SoMachine Basic

Generic Functions Library Guide

11/2014

EIO0000001474.03

www.schneider-electric.com

Schneider Electric

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

When devices are used for applications with technical safety requirements, the relevant instructions must be followed.

Failure to use Schneider Electric software or approved software with our hardware products may result in injury, harm, or improper operating results.

Failure to observe this information can result in injury or equipment damage.

Table of Contents

Safety Information

Important Information

NOTICE

Read these instructions carefully, and look at the equipment to become familiar with the device before trying to install, operate, 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.

About the Book

At a Glance

Document Scope

This guide describes how to use Function Blocks and Instructions in programs you create with SoMachine Basic software. The descriptions apply to all logic controllers supported by SoMachine Basic.

Validity Note

The information in this manual is applicable **only** for SoMachine Basic compatible products.

This document has been updated with the release of SoMachine Basic V1.3.

The technical characteristics of the devices described in this document also appear online. To access this information online:

Step	Action
1	Go to the Schneider Electric home page *www.schneider-electric.com*.
2	In the **Search** box type the reference of a product or the name of a product range. ● Do not include blank spaces in the model number/product range. ● To get information on grouping similar modules, use asterisks (*).
3	If you entered a reference, go to the **Product Datasheets** search results and click on the reference that interests you. If you entered the name of a product range, go to the **Product Ranges** search results and click on the product range that interests you.
4	If more than one reference appears in the **Products** search results, click on the reference that interests you.
5	Depending on the size of your screen, you may need to scroll down to see the data sheet.
6	To save or print a data sheet as a .pdf file, click **Download XXX product datasheet**.

The characteristics that are presented in this manual should be the same as those characteristics that appear online. In line with our policy of constant improvement, we may revise content over time to improve clarity and accuracy. If you see a difference between the manual and online information, use the online information as your reference.

Related Documents

Title of Documentation	Reference Number
SoMachine Basic Operating Guide	EIO0000001354 (ENG) EIO0000001355 (FRA) EIO0000001356 (GER) EIO0000001357 (SPA) EIO0000001358 (ITA) EIO0000001359 (CHS) EIO0000001366 (POR) EIO0000001367 (TUR)
Modicon M221 Logic Controller Advanced Functions Library Guide	EIO0000002007 (ENG) EIO0000002008 (FRA) EIO0000002009 (GER) EIO0000002010 (SPA) EIO0000002011 (ITA) EIO0000002012 (CHS) EIO0000002013 (POR) EIO0000002014 (TUR)

You can download these technical publications and other technical information from our website at www.schneider-electric.com.

Product Related Information

⚠ WARNING
LOSS OF CONTROL

- The designer of any control scheme must consider the potential failure modes of control paths and, for certain critical control functions, provide a means to achieve a safe state during and after a path failure. Examples of critical control functions are emergency stop and overtravel stop, power outage and restart.
- Separate or redundant control paths must be provided for critical control functions.
- System control paths may include communication links. Consideration must be given to the implications of unanticipated transmission delays or failures of the link.
- Observe all accident prevention regulations and local safety guidelines.[1]
- Each implementation of this equipment must be individually and thoroughly tested for proper operation before being placed 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.

Chapter 1
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.

What Is in This Chapter?

This chapter contains the following topics:

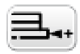
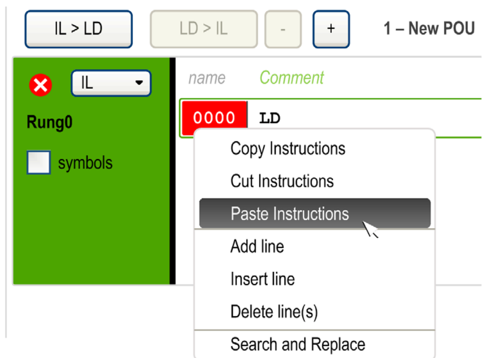
How to Use the Source Code Examples

Overview

Except where explicitly mentioned, the source code examples contained in this book 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 SoMachine 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:

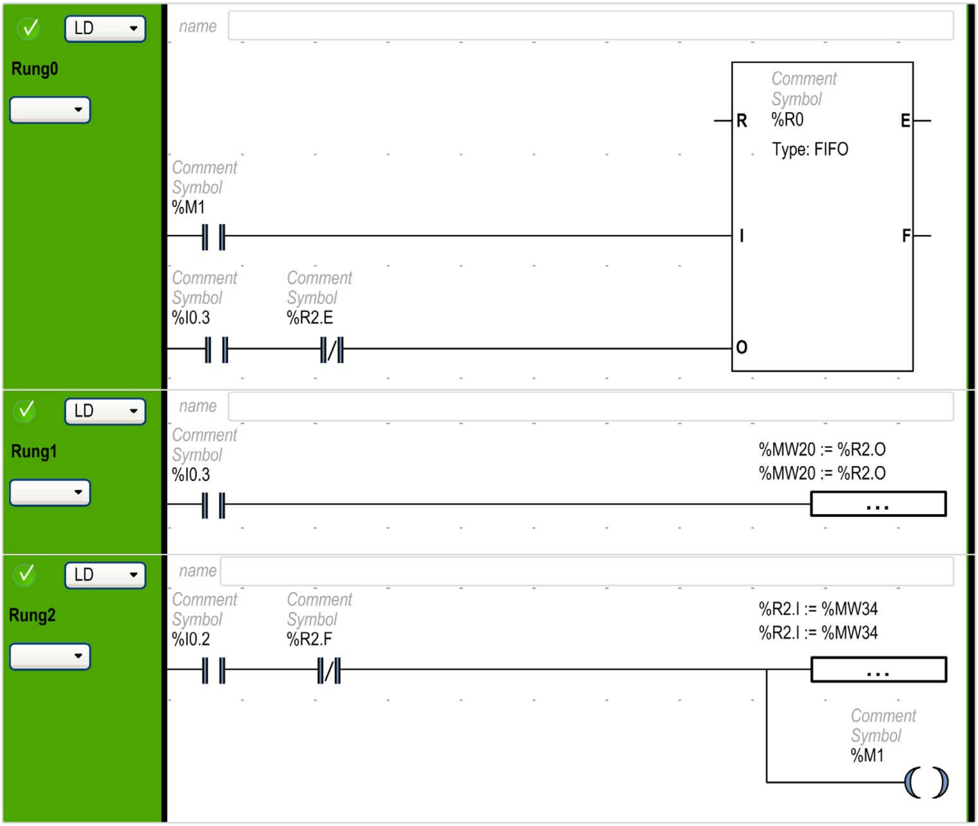
Rung	Source Code

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:



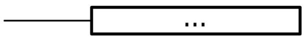
Operation Blocks

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:

Symbol

operation expression



To insert an operation block in a rung:

Step	Action
1	Click the **Operation Block**  button on the toolbar.
2	Click in the Action zone (last 2 columns) of the rung to insert the **Operation Block**.
3	Double-click the **operation expression** line.
4	Type a valid Instruction List operation or assignment instruction and press ENTER.

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:

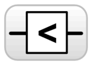
- Move the mouse over the **operation expression** line, or
- Select **Tools** →**Program Messages**.

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:

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.

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**.

Chapter 2
Language Objects

What Is in This Chapter?

This chapter contains the following topics:

Objects

Overview

In SoMachine 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:

%	M, S, or X	i
Symbol	Object type	Object instance identifier

This table describes the elements in the addressing format:

Group	Item	Description
Symbol	%	The percent symbol always 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 *(see page 23)*.

For information on addressing of bit extracted from word, refer to Extracting Bit from Word Object *(see page 28)*.

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:

%	I, Q, IW, QW, IWS, or QWS	y	.	z
Symbol	Object type	Module number	point	Channel number

This table describes the components of the addressing format:

Component	Item	Value	Description
Symbol	`%`	–	The percent symbol always 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 status (word object)
	`QWS`	–	Analog output status (word object)
Module number	`y`	0	Embedded I/O channel on 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.

(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.

Component	Item	Value	Description
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 model or expansion module type.
(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.			

Description

This table lists and describes all 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 *(see page 23)* 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).
%IWS0.1	Analog input status of analog input 1 on the controller (embedded I/O).
%QWS1.1	Analog output status of analog output 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:

%	M, K or S	W	i
Symbol	Object type	Format	Object instance identifier

This table describes the elements in the addressing format:

Group	Item	Description
Symbol	%	The percent symbol always 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 SoMachine 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:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Bit position
±	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	Bit value

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 *(see page 26)* 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 always 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

Object	Value
%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, all other 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 only be written or modified by using software during configuration.	%KWi	No
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 Bit from Word Object

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 page 23)*. **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**	**#QNAN (Invalid)**	**#INF (Infinite)**
Square root of an operand	`SQRT(x)`	x < 0	x > 1.7E38
Power of an integer by a real `EXPT(%MF,%MW)`	`EXPT(y, x)` (where: `x^y = %MW^%MF`)	x < 0	y.ln(x) > 88
Base 10 logarithm	`LOG(x)`	x <= 0	x > 2.4E38
Natural logarithm	`LN(x)`	x <= 0	x > 1.65E38
Natural exponential	`EXP(x)`	x < 0	x > 88.0

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 (1.#NAN or -1.#NAN)
%SW17:X1	Reserved
%SW17:X2	Division by 0, result is infinite (-1.#INF or 1.#INF)
%SW17:X3	Result greater in absolute value than +3.402824E+38, result is infinite (-1.#INF or 1.#INF)
%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:

%	M or K	F	i
Symbol	Object type	Format	Object instance identifier

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

%	M or K	D	i
Symbol	Object type	Format	Object instance identifier

This table describes the elements in the addressing format:

Group	Item	Description
Symbol	%	The percent symbol always 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 (not by program)
Double constant		%KDi	Yes (not by program)

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. %KF543 corresponds to %KW543 and %KW544.

Structured Objects

Introduction

Structured objects are combinations of adjacent objects. SoMachine 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`

%M8	%M9	%M10	%M11	%M12	%M13

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 *(see page 48)*.

Available Types of Bits

Available types of bits for bit strings:

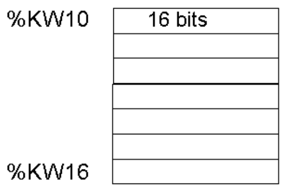
Type	Address	Write Access
Digital input bits	`%I0.0:L` or `%I1.0:L`(1)	No
Digital output bits	`%Q0.0:L` or `%Q1.0:L`(1)	Yes
System bits	`%Si:L` with i multiple of 8	Depending on i
Grafcet step bits	`%Xi:L` with i multiple of 8	Yes (by program)
Memory bits	`%Mi:L` with i multiple of 8	Yes
(1) Only I/O 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 *(see page 48)*.

Available Types of Words

Available types of words for word tables:

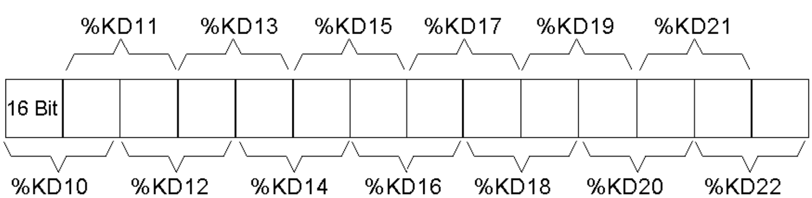
Type	Address	Write Access
Memory words	`%MWi:L`	Yes
Constant words	`%KWi:L`	No
System words	`%SWi:L`	Depending on i

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

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 *(see page 48)*.

Available Types of Double Words

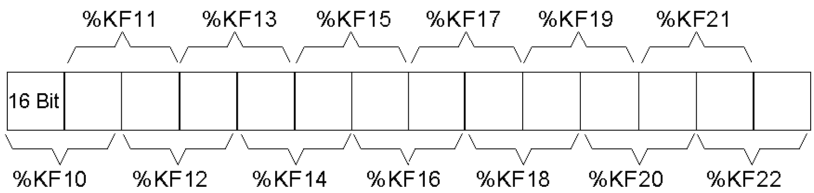
Available types of words for double word tables:

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 *(see page 48)*.

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, or floating point 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
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
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.		

Indexed objects can be used with the Assignment instruction *(see page 61)* and in Comparison instructions *(see page 57)*.

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: You are responsible for monitoring any 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.

⚠ 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

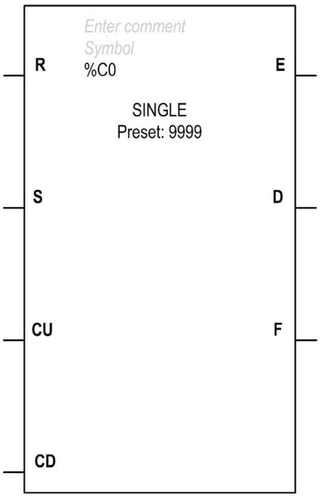
Introduction

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 always 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 StepCounter 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 *(see page 129)*.
- 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, simply 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.

Chapter 3
Instructions

What Is in This Chapter?

This chapter contains the following sections:

Section 3.1
Boolean Processing

Aim of This Section

This section provides an introduction to Boolean processing instructions.

What Is in This Section?

This section contains the following topics:

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`

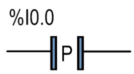
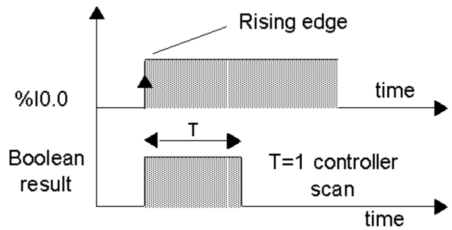
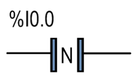
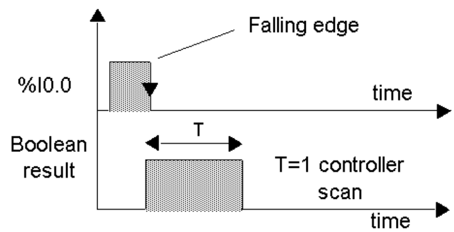
NOTE: Refer to the reversibility procedure *(see page 14)* to obtain the equivalent Ladder Diagram.

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 edges:

Edge	IL Instruction	Ladder Diagram	Timing Diagram
Rising Edge	`LDR %I0.0`		
Falling Edge	`LDF %I0.0`		

NOTE: Rising and falling edge can be apply only with `%I` and `%M`.

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 *(see 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

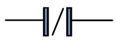
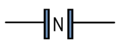
Load Operators (LD, LDN, LDR, LDF)

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 permitted operands:

Operators	Ladder Diagram Equivalent	Permitted 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
LDN		
LDR		%I, %M
LDF		

Coding Examples

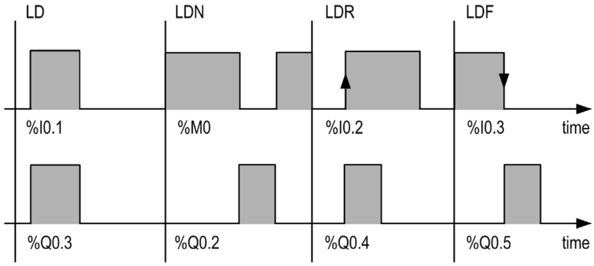
Examples of Load instructions:

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 *(see 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):



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 permitted operands:

Operators	Ladder Diagram Equivalent	Permitted Operands
ST	—( )	%Q, %M, %S, %X, %BLK.x %QW:Xk, %MW:Xk, %SW:Xk(1)
STN	—(/)	
S	—(S)	
R	—(R)	
(1) %SW:Xk is on non-read-only system words.		

Coding Examples

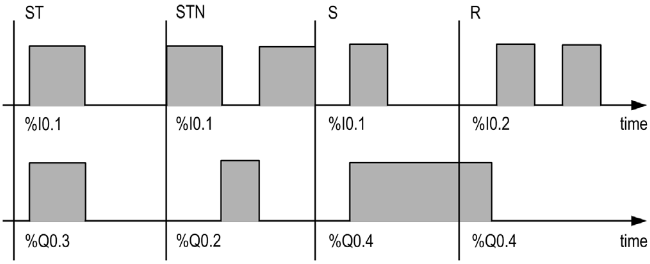
Examples of Assignment instructions:

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 *(see 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):



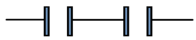
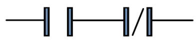
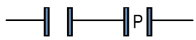
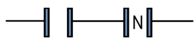
Logical AND Operators (AND, ANDN, ANDR, ANDF)

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 permitted operands:

Operators	Ladder Diagram Equivalent	Permitted 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
ANDN		
ANDR		%I, %M
ANDF		

Coding Examples

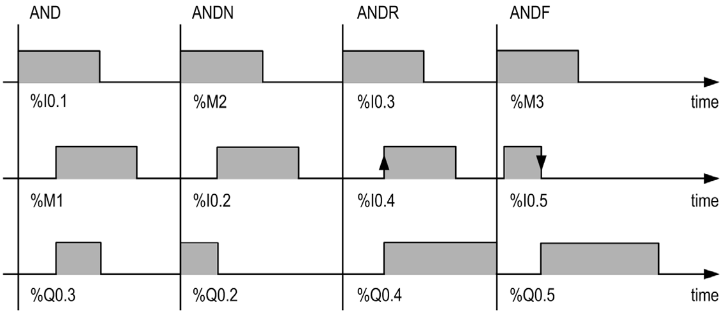
Examples of logical AND instructions:

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 *(see 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):



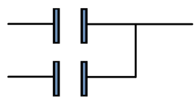
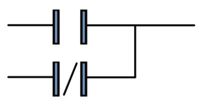
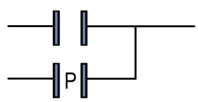
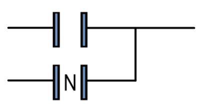
Logical OR Operators (OR, ORN, ORR, ORF)

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 permitted operands:

Operators	Ladder Diagram Equivalent	Permitted 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
ORN		
ORR		%I, %M
ORF		

Coding Examples

Examples of logical OR instructions:

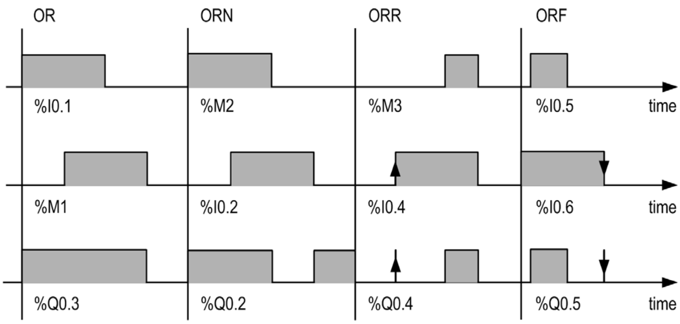
Rung	Instruction
0	LD %I0.1 OR %M1 ST %Q0.0
1	LD %I0.2 ORN %M2 ST %Q0.1

Rung	Instruction
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 *(see 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.

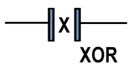
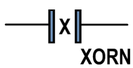
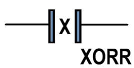
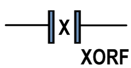
The XORN operator performs an exclusive OR operation between the inverse of the operand and the Boolean result of the preceding instruction.

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 permitted operands:

Operators	Ladder Diagram Equivalent	Permitted 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

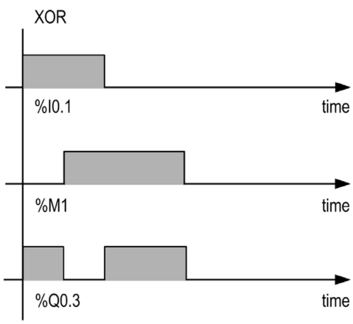
Equivalent logical instructions of the XOR operator:

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

NOTE: Refer to the reversibility procedure *(see page 14)* to obtain the equivalent Ladder Diagram.

Timing Diagram

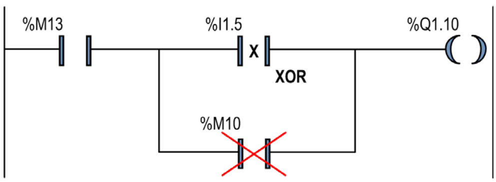
The following diagram illustrates the timing, and the effect on the output, of the code from the coding example(s):



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 detected error:



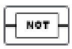
NOT Operator (N)

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	Permitted Operands
N		Not applicable.

Coding Examples

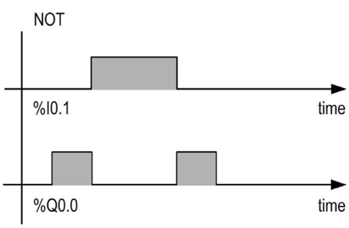
Example of `NOT` instruction:

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

NOTE: Refer to the reversibility procedure *(see 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):



Comparison Instructions

Introduction

Comparison operators are used to compare 2 operands.

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 *(see 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 3.2
Numerical Processing

Aim of This Section

This section provides an introduction to Numerical Processing.

What Is in This Section?

This section contains the following topics:

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 *(see page 17)* 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

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 *(see page 17)* 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 *(see 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 *(see page 17)* 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, %QWi, %SWi %MWi[MWi], %MDi, %MDi[%MWj], %Mi:L, %Qi:L, %Si:L, %Xi:L	Immediate value, %MWi, %KWi, %IW, %QWi, %IWSi, QWSi, %SWi, %MWi[MWi], %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 *(see 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 *(see page 17)* 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]	Immediate value, %MWi, %KWi, %IWi, %QWi, %IWSi, %QWSi, %SWi, %BLK.x[2]
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.		

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 *(see 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 *(see page 68)*). 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 always 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. It is your responsibility to monitor for these potential errors, and to 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 is responsible for managing 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 *(see 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 *(see page 17)* 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Application Example

Logical AND instruction:

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

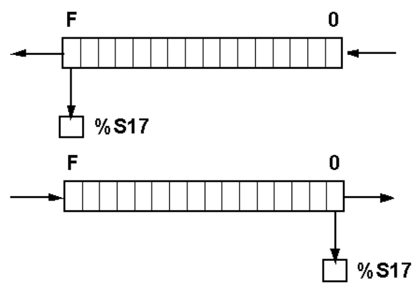
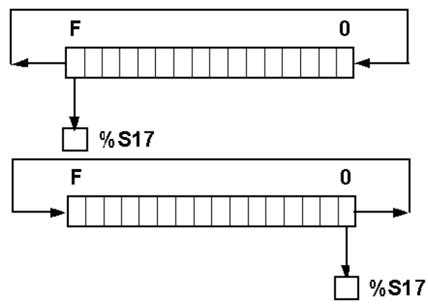
When %MW32 = 0001 1011 (binary) (27 (decimal)) and %MW12 = 0011 0110 (binary) (54 (decimal)) then the result will be %MW15 = 0001 0010 (binary) (18 (decimal)).

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.	
n 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 *(see page 17)* in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax for Shift instructions:

Operator	Syntax
SHL, SHR	[Op1: = operator (Op2,n)]
ROL, ROR	
n 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 *(see 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:

Decimal	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 *(see page 17)* 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 *(see 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 *(see page 17)* 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	[-]
CONCATW	Op1 = operator (Op2, Op3))	%MDi, %KDi, %BLK.x	%MWi, %KWi, immediate value	%MWi, %KWi, immediate value
DWORD	Op1 = operator (Op2)	%MDi, %KDi, %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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 3.3
Program

Aim of This Section

This section provides an introduction to program instructions.

What Is in This Section?

This section contains the following topics:

END Instructions

Introduction

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

END, ENDC, and ENDCN

3 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

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.

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 *(see 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Guidelines

- Jump instructions are not permitted 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, attention must be paid to the program scan time. Extended scan time can cause triggering of the watchdog timer.

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 SoMachine Basic, Operating Guide)* for information on creating a Free POU and subroutine, and defining the subroutine number. Also, refer to Managing POUs *(see SoMachine 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 subroutines.
3 The subroutine instruction must be written Free POU independent of the main program.

For more information about subroutines, refer to Creating Periodic Task *(see SoMachine 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 *(see 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 generator a detected compiler error.
- Subroutine instructions are not permitted 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.

Section 3.4
Floating Point

Aim of This Section

This section describes the advanced instructions of floating point.

What Is in This Section?

This section contains the following topics:

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 *(see page 17)* 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: SoMachine 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Rules of Use

- Operations on floating point and integer values cannot be directly mixed. Conversion operations *(see page 88)* convert into one or other of these formats.
- The system bit `%S18` is managed in the same way as integer operations *(see page 88)*, 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 *(see page 17)* 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 *(see 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 4096π but their precision decreases progressively for angles outside the period -2π and $+2\pi$ because of the imprecision brought 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 *(see page 17)* 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 *(see 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 4096π (for RAD_TO_DEG conversions).

For values outside these ranges, the displayed result will be + 1.#QNAN, the %S18 and %SW17:X0 bits being set at 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 *(see page 17)* 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	%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 *(see 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.

Section 3.5
ASCII

Aim of This Section

This section describes the advanced instructions of ASCII.

What Is in This Section?

This section contains the following topics:

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 *(see page 17)* 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 always 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 checks 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 *(see page 17)* 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)]
```

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 always reads the most significant byte first.
- Any 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 an "separator" character, the value 0 is returned and the bit %S18 is set to 1.
- The scientific notation is not authorized.

Application Example

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

Parameter	Hexadecimal Value	ASCII Value
%MW10	16#3932	9, 2
%MW11	16#3133	1, 3
%MW12	6#2038	‘ ‘, 8
%MW13	16#3820	8, ‘ ‘

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 *(see page 17)* 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 always writes the most significant byte first.
- End character is "Enter" (ASCII 13).
- The function automatically determines how many %MWs should be filled with ASCII values (from 1 to 4).

Syntax Errors

The software checks 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):`

If ...	Then...	
Integer Value	**Hexadecimal Value**	**ASCII Value**
%MW1 = 123	%MW10 = 16#3231	2, 1
	%MW11 = 16#0D33	3
%MW1 = 45	%MW10 = 16#3534	5, 4
	%MW11 = 16#000D	'enter'
%MW1 = 7	%MW10 = 16#0D37	'enter', 7
%MW1 = -12369	%MW10 = 16#3145	1, '-'
	%MW11 = 16#3332	3, 2
	%MW10 = 16#3936	9, 6
	%MW11 = 16#000D	'enter'

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 *(see page 17)* 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 always 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 "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 1.#QNAN 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 Value
%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 = 1.#QNAN
%MF20 := ASCII_TO_FLOAT(%MW12)	%MF20 = 6500.0
%MF20 := ASCII_TO_FLOAT(%MW13)	%MF20 = 1.#QNAN
%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 *(see page 17)* 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 always 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 is "Enter" (ASCII 13),
- The function automatically determines how many %MWs should be filled with ASCII values,
- Conversion precision is 6 figures
- The scientific notation is not authorized.

Syntax Errors

The software checks 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	1.23456e+09
0.000000921	9.21e-07
9.87654321	9.87654
1234	1.234e+03

Section 3.6
Stack Operators

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.

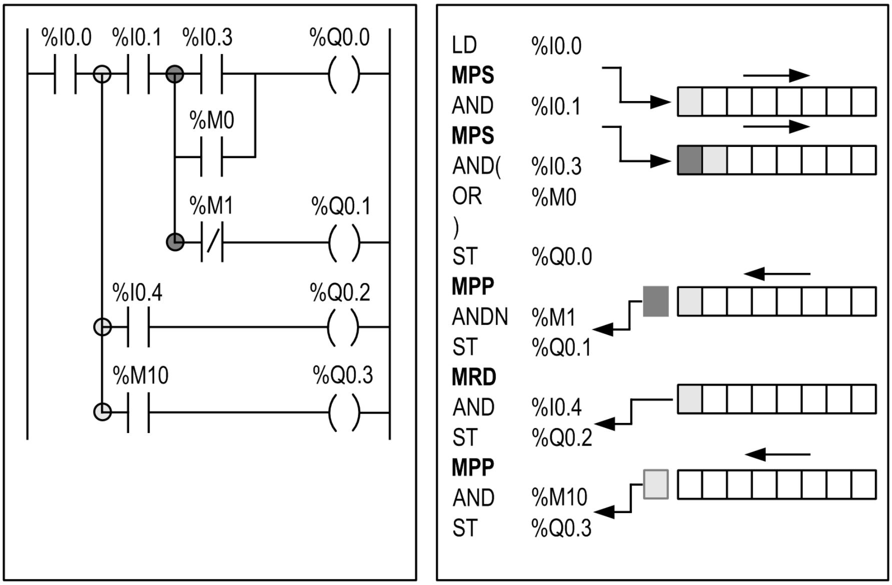
Syntax

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 stack (a push) and shifts the other values to the bottom of the stack.
`MRD`	Memory Read from stack	Reads the top of stack into the accumulator.
`MPP`	Memory Pop from stack	Copies the value at the top of stack into the accumulator (a pop) and shifts the other values towards the top of the stack.

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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 3.7
Instructions on Object Tables

Aim of This Section

This section describes instructions to Object Tables:
- Of double words,
- Of floating point objects.

What Is in This Section?

This section contains the following topics:

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 *(see page 106)*) or double word table
- Word to word table (see rung 1 of structure example *(see page 106)*)
- Word table to word table (see rung 2 of structure example *(see page 106)*)
  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 *(see page 17)* 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	%MWi:L, %SWi:L, immediate whole value, %MWi, %KWi, %IW, %QW, %SWi, %BLK.x
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:=%KW20:30]`

NOTE: Refer to the reversibility procedure *(see 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 *(see page 17)* 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Application Example

```
%MD4:=SUM(%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 whole table.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions *(see page 17)* 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: it is mandatory that the tables are of the same length and same 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 *(see 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 (different first 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 *(see page 17)* 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 *(see 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 *(see page 17)* 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 Object instance identifier for the memory variable. **n** Memory index of the table that indicates the base address for the search. **L** 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 dring the search. For more information, refer to Possibility of Overlap Between Objects *(see page 31)*.

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 *(see 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 a number of elements equal to a given value.

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions *(see page 17)* in Ladder Diagram rungs using an **Operation Block** graphical element.

Syntax of table search instructions for max and min values:

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

Parameters of table search instructions for max and min values:

Type	Result (Res)	Table (Tab)	Value (Val)
Double word tables	%MWi	%MDi:L,%KDi:L	%MDi,%KDi
Floating word tables	%MFi	%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 *(see page 14)* to obtain the equivalent Ladder Diagram.

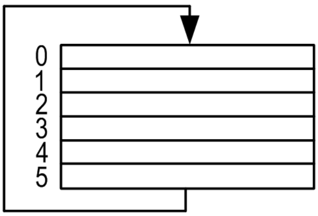
Table Rotate Shift Functions

Introduction

There are 2 shift functions:

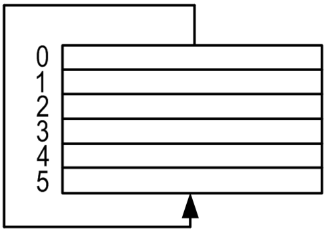
- ROL_ARR: performs a rotate shift of n positions from top to bottom of the elements in a floating word table

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 *(see page 17)* 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, immediate value`	`%MFi:L`
Double word tables	`%MWi, 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 *(see 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 *(see page 17)* 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, immediate value	%MDi:L
Floating word tables	%MWi, 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 *(see 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 the linear interpolation rule, as defined in this equation:

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

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

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

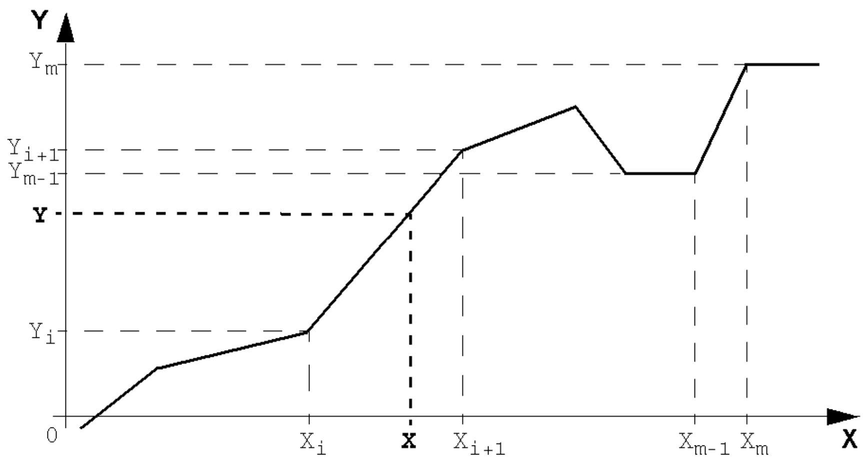
NOTE: If any of two consecutive Xi values are equal ($X_i = X_{i+1} = X$), equation (1) yields an invalid exception. In this case, to cope with this exception the following algorithm is used in place of equation (1):

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

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 *(see page 17)* 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, $X_m < X_{m-1}$

Op1 (%MWi)	Description
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 *(see page 119)*.

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: $X_1 \le Op2 \le X_m$.

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 *(see 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 *(see page 17)* 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 3.8
Instructions on I/O Objects

Aim of This Section

This section describes the instructions on I/O objects.

What Is in This Section?

This section contains the following topics:

Read Immediate Digital Embedded Input (READ_IMM_IN)

Introduction

The `READ_IMM_IN` instruction reads a digital embedded input 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 (inputs integrated into the logic controller).

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions *(see page 17)* 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 *(see 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 a digital embedded output immediately, the value is 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 (outputs integrated into the logic controller).

Syntax

The following describes Instruction List syntax. You can insert Instruction List operations and assignment instructions *(see page 17)* 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 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_IN(%MW4)]`

NOTE: Refer to the reversibility procedure *(see page 14)* to obtain the equivalent Ladder Diagram.

Chapter 4
Software Objects

What Is in This Chapter?

This chapter contains the following sections:

Section 4.1
Using Function Blocks

What Is in This Section?

This section contains the following topics:

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.

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 program:
1. Insert *(see page 131)* the function block into a rung,
2. Wire the inputs and outputs as necessary,
3. Configure *(see page 133)* 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).

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 *(see 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Adding a Function Block

To Insert a Function Block Into a Ladder Diagram Program

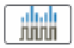
Follow this procedure:

Step	Action
1	Create a new Ladder Diagram rung in the programming workspace of SoMachine Basic. Refer to the SoMachine 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.

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`
123	`Counter`
1123	`Fast Counter`
11123	`High Speed Counter`
	`Drum Register`
	`Pulse`

Function Block Object	Description
	Pulse Width Modulation
	Message
	Pulse Train Output **NOTE:** For a complete list of PTO objects, refer to the M221 Advanced Functions Library Guide, PTO Function Blocks.
	Communication function blocks **NOTE:** For a complete list of communication function blocks, refer to Communication Objects *(see page 201)*.

Configuring a Function Block

To Configure a Function Block in a Ladder Diagram Program

Follow this procedure:

Step	Action
1	Click the **[Address]** label 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 the **[Enter comment]** label within the function block, type a short description of the function block. For example, **Pulse Timer**.
3	Optionally, click the **[Symbol]** label 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 SoMachine Basic Operating Guide for details on using symbols.
4	Configure the available parameters of each function block, as described in the “Parameters” topic of individual function block descriptions.

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

Section 4.2
Timer (%TM)

Using Timer Function Blocks

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

What Is in This Section?

This section contains the following topics:

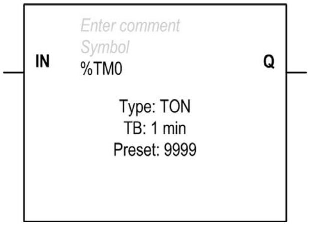
Description

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 *(see page 133)* and read the description of Memory Allocation Modes in the SoMachine Basic Operating Guide.

The `Timer` function block has the following parameters:

Parameter	Description	Value
Used	Address used	If selected, this address is currently in use in a program.
Address	`Timer` object address (%TMi)	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.
Symbol	Symbol	The symbol associated with this object. Refer to the SoMachine Basic Operating Guide, Defining and Using Symbols for details.
Type	`Timer` type	One of the following: ● TON *(see page 137)*: Timer on-Delay (default) ● TOF *(see page 138)*: Timer off-Delay ● TP *(see page 139)*: Pulse timer (monostable)
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** (only for the first 6 instances) ● **10 ms** ● **100 ms** ● **1 sec** ● **1 min**; (default)
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`.
Comment	Comment	A comment can be associated with this object.

Objects

The `Timer` function block has the following objects:

Object	Description	Value
`%TMi.P`	Preset value	See description in Parameters table above.
`%TMi.V`	Current 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.

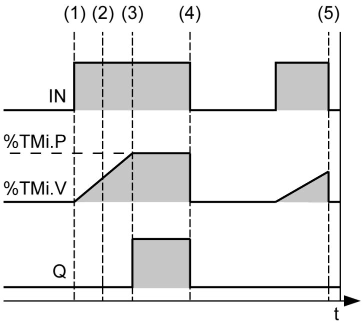
TON: On-Delay Timer

Introduction

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

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 current 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 current 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. %TMi.V is set to 0

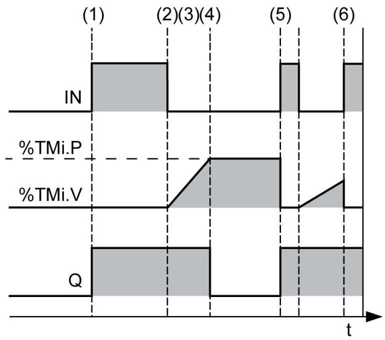
TOF: Off-Delay Timer

Introduction

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

Timing Diagram

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



(1) At a rising edge of `IN` input, `%TMi.Q` is set to 1
(2) The `Timer` starts on the falling edge of input `IN`
(3) The current 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 current 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

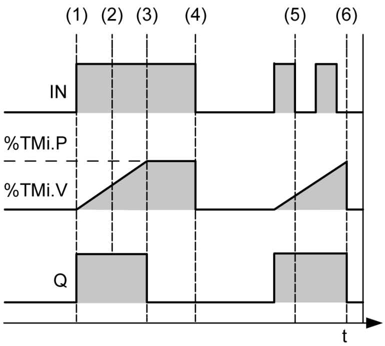
TP: Pulse Timer

Introduction

The TP (Pulse Timer) type of `Timer` is used to create pulses of a precise duration. This delay is programmable using the software.

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

Programming Example

Introduction

`Timer` function blocks have the following operating modes:

- TON (Timer On-Delay) *(see page 137)*: used to specify a period of time between a specified input being activated and an output sensor being switched on.
- TOF (Timer Off-Delay) *(see page 138)*: 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) *(see page 139)*: 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 4.3
LIFO/FIFO Register (%R)

Using LIFO/FIFO Register Function Blocks

This section provides descriptions and programming guidelines for using `LIFO/FIFO Register` function blocks.

What Is in This Section?

This section contains the following topics:

Description

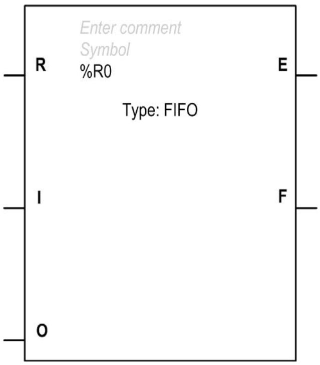
Introduction

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 *(see page 133)* and read the description of Memory Allocation Modes in the SoMachine Basic Operating Guide.

The LIFO/FIFO 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	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.
Symbol	Symbol	The symbol associated with this object. Refer to the SoMachine Basic Operating Guide, Defining and Using Symbols for details.
Type	LIFO/FIFO Register type	**FIFO** (queue) or **LIFO** (stack).
Comment	Comment	A comment can be associated with this object.

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.

NOTE: Effect of **INIT** is the same as %S0=1.

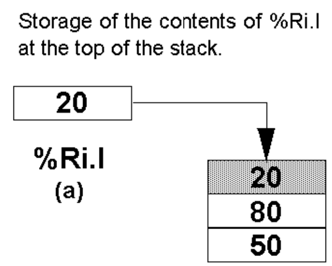
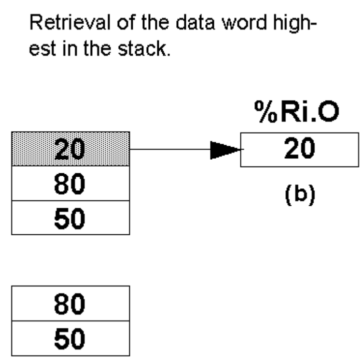
LIFO Register Operation

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 (Fig. a). When the stack is full (output `F`=1), no further storage is possible.	
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` (Fig. b). 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.	
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).	–

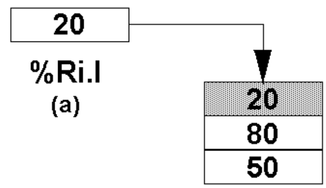
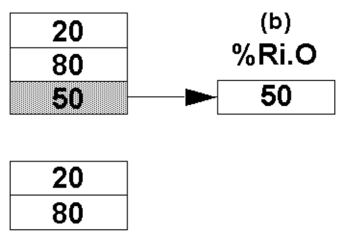
FIFO Register Operation

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 (Fig. a). 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 queue. 
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 (Fig. b). 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 4.4
Shift Bit Register (%SBR)

Using Shift Bit Register Function Blocks

This section provides descriptions and programming guidelines for using `Shift Bit Register` function blocks.

What Is in This Section?

This section contains the following topics:

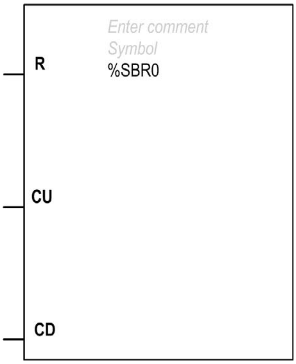
Description

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 current value of the `Shift Bit Register` is displayed in the centre of the function block:

- Decimal value eg 7
- Binary value eg 111
- Hex value eg 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 *(see page 133)* and read the description of Memory Allocation Modes in the SoMachine 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 SoMachine Basic Operating Guide, Defining and Using Symbols for details.
Comment	Comment	A comment can be associated with this object.

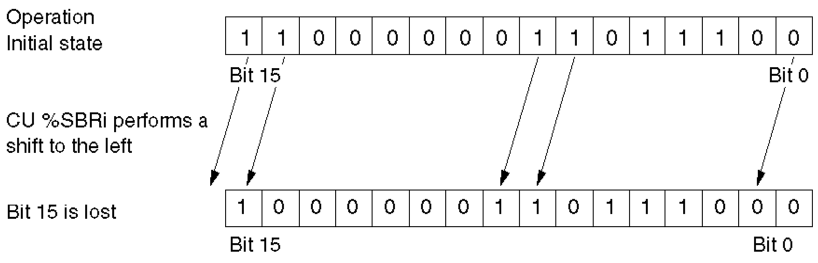
Objects

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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 4.5
Step Counter (%SC)

Using Step Counter Function Blocks

This section provides descriptions and programming guidelines for using `Step Counter` function blocks.

What Is in This Section?

This section contains the following topics:

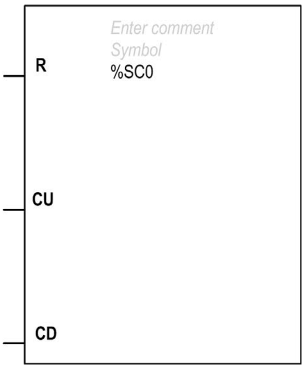
Description

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 *(see page 133)* and read the description of Memory Allocation Modes in the SoMachine 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 SoMachine 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:

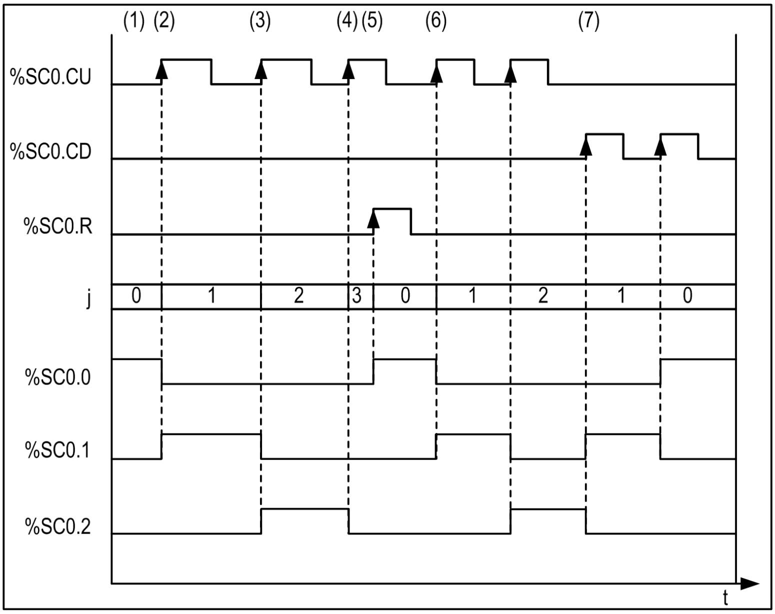
Rung	Non-Reversible Instruction
0	LD %SC0.3 OR %I0.3 R %SC0
1	LD %I0.2 CU %SC0
2	LD %I0.1 CD %SC0
3	LD %SC0.0 ST %Q0.1

Rung	Non-Reversible Instruction
4	`LD %SC0.1` `ST %Q0.2`
5	`LD %SC0.2` `ST %Q0.3`

NOTE: Refer to the reversibility procedure *(see 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 CPU cycle
(5) When **Reset** is active, the current step is set to 0 and the reset input is set to 0 after one CPU 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

Section 4.6
Counter (%C)

Using Counter Function Blocks

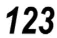
This section provides descriptions and programming guidelines for using Counter function blocks.

What Is in This Section?

This section contains the following topics:

Description

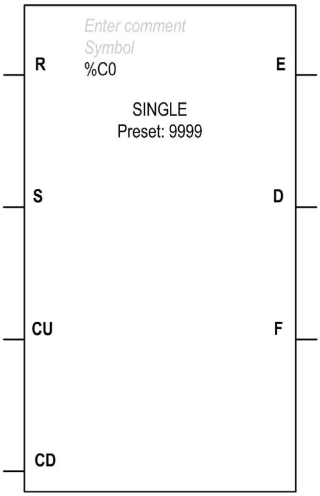
Introduction

The `Counter` function block 

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 *(see page 133)* and read the description of Memory Allocation Modes in the SoMachine Basic Operating Guide.

The `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	`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.
Symbol	Symbol	The symbol associated with this object. Refer to the SoMachine Basic Operating Guide, Defining and Using Symbols for details.
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.
Comment	Comment	A comment can be associated with this object.

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 *(see page 159)*). Can be only read. It can be modified in an animation table.
`%Ci.P`	Preset value	See Parameters table *(see page 161)*. It can be modified in an animation table.
`%Ci.E`	Empty	See Outputs table *(see page 160)*. It can be modified in an animation table.
`%Ci.D`	Done	See Outputs table *(see page 160)*. It can be modified in an animation table.
`%Ci.F`	Full	See Outputs table *(see page 160)*. 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

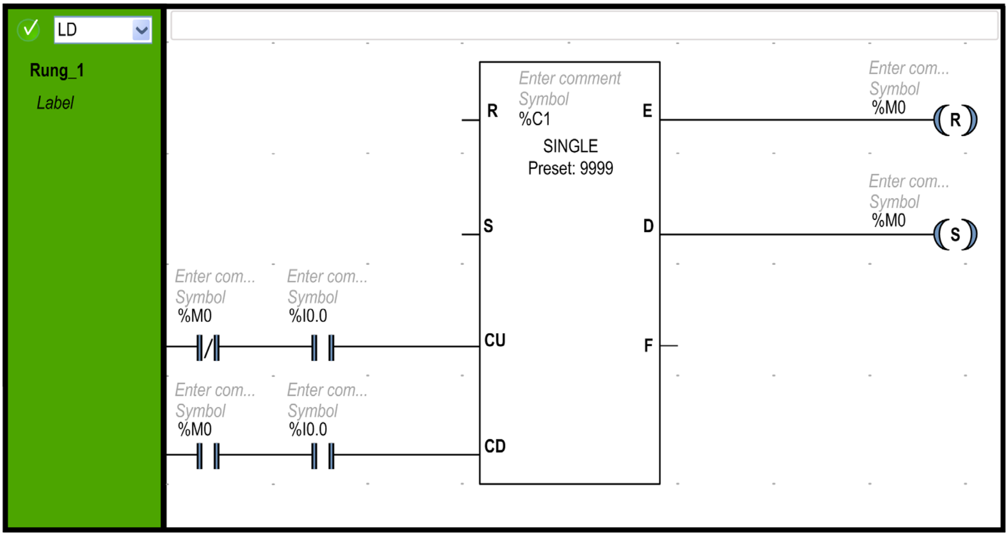
Configuration

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).

Section 4.7
Fast Counter (%FC)

Fast Counter

Overview

Refer to the *Advanced Functions Library Guide* of your controller.

Section 4.8
High Speed Counter (%HSC)

High Speed Counter

Overview

Refer to the *Advanced Functions Library Guide* of your controller.

Section 4.9
Drum Register (%DR)

Using Drum Register Function Blocks

This section provides descriptions and programming guidelines for using Drum Register function blocks.

What Is in This Section?

This section contains the following topics:

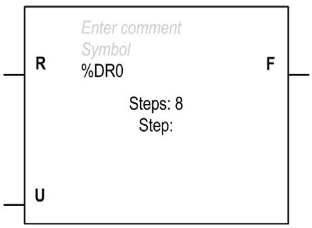
Description

Introduction

The `Drum Register` function block operates on a principle similar to an electromechanical `Drum Register` which changes step according to external events. On each step, the high point of a cam gives a command which is executed by the controller. In the case of a `Drum Register` 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 Register` 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 Register` function block has the following inputs:

Label	Description	Value
R	To return to step 0 (or instruction)	At state 1, sets the **Drum Register** to step 0.
U	Advance input (or instruction)	On a rising edge, causes the **Drum Register** to advance by 1 step and updates the control bits.

Outputs

The `Drum Register` 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 *(see page 133)* and read the description of Memory Allocation Modes in the SoMachine Basic Operating Guide.

The `Drum 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	`Drum Register` object address	A program can contain only a limited number of `Drum Register` objects. Refer to the *Programming Guide* of your controller for the maximum number of `Drum Register`.
Symbol	Symbol	The symbol associated with this object. Refer to the SoMachine Basic Operating Guide, Defining and Using Symbols for details.
Configuration	Drum assistant	**Number of steps**: 1...8. Outputs or memory bits associated with the steps: **Bit0 ... Bit15**.
Comment	Comment	A comment can be associated with this object.

Objects

The `Drum Register` function block has the following object:

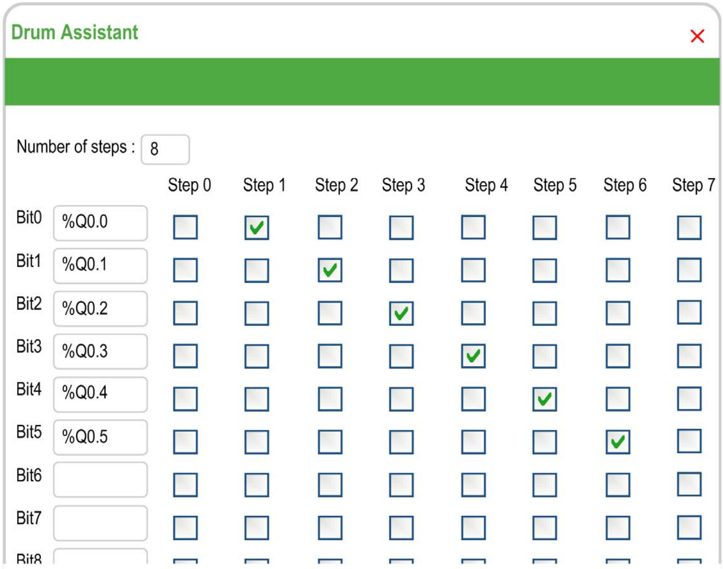
Object	Description	Value
`%DRi.S`	Current step number	0<=`%DRi.S`<=7. Word which can be read and written. Written value must be a decimal immediate value. When written, the effect takes place on the next execution of the function block. It can be modified in an animation table.
`%DRi.F`	Full	See Outputs table *(see page 168)*.

Operation

The `Drum Register` 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.

This example summarizes the main characteristics of the `Drum Register`:



NOTE: The configuration can also be realized on memory bits (%Mi).

Rung	Instruction

0	`BLK    %DR0`
	`LD     %M10`
	`R`
	`LD     %M11`
	`U`
	`END_BLK`

Create the following entries in an animation table: `%M10, %M11, %DR0, %Q0.0` to `%Q0.5`.

Observe the values of `%DR0.S, %DR0.F, %Q0.0` to `%Q0.5` when changing the value of `%M11` (evolution of the Drum), the value of `%M10` (reset of Drum). Then look at the case of overflow (it returns to step 0).

Special Cases

This table contains a list of special cases for Drum Register operation:

Special Case	Description
Effects of a cold restart (%S0=1)	Resets the Drum Register 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 Register 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 Register 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 *(see page 174)*).

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 register output **F** (%DRi.F = 0)
- Current step number (%DRi.S = 0)

Programming

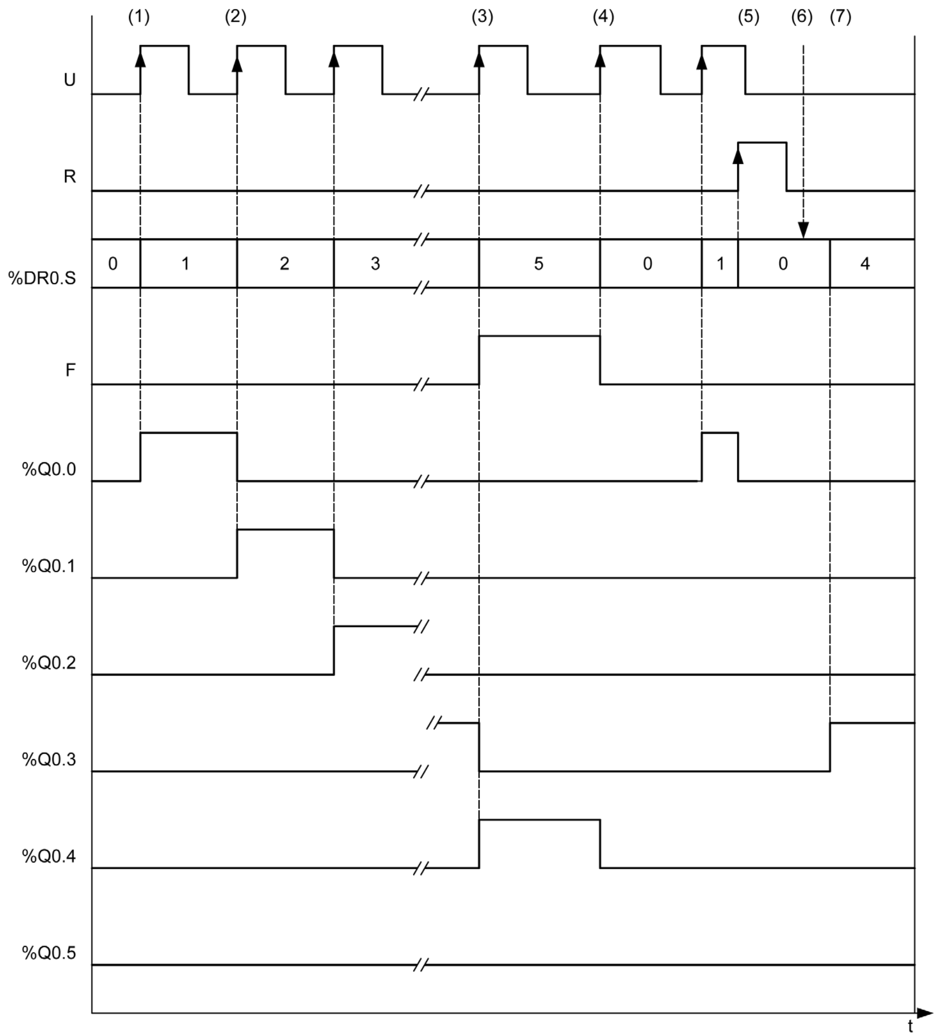
This example is a Drum Register function block program:

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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Timing Diagram

This diagram illustrates the operation of the `Drum Register`:



(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 Register` 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`

Section 4.10
Pulse (%PLS)

Pulse

Overview

Refer to the *Advanced Functions Library Guide* of your controller.

Section 4.11
Pulse Width Modulation (%PWM)

Pulse Width Modulation

Overview

Refer to the *Advanced Functions Library Guide* of your controller.

Section 4.12
Message (%MSG) and Exchange (EXCH)

Using Message Function Blocks

This section provides descriptions and programming guidelines for using `Message` function blocks.

What Is in This Section?

This section contains the following topics:

Overview

Introduction

A logic controller can be configured to communicate in Modbus protocol or can send and/or receive messages in character mode (ASCII).

SoMachine 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 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	Exchange Instruction	Message Function Block
2 serial lines	EXCH1	%MSG1
	EXCH2	%MSG2
1 serial line and 1 Ethernet	EXCH1	%MSG1
	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 *(see page 184)*. 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 simple character mode protocol to transmit and/or receive data with a simple 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 all 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 simple 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:

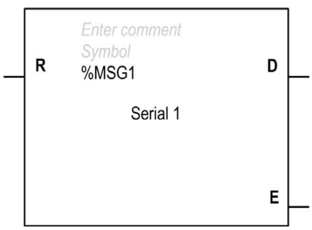
- Communications error checking:
  Error checking verifies the size of each **Exchange** table, and verifies the validity of the exchange related to the configuration.
- Coordination of multiple messages:
  To ensure coordination when sending multiple messages, the `Message` function block provides the 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	State 1: reinitializes communication; `%MSGx.E = 0` and `%MSGx.D = 1`. State 0: normal mode.

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 is too great (> 250) 2 - transmission table too small 3 - word table too small 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 caused 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`.

System word	Function	Description
%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 - no such IP address 102 - the TCP connection is broken 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 - unknown index (remote device is not indexed in configuration table) 121 - unrecoverable (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 word %SW63 contains the error code for port 1, and %SW64 contains the error code for port 2. Refer to the System Words chapter of your logic controller Function Library 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 a problem exists in sending a Modbus message to a Modbus device. In this case, you should check 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 checked only at power-up or reset
- Port 2 (for Modbus protocol) presence and configuration (RS-485) are checked at power-up or reset
- **Any message processing on port 1 is aborted** when SoMachine 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.
- 256 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 (250 max.), 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 max.), 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 always 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 are filled in prior to executing the **Exchange** instruction, and can be of type `%KW` or `%MW`. 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** instruction can be executed.

When in Transmit/Receive mode, the control and transmission tables are 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 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** instruction. At execution time, the logic controller determines what the data link layer is, and performs all 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 theregister 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 %MSG2.D`
	`AND %M0`
	`[EXCH2%MW2:5]`
	`R %M0`
		`%MSG2.D`: Detects whether the port is busy or not and thereby manages coordination of multiple messages.

2	`LDR %I0.0`
	`AND %MSG2.D`
	`[EXCH2%MW2:5]`
	`S %M0`
		–

NOTE: Refer to the reversibility procedure *(see 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 *(see 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Use SoMachine 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 checked 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, check 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 sensed 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)
(1) This byte also receives the length of the string transmitted after response.			

	Table Index	Most Significant Byte	Least Significant Byte
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 ensures 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)

	Table Index	Most Significant Byte	Least Significant Byte
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)

	Table Index	Most Significant Byte	Least Significant Byte
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#0106]`
	`[%MW801:=16#0000]`
	`[%MW802:=16#032B]`
	`[%MW803:=16#0E01]`
	`[%MW804:=16#0000]`
		`[%MW800:=16#0106]`: 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Using SoMachine 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 checks the communication bit associated with `%MSG2`. Finally, the error status of the `%MSG2` is sensed and stored on the first output bit on the local base controller I/O. Additional error checking 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 check 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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Using SoMachine 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 checks the communication bit associated with %MSG2. Finally, the error status of the %MSG2 is sensed and stored on the first output bit on the local base controller I/O. Additional detected error checking 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.

Section 4.13
Pulse Train Output (%PTO)

Pulse Train Output

Overview

Refer to the *Advanced Functions Library Guide* of your controller.

Chapter 5
Communication Objects

Introduction

The communication function blocks are used for communication with Modbus devices and send/receive messages in character mode (ASCII).

NOTE: Do not use the `EXCH` instruction (with the `%MSG` function block) concurrently with the communication function blocks.

What Is in This Chapter?

This chapter contains the following sections:

Section 5.1
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.

What Is in This Section?

This section contains the following topics:

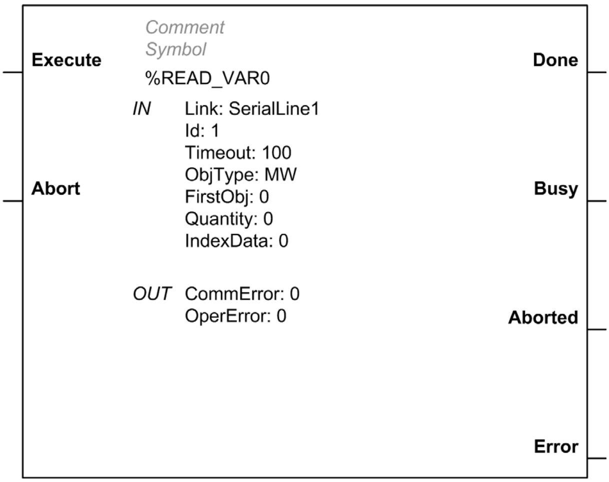
Description

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 *(see page 204)* and the Operation Error Codes *(see page 205)* .

Communication Error Codes

This table describes the error codes written to the %READ_VARi.CommError object:

Name	Detected error code	Description
CommunicationOK	00 hex	Exchange is correct.
TimedOut	01 hex	Exchange stopped because timeout expired.
Abort	02 hex	Exchange stopped on user request (**Abort** input).
BadAddress	03 hex	Address format is incorrect.
BadRemoteAddr	04 hex	Remote address is incorrect.
BadMgtTable	05 hex	Management table format is incorrect.
BadParameters	06 hex	Specific parameters are incorrect.
ProblemSendingRq	07 hex	Unsuccessful sending request to destination.
RecvBufferTooSmall	09 hex	Reception buffer size is too small.
SendBufferTooSmall	0A hex	Transmission buffer size is too small.
SystemResourceMissing	0B hex	System resource is missing.
BadLength	0E hex	Length is incorrect.
ProtocolSpecificError	FE hex	Indicates a Modbus protocol error. For more details, refer to Operation Error Codes. *(see page 205)*
Refused	FF hex	Message is refused. For more details, refer to Operation Error Codes. *(see page 205)*.

Operation Error Codes

This return code is significant when the communication error code (`CommError` object) has the value:

- 00 hex (correct)
- FF hex (refused)
- FE hex (Modbus exception code)

This table describes the error codes written to the `%READ_VARi.OperError` object:

CommError	Name	Detected error code	Description
00 hex (correct)	`OperationOK`	00 hex	Exchange is correct.
	`NotProcessed`	01 hex	Request has not been processed.
	`BadResponse`	02 hex	Received response is incorrect.
FF hex (refused)	`TargetResourceMissing`	01 hex	Target system resource is missing.
	`BadLength`	05 hex	Length is incorrect.
	`CommChannelErr`	06 hex	Error detected on the communication channel.
	`BadAddr`	07 hex	Address is incorrect.
	`SystemResourceMissing`	0B hex	System resource is missing.
	`TargetCommInactive`	0C hex	Target communication function is not active.
	`TargetMissing`	0D hex	Target is absent.
	`ChannelNotConfigured`	0F hex	Channel not configured.
FE hex (Modbus exception code)	`IllegalFunction`	01 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`	02 hex	The data address received by the slave is not an authorized address for the slave.
	`IllegalDataValue`	03 hex	The value in the request data field is not an authorized value for the slave.
	`SlaveDeviceFailure`	04 hex	The slave cannot perform a requested action because of an unrecoverable error.
	`Acknowledge`	05 hex	The slave acknowledged the request but communications timed out before the slave complied.
	`SlaveDeviceBusy`	06 hex	The slave is busy processing another command.
	`MemoryParityError`	08 hex	The slave detects a parity error in the memory when attempting to read extended memory.
	`GatewayPathUnavailable`	0A hex	The gateway is overloaded or not correctly configured.
	`GatewayTargetDeviceFailedToRespond`	0B hex	The slave is not present on the network.

Function Configuration

Properties

Double-click on the function block to open the function properties table.

The `%READ_VAR` function block has the following properties:

Property	Value	Description
Used	Activated / deactivated checkbox	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 table *(see Modicon M221, Logic Controller, Programming Guide)*.
Symbol	User-defined text	The symbol uniquely identifies this object. For details, refer to the SoMachine Basic Operating Guide (Defining and Using Symbols) *(see SoMachine 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	The unit is in ms, with a default of 100. 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 SoMachine 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	The type of objects to read: • **%MW (Mbs Fct 3)**: memory words (default) • **%I (Mbs Fct 2)**: input bits • **%Q (Mbs Fct 1)**: output bits • **%IW (Mbs Fct 4)**: input words	The types of Modbus read function codes are: • **Mbs Fct 3**: equivalent to Modbus function code 03 • **Mbs Fct 2**: equivalent to Modbus function code 02 • **Mbs Fct 1**: equivalent to Modbus function code 01 • **Mbs Fct 4**: equivalent to Modbus function code 04

Property	Value	Description
FirstObj	0...65535	Address of the first object to read.
Quantity	• 0...124 for **%MW** • 0...127 for **%I** • 0...127 for **%Q** • 0...124 for **%IW**	Number of objects to read
IndexData	0...65535	The first address of the word table to which read values are stored (`%MW`).
Comment	User-defined text	A comment to associate with this object.

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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 5.2
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.

What Is in This Section?

This section contains the following topics:

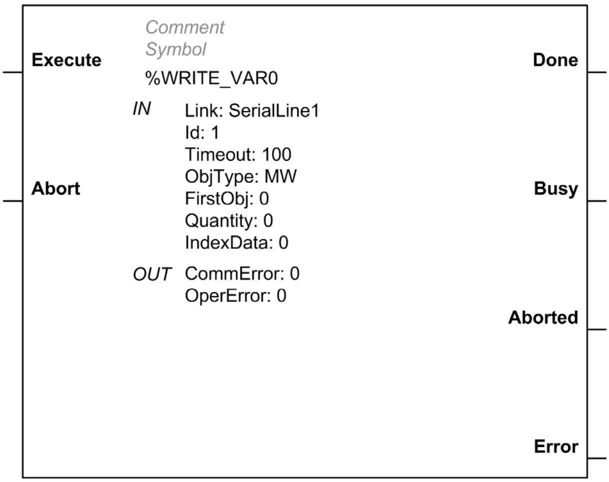
Description

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 *(see page 204)* and the Operation Error Codes *(see page 205)* .

Communication Error Codes

Refer to Communication Error Codes *(see page 204)*.

Operation Error Codes

Refer to Operation Error Codes *(see page 205)*.

Function Configuration

Properties

Double-click on the function block to open the function properties table.

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 SoMachine Basic Operating Guide (Defining and Using Symbols).
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	The unit is in ms, with a default of 100. 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 SoMachine Basic configuration screens (Modbus TCP Configuration *(see Modicon M221, Logic Controller, Programming Guide)* and Serial Line Configuration *(see Modicon M221, Logic Controller, Programming Guide)*).

Property	Value	Description
ObjType	The type of objects to write: ● **%MW (Mbs Fct 16)**: memory words (default) ● **%Q (Mbs Fct 15)**: output bits	The types of Modbus write function codes are: ● **Mbs Fct 16**: equivalent to Modbus function code 16 ● **Mbs Fct 15**: equivalent to Modbus function code 15
FirstObj	0...65535	Address of the first object from which values are used to write
Quantity	● 0...124 for **%MW** ● 0...127 for **%Q**	Number of objects to write
IndexData	0...65535	The first address of the word table to which values are to be written (`%MW`).
Comment	User-defined text	A comment to associate with this object.

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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 5.3
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.

What Is in This Section?

This section contains the following topics:

Description

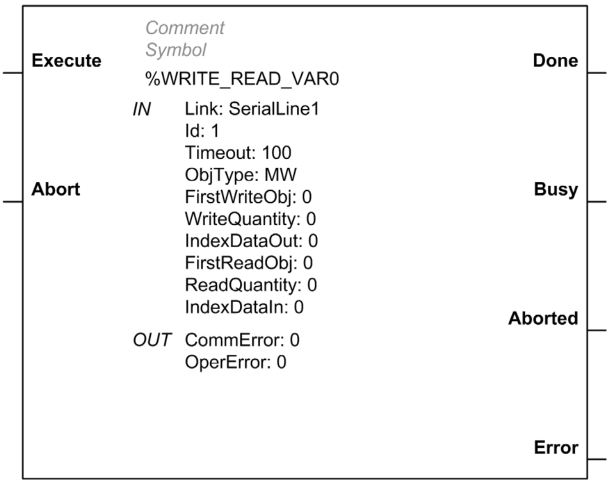
Introduction

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 *(see page 204)* and the Operation Error Codes *(see page 205)* .

Communication Error Codes

Refer to Communication Error Codes *(see page 204)*.

Operation Error Codes

Refer to Operation Error Codes *(see page 205)*.

Function Configuration

Properties

Double-click on the function block to open the function properties table.

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 SoMachine Basic Operating Guide (Defining and Using Symbols) *(see SoMachine 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	The unit is in ms, with a default of 100. 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 SoMachine Basic configuration screens (Modbus TCP Configuration *(see Modicon M221, Logic Controller, Programming Guide)* and Serial Line Configuration *(see Modicon M221, Logic Controller, Programming Guide)*).

Property	Value	Description
ObjType	**%MW (Mbs Fct 23)**: memory words	The type of Modbus read/write function code is **Mbs Fct 23**, which is equivalent to Modbus function code 23.
FirstWriteObj	0...65535	Address of the first object from which values are used to write
WriteQuantity	0...120	Number of objects to write
IndexDataOut	0...65535	The first address of the word table to which values are to be written (%MW).
FirstReadObj	0...65535	Address of the first object to read
ReadQuantity	0...124	Number of objects to read
IndexDataIn	0...65535	The first address of the word table to which read values are stored (%MW).
Comment	User-defined text	A comment to associate with this object.

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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Section 5.4
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.

What Is in This Section?

This section contains the following topics:

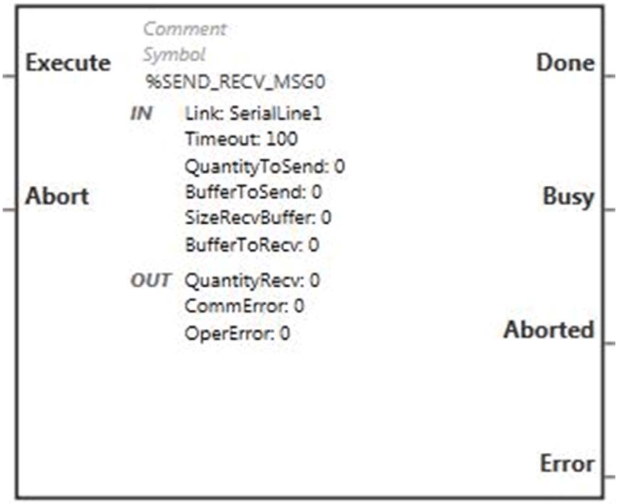
Description

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 *(see page 204)* and the Operation Error Codes *(see page 205)* .

Communication Error Codes

Refer to Communication Error Codes *(see page 204)*.

Operation Error Codes

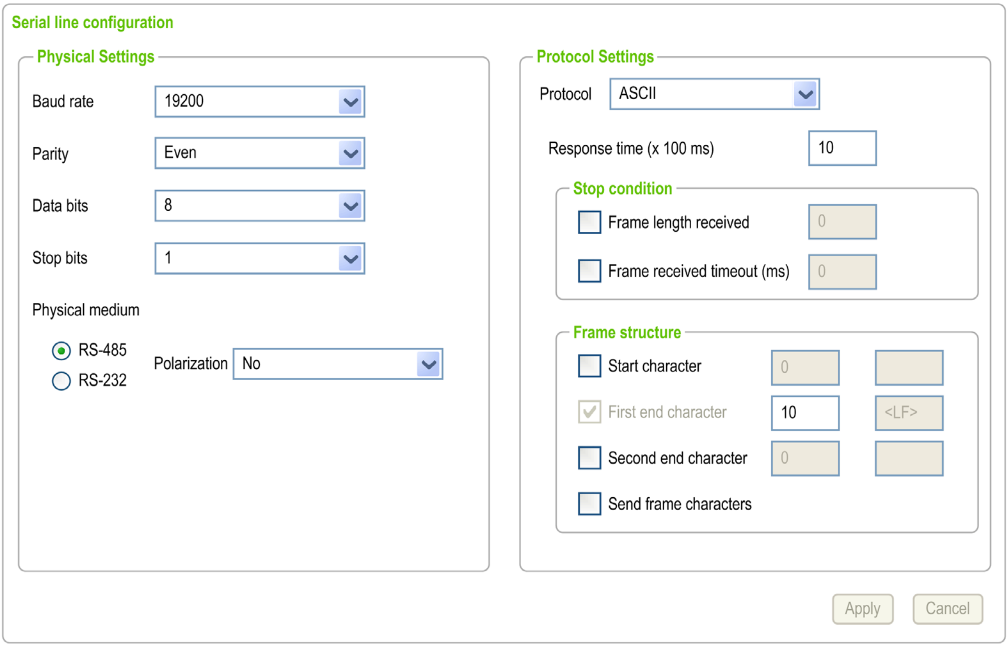
Refer to Operation Error Codes *(see page 205)*.

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 satisfied. 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 satisfied (same as receive-only).

Function Configuration

Properties

Double-click on the function block to open the function properties table.

The `%SEND_RECV_MSG` function block has the following properties:

Property	Value	Description
Used	Activated / deactivated checkbox	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 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 SoMachine Basic Operating Guide (Defining and Using Symbols) *(see SoMachine 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.
Timeout	The unit is in ms, with a default of 100. 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 SoMachine Basic configuration screens (Modbus TCP Configuration *(see Modicon M221, Logic Controller, Programming Guide)* and Serial Line Configuration *(see Modicon M221, Logic Controller, Programming Guide)*).
QuantityToSend	0...254 A value of 0 means that the function block only receives data.	Number of bytes to send
BufferToSend	0...65535	Address of the first object to send

Property	Value	Description
SizeRecvBuffer	0...254 A value of 0 means that the function block only sends data.	Available size in bytes of the receive buffer.
BufferToRecv	0...65535	The first address of the word table to which read values are stored (%MW).
QuantityRecv	0...254	Quantity of received data in bytes
Comment	User-defined text	A comment to associate with this object.

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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Chapter 6
Schedule Blocks (%SCH)

Using Schedule Blocks

This section provides descriptions and programming guidelines for using `Schedule blocks`.

What Is in This Chapter?

This chapter contains the following topics:

Description

Introduction

Schedule blocks are used to control actions at a predefined month, day, and time.

Schedule blocks are only configured in SoMachine Basic; they are not inserted into a program rung in the same way as other function blocks.

NOTE: Check 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.

Parameters

To configure parameters, follow the Configuring a Function Block procedure *(see page 133)* and read the description of Memory Allocation Modes.

The Schedule blocks has the following parameters:

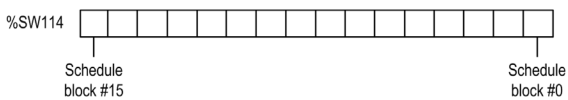
Parameter	Description	Value
Used	Address used	If selected, this address is currently in use in a program.
Address	Schedule blocks object address	A program can contain only a limited number of Schedule blocks objects. Refer to the Programming Guide of the hardware platform for the maximum number of Schedule 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**

Parameter	Description	Value
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.

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) all bits of this system word are set to 1. 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 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
Address	**Real-Time Clock 6**	`Schedule blocks` number 6
Configured	Box checked	Box checked to configure the `Schedule blocks` number 6.
Output bit	**%Q0.2**	Activate output `%Q0.2`
Start Day	**21**	Start activity on the 21 day of June
Start Month	**June**	Start activity in June
Start Time	**21**	Start activity at 21:00
End Day	**21**	Stop activity on the 21st of September
End Month	**September**	Stop activity in September
End Time	**22**	Stop activity at 22:00
Monday	Box checked	Run activity on Monday
Tuesday	Box not checked	No activity
Wednesday	Box checked	Run activity on Wednesday
Thursday	Box not checked	No activity
Friday	Box checked	Run activity on Friday
Saturday	Box not checked	No activity
Sunday	Box not checked	No activity

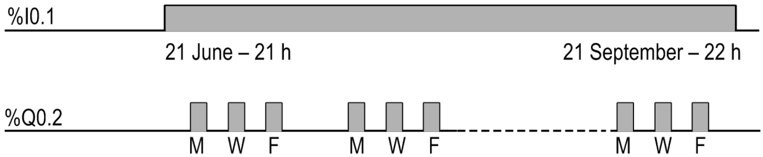
Using this program, the `Schedule blocks` can be disabled through a switch or a humidity detector wired to input `%I0.1`:

Rung	Instruction	Comment
0	`LD %I0.1` `ST %SW114:X6`	In this example, the `%SCH6` is validated.

NOTE: Refer to the reversibility procedure *(see page 14)* to obtain the equivalent Ladder Diagram.

Timing Diagram

This timing diagram shows the activation of output `%Q0.2`:



Chapter 7
PID Function

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 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

Chapter 8
Clock Functions

Overview

This chapter describes the time management functions for controllers.

What Is in This Chapter?

This chapter contains the following topics:

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 SoMachine Basic is connected to the logic controller:

- **Schedule** function blocks *(see page 229)* are used to control actions at predefined or calculated times.
- **Time/date stamping** *(see page 239)* is used to assign time and dates to events and measure event duration.

The time-of-day clock can be set by a program *(see page 239)*. 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 *(see page 73)*.

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 *(see 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 date and time settings by using one of the following methods:

- SoMachine Basic
  The user can choose between 2 modes for setting the logic controller time:
  - Manual: this mode displays a date/time picker and lets you manually choose what time to set in the logic controller.
  - Automatic: this mode displays the current time of the PC on which SoMachine Basic is running and which are used to set the time is the logic controller.

  (See the SoMachine Basic Operating Guide for details.)
- System words
  Use system words `%SW49` to `%SW53` or system word `%SW59`.

NOTE: The date and time can be set when the RTC function 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 0. Once you set the date and time, bit %S51 must then be set to 1. This results in the following:

- Cancels the update of words `%SW49` to `%SW53` via the internal clock.
- Transmits the values written in words `%SW49` to `%SW53` to the internal clock.

This table lists the system word containing current 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`
	`[%SW49:=%MW10]`
	`[%SW50:=%MW11]`
	`[%SW51:=%MW12]`
	`[%SW52:=%MW13]`
	`[%SW53:=%MW14]`
	`S %S50`
		Refer to BCD/Binary Conversion Instruction *(see page 73)*.

NOTE: Refer to the reversibility procedure *(see page 14)* to obtain the equivalent Ladder Diagram.

Words %MW10 to %MW14 contain the new date and time (see Review of BCD Code *(see page 73)*) and corresponds to the coding of words %SW49 to %SW53.

The word table must contain the new date and time:

Encoding	Most Significant Byte	Least Significant Byte
%MW10	–	Day of the week (1)
%MW11	–	Second
%MW12	Hour	Minute
%MW13	Month	Day
%MW14	Century	Year
(1) 1 = Monday, 2 = Tuesday, 3 = Wednesday, 4 = Thursday, 5 = Friday, 6 = Saturday, 7 = Sunday		

Example data for Monday 03 June 2013:

Word	Value (hex)	Meaning
%MW10	0001	Monday
%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.

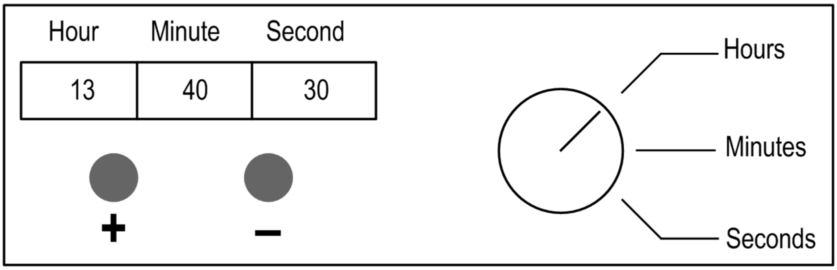
This table describes each bit of the system word %SW59 for adjusting date and time parameters:

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 *(see page 14)* to obtain the equivalent Ladder Diagram.

Glossary

!

%

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.

%KW

According to the IEC standard, %KW represents a constant word.

%MW

According to the IEC standard, %MW represents a memory word register (for example, a language object of type memory word).

%Q

According to the IEC standard, %Q represents an output bit (for example, a language object of type digital OUT).

A

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).

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).

Index

L

M

N

O

P

R

S

T

W

X