SoMachine

Data Logging Functions
DataLogging Library Guide

04/2014

EIO0000000551.03

www.schneider-electric.com

Schneider Electric

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 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 an imminently hazardous situation which, if not avoided, **will result in** death or serious injury.

⚠ WARNING

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

⚠ CAUTION

CAUTION indicates a potentially hazardous situation which, if not avoided, **can 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 explains the data logging functionality for controllers that support file management operations.

Validity Note

This document has been updated with the release of SoMachine V4.0.

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.

Chapter 1
Data Logging

Introduction

This document explains the data logging functionality for controllers that support file management operations.

A data log is a text file of user-defined strings that contain application data information for a process or machine. The data log file is stored on the controller. You can upload the file and open it with a standard text editor. The information includes embedded variable values and their associated text. Time and date stamping are additional options.

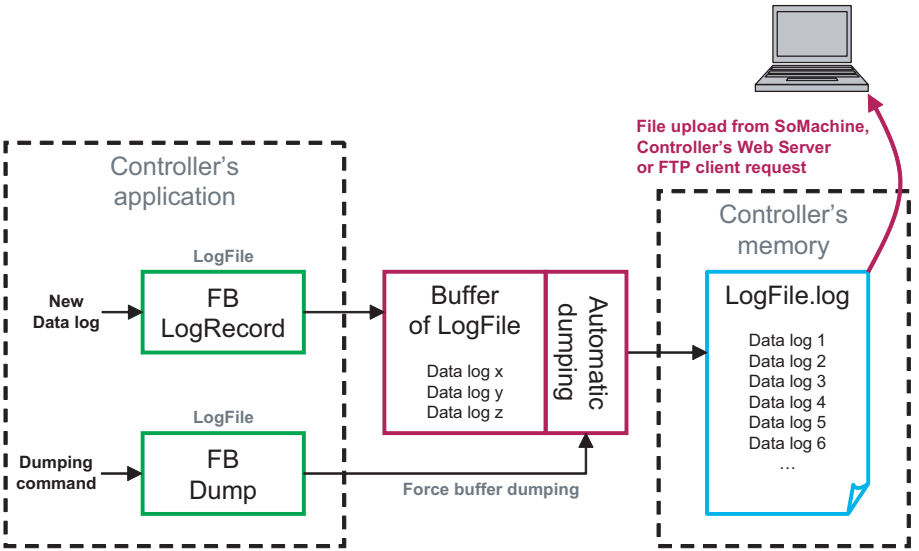
What Is in This Chapter?

This chapter contains the following topics:

Introduction to Data Logging

Overview

You can monitor and analyze application data by examining the data log file (.log).



The figure shows an application that includes the 2 function blocks, `LogRecord` and `Dump`. The `LogRecord` function block writes data to the buffer, which empties into the data log file (.log) located into the controller's memory. The buffer dumping is automatic when 80% full or it can be forced by the `Dump` function. As a standard FTP client, a PC can access this data log file when the controller acts as an FTP server. It is also possible to upload the file with SoMachine or by the controller's web server.

NOTE: Only controllers with file management functionality can support data logging. Refer to your controller's programming manual to see if it supports file management. The software itself does not evaluate your controller for compatibility with data logging activities.

Sample Data Log File (.log)

```
Entries in File: 8; Last Entry: 8;
18/06/2009;14:12:33;cycle: 1182;
18/06/2009;14:12:35;cycle: 1292;
18/06/2009;14:12:38;cycle: 1450;
18/06/2009;14:12:40;cycle: 1514;
18/06/2009;14:12:41;cycle: 1585;
18/06/2009;14:12:43;cycle: 1656;
18/06/2009;14:14:20;cycle: 6346;
18/06/2009;14:14:26;cycle: 6636;
```

Implementation Procedure

You must first declare and configure the data log files in your application before starting to write your program.

Configuring the Data Log

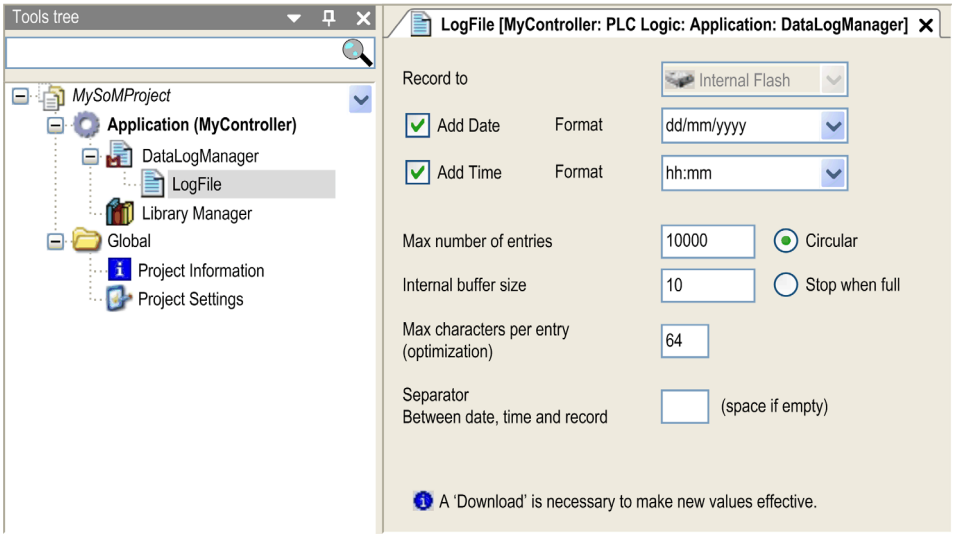
Add a Data Log Manager

Add a data log manager to your configuration before configuring data logging:

Step	Action
1	In the **Tools tree**, select the **Application** node, click the green plus sign and select **Add other objects** → **DataLogManager...**. **Result:** The **Add Data Log Manager** dialog box appears.
2	In the **Add Data Log Manager** dialog box, click **Add**. **Result:** A **DataLogManager** node appears below the **Application** node.
3	Select the **DataLogManager** node, click the green plus sign, and select **DataLog...**. **Result:** The **Add DataLog** dialog box appears.
4	In the **Data Logging File Name** text box, enter a name for your data log file, and click **Add**. **Result:** The data log file with the given name appears below the **DataLogManager** node, and the configuration screen opens in the editor view in the middle of the SoMachine Logic Builder screen. **Note:** The data log file name cannot be changed later.
5	Set the data log file parameters *(see page 12)*.
6	Repeat steps 3 to 5 to create additional data log files.

Configuration Screen

This configuration screen appears after you add a data log file to your configuration:



Configuration Parameters

Parameter	Description	
Add Date	These options print the respective date or time for each record. For example, an instance associated with 10 June 2009 at 2:30 p.m. could be represented as `10/06/2009` or `06/10/2009` or `20090610` ... at `14:30` or `02:30:00 pm`, etc.	
Add Time		
Max number of entries	This option sets the maximum number of records contained in the data log file. Valid values are from 10 to 65536. (The default is 10000.)	
Mode	**Circular** (default)	When the `Max number of entries` is reached, new records overwrite old records. The first line of the data log file can be used to get the actual position of the last record as well as the rank of other records. The following cases are based on a file with a 10-record maximum: ● **Case 1:** `Entries in File: 8; Last Entry: 8` **Meaning:** The number of entries in the file equals the value of `Last Entry`. The records are ranked from N° 1 (the oldest) to N° 8 (the most recent). ● **Case 2:** `Entries in File: 10; Last Entry: 5` **Meaning:** The number of entries in the file is greater than the value of `Last Entry`. The file is full and a new record replaces the oldest one in the file. The 10 records are ranked in this order (oldest to newest): 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
	Stop when full	When the `Max number of entries` is reached, a new record attempt returns a detected error message.
Internal Buffer Size	Configure the size of the RAM buffer that stores the added records. Valid values are from 1 to the value configured in **Max number of entries**. (The default is 10.)	
Max characters per entry (optimization)	Set the maximum length of each entry. Valid values are from 10 to 255. (The default is 64.) The maximum length includes separators and optional date and time. Additional spaces are added at the end of the line to accommodate the number of defined characters.	
Separator between date, time and record	Define the character to be inserted between different fields in the data log. An additional separator is added after the record.	
NOTE: For each configured data log file, there is a `LogRecord` function block instance with the same name as this data log file that handles all internal data and allows data logging management.		

Managing Data Log Files

Introduction

One function block, `LogRecord`, is provided for writing text string entries to the data log file. This function block stores the input string in an internal buffer. When this buffer reaches 80 % of capacity, it is moved to the real file on the controller. You can force this save mechanism by using the `Dump` function block.

When power is removed, you may lose the data in the internal buffer or increase the cycle time until the buffer is emptied.

NOTICE
LOSS OF DATA ● Do not remove power to the controller until all internal buffer information has been moved to the actual file system. ● If the data being recorded is important to your application, configure your internal buffer size to 1. **Failure to follow these instructions can result in equipment damage.**

Adding a Record with the LogRecord Function Block

This function block is available for logging a UNICODE string in a specific log:

Input Parameters

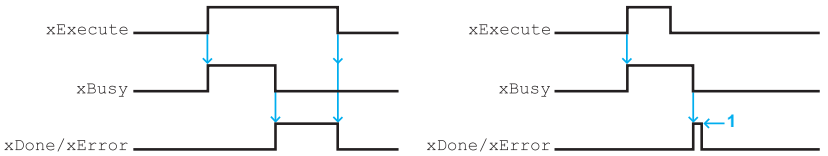
Parameter	Type	Comment
`xExecute`	`BOOL`	The function is executed on the rising edge of this input. **NOTE:** When `xExecute` is set to TRUE at the first task cycle in RUN after a cold or warm reset, the rising edge is not detected.
`wsRecord`	`WSTRING`	This user-specified UNICODE text string is written to the data log file. **NOTE:** The WSTRING *(see page 20)* type is available in the `Standard64.lib` library that is automatically inserted when the data log manager is added to the application.

Output Parameters

Parameter	Type	Comment
xDone	BOOL	This output is set to TRUE when the record is successfully saved to the internal buffer without error messages.
xBusy	BOOL	This output remains TRUE while LogRecord is busy (until the transfer to the buffer is complete).
xError	BOOL	This output is set to TRUE when an error occurs (for example, when the internal buffer is full).
eError	ERROR	This output contains the error code when xError is TRUE: ● NO_ERROR ● INIT_ERROR ● DUMP_ERROR ● BUFFER_FULL_ERROR ● FILE_FULL_ERROR ● DUMP_INCOMPLETED ● INPUT_ERROR ● FILE_OPEN_ERROR ● FILE_SETPOINTER_ERROR ● FILE_WRITE_ERROR ● FILE_CLOSE_ERROR

NOTE: If the record exceeds the configured length, it is truncated.

The xDone and xError outputs remain TRUE as long as xExecute is TRUE. When xExecute is set to FALSE before xDone or xError is set to TRUE (xBusy is still TRUE), one of each is set to TRUE when the function block is completed during one controller cycle so that the application detects this end:



1 One cycle as Ex is FALSE.

The LogRecord function block includes an instance that corresponds to each configured data log file. You do not need to explicitly declare an instance of the function block because the instance is declared automatically. Add the function block to your POU and specify the appropriate data log file instance with input help (see Adding LogRecord *(see page 19)*).

Forcing a Save with the Dump Method

You can use the `Dump` method (a method can be regarded as sub-function of a function block, in this case `LogRecord`) to force the application data to move stored records in the internal buffer to the real file in the controller's file system:

Parameters

Input	Type	Comment
xExecute	`BOOL`	The save function is executed on the rising edge of this input. **NOTE:** When `xExecute` is set to 1 at first controller task cycle, the rising edge is not detected.

Output	Type	Comment
xDone	`BOOL`	This output is set to TRUE when the records are successfully saved without error messages.
xBusy	`BOOL`	This output remains TRUE while Dump is busy (until the file is completely written).
xError	`BOOL`	This output is set to TRUE when an error is detected (for example, when the data log file is full).
eError	`ERROR`	This output contains the error code when `xError` is TRUE: ● NO_ERROR ● INIT_ERROR ● DUMP_ERROR ● BUFFER_FULL_ERROR ● FILE_FULL_ERROR ● DUMP_INCOMPLETED ● INPUT_ERROR ● FILE_OPEN_ERROR ● FILE_SETPOINTER_ERROR ● FILE_WRITE_ERROR ● FILE_CLOSE_ERROR

Refer to Adding a Dump Method *(see page 19)* for the `Dump` method implementation.

Automatic Save to Data Log File

When the `LogRecord` function block is used, the system automatically moves the data from the internal buffer to the data log file when this buffer is 80 % full. The `Dump` function block allows you to force this move before reaching the 80 %. This 80 % limit allows the write process to begin before the buffer is full and while a new record is added.

This table shows the number of records to which the file is stored for a given configured buffer size (80% of buffer size rounded down to the nearest integer):

Buffer Size	80 % Limit	Comment
1	1	The save to the data log file is activated as soon as a record is added with an additional slot available for a new record that arrives during this save.
2	1	The save to the data log file is activated when the buffer reaches 80% with an additional slot available for a new record that arrives during this save.
3	2	
4	3	
5	4	The save to the data log file is activated when the buffer reaches 80% with additional slots available for new records that arrive during this save.
6	4	
7	5	
8	6	
...		

In the case of either an explicit (`Dump`) or automatic (80 %) save, the data log file closes after each record (or record group) in case there is a subsequent external power failure.

Additional Data Log File Information

Data Log Properties

The data log properties can be accessed after the function block is configured.

The **LogRecord** properties are additional variables (read only) automatically attached to the **LogRecord** instance that provide information about the data log file status:

Variables	Type	Description
< Data Log File name >.NumberOfRecords	`UDINT`	the number of records in the data log file
< Data Log File name >.NumberOfBufferedRecords	`UINT`	the number of records in the buffer
< Data Log File name > .FileStatus	`FILE_STATUS`	status information about the data log file (`FILE_STATUS` type): ● 0: OK ● 1: FILE_FULL ● 2: NO_WRITE_ACCESS ● 3: FILE_NOT_EXISTS
< Data Log File name > .DumpInProgress	`BOOL`	TRUE when the buffered records are being saved in the data log file

Recommendation

The **LogRecord** function block needs much more than 15 interval cycles after its activation (with `xExecute`) to save a record in the logfile. Therefore, it is fort recommended to use this function block within a FAST task:

Task Type	Interval (ms)	Minimum Time Needed to Save Record
cyclic	20	300 ms
cyclic	1	15 ms: Fort Recommended
event		needs 15 events: Not Recommended

Adding LogRecord and Dump Function Blocks

Overview

Follow these steps to add the `LogRecord` function block and `Dump` method to your project.

Adding LogRecord

Add the `LogRecord` function block to your project:

Step	Action	Comment
1	Insert a `LogRecord` function block in your POU using the **Input Assistant** or by directly typing **LogRecord**.	From the **Input Assistant** dialog box, make these selections: ● Categories: **Function Blocks (Libraries)** ● Items: **{} SEDL →LogRecord** (Items in structured view)
2	Click **OK** or press **ENTER**.	The `LogRecord` function block is now part of your project.
3	Select the appropriate data log file as **LogRecord** instance name	From the input assistant, select the appropriate data log file or directly enter the data log file name.
4	Configure the inputs and outputs as you would for other function blocks.	Refer to the description of this function block's parameters *(see page 14)*.

Adding the Dump Method

Add the `Dump` method to your project:

Step	Action	Comment
1	Insert a `Dump` method to your POU using the **Input Assistant** or by directly typing **LogRecord.Dump**.	From the **Input Assistant** dialog box, make these selections: ● Categories: **Function Blocks (Libraries)** ● Items: **SEDL →LogRecord →Dump** (Items in structured view)
2	Click **OK** or press **ENTER**.	The **LogRecord.Dump** method is now part of your project.
3	Select the appropriate data log file as **LogRecord** instance name	From the **Input Assistant**, select the appropriate data log file or directly enter the data log file name.
4	Configure the inputs and outputs as you would for other functions.	Refer to the description of this function block's parameters *(see page 16)*.

Building a Wide WSTRING

Overview

The `wsRecord` input of the `LogRecord` function block is of the type `WSTRING` (wide string). To build the log string, you must first add the `Standard64` library to your application and use wide string functions.

Example

This figure shows the creation of a sample `WSTRING` that includes a variable value:

```
PROGRAM POU_1
VAR
    cycle: INT;
    str: WSTRING;
END_VAR
```

```
cycle := cycle + 1;
str := wconcat("cycle: ", INT_TO_WSTRING(cycle));
```

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:

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`:

```
1  PROGRAM MyProgram_ST
2  VAR
3      Timer_ON: TON;  // Function Block Instance
4      Timer_RunCd: BOOL;
5      Timer_PresetValue: TIME := T#5S;
6      Timer_Output: BOOL;
7      Timer_ElapsedTime: TIME;
8  END_VAR
```

```
1  Timer_ON(
2      IN:=Timer_RunCd,
3      PT:=Timer_PresetValue,
4      Q=>Timer_Output,
5      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.

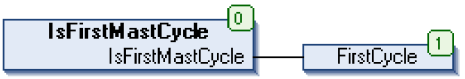
Functions `IsFirstMastCycle` and `SetRTCDrift` and Function Block `TON` are used as examples to show implementations.

Using a Function in IL Language

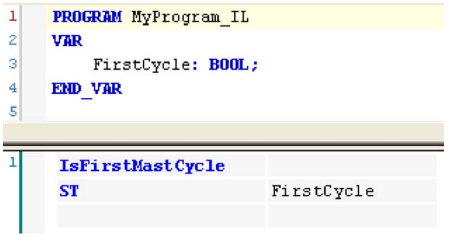
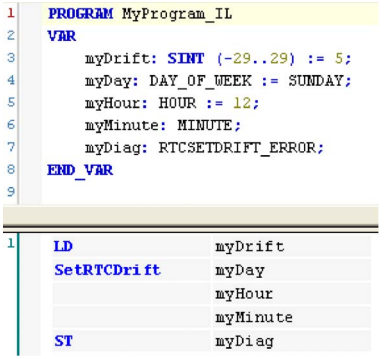
This procedure describes how to insert a function in IL language:

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, Declaring an Calling POUs.
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, Declaring an Calling POUs.
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	`PROGRAM MyProgram_IL`
	`VAR`
	`  Timer_ON: TON;  // Function Block instance declaration`
	`  Timer_RunCd: BOOL;`
	`  Timer_PresetValue: TIME := T#5S;`
	`  Timer_Output: BOOL;`
	`  Timer_ElapsedTime: TIME;`
	`END_VAR`
	
	`CAL  Timer_ON(`
	`  IN:= Timer_RunCd,`
	`  PT:= Timer_PresetValue,`
	`  Q=> Timer_Output,`
	`  ET=> Timer_ElapsedTime)`

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, Declaring and Calling POUs .
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.
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	1 PROGRAM MyProgram_ST
	2 VAR
	3 Timer_ON: TON; // Function Block Instance
	4 Timer_RunCd: BOOL;
	5 Timer_PresetValue: TIME := T#5S;
	6 Timer_Output: BOOL;
	7 Timer_ElapsedTime: TIME;
	8 END_VAR
	
	1 Timer_ON(
	2 IN:=Timer_RunCd,
	3 PT:=Timer_PresetValue,
	4 Q=>Timer_Output,
	5 ET=>Timer_ElapsedTime);

Glossary

B

byte

A type that is encoded in an 8-bit format, ranging from `16#00` to `16#FF` in hexadecimal representation.

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.

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.

FBD

(*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

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

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.

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