SoMachine

Modbus and ASCII Read/Write Functions PLCCommunication Library Guide

09/2016

EIO0000000361.10

www.schneider-electric.com

Schneider Electric

The information provided in this documentation contains general descriptions and/or technical characteristics of the performance of the products contained herein. This documentation is not intended as a substitute for and is not to be used for determining suitability or reliability of these products for specific user applications. It is the duty of any such user or integrator to perform the appropriate and complete risk analysis, evaluation and testing of the products with respect to the relevant specific application or use thereof. Neither Schneider Electric nor any of its affiliates or subsidiaries shall be responsible or liable for misuse of the information contained herein. If you have any suggestions for improvements or amendments or have found errors in this publication, please notify us.

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

© 2016 Schneider Electric. All rights reserved.

Table of Contents

	Safety Information	5
	About the Book	7
Chapter 1	**Communication Principles**	11
	Communication Functions on Controllers	12
	Generic Parameters	13
Chapter 2	**Data Types**	17
	CommunicationErrorCodes: Communication Error Codes	18
	OperationErrorCodes: Operation Error Codes	19
	LinkNumber: Communication Port Number	20
	ObjectType: Available Object Types to be Read/Written	21
	ADDRESS: External Device Address	22
	ADDR_EXT: Address Extension	23
	TCP_ADDR_EXT: Address Extension for TCP Devices	24
Chapter 3	**Function Block Descriptions**	25
	`ADDM`: Convert a String into an Address	26
	`READ_VAR`: Read Data from a Modbus Device	31
	`WRITE_VAR`: Write Data to a Modbus Device	33
	`WRITE_READ_VAR`: Read and Write Internal Registers on a Modbus Device	35
	`SINGLE_WRITE`: Write a Single Register to a Modbus Device	38
	SEND_RECV_MSG: Send and/or Receive User Defined Messages	40
Appendices		45
Appendix A	**Function and Function Block Representation**	47
	Differences Between a Function and a Function Block	48
	How to Use a Function or a Function Block in IL Language	49
	How to Use a Function or a Function Block in ST Language	52
Glossary		55
Index		59

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, service, or maintain it. The following special messages may appear throughout this documentation or on the equipment to warn of potential hazards or to call attention to information that clarifies or simplifies a procedure.

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

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

⚠ DANGER

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

⚠ WARNING

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

⚠ CAUTION

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

NOTICE

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

PLEASE NOTE

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

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

About the Book

At a Glance

Document Scope

This document describes the PLCCommunication library for controllers.

Validity Note

This document has been updated for the release of SoMachine V4.2.

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.

Terminology Derived from Standards

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

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

Among others, these standards include:

Standard	Description
EN 61131-2:2007	Programmable controllers, part 2: Equipment requirements and tests.
ISO 13849-1:2008	Safety of machinery: Safety related parts of control systems. General principles for design.
EN 61496-1:2013	Safety of machinery: Electro-sensitive protective equipment. Part 1: General requirements and tests.
ISO 12100:2010	Safety of machinery - General principles for design - Risk assessment and risk reduction
EN 60204-1:2006	Safety of machinery - Electrical equipment of machines - Part 1: General requirements
EN 1088:2008 ISO 14119:2013	Safety of machinery - Interlocking devices associated with guards - Principles for design and selection
ISO 13850:2006	Safety of machinery - Emergency stop - Principles for design
EN/IEC 62061:2005	Safety of machinery - Functional safety of safety-related electrical, electronic, and electronic programmable control systems
IEC 61508-1:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: General requirements.
IEC 61508-2:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: Requirements for electrical/electronic/programmable electronic safety-related systems.
IEC 61508-3:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: Software requirements.
IEC 61784-3:2008	Digital data communication for measurement and control: Functional safety field buses.
2006/42/EC	Machinery Directive
2014/30/EU	Electromagnetic Compatibility Directive
2014/35/EU	Low Voltage Directive

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

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

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

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

Chapter 1
Communication Principles

Introduction

The communication function blocks for the controller are located in the PLCCommunication library. This library is automatically added to the library manager when a controller with Ethernet connectivity is added to your project or when a Modbus or ASCII manager is added to a controller's serial line.

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Communication Functions on Controllers	12
Generic Parameters	13

Communication Functions on Controllers

Introduction

This topic describes management and operations of the controllers' communication functions. The functions facilitate communications between specific devices. Most of the functions are dedicated to Modbus exchanges. One function (`SEND_RECV_MSG`) is used by an ASCII manager to manage the exchange of data between devices operating on other protocols than Modbus.

NOTE: Communication functions are processed asynchronously with regard to the application task that called the function.

NOTE: This library is not supported by the ATV IMC Drive Controller.

NOTE: Do not use function blocks of the PLCCommunication library on a serial line with a Modbus IOScanner configured. This disrupts the Modbus IOScanner exchange.

NOTE: The Ethernet port of the HMI SCU controller is not supported for Modbus communication.

NOTE: The HMI SCU controller do not allow adding the **ASCII Manager** node under COM1 node.

Available Function Blocks

This table describes the communication function blocks available to controllers:

Function	Description
ADDM *(see page 26)*	This function takes the destination address of an external device and converts its string representation to an ADDRESS structure.
READ_VAR *(see page 31)*	This function reads standard bits or registers from a Modbus device.
WRITE_VAR *(see page 33)*	This function writes standard bits or registers to a Modbus device.
WRITE_READ_VAR *(see page 35)*	This function reads and writes standard bits or registers on Modbus devices.
SINGLE_WRITE *(see page 38)*	This function writes a single register to an external device.
SEND_RECV_MSG *(see page 40)*	This function sends and receives user defined messages on selected media for example, a serial line (not supported by XBT GC, XBT GT, XBT GK and HMI SCU).

Generic Parameters

Introduction

This topic describes the management and operations of the controllers' communication functions using the READ_VAR function block as an example. (The PLCopen standard defines rules for function blocks.)

NOTE: These parameters are common to all PLCCommunication function blocks (except ADDM).

Graphical Representation

The parameters that are common to all function blocks in the PLCCommunication library are highlighted in this graphic:

Common Parameters

These parameters are shared by several function blocks in the PLCCommunication library:

Input	Type	Comment
Execute	BOOL	The function is executed on the rising edge of this input. **NOTE:** When xExecute is set to TRUE at the first task cycle in RUN after a cold or warm reset, the rising edge is not detected.
Abort	BOOL	Aborts the ongoing operation at the rising edge
Addr	ADDRESS	Address of the targeted external device (can be the output of the ADDM function block *(see page 26)*)
Timeout	WORD	Exchange timeout is a multiple of 100 ms (0 for infinite) **NOTE:** The Timeout time is fixed at ≅1 s for the HMI SCU and cannot be set for the Modbus communication function blocks.
NOTE: A function block operation may require several exchanges. The timeout applies to each exchange between the controller and the modem, so the global duration of the function block might be longer than the timeout.		

Output	Type	Comment
Done	BOOL	Done is set to TRUE when the function is completed successfully.

Output	Type	Comment
Busy	BOOL	Busy is set to TRUE while the function is ongoing.
Aborted	BOOL	Aborted is set to TRUE when the function is aborted with the Abort input. When the function is aborted, CommError contains the code Canceled - 16#02 (exchange stopped by a user request).
Error	BOOL	Error is set to TRUE when the function stops because of a detected error. When there is a detected error, CommError and OperError contain information about the detected error.
CommError	BYTE	CommError contains communication error codes *(see page 18)*.
OperError	DWORD	OperError contains operation error codes *(see page 19)*.

NOTE:
As soon as the Busy output is reset to 0, one (and only one) of these 3 outputs is set to 1:

- Done
- Error
- Aborted

Function blocks require a rising edge in order to be initiated. The function block needs to first see the Execute input as false in order to detect a subsequent rising edge.

⚠ WARNING

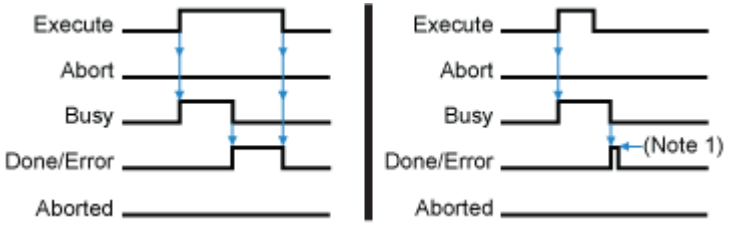
UNINTENDED EQUIPMENT OPERATION

Always make the first call to a function block with its Execute input set to FALSE so that it will detect a subsequent rising edge.

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

Function Execution

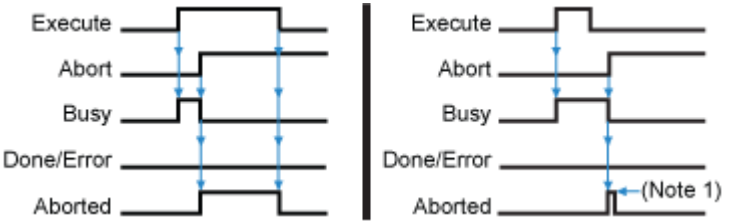
The function starts at the rising edge of the `Execute` input. The `Busy` output is then set to `TRUE`. This figure shows the function block's behavior when the operation is automatically completed (with or without detected errors):



Note 1: The `Done` or `Error` bit is set to `TRUE` during a task cycle only if `Execute` has already been reset to `FALSE` when the operation ended.

Function Aborted

This figure shows the function being aborted by the application. The rising edge of the `Abort` input cancels the ongoing function. In such cases, the aborted output is set to 1 and `CommError` contains the code `Canceled - 16#02` (exchange stopped by a user request):



Note 1: The `Abort` bit is set to `TRUE` during a task cycle only if `Execute` has already been reset to `FALSE` when the abort request occurred.

Chapter 2
Data Types

Introduction

This chapter describes the data types used by the PLCCommunication library.

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
CommunicationErrorCodes: Communication Error Codes	18
OperationErrorCodes: Operation Error Codes	19
LinkNumber: Communication Port Number	20
ObjectType: Available Object Types to be Read/Written	21
ADDRESS: External Device Address	22
ADDR_EXT: Address Extension	23
TCP_ADDR_EXT: Address Extension for TCP Devices	24

CommunicationErrorCodes: Communication Error Codes

Enumerated Type Description

The `CommunicationErrorCodes` enumerated type contains information about communication diagnostics, such as interruptions and detected errors. It contains these values:

Enumerator	Value (hex)	Description
`CommunicationOK`	`00`	The exchange is valid.
`TimedOut`	`01`	The exchange stopped when the timeout expired.
`Canceled`	`02`	The exchange was stopped by a user request (the `Abort` command).
`BadAddress`	`03`	The address format is incorrect.
`BadRemoteAddr`	`04`	The remote address is incorrect.
`BadMgtTable`	`05`	The management table format is incorrect.
`BadParameters`	`06`	Specific parameters are incorrect.
`ProblemSendingRq`	`07`	There was a problem while sending the request to the destination.
`RecvBufferTooSmall`	`09`	The reception buffer size is too small.
`SendBufferTooSmall`	`0A`	The transmission buffer size is too small.
`SystemRessourceMissing`	`0B`	A system resource is missing.
`BadTransactionNb`	`0C`	The transaction number is incorrect.
`BadLength`	`0E`	The length is incorrect.
`ProtocolSpecificError`	`FE`	The operation error code contains a protocol-specific code.
`Refused`	`FF`	The message was refused.

OperationErrorCodes: Operation Error Codes

Enumerated Type Description

The `OperationErrorCodes` enumerated type contains codes that correspond to detected errors.

00

When the communication error code is 00 hex (correct transaction), the `OperationErrorCodes` enumerated type can return these values:

Enumerator	Value (hex)	Description
`OperationOK`	`00`	The exchange is valid.
`NotProcessed_or_TargetResourceMissing`	`01`	The request has not been processed.
`BadResponse`	`02`	The received response is incorrect.

FF

When the communication error code is FF hex (message refused), the `OperationErrorCodes` enumerated type can return these values:

Enumerator	Value (hex)	Description
`NotProcessed_or_TargetResourceMissing`	`01`	The target system's resource is absent.
`BadLength`	`05`	The length is incorrect.
`CommChannelErr`	`06`	The communication channel is associated with a detected error.
`BadAddr`	`07`	The address is incorrect.
`SystemResourceMissing`	`0B`	A system resource is missing.
`TargetCommInactive`	`0C`	A target communication function is not active.
`TargetMissing`	`0D`	The target is absent.
`ChannelNotConfigured`	`0F`	The channel is not configured.

FE

When the communication error code is FE hex, the `OperationErrorCodes` enumerated type contains the protocol-specific error detection code. (Refer to your specific protocol's error detection codes.)

LinkNumber: Communication Port Number

Enumerated Type Description

The `LinkNumber` enumerated data type is a list of the available communication ports. It contains these values:

Enumerator	Value (hex)	Description
`USBConsole`	`00`	USB port not available for communication exchanges
`COM1`	`01`	Serial COM 1 (embedded serial link)
`COM2`	`02`	Serial COM 2
`EthEmbed`	`03`	Embedded Ethernet link
`CANEmbed`	`04`	Embedded CANopen link
`COM3`	`05`	Serial COM 3

If one serial PCI module is installed then the serial PCI module link is COM2, regardless of the physical PCI slots used.

If two serial PCI modules are installed then the serial PCI module plugged on the left side PCI slots is COM2 and the serial PCI module plugged on the right side PCI slots is COM3.

ObjectType: Available Object Types to be Read/Written

Enumerated Type Description

The `ObjectType` enumerated data type contains the object types that are available for reading and/or writing.

The table below lists the data type values, the corresponding object types and the Modbus request function codes associated with each function block:

			Read / Write Functions and associated Modbus Request Function Code			
Enumerator	**Value (hex)**	**Object Type)**	**READ_VAR**	**WRITE_VAR**	**SINGLE_WRITE**	**WRITE_READ_VAR**
`MW`	`00`	Holding Register (16 bits)	#3 (Read Holding Registers)	#16 (Write Multiple Registers)	#6 (Write Single Register)	#23 (Write-Read Multiple Registers)
`I`	`01`	Digital Input (1 bit)	#2 (Read Digital Inputs)	—	—	—
`Q`	`02`	Internal bit or Digital output (coil) (1 bit)	#1 (Read Coils)	#15 (Write Multiple Coils)	—	—
`IW`	`03`	Input register (16 bits)	#4 (Read Input Registers)	—	—	—

ADDRESS: External Device Address

Structure Description

The `ADDRESS` data structure contains the external device address. It contains these variables:

Variable	Type	Description
`_Type`	BYTE	Reserved
`_CliID`	BYTE	Reserved
`Rack`	BYTE	Rack number (always 0)
`Module`	BYTE	Module number (always 0)
`Link`	LinkNumber *(see page 20)*	Communication port number
`_ProtId`	BYTE	Reserved (0 for Modbus)
`AddrLen`	BYTE	Length of `Unitld` and `AddrExt` variables (in bytes)
`UnitId`	BYTE	Equipment number (e.g., Modbus slave address)
`AddrExt`	ADDR_EXT *(see page 23)*	Contains an address extension as an array or as a specific structure

ADDR_EXT: Address Extension

Union Description

`ADDR_EXT` is a UNION data type that contains an address extension as an array or as a specific structure for the TCP/IP address. It contains these variables:

Variable	Type	Description
`as_array`	ARRAY [0...7] OF BYTE	Reserved (open for other protocol addressing)
`TcpAddr`	TCP_ADDR_EXT	The specific structure for TCP remote devices

TCP_ADDR_EXT: Address Extension for TCP Devices

Structure Description

The `TCP_ADDR_EXT` structure data type contains the address extension for TCP external devices. It contains these variables:

Variable	Type	Description
`A`	BYTE	First value in IP address A.B.C.D
`B`	BYTE	Second value in IP address A.B.C.D
`C`	BYTE	Third value in IP address A.B.C.D
`D`	BYTE	Last value in IP address A.B.C.D
`port`	WORD	TCP port number (Modbus default: 502)

Chapter 3
Function Block Descriptions

Introduction

This chapter describes the function blocks in the PLCCommunication library.

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
ADDM: Convert a String into an Address	26
READ_VAR: Read Data from a Modbus Device	31
WRITE_VAR: Write Data to a Modbus Device	33
WRITE_READ_VAR: Read and Write Internal Registers on a Modbus Device	35
SINGLE_WRITE: Write a Single Register to a Modbus Device	38
SEND_RECV_MSG: Send and/or Receive User Defined Messages	40

ADDM: Convert a String into an Address

Function Description

The ADDM function block converts a destination address that is represented as a string to an ADDRESS structure. You can use the ADDRESS structure as an entry in a communication function block.

Graphical Representation

ADDM - Specific Parameter Description

Input/Output	Type	Comment
AddrTable	ADDRESS	This is the ADDRESS structure to be filled by the function block.

Input	Type	Comment
Execute	BOOL	Executes the function at the rising edge.
Addr	STRING	Address in STRING type to be converted in ADDRESS type (see details below).

Output	Type	Comment
Done	BOOL	Done is set to TRUE when the function is completed successfully. **NOTE:** When the operation is aborted with the Abort input, Done is not set to 1 (only Aborted).
Error	BOOL	Error is set to TRUE when the function stops due to detection of an error. When there is a detected error, CommError and OperError contain information about the detected error.
CommError	BYTE	CommError contains communication error codes *(see page 18)*.

NOTE: A rising edge on the Execute input executes the conversion and returns an immediate update of AddrTable. However, AddrTable retains the last value when an error is detected (that is, when the Addr string is not correct).

Function blocks require a rising edge in order to be initiated. The function block needs to first see the Execute input as FALSE in order to detect a subsequent rising edge.

⚠ WARNING
UNINTENDED EQUIPMENT OPERATION Always make the first call to a function block with its `Execute` input set to `FALSE` so that it may detect a subsequent rising edge. **Failure to follow these instructions can result in death, serious injury, or equipment damage.**

`Addr` STRING for ASCII Address Format

For ASCII addressing, only the communication port number is requested:

`'<communication port number>'`

For example, to send a user defined message on serial line 2, use the string `'2'`.

This table defines the fields in the ADDM output for the ASCII address format:

Field	Type	Value	Example
`_Type`	BYTE	Reserved	not used
`_CliID`	BYTE	Reserved	not used
`Rack`	BYTE	Rack number (always 0)	0
`Module`	BYTE	Module number (always 0)	0
`Link`	LinkNumber *(see page 20)*	`<communication port number>`	2
`_ProtId`	BYTE	Not used	not used
`AddrLen`	BYTE	0	0
`UnitId`	BYTE	Not used	not used
`AddrExt`	ADDR_EXT	Not used	not used

Addr STRING for Modbus Serial Address Format

For Modbus serial addressing, use the communication port and the destination slave address (0 to 247), separated by a dot: `'<communication port number>.<slave address>'`

For example, send a message to slave 8 on serial line 1with this syntax: `'1.8'`

The `ADDM` function fills the `AddrTable` input/output with these values:

Field	Type	Value	Example
`_Type`	BYTE	Reserved	not used
`_CliID`	BYTE	Reserved	not used
`Rack`	BYTE	Rack number (always 0)	0
`Module`	BYTE	Module number (always 0)	0

Field	Type	Value	Example
Link	LinkNumber *(see page 20)*	<communication port number>	2
_ProtId	BYTE	0 for Modbus	0
AddrLen	BYTE	1	1
UnitID	BYTE	<slave address>	8
AddrExt	ADDR_EXT	Not used	not used

Addr STRING for Modbus TCP Address Format

Address of a Modbus TCP Standard Slave

For the Modbus TCP standard slave address format, the communication port number (3 for the embedded Ethernet port) and the destination IP address {A.B.C.D} (offset with brackets) are requested:

```
'<communication port number>{<IP address A.B.C.D>}'
```

NOTE: A Modbus TCP standard slave uses Modbus address 255 (the `UnitId` default value). However, a Modbus TCP device may have different value (for example, a Tesys has Modbus address 1). In this case, add the `UnitId` value.

TCP port 502 is used by default. It's possible to use a non-standard port by adding the requested port number to the IP address:

```
'<communication port number A.B.C.D>{<IP address>:<port>}'
```

For example, to send a message at Modbus TCP slave IP address 192.168.1.2 using standard TCP port 502, use this string: `'3{192.168.1.2}'`

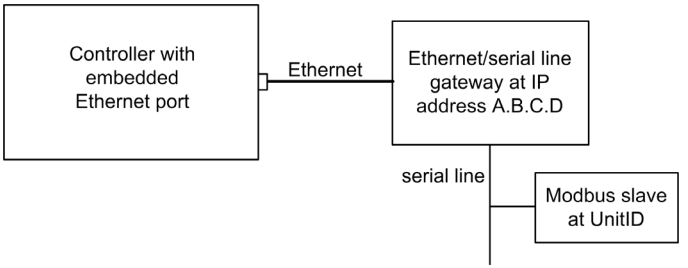
The `ADDM` function fills the `AddrTable` input/output with these values:

Field	Type	Value	Example
_Type	BYTE	Reserved	not used
_CliID	BYTE	Reserved	not used
Rack	BYTE	Rack number	0
Module	BYTE	Module number	0
Link	LinkNumber *(see page 20)*	<communication port number>	3
_ProtId	BYTE	0 for Modbus	0
AddrLen	BYTE	`UnitID + AdrExt` length in bytes	7
UnitId	BYTE	Modbus address (255 by default)	255

Field	Type	Value	Example
AddrExt	TCP_ADDR_EXT	A	192
		B	168
		C	1
		D	2
		<port> (default = 502)	502

Address of a Modbus Serial Slave Through Ethernet/Serial Line Gateway

It's also possible to address a Modbus slave through an Ethernet/serial line gateway:



The request includes the communication port number, gateway IP address {A.B.C.D} (offset with brackets with or without TCP port), and the Modbus serial slave address (UnitId parameter):

```
'<communication port number>{<IP address A.B.C.D>}<slave address>'
```

For example, to send a message at Modbus Serial slave address 5 through a Ethernet/serial line gateway at IP address 192.168.1.2 using standard TCP port 502, use this string:
`'3{192.168.1.2}5'`

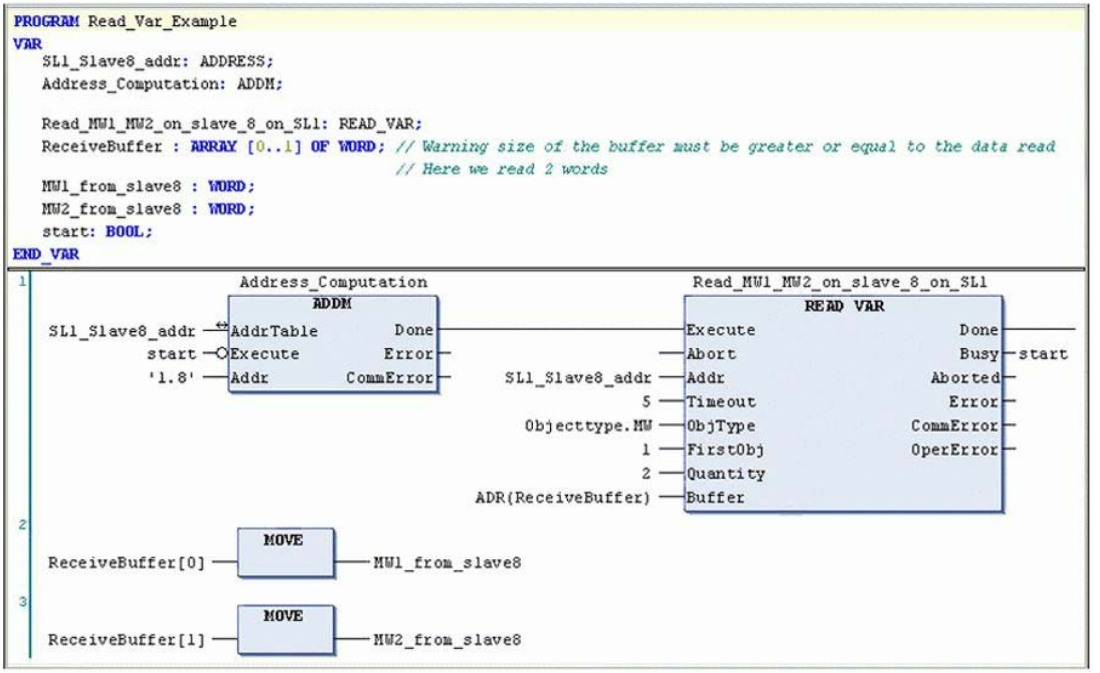
The `ADDM` function fills the `AddrTable` input/output with these values:

Field	Size	Value	Example
_Type	BYTE	Reserved	not used
_CliID	BYTE	Reserved	not used
Rack	BYTE	Rack number	0
Module	BYTE	Module number	0
Link	LinkNumber *(see page 20)*	<communication port number>	3
_ProtId	BYTE	0 for Modbus	0
AddrLen	BYTE	UnitID + AdrExt length in bytes	7
UnitId	BYTE	<Slave address>	5

Field	Size	Value	Example
AddrExt	TCP_ADDR_EXT	A	192
		B	168
		C	1
		D	2
		TCP port number (default = 502)	502

Example

This example shows the declaration and use of ADDM as an input to the READ_VAR function block. ADDM converts the address of slave 8 on serial line 1 from the string '1.8' to an ADDRESS type:



NOTE: The Busy output assigned to start allows for the continuous execution of READ_VAR. The start variable must be set to TRUE (by the user online or the application) after the first cycle to initiate continuous reading. This example does not show the management of any detected communication errors.

READ_VAR: Read Data from a Modbus Device

Function Description

The READ_VAR function block reads data from an external device in the Modbus protocol.

Graphical Representation

READ_VAR - Specific Parameter Description

Input	Type	Comment
ObjType	ObjectType	ObjType is the type of object to be read (MW, I, IW, Q) *(see page 21)*.
FirstObj	DINT	FirstObj is the index of the first object to be read.
Quantity	UINT	Quantity is the number of objects to be read: ● 1...125: registers (MW and IW types) ● 1...2000: bits (I and Q types)
Buffer	POINTER TO BYTE	Buffer is the address of the buffer in which object values are stored. The Addr standard function must be used to define the associated pointer. (See the example below.) The buffer is a table that receives the values that are read in the device. For example, the reading of 4 registers is stored in a table of 4 words and the reading of 32 bits requires a table of 2 words or 4 bytes, each bit of which is set to the corresponding value of the remote device.

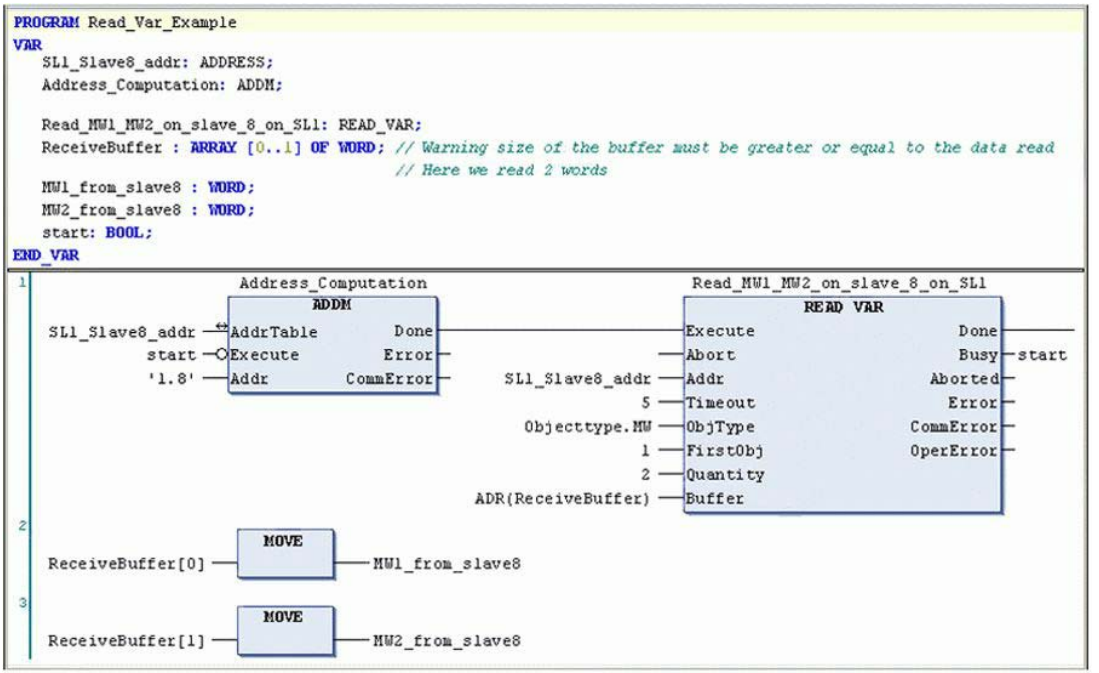
NOTE: If you execute a multiple download to HMI controllers, and if the WRITE_VAR or READ_VAR function blocks are included in the controller application, they may not operate correctly on startup after the download is completed. The HMI may stop functioning, and SoMachine may lose connection with the device for 2 to 3 minutes. For HMI controllers, whenever the READ_VAR or WRITE_VAR function blocks are used, if the conditions above occur, restart the HMI controller or wait for the HMI to recover.

The input and output parameters that are common to all PLCCommunication library function blocks are described elsewhere *(see page 13)*.

Example

This POU *(see page 56)* allows the reading of internal registers 1 and 2 (MW1 and MW2) of the Modbus slave with address 8 on serial line 1.

This figure shows the declaration and use of the `READ_VAR` function:



NOTE: The `Busy` output assigned to `start` allows for the continuous execution of ADDM and READ_VAR. The `start` variable must be set to TRUE (online by the user or by the application) after the first cycle to initiate continuous reading. This example does not show the management of exchange errors.

WRITE_VAR: Write Data to a Modbus Device

Function Description

The WRITE_VAR function block writes data to an external device in the Modbus protocol.

Graphical Representation

WRITE_VAR - Specific Parameter Description

Input	Type	Comment
ObjType	ObjectType	ObjType describes the type of object(s) to write (MW, Q) *(see page 21)*.
FirstObj	DINT	FirstObj is the index of the first object to write.
Quantity	UINT	Quantity is the number of objects to be read: ● 1...123: registers (MW type) ● 1...1968: bits (Q type)
Buffer	POINTER TO BYTE	Buffer is the address of the buffer in which object values are stored. Use the Addr standard function to define the associated pointer. (See the example below.) The buffer is a table that receives the values that have to be written in the device. For example, the written values of 4 registers are stored in a table of 4 words and the written values of 32 bits require a table of 2 words or 4 bytes, each bit of which is set to the corresponding value.

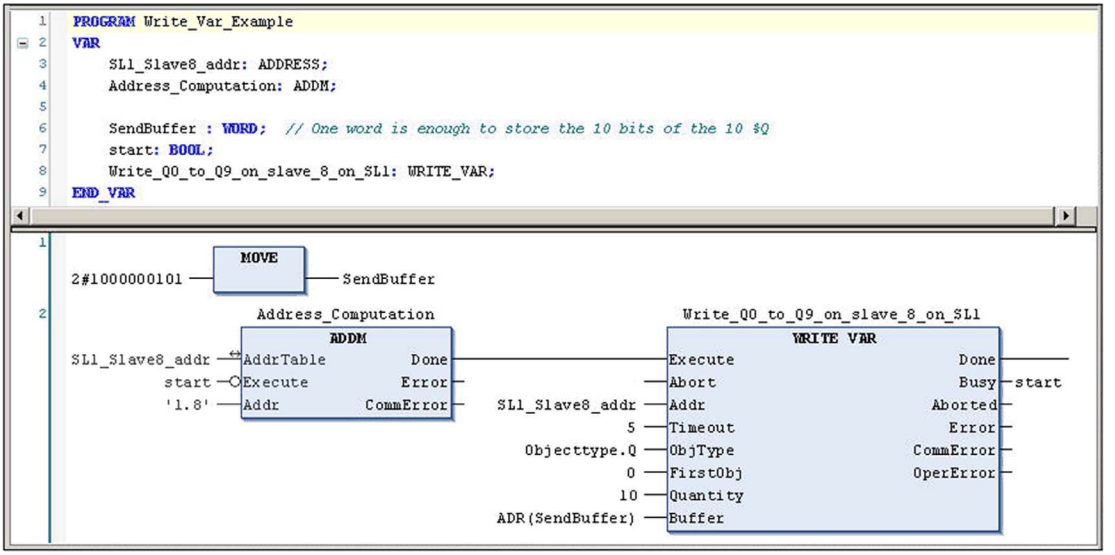
NOTE: If you execute a multiple download to HMI controllers, and if the WRITE_VAR or READ_VAR function blocks are included in the controller application, they may not operate correctly on startup after the download is completed. The HMI may stop functioning, and SoMachine may lose connection with the device for 2 to 3 minutes. For HMI controllers, whenever the READ_VAR or WRITE_VAR function blocks are used, if the conditions above occur, restart the HMI controller or wait for the HMI to recover.

The input and output parameters that are common to all PLCCommunication library function blocks are described elsewhere *(see page 13)*.

⚠ WARNING
EXCHANGED DATA INCOMPATIBILITY Verify that the exchanged data are compatible because data structure alignments are not the same for all devices. **Failure to follow these instructions can result in death, serious injury, or equipment damage.**

Example

This POU allows the writing to digital outputs/internal bits 0 to 9 (Q0 to Q9) of a Modbus slave at address 8 on serial line 1:



NOTE: The `Busy` output assigned to `start` allows for the continuous execution of ADDM and WRITE_VAR. The `start` variable must be set to TRUE (online by the user or by the application) after the first cycle to initiate continuous reading/writing. This example does not show the management of exchange errors.

WRITE_READ_VAR: Read and Write Internal Registers on a Modbus Device

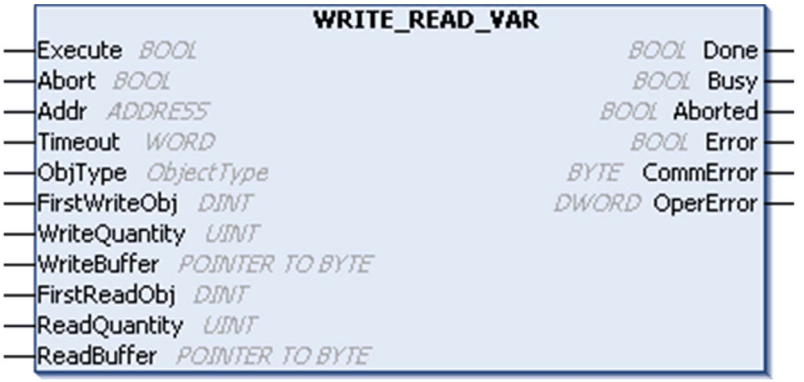
Function Description

This function reads and writes internal registers (MW type only) to an external device in the Modbus protocol. The read and write operations are contained in a single transaction.

The write operation is performed first. The WRITE_READ_VAR function can then:

- write consecutive internal registers and immediately read back their values for verification
- write some consecutive internal registers and read others in a single unique request

Graphical Representation



WRITE_READ_VAR - Specific Parameter Description

Input	Type	Comment
ObjType	ObjectType *(see page 21)*	ObjType is the object type to be written and read (MW only).
FirstWriteObj	DINT	FirstWriteObj is the index of the first object to write.
WriteQuantity	UINT	WriteQuantity is the number of objects to write: ● 1...121: registers (MW type)
WriteBuffer	POINTER TO BYTE	WriteBuffer is the address of the buffer in which objects values are stored. Use the Addr standard function to define the associated pointer. (See the example below.) The buffer is a table that receives the values that are written in the device.
FirstReadObj	DINT	ReadFirstObj is the index of the first object to be read.
ReadQuantity	UINT	ReadQuantity represents the number of objects to be read: ● 1...125: registers (MW type)

Input	Type	Comment
`ReadBuffer`	POINTER TO BYTE	`ReadBuffer` is the address of the buffer in which objects values are stored. Use the `Addr` standard function must be used to define the associated pointer. (See the example below.) The buffer is a table that receives the values that are read in the device.

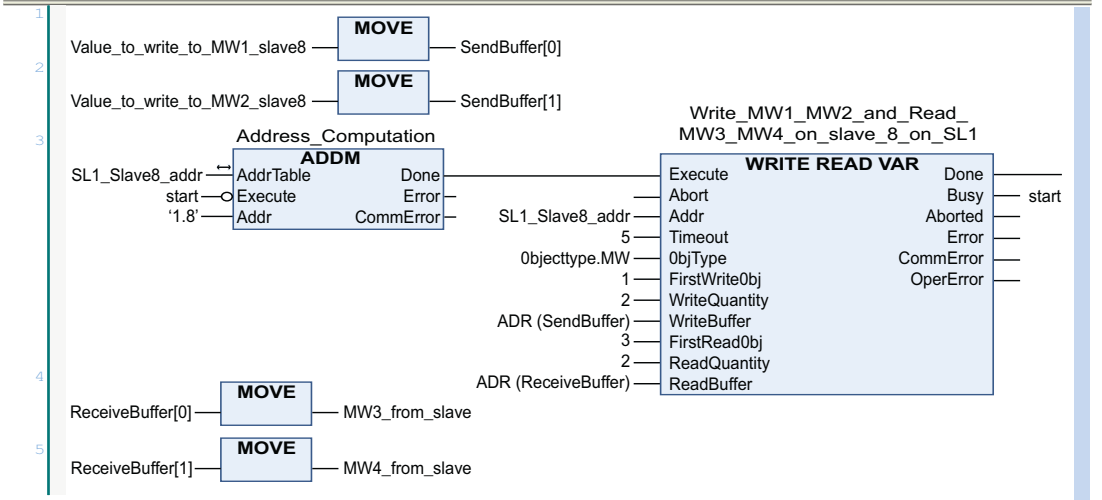
The input and output parameters that are common to all PLCCommunication library function blocks are described elsewhere *(see page 13)*.

Example

This POU allows the writing to internal registers 1 and 2 (MW1 and MW2) and the reading of internal registers 3 and 4 (MW3 and MW4) of a Modbus slave at address 8 on serial line 1:

```
PROGRAM Write_Read_Var_Example
VAR
   SL1_Slave8_addr: ADDRESS;
   Address_Computation: ADDM;

   SendBuffer : ARRAY [0..1] OF WORD;
   ReceiveBuffer : ARRAY [0..1] OF WORD;
   start: BOOL;
   Value_to_write_to_MW1_slave8: WORD;
   Value_to_write_to_MW2_slave8: WORD;
   MW3_from_slave8: WORD;
   MW4_from_slave8: WORD;
   Write_MW1_MW2_and_Read_MW3_MW4_on_slave_8_on_SL1: WRITE_READ_VAR;
END_VAR
```



NOTE: The `Busy` output assigned to `start` allows for the continuous execution of `ADDM` and WRITE_READ_VAR. The `start` variable must be set to TRUE (online by the user or by the application) after the first cycle to initiate continuous writing/reading. This example does not show the management of exchange errors.

SINGLE_WRITE: Write a Single Register to a Modbus Device

Function Description

The SINGLE_WRITE function block writes a single internal register to an external Modbus device.

Graphical Representation

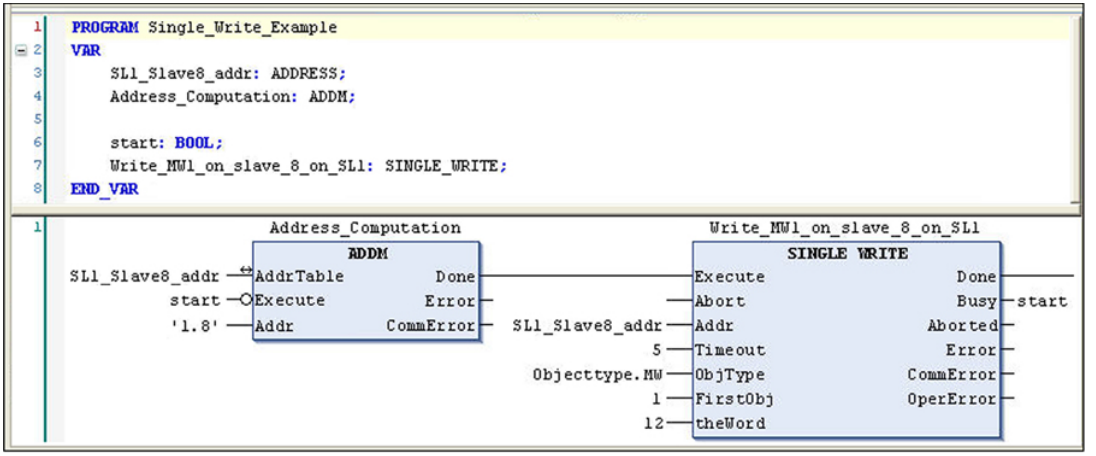
SINGLE_WRITE - Specific Parameter Description

Input	Type	Comment
ObjType	ObjectType	ObjType describes the type of object(s) to write (MW only) *(see page 21)*.
FirstObject	DINT	FirstObject is the index of the object to write.
theWord	WORD	This input contains the value to write.

The input and output parameters that are common to all PLCCommunication library function blocks are described elsewhere *(see page 13)*.

Example

This POU allows the writing of value 12 to internal register 1 (MW1) of a Modbus slave at address 8 on serial line 1:



NOTE: The `Busy` output assigned to `start` allows for the continuous execution of ADDM and SINGLE_WRITE. The `start` variable must be set to TRUE (by the user online or the application) after the first cycle to initiate continuous reading/writing. This example does not show the management of exchange errors.

SEND_RECV_MSG: Send and/or Receive User Defined Messages

Function Description

The SEND_RECV_MSG function block sends and receives user defined messages. It sends a message on the selected media (for example, a serial line) and then waits for a response. It is also possible to either send without waiting for a response or to receive a message without sending one.

This function should be used with an ASCII manager. It could also be used with a Modbus manager if you want to send a request that is not implemented in the communication library. In this case, you have to build the request yourself.

The SEND_RECV_MSG function block is not supported by XBT GC, XBT GT, XBT GK and HMI SCU

Graphical Representation

SEND_RECV_MSG-Specific Parameter Descriptions

Input	Type	Comment
QuantityToSend	UINT	QuantityToSend is the number of bytes to send. Controller limitation: ● M238/LMC078: 252 bytes ● M258/LMC058: 1050 bytes ● M241/M251: 252 bytes
BufferToSend	POINTER TO BYTE	BufferToSend is the address of the buffer (array of bytes) in which the message to send is stored. The ADR standard function must be used to define the associated pointer. (See the example below.) If 0, the function makes a receive-only.

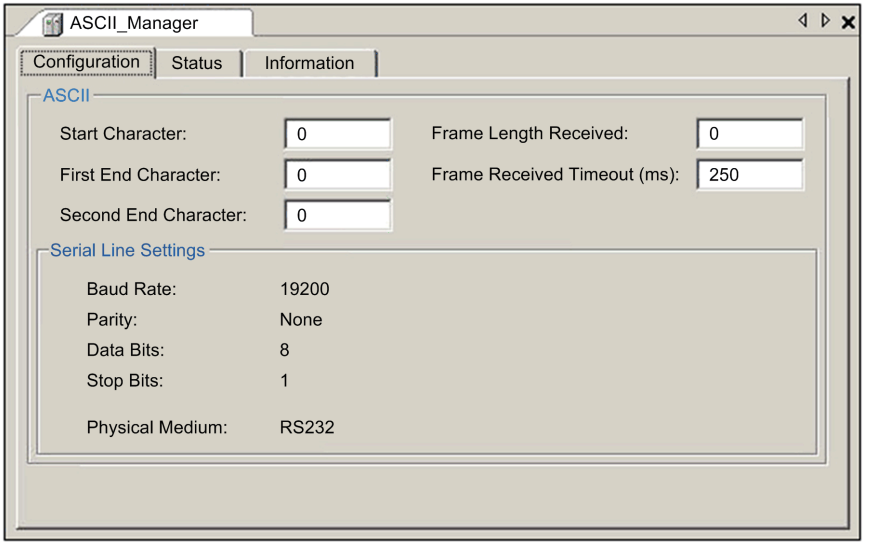
Input	Type	Comment
SizeRecvBuffer	UINT	SizeRecvBuffer is the available size (in bytes) of the receive buffer. The size of the received data (in bytes) is available in the function block instance internal property (internal variable): <Instance Name>.NbRecvBytes. Controller limitation: ● M238/LMC078: 252 bytes ● M258/LMC058: 1050 bytes ● M241/M251: 252 bytes
BufferToRecv	POINTER TO BYTE	BufferToRecv is the address of the buffer (array of SizeRecvBuffer bytes) in which the received message will be stored. The ADR standard function must be used to define the associated pointer. (See the example below.) If 0, the function makes a send-only.

For send only operations, the exchange is complete (Busy reset to 0) when all data (including eventual start and stop characters) have been sent to the line.

For a send/receive or receive only operation, the system receives characters until the ending condition. When the ending condition is reached, the exchange is complete (Busy reset to 0). Received characters are then copied into the receive buffer up to sizeRecvBuffer characters and the size of the received data (in bytes) is available in function block instance property (internal variable): <Instance Name>.NbRecvBytes. The sizeRecvBuffer input does not represent an ending condition.

The input and output parameters that are common to all PLCCommunication library function blocks are described elsewhere *(see page 13)*.

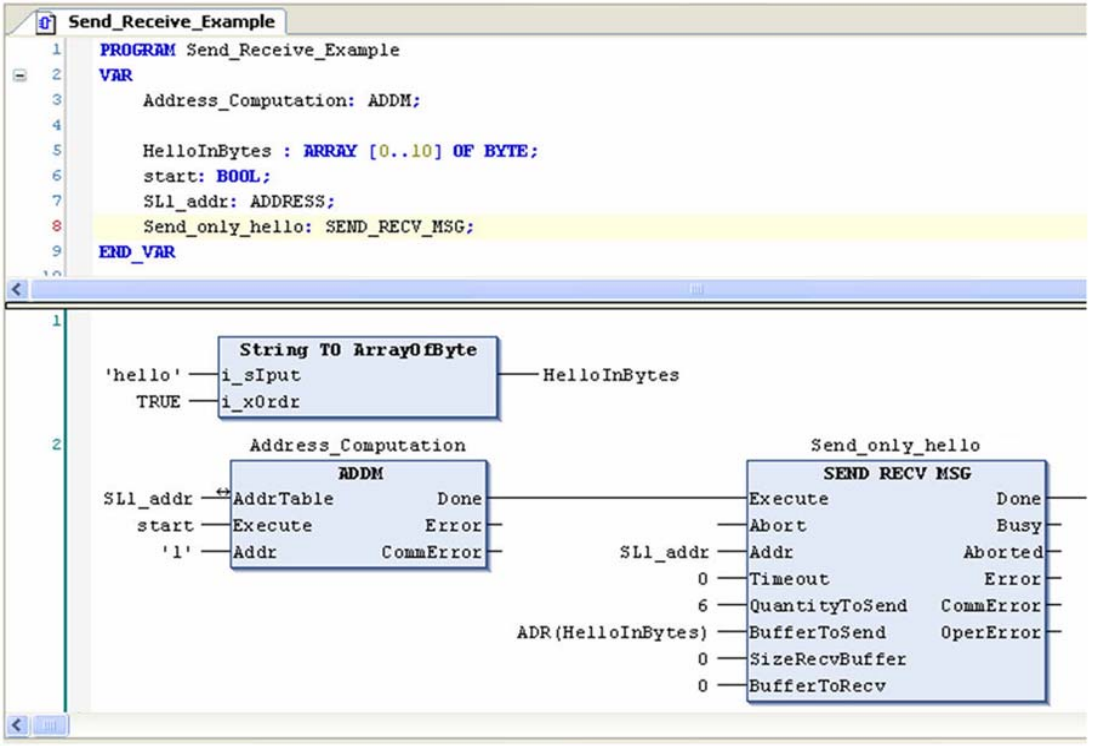
The starting and ending conditions of user defined messages are configured in the ASCII manager's configuration dialog box:



NOTE: There are no start and end characters in this example. The received frame ending condition is a 250 ms Timeout.

Example

This POU allows the send-only of the user defined message “hello” on serial line 1:



NOTE: A rising edge on the `Start` variable launches the conversion of an address and the sending of the message.

Appendices

Appendix A
Function and Function Block Representation

Overview

Each function can be represented in the following languages:

- IL: Instruction List
- ST: Structured Text
- LD: Ladder Diagram
- FBD: Function Block Diagram
- CFC: Continuous Function Chart

This chapter provides functions and function blocks representation examples and explains how to use them for IL and ST languages.

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Differences Between a Function and a Function Block	48
How to Use a Function or a Function Block in IL Language	49
How to Use a Function or a Function Block in ST Language	52

Differences Between a Function and a Function Block

Function

A function:

- is a POU (Program Organization Unit) that returns one immediate result.
- is directly called with its name (not through an instance).
- has no persistent state from one call to the other.
- can be used as an operand in other expressions.

Examples: boolean operators (`AND`), calculations, conversion (`BYTE_TO_INT`)

Function Block

A function block:

- is a POU (Program Organization Unit) that returns one or more outputs.
- needs to be called by an instance (function block copy with dedicated name and variables).
- each instance has a persistent state (outputs and internal variables) from one call to the other from a function block or a program.

Examples: timers, counters

In the example, `Timer_ON` is an instance of the function block `TON`:

```
PROGRAM MyProgram_ST
VAR
    Timer_ON: TON;  // Function Block Instance
    Timer_RunCd: BOOL;
    Timer_PresetValue: TIME := T#5S;
    Timer_Output: BOOL;
    Timer_ElapsedTime: TIME;
END_VAR
```

```
Timer_ON(
    IN:=Timer_RunCd,
    PT:=Timer_PresetValue,
    Q=>Timer_Output,
    ET=>Timer_ElapsedTime);
```

How to Use a Function or a Function Block in IL Language

General Information

This part explains how to implement a function and a function block in IL language.

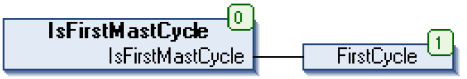
Functions `IsFirstMastCycle` and `SetRTCDrift` and Function Block `TON` are used as examples to show implementations.

Using a Function in IL Language

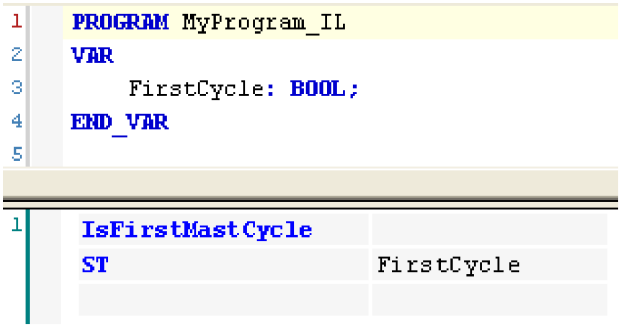
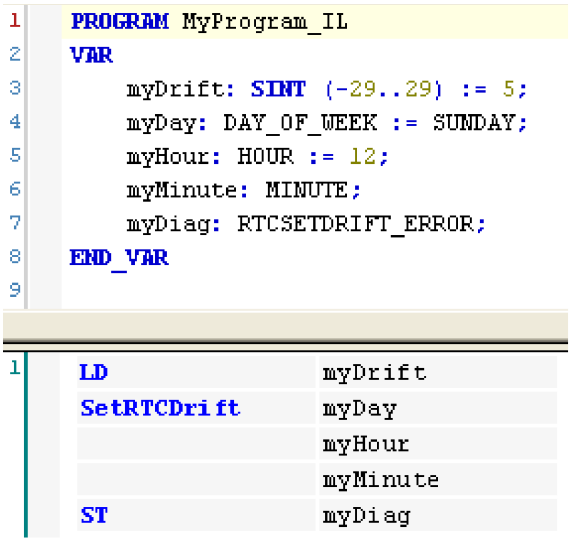
This procedure describes how to insert a function in IL language:

Step	Action
1	Open or create a new POU in Instruction List language. **NOTE:** The procedure to create a POU is not detailed here. For more information, refer to Adding and Calling POUs *(see SoMachine, Programming Guide)*.
2	Create the variables that the function requires.
3	If the function has 1 or more inputs, start loading the first input using LD instruction.
4	Insert a new line below and: ● type the name of the function in the operator column (left field), or ● use the **Input Assistant** to select the function (select **Insert Box** in the context menu).
5	If the function has more than 1 input and when Input Assistant is used, the necessary number of lines is automatically created with `???` in the fields on the right. Replace the `???` with the appropriate value or variable that corresponds to the order of inputs.
6	Insert a new line to store the result of the function into the appropriate variable: type ST instruction in the operator column (left field) and the variable name in the field on the right.

To illustrate the procedure, consider the Functions `IsFirstMastCycle` (without input parameter) and `SetRTCDrift` (with input parameters) graphically presented below:

Function	Graphical Representation
without input parameter: `IsFirstMastCycle`	
with input parameters: `SetRTCDrift`	

In IL language, the function name is used directly in the operator column:

Function	Representation in SoMachine POU IL Editor
IL example of a function without input parameter: `IsFirstMastCycle`	
IL example of a function with input parameters: `SetRTCDrift`	

Using a Function Block in IL Language

This procedure describes how to insert a function block in IL language:

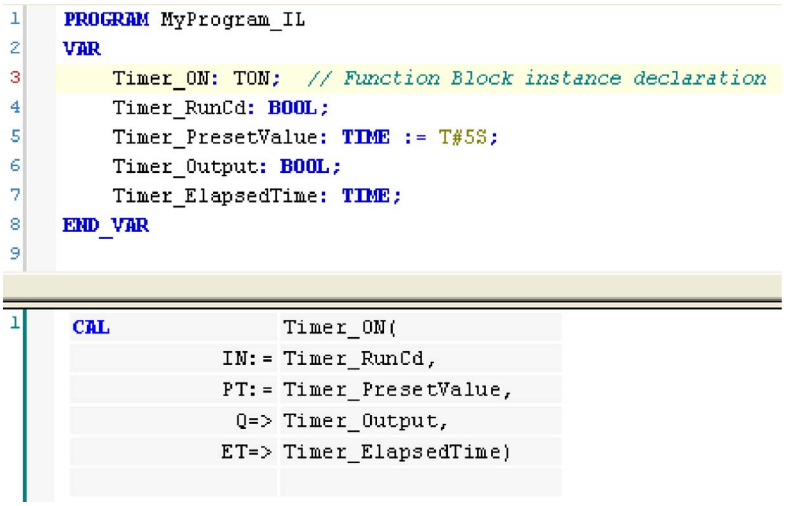
Step	Action
1	Open or create a new POU in Instruction List language. **NOTE:** The procedure to create a POU is not detailed here. For more information, refer to Adding and Calling POUs *(see SoMachine, Programming Guide)*.

Step	Action
2	Create the variables that the function block requires, including the instance name.
3	Function Blocks are called using a CAL instruction: ● Use the **Input Assistant** to select the FB (right-click and select **Insert Box** in the context menu). ● Automatically, the CAL instruction and the necessary I/O are created. Each parameter (I/O) is an instruction: ● Values to inputs are set by ":=". ● Values to outputs are set by "=>".
4	In the CAL right-side field, replace ??? with the instance name.
5	Replace other ??? with an appropriate variable or immediate value.

To illustrate the procedure, consider this example with the TON Function Block graphically presented below:

Function Block	Graphical Representation
TON	

In IL language, the function block name is used directly in the operator column:

Function Block	Representation in SoMachine POU IL Editor
TON	

How to Use a Function or a Function Block in ST Language

General Information

This part explains how to implement a Function and a Function Block in ST language.

Function `SetRTCDrift` and Function Block `TON` are used as examples to show implementations.

Using a Function in ST Language

This procedure describes how to insert a function in ST language:

Step	Action
1	Open or create a new POU in Structured Text language. **NOTE:** The procedure to create a POU is not detailed here. For more information, refer to Adding and Calling POUs *(see SoMachine, Programming Guide)*.
2	Create the variables that the function requires.
3	Use the general syntax in the **POU ST Editor** for the ST language of a function. The general syntax is: `FunctionResult:= FunctionName(VarInput1, VarInput2,.. VarInputx);`

To illustrate the procedure, consider the function `SetRTCDrift` graphically presented below:

Function	Graphical Representation
`SetRTCDrift`	

The ST language of this function is the following:

Function	Representation in SoMachine POU ST Editor

`SetRTCDrift`	`PROGRAM MyProgram_ST`
	`VAR myDrift: SINT(-29..29) := 5;`
	`myDay: DAY_OF_WEEK := SUNDAY;`
	`myHour: HOUR := 12;`
	`myMinute: MINUTE;`
	`myRTCAdjust: RTCDRIFT_ERROR;`
	`END_VAR`
	`myRTCAdjust:= SetRTCDrift(myDrift, myDay, myHour, myMinute);`

Using a Function Block in ST Language

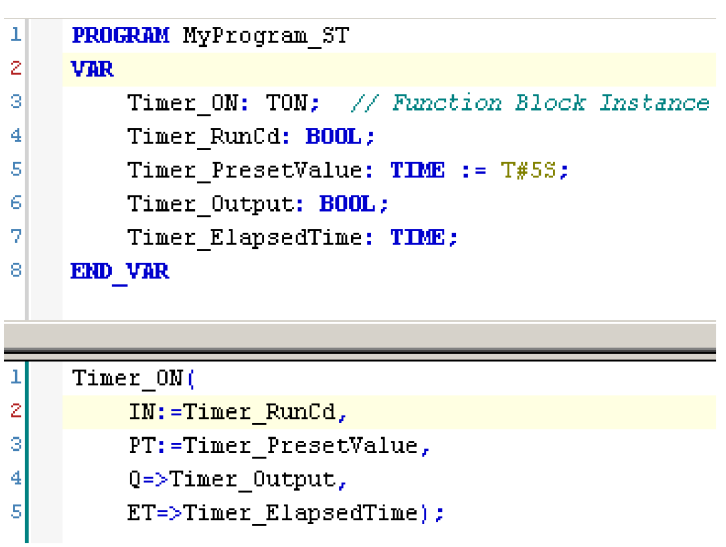
This procedure describes how to insert a function block in ST language:

Step	Action
1	Open or create a new POU in Structured Text language. **NOTE:** The procedure to create a POU is not detailed here. For more information on adding, declaring and calling POUs, refer to the related documentation *(see SoMachine, Programming Guide)*.
2	Create the input and output variables and the instance required for the function block: ● Input variables are the input parameters required by the function block ● Output variables receive the value returned by the function block
3	Use the general syntax in the **POU ST Editor** for the ST language of a Function Block. The general syntax is: `FunctionBlock_InstanceName(Input1:=VarInput1, Input2:=VarInput2,... Ouput1=>VarOutput1, Ouput2=>VarOutput2,...);`

To illustrate the procedure, consider this example with the `TON` function block graphically presented below:

Function Block	Graphical Representation
`TON`	

This table shows examples of a function block call in ST language:

Function Block	Representation in SoMachine POU ST Editor
TON	

```
PROGRAM MyProgram_ST
VAR
    Timer_ON: TON;  // Function Block Instance
    Timer_RunCd: BOOL;
    Timer_PresetValue: TIME := T#5S;
    Timer_Output: BOOL;
    Timer_ElapsedTime: TIME;
END_VAR
```

```
Timer_ON(
    IN:=Timer_RunCd,
    PT:=Timer_PresetValue,
    Q=>Timer_Output,
    ET=>Timer_ElapsedTime);
```

Glossary

A

application

A program including configuration data, symbols, and documentation.

B

byte

A type that is encoded in an 8-bit format, ranging from 00 hex to FF hex.

C

CFC

(*continuous function chart*) A graphical programming language (an extension of the IEC 61131-3 standard) based on the function block diagram language that works like a flowchart. However, no networks are used and free positioning of graphic elements is possible, which allows feedback loops. For each block, the inputs are on the left and the outputs on the right. You can link the block outputs to the inputs of other blocks to create complex expressions.

configuration

The arrangement and interconnection of hardware components within a system and the hardware and software parameters that determine the operating characteristics of the system.

controller

Automates industrial processes (also known as programmable logic controller or programmable controller).

E

expansion bus

An electronic communication bus between expansion I/O modules and a controller.

F

FB

(*function block*) A convenient programming mechanism that consolidates a group of programming instructions to perform a specific and normalized action, such as speed control, interval control, or counting. A function block may comprise configuration data, a set of internal or external operating parameters and usually 1 or more data inputs and outputs.

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

function block diagram

One of the 5 languages for logic or control supported by the standard IEC 61131-3 for control systems. Function block diagram is a graphically oriented programming language. It works with a list of networks where each network contains a graphical structure of boxes and connection lines representing either a logical or arithmetic expression, the call of a function block, a jump, or a return instruction.

I

I/O

(*input/output*)

IL

(*instruction list*) A program written in the 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 (refer to IEC 61131-3).

INT

(*integer*) A whole number encoded in 16 bits.

L

LD

(*ladder diagram*) 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 (refer to IEC 61131-3).

P

PLCopen

For more information, refer to *http://www.plcopen.org/*.

POU

(*program organization unit*) A variable declaration in source code and a corresponding instruction set. POUs facilitate the modular re-use of software programs, functions, and function blocks. Once declared, POUs are available to one another.

program

The component of an application that consists of compiled source code capable of being installed in the memory of a logic controller.

S

ST

(*structured text*) A language that includes complex statements and nested instructions (such as iteration loops, conditional executions, or functions). ST is compliant with IEC 61131-3.

V

variable

A memory unit that is addressed and modified by a program.

Index

A

ADDM
  Function Block, *26*
ADDR_EXT
  Data Types, *23*
ADDRESS
  Data Types, *22*

C

CommunicationErrorCodes
  Data Types, *18*

D

Data Types
  ADDR_EXT, *23*
  ADDRESS, *22*
  CommunicationErrorCodes, *18*
  LinkNumber, *20*
  ObjectType, *21*
  OperationErrorCodes, *19*
  TCP_ADDR_EXT, *24*

F

Function Block
  ADDM, *26*
  READ_VAR, *31*
  SEND_RECV_MSG, *40*
  SINGLE_WRITE, *38*
  WRITE_READ_VAR, *35*
  WRITE_VAR, *33*
functions
  differences between a function and a function block, *48*
  how to use a function or a function block in IL language, *49*
  how to use a function or a function block in ST language, *52*

L

LinkNumber
  Data Types, *20*

O

ObjectType
  Data Types, *21*
OperationErrorCodes
  Data Types, *19*

R

READ_VAR
  Function Block, *31*

S

SEND_RECV_MSG
  Function Block, *40*
SINGLE_WRITE
  Function Block, *38*

T

TCP_ADDR_EXT
  Data Types, *24*

W

WRITE_READ_VAR
  Function Block, *35*
WRITE_VAR
  Function Block, *33*