answer will be a copy of the request. This mode is referred to as Echo or Mirror mode.

Example 1: Modbus Application Writing

Master program:

Rung	Instruction	Comment
0	LD 1 [%MW0:=16#0106] [%MW1:=16#0300] [%MW2:=16#0203] [%MW3:=16#0000] [%MW4:=16#0004]	[%MW0:=16#0106]: Transmission length = 6 [%MW1:=16#0300] : Offset reception = 3, offset Transmission = 0 %MW2 to %MW4: Transmission [%MW2:=16#0203]: Slave 2, Fonction 3 (Read multi-words) [%MW3:=16#0000]: First word address to read in the slave: to 0 address [%MW4:=16#0004]: Number of word to read: 4 words (%MW0 to %MW3)
1	LD 1 AND %MSG2.D [EXCH2 %MW0:11]	–
2	LD %MSG2.E ST %Q0.0 END	–

Slave program:

Rung	Instruction	Comment
0	LD 1 [%MW0:=16#6566] [%MW1:=16#6768] [%MW2:=16#6970] [%MW3:=16#7172] END	–

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Using EcoStruxure Machine Expert - Basic, create an application program for both the master and the slave. For the slave, write some memory words to a set of known values. In the master, the word table of the **Exchange** instruction is initialized to read 4 words from the slave at Modbus address 2 starting at location %MW0.

NOTE: Note the use of the Reception offset set in %MW1 of the Modbus master. The offset of 3 will add a byte (value = 0) at the third position in the reception area of the table. This aligns the words in the master so that they fall correctly on word boundaries. Without this offset, each word of data would be split between 2 words in the **Exchange** block. This offset is used for convenience.

Before executing the **EXCH2** instruction, the application verifies the communication bit associated with %MSG2. Finally, the error status of the %MSG2 is detected and stored on the first output bit on the local base controller I/O. Additional error verification using %SW64 could also be added to make this more accurate.

Animation table initializing in online mode corresponding with the reception table part:

Address	Value	Format
%MW5	0203	Hexadecimal
%MW6	0008	Hexadecimal
%MW7	6566	Hexadecimal
%MW8	6768	Hexadecimal
%MW9	6970	Hexadecimal
%MW10	7172	Hexadecimal

After downloading and setting each logic controller to run, open an animation table on the master. Examine the response section of the table to verify that the response code is 3 and that the correct number of bytes was read. Also in this example, the words read from the slave (beginning at %MW7) are aligned correctly with the word boundaries in the master.

Example 2: Modbus Application Writing

Master program:

Rung	Instruction	Comment
0	LD 1 [%MW0: =16#010C] [%MW1: =16#0007] [%MW2: =16#0210] [%MW3: =16#0010] [%MW4: =16#0002] [%MW5: =16#0004] [%MW6: =16#6566] [%MW7: =16#6768]	[%MW0:=16#010C]: Transmission table length: 0C hex = 12 dec, from *%MW2* to *%MW7* [%MW1:=16#0007] [%MW2:=16#0210]: slave address 2, 10h function code write words [%MW3:=16#0010]: from address 16 in the slave [%MW4:=16#0002]: write of 2 words [%MW5:=16#0004]: number of bytes to write [%MW6:=16#6566]: value of the first word [%MW7:=16#6768]: value of the second word
1	LD 1 AND %MSG2.D [EXCH2 % MW0:12]	–
2	LD %MSG2.E ST %Q0.0 END	–

Slave program:

Rung	Instruction	Comment
0	LD 1 [%MW18: =16#FFFF] END	–

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Using EcoStruxure Machine Expert - Basic, create an application program for both the master and the slave. For the slave, write a single memory word %MW18. This will allocate space on the slave for the memory addresses from %MW0 through %MW18. Without allocating the space, the Modbus request would be trying to write to locations that did not exist on the slave.

In the master, the word table of the EXCH2 instruction is initialized to read 4 bytes to the slave at Modbus address 2 at the address %MW16 (10 hexadecimal).

NOTE: Note the use of the Transmission offset set in %MW1 of the Modbus master application. The offset of 7 will suppress the high byte in the sixth word (the value 00 hexadecimal in %MW5). This works to align the data values in the transmission table of the word table so that they fall correctly on word boundaries.

Before executing the EXCH2 instruction, the application verifies the communication bit associated with %MSG2. Finally, the error status of the %MSG2 is detected and stored on the first output bit on the local base controller I/O. Additional detected error verification using %SW64 could also be added to make this more accurate.

Animation table initialization on the master:

Address	Value	Format
%MW0	010C	Hexadecimal
%MW1	0007	Hexadecimal
%MW2	0210	Hexadecimal
%MW3	0010	Hexadecimal
%MW4	0002	Hexadecimal
%MW5	0004	Hexadecimal
%MW6	6566	Hexadecimal
%MW7	6768	Hexadecimal
%MW8	0210	Hexadecimal
%MW9	0010	Hexadecimal
%MW10	0004	Hexadecimal

Animation table initialization on the slave:

Address	Value	Format
%MW16	6566	Hexadecimal
%MW17	6768	Hexadecimal

After downloading and setting each logic controller to run, open an animation table on the slave controller. The 2 values in %MW16 and %MW17 are written to the slave.

In the master, an animation table can be used to examine the reception table portion of the exchange data. This data displays the slave address, the response code, the first word written, and the number of words written starting at %MW8 in the example above.

LIFO/FIFO Register (%R)

Using LIFO/FIFO Register Function Blocks

This section provides descriptions and programming guidelines for using *LIFO/FIFO Register* function blocks.

Description

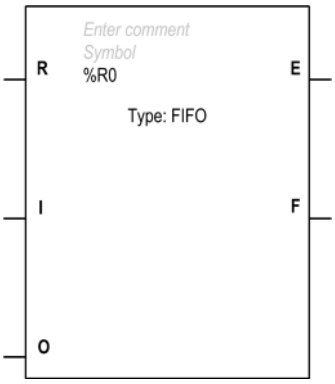
Introduction

A *LIFO/FIFO Register* function block  is a memory block which can store up to 16 words of 16 bits each in 2 different ways:

- Queue (First In, First Out) known as FIFO.
- Stack (Last In, First Out) known as LIFO.

Illustration

This illustration is the *LIFO/FIFO Register* function block.



Inputs

The *LIFO/FIFO Register* function block has the following inputs:

Label	Description	Value
R	Reset input (or instruction)	At state 1, initializes the *LIFO/FIFO Register*.
I	Storage input (or instruction)	On a rising edge, stores the contents of associated word `%Ri.I` in the *LIFO/FIFO Register*.
O	Retrieval input (or instruction)	On a rising edge, loads a data word of the *LIFO/FIFO Register* into associated word `%Ri.O`.

Outputs

The *LIFO/FIFO Register* function block has the following outputs:

Label	Description	Value
E	Empty output (`%Ri.E`)	The associated bit `%Ri.E` indicates that the *LIFO/FIFO Register* is empty. The value of `%Ri.E` can be tested, for example, in an animation table or with an instruction.
F	Full output (`%Ri.F`)	The associated bit `%Ri.F` indicates that the *LIFO/FIFO Register* is full. The value of `%Ri.F` can be tested, for example, in an animation table or with an instruction.

Configuration

Parameters

To configure parameters, follow the Configuring a Function Block procedure, page 106 and read the description of Memory Allocation Modes in the EcoStruxure Machine Expert - Basic Operating Guide.

The *LIFO/FIFO Register* function block has the following parameters:

Parameter	Description	Value	Editable in Online Mode?
Used	Address used	If selected, this address is currently in use in a program.	No
Address	*LIFO/FIFO Register* object address	A program can contain only a limited number of *LIFO/FIFO Register* objects. Refer to the Programming Guide of the hardware platform for the maximum number of registers.	No
Symbol	Symbol	The symbol associated with this object. Refer to the EcoStruxure Machine Expert - Basic Operating Guide, Defining and Using Symbols (see EcoStruxure Machine Expert - Basic, Operating Guide) for details.	No
Type	*LIFO/FIFO Register* type	**FIFO** (queue) or **LIFO** (stack).	Yes
Comment	Comment	A comment can be associated with this object.	No

Objects

The *LIFO/FIFO Register* function block has the following objects:

Object	Description	Value
`%Ri.I`	*LIFO/FIFO Register* input word	Can be read, tested, and written. It can be modified in an animation table.
`%Ri.O`	*LIFO/FIFO Register* output word	Can be read, tested, and written. It can be modified in an animation table.
`%Ri.E`	Empty output	See Outputs table above.
`%Ri.F`	Full output	See Outputs table above.

Special Cases

This table contains a list of special cases for programming the *LIFO/FIFO Register* function block:

Special Case	Description
Effect of a cold restart (`%S0=1`) or **INIT**	Initializes the contents of the *LIFO/FIFO Register*. The output bit *%Ri.E* associated with the output *E* is set to 1.
Effect of a warm restart (`%S1=1`) or a controller stop	Has no effect on the current value of the *LIFO/FIFO Register*, nor on the state of its output bits.
Rising edge detected on both `%Ri.O` and `%Ri.I`	If a rising edge on both `%Ri.O` and `%Ri.I` is detected in the same `LIFO/FIFO Register` function block call, the values are neither stored nor retrieved; your program must manage the storage or retrieval of the values.

NOTE: Effect of **INIT** is the same as `%S0=1`.

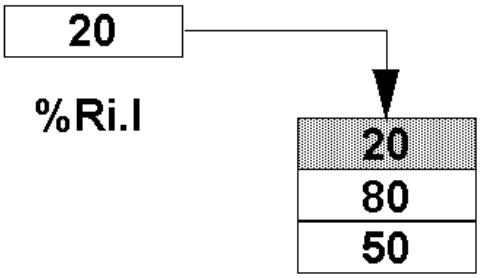
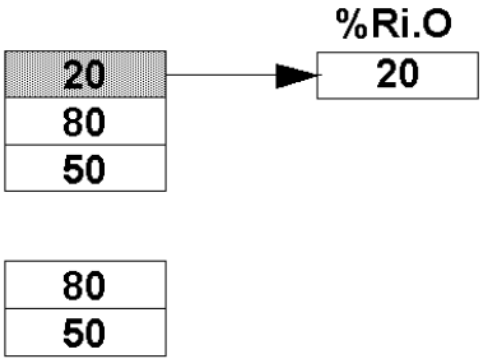
LIFO Register Operation

Introduction

In LIFO operation (Last In, First Out), the last data item entered is the first to be retrieved.

Operation

This table describes LIFO operation:

Stage	Description	Example
1	**Storage**: When a storage request is received (rising edge at input *I* or activation of instruction *I*), the contents of input word *%Ri.I* are stored at the top of the stack. When the stack is full (output *F*=1), no further storage is possible.	Storage of the contents of *%Ri.I* at the top of the stack. 
2	**Retrieval**: When a retrieval request is received (rising edge at input *O* or activation of instruction *O*), the highest data word (last word to be entered) is loaded into word *%Ri.O*. When the *LIFO/FIFO Register* is empty (output *E*=1), no further retrieval is possible. Output word *%Ri.O* does not change and retains its value.	Retrieval of the data word highest in the stack. 
3	**Reset**: The stack can be reset at any time (state 1 at input *R* or activation of instruction *R*). The stack is empty after a reset (*%Ri.E* =1).	–

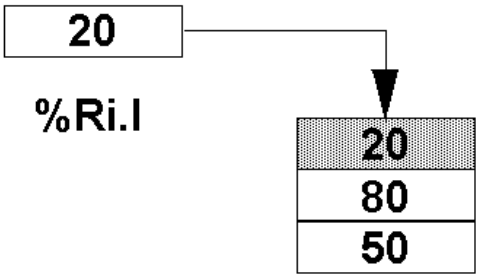
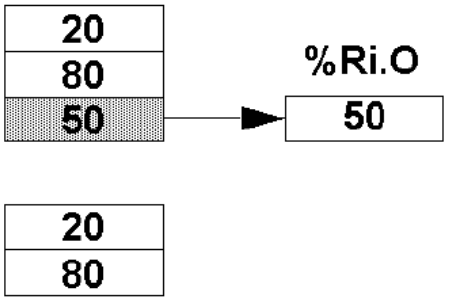
FIFO Register Operation

Introduction

In FIFO operation (First In, First Out), the first data item entered is the first to be retrieved.

Operation

This table describes FIFO operation:

Stage	Description	Example
1	**Storage**: When a storage request is received (rising edge at input *I* or activation of instruction *I*), the contents of input word *%Ri.I* are stored at the top of the queue. When the queue is full (output *F*=1), no further storage is possible.	Storage of the contents of *%Ri.I* at the top of the stack. 
2	**Retrieval**: When a retrieval request is received (rising edge at input *O* or activation of instruction *O*), the data word lowest in the queue is loaded into output word *%Ri.O* and the contents of the *LIFO/FIFO Register* are moved down one place in the queue. When the *LIFO/FIFO Register* is empty (output *E*=1), no further retrieval is possible. Output word *%Ri.O* does not change and retains its value.	Retrieval of the first data item which is then loaded into *%Ri.O*. 
3	**Reset**: The queue can be reset at any time (state 1 at input *R* or activation of instruction *R*). The queue is empty after a reset (*%Ri.E*=1).	–

Programming Example

Introduction

The following programming example shows the content of a memory word (%MW34) being loaded into a *LIFO/FIFO Register* (*%R2.I*) on reception of a storage request (%I0.2) if *LIFO/FIFO Register %R2* is not full (%R2.F = 0). The storage request in the *LIFO/FIFO Register* is made by %M1. The retrieval request is confirmed by input %I0.3, and *%R2.O* is loaded into %MW20 if the register is not empty (%R2.E = 0).

Programming

This example is a *LIFO/FIFO Register* function block with reversible instructions:

Rung	Reversible Instruction
0	BLK %R2 LD %M1 I LD %I0.3 ANDN %R2.E O END_BLK
1	LD %I0.3 [%MW20:=%R2.O]
2	LD %I0.2 ANDN %R2.F [%R2.I:=%MW34] ST %M1

This example is the same *LIFO/FIFO Register* function block with non-reversible instructions:

Rung	Non-Reversible Instruction
0	LD %M1 I %R2
1	LD %I0.3 ANDN %R2.E O %R2
2	LD %I0.3 [%MW20:=%R2.O]
3	LD %I0.2 ANDN %R2.F [%R2.I:=%MW34] ST %M1

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Drums (%DR)

Using Drum Function Blocks

This section provides descriptions and programming guidelines for using *Drum* function blocks.

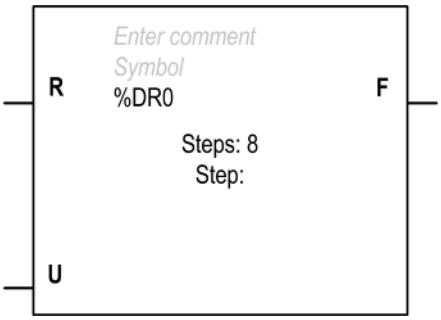
Description

Introduction

The *Drum* function block operates on a principle similar to an electromechanical drum sequencer, which changes step according to external events. On each step, the high point of a cam gives a command, which is then executed by the controller. In the case of a *Drum* function block, these high points are symbolized by state 1 for each step and are assigned to output bits *%Qi.j*, or memory bits `%Mi`.

Illustration

This illustration is the *Drum* function block in offline mode.



Steps Displays the total number of steps configured in the **Drum Assistant**.

Step Appears in offline mode when a block is created. In online mode, it displays the current step number.

Inputs

The *Drum* function block has the following inputs:

Label	Description	Value
R	To return to step 0 (or instruction)	At state 1, sets the **Drum** to step 0.
U	Advance input (or instruction)	On a rising edge, causes the **Drum** to advance by 1 step and updates the control bits.

Outputs

The *Drum* function block has the following output:

Label	Description	Value
F	Output (`%DRi.F`)	Indicates that the current step equals the last step defined. The associated bit *%DRi.F* can be tested.

Configuration

Parameters

To configure parameters, follow the Configuring a Function Block procedure, page 106 and read the description of Memory Allocation Modes in the EcoStruxure Machine Expert - Basic Operating Guide.

The *Drum* function block has the following parameters:

Parameter	Description	Value	Editable in online mode?
Used	Address used	If selected, this address is currently in use in a program.	No
Address	*Drum* object address	A program can contain only a limited number of *Drum* objects. Refer to the *Programming Guide* of your controller for the maximum number of *Drum* objects.	No
Symbol	Symbol	The symbol associated with this object. Refer to the EcoStruxure Machine Expert - Basic Operating Guide, Defining and Using Symbols (see EcoStruxure Machine Expert - Basic, Operating Guide) for details.	No
Configuration	Drum assistant	Click to display the Drum Assistant, page 148.	Yes (all parameters on the Drum Assistant window)
Comment	Comment	A comment can be associated with this object.	No

Objects

The *Drum* function block has the following object:

Object	Description	Value
`%DRi.S`	Current step number	0<=*%DRi.S*<=7. Word which can be read and written. Written value must be a decimal immediate value. When written, the effect takes place on the next execution of the function block. It can be modified in an animation table or in online mode.
`%DRi.F`	Full	See Outputs table, page 147.

Operation

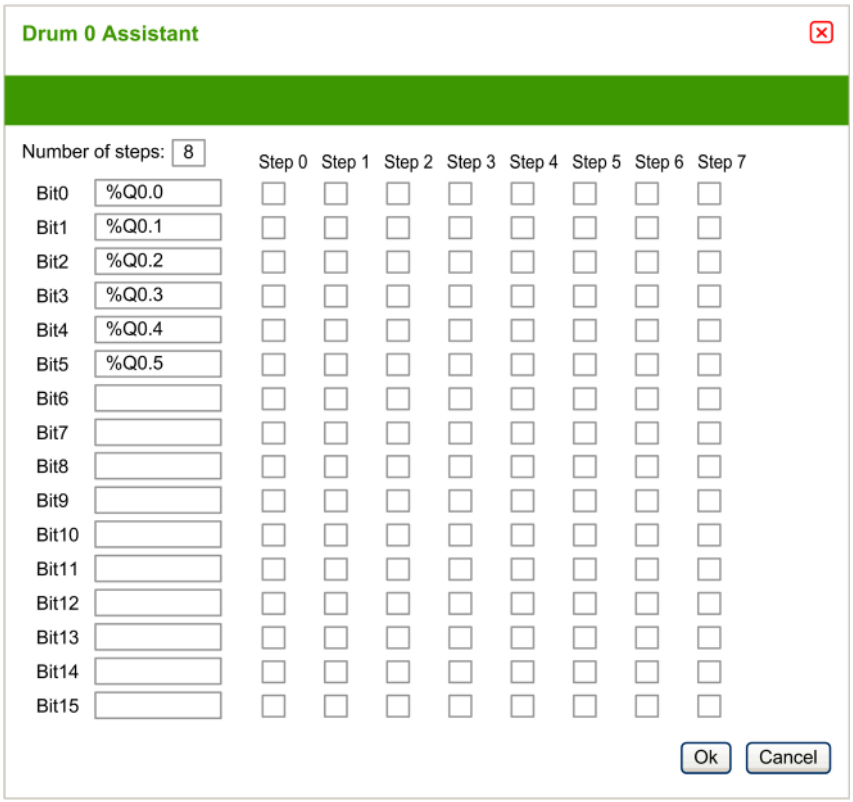
The *Drum* function block consists of:

- A matrix of constant data (the cams) organized in 8 steps (0 to 7) and 16 bits (state of the step) arranged in columns numbered 0 to 15.
- A list of control bits is associated with a configured output (`%Qi.j`), or memory word (`%Mi`). During the current step, the control bits take on the binary states defined for this step.

Drum Assistant

Use the **Drum Assistant** to configure the Drum function block.

Configure the **Number of steps**: 1...8 and the outputs or memory bits associated with each step: **Bit0 ... Bit15**, then click **OK**.



NOTE: The configuration can also be realized using memory bits (%Mi).

Special Cases

This table contains a list of special cases for *Drum* operation:

Special Case	Description
Effects of a cold restart (`%S0=1`)	Resets the *Drum* to step 0 (update of control bits).
Effect of a warm restart (`%S1=1`)	Updates the control bits after the current step.
Effect of a program jump	The fact that the *Drum* is no longer scanned means the control bits retain their last state.
Updating the control bits	Only occurs when there is a change of step or in the case of a warm or cold restart.

Programming Example

Introduction

The following is an example of programming a *Drum* that is configured such that none of the controls are set in step 0 and the controls are set for step 1 to step 6 on the outputs `%Q0.0` to `%Q0.5` respectively (see the Configuration, page 152).

The first 6 outputs `%Q0.0` to `%Q0.5` are activated in succession each time input `%I0.1` is set to 1. Input `%I0.0` resets the following to 0 when it is high:

- Drum output **F** (*%DRi.F* = 0)
- Current step number (*%DRi.S* = 0)

Programming

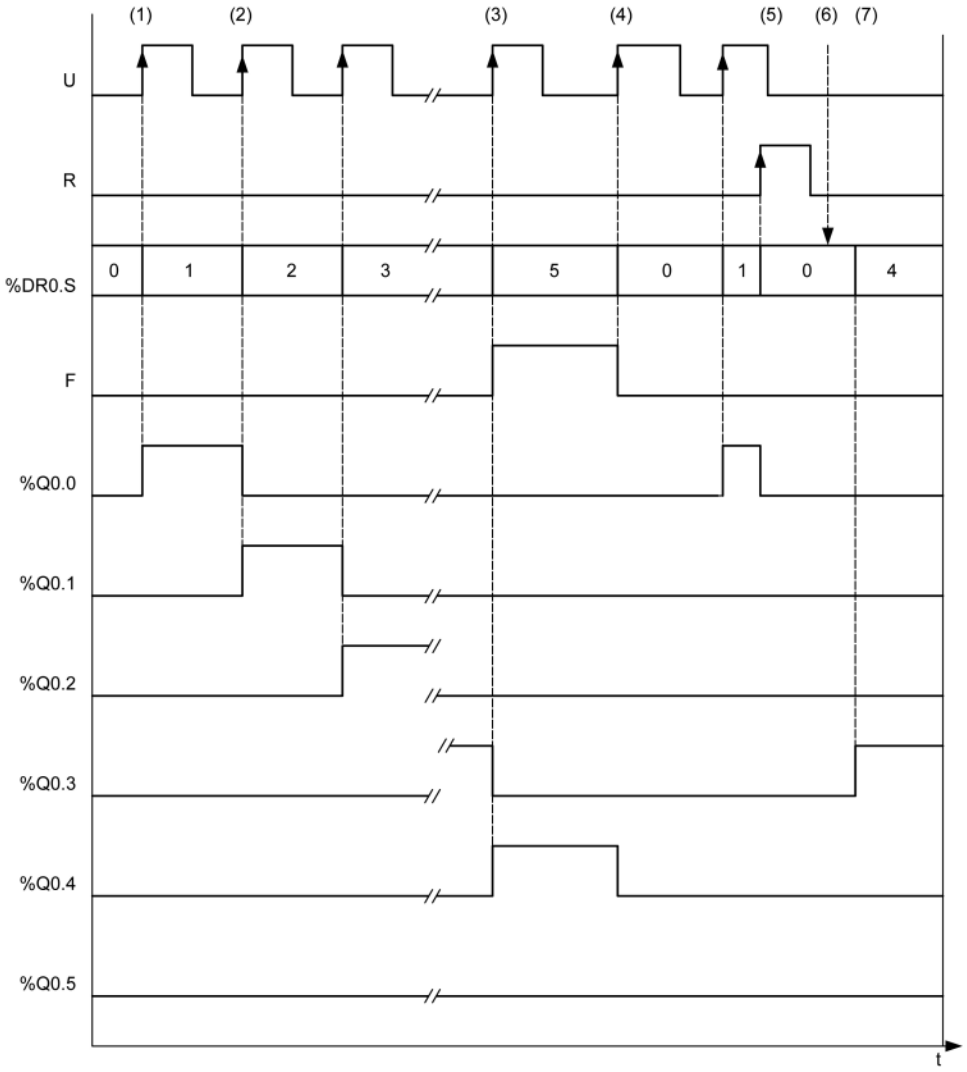
This example is a *Drum* function block program:

Rung	Instruction
0	BLK %DR1 LD %I0.0 R LD %I0.1 U OUT_BLK LD F ST %Q0.7 END_BLK

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

This diagram illustrates the operation of the *Drum*:



(1) At a rising edge on *U* input the current step is incremented

(2) When the current step is updated, the outputs are updated

(3) When the last step is reached, the output *F* is set to 1

(4) A rising edge at *U* input when the last step is active, resets the current step to 0

(5) `%DR0.R = 1` (rising edge) the current value is set to 0

(6) The user writes the value of the step number: `%DR0.S = 4`

(7) The value written by the user is updated at the next execution time

Configuration

The following information is defined during configuration:

- Number of steps: 6
- The output states (control bits) for each *Drum* step:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Step 0:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Step 1:	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Step 2:	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Step 3:	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Step 4:	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Step 5:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

- Assignment of the control bits:

This table presents the associated outputs of the control bits:

Bit	Associated Output
0	No associated output
1	%Q0.1
2	%Q0.2
3	%Q0.3
4	%Q0.4
5	%Q0.5

Shift Bit Register (%SBR)

Using Shift Bit Register Function Blocks

This section provides descriptions and programming guidelines for using *Shift Bit Register* function blocks.

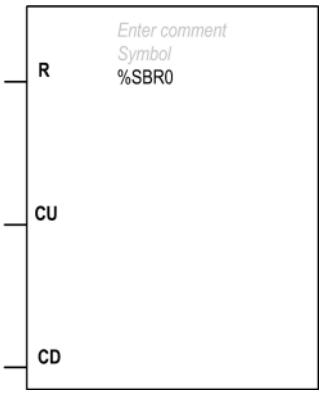
Description

Introduction

The *Shift Bit Register* function block  provides a left or right shift of binary data bits (0 or 1).

Illustration

This illustration is the *Shift Bit Register* function block:



The value of the *Shift Bit Register* is displayed in the centre of the function block:

- Decimal value, for example 7
- Binary value, for example 111
- Hexadecimal value, for example 16#7

Inputs

The *Shift Bit Register* function block has the following inputs:

Label	Description	Value
R	Reset input (or instruction)	When function parameter *R* is 1, this sets register bits 0 to 15 *%SBRi.j* to 0.
CU	Shift to left input (or instruction)	On a rising edge, shifts a register bit to the left.
CD	Shift to right input (or instruction)	On a rising edge, shifts a register bit to the right.

Configuration

Parameters

To configure parameters, follow the Configuring a Function Block procedure, page 106 and read the description of Memory Allocation Modes in the EcoStruxure Machine Expert - Basic Operating Guide.

The *Shift Bit Register* function block has the following parameters:

Parameter	Description	Value
Used	Address used	If selected, this address is currently in use in a program.
Address	**Shift Bit Register** object address	A program can contain only a limited number of *Shift Bit Register* objects. Refer to the Programming Guide of the hardware platform for the maximum number of registers.
Symbol	Symbol	The symbol associated with this object. Refer to the EcoStruxure Machine Expert - Basic Operating Guide, Defining and Using Symbols (see EcoStruxure Machine Expert - Basic, Operating Guide) for details.
Comment	Comment	A comment can be associated with this object.

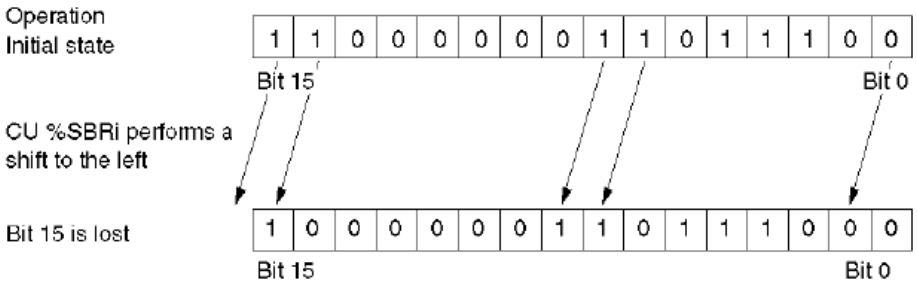
Objects

The *Shift Bit Register* function block has the following objects:

Object	Description	Value
%SBRi	Register number	0 to 7 It can be modified in an animation table.
%SBRi.j	Register bit	Bits 0 to 15 (j = 0 to 15) of the shift register can be tested by a test instruction and written using an Assignment instruction.

Operation

This illustration shows a bit pattern before and after a shift operation:



This is also true of a request to shift a bit to the right (bit 15 to bit 0) using the *CD* instruction. Bit 0 is lost.

If a 16-bit register is not adequate, it is possible to use the program to cascade several Register.

Special Cases

This table contains a list of special cases for programming the *Shift Bit Register* function block:

Special Case	Description
Effect of a cold restart (%S0=1)	Sets all the bits of the register word to 0.
Effect of a warm restart (%S1=1)	Has no effect on the bits of the register word.

Programming Example

Introduction

The *Shift Bit Register* function block provides a left or right shift of binary data bits (0 or 1).

Programming

In this example, a bit is shifted to the left every second while bit 0 assumes the state to bit 15.

In reversible instructions:

Rung	Reversible Instruction
0	BLK %SBR0 LD %S6 CU END_BLK
1	LD %SBR0.15 ST %SBR0.0

In non-reversible instructions:

Rung	Non-Reversible Instruction
0	LD %S6 CU %SBR0
1	LD %SBR0.15 ST %SBR0.0

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Step Counter (%SC)

Using Step Counter Function Blocks

This section provides descriptions and programming guidelines for using *Step Counter* function blocks.

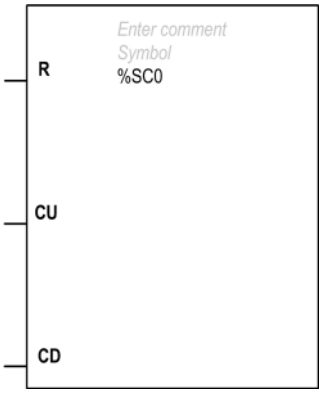
Description

Introduction

A *Step Counter* function block  provides a series of steps to which actions can be assigned. Moving from one step to another depends on external or internal events. Each time a step is active, the associated bit (*Step Counter* bit *%SCi.j*) is set to 1. Only one step of a *Step Counter* can be active at a time.

Illustration

This illustration is a *Step Counter* function block:



Inputs

The *Step Counter* function block has the following inputs:

Label	Description	Value
R	Reset input (or instruction)	When function parameter *R* is 1, this resets the *Step Counter*.
CU	Increment input (or instruction)	On a rising edge, increments the *Step Counter* by one step.
CD	Decrement input (or instruction)	On a rising edge, decrements the *Step Counter* by one step.

Configuration

Parameters

To configure parameters, follow the Configuring a Function Block procedure, page 106 and read the description of Memory Allocation Modes in the EcoStruxure Machine Expert - Basic Operating Guide.

The *Step Counter* function block has the following parameters:

Parameter	Description	Value
Used	Address used	If selected, this address is currently in use in a program.
Address	*Step Counter* object address	A program can contain only a limited number of *Step Counter* objects. Refer to the Programming Guide of the hardware platform for the maximum number of *Step Counter*.
Symbol	Symbol	The symbol associated with this object. Refer to the EcoStruxure Machine Expert - Basic Operating Guide, Defining and Using Symbols for details.
Comment	Comment	A comment can be associated with this object.

Objects

The *Step Counter* function block has the following object:

Object	Description	Value
`%SCi.j`	*Step Counter* bit	*Step Counter* bits 0 to 255 (j = 0 to 255) can be tested by a load logical operation and written by an Assignment instruction. It can be modified in an animation table.

Special Case

This table contains a list of special cases for operating the *Step Counter* function block:

Special Case	Description
Effect of a cold restart (`%S0=1`)	Initializes the *Step Counter*.
Effect of a warm restart (`%S1=1`)	Has no effect on the *Step Counter*.

Programming Example

Introduction

This example is a *Step Counter* function block.

- *Step Counter* 0 is decremented by input %I0.1.
- *Step Counter* 0 is incremented by input %I0.2.
- *Step Counter* 0 is reset to 0 by input %I0.3 or when it arrives at step 3.
- Step 0 controls output %Q0.1, step 1 controls output %Q0.2, and step 2 controls output %Q0.3.

Programming

This example is a *Step Counter* function block with reversible instructions:

Rung	Reversible Instruction
0	BLK %SC0 LD %SC0.3 OR %I0.3 R LD %I0.2 CU LD %I0.1 CD END_BLK
1	LD %SC0.0 ST %Q0.1
2	LD %SC0.1 ST %Q0.2
3	LD %SC0.2 ST %Q0.3

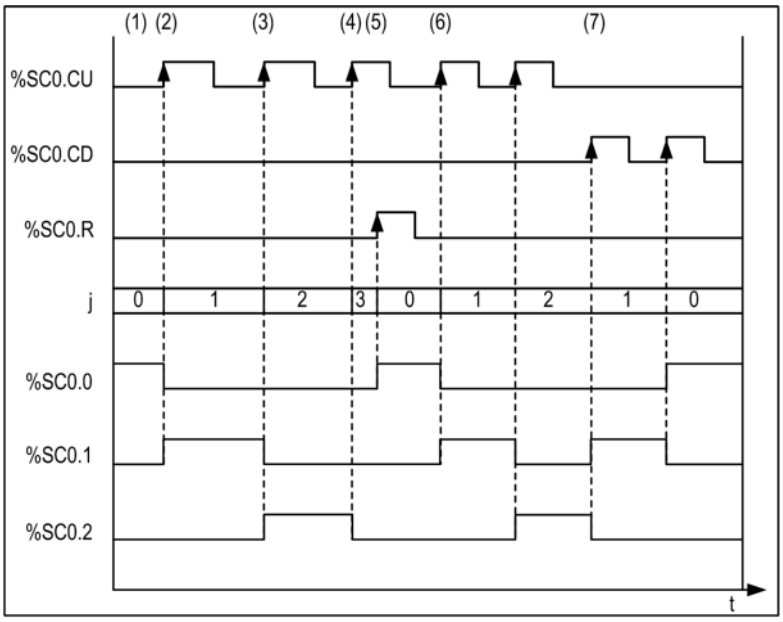
This example is a *Step Counter* function block with non-reversible instructions:

Rung	Non-Reversible Instruction
0	LD %SC0.3 OR %I0.3 R %SC0
1	LD %I0.2 CU %SC0
2	LD %I0.1 CD %SC0
3	LD %SC0.0 ST %Q0.1
4	LD %SC0.1 ST %Q0.2
5	LD %SC0.2 ST %Q0.3

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

This diagram illustrates the operation of the *Step Counter* function block:



(1) Step 0 is active so *%SC0.0* is set to 1

(2) At the rising edge of *CU* input, the step is incremented and the outputs are updated

(3) The step is incremented and outputs are updated

(4) The step 3 is active so the **Reset** input is active after one cycle

(5) When **Reset** is active, the current step is set to 0 and the reset input is set to 0 after one cycle

(6) The current step is incremented at rising edge of *CU* input

(7) At rising edge of *CD* input, the step is decremented and outputs are updated

Schedule Blocks (%SCH)

Using Schedule Blocks

This section provides descriptions and programming guidelines for using *Schedule blocks*.

Description

Introduction

Schedule blocks are used to control actions at a predefined month, day, and time.

Schedule blocks are only configured in EcoStruxure Machine Expert - Basic; they cannot be inserted into a program rung in the same way as other function blocks.

> **NOTE:** Verify system bit `%S51` and system word `%SW118` to confirm that the Real-Time Clock (RTC) option is installed. The RTC option is required for using *Schedule blocks*.

Schedule blocks can be configured either:

- In the **Schedule block properties** window (**Tools tab > Schedule Blocks**).
- Using dedicated objects %SCHi.xxx, page 159.

Configuring Schedule Block Properties

To configure a schedule block, read the description of Memory Allocation Modes (see EcoStruxure Machine Expert - Basic, Operating Guide).

The *Schedule blocks* has the following parameters:

Parameter	Description	Value
Used	Address used	If selected, this address is currently in use in a program.
Address	*Schedule blocks* object address	A program can contain only a limited number of *Schedule blocks* objects. Refer to the Programming Guide of the hardware platform for the maximum number of *Schedule blocks*.
Configured	Whether the selected *Schedule blocks* number is configured for use.	If checkbox is selected, it is configured for use. Otherwise, it is not used.
Output bit	Output bit	Output assignment is activated by the *Schedule blocks*: `%Mi` or `%Qj.k`. This output is set to 1 when the current date and time are between the setting of the start of the active period and the setting of the end of the active period.
Start Day	The day in the month to start the *Schedule blocks*.	**1...31**
Start Month	The month to start the *Schedule blocks*.	*Schedule blocks*.
End Day	The day in the month to end the *Schedule blocks*.	**1...31**
End Month	The month to end the *Schedule blocks*.	**January...December**
Start Time	The time-of-day, hours, and minutes to start the *Schedule blocks*.	Hour: **0...23** Minute: **0...59**
End Time	The time-of-day, hours, and minutes to end the *Schedule blocks*.	Hour: **0...23** Minute: **0...59**
Monday	Check boxes that identify the day(s) of the week for activation of the *Schedule blocks*.	If checkbox is selected, it is configured for use. Otherwise, it is not used.
Tuesday		
Wednesday		
Thursday		
Friday		
Saturday		
Sunday		
Comment	Comment	A comment can be associated with this object.

Objects

These objects are available only if the block is configured as explained above.

The application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 6.0**.

The objects of the schedule blocks are:

Objects	Description
%SCHi.STARTDAY	The day in the month to start the *Schedule blocks*.
%SCHi.STARTMONTH	The month to start the *Schedule blocks*.
%SCHi.STARTHOUR	The hour to start the *Schedule blocks*.
%SCHi.STARTMIN	The minute to start the *Schedule blocks*.
%SCHi.ENDDAY	The day in the month to end the *Schedule blocks*.
%SCHi.ENDMONTH	The month to end the *Schedule blocks*.
%SCHi.ENDHOUR	The hour to end the *Schedule blocks*.
%SCHi.ENDMIN	The minute to end the *Schedule blocks*.
%SCHi.DOW	The day of the week to activate the *Schedule blocks*.

Modifying these object values in the program allows to dynamically change schedule block configuration. The modification is taken into account at the next MAST scan.

After a cold restart (`%S0`=1), the object values are reset to the value defined during configuration.

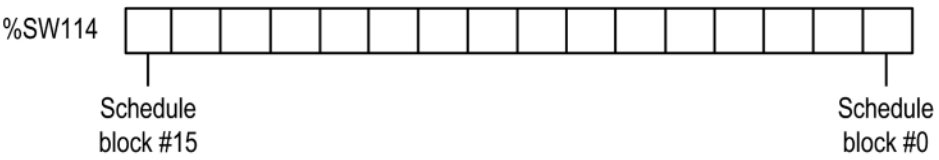
Configuring Schedule Blocks Using System or Memory Words

You can also configure **Schedule Blocks** using system words or memory words.

Enabling Schedule Blocks

The bits of system word `%SW114` enable (bit set to 1) or disable (bit set to 0) the operation of each of the 16 Schedule blocks.

Assignment of *Schedule blocks* in `%SW114`:



By default (or after a cold restart) the bits of this system word are set to 1 (enabled). Use of these bits by the program is optional.

Output of Schedule Blocks

If the same output (`%Mi` or `%Qj.k`) is assigned by several schedule blocks, it is the *OR* of the results of each of the blocks which is finally assigned to this object (it is possible to have several Schedule blocks for the same output).

For example, schedule block `%SCH0` and `%SCH1` are both assigned to output `%Q0.0`. `%SCH0` sets the output from 12:00 h to 13:00 h on Monday, and `%SCH1` sets the output from 12:00 h to 13:00 h on Tuesday. The result is that the output is set from 12:00 h to 13:00 h on both Monday and Tuesday.

Programming and Configuring

Introduction

Schedule blocks are used to control actions at a predefined month, day, and time.

Programming Example

This table shows the parameters for a summer month spray program example:

Parameter	Value	Description	Editable in online mode?
Address	**Real-Time Clock 6**	*Schedule blocks* number 6	No
Configured	Box checked	Box checked to configure the *Schedule blocks* number 6.	No
Output bit	**%Q0.2**	Activate output `%Q0.2`	Yes
Start Day	**21**	Start activity on the 21 day of June	Yes
Start Month	**June**	Start activity in June	Yes
Start Time	**21**	Start activity at 21:00	Yes
End Day	**21**	Stop activity on the 21st of September	Yes
End Month	**September**	Stop activity in September	Yes
End Time	**22**	Stop activity at 22:00	Yes
Monday	Box checked	Run activity on Monday	Yes
Tuesday	Box not checked	No activity	Yes
Wednesday	Box checked	Run activity on Wednesday	Yes
Thursday	Box not checked	No activity	Yes
Friday	Box checked	Run activity on Friday	Yes
Saturday	Box not checked	No activity	Yes
Sunday	Box not checked	No activity	Yes

Using this program, the *Schedule blocks* can be disabled through a switch or a humidity detector wired to input `%I0.1`:

Rung	Instruction	Comment
0	`LD %I0.1` `ST %SW114:X6`	In this example, the `%SCH6` is validated.

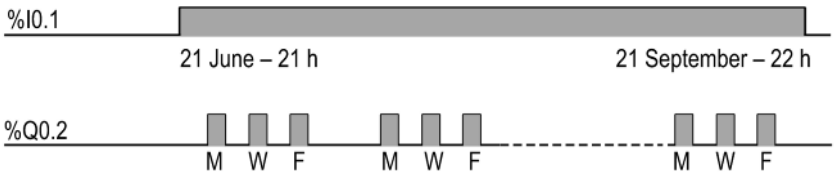
NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

The following table presents a program example:

Rung	Instruction
0	LD %M4 [%M0:4 := 7]
1	LD %M0 RISING1 OPER [%SCH0.STARTDAY := %MW0] OPER [%SCH0.STARTMONTH := %MW1] OPER [%SCH0.STARTHOUR := %MW2] OPER [%SCH0.STARTMIN := %MW3] R %M0
2	LD %M1 RISING0 OPER [%SCH0.ENDDAY := %MW4 OPER [%SCH0.ENDMONTH := %MW5] OPER [%SCH0.ENDHOUR := %MW6] OPER [%SCH0.ENDMIN := %MW7] R %M1
3	LD %M2 RISING2 [%SCH0.DOW := %MW8]

Timing Diagram

This timing diagram shows the activation of output %Q0.2:



Real Time Clock (%RTC)

Using RTC Function Blocks

This section provides descriptions and programming guidelines for using *RTC* function blocks.

NOTE: Your application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 5.0** to use RTC function blocks.

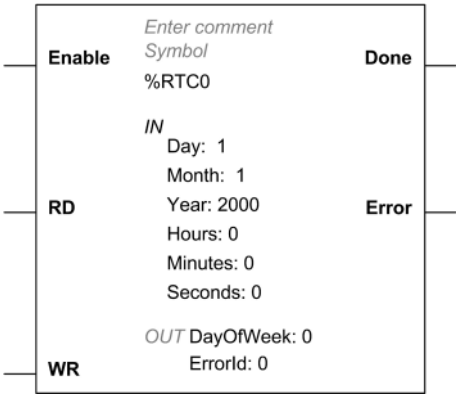
Description

Introduction

The RTC function block allows you to read from and write to the Real-Time Clock (RTC) of the M221 Logic Controller.

Illustration

This illustration shows the RTC function block.



Inputs

The *RTC* function block has the following inputs:

Label	Object	Value
Enable	-	Enables the function block when a rising edge of this input is detected. At state 1, the **RD** and **WR** input values are read continuously to determine the action to take. At state 0, the function block is disabled and the outputs are reset.
RD	-	A combination of the values of the 2 inputs determines the action to take: • *RD* = 0 and *WR* = 0. No action. The *Done* and *Error* outputs are set to 0. • *RD* = 1 and *WR* = 0. Read the RTC value. If successful, the *Done* output is set to 1 and the *Error* output is set to 0. The input objects are continuously updated with the values read from the RTC. Use an animation table (see EcoStruxure Machine Expert - Basic, Operating Guide) to display the object values. • *RD* = 0 and *WR* = 1. When the rising edge of *WR* is detected, update the RTC using the object values specified in the RTC parameters associated with this function block (see below). If the update is successful, the *Done* output is set to 1 and the *Error* output is set to 0. The RTC is updated. If not successful, the *Done* output is set to 0 and the *Error* output is set to 1. • *RD* = 1 and *WR* = 1. Not supported. The *Done* output is set to 0, the *Error* output is set to 1, and the ErrorId output, page 165 is set to 256 (Simultaneous read and write).
WR	-	
Day	*%RTCi.DAY*	Day Default value: 12 1...31
Month	*%RTCi.MONTH*	Month Default value: 6 1...12
Year	*%RTCi.YEAR*	Year Default value: 2017 Greater than or equal to 2000
Hours	*%RTCi.HOURS*	Hours Default value: 0 0...23
Minutes	*%RTCi.MINUTES*	Minutes Default value: 0 0...59
Seconds	*%RTCi.SECONDS*	Seconds Default value: 0 0...59

Outputs

The **RTC** function block has the following outputs:

Label	Object	Value
Done	*%RTCi.Done*	Set to 1 when the RTC is successfully read from or written to. Set to 0 when the read or write operation was not successful.
Error	*%RTCi.Error*	Set to 1 if an error occurs during execution. Function block execution is finished. The *ErrorId* output object indicates the cause of the error.
DayOfWeek	*%RTCi. DayOfWeek*	Returns the day of the week, calculated from the current week value. Range: 0...7 0: Function block not yet executed 1...7: Day of the week.
ErrorId	*%RTCi.ErrorId*	Error code identifier. See Error Codes, page 165 below.

Error Codes

The following codes can be returned in the *%RTCi.ErrorId* object when the **Error** output is set to 1.

Error Code	Description
0	No error
1	Year error
2	Month error
3	Day error
4	Week error
5	Hour error
6	Minute error
7	Seconds error
8	Combination error
9	RTC internal error
256	Simultaneous read and write
257	RTC update already in progress (system bit %S50 = 1)

Configuration

Presentation

Configure the RTC function block properties with the values to use to update the RTC in the logic controller.

To display the **RTC properties** page, either:

- Double-click an RTC function block.
- On the **Programming** tab, choose **Tools > Software objects > RTC**.

RTC Properties

To configure parameters, follow the Configuring a Function Block procedure, page 106 and read the description of Memory Allocation Modes in the EcoStruxure Machine Expert - Basic Operating Guide.

The **RTC Properties** page displays the following properties:

Property	Description	Value	Editable in online mode?
Used	Object address is in use	If selected, this address is currently in use in a program.	No
Address	*RTC* object address	*%RTC*i, where i is the object number. A program can contain only a limited number of *RTC* objects. Refer to the *Programming Guide* of your controller for the maximum number of *RTC* objects.	No
Symbol	Symbol	The symbol associated with this object. Refer to the EcoStruxure Machine Expert - Basic Operating Guide, Defining and Using Symbols (see EcoStruxure Machine Expert - Basic, Operating Guide) for details.	Yes
Day	The day of the month	1...31	Yes
Month	The month of the year	1...12	Yes
Year	The year	Greater than or equal to 2000	Yes
Hours	The hour of the day	0...23	Yes
Minutes	The minutes of the hour	0...59	Yes
Seconds	The seconds of the minute	0...59	Yes
Comment	Comment	A comment can be associated with this object.	Yes

PID Function

Introduction

The PID function is used to control a dynamic process continuously. The purpose of PID control is to keep a process running as close as possible to a desired set point.

Refer to the Modicon M221 Logic Controller Advanced Functions, Library Guide for detailed information on the PID behavior, functionalities, and implementation of the PID function:

- PID Operating Modes
- PID Auto-Tuning Configuration
- PID Standard Configuration
- PID Assistant
- PID Programming
- PID Parameters
  - Role and Influence of PID Parameters
  - PID Parameter Adjustment Method

Data Logging

Function Block Description

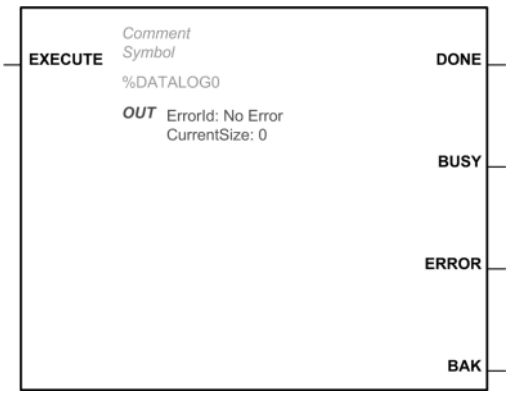
The data logging feature allows you to retentively store data from objects or strings in the file */user/DATALOGx.csv*, where x is the integer corresponding to the data logging instance number. This file is stored on the SD card.

When *DATALOGx.csv* exceeds the defined maximum file size, it is renamed as *DATALOGx.bak*, overwriting any existing *DATALOGx.bak* file. New logging data is then stored in the new *DATALOGx.csv* file. *DATALOGx.bak* is also stored on the SD card.

The SD card name must be `DATA`.

NOTE: The application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 6.0** and the boot loader version 50 or later. Refer to the system word (see Modicon M221 Logic Controller, Programming Guide) *%SW13*.

Graphical representation



Configuring Data Logging

Step	Action
1	In the **Programming** window, click **Tools** > **Software Objects** > **Data Logging**.
2	In the **Configuration** column, click . **Result:** The **DATALOG assistant** appears.
3	Type the **Maximum file size**. The **Maximum file size** can be 1...500. The default value is 100.
4	Select **Historical data** or **Event log**. The **Historical data** feature allows you to store several data from objects. Only the EcoStruxure Machine Expert - Basic objects are allowed. You can store up to 32 objects. The **Event log** feature allows you to store one string (see EcoStruxure Machine Expert - Basic, Operating Guide).
5	Clear or Select **Add time stamping**. This feature adds the time of the event into the `CSV` file.
6	Type the object in the **Address** column. If you selected **Event log**, type the first `%MW` of the string. If you assigned a symbol to the object, it is displayed in the **Symbol** column. You cannot change the symbol here.
7	Click **Apply**.
8	To insert the function block in the Ladder editor, do one of the following methods: • Click , then . • Drag and drop from **Data Logging properties**.

Input

The following table describes the input of the function block:

Label	Type	Description
`EXECUTE`	BOOL	Starts function block execution when a rising edge is detected. If a second rising edge is detected during the execution of the function block, it is ignored and the ongoing command is not affected.

Outputs

The following table describes the outputs of the function block:

Label	Type	Description
`DONE`	BOOL	If TRUE, indicates that the function block execution is completed successfully with no errors detected.
`BUSY`	BOOL	If `TRUE`, indicates that the function block execution is in progress.
`ERROR`	BOOL	If `TRUE`, indicates that an error was detected. Function block execution is stopped. The `Errorld` output indicates the error code.
`BAK`	BOOL	If `TRUE`, indicates that the `BAK` file is created.

Error Codes

Error Code	Description
0	No error.
1	Operation in progress.
2	Error when storing data in the CSV file.
3	Error when creating the BAK file.
4	Datalog instance already used.
10	Restore timeout. Eject the SD card.
11	SD card not detected.
12	SD card write protected.
50	File system error.
51	Error when opening the CSV file.

Grafcet Steps

Overview

Grafcet Step objects (%X*i*) are used to identify the status of the corresponding Grafcet step *i* in a program.

Parameters

The Grafcet Step Properties window shows the following properties:

Parameter	Description	Value
Used	Address used	If selected, this address is in use in a program.
Address	*Grafcet* Step object address	The maximum number of Grafcet Step objects that the application can contain is: • 96, if the **Functional Level** < **10.0**. • 200, if the **Functional Level** >= **10.0**.
Symbol	Symbol	The symbol associated with this object. Refer to the EcoStruxure Machine Expert - Basic Operating Guide, Defining and Using Symbols (see EcoStruxure Machine Expert - Basic, Operating Guide) for details.
Comment	Comment	A comment associated with this object.

PTO Objects

Motion Task Table (%MT)

Overview

Refer to Motion Task Table in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

Pulse Train Output (%PTO)

Overview

Refer to Pulse Train Output (%PTO) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

Drive Objects

Overview

Drive objects control ATV drives and other devices configured on the Modbus Serial IOScanner or Modbus TCP IOScanner. Refer to Drive (%DRV) in the Modicon M221 Logic Controller Advanced Functions, Library Guide.

Communication Objects

Introduction

The communication function blocks are used for communication with Modbus devices and send/receive messages in character mode (ASCII).

NOTE: Only one communication function block can be active at a time on a communication port during a master task cycle. If you attempt to use several communication function blocks or EXCH instructions concurrently on the same communication port, the function blocks return an error code. Therefore, verify that no active exchange (%MSGx.D is TRUE) is in progress on a communication port before starting a communication function block or EXCH instruction.

Read Data from a Remote Device (%READ_VAR)

Using %READ_VAR Function Blocks

This section provides descriptions and programming guidelines for using *%READ_VAR* function blocks.

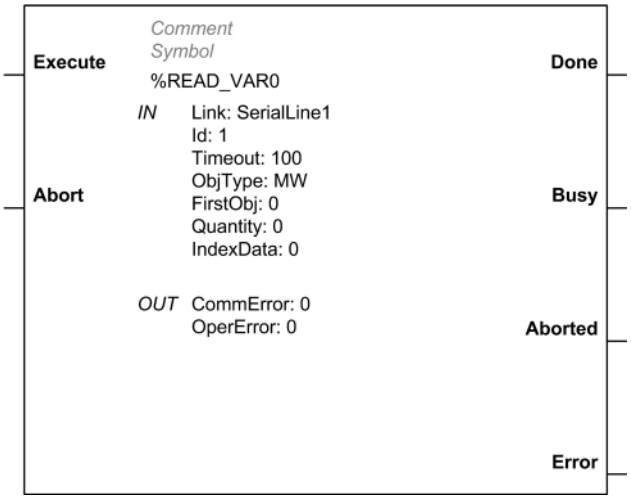
Description

Introduction

The *%READ_VAR* function block is used to read data from a remote device on Modbus SL or Modbus TCP.

Illustration

This illustration is the *%READ_VAR* function block:



Inputs

The *%READ_VAR* function block has the following inputs:

Label	Type	Value
Execute	BOOL	Starts function block execution when a rising edge is detected. If a second rising edge is detected during the execution of the function block, it is ignored and the ongoing command is not affected.
Abort	BOOL	Stops function block execution when a rising edge is detected. The **Aborted** output is set to 1 and the *%READ_VARi.CommError* object contains the code 02 hex (exchange stopped by a user request).

NOTE: Setting **Execute** or **Abort** input to *TRUE* at the first task cycle in RUN is not detected as a rising edge. The function block needs to first see the input as *FALSE* in order to detect a subsequent rising edge.

Outputs

The *%READ_VAR* function block has the following outputs:

Label	Type	Value
Done	BOOL	If *TRUE*, indicates that the function block execution is completed successfully with no detected errors.
Busy	BOOL	If *TRUE*, indicates that the function block execution is in progress.
Aborted	BOOL	If *TRUE*, indicates that the function block execution was canceled with the **Abort** input.
Error	BOOL	If *TRUE*, indicates that an error was detected. Function block execution is stopped. For details on the *CommError* and *OperError*, refer to the tables Communication Error Codes, page 173 and the Operation Error Codes, page 174.

This table describes the output objects of the function block:

Output object	Type	Description
CommError	*BYTE*	For details on *%READ_VARi.CommError*, refer to the table Communication Error Codes, page 173.
OperError	*DWORD*	For details on *%READ_VARi.OperError*, refer to the table Operation Error Codes, page 174.

Communication Error Codes

This table describes the error codes written to the *%READ_VARi.CommError* word object:

Name	Detected error code	Description
`CommunicationOK`	0 (00 hex)	Exchange is correct.
`TimedOut`	1 (01 hex)	Exchange stopped because timeout expired.
`Abort`	2 (02 hex)	Exchange stopped on user request (**Abort** input).
`BadAddress`	3 (03 hex)	Address format is incorrect.
`BadRemoteAddr`	4 (04 hex)	Remote address is incorrect.
`BadMgtTable`	5 (05 hex)	Management table format is incorrect.
`BadParameters`	6 (06 hex)	Specific parameters are incorrect.
`ProblemSendingRq`	7 (07 hex)	Unsuccessful sending request to destination.
`RecvBufferNotAlloc`	9 (09 hex)	Reception buffer size is insufficient.
`SendBufferNotAlloc`	10 (0A hex)	Transmission buffer size is insufficient.
`SystemResourceMiss-ing`	11 (0B hex)	System resource is unavailable.
`BadLength`	14 (0E hex)	Length is incorrect.
`ProtocolSpecificEr-ror`	254 (FE hex)	Indicates a Modbus protocol error. For more details, refer to Operation Error Codes, page 174.
`Refused`	255 (FF hex)	Message is refused. For more details, refer to Operation Error Codes, page 174.

Operation Error Codes

This return code is significant when the communication error code (*CommError* object) has the value:

- 0 (00 hex) (correct)
- 254 (FE hex) (Modbus exception code)
- 255 (FF hex) (refused)

This table describes the error codes written to the *%READ_VARi.OperError* double word object:

CommError	Name	Detected error code	Description
0 (00 hex) (correct)	`OperationOK`	0 (00000000 hex)	Exchange is correct.
	`NotProcessed`	1 (00000001 hex)	Request has not been processed.
	`BadResponse`	2 (00000002 hex)	Received response is incorrect.
254 (FE hex) (Modbus exception code)	`IllegalFunction`	1 (00000001 hex)	The function code received in the request is not an authorized action for the slave. The slave may not be in the correct state to process a specific request.
	`IllegalDataAddress`	2 (00000002 hex)	The data address received by the slave is not an authorized address for the slave.
	`IllegalDataValue`	3 (00000003 hex)	The value in the request data field is not an authorized value for the slave.
	`SlaveDeviceFailure`	4 (00000004 hex)	The slave cannot perform a requested action because of an unrecoverable error.
	`Acknowledge`	5 (00000005 hex)	The slave acknowledged the request but communications timed out before the slave complied.
	`SlaveDeviceBusy`	6 (00000006 hex)	The slave is busy processing another command.
	`MemoryParityError`	8 (00000008 hex)	The slave detects a parity error in the memory when attempting to read extended memory.
	`GatewayPathUnavailable`	10 (0000000A hex)	The gateway is overloaded or not correctly configured.
	`GatewayTargetDeviceFailed-ToRespond`	11 (0000000B hex)	The slave is not present on the network.
255 (FF hex) (refused)	`TargetResourceMissing`	1 (00000001 hex)	Target system resource is unavailable.
	`BadLength`	5 (00000005 hex)	Length is incorrect.
	`CommChannelErr`	6 (00000006 hex)	Error detected on the communication channel.
	`BadAddr`	7 (00000007 hex)	Address is incorrect.
	`SystemResourceMissing`	11 (0000000B hex)	System resource is unavailable.
	`TargetCommInactive`	12 (0000000C hex)	Target communication function is not active.
	`TargetMissing`	13 (0000000D hex)	Target is incommunicative.
	`ChannelNotConfigured`	15 (0000000F hex)	Channel not configured.

Function Configuration

Properties

Double-click the function block to open the function properties table.

The properties of this function block cannot be modified in online mode.

The *%READ_VAR* function block has the following properties:

Property	Value	Description
Used	Activated / deactivated check box	Indicates whether the address is in use.
Address	`%READ_VARi`, where *i* is from 0 to the number of objects available on this logic controller	*i* is the instance identifier. For the maximum number of instances, refer to Maximum Number of Objects (see Modicon M221 Logic Controller, Programming Guide).
Symbol	User-defined text	The symbol uniquely identifies this object. For details, refer to the EcoStruxure Machine Expert - Basic Operating Guide (Defining and Using Symbols) (see EcoStruxure Machine Expert - Basic, Operating Guide).
Link	• **SL1**: Serial 1 • **SL2**: Serial 2 • **ETH1**: Ethernet	Port selection. **NOTE: SL2** and **ETH1** embedded communication ports are available on certain controller references only. See the Modicon M221 Logic Controller, Hardware Guide for more information.
Id	This parameter depends on the link configuration: • 1...247 for serial lines slave address • 1...16 for Ethernet index	Device identifier. For more details about the Ethernet index, refer to Adding Remote Servers (see Modicon M221 Logic Controller, Programming Guide).
Timeout	Specified in units of 100 ms, with a default of 100 (10 seconds). A value of 0 means no timeout enforced.	The timeout sets the maximum time to wait to receive an answer. If the timeout expires, the exchange terminates in error with an error code (`CommError` = 01 hex). If the system receives a response after the timeout expiration, this response is ignored. **NOTE:** The timeout set on the function block overrides the value configured into EcoStruxure Machine Expert - Basic configuration screens (Modbus TCP Configuration and Serial Line Configuration (see Modicon M221 Logic Controller, Programming Guide)).
ObjType	The object type to read: • 0 for words • 1 for digital inputs • 2 for digital outputs • 3 for input words	The types of Modbus read function codes are: • **Mbs 0x03 - Read multiple words (holding register)** • **Mbs 0x02 - Read multiple bits (digital inputs)** • **Mbs 0x01 - Read multiple bits (digital outputs)** • **Mbs 0x04 - Read multiple words (input register)**
FirstObj	0...65535	The address of the first object on the remote device from which values are read.
Quantity	• 1...125 for **%MW** • 1...2000 for **%I** • 1...2000 for **%Q** or **%M** • 1...125 for **%IW**	The number of objects to read from the remote device.
IndexData	0...7999	The address of the local word table (*%MW*) into which the values read are stored. When reading bits (*%I* or *%Q*), the retrieved bits are written into the word table starting at the first address specified. For example, when reading 16 bits with **IndexData** = 10 and **Quantity** = 16, the result is stored in *%MW10:X0* to *%MW10:X15*
Comment	User-defined text	A comment to associate with this object.

Objects

The *%READ_VAR* function block has the following objects:

Object	Description	Value
%READ_VARi.LINK	Port selection	Refer to Properties, page 175. Can be read and written. Can be modified in an animation table.
%READ_VARi.ID	Remote device identifier	Refer to Properties, page 175. Can be read and written. Can be modified in an animation table.
%READ_VARi.TIMEOUT	Function block timeout	Refer to Properties, page 175. Can be read and written. Can be modified in an animation table.
%READ_VARi.OBJTYPE	Type of objects to read	Refer to Properties, page 175. Can be read and written. Can be modified in an animation table.
%READ_VARi.FIRSTOBJ	The address of the first object on the remote device from which values are read.	Refer to Properties, page 175. Can be read and written. Can be modified in an animation table.
%READ_VARi.QUANTITY	The number of objects to read from the remote device.	Refer to Properties, page 175. Can be read and written. Can be modified in an animation table.
%READ_VARi.INDEXDATA	The address of the local word table (*%MW*) into which the values read are stored.	Refer to Properties, page 175. Can be read and written. Can be modified in an animation table.
%READ_VARi.COMMERROR	Communication Error Codes	Refer to Communication Error Codes, page 173. Read only. Can be read in an animation table.
%READ_VARi.OPERERROR	Operation Error Codes	Refer to Operation Error Codes, page 174. Read only. Can be read in an animation table.
%READ_VARi.DONE	Execution completed successfully	Refer to Outputs, page 172. Read only. Can be read in an animation table.
%READ_VARi.BUSY	Execution is in progress	Refer to Outputs, page 172. Read only. Can be read in an animation table.
%READ_VARi.ABORTED	Execution was canceled	Refer to Outputs, page 172. Read only. Can be read in an animation table.
%READ_VARi.ERROR	An error was detected	Refer to Outputs, page 172. Read only. Can be read in an animation table.

Programming Example

Introduction

The *%READ_VAR* function block can be configured as presented in this programming example.

Programming

This example is a *%READ_VAR* function block:

Rung	Instruction
0	BLK %READ_VAR0 LD %I0.0 EXECUTE LD %I0.1 ABORT OUT_BLK LD DONE ST %Q0.0 LD BUSY ST %Q0.1 LD ABORTED ST %M1 LD ERROR ST %Q0.2 END_BLK

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

Timing Diagrams for Communication Function Blocks, page 204.

Write Data to a Modbus Device (%WRITE_VAR)

Using %WRITE_VAR Function Blocks

This section provides descriptions and programming guidelines for using *%WRITE_VAR* function blocks.

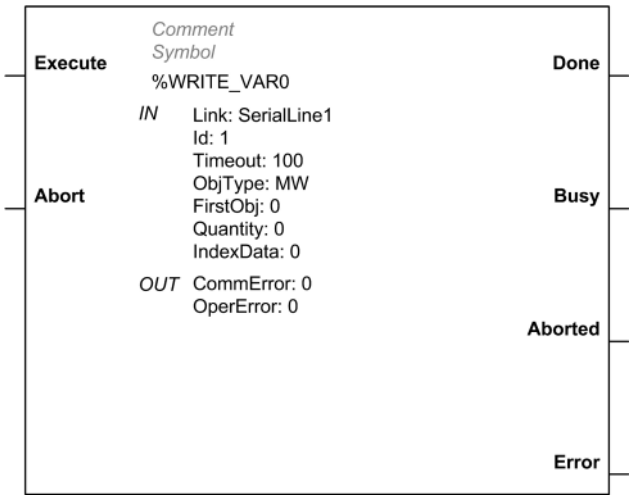
Description

Introduction

The *%WRITE_VAR* function block is used to write data to an external device using the Modbus SL or Modbus TCP protocol.

Illustration

This illustration is the *%WRITE_VAR* function block:



Inputs

The *%WRITE_VAR* function block has the following inputs:

Label	Type	Value
Execute	BOOL	Starts function block execution when a rising edge is detected. If a second rising edge is detected during the execution of the function block, it is ignored and the ongoing command is not affected.
Abort	BOOL	Stops function block execution when a rising edge is detected. The **Aborted** output is set to 1 and the *%WRITE_VARi.CommError* object contains the code 02 hex (exchange stopped by a user request).

NOTE: Setting **Execute** or **Abort** input to *TRUE* at the first task cycle in RUN is not detected as a rising edge. The function block needs to first see the input as *FALSE* in order to detect a subsequent rising edge.

Outputs

The *%WRITE_VAR* function block has the following outputs:

Label	Type	Value
Done	BOOL	If *TRUE*, indicates that the function block execution is completed successfully with no detected errors.
Busy	BOOL	If *TRUE*, indicates that the function block execution is in progress.
Aborted	BOOL	If *TRUE*, indicates that the function block execution was canceled with the **Abort** input.
Error	BOOL	If *TRUE*, indicates that an error was detected. Function block execution is stopped. For details on the *CommError* and *OperError*, refer to the tables Communication Error Codes, page 173 and the Operation Error Codes, page 174 .

This table describes the output objects of the function block:

Output object	Type	Description
CommError	*BYTE*	For details on *%READ_VARi.CommError*, refer to the table Communication Error Codes, page 173.
OperError	*DWORD*	For details on *%READ_VARi.OperError*, refer to the table Operation Error Codes, page 174.

Communication Error Codes

Refer to Communication Error Codes, page 173.

Operation Error Codes

Refer to Operation Error Codes, page 174.

Function Configuration

Properties

Double-click on the function block to open the function properties table.

The properties of this function block cannot be modified in online mode.

The *%WRITE_VAR* function block has the following properties:

Property	Value	Description
Used	Activated / deactivated checkbox	Indicates whether the address is in use.
Address	`%WRITE_VARi`, where *i* is from 0 to the number of objects available on this logic controller	*i* is the instance identifier. For the maximum number of instances, refer to Maximum Number of Objects table (see Modicon M221 Logic Controller, Programming Guide).
Symbol	User-defined text	The symbol uniquely identifies this object. For details, refer to the EcoStruxure Machine Expert - Basic Operating Guide (Defining and Using Symbols) (see EcoStruxure Machine Expert - Basic, Operating Guide).
Link	• **SL1**: Serial 1 • **SL2**: Serial 2 • **ETH1**: Ethernet	Port selection. **NOTE: SL2** and **ETH1** embedded communication ports are available on certain controller references only. See the Modicon M221 Logic Controller, Hardware Guide for more information.
Id	This parameter depends on the link configuration: • 0 for broadcast • 1...247 for serial lines slave address • 1...16 for Ethernet index	Device identifier. For value 0, the Modbus master controller initiates a broadcast to all connected slaves. In broadcast mode, the slaves do not answer the master. For more details about the Ethernet index, refer to Adding Remote Servers (see Modicon M221 Logic Controller, Programming Guide).
Timeout	Specified in units of 100 ms, with a default of 100 (10 seconds). A value of 0 means no timeout enforced.	The timeout sets the maximum time to wait to receive an answer. If the timeout expires, the exchange terminates in error with an error code (`CommError` = 01 hex). If the system receives a response after the timeout expiration, this response is ignored. **NOTE:** The timeout set on the function block overrides the value configured into EcoStruxure Machine Expert - Basic configuration screens (Modbus TCP Configuration and Serial Line Configuration (see Modicon M221 Logic Controller, Programming Guide)).
ObjType	The following Modbus function codes are supported: • 0 • 2 • 4 • 5	The type of objects to write: • **Mbs 0x10 - Write multiple words (register)** • **Mbs 0x0F - Write multiple bits (digital outputs)** • **Mbs 0x05 - Write single bit (digital output)** • **Mbs 0x06 - Write single word (register)** **NOTE:** The application must be configured with a functional level of at least 5.0 to use the Single Coil (Mbs 0x05) or Single Register (Mbs 0x06) Modbus function codes.
FirstObj	0...65535	The address of the first object on the remote device to which values are written.
Quantity	• 1...123 (internal register) for **%MW** • 1...1968 (internal bit) for **%M** or **%Q**	The number of objects to write to the remote device. Ignored for single coil and single register object types.
IndexData	0...7999	The address of the local word table (*%MW*) containing the values to be written to the remote device. This memory word table (*%MW*) is used for all supported function codes. When writing bits (*%M* or *%Q*), the values to be written are retrieved from the word table starting at the first address specified. For example, when writing 16 bits with **IndexData** = 10 and **Quantity** = 16, the values are retrieved from *%MW10:X0* to *%MW10:X15*
Comment	User-defined text	A comment to associate with this object.

Objects

The *%WRITE_VAR* function block has the following objects:

Object	Description	Value
%WRITE_VARi.LINK	Port selection	Refer to Properties, page 179. Can be read and written. Can be modified in an animation table.
%WRITE_VARi.ID	Remote device identifier	Refer to Properties, page 179. Can be read and written. Can be modified in an animation table.
%WRITE_VARi.TIMEOUT	Function block timeout	Refer to Properties, page 179. Can be read and written. Can be modified in an animation table.
%WRITE_VARi.OBJTYPE	Type of objects to write	Refer to Properties, page 179. Can be read and written. Can be modified in an animation table.
%WRITE_VARi.FIRSTOBJ	The address of the first object on the remote device to which values are written.	Refer to Properties, page 179. Can be read and written. Can be modified in an animation table.
%WRITE_VARi.QUANTITY	The number of objects to write to the remote device.	Refer to Properties, page 179. Can be read and written. Can be modified in an animation table.
%WRITE_VARi.INDEXDATA	The address of the local word table (*%MW*) containing the values to be written to the remote device.	Refer to Properties, page 179. Can be read and written. Can be modified in an animation table.
%WRITE_VARi.COMMERROR	Communication Error Codes	Refer to Communication Error Codes, page 179. Read only. Can be read in an animation table.
%WRITE_VARi.OPERERROR	Operation Error Codes	Refer to Operation Error Codes, page 179. Read only. Can be read in an animation table.
%WRITE_VARi.DONE	Execution completed successfully	Refer to Outputs, page 178. Read only. Can be read in an animation table.
%WRITE_VARi.BUSY	Execution is in progress	Refer to Outputs, page 178. Read only. Can be read in an animation table.
%WRITE_VARi.ABORTED	Execution was canceled	Refer to Outputs, page 178. Read only. Can be read in an animation table.
%WRITE_VARi.ERROR	An error was detected	Refer to Outputs, page 178. Read only. Can be read in an animation table.

Programming Example

Introduction

The *%WRITE_VAR* function block can be configured as presented in this programming example.

Programming

This example is a *%WRITE_VAR* function block:

Rung	Instruction
0	BLK %WRITE_VAR0 LD %I0.0 EXECUTE LD %I0.1 ABORT OUT_BLK LD DONE ST %Q0.0 LD BUSY ST %Q0.1 LD ABORTED ST %M1 LD ERROR ST %Q0.2 END_BLK

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

Timing Diagrams for Communication Function Blocks, page 204.

Read and Write Data on a Modbus Device (%WRITE_READ_VAR)

Using %WRITE_READ_VAR Function Blocks

This section provides descriptions and programming guidelines for using *%WRITE_READ_VAR* function blocks.

Description

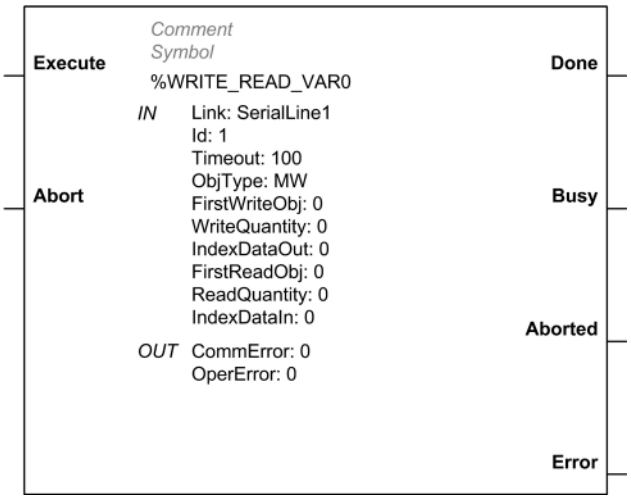
Introduction

The *%WRITE_READ_VAR* function block is used to read and write data stored in internal memory words to an external device using the Modbus SL or Modbus TCP protocol.

This function block performs a single write request followed by a single read request in the same transaction.

Illustration

This illustration is the *%WRITE_READ_VAR* function block:



Inputs

The *%WRITE_READ_VAR* function block has the following inputs:

Label	Type	Value
Execute	BOOL	Starts function block execution when a rising edge is detected. If a second rising edge is detected during the execution of the function block, it is ignored and the ongoing command is not affected.
Abort	BOOL	Stops function block execution when a rising edge is detected. The **Aborted** output is set to 1 and the *%WRITE_READ_VARi.CommError* object contains the code 02 hex (exchange stopped by a user request).

NOTE: Setting **Execute** or **Abort** input to *TRUE* at the first task cycle in RUN is not detected as a rising edge. The function block needs to first see the input as *FALSE* in order to detect a subsequent rising edge.

Outputs

The *%WRITE_READ_VAR* function block has the following outputs:

Label	Type	Value
Done	BOOL	If *TRUE*, indicates that the function block execution is completed successfully with no detected errors.
Busy	BOOL	If *TRUE*, indicates that the function block execution is in progress.
Aborted	BOOL	If *TRUE*, indicates that the function block execution was canceled with the **Abort** input.
Error	BOOL	If *TRUE*, indicates that an error was detected. Function block execution is stopped. For details on the *CommError* and *OperError*, refer to the tables Communication Error Codes, page 173 and the Operation Error Codes, page 174.

This table describes the output objects of the function block:

Output object	Type	Description
CommError	*BYTE*	For details on *%READ_VARi.CommError*, refer to the table Communication Error Codes, page 173.
OperError	*DWORD*	For details on *%READ_VARi.OperError*, refer to the table Operation Error Codes, page 174.

Communication Error Codes

Refer to Communication Error Codes, page 173.

Operation Error Codes

Refer to Operation Error Codes, page 174.

Function Configuration

Properties

Double-click on the function block to open the function properties table.

The properties of this function block cannot be modified in online mode.

The *%WRITE_READ_VAR* function block has the following properties:

Property	Value	Description
Used	Activated / deactivated checkbox	Indicates whether the address is in use.
Address	`%WRITE_READ_VARi`, where *i* is from 0 to the number of objects available on this logic controller	*i* is the instance identifier. For the maximum number of instances, refer to Maximum Number of Objects table (see Modicon M221 Logic Controller, Programming Guide).
Symbol	User-defined text	The symbol uniquely identifies this object. For details, refer to the EcoStruxure Machine Expert - Basic Operating Guide (Defining and Using Symbols) (see EcoStruxure Machine Expert - Basic, Operating Guide).
Link	• **SL1**: Serial 1 • **SL2**: Serial 2 • **ETH1**: Ethernet	Port selection **NOTE: SL2** and **ETH1** embedded communication ports are available on certain controller references only.
Id	This parameter depends on the link configuration: • 1...247 for serial lines slave address • 1...16 for Ethernet index	Device identifier For more details about the Ethernet index, refer to Adding Remote Servers (see Modicon M221 Logic Controller, Programming Guide).
Timeout	Specified in units of 100 ms, with a default of 100 (10 seconds). A value of 0 means no timeout enforced.	The timeout sets the maximum time to wait to receive an answer. If the timeout expires, the exchange terminates in error with an error code (`CommError` = 01 hex). If the system receives a response after the timeout expiration, this response is ignored. **NOTE:** The timeout set on the function block overrides the value configured into EcoStruxure Machine Expert - Basic configuration screens (Modbus TCP Configuration (see Modicon M221 Logic Controller, Programming Guide) and Serial Line Configuration (see Modicon M221 Logic Controller, Programming Guide)).
ObjType	**%MW (Mbs Fct 17)**: memory words	The type of Modbus read/write function code is **Mbs Fct 17**, which is equivalent to Modbus function code 17.
FirstWriteObj	0...65535	The address of the first object on the remote device to which values are written.
WriteQuantity	1...121	The number of objects to write to the remote device.
IndexDataOut	0...7999	The address of the local word table (*%MW*) containing values to be written to the remote device.
FirstReadObj	0...65535	The address of the first object on the remote device from which values are read.
ReadQuantity	1...125	The number of objects to read from the remote device.
IndexDataIn	0...7999	The address of the local word table (*%MW*) into which the values read are stored.
Comment	User-defined text	A comment to associate with this object.

Objects

The *%WRITE_READ_VAR* function block has the following objects:

Object	Description	Value
%WRITE_READ_VARi.LINK	Port selection	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.ID	Remote device identifier	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.TIMEOUT	Function block timeout	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.OBJTYPE	Type of objects to read	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.FIRSTWRITEOBJ	The address of the first object on the remote device to which values are written.	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.WRITEQUANTITY	The number of objects to write to the remote device.	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.INDEXDATAOUT	The address of the local word table (*%MW*) containing values to be written to the remote device.	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.FIRSTREADOBJ	The address of the first object on the remote device from which values are read.	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.READQUANTITY	The number of objects to read from the remote device.	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.INDEXDATAIN	The address of the local word table (*%MW*) into which the values read are stored.	Refer to Properties, page 184. Can be read and written. Can be modified in an animation table.
%WRITE_READ_VARi.COMMERROR	Communication Error Codes	Refer to Communication Error Codes, page 184. Read only. Can be read in an animation table.
%WRITE_READ_VARi.OPERERROR	Operation Error Codes	Refer to Operation Error Codes, page 184. Read only. Can be read in an animation table.
%WRITE_READ_VARi.DONE	Execution completed successfully	Refer to Outputs, page 183. Read only. Can be read in an animation table.
%WRITE_READ_VARi.BUSY	Execution is in progress	Refer to Outputs, page 183. Read only. Can be read in an animation table.
%WRITE_READ_VARi.ABORTED	Execution was canceled	Refer to Outputs, page 183. Read only. Can be read in an animation table.
%WRITE_READ_VARi.ERROR	An error was detected	Refer to Outputs, page 183. Read only. Can be read in an animation table.

Programming Example

Introduction

The *%WRITE_READ_VAR* function block can be configured as presented in this programming example.

Programming

This example is a *%WRITE_READ_VAR* function block:

Rung	Instruction
0	BLK %WRITE_READ_VAR0 LD %I0.0 EXECUTE LD %I0.1 ABORT OUT_BLK LD DONE ST %Q0.0 LD BUSY ST %Q0.1 LD ABORTED ST %M1 LD ERROR ST %Q0.2 END_BLK

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

Timing Diagrams for Communication Function Blocks, page 204.

Communication on an ASCII Link (%SEND_RECV_MSG)

Using %SEND_RECV_MSG Function Blocks

This section provides descriptions and programming guidelines for using *%SEND_RECV_MSG* function blocks.

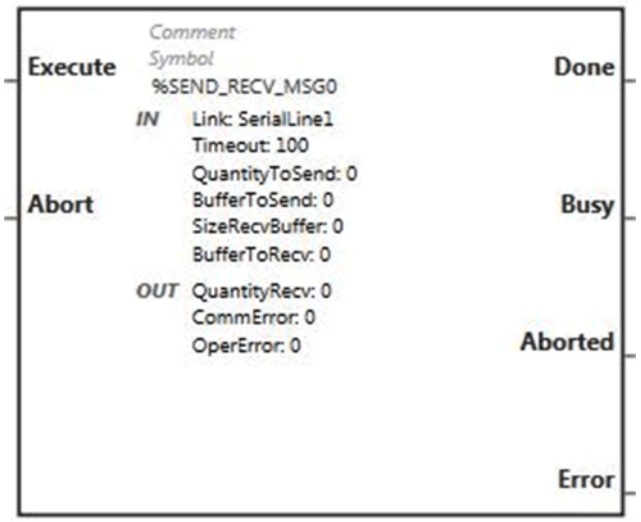
Description

Introduction

The *%SEND_RECV_MSG* function block is used to send or receive data on a serial line configured for the ASCII protocol.

Illustration

This illustration is the *%SEND_RECV_MSG* function block:



Inputs

The *%SEND_RECV_MSG* function block has the following inputs:

Label	Type	Value
Execute	BOOL	Starts function block execution when a rising edge is detected. If a second rising edge is detected during the execution of the function block, it is ignored and the ongoing command is not affected.
Abort	BOOL	Stops function block execution when a rising edge is detected. The **Aborted** output is set to 1 and the *%SEND_RECV_MSGi.CommError* object contains the code 02 hex (exchange stopped by a user request).

NOTE: Setting **Execute** or **Abort** input to *TRUE* at the first task cycle in RUN is not detected as a rising edge. The function block needs to first see the input as *FALSE* in order to detect a subsequent rising edge.

Outputs

The *%SEND_RECV_MSG* function block has the following outputs:

Label	Type	Value
Done	BOOL	If *TRUE*, indicates that the function block execution is completed successfully with no detected errors.
Busy	BOOL	If *TRUE*, indicates that the function block execution is in progress.
Aborted	BOOL	If *TRUE*, indicates that the function block execution was canceled with the **Abort** input.
Error	BOOL	If *TRUE*, indicates that an error was detected. Function block execution is stopped. For details on the *CommError* and *OperError*, refer to the tables Communication Error Codes, page 173 and the Operation Error Codes, page 174.

Communication Error Codes

Refer to Communication Error Codes, page 173.

Operation Error Codes

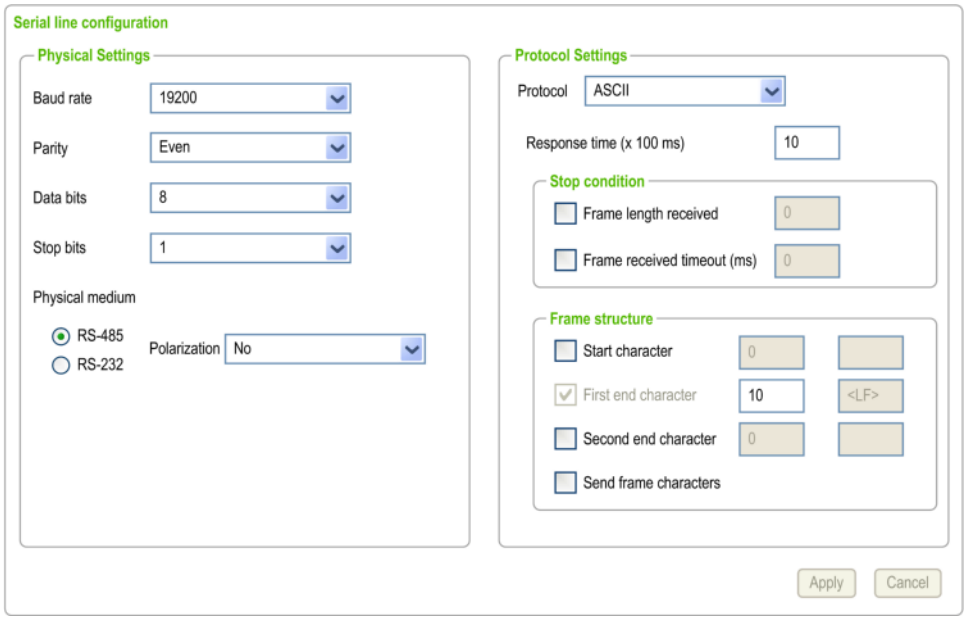
Refer to Operation Error Codes, page 174.

End Conditions

For a send-only operation, the **Done** output is set to *TRUE* when all data (including any start/stop characters) have been sent.

For a receive-only operation, the system receives characters until the end condition is *TRUE*. When the end condition is reached, the **Done** output is set to *TRUE*. Received characters are then copied into **BufferToRecv**, up to **sizeRecvBuffer** characters. **sizeRecvBuffer** is not an end condition.

The end condition must be set in the Serial line configuration screen (see Modicon M221 Logic Controller, Programming Guide):



The end condition can be set to:

- A number of bytes received: **Frame length received**
- An end of frame silence: **Frame received timeout (ms)**
- A frame structure: **First end character**

For a send-receive operation, characters are first sent to the line, then characters are received until the end condition is *TRUE* (same as receive-only).

Function Configuration

Properties

Double-click the function block to open the function properties table.

The properties of this function block cannot be modified in online mode.

The *%SEND_RECV_MSG* function block has the following properties:

Property	Value	Description
Used	Activated / deactivated check box	Indicates whether the address is in use.
Address	`%SEND_RECV_MSGi`, where *i* is from 0 to the number of objects available on this logic controller	*i* is the instance identifier. For the maximum number of instances, refer to the Programming Guide for the logic controller.
Symbol	User-defined text	The symbol uniquely identifies this object. For details, refer to the EcoStruxure Machine Expert - Basic Operating Guide (Defining and Using Symbols) (see EcoStruxure Machine Expert - Basic, Operating Guide).
Link	• **SL1**: Serial 1 • **SL2**: Serial 2	Port selection **NOTE: SL2** embedded communication port is available on certain controller references only.
Timeout	Specified in units of 100 ms, with a default of 100 (10 seconds). A value of 0 means no timeout enforced.	The timeout sets the maximum time to wait to receive an answer. If the timeout expires, the exchange terminates in error with an error code (`CommError` = 01 hex). If the system receives a response after the timeout expiration, this response is ignored. **NOTE:** The timeout set on the function block overrides the value configured into EcoStruxure Machine Expert - Basic configuration screens (*Modbus TCP Configuration* and *Serial Line Configuration*, refer to the Programming Guide for the logic controller.).
QuantityToSend	0...255 A value of 0 means that the function block only receives data.	Number of bytes to send
BufferToSend	0...7999	Address of the first object to send
SizeRecvBuffer	0...255 A value of 0 means that the function block only sends data.	Available size in bytes of the receive buffer.
BufferToRecv	0...7999	The first address of the word table to which read values are stored (*%MW*).
QuantityRecv	0...255	Quantity of received data in bytes
Comment	User-defined text	A comment to associate with this object.

Objects

The *%SEND_RECV_MSG* function block has the following objects:

Object	Description	Value
%SEND_RECV_MSGi.LINK	Port selection	Refer to Properties, page 189. Can be read and written. Can be modified in an animation table.
%SEND_RECV_MSGi.TIMEOUT	Function block timeout	Refer to Properties, page 189. Can be read and written. Can be modified in an animation table.
%SEND_RECV_MSGi. QUANTITYTOSEND	Number of bytes to send	Refer to Properties, page 189. Can be read and written. Can be modified in an animation table.
%SEND_RECV_MSGi.BUFFERTOSEND	Address of the first object to send	Refer to Properties, page 189. Can be read and written. Can be modified in an animation table.
%SEND_RECV_MSGi.SIZERECVBUFFER	Available size in bytes of the receive buffer	Refer to Properties, page 189. Can be read and written. Can be modified in an animation table.
%SEND_RECV_MSGi.BUFFERTORECV	First address of the word table to which read values are to be stored	Refer to Properties, page 189. Can be read and written. Can be modified in an animation table.
%SEND_RECV_MSGi.QUANTITYRECV	Quantity of received data in bytes	Refer to Properties, page 189. Can be only read. Can be modified in an animation table.
%SEND_RECV_MSGi.COMMERROR	Communication Error Codes	Refer to Communication Error Codes, page 189. Read only. Can be read in an animation table.
%SEND_RECV_MSGi.OPERERROR	Operation Error Codes	Refer to Operation Error Codes, page 189. Read only. Can be read in an animation table.
%SEND_RECV_MSGi.DONE	Execution completed successfully	Refer to Outputs, page 188. Read only. Can be read in an animation table.
%SEND_RECV_MSGi.BUSY	Execution is in progress	Refer to Outputs, page 188. Read only. Can be read in an animation table.
%SEND_RECV_MSGi.ABORTED	Execution was canceled	Refer to Outputs, page 188. Read only. Can be read in an animation table.
%SEND_RECV_MSGi.ERROR	An error was detected	Refer to Outputs, page 188. Read only. Can be read in an animation table.

Programming Example

Introduction

The *%SEND_RECV_MSG* function block can be configured as presented in this programming example.

Programming

This example is a *%SEND_RECV_MSG* function block:

Rung	Instruction
0	BLK %SEND_RECV_MSG0 LD %I0.0 EXECUTE LD %I0.1 ABORT OUT_BLK LD DONE ST %Q0.0 LD BUSY ST %Q0.1 LD ABORTED ST %M1 LD ERROR ST %Q0.2 END_BLK

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Timing Diagram

Timing Diagrams for Communication Function Blocks, page 204.

Send Receive SMS (%SEND_RECV_SMS)

Using %SEND_RECV_SMS Function Block

This section provides description and programming guidelines for using *%SEND_RECV_SMS* function block.

Description

Introduction

The *%SEND_RECV_SMS* function block is used to send and receive Short Message Service (SMS) messages through a modem connected to a serial line. For example, the controller can send an SMS to transmit an alarm to a specified mobile phone, or receive an SMS to terminate a function of the machine.

NOTE: The application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 3.2** to use the SMS functionality.

The *%SEND_RECV_SMS* function block is used to either:

- send an SMS to one recipient only, or
- receive an SMS filtered by a table of approved phone numbers.

Only 1 *%SEND_RECV_SMS* function block can be used in a program.

Care must be taken and provisions made for use of the SMS functionality as a remote control device to avoid inadvertent consequences of commanded machine operation, controller state changes, or alteration of data memory or machine operating parameters.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Ensure that there is a local, competent, and qualified observer present when operating from a remote location.
- Configure and install the Run/Stop input for the application so that local control over the starting or stopping of the controller can be maintained regardless of the remote commands sent to the controller.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

The SMS functionality is dependent on an external telecommunication network and parameters. SMS commands and messages transmitted to the machine may be delayed or not sent nor received. Do not use the SMS functionality for safety critical functions or other critical purposes.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

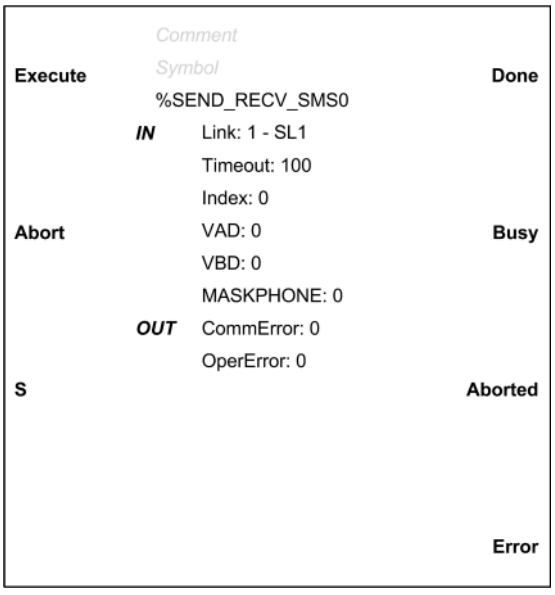
- Do not allow safety critical functions in SMS commands.
- Do not use SMS commands or messages for any mission critical purposes.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

NOTE: Verify the SMS functionality and associated telecommunication network during commissioning, and test it periodically to verify the network coverage.

Illustration

This illustration is the *%SEND_RECV_SMS* function block:



Inputs

This table describes the inputs of the function block:

Label	Type	Value
Execute	BOOL	Starts the function block execution when a rising edge is detected. If a second rising edge is detected during the execution of the function block, it is ignored and the ongoing command is not affected.
Abort	BOOL	Stops the function block execution when a rising edge is detected. The **Aborted** output is set to 1 and the *%SEND_RECV_SMSi.CommError* object contains the code *02 hex* (exchange stopped on a rising edge on the **Abort** input).
S	BOOL	If *1*, the function block is configured to send an SMS. If *0*, the function block is configured to receive an SMS. The value change is taken into account when the function block is aborted.

NOTE: Setting **Execute** or **Abort** input to *1* at the first task cycle in RUN is not detected as a rising edge. The function block needs to first see the input as *0* in order to detect a subsequent rising edge.

This table describes the input objects of the function block:

Input object	Type	Value range	Description
Link	*BYTE*	1 - SL1 2 - SL2	Indicates the serial line used to communicate through the modem.
Timeout	*WORD*	0...255	Sets the maximum time to wait to receive a response from the modem. Specified in units of 100 ms, with a default of 100 (10 seconds). A value of 0 means no timeout is enforced.
Index	*WORD*	0...15	The value of the index is used: • While sending, to select a text to send from the **Messages** table. 0 corresponds to the first string in the table. • While receiving, to select a string in the **Commands** table that matches the received text. Set to FFFF hex if no matching string is found.
VAD	*DINT*	-2147483648... 2147483647	• While sending, the value in *%SEND_RECV_SMSi.VAD* replaces the placeholder *$VAD* in the text of the SMS. • While receiving, the value in *%SEND_RECV_SMSi.VAD* contains the value where the placeholder *$VAD* is inserted in the SMS stored in the **Commands** table.
VBD	*DINT*	-2147483648... 2147483647	• While sending, the value in *%SEND_RECV_SMSi.VBD* replaces the placeholder *$VBD* in the text of the SMS. • While receiving, the value in *%SEND_RECV_SMSi.VBD* contains the value where the placeholder *$VBD* is inserted in the SMS stored in the **Commands** table.
MASKPHONE	*WORD*	0...15	• While sending, the mask is used to select the recipient of the SMS from the **Phone numbers** table. • While receiving, the mask is applied to the **Phone numbers** table to create a list of valid numbers.

Outputs

This table describes the outputs of the function block:

Label	Type	Value
Done	BOOL	If *TRUE*, indicates that the function block execution completed successfully with no errors detected.
Busy	BOOL	If *TRUE*, indicates that the function block execution is in progress.
Aborted	BOOL	If *TRUE*, indicates that the function block execution was canceled with the *%SEND_RECV_SMSi.Abort* input.
Error	BOOL	If *TRUE*, indicates that an error has been detected. The function block execution is stopped. For details on *%SEND_RECV_SMSi.CommError* and *%SEND_RECV_SMSi.OperError*, refer to the tables Communication Error Codes, page 196 and Operation Error Codes, page 196.

NOTE: When the **Busy** output is set to *TRUE*, the execution continues until one of the **Done**, **Aborted**, or **Error** outputs is set to *TRUE*.

NOTE: While the **Busy** output is set to *TRUE*, changes in the **Execute** input do not affect the execution of the ongoing function block. However, if another *%SEND_RECV_SMS* function block is called, it terminates with an error (*CommError = 255 (FF hex)* and *OperError = 11 (0B hex)*).

This table describes the output objects of the function block:

Output object	Type	Description
CommError	*BYTE*	For details on *%SEND_RECV_SMSi.CommError*, refer to the table Communication Error Codes, page 196.
OperError	*DWORD*	For details on *%SEND_RECV_SMSi.OperError*, refer to the table Operation Error Codes, page 196.

Communication Error Codes

This table describes the error codes written to the *%SEND_RECV_SMSi.CommError* output object:

Decimal (hexadecimal) detected error code	Name	Description
0 (00 hex)	`CommunicationOK`	Exchange is correct. **NOTE:** In this case, the *%SEND_RECV_SMSi.OperError* output object contains the modem signal level, as opposed to an error code.
1 (01 hex)	`TimedOut`	Exchange stopped because timeout expired.
2 (02 hex)	`Abort`	Exchange stopped on a rising edge on the *%SEND_RECV_SMSi.Abort* input.
3 (03 hex)	`BadLink`	Link is incorrect.
4 (04 hex)	`BadCommand`	Command is incorrect.
5 (05 hex)	`BadMgtTable`	Management table format is incorrect.
6 (06 hex)	`BadParameters`	Specific parameters are incorrect.
7 (07 hex)	`ProblemSendingSms`	SMS send command unsuccessful.
9 (09 hex)	`RecvCmdError`	Invalid command.
10 (0A hex)	`SendValueError`	Invalid value.
11 (0B hex)	`SystemResourceMissing`	System resource is unavailable.
14 (0E hex)	`BadLength`	Length is incorrect.
254 (FE hex)	`ProtocolSpecificError`	Indicates that a protocol error has been detected. **NOTE:** In this case, the *%SEND_RECV_SMSi.OperError* output object contains more details. Refer to Operation Error Codes., page 196
255 (FF hex)	`Refused`	SMS is refused. **NOTE:** In this case, the *%SEND_RECV_SMSi.OperError* output object contains more details. Refer to Operation Error Codes., page 196

Operation Error Codes

This return code is significant when the communication error code (*%SEND_RECV_SMSi.CommError* output object) has the value:

- *0 (00 hex)* (correct protocol)
- *254 (FE hex)* (incorrect protocol)
- *255 (FF hex)* (SMS refused)

When the *%SEND_RECV_SMSi.CommError* is *0 (00 hex)* (correct protocol), the *%SEND_RECV_SMSi.OperError* output object indicates the Received Signal Strength Indication (RSSI):

Decimal value in the *%SEND_RECV_SMSi.OperError* object	RSSI modem signal level
Less than 9	Marginal value (the attenuation exceeds the limit needed to keep the wireless network up)
10 to 14	Ok
15 to 19	Good
Greater than 20	Excellent

When the *%SEND_RECV_SMSi.CommError* is *254 (FE hex)* (incorrect protocol), the *%SEND_RECV_SMSi.OperError* output object returns more details:

Decimal (hexadecimal) value in the *%SEND_RECV_SMSi.OperError* object	**Name**	**Description**
256 (00000100 hex)	*ModemConfSLAsciiFailed*	The ASCII configuration of the serial line is incorrect.
512 (00000200 hex)	*ModemReconfSLFailed*	The configuration of the serial line back to the user configuration is incorrect.
768 (00000300 hex)	*ModemBusy*	The modem answers *BUSY* to the dial command.
1024 (00000400 hex)	*ModemNoDialtone*	The modem answers *NODIALTONE* to the dial command.
1280 (00000500 hex)	*ModemNoCarrier*	The modem carrier signal has been lost or disconnected. The modem answers *NO CARRIER* to the dial command.
1536 (00000600 hex)	*ModemBadAnswer*	The response from the modem is incorrect.
Specific errors for SIM card use		
4096 (00001000 hex)	*SimConfigurationFailed*	The SIM card configuration is incorrect. For example, a PUK code is requested.
8192 (00002000 hex)	*SimPinCodeInvalid*	The PIN code is incorrect.
16384 (00004000 hex)	*SimSmsCenterInvalid*	The SMS center phone number is incorrect.

When the *%SEND_RECV_SMSi.CommError* is *255 (FF hex)* (SMS refused), the *%SEND_RECV_SMSi.OperError* output object returns more details:

Decimal (hexadecimal) value in the *%SEND_RECV_SMSi.OperError* object	**Name**	**Description**
1 (00000001 hex)	*TargetResourceMissing*	The target system resource is unavailable.
5 (00000005 hex)	*BadLength*	The length is incorrect.
6 (00000006 hex)	*CommChannelErr*	An error has been detected on the communication channel.
11 (0000000B hex)	*SystemResourceMissing*	The system resource is unavailable.
12 (0000000C hex)	*TargetCommInactive*	The target communication function is not active.
13 (0000000D hex)	*TargetMissing*	The target is unavailable.
15 (0000000F hex)	*ChannelNotConfigured*	The communication channel is not configured.
16 (00000010 hex)	*PhoneNumberNotMatching*	The phone number in the received message does not match with the list of approved numbers (whitelist).
17 (00000011 hex)	*MessageNotMatching*	The received message does not match with any message in the command list. Issued only if the sender phone number matches an entry in the list of approved numbers (whitelist).

Function Configuration

Main Steps

The following procedure describes the main steps to configure the *%SEND_RECV_SMS* function block, after connecting a modem to the serial line:

Step	Action
1	In the **Configuration** tab in EcoStruxure Machine Expert - Basic, configure the serial line with the modem, the **Init Command** and ASCII protocol. For more details, refer to the Programming Guide for your logic controller.
2	Verify that the modem is connected to the serial line of the controller and that: • the SIM card is unlocked, that is, not protected by a PIN code. • the SMS center phone number is correctly configured in the SIM card.
3	In the **Programming** tab: • Add the *%SEND_RECV_SMS* function block and double-click the function block to display the function properties table. • Click the **SMS Configuration** button to open the **SMS Assistant** window. • Edit the **Messages**, **Commands** and **Phone numbers** tables. For more details, refer to **SMS Assistant**, page 198. • Click **Apply** to close the **SMS Assistant**. **NOTE:** If the function block address (for example *%SEND_RECV_SMS0*) is invalid and double-clicking is disabled, verify that the functional level of your application (**Programming** tab > **Tasks** > **Behavior**) is at least **Level 3.2**).
4	In the **Programming** tab, edit the fields in the function properties table. For details about these fields, refer to Properties, page 201.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Verify that the indices for the messages, commands and phone numbers used in the function block are valid (those that you intend to use) before using the function block.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

For more details on the installation and setting of SR2MOD03 modems, refer to *SR2MOD02 and SR2MOD03 Wireless Modem User Guide* (EIO00000001575).

SMS Assistant

To use the *%SEND_RECV_SMS* function block, configure the commands, messages, and phone numbers tables.

Click the **SMS Configuration** button in the **Send Receive SMS properties** area to display the **SMS Assistant** window.

The **SMS Assistant** window contains three tabs with tables to configure:

- **Messages**

  Enter the strings that are used when the controller sends an SMS. Use placeholders to include variables, date, and time. Consider limits on the number of characters and the format.

- **Commands**

  Enter the strings that are used when the controller receives an SMS. Use placeholders to include variables. Consider the limits on the number of characters and the format.

- **Phone numbers**

  When programming the function block to send SMS messages, you select the recipient from this table.

  When programming the function block to receive SMS messages, you select the authorized originating phone numbers from this list. When a call and subsequent SMS is transmitted to your application via the modem, the originating phone number is validated before acting upon the incoming SMS.

  For more details, refer to the **MASKPHONE** line in Properties, page 201.

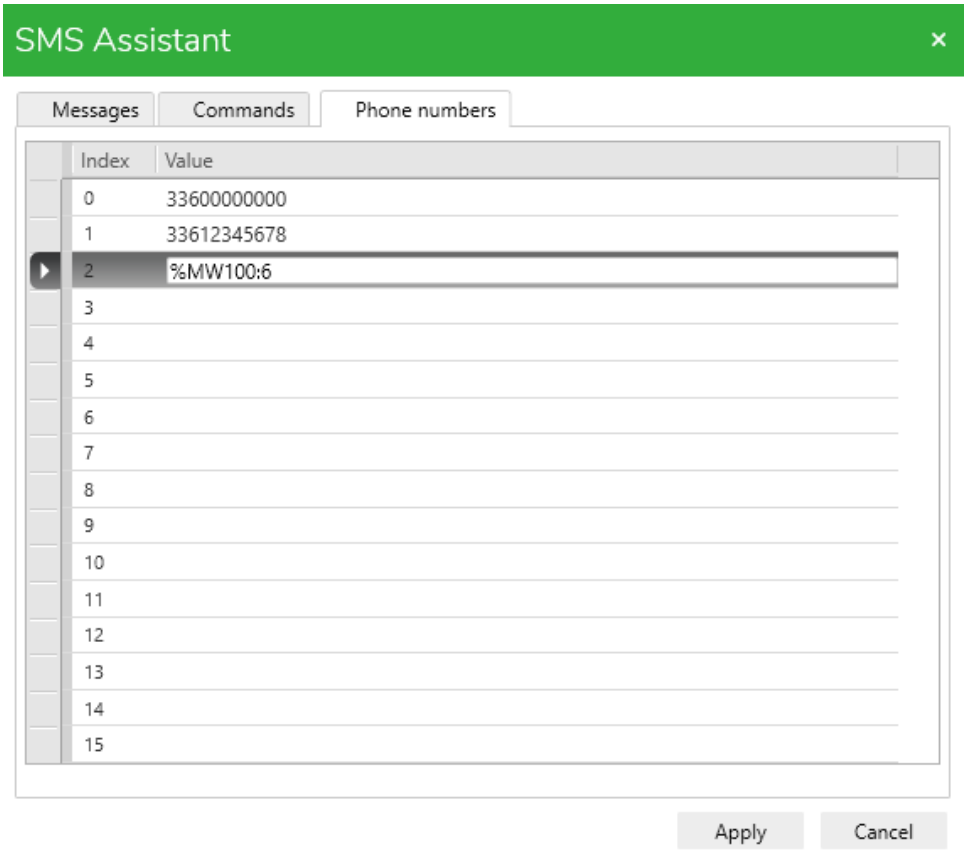
  NOTE: Refer to your modem documentation for international dialing code formats.

If the application is configured with a functional level of at least Level 6.1, you can declare the strings in a `%MW` variable and enter this memory variable in the **Value** field.

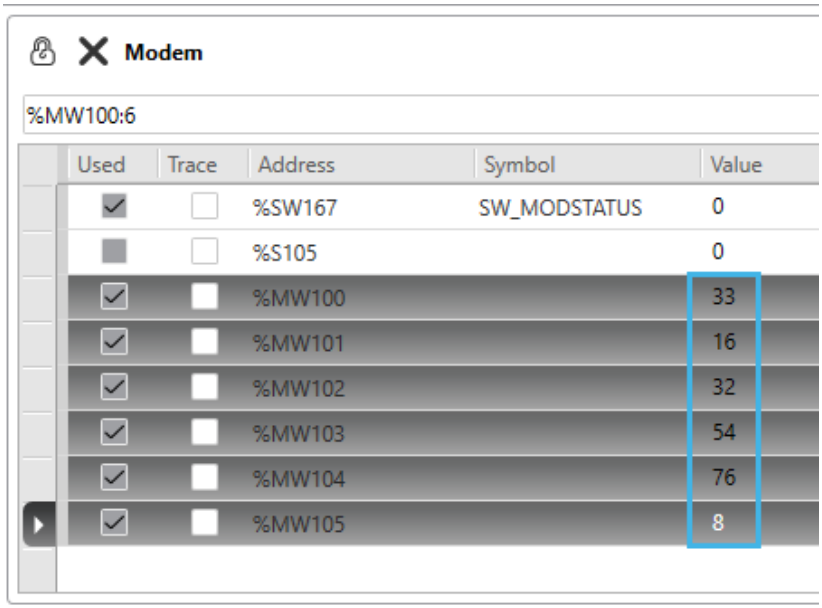
NOTE: `%MW` variables are supported as variables, not as strings.

Each table contains a maximum of 16 entries with an index on each line from 0 to 15.

Example:







In the above example, the phone number is +33 61 23 45 67 8.

The strings contained in the assistant tables can be interpreted in the following formats and message size limit:

Character format	Messages and Commands tables
GSM 7-bit	105 characters max.
UNICODE	45 characters max.
NOTE: The character format is determined automatically by the characters in the text field.	

The following placeholders can be added in the text of **Messages** or **Commands** to be interpreted as variables:

Placeholder	Replaced at execution by:	Number of characters in GSM 7-bit formats	Number of characters in UNICODE format
$DATE(1)	*YY/MM/DD* (present date)	8 + 1	16 + 2
$TIME(1)	*HH:MM:SS* (present time)	8 + 1	16 + 2
$VAD	The *DWORD* value of parameter *%SEND_RECV_SMSi.VAD* converted to text.	12 maximum	24 maximum
$VBD	The *DWORD* value of parameter *%SEND_RECV_SMSi.VBD* converted to text.	12 maximum	24 maximum
$$	The symbol *$*	1	2
NOTE: When the entered text is valid (characters limit not exceeded, valid placeholders), the **Apply** button is active. **(1)** Ignored for **Commands**.			

Examples

This example illustrates the use of placeholders in messages:

Messages	
Message configured	*$DATE* : *$TIME* - Value A = *$VAD* and Value B = *$VBD* !
Placeholder values	*VAD* = 10; *VBD* = 2000
Final SMS sent	15/04/27 : 11:15:43 - Value A = 10 and Value B = 2000 !

This example illustrates the use of placeholders in commands:

Commands	
Command configured	Value A = *$VAD* and Value B = *$VBD* !
SMS received	Value A = 300 and Value B = 2 !
Values captured	*VAD* = 300; *VBD* = 2

Properties

Double-click the function block to open the function properties table.

The properties of this function block cannot be modified in online mode.

The *%SEND_RECV_SMS* function block has the following properties:

Property	Value	Description
Used	Activated / deactivated check box.	Indicates whether the address is in use.
Address	`%SEND_RECV_SMSi`, where *i* is from 0 to the number of objects available on this logic controller.	*i* is the instance identifier. For the maximum number of instances, refer to the Programming Guide of your logic controller.
Symbol	User-defined text.	The symbol uniquely identifies this object. For details, refer to Defining and Using Symbols (see EcoStruxure Machine Expert - Basic, Operating Guide).
Link	**1 - SL1** **2 - SL2**	The serial line on which the modem is configured (**Configuration** tab).
Timeout	0...255 Specified in units of 100 ms, with a default of 100 (10 seconds). A value of 0 means no timeout enforced.	The timeout sets the maximum time to wait to receive a response from the modem. If the timeout expires, the exchange terminates with an error code (`%SEND_RECV_SMSi.CommError` = *01 hex*). If the system receives a response after the timeout expiration, this response is ignored. **NOTE:** The timeout set on the function block overrides the value configured on the EcoStruxure Machine Expert - Basic configuration screen. For more details, refer to the Programming Guide for your logic controller.
Index	0...15 **NOTE:** 0 corresponds to the first string of the list.	• While sending, the value of the index is used to select a text to send from the **Messages** table. • While receiving, the value corresponds to the index in the **Commands** table that matches the received text.
VAD	-214748364 ...2147483647	• While sending, the value in *%SEND_RECV_SMSi.VAD* replaces the placeholder *$VAD* in the text of the SMS. • While receiving, the value in *%SEND_RECV_SMSi.VAD* contains the value where the placeholder *$VAD* is inserted in the SMS stored in the **Commands** table.
VBD	-214748364 ...2147483647	• While sending, the value in *%SEND_RECV_SMSi.VBD* replaces the placeholder *$VBD* in the text of the SMS. • While receiving, the value in *%SEND_RECV_SMSi.VBD* contains the value where the placeholder *$VBD* is inserted in the SMS stored in the **Commands** table.
MASKPHONE	*0000000000000000 bin* to *1000000000000000 bin*	The initial value of the mask. • While sending, this mask is used to select the recipient of the SMS from the **Phone numbers** table. Example: *0000000000000010 bin* = the SMS is sent to the second phone number (index 1) listed in the **Phone numbers** table. • While receiving, the mask is applied to the **Phone numbers** table to create a list of valid originator phone numbers. A bit of the mask indicates which phone number was used to send the SMS to the logic controller. Example: *0000000000000100 bin* means the third phone number of the **Phone numbers** list (index 2) has sent the SMS.
Comment	User-defined text	A comment to associate with this object.

Objects

The *%SEND_RECV_SMS* function block has the following objects:

Object	Description	Value
%SEND_RECV_SMSi.LINK	Port selection	Refer to Properties, page 201. Can be read and written. Can be modified in an animation table.
%SEND_RECV_SMSi.TIMEOUT	Function block timeout	Refer to Properties, page 201. Can be read and written. Can be modified in an animation table.
%SEND_RECV_SMSi.INDEX	Index in messages or commands table	Refer to Properties, page 201. Can be read and written. Can be modified in an animation table.
%SEND_RECV_SMSi.VAD	VAD - placeholder A	Refer to Properties, page 201. Can be read and written. Can be modified in an animation table.
%SEND_RECV_SMSi.VBD	VBD - placeholder B	Refer to Properties, page 201. Can be read and written. Can be modified in an animation table.
%SEND_RECV_SMSi.MASKPHONE	Mask to select entries in the phone number table	Refer to Properties, page 201. Can be read and written. Can be modified in an animation table.
%SEND_RECV_SMSi.COMMERROR	Communication Error Codes	Refer to Communication Error Codes, page 196. Can be only read. Can be modified in an animation table.
%SEND_RECV_SMSi.OPERERROR	Operation Error Codes	Refer to Operation Error Codes, page 196. Can be only read. Can be modified in an animation table.
%SEND_RECV_SMSi.DONE	Execution completed successfully	Refer to Outputs, page 195. Can be only read. Can be modified in an animation table.
%SEND_RECV_SMSi.BUSY	Execution is in progress	Refer to Outputs, page 195. Can be only read. Can be modified in an animation table.
%SEND_RECV_SMSi.ABORTED	Execution was canceled	Refer to Outputs, page 195. Can be only read. Can be modified in an animation table.
%SEND_RECV_SMSi.ERROR	An error was detected	Refer to Outputs, page 195. Can be only read. Can be modified in an animation table.

Timing Diagrams

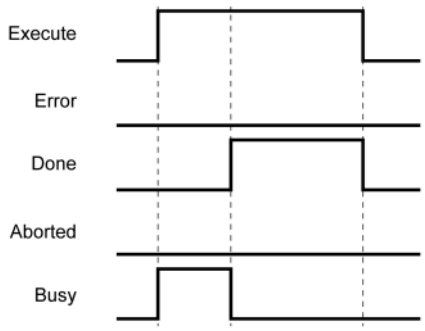
Refer to Signal Behavior of Function Blocks with the input Execute, page 204.

Timing Diagrams for Communication Object Function Blocks

Examples of Timing Diagrams

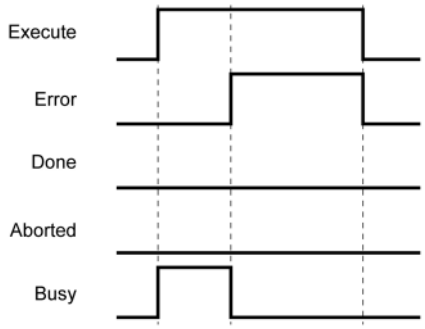
Example 1

Execution completed without an error:



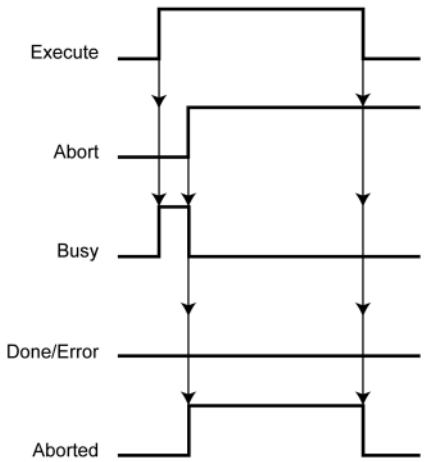
Example 2

Execution completed with an error:

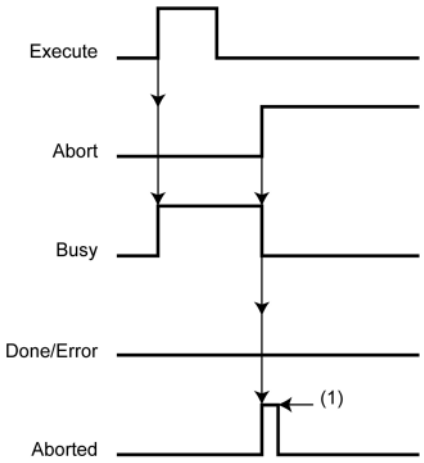


Example 3

Function block aborted by the application:



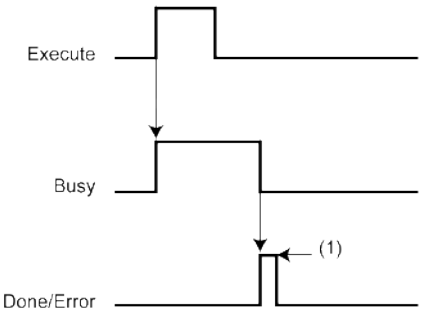
If *Execute* has already been reset to FALSE when the abort request occurred, *Abort* is set to TRUE for one cycle only.



(1) Set to TRUE for one cycle only, as *Execute* has already been set to FALSE.

Example 4

Execution completed without an error after *Execute* has been set to FALSE.



(1) Set to TRUE for one cycle only, as *Execute* has already been set to FALSE.

User-Defined Elements

User-Defined Functions

Overview

A user-defined function allows you to create new functions with one or more input parameters, local variables and a return value. User-defined functions are stored as part of the EcoStruxure Machine Expert - Basic project.

You can call user-defined functions in:

- Master task
- Periodic tasks
- Free POUs

To create user-defined functions, the application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 6.0**.

For information about using user-defined functions, refer to User-Defined Functions in the EcoStruxure Machine Expert - Basic, Operating Guide.

User-Defined Function Blocks

Overview

A user-defined function block allows you to create new function blocks with one or more input and outputs parameters, local variables and a return value. User-defined function blocks are stored as part of the EcoStruxure Machine Expert - Basic project.

You can call user-defined function blocks in:

- Master task
- Periodic tasks
- Events
- Free POUs

To create user-defined function blocks, the application must be configured with a functional level (see EcoStruxure Machine Expert - Basic, Operating Guide) of at least **Level 6.0**.

For information about using user-defined function blocks, refer to User-Defined Function Blocks in the EcoStruxure Machine Expert - Basic, Operating Guide.

Clock Functions

Overview

This chapter describes the time management functions for controllers.

Clock Functions

Introduction

On logic controllers equipped with a Real-Time Clock (RTC) feature, you can use the following time-of-day clock functions when EcoStruxure Machine Expert - Basic is connected to the logic controller:

- **RTC** function blocks, page 163 are used to read the time and date from the RTC, or update the RTC in the logic controller with a user-defined time and date.
- **Schedule** function blocks, page 158 are used to control actions at predefined or calculated times.
- **Time/date stamping**, page 207 is used to assign time and dates to events and measure event duration.

The time-of-day clock can be set by a program, page 207. The controller battery facilitates Clock settings to continue operating for up to 1 year when the controller is turned off. The controller does not have a rechargeable battery. The battery has an average lifetime of 4 years and should be replaced prior to its end of life. In order not to lose the data during battery replacement, change the battery within 120 seconds after the battery is removed from the controller.

The time-of-day clock has a 24-hour format and takes leap years into account.

Time and Date Stamping

Introduction

System words `%SW49` to `%SW53` contain the current date and time in BCD format which is useful for display on or transmission to a peripheral device. These system words can be used to store the time and date of an event.

The *BTI* instructions are used to convert dates and times from BCD format to binary format. For more information, refer to the BCD/Binary conversion instructions, page 58.

Dating an Event

To associate a date with an event, it is sufficient to use assignment operations to transfer the contents of system words to memory words, and then process these memory words (for example, transmission to a display unit using the *EXCH* instruction).

Programming Example

This example shows how to date a rising edge on input %I0.1:

Rung	Instruction
0	LDR %I0.1 [%MW11:5:=%SW49:5]

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Once an event is detected, the word table contains:

Encoding	Most Significant Byte	Least Significant Byte
%MW11	-	Day of the week (1)
%MW12	00	Second
%MW13	Hour	Minute
%MW14	Month	Day
%MW15	Century	Year
(1) 1 = Monday, 2 = Tuesday, 3 = Wednesday, 4 = Thursday, 5 = Friday, 6 = Saturday, 7 = Sunday		

Example of Word Table

Example data for 13:40:30 on Monday 03 June 2013:

Word	Value (hex)	Meaning
%MW11	0001	Monday
%MW12	0030	30 seconds
%MW13	1340	13 hours, 40 minutes
%MW14	0603	06 = June, 03rd
%MW15	2013	2013

Date and Time of the Last Stop

System words %SW54 to %SW57 contain the date and time of the last stop, and word %SW58 contains the code showing the cause of the last stop, in BCD format.

Setting Date and Time

Introduction

You can update the time and date settings in the logic controller by using one of the following methods:

- Using the **RTC Management** tab on the **Commissioning** tab of EcoStruxure Machine Expert - Basic. This method is only available when in online mode (see EcoStruxure Machine Expert - Basic, Operating Guide).

  You can choose between 2 methods:

  - Manual: this method displays a time/date picker and lets you manually choose what time to set in the logic controller.
  - Automatic: this method uses the time of the PC on which EcoStruxure Machine Expert - Basic is running.

  For more details, refer to RTC Management (see EcoStruxure Machine Expert - Basic, Operating Guide).
- In a program, using RTC function blocks, page 163.
- When in online mode, by updating the system words, either directly or programmatically using operating blocks, `%SW49` to `%SW53` or system word `%SW59`.

  NOTE: The date and time can only be set when the RTC feature is available in your logic controller (refer to the *programming guide* of your logic controller).

Using %SW49 to %SW53

To use system words `%SW49` to `%SW53` to set the date and time, bit `%S50` must be set to 1. While `%S50` is set to 1, system words `%SW49` to `%SW53` are no longer updated by the controller. On a falling edge of `%S50` (`%S50` set to 0), the internal RTC of the controller is updated by the values in `%SW49` to `%SW53`.The controller then resumes updating `%SW49` to `%SW53` using the RTC.

This table lists the system words containing date and time values (in BCD) for real-time clock (RTC) functions:

System Word	Description
`%SW49`	xN Day of week (N=1 for Monday)
`%SW50`	00SS: seconds
`%SW51`	HHMM: hour and minute
`%SW52`	MMDD: month and day
`%SW53`	CCYY: century and year

Refer to the *programming guide* of your controller for a complete list of system bits and words.

Programming example:

Rung	Instruction	Comment
0	`LD %S50` `R %S50`	–

1	`LD %I0.1`
	`[%SW50:=%MW11]`
	`[%SW51:=%MW12]`
	`[%SW52:=%MW13]`
	`[%SW53:=%MW14]`
	`S %S50`
		Refer to BCD/Binary Conversion Instruction, page 58.

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Words %MW11 to %MW14 contain the new date and time (see Review of BCD Code, page 58) and corresponds to the coding of words %SW50 to %SW53.

NOTE: %SW49 (Day of the week) is automatically calculated based upon the date supplied.

The word table must contain the new date and time:

Encoding	Most Significant Byte	Least Significant Byte
%MW11	–	Second
%MW12	Hour	Minute
%MW13	Month	Day
%MW14	Century	Year

Example data for 03 June 2013:

Word	Value (hex)	Meaning
%MW11	0030	30 seconds
%MW12	1340	13 hours, 40 minutes
%MW13	0603	06 = June, 03rd
%MW14	2013	2013

Using %SW59

Another method of updating the date and time is to use system bit %S59 and date adjustment system word %SW59.

Setting bit %S59 to 1 enables adjustment of the current date and time by word %SW59. %SW59 increments or decrements each of the date and time components on a rising edge.

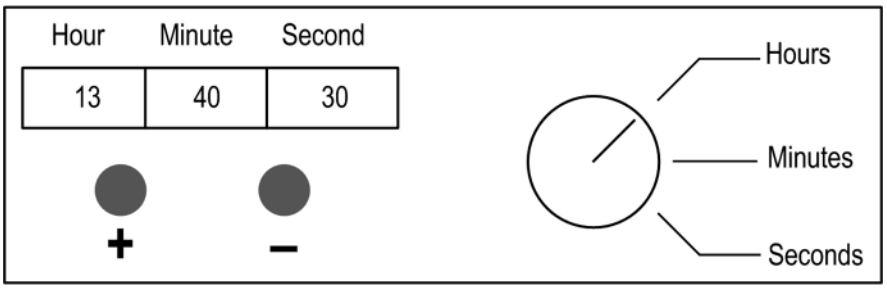
This table describes each bit of the system word %SW59 for adjusting date and time parameters:

Increment	Decrement	Parameter
Bit 0	Bit 8	Day of week(1)
Bit 1	Bit 9	Seconds
Bit 2	Bit 10	Minutes
Bit 3	Bit 11	Hours
Bit 4	Bit 12	Days
Bit 5	Bit 13	Month
Bit 6	Bit 14	Years
Bit 7	Bit 15	Centuries(1)
(1) Day of week and centuries cannot be modified (increment or decrement) by the user.		

Refer to the *programming guide* of your controller for a complete list of system bits and words.

Application Example

This front panel is created to modify the hour, minutes, and seconds of the internal clock.



Description of the commands:

- The Hours/Minutes/Seconds switch selects the time display to change using inputs `%I0.2`, `%I0.3`, and `%I0.4` respectively.
- Push button "+" increments the selected time display using input `%I0.0`.
- Push button "-" decrements the selected time display using input `%I0.1`.

This program reads the inputs from the panel and sets the internal clock:

Rung	Instruction	Comment
0	`LD %M0` `ST %S59`	–

1	`LD %I0.2`
	`ANDR %I0.0`
	`ST %SW59:X3`
		Hour

2	`LD %I0.2`
	`ANDR %I0.1`
	`ST %SW59:X11`
		–

3	`LD %I0.3`
	`ANDR %I0.0`
	`ST %SW59:X2`
		Minute

4	`LD %I0.3`
	`ANDR %I0.1`
	`ST %SW59:X10`
		–

5	`LD %I0.4`
	`ANDR %I0.0`
	`ST %SW59:X1`
		Second

6	`LD %I0.4`
	`ANDR %I0.1`
	`ST %SW59:X9`
		–

NOTE: Refer to the reversibility procedure, page 14 to obtain the equivalent Ladder Diagram.

Glossary

A

%:

According to the IEC standard, % is a prefix that identifies internal memory addresses in the logic controller to store the value of program variables, constants, I/O, and so on.

analog input:

Converts received voltage or current levels into numerical values. You can store and process these values within the logic controller.

analog output:

Converts numerical values within the logic controller and sends out proportional voltage or current levels.

ASCII:

(*American standard code for Information Interchange*) A protocol for representing alphanumeric characters (letters, numbers, certain graphics, and control characters).

C

conditional element:

Allows to implement conditions in the program in offline mode.

D

data logging:

Stores permanently data from objects or strings.

F

function block:

A programming unit that has 1 or more inputs and returns 1 or more outputs. FBs are called through an instance (function block copy with dedicated name and variables) and each instance has a persistent state (outputs and internal variables) from 1 call to the other.

Examples: timers, counters

I

instruction list language:

A program written in the instruction list language that is composed of a series of text-based instructions executed sequentially by the controller. Each instruction includes a line number, an instruction code, and an operand (see IEC 61131-3).

L

ladder diagram language:

A graphical representation of the instructions of a controller program with symbols for contacts, coils, and blocks in a series of rungs executed sequentially by a controller (see IEC 61131-3).

loop element:

Allows to implement a sequence of instructions in the program in offline mode.

Q

%Q:

According to the IEC standard, %Q represents an output bit (for example, a language object of type digital OUT).

R

RTC:

(*real-time clock*) A battery-backed time-of-day and calender clock that operates continuously, even when the controller is not powered for the life of the battery.

Index

A

ABS 67
absolute value 53
ACOS 69
add 53
addressing
  format 22
  I/O objects 21
AND 42
AND operators 42
ANDF 42
ANDN 42
ANDR 42
arithmetic instructions 53
ASCII
  examples 133
ASIN 69
assignment instructions
  bit strings 51
  inserting in Ladder Diagram rungs 15
  numerical 50
  object tables 84
  words 52
assignment operators 41
ATAN 69

B

bit objects
  function block 36
bit strings 31
boolean instructions 37

C

%C 120
calculation 53
clock functions
  overview 207
  setting date and time 209
  time and date stamping 207
comparison blocks
  inserting IL expressions in 18
comparison expression
  inserting in Ladder Diagram rungs 18
comparison instructions 49
conditional element 62
  configuring 63
conversion instructions
  BCD/Binary 58
  single/double word 59
COS 69
counter
  configuration 121
  description 120
  programming example 122

D

data logging 167
  configuring 168
decrement 53
DEG_TO_RAD 70
DINT_TO_REAL 71
divide 53
double word objects
  description 27
  function block 36
%DR 147
Drive objects 170
Drum
  configuration 148
  description 147
  programming example 149
Drum assistant 148

E

END instructions 60
EQUAL_ARR 86
examples, source code 14
EXCH 124
exchange instructions
  EXCH1 124
  EXCH2 124
  EXCH3 124
exclusive OR operators 44
EXP 67
EXPT 67

F

FALLING
  operator 47
falling edge
  detection with FALLING operator 39
  detection with LDF instruction 39
FIND_ 87
floating point objects
  description 27
function blocks
  counter 120
  Drum 147
  general description 35
  LIFO/FIFO register 141
  message 126
  programming principles 103
  %READ_VAR 171
  Real-Time Clock (RTC) 163
  Schedule blocks 158
  %SEND_RECV_MSG 187
  %SEND_RECV_SMS 192
  shift bit register 152
  step counter 155
  timer 106
  user-defined 206
  %WRITE_READ_VAR 182
  %WRITE_VAR 177

G

Grafcet Step properties 169

I

%I 21
increment 53
index overflow 34
input/output address format 21
instructions

angle conversion....70
arithmetic....53
ASCII....72
ASCII to double word conversion....79
ASCII to float conversion....76
ASCII to integer conversion....74
boolean....37
comparison....49
double word to ASCII conversion....81
END....60
exchange....124
float to ASCII conversion....78
input/output objects....101
integer to ASCII conversion....75
integer/floating conversion....71
jump....61
NOP....61
object table....83
object tables....85
ROUND....72
SR....65
stack....82
subroutine....65
trigonometric....69
INT_TO_REAL....71
%IW....21
%IWS....21

K

%KD....27
%KF....27
%KW....23

L

LD....39
LDF....39
LDN....39
LDR....38–39
LIFO/FIFO register
configuration....142
description....141
FIFO....145
LIFO....144
programming example....146
LKUP....92
LN....67
load operators....39
LOG....67
logic instructions....55
loop element....64
configuring....65

M

%M....20
MAX_ARR....88
%MD....27
MEAN....95
memory bit objects
description....20
message
configuration....129
description....126
programming example....132
%MF....27
MIN_ARR....88
Modbus
standard requests and examples....135
%MSG....126
multiply....53
%MW....23

N

N....46
network objects....102
NOP instruction....61
NOT operator....46
numerical instructions
shift....56
numerical processing
assignment....50
overview....50

O

objects
definition of....19
direct address....33
indexed....33
indexed address....33
network....102
software....103
structured....31
tables....31
OCCUR_ARR....89
operation blocks
inserting assignment instructions in....15
operations
inserting in Ladder Diagram rungs....15
operators
AND....42
assignment....41
FALLING....47
load....39
NOT....46
OR....43
RISING....47
XOR....44
OR....43
OR operators....43
ORF....43
ORN....43
ORR....43
overflow
index....34

P

PID....166

Q

%Q....21
%QW....21
%QWS....21

R

R....41, 141
RAD_TO_DEG....70
READ_IMM_IN....96

%READ_VAR 171
configuration 175
description 171
programming example 177
REAL_TO_DINT 71
REAL_TO_INT 71
remainder 53
RISING
operator 47
rising edge
detection with LDR instruction 38
detection with RISING operator 38
ROL_ARR 90
ROR_ARR 90
RTC
configuration 165

S

%S 20, 41
%SBR 152
%SC 155
%SCH 158
schedule blocks
description 158
programming and configuring 161
%SEND_RECV_MSG 187
configuration 189
description 187
programming example 191
%SEND_RECV_SMS 192
configuration 198
description 192
shift bit register
configuration 153
description 152
programming example 154
shift instructions 56
SIN 69
SORT_ARR 91
source code, using example 14
SQRT 67
square root 53
SR (subroutine) instructions 65
ST 41
stack instructions
MPP 82
MPS 82
MRD 82
step counter
configuration 156
description 155
programming example 157
STN 41
subtract 53
SUM_ARR 85
%SW 23
SWAP 96
system bits
%S18 28
system words
%SW17 28

T

tables
instructions on 83
TAN 69
timer
configuration 107
description 106
programming example 119
TOF type 112
TON type 109
TP type 116
%TM 106
tools
Drive objects 170
network objects 102
TRUNC 67

U

user-defined function blocks 206
user-defined functions 206
using source code examples 14

W

word objects
description 23
function block 36
WRITE_IMM_OUT 98
%WRITE_READ_VAR 182
configuration 184
description 182
programming example 186
%WRITE_VAR 177
configuration 179
description 177
programming example 181

X

%X 20
%Xi (Grafcet Step) properties 169
XOR 44
XORF 44
XORN 44
XORR 44

Schneider Electric
35 rue Joseph Monier
92500 Rueil Malmaison
France

+ 33 (0) 1 41 29 70 00

www.se.com

As standards, specifications, and design change from time to time, please ask for confirmation of the information given in this publication.

© 2025 Schneider Electric. All rights reserved.

EIO0000003289.04