KAS IDE – PLC Library

Reference Manual

Valid for Software Revision 2.5

Keep all manuals as a product component during the life span of the product.
Pass all manuals to future users / owners of the product.

Trademarks and Copyrights

Copyrights

Copyright © 2009-12 Kollmorgen™
Information in this document is subject to change without notice. The software package described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of those agreements.

This document is the intellectual property of Kollmorgen™ and contains proprietary and confidential information. The reproduction, modification, translation or disclosure to third parties of this document (in whole or in part) is strictly prohibited without the prior written permission of Kollmorgen™.

Trademarks

KAS and AKD are registered trademarks of Kollmorgen™.
SERVO**STAR** is a registered trademark of Kollmorgen™.
Kollmorgen™ is part of the Danaher Motion company.
Windows® is a registered trademark of Microsoft Corporation
EnDat is a registered trademark of Dr. Johannes Heidenhain GmbH.
EtherCAT® is registered trademark of Ethercat Technology Group.
PLCopen is an independent association providing efficiency in industrial automation.
INtime® is a registered trademark of TenAsys® Corporation.
Codemeter is a registered trademark of WIBU-Systems AG.
SyCon® is a registered trademark of Hilscher GmbH.

Kollmorgen Automation Suite is based on the work of:

- Qwt project (distributed under the terms of the GNU Lesser General Public License - see also GPL terms)
- Zlib software library
- Curl software library
- Mongoose software (distributed under the MIT License - see terms)
- JsonCpp software (distributed under the MIT License – see terms)
- U-Boot, a universal boot loader is used by the AKD-PDMM (distributed under the terms of the GNU General Public License). The U-Boot source files, copyright notice, and readme are available on the distribution disk that is included with the AKD-PDMM.

All other product and brand names listed in this document may be trademarks or registered trademarks of their respective owners.

Disclaimer

The information in this document (Version 2.5 published on 5/7/2012) is believed to be accurate and reliable at the time of its release. Notwithstanding the foregoing, Kollmorgen assumes no responsibility for any damage or loss resulting from the use of this help, and expressly disclaims any liability or damages for loss of data, loss of use, and property damage of any kind, direct, incidental or consequential, in regard to or arising out of the performance or form of the materials presented herein or in any software programs that accompany this document.

All timing diagrams, whether produced by Kollmorgen or included by courtesy of the PLCopen organization, are provided with accuracy on a best-effort basis with no warranty, explicit or implied, by Kollmorgen. The user releases Kollmorgen from any liability arising out of the use of these timing diagrams.

Table of Contents

This page intentionally left blank.

1 Programming languages

This chapter presents details on the syntax, structure and use of the declarations and statements supported by the KAS IDE application language.

Below are the available programming languages of the IEC 61131-3 standard:

SFC: Sequential Function Chart
FBD: Function Block Diagram
FFLD: Free Form Ladder Diagram
ST: Structured Text
IL: Instruction List

Use of ST instructions in graphic languages

You have to select a language for each program or User-Defined Function Block of the application.

1.1 Sequential Function Chart (SFC)

The SFC language is a state diagram. Graphical steps are used to represent stable states, and transitions describe the conditions and events that lead to a change of state. Using SFC highly simplifies the programming of sequential operations as it saves a lot of variables and tests just for maintaining the program context.

Warning

You must not use SFC as a decision diagram. Using a step as a point of decision and transitions as conditions in an algorithm must never appear in an SFC chart. Using SFC as a decision language leads to poor performance and complicate charts. ST must be preferred when programming a decision algorithm that has no sense in term of "program state"

Below are basic components of an SFC chart:

Chart:	*Programming:*
Steps and initial steps	Actions within a step
Transitions and divergences	Timeout on a step
Parallel branches	Programming a transition condition
Macro-steps	
Jump to a step	How SFC is executed
	UDFBs programmed in SFC

The KAS IDE fully supports SFC programming with several hierarchical levels of charts: i.e. a chart that controls another chart. Working with a hierarchy of SFC charts is an easy and powerful way for managing complex sequences and saves performances at run time. Refer to the following sections for further details:

Defining a hierarchy of SFC programs
How to control an SFC child?

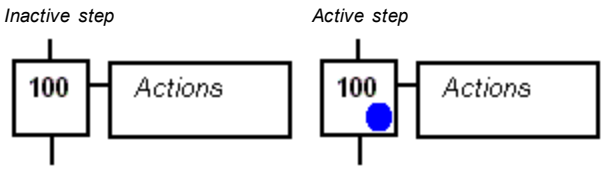
1.1.1 SFC Steps

A step represents a stable state. It is drawn as a square box in the SFC chart. Each step of a program is identified with a unique number. At run time, a step can be either active or inactive according to the state of the program.

Note

To change the number of a step, transition or jump, select it and hit **Ctrl+ENTER** keys.

All actions linked to the steps are executed according to the activity of the step.



In conditions and actions of the SFC program, you can test the step activity by specifying its name ("GS" plus the step number) followed by "**.X**".
For example:

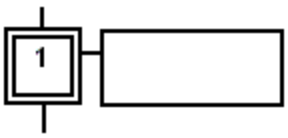
`GS100.X`	is TRUE if step 100 is active (expression has the BOOL data type)

You can also test the activity time of a step, by specifying the step name followed by "**.T**". It is the time elapsed since the activation of the step. When the step is deactivated, this time remains unchanged. It will be reset to 0 on the next step activation. For example:

`GS100.T`	is the time elapsed since step 100 was activated (expression has the TIME data type)

Initial steps

Initial steps represent the initial situation of the chart when the program is started. There must be at least one initial step in each SFC chart. An initial step is marked with a double line:



1.1.2 SFC Transitions

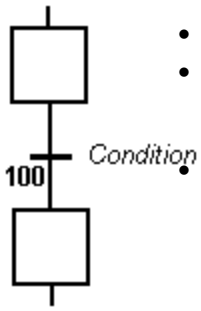
Transitions represent a condition that changes the program activity from a step to another.

Note

To change the number of a step, transition or jump, select it and hit **Ctrl+ENTER** keys.

The transition is marked by a small horizontal line that crosses a link drawn between the two steps.

The default direction for vertical links is from the top to the bottom



- Each transition is identified by a unique number in the SFC program.
- Each transition must be completed with a boolean condition that indicates if the transition can be crossed. The condition is a BOOL expression. If no condition is entered, it is assumed as always TRUE.
- In order to simplify the chart and reduce the number of drawn links, you can specify the activity flag of a step (GS*nnn*.X) in the condition of the transition.

Transitions define the dynamic behavior of the SFC chart, according to the following rules:

- A transition in crossed if:
  - its condition is TRUE
  - and if all steps linked to the top of the transition (i.e. before) are active
- When a transition is crossed:

- all steps linked to the top of the transition (i.e. before) are deactivated
- all steps linked to the bottom of the transition (i.e. after) are activated

Note

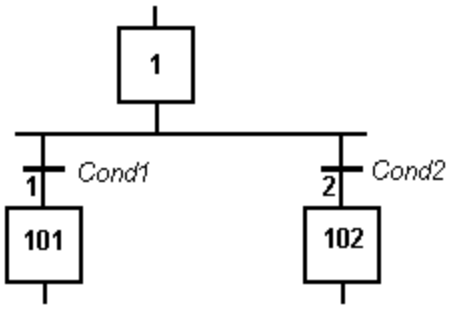
When the same step is linked before and after the transition, it remains active (no pulse in its activity signal)

Divergences

It is possible to link a step to several transitions and thus create a divergence. The divergence is represented by a horizontal line. Transitions after the divergence represent several possible changes in the situation of the program.

All conditions are considered as exclusive, according to a "left to right" priority order. It means that a transition is considered as FALSE if at least one of the transitions connected to the same divergence on its left side is TRUE.

Below is an example:



Transition 1 is crossed if:
step 1 is active
and *Cond1* is TRUE

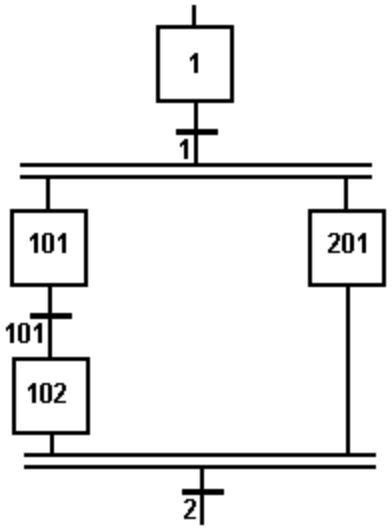
Transition 2 is crossed if:
step 1 is active
and *Cond2* is TRUE
and *Cond1* is FALSE

Warning

Some run-time systems can support exclusivity of the transitions within a divergence or not. Please refer to OEM instructions for further information about SFC support.

1.1.3 SFC parallel branches

Parallel branches are used in SFC charts to represent parallel operations. Parallel branches occur when more than several steps are connected after the same transition. Parallel branches are drawn as double horizontal lines:



When the transition before the divergence (1 on this example) is crossed, all steps beginning the parallel branches (101 and 201 here) are activated.

Sequencing of parallel branches can take different timing according to each branch execution.

The transition after the convergence (2 on this example) is crossed when all the steps connected before the convergence line (last step of each branch) are active. The transition indicates a synchronization of all parallel branches.

If needed, a branch can be finished with an "empty" step (with no action). It represents the state where the branch "waits" for the other ones to be completed.

You must take care of the following rules when drawing parallel lines in order to avoid dead locks in the execution of the program:

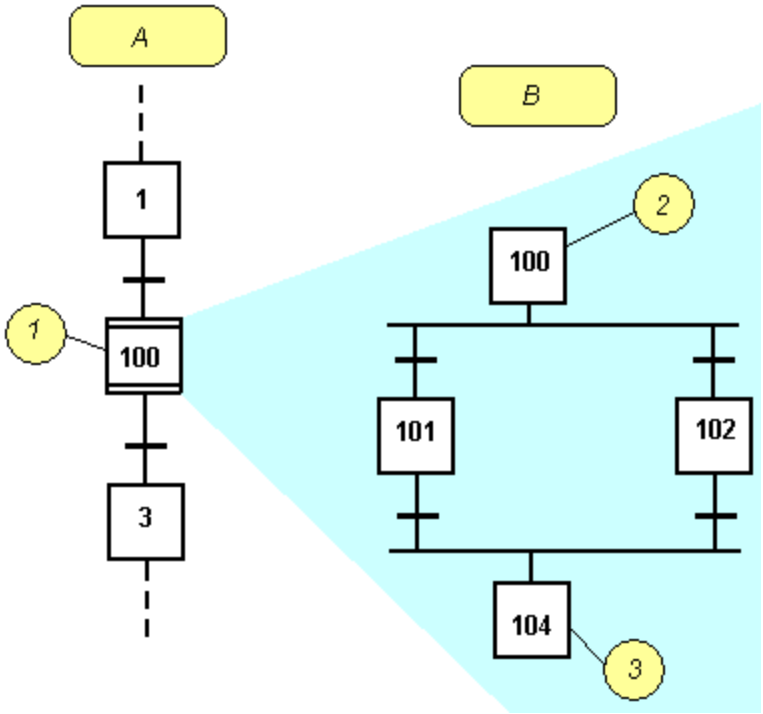
- All branches must be connected to the divergence and the convergence.
- An element of a branch must not be connected to an element outside the divergence.

How to create parallel branches?

To create double bars for a parallel branch you have to highlight a horizontal line and click the **Spacebar** to switch back and forth between single and double lines.

1.1.4 SFC macro steps

A macro step is a special symbol that represents, within an SFC chart, a part of the chart that begins with a step and ends with a step. The body of the macro-step must be declared in the same program. The body of a macro-step begins with a special "begin" step with no link before, and ends with a special "end" step with no link after. The symbol of the macros step in the main chart has double horizontal lines:



A: Main Chart
B: Body of the macro-step

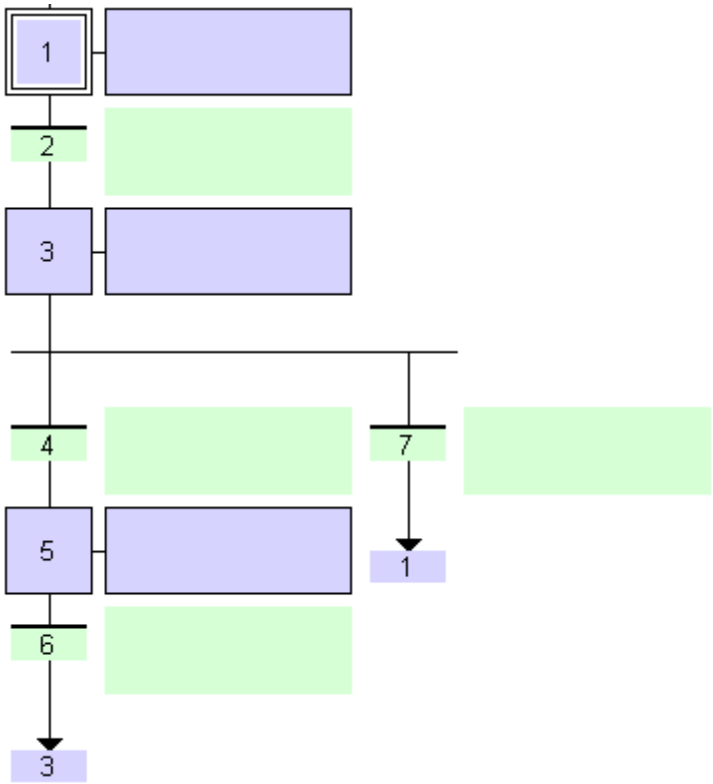
1: Macro step symbol
2: "Begin" step
3: "End" step

Warning

- The macro-step symbol and the beginning step must have the same number.
- The body of the macro-step must have no link with other parts of the main diagram (must be connected).
- A macro step is not a "sub program". It is just a drawing features that enables you to make clearer charts. You must never insert several macro-step symbols referring to the same macro-step body.

1.1.5 Jump to an SFC step

"Jump" symbols can be used in SFC charts to represent a link from a transition to a step without actually drawing it. The jump is represented by an arrow identified with the number of the target step.



Note

To change the number of a step, transition or jump, select it and hit **Ctrl+ENTER** keys.

You cannot insert a jump to a transition as it may lead to a non explicit convergence of parallel branches (several steps leading to the same transition) and generally leads to mistakes due to a bad understanding of the chart.

All parallel convergences must be explicitly drawn.

1.1.6 Actions in an SFC step

Each step has a list of action blocks, that are instructions to be executed according to the activity of the step. Actions can be simple boolean or SFC actions, that consists in assigning a boolean variable or control a child SFC program using the step activity, or action blocks entered using another language (FBD, FFLD, ST or IL).

Runtime check:

Below are the possible syntaxes you can use within an SFC step to perform runtime safety checks:

`__StepTimeout (...);`	Check for a timeout on the step activity duration.

Simple boolean actions:

Below are the possible syntaxes you can use within an SFC step to perform a simple boolean action:

`BoolVar (N);`	Forces the variable "BoolVar" to TRUE when the step is activated, and to FALSE when the step is deactivated.
`BoolVar (S);`	Sets the variable "BoolVar" to TRUE when step is activated
`BoolVar (R);`	Sets the variable "BoolVar" to FALSE when step is activated
`/ BoolVar;`	Forces the variable "BoolVar" to FALSE when the step is activated, and to TRUE when the step is deactivated.

Alarms:

The following syntax enables you to manage timeout alarm variables:

`BoolVar (A-, duration);`	Specifies a **Min** timeout variable to be associated to the step. - "BoolVar" must be a simple boolean variable - "duration" is the timeout, expressed either as a constant or as a single TIME variable (complex expressions cannot be used for this parameter) When the min timeout is elapsed, the alarm variable is turned to TRUE.
`BoolVar (A+, duration);`	Specifies a **Max** timeout variable to be associated to the step. - "BoolVar" must be a simple boolean variable - "duration" is the timeout, expressed either as a constant or as a single TIME variable (complex expressions cannot be used for this parameter) When the timeout is elapsed, the alarm variable is turned to TRUE, and the transition(s) following the step cannot be crossed until the alarm variable is reset to FALSE.
`BoolVar (A, duration);`	Another syntax to specify the **Max** timeout variable.

Simple SFC actions:

Below are the possible syntaxes you can use within an SFC step to control a child SFC program:

`Child (N);`	Starts the child program when the step is activated and stops (kills) it when the step is deactivated.
`Child (S);`	Starts the child program when the step is activated
`Child (R);`	Stops (kills) the child program when the step is activated

Programmed action blocks:

Programs in other languages (FBD, FFLD, ST or IL) can be entered to describe an SFC step action. There are three main types of programmed action blocks, that correspond to the following identifiers:

P1	Executed only once when the step becomes active
N	Executed on each cycle while the step is active
P0	Executed only once when the step becomes inactive

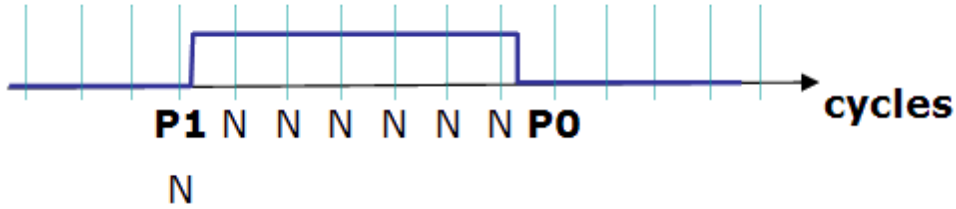


Figure 1-1: SFC step action blocks

The KAS IDE provides you templates for entering P1, N and P0 action blocks in either ST, FFLD or FBD language. Alternatively, you can insert action blocks programmed in ST language directly in the list of simple actions, using the following syntax:

ACTION (*qualifier*) :
statements...
END_ACTION;

Where *qualifier* is "`P1`", "`N`" or "`P0`".

1.1.7 Check timeout on an SFC step

SFC step implicitly contains a timer, so you do not need to add any timer function.

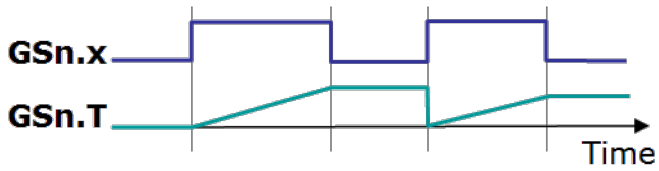


Figure 1-2: SFC Time Diagram - Timer vs Step Activation

The system can check timeout on any SFC step activity duration. For that, you need to enter the following instruction in the main "Action" list of the step:

```
__StepTimeout ( timeOut , errString );
```

Where:

timeout is a time constant or a time variable specifying the timeout duration
errString is a string constant or a string variable specifying the error message to be output

At runtime, each time the activation time of the step becomes greater than the specified timeout:

- The error string is sent to the KAS IDEand displayed in the Log window
- The transition is not passed

Note

You can also put this statement within a "`#ifdef __DEBUG`" test so that timeout checking is enabled only in debug mode.

Alternatively, if you need to make more specific handling of timeouts, you can enter the following ST program in the "N" action block of the step:

```
if GSn.T > timeout then /* 'n' is the number of the step */
...statements...
end_if;
```

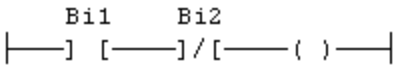
1.1.8 Condition of an SFC transition

Each SFC transitions must have a boolean condition that indicates if the transition can be crossed. The condition is a boolean expression that can be programmed either in ST or FFLD language.

In ST language, enter a boolean expression. It can be a complex expression including function calls and parentheses. For example:

```
bForce AND (bAlarm OR min (iLevel, 1) <> 1)
```

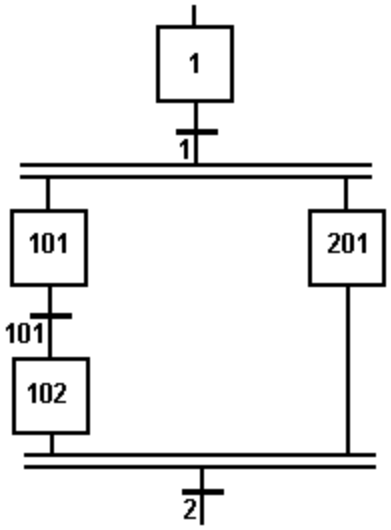
In FFLD language, the condition is represented by a single rung. The coil at the end of the rung represents the transition and must have no symbol attached. For example:



1.1.9 SFC execution at run time

SFC programs are executed sequentially within a target cycle, according to the order defined when entering programs in the hierarchy tree. A parent SFC program is executed before its children. This implies that when a parent starts or stops a child, the corresponding actions in the child program are performed during the same cycle.

Within a chart, all valid transitions are evaluated first, and then actions of active steps are performed. The chart is evaluated from the left to the right and from the top to the bottom. Below is an example:



Execution order:

- Evaluate transitions:

  1, 101, 2

- Manage steps:

  1, 101, 201, 102

In case of a divergence, all conditions are considered as exclusive, according to a "left to right" priority order. It means that a transition is considered as FALSE if at least one of the transitions connected to the same divergence on its left side is TRUE.

The initial steps define the initial status of the program when it is started. All top level (main) programs are started when the application starts. Child programs are explicitly started from action blocks within the parent programs.

The evaluation of transitions leads to changes of active steps, according to the following rules:

- A transition in crossed if:
  - its condition is TRUE
  - and if all steps linked to the top of the transition (before) are active
- When a transition is crossed:
  - all steps linked to the top of the transition (before) are deactivated
  - all steps linked to the bottom of the transition (after) are activated

Warning

Execution of SFC within the IEC 61131 target is sampled according to the target cycles. When a transition is crossed within a cycle, the following steps are activated, and the evaluation of the chart will continue on the next cycle. If several consecutive transitions are TRUE within a branch, only one of them is crossed within one target cycle.

Warning

• This section describes the execution model of a standard IEC 61131 target. SFC execution rules can differ for other target systems. Please refer to OEM instructions for further details about SFC execution at run time.
• Some run-time systems can support exclusivity of the transitions within a divergence or not. Please refer to OEM instructions for further information about SFC support.

1.1.10 Hierarchy of SFC programs

Each SFC program can have one or more "child programs". Child programs are written in SFC and are started (launched) or stopped (killed) in the actions of the father program. A child program can also have children. The number of hierarchy levels must not exceed 19.

When a child program is stopped, its children are also implicitly stopped.

When a child program is started, it must explicitly in its actions start its children.

A child program is controlled (started or stopped) from the action blocks of its parent program. Designing a child program is a simple way to program an action block in SFC language.

Using child programs is very useful for designing a complex process and separate operations due to different aspects of the process. For instance, it is common to manage the execution modes in a parent program and to handle details of the process operations in child programs.

1.1.11 Controlling a SFC child program

Controlling a child program can be simply achieved by specifying the name of the child program as an action block in a step of its parent program. Below are possible qualifiers that can be applied to an action block for handling a child program:

`Child (N);`	Starts the child program when the step is activated and stops (kills) it when the step is deactivated.
`Child (S);`	Starts the child program when the step is activated (Initial steps of the child program are activated)
`Child (R);`	Stops (kills) the child program when the step is activated (All active steps of the child program are deactivated)

Alternatively, you can use the following statements in an action block programmed in ST language. In the following table, "`prog`" represents the name of the child program:

`GSTART (prog);`	Starts the child program when the step is activated (Initial steps of the child program are activated)
`GKILL (prog);`	Stops (kills) the child program when the step is activated (All active steps of the child program are deactivated)
`GFREEZE (prog);`	Suspends the execution of a child program
`GRST (prog);`	Restarts a program suspended by a `GFREEZE`command.

You can also use the "GSTATUS" function in expressions. This function returns the current state of a child SFC program:

`GSTATUS (prog)`	Returns the current state of a child SFC program: 0: program is inactive 1: program is active 2: program is suspended

Note

When a child program is started by its parent program, it keeps the "inactive" status until it is executed (further in the cycle). If you start a child program in an SFC chart, GSTATUS will return 1 (active) on the next cycle.

1.1.12 User-Defined Function Blocks programmed in SFC

The KAS IDE enables you to create User-Defined Function Blocks (UDFBs) programmed with SFC language. This section details specific features related to such function blocks.

The execution of UDFBs written in SFC requires a runtime system version SR7-1 or later.

Declaration

From the Workspace contextual menu, run the "Insert New Program" command. Then specify a valid name for the function block. Select "SFC" language and "UDFB" execution style.

Parameters

When a UDFB programmed in SFC is created, the KAS IDEautomatically declares 3 special inputs to the block:

RUN: The SFC state machine is not activated when this input is FALSE.
RESET: The SFC chart is reset to its initial situation when this input is TRUE.
KILL: Any active step of the SFC chart is deactivated when this input is TRUE.

You can freely add other input and output variables to the UDFB. You can also remove any of the automatically created input if not needed. If the RUN input is removed, then it is considered as always TRUE. If RESET or KILL inputs are removed, then they are considered as always FALSE.

Below is the truth table showing priorities among special input:

RUN	RESET	KILL	
FALSE	FALSE	FALSE	do nothing
FALSE	FALSE	TRUE	kill the SFC chart
FALSE	TRUE	FALSE	reset the SFC chart
FALSE	TRUE	TRUE	kill the SFC chart
TRUE	FALSE	FALSE	activate the SFC chart
TRUE	FALSE	TRUE	kill the SFC chart
TRUE	TRUE	FALSE	reset the SFC chart
TRUE	TRUE	TRUE	kill the SFC chart

Steps

All steps inserted in the SFC chart of the UDFB are automatically declared as local instances of special reserved function blocks with the local variables of the UDFBs. The following FB types are used:

isfcSTEP : a normal step
isfcINITSTEP : an initial step

The editor takes care of updating the list of declared step instances. You should never remove, rename or change them in the variable editor. All steps are named with "GS" followed by their number.

Execution

The SFC chart is operated only when the UDFB is called by its parent program.

If the RESET input is TRUE, the SFC chart is reset to its initial situation. If the KILL input is TRUE, any active step of the SFC chart is deactivated.

When the RUN input is TRUE and KILL/RESET are FALSE, the SFC chart is operated in the same way as for other SFC programs:

1- Check valid transitions and evaluate related conditions
2- Cross TRUE valid transitions
3- Execute relevant actions of the active steps

Notes

In a UDFB programmed in SFC, you cannot use SFC actions to pilot a "child SFC program". This feature is reserved for SFC programs only. Instead, a UDFB programmed in SFC can pilot from its actions another UDFB programmed in SFC.

1.2 Function Block Diagram (FBD)

A function block Diagram is a data flow between constant expressions or variables and operations represented by rectangular blocks. Operations can be basic operations, function calls, or function block calls.

Use of ST instructions in graphic languages

The name of the operation or function, or the type of function block is written within the block rectangle. In case of a function block call, the name of the called instance is written in the header of the block rectangle, such as in the example below:

1.2.1 Data flow

The data flow represents values of any data type. All connections must be from input and outputs points having the same data type.

In case of a boolean connection, you can use a connection link terminated by a small circle, that indicates a boolean **negation** of the data flow. .

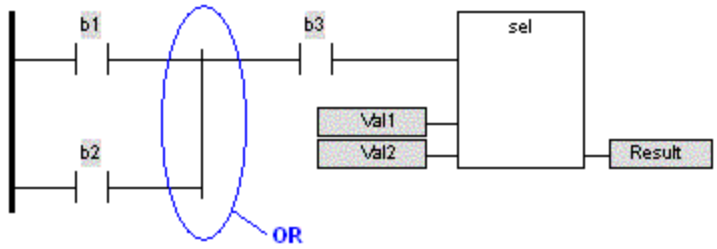
The data flow must be understood from the left to the right and from the top to the bottom. It is possible to use labels and jumps to change the default data flow execution.

1.2.2 FFLD symbols

FFLD symbols can also be entered in FBD diagrams and linked to FBD objects. Refer to the following sections for further information about components of the FFLD language:

Contacts
Coils
Power Rails

Special vertical lines are available in FBD language for representing the merging of FFLD parallel lines. Such vertical lines represent a OR operation between the connected inputs. Below is an example of an OR vertical line used in a FBD diagram:



1.3 Structured Text (ST)

ST is a structured literal programming language. A ST program is a list of *statements*. Each statement describes an action and must end with a semi-colon (";").

The presentation of the text has no meaning for a ST program. You can insert blank characters and line breaks where you want in the program text.

1.3.1 Comments

Comment texts can be entered anywhere in a ST program. Comment texts have no meaning for the execution of the program. A comment text must begin with "**(***" and end with "***)**". Comments can be entered on several lines (i.e. a comment text can include line breaks). Comment texts cannot be nested.

You can also use *//* to add a comment on a single line as shown below:

```
//My main comment
(* My comment *)
a := d + e;
(* A comment can also
be on several lines *)
b := d * e;

c := d - e; (* My comment *)
```

1.3.2 Expressions

Each statement describes an action and can include evaluation of complex expressions. An expression is evaluated:

- from the left to the right
- according to the default priority order of operators
- the default priority can be changed using parentheses

Arguments of an expression can be:
- declared variables
- constant expressions
- function calls

1.3.3 Statements

Below are available basic statements that can be entered in a ST program:

- assignment
- function block calling

Below are the available conditional statements in ST language:

- IF / THEN / ELSE
  Simple binary switch.
  One or several ELSIF are allowed.

```
IF a = b THEN
   c := 0;
ELSIF a < b THEN
   c := 1;
ELSE
   c := -1;
END_IF;
```

- CASE
  Switch between enumerated statements according to an expression.
  The selector can be any integer or a STRING.

```
CASE iChoice OF
0:
MyString := 'Nothing';
1 .. 2,5:
MyString := 'First case';
3,4:
MyString := 'Second case';
ELSE
MyString := 'Other case';
END_CASE;
```

Below are the available statements for describing loops in ST language:

Warning

Loop instructions can lead to infinite loops that block the target cycle.
Never test the state of an input in the condition as the input will not be refreshed before the next cycle.

- WHILE
 Repeat a list of statements.
 Condition is evaluated on loop **entry** before the statements.

```
iCount := 0;
WHILE iCount < 100 DO
iCount := iCount +1;
MyVar := MyVar + 1;
END_WHILE;
```

- REPEAT
 Repeat a list of statements.
 Condition is evaluated on loop **exit** after the statements.

```
iCount := 0;
REPEAT
MyVar := MyVar + 1;
iCount := iCount + 1;
UNTIL iCount < 100 END_REPEAT;
```

- FOR
 Iteration of statement execution.
 The **BY** statement is optional (default value is 1)

```
FOR iCount := 0 TO 100 BY 2 DO
MyVar := MyVar + 1;
END_FOR;
```

Note

Loops with FOR instructions are slow, so you can optimize your code by replacing such iterations with a WHILE statement.

Below are some other statements in ST language:

- WAIT / WAIT_TIME (suspend the execution)

- ON … DO (conditional execution of statements: provides a simpler syntax for checking the rising edge of a Boolean condition)

Tip

ST also provides an automatic completion of typed words. See .

1.4 Instruction List (IL)

This language is more appropriate when your algorithm refers to the Boolean algebra.

A program written in IL language is a list of *instructions*.
Each instruction is written on one line of text.
An instruction can have one or more *operands*.
Operands are variables or constant expressions.
Each instruction can begin with a label, followed by the "**:**" character.
Labels are used as destination for jump instructions.

The KAS IDE allows you to mix ST and IL languages in textual program. ST is the default language. When you enter IL instructions, the program must be entered between "BEGIN_IL" and "END_IL" keywords, such as in the following example

```
BEGIN_IL
FFLD    var1
ST    var2
END_IL
```

1.4.1 Comments

Comment texts can be entered at the end of a line containing an instruction. Comment texts have no meaning for the execution of the program. A comment text must begin with "**(***" and end with "***)**". Comments can also be entered on empty lines (with no instruction), and on several lines (i.e. a comment text can include line breaks). Comment texts cannot be nested.

```
(* My comment *)
LD a
ST b (* Store value in d *)
```

1.4.2 Data flow

An IL complete statement is made of instructions for:
- first: evaluating an expression (called *current result*)
- then: use the current result for performing actions

1.4.3 Evaluation of expressions

The order of instructions in the program is the one used for evaluating expressions, unless parentheses are inserted. Below are the available instructions for evaluation of expressions:

Instruction	Operand	Meaning
FFLD / FFLDN	any type	loads the operand in the current result
AND (&)	boolean	AND between the operand and the current result
OR / ORN	boolean	OR between the operand and the current result
XOR / XORN	boolean	XOR between the operand and the current result
ADD	numerical	adds the operand and the current result
SUB	numerical	subtract the operand from the current result
MUL	numerical	multiply the operand and the current result
DIV	numerical	divide the current result by the operand

Instruction	Operand	Meaning
GT	numerical	compares the current result with the operand
GE	numerical	compares the current result with the operand
LT	numerical	compares the current result with the operand
LE	numerical	compares the current result with the operand
EQ	numerical	compares the current result with the operand
NE	numerical	compares the current result with the operand
Function call	func. arguments	calls a function
Parenthesis		changes the execution order

Note

Instructions suffixed by **N** uses the boolean negation of the operand.

1.4.4 Actions

The following instructions perform actions according to the value of current result. Some of these instructions do not need a current result to be evaluated:

Instruction	Operand	Meaning
ST / STN	any type	stores the current result in the operand
JMP	label	jump to a label - no current result needed
JMPC	label	jump to a label if the current result is TRUE
JMPNC / JMPCN	label	jump to a label if the current result is FALSE
RET		Jump to the end of the current program - no current result needed
RETC / RETNC / RETCN		Jump to the end of the current program if the current result is TRUE / FALSE
S	boolean	sets the operand to TRUE if the current result is TRUE
R	boolean	sets the operand to FALSE if the current result is TRUE
CAL	f. block	calls a function block (no current result needed)
CALC	f. block	calls a function block if the current result is TRUE
CALNC / CALCN	f. block	calls a function block if the current result is FALSE

Note

Instructions suffixed by **N** uses the boolean negation of the operand.

Note

IL program cannot be called if there is no entry variable, or if its type is complex (e.g. array)

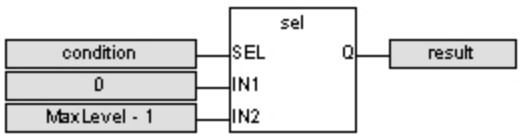
1.5 Use of ST expressions in graphic language

The KAS IDE enables any complex ST expression to be associated with a graphic element in either FFLD or FBD language. This feature makes possible to simplify FFLD and FBD diagrams when some trivial calculation has to be entered. It also enables you to use graphic features for representing a main algorithm as text is used for details of implementation.

Expression must be written in ST language. An expression is anything you can imagine between parentheses in a ST program. Obviously the ST expression must fit the data type required by the diagram (e.g. an expression put on a contact must be boolean).

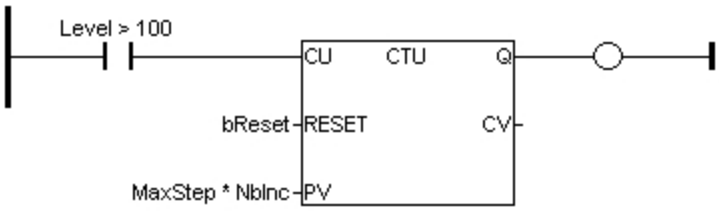
FBD language:

A complex ST expression can be entered in any "variable box" of a FBD diagram, if the box is not connected on its input. Below is an example:



FFLD language:

A complex ST expression can be entered on any kind of contact, and on any input of a function or function block. Below is an example:



1.6 Free Form Ladder Diagram (FFLD)

A Ladder Diagram is a list of *rungs*. Each rung represents a boolean data flow from a power rail on the left. The power rail represents the TRUE state. The data flow must be understood from the left to the right. Each symbol connected to the rung either changes the rung state or performs an operation. Below are possible graphic items to be entered in FFLD diagrams:

Power Rails
Contacts and Coils
Operations, Functions and Function blocks, represented by rectangular blocks
Labels and Jumps
Use of ST instructions in graphic languages

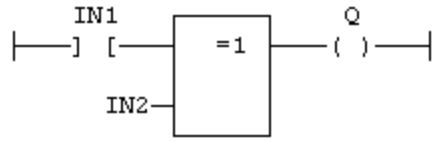
Use of the "EN" input and the "ENO" output for blocks

The rung state in a FFLD diagram is always boolean. Blocks are connected to the rung with their first input and output. This implies that special "EN" and "ENO" input and output are added to the block if its first input or output is not boolean.

The "EN" input is a condition. It means that the operation represented by the block is not performed if the rung state (EN) is FALSE. The "ENO" output always represents the sane status as the "EN" input: the rung state is not modified by a block having an ENO output.

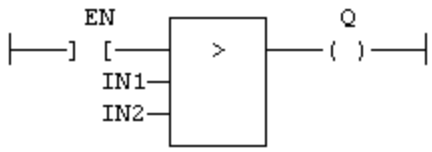
Below is the example of the "XOR" block, having boolean inputs and outputs, and requiring no EN or ENO pin:

(* First input is the rung. The rung is the output *)



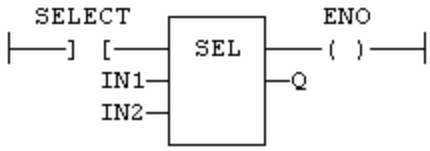
Below is the example of the ">" (greater than) block, having non boolean inputs and a boolean output. This block has an "EN" input in FFLD language:

(* The comparison is executed only if EN is TRUE *)



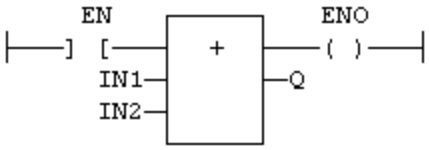
Below is the example of the "SEL" function, having a first boolean input, but an integer output. This block has an "ENO" output in FFLD language:

(* the input rung is the selector *)
(* ENO has the same value as SELECT *)



Finally, below is the example of an addition, having only numerical arguments. This block has both "EN" and "ENO" pins in FFLD language:

(* The addition is executed only if EN is TRUE *)
(* ENO is equal to EN *)



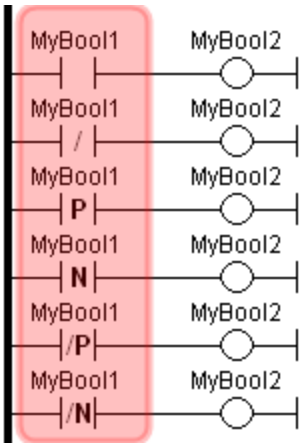
1.6.1 *Contacts and coils*

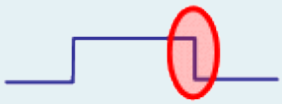
The table below contains a list of the contact and coil types available:

Contacts	Coils
Normally Open -| |-	Energize -( )-
Normally Closed -|/|-	De-energize -(/)-
Positive Transition -|P|-	Set (Latch) -(S)-
Negative Transition -|N|-	Reset (Unlatch) -(R)-
Normally closed positive transition -|/P|-	Positive transition sensing coil -(P)-
Normally closed negative transition -|/N|-	Negative transition sensing coil -(N)-

Contacts are basic graphic elements of the FFLD language. A contact is associated with a boolean variable which is displayed above the graphic symbol. A contact sets the state of the rung on its right-hand side, according to the value of the associated variable and the rung state on its left-hand side.

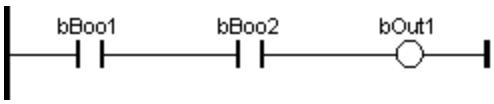
Below are the six possible contact symbols and how they change the flow:



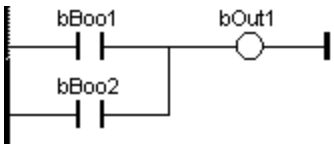
Contacts	Description
boolVariable -] [-	**Normal**: the flow on the right is the boolean AND operation between: (1) the flow on the left and (2) the associated variable.
boolVariable -]/[-	**Negated**: the flow on the right is the boolean AND operation between: (1) the flow on the left and (2) the negation of the associated variable.
boolVariable -]P[-	**Positive pulse**: the flow on the right is TRUE only when the flow on the left is TRUE and the associated variable changes from FALSE to TRUE (rising edge) 
boolVariable -]N[-	**Negative pulse**: the flow on the right is TRUE only when the flow on the left is TRUE and the associated variable changes from TRUE to FALSE (falling edge) 
boolVariable -]/P[-	**Normally Closed Positive pulse**: the flow on the right is TRUE only when the flow on the left is TRUE and the negation of the associated variable changes from FALSE to TRUE (rising edge)
boolVariable -]/N[-	**Normally Closed Negative pulse**: the flow on the right is TRUE only when the flow on the left is TRUE and the negation of the associated variable changes from TRUE to FALSE (falling edge)

Serialized and Parallel contacts

Two serial normal contacts represent an AND operation.



Two contacts in parallel represent an OR operation.



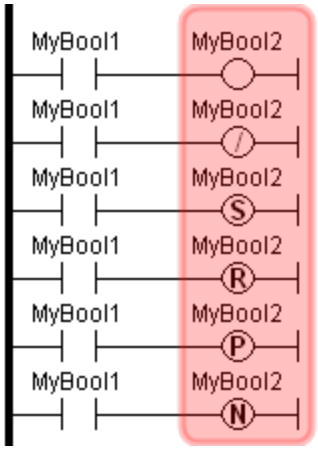
About Pulse

Each pulse is a single instance having its own memory.

After the pulse has been evaluated, its memory contains the previous value. Conversely, if a pulse is not evaluated during a scan, its memory is not updated.

Coils are basic graphic elements of the FFLD language. A coil is associated with a boolean variable which is displayed above the graphic symbol. A coil performs a change of the associated variable according to the flow on its left-hand side.

Below are the six possible coil symbols:



Coils	Description
`boolVariable -( )-`	**Normal**: the associated variable is forced to the value of the flow on the left of the coil.
`boolVariable -(/)-`	**Negated**: the associated variable is forced to the negation of the flow on the left of the coil.

`boolVariable -(S)-`	**Set**: the associated variable is forced to TRUE if the flow on the left is TRUE. (no action if the flow is FALSE)
	Rules for Set coil animation:
	• Power Flow on left is TRUE:
	  ◦ The horizontal wires on either side of the (S) are red
	  ◦ The variable and the (S) are red
	• Power Flow on left is FALSE and the (S) variable is Energized (ON)
	  ◦ The horizontal lines on either sided of (S) are black
	  ◦ The variable and the (S) are red
	• In all other cases:
	  ◦ The horizontal wires are black
	  ◦ The variable and the (S) are black

`boolVariable -(R)-`	**Reset**: the associated variable is forced to FALSE if the flow on the left is TRUE. (no action if the rung state is FALSE)
	Rules for Reset coil animation:
	• Power Flow on left is TRUE:
	  ◦ The horizontal lines are red
	  ◦ The variable above (R) is black
	  ◦ The R and the circle around the R are black
	• Power Flow on left is FALSE and variable above reset coil is NOT Energized (OFF)
	  ◦ The horizontal lines are black
	  ◦ The variable above (R) is black
	  ◦ The R and the circle around the R are black
	• Power Flow on left is FALSE and variable above reset coil is Energized (ON)
	  ◦ The horizontal lines are black
	  ◦ The variable above (R) is red
	  ◦ The R and the circle around the R are red

`boolVariable -(P)-`	**Positive transition**: the associated variable is forced to TRUE if the flow on the left changes from **FALSE to TRUE**(and forced to FALSE in all other cases)

Coils	Description
boolVariable -(N)-	**Negative transition**: the associated variable is forced to TRUE if the flow on the left changes from **TRUE to FALSE**(and forced to FALSE in all other cases)

Tip

When a contact or coil is selected, you can press the **Spacebar** to change its type (normal, negated...)
When your application is running, you can select a contact and press the **Spacebar** to swap its value between TRUE and FALSE

Warning

Although coils are commonly put at the end, the rung can be continued after a coil. The flow is **never changed** by a coil symbol.

1.6.2 Power Rails

Vertical power rails are used in FFLD language for designing the limits of a rung.

The power rail on the left represents the TRUE value and initiates the rung state. The power rail on the right receives connections from the coils and has no influence on the execution of the program.

Power rails can also be used in FBD language. Only boolean objects can be connected to left and right power rails.

See also

Contacts Coils

This page intentionally left blank.

2 Programming features and standard blocks

Refer to the following pages for an overview of the IEC 61131-3 programming languages:

Program organization units
Data types
Structures
Variables
Arrays
Constant expressions
Conditional compiling
Handling exceptions

SFC: Sequential Function Chart
FBD: Function Block Diagram
FFLD: Free Form Ladder Diagram
ST: Structured Text
IL: Instruction List
Use of ST instructions in graphic languages

The following topics detail the set of programming features and standard blocks:

Basic operations
Boolean operations
Arithmetic operations
Comparisons
Type conversion functions
Selectors
Registers
Counters
Timers
Maths
Trigonometrics
String operations
Advanced

Note: Some other functions not documented here are reserved for diagnostics and special operations. Please contact your technical support for further information.

2.1 Basic Operations

Below are the language features for basic data manipulation:

- Variable assignment
- Bit access
- Parenthesis
- Calling a function
- Calling a function block
- Calling a sub-program
- MOVEBLOCK: Copying/moving array items
- COUNTOF: Number of items in an array
- INC: Increase a variable
- DEC: decrease a variable
- NEG: integer negation (unary operator)

Below are the language features for controlling the execution of a program:

- Labels
- Jumps

- RETURN

Below are the structured statements for controlling the execution of a program:

IF	Conditional execution of statements.
WHILE	Repeat statements while a condition is TRUE.
REPEAT	Repeat statements until a condition is TRUE.
FOR	Execute iterations of statements.
CASE	Switch to one of various possible statements.
EXIT	Exit from a loop instruction.
WAIT	Delay program execution.
ON	Conditional execution.

2.1.1 := FFLD FFLDN ST STN

Operator - variable assignment.

2.1.1.1 Inputs

`IN : ANY` Any variable or complex expression

2.1.1.2 Outputs

`Q : ANY` Forced variable

2.1.1.3 Remarks

The output variable and the input expression must have the same type. The forced variable cannot have the "read only" attribute. In FFLD and FBD languages, the "1" block is available to perform a "1 gain" data copy (1 copy). In FFLD language, the input rung (EN) enables the assignment, and the output rung keeps the state of the input rung. In IL language, the FFLD instruction loads the first operand, and the ST instruction stores the current result into a variable. The current result and the operand of ST must have the same type. Both FFLD and ST instructions can be modified by "N" in case of a boolean operand for performing a boolean negation.

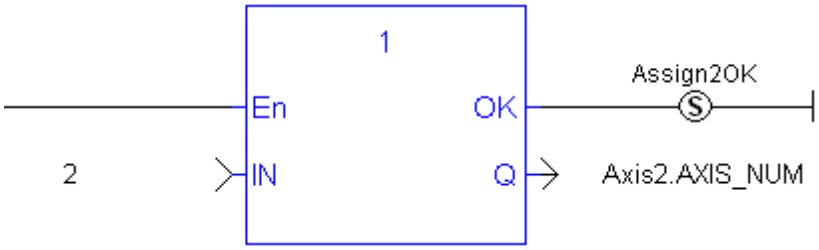
2.1.1.4 ST Language

```
Q := IN; (* copy IN into variable Q *)
 Q := (IN1 + (IN2 / IN 3)) * IN4; (* assign the result of a complex expression *)
 result := SIN (angle); (* assign a variable with the result of a function *)
 time := MyTon.ET; (* assign a variable with an output parameter of a function block
*)
```

2.1.1.5 FBD Language

2.1.1.6 FFLD Language

```
(* The copy is executed only if EN is TRUE *)
```



2.1.1.7 IL Language:

```
Op1: FFLD  IN   (* current result is: IN *)
 ST  Q    (* Q is: IN *)
 FFLDN IN1  (* current result is: NOT (IN1) *)
 ST  Q    (* Q is: NOT (IN1) *)
 FFLD  IN2  (* current result is: IN2 *)
 STN Q    (* Q is: NOT (IN2) *)
```

See also:

Parenthesis

2.1.2 Access to bits of an integer

You can directly specify a bit within n integer variable in expressions and diagrams, using the following notation:

Variable.BitNo

Where:

Variable: is the name of an integer variable
BitNo: is the number of the bit in the integer.

The variable can have one of the following data types:

SINT, USINT, BYTE (8 bits from .0 to .7)
INT, UINT, WORD (16 bits from .0 to .15)
DINT, UDINT, DWORD (32 bits from .0 to 31)
LINT, ULINT, LWORD, (64 bits from 0 to 63)

0 always represents the less significant bit.

2.1.3 Calling a function

A function ( ) calculates a result according to the current value of its inputs. A function has no internal data and is not linked to declared instances, unlike a function block. A function has only one output: the result of the function. A function can be:

- a standard function (SHL, SIN...)
- a function written in "C" language and embedded on the target

2.1.3.1 ST Language

To call a function block in ST, you have to enter its name, followed by the input parameters written between parentheses and separated by comas. The function call can be inserted into any complex expression. A function call can be used as an input parameter of another function. The following example demonstrates a call to "ODD" and "SEL" functions:

```
(* the following statement converts any odd integer value into the
nearest even integer *)
iEvenVal := SEL ( ODD( iValue ), iValue, iValue+1 );
```

2.1.3.2 FBD and FFLD Languages

To call a function block in FBD or FFLD languages, you just need to insert the function in the diagram and to connect its inputs and output.

2.1.3.3 IL Language:

To call a function block in IL language, you must load its first input parameter before the call, and then use the function name as an instruction, followed by the other input parameters, separated by comas. The result of the function is then the current result. The following example demonstrates a call to "ODD" and "SEL" functions:

```
(* the following statement converts any odd integer into "0" *)
Op1: FFLD   iValue
     ODD
     SEL    iValue, 0
     ST    iResult
```

See also:

Differences Between Functions and Function Blocks

2.1.4 Calling a function block CAL CALC CALNC CALCN

A function block () groups an algorithm and a set of private data. It has inputs and outputs. A function block can be:

- a standard function block (RS, TON...)
- a block written in "C" language and embedded on the target
- a User-Defined Function Block (UDFB) written in ST, FBD, FFLD or IL

To use a function block, you have to declare an instance of the block as a variable, identified by a unique name. Each instance of a function block as its own set of private data and can be called separately. A call to a function block instance processes the block algorithm on the private data of the instance, using the specified input parameters.

Tip

Best Practice: It is recommended that function blocks be put in an N step and not in P0 or P1, as those steps are executed only once. If you must use an FB in P0 or P1 be sure to call it again in the N state so it may finish.

2.1.4.1 ST Language

To call a function block in ST, you have to specify the name of the instance, followed by the input parameters written between parentheses and separated by comas. To have access to an output parameter, use the name of the instance followed by a dot '.' and the name of the wished parameter. The following example demonstrates a call to an instance of TON function block:

```
(* MyTimer is declared as an instance of TON *)
MyTimer (bTrig, t#2s); (* calls the function block *)
TimerOutput := MyTimer.Q;
ElapsedTime := MyTimer.ET;
```

2.1.4.2 FBD and FFLD Languages

To call a function block in FBD or FFLD languages, you just need to insert the block in the diagram and to connect its inputs and outputs. The name of the instance must be specified upon the rectangle of the block.

2.1.4.3 IL Language

To call a function block in IL language, you must use the CAL instruction, and use a declared instance of the function block. The instance name is the operand of the CAL instruction, followed by the input parameters written between parentheses and separated by comas. Alternatively the CALC, CALCN or CALNC conditional instructions can be used:

CAL	calls the function block
CALC	calls the function block if the current result is TRUE
CALNC	calls the function block if the current result is FALSE
CALCN	same as CALNC

The following example demonstrates a call to an instance of TON function block:

```
(* MyTimer is declared as an instance of TON *)
Op1: CAL   MyTimer (bTrig, t#2s)
FFLD       MyTimer.Q
ST         TimerOutput
FFLD       MyTimer.ET
ST         ElapsedTimer

Op2: FFLD  bCond
CALC  MyTimer (bTrig, t#2s)  (* called only if bCond is TRUE *)
Op3: FFLD  bCond
CALNC  MyTimer (bTrig, t#2s)  (* called only if bCond is FALSE *)
```

See also:

Differences Between Functions and Function Blocks

2.1.5 Calling a sub-program

A sub-program is called by another program. Unlike function blocks, local variables of a sub-program are not instantiated, and thus you do not need to declare instances. A call to a sub-program processes the block algorithm using the specified input parameters. Output parameters can then be accessed.

2.1.5.1 ST Language

To call a sub-program in ST, you have to specify its name, followed by the input parameters written between parentheses and separated by comas. To have access to an output parameter, use the name of the sub-program followed by a dot '.' and the name of the wished parameter:

```
MySubProg (i1, i2); (* calls the sub-program *)
Res1 := MySubProg.Q1;
Res2 := MySubProg.Q2;
```

Alternatively, if a sub-program has one and only one output parameter, it can be called as a function in ST language:

```
Res := MySubProg (i1, i2);
```

2.1.5.2 FBD and FFLD Languages

To call a sub-program in FBD or FFLD languages, you just need to insert the block in the diagram and to connect its inputs and outputs.

2.1.5.3 IL Language

To call a sub-program in IL language, you must use the CAL instruction with the name of the sub-program, followed by the input parameters written between parentheses and separated by comas. Alternatively the CALC, CALCN or CALNC conditional instructions can be used:

`CAL`	Calls the sub-program
`CALC`	Calls the sub-program if the current result is TRUE
`CALNC`	Calls the sub-program if the current result is FALSE
`CALCN`	same as `CALNC`

Here is an example:

```
Op1: CAL  MySubProg (i1, i2)
FFLD  MySubProg.Q1
ST  Res1
FFLD  MySubProg.Q2
ST  Res2
```

2.1.6 CASE OF ELSE END_CASE

Statement - switch between enumerated statements.

2.1.6.1 Syntax

```
CASE <DINT expression> OF
<value> :
    <statements>
<value> , <value> :
    <statements>;
<value> .. <value> :
    <statements>;
ELSE
    <statements>
END_CASE;
```

2.1.6.2 Remarks

All enumerated values correspond to the evaluation of the DINT expression and are possible cases in the execution of the statements. The statements specified after the ELSE keyword are executed if the expression takes a value which is not enumerated in the switch. For each case, you must specify either a value, or a list of possible values separated by comas (",") or a range of values specified by a "min .. max" interval. You must enter space characters before and after the ".." separator.

2.1.6.3 ST Language

```
(* this example check first prime numbers *)
CASE iNumber OF
0 :
  Alarm := TRUE;
  AlarmText := '0 gives no result';
1 .. 3, 5 :
  bPrime := TRUE;
4, 6 :
  bPrime := FALSE;
ELSE
  Alarm := TRUE;
  AlarmText := 'I don't know after 6 !';
END_CASE;
```

2.1.6.4 FBD Language

Not available

2.1.6.5 FFLD Language

Not available

2.1.6.6 IL Language

Not available

See also

IF WHILE REPEAT FOR EXIT

2.1.7 COUNTOF

Function - Returns the number of items in an array

2.1.7.1 Inputs

`ARR : ANY` Declared array

2.1.7.2 Outputs

`Q : DINT` Total number of items in the array

2.1.7.3 Remarks

The input must be an array and can have any data type. This function is particularly useful to avoid writing directly the actual size of an array in a program, and thus keep the program independent from the declaration. Example:

```
FOR i := 1 TO CountOf (MyArray) DO
   MyArray[i-1] := 0;
END_FOR;
```

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

Examples

array	return
Arr1 [0..9]	10
Arr2 [0..4 , 0..9]	50

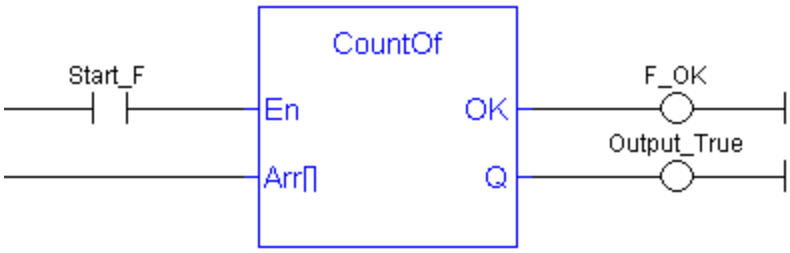
2.1.7.4 ST Language

```
Q := CountOf (ARR);
```

2.1.7.5 FBD Language

2.1.7.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.1.7.7 IL Language

Not available

2.1.8 DEC

Function - Decrease a numerical variable

2.1.8.1 Inputs

`IN : ANY` Numerical variable (increased after call).

2.1.8.2 Outputs

`Q : ANY` Decreased value

2.1.8.3 Remarks

When the function is called, the variable connected to the "IN" input is decreased and copied to Q. All data types are supported except BOOL and STRING: for these types, the output is the copy of IN.

For real values, variable is decreased by "1.0". For time values, variable is decreased by 1 ms.

The IN input must be directly connected to a variable, and cannot be a constant or complex expression.

This function is particularly designed for ST language. It allows simplified writing as assigning the result of the function is not mandatory.

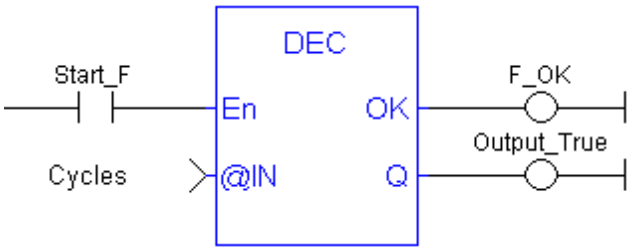
2.1.8.4 ST Language

```
IN := 2;
Q := DEC (IN);
(* now: IN = 1 ; Q = 1 *)

DEC (IN); (* simplified call *)
```

2.1.8.5 FBD Language

2.1.8.6 FFLD Language



2.1.8.7 IL Language

not available

2.1.9 EXIT

Statement - Exit from a loop statement

2.1.9.1 Remarks

The EXIT statement indicates that the current loop (WHILE, REPEAT or FOR) must be finished. The execution continues after the END_WHILE, END_REPEAT or END_FOR keyword or the loop where the EXIT is. EXIT quits only one loop and cannot be used to exit at the same time several levels of nested loops.

Warning

loop instructions can lead to infinite loops that block the target cycle.

2.1.9.2 ST Language

```
(* this program searches for the first non null item of an array *)
iFound = -1; (* means: not found *)
FOR iPos := 0 TO (iArrayDim - 1) DO
   IF iPos <> 0 THEN
      iFound := iPos;
      EXIT;
   END_IF;
END_FOR;
```

2.1.9.3 FBD Language

Not available

2.1.9.4 FFLD Language

Not available

2.1.9.5 IL Language

Not available

See also

IF WHILE REPEAT FOR CASE

2.1.10 FOR TO BY END_FOR

Statement - Iteration of statement execution.

2.1.10.1 Syntax

```
FOR <index> := <minimum> TO <maximum> BY <step> DO
  <statements>
END_FOR;
```

index = DINT internal variable used as index
minimum = DINT expression: initial value for *index*
maximum = DINT expression: maximum allowed value for *index*
step = DINT expression: increasing step of *index* after each iteration (default is 1)

2.1.10.2 Remarks

The "BY <*step*>" statement can be omitted. The default value for the step is 1.

2.1.10.3 ST Language

```
iArrayDim := 10;

(* resets all items of the array to 0 *)
FOR iPos := 0 TO (iArrayDim - 1) DO
   MyArray[iPos] := 0;
END_FOR;

(* set all items with odd index to 1 *)
FOR iPos := 1 TO 9 BY 2 DO
   MyArray[ipos] := 1;
END_FOR;
```

2.1.10.4 FBD Language

Not available

2.1.10.5 FFLD Language

Not available

2.1.10.6 IL Language

Not available

See also

IF WHILE REPEAT CASE EXIT

2.1.11 IF THEN ELSE ELSIF END_IF

Statement - Conditional execution of statements.

2.1.11.1 Syntax

```
IF <BOOL expression> THEN
<statements>
ELSIF <BOOL expression> THEN
<statements>
ELSE
<statements>
END_IF;
```

2.1.11.2 Remarks

The IF statement is available in ST only. The execution of the statements is conditioned by a boolean expression. ELSIF and ELSE statements are optional. There can be several ELSIF statements.

2.1.11.3 ST Language

```
(* simple condition *)
          IF bCond THEN
Q1 := IN1;
Q2 := TRUE;
END_IF;

(* binary selection *)
          IF bCond THEN
Q1 := IN1;
Q2 := TRUE;
ELSE
Q1 := IN2;
Q2 := FALSE;
END_IF;

(* enumerated conditions *)
IF bCond1 THEN
Q1 := IN1;
ELSIF bCond2 THEN
Q1 := IN2;
ELSIF bCond3 THEN
Q1 := IN3;
ELSE
Q1 := IN4;
END_IF;
```

2.1.11.4 FBD Language

Not available

2.1.11.5 FFLD Language

Not available

2.1.11.6 IL Language

Not available

See also

WHILE REPEAT FOR CASE EXIT

2.1.12 INC

Function - Increase a numerical variable

2.1.12.1 Inputs

`IN : ANY` Numerical variable (increased after call).

2.1.12.2 Outputs

`Q : ANY` Increased value

2.1.12.3 Remarks

When the function is called, the variable connected to the "IN" input is increased and copied to Q. All data types are supported except BOOL and STRING: for these types, the output is the copy of IN.

For real values, variable is increased by "1.0". For time values, variable is increased by 1 ms.

The IN input must be directly connected to a variable, and cannot be a constant or complex expression.

This function is particularly designed for ST language. It allows simplified writing as assigning the result of the function is not mandatory.

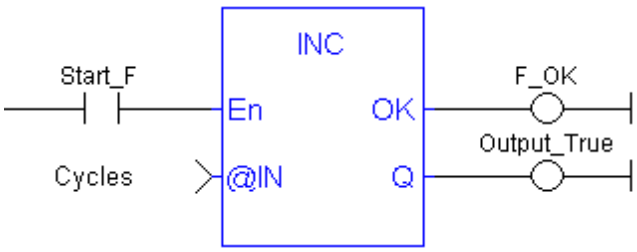
2.1.12.4 ST Language

```
IN := 1;
Q := INC (IN);
(* now: IN = 2 ; Q = 2 *)

INC (IN); (* simplified call *)
```

2.1.12.5 FBD Language

2.1.12.6 FFLD Language



2.1.12.7 IL Language

not available

2.1.13 Jumps JMP JMPC JMPNC JMPCN

Statement - Jump to a label.

2.1.13.1 Remarks

A jump to a label branches the execution of the program after the specified label.

In ST language, labels and jumps cannot be used.

In FBD language, a jump is represented by a signpost containing the label name. The input of the signpost must be connected to a valid boolean signal. The jump is performed only if the input is TRUE.

In FFLD language, the "-->>" symbol, followed by the target label name, is used as a coil at the end of a rung. The jump is performed only if the rung state is TRUE.

In IL language, JMP, JMPC, JMPCN and JMPNC instructions are used to specify a jump. The destination label is the operand of the jump instruction.

Warning

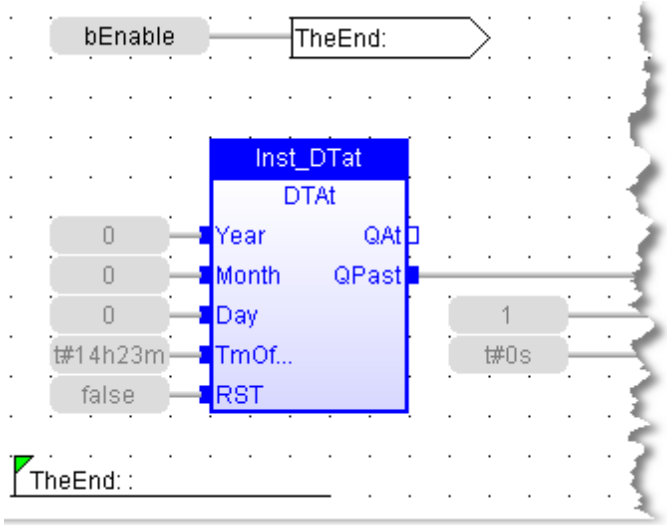
Backward jumps can lead to infinite loops that block the target cycle.

2.1.13.2 ST Language

Not available

2.1.13.3 FBD Language

```
(* In this example the TON block will not be called if bEnable is TRUE *)
```

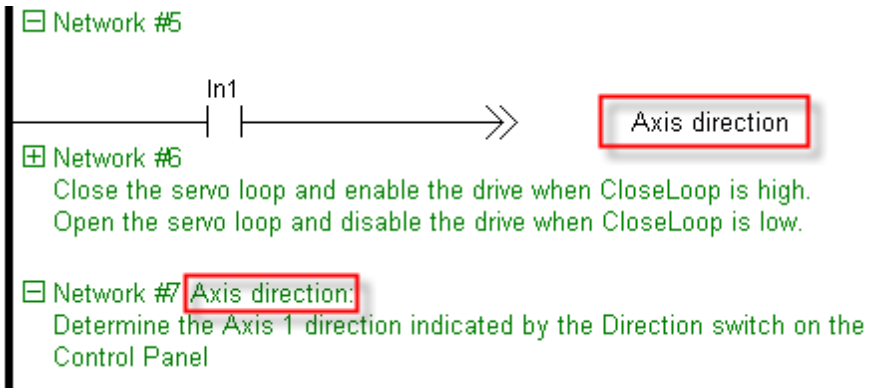


2.1.13.4 FFLD Language

Each rung can begin with a label.

Labels are used as destination for jump instructions.

In this example the network #6 is skipped if IN1 is TRUE.



2.1.13.5 IL Language

Below is the meaning of possible jump instructions:

`JMP`	Jump always
`JMPC`	Jump if the current result is TRUE
`JMPNC`	Jump if the current result is FALSE
`JMPCN`	Same as `JMPNC`

```
(* My comment *)

Start:   FFLD   IN1
         JMPC   TheRest (* Jump to "TheRest" if IN1 is TRUE *)
```

```
          FFLD    IN2      (* these three instructions are not
executed *)
          ST      Q2       (* if IN1 is TRUE *)
          JMP     TheEnd   (* unconditional jump to "TheEnd" *)

TheRest:  FFLD    IN3
          ST      Q3
TheEnd:
```

See also

Labels RETURN

2.1.14 LABELS

Statement - Destination of a Jump instruction.

2.1.14.1 Remarks

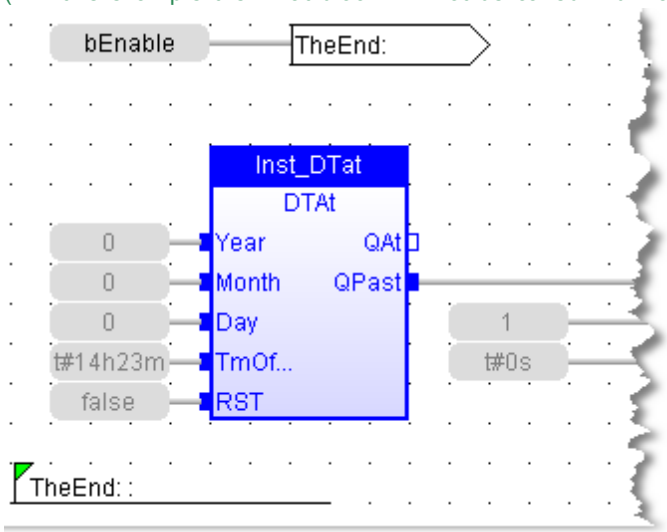
Labels are used as a destination of a jump instruction in FDB, FFLD or IL language. Labels and jumps cannot be used in structured ST language. A label must be represented by a unique name, followed by a colon (":"). In FBD language, labels can be inserted anywhere in the diagram, and are connected to nothing. In FFLD language, a label must identify a rung, and is shown on the left side of the rung. In IL language, labels are destination for JMP, JMPC, JMPCN and JMPNC instructions. They must be written before the instruction at the beginning of the line, and must index the beginning of a valid IL statement: FFLD (load) instruction, or unconditional instructions such as CAL, JMP or RET. The label can also be written alone on a line before the indexed instruction. In all languages, it is not mandatory that a label be a target of a jump instruction. You can also use label for marking parts of the programs in order to increase its readability.

2.1.14.2 ST Language

Not available

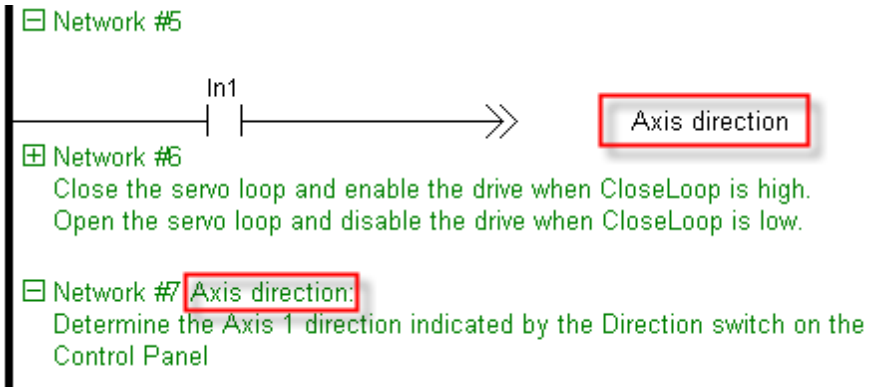
2.1.14.3 FBD Language

(* In this example the DTat block will not be called if bEnable is TRUE *)



2.1.14.4 FFLD Language

In this example the network #6 is skipped if IN1 is TRUE.



2.1.14.5 IL Language

```
Start:    FFLD    IN1      (* unused label - just for readability *)
          JMPC    TheRest  (* Jump to "TheRest" if IN1 is TRUE *)

          FFLD    IN2      (* these two instructions are not executed
*)
          ST      Q2       (* if IN1 is TRUE *)

TheRest:  FFLD    IN3      (* label used as the jump destination *)
          ST      Q3
```

See also

Jumps RETURN

2.1.15 MOVEBLOCK

Function - Move/Copy items of an array.

2.1.15.1 Inputs

`SRC: ANY (*)`	Array containing the source of the copy
`DST : ANY (*)`	Array containing the destination of the copy
`PosSRC: DINT`	Index of the first character in SRC
`PosDST : DINT`	Index of the destination in DST
`NB : DINT`	Number of items to be copied

(*) `SRC and DST` cannot be a STRING

2.1.15.2 Outputs

`OK : BOOL`	TRUE if successful

2.1.15.3 Remarks

Arrays of string are not supported by this function.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The function is not available in IL language.

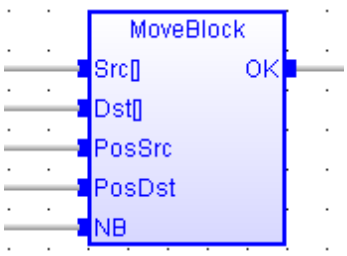
The function copies a number (NB) of consecutive items starting at the PosSRC index in SRC array to PosDST position in DST array. SRC and DST can be the same array. In that case, the function avoids lost items when source and destination areas overlap.

This function checks array bounds and is always safe. The function returns TRUE if successful. It returns FALSE if input positions and number do not fit the bounds of SRC and DST arrays.

2.1.15.4 ST Language

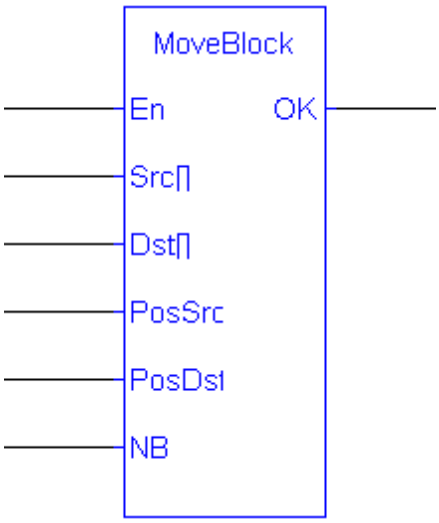
```
OK := MOVEBLOCK (SRC, DST, PosSRS, PosDST, NB);
```

2.1.15.5 FBD Language



2.1.15.6 FFLD Language

(* The function is executed only if EN is TRUE *)



2.1.15.7 IL Language

Not available

2.1.16 NEG -

Operator - Performs an integer negation of the input.

2.1.16.1 Inputs

`IN : DINT` Integer value

2.1.16.2 Outputs

`Q : DINT` Integer negation of the input

2.1.16.3 Truth table (examples)

IN	Q
0	0
1	-1
-123	123

2.1.16.4 Remarks

In FBD and FFLD language, the block "NEG" can be used.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

This feature is not available in IL language. In ST language, "-" can be followed by a complex boolean expression between parentheses.

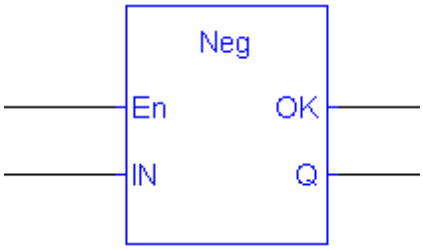
2.1.16.5 ST Language

```
Q := -IN;
Q := - (IN1 + IN2);
```

2.1.16.6 FBD Language

2.1.16.7 FFLD Language

```
(* The negation is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.1.16.8 IL Language

Not available

2.1.17 ON

Statement - Conditional execution of statements.

The ON instruction provides a simpler syntax for checking the rising edge of a Boolean condition.

2.1.17.1 Syntax

```
ON <BOOL expression> DO
    <statements>
END_DO;
```

2.1.17.2 Remarks

Statements within the ON structure are executed only when the boolean expression rises from FALSE to TRUE. The ON instruction avoids systematic use of the R_TRIG function block or other "last state" flags.

The ON syntax is available in any program, sub-program or UDFB. It is available in both T5 p-code or native code compilation modes.

This statement is an extension to the standard and is not IEC61131-3 compliant.

Warning

This instruction **should not be used inside UDFBs**. This instruction is not UDFB safe.

2.1.17.3 ST Language

```
(* This example counts the rising edges of variable bIN *)
ON bIN DO
   diCount := diCount + 1;
END_DO;
```

2.1.18 ( )

Operator - force the evaluation order in a complex expression.

2.1.18.1 Remarks

Parentheses are used in ST and IL language for changing the default evaluation order of various operations within a complex expression. For instance, the default evaluation of "2 * 3 + 4" expression in ST language gives a result of 10 as "*" operator has highest priority. Changing the expression as "2 * (3 + 4)" gives a result of 14. Parentheses can be nested in a complex expression.

Below is the default evaluation order for ST language operations (1rst is highest priority):

Unary operators	- NOT
Multiply/Divide	* /
Add/Subtract	+ -
Comparisons	< > <= >= = <>
Boolean And	& AND
Boolean Or	OR
Exclusive OR	XOR

In IL language, the default order is the sequence of instructions. Each new instruction modifies the current result sequentially. In IL language, the opening parenthesis "(" is written between the instruction and its operand. The closing parenthesis ")" must be written alone as an instruction without operand.

2.1.18.2 ST Language

```
Q := (IN1 + (IN2 / IN 3)) * IN4;
```

2.1.18.3 FBD Language

Not available

2.1.18.4 FFLD Language

Not available

2.1.18.5 IL Language

```
Op1: FFLD(  IN1
     ADD( IN2
     MUL  IN3
     )
     SUB  IN4
     )
     ST   Q     (* Q is: (IN1 + (IN2 * IN3) -  IN4) *)
```

See also

Assignment

2.1.19 REPEAT UNTIL END_REPEAT

Statement - Repeat a list of statements.

2.1.19.1 Syntax

```
REPEAT
  <statements>
UNTIL <BOOL expression> END_REPEAT;
```

2.1.19.2 Remarks

The statements between "REPEAT" and "UNTIL" are executed until the boolean expression is TRUE. The condition is evaluated ***after*** the statements are executed. Statements are executed at least once.

Warning

Loop instructions can lead to infinite loops that block the target cycle. Never test the state of an input in the condition as the input will not be refreshed before the next cycle.

2.1.19.3 ST Language

```
iPos := 0;
REPEAT
   MyArray[iPos] := 0;
   iNbCleared := iNbCleared + 1;
   iPos := iPos + 1;
UNTIL iPos = iMax END_REPEAT;
```

2.1.19.4 FBD Language

Not available

2.1.19.5 FFLD Language

Not available

2.1.19.6 IL Language

Not available

See also

IF WHILE FOR CASE EXIT

2.1.20 RETURN RET RETC RETNC RETCN

Statement - Jump to the end of the program.

2.1.20.1 Remarks

The "RETURN" statement jumps to the end of the program. In FBD language, the return statement is represented by the "<RETURN>" symbol. The input of the symbol must be connected to a valid boolean signal. The jump is performed only if the input is TRUE. In FFLD language, the "<RETURN>" symbol is used as a coil at the end of a rung. The jump is performed only if the rung state is TRUE. In IL language, RET, RETC, RETCN and RETNC instructions are used.

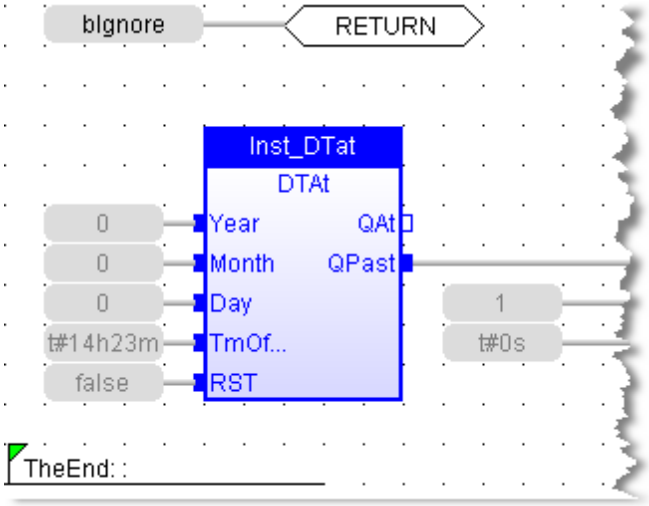
When used within an action block of an SFC step, the RETURN statement jumps to the end of the action block.

2.1.20.2 ST Language

```
IF NOT bEnable THEN
   RETURN;
END_IF;
(* the rest of the program will not be executed if bEnable is FALSE
*)
```

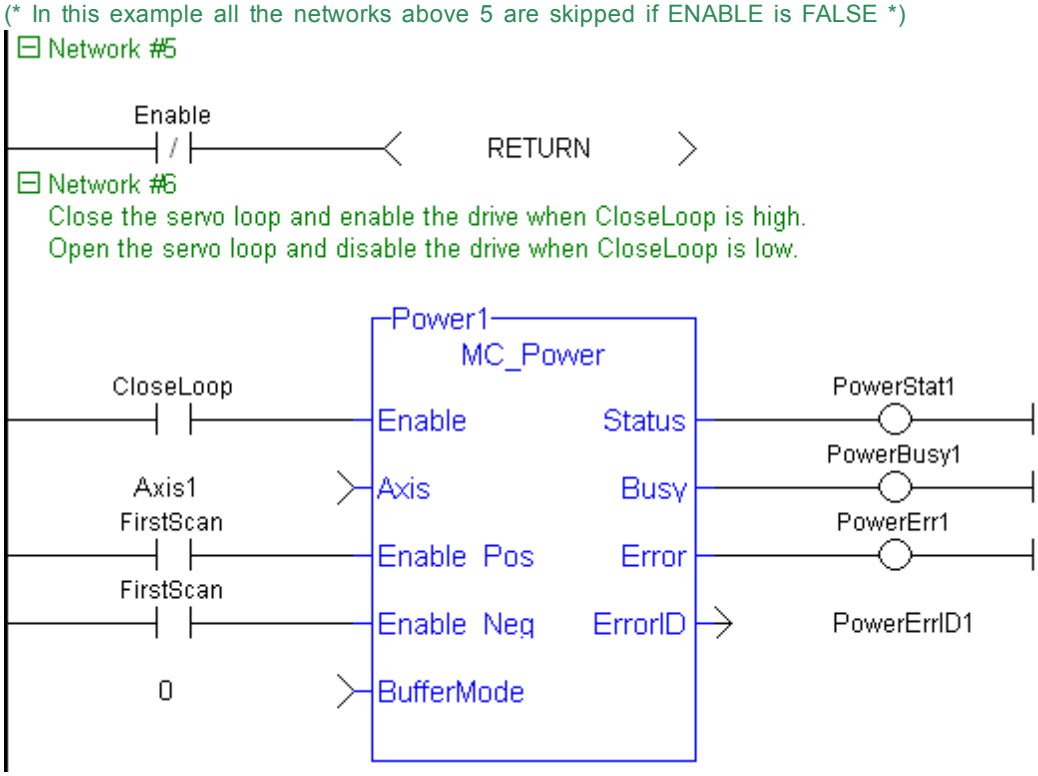
2.1.20.3 FBD Language

(* In this example the DTat block will not be called if bIgnore is TRUE *)



2.1.20.4 FFLD Language

(* In this example all the networks above 5 are skipped if ENABLE is FALSE *)



2.1.20.5 IL Language

Below is the meaning of possible instructions:

`RET` Jump to the end always
`RETC` Jump to the end if the current result is TRUE
`RETNC` Jump to the end if the current result is FALSE
`RETCN` Same as `RETNC`

```
Start: FFLD   IN1
       RETC          (* Jump to the end if IN1 is TRUE *)

       FFLD   IN2    (* these instructions are not executed *)
       ST     Q2      (* if IN1 is TRUE *)
       RET           (* Jump to the end unconditionally *)

       FFLD   IN3    (* these instructions are never executed *)
       ST     Q3
```

See also

Labels Jumps

2.1.21 WHILE DO END_WHILE

Statement - Repeat a list of statements.

2.1.21.1 Syntax

```
WHILE <BOOL expression> DO
  <statements>
END_WHILE ;
```

2.1.21.2 Remarks

The statements between "DO" and "END_WHILE" are executed while the boolean expression is TRUE. The condition is evaluated ***before*** the statements are executed. If the condition is FALSE when WHILE is first reached, statements are never executed.

Warning

Loop instructions can lead to infinite loops that block the target cycle. Never test the state of an input in the condition as the input will not be refreshed before the next cycle.

2.1.21.3 ST Language

```
iPos := 0;
WHILE iPos < iMax DO
   MyArray[iPos] := 0;
   iNbCleared := iNbCleared + 1;
END_WHILE;
```

2.1.21.4 FBD Language

Not available

2.1.21.5 FFLD Language

Not available

2.1.21.6 IL Language

Not available

See also

IF REPEAT FOR CASE EXIT

2.2 Boolean operations

Below are the standard operators for managing booleans:

AND	performs a boolean AND
OR	performs a boolean OR
XOR	performs an exclusive OR
NOT	performs a boolean negation of its input
QOR	qualified OR
S	force a boolean output to TRUE
R	force a boolean output to FALSE

Below are the available blocks for managing boolean signals:

RS	reset dominant bistable
SR	set dominant bistable
R_TRIG	rising pulse detection
F_TRIG	falling pulse detection
SEMA	semaphore
FLIPFLOP	flipflop^bistable

2.2.1 AND ANDN &

Operator - Performs a logical AND of all inputs.

2.2.1.1 Inputs

`IN1 : BOOL` First boolean input
`IN2 : BOOL` Second boolean input

2.2.1.2 Outputs

`Q : BOOL` Boolean AND of all inputs

2.2.1.3 Truth table

IN1	IN2	Q
0	0	0
0	1	0
1	0	0
1	1	1

2.2.1.4 Remarks

In FBD and FFLD languages, the block is called "&" and can have up to 16 inputs. To select the number, right-click on the block and choose the **Set number of inputs** command in the contextual menu.

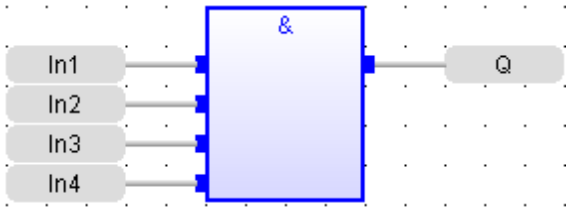
In IL language, the AND instruction performs a logical AND between the current result and the operand. The current result must be boolean. The ANDN instruction performs an AND between the current result and the boolean negation of the operand. In ST and IL languages, "&" can be used instead of "AND".

2.2.1.5 ST Language

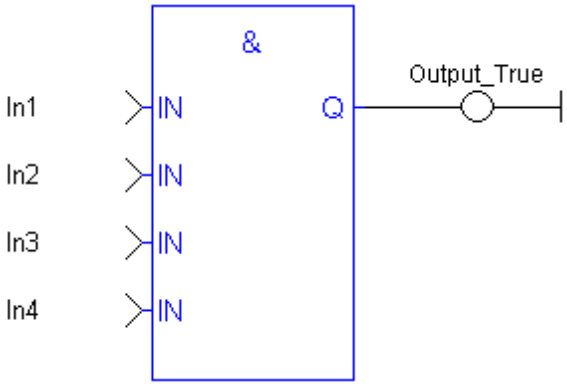
```
Q := IN1 AND IN2;
Q := IN1 & IN2 & IN3;
```

2.2.1.6 FBD Language

```
(* the block can have up to 16 inputs *)
```



2.2.1.7 FFLD Language



2.2.1.8 IL Language:

```
Op1: FFLD  IN1
   &  IN2  (* "&" or "AND" can be used *)
   ST  Q   (* Q is equal to: IN1 AND IN2 *)
Op2: FFLD  IN1
   AND IN2
   &N  IN3  (* "&N" or "ANDN" can be used *)
   ST  Q   (* Q is equal to: IN1 AND IN2 AND (NOT IN3) *)
```

See also

OR XOR NOT

2.2.2 FLIPFLOP

Function Block - Flipflop bistable.

2.2.2.1 Inputs

`IN` : `BOOL` Swap command (on rising edge)
`RST` : `BOOL` Reset to FALSE

2.2.2.2 Outputs

`Q` : `BOOL` Output

2.2.2.3 Remarks

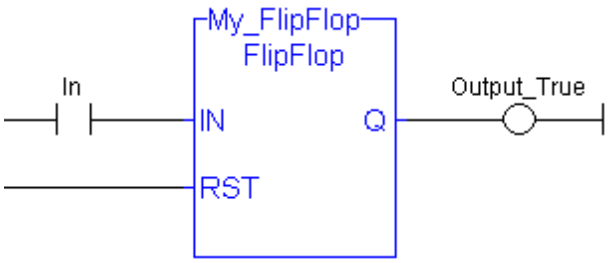
The output is systematically reset to FALSE if RST is TRUE.
The output changes on each rising edge of the IN input, if RST is FALSE.

2.2.2.4 ST Language

```
(* MyFlipFlop is declared as an instance of FLIPFLOP function block *)
MyFlipFlop (IN, RST);
Q := MyFlipFlop.Q;
```

2.2.2.5 FBD Language

2.2.2.6 FFLD Language



2.2.2.7 IL Language

```
(* MyFlipFlop is declared as an instance of FLIPFLOP function block *)
Op1: CAL MyFlipFlop (IN, RST)
    FFLD  MyFlipFlop.Q
    ST  Q1
```

See also

R S SR

2.2.3 F_TRIG

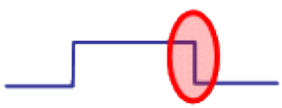
Function Block - Falling pulse detection

2.2.3.1 Inputs

`CLK : BOOL` Boolean signal

2.2.3.2 Outputs

`Q : BOOL` TRUE when the input changes from TRUE to FALSE



2.2.3.3 Truth table

CLK	CLK *prev*	Q
0	0	0
0	1	1
1	0	0
1	1	0

2.2.3.4 Remarks

Although]P[and]N[contacts can be used in FFLD language, it is recommended to use declared instances of R_TRIG or F_TRIG function blocks in order to avoid contingencies during an Online Change.

2.2.3.5 ST Language

```
(* MyTrigger is declared as an instance of F_TRIG function block *)
MyTrigger (CLK);
Q := MyTrigger.Q;
```

2.2.3.6 FBD Language

2.2.3.7 FFLD Language

2.2.3.8 IL Language:

```
(* MyTrigger is declared as an instance of F_TRIG function block *)
Op1: CAL MyTrigger (CLK)
FFLD  MyTrigger.Q
ST  Q
```

See also

R_TRIG

2.2.4 NOT

Operator - Performs a boolean negation of the input.

2.2.4.1 Inputs

`IN : BOOL` Boolean value

2.2.4.2 Outputs

`Q : BOOL` Boolean negation of the input

2.2.4.3 Truth table

IN	Q
0	1
1	0

2.2.4.4 Remarks

In FBD language, the block "NOT" can be used. Alternatively, you can use a link terminated by a "o" negation. In FFLD language, negated contacts and coils can be used. In IL language, the "N" modifier can be used with instructions FFLD, AND, OR, XOR and ST. It represents a negation of the operand. In ST language, NOT can be followed by a complex boolean expression between parentheses.

2.2.4.5 ST Language

```
Q := NOT IN;
Q := NOT (IN1 OR IN2);
```

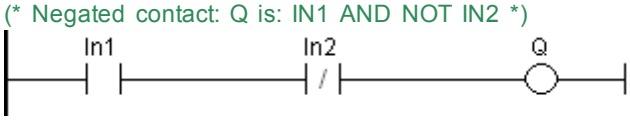
2.2.4.6 FBD Language

```
(* explicit use of the "NOT" block *)
```

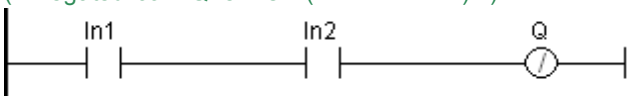

```
(* use of a negated link: Q is IN1 AND NOT IN2 *)
```

2.2.4.7 FFLD Language

```
(* Negated contact: Q is: IN1 AND NOT IN2 *)
```




```
(* Negated coil: Q is NOT (IN1 AND IN2) *)
```



2.2.4.8 IL Language:

```
Op1: FFLDN IN1
    OR  IN2
    ST  Q   (* Q is equal to: (NOT IN1) OR IN2 *)
Op2: FFLD  IN1
    AND IN2
    STN Q   (* Q is equal to: NOT (IN1 AND IN2) *)
```

See also

AND OR XOR

2.2.5 OR ORN

Operator - Performs a logical OR of all inputs.

2.2.5.1 Inputs

IN1 : BOOL First boolean input
IN2 : BOOL Second boolean input

2.2.5.2 Outputs

Q : BOOL Boolean OR of all inputs

2.2.5.3 Truth table

IN1	IN2	Q
0	0	0
0	1	1
1	0	1
1	1	1

2.2.5.4 Remarks

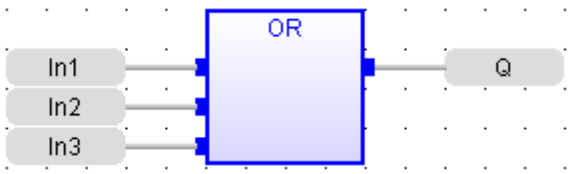
In FBD language, the block can have up to 16 inputs. The block is called ">=1" in FBD language. In FFLD language, an OR operation is represented by contacts in parallel. In IL language, the OR instruction performs a logical OR between the current result and the operand. The current result must be boolean. The ORN instruction performs an OR between the current result and the boolean negation of the operand.

2.2.5.5 ST Language

```
Q := IN1 OR IN2;
Q := IN1 OR IN2 OR IN3;
```

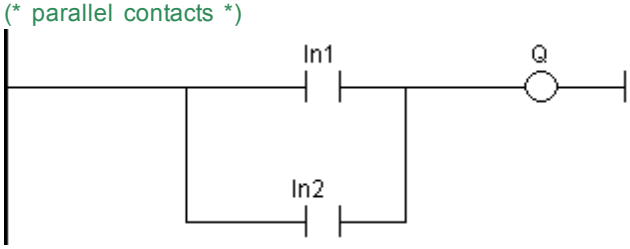
2.2.5.6 FBD Language

```
(* the block can have up to 16 inputs *)
```



2.2.5.7 FFLD Language

```
(* parallel contacts *)
```



2.2.5.8 IL Language

```
Op1: FFLD  IN1
     OR  IN2
     ST  Q   (* Q is equal to: IN1 OR IN2 *)
Op2: FFLD  IN1
     ORN IN2
     ST Q    (* Q is equal to: IN1 OR (NOT IN2) *)
```

See also

AND XOR NOT

2.2.6 R

Operator - Force a boolean output to FALSE.

2.2.6.1 Inputs

`RESET : BOOL` Condition

2.2.6.2 Outputs

`Q : BOOL` Output to be forced

2.2.6.3 Truth table

RESET	Q *prev*	Q
0	0	0
0	1	1
1	0	0
1	1	0

2.2.6.4 Remarks

S and R operators are available as standard instructions in the IL language. In FFLD languages they are represented by (S) and (R) coils. In FBD language, you can use (S) and (R) coils, but you must prefer RS and SR function blocks. Set and reset operations are not available in ST language.

2.2.6.5 ST Language

Not available.

2.2.6.6 FBD Language

Not available. Use RS or SR function blocks.

2.2.6.7 FFLD Language

```
(* use of "R" coil *)
```

2.2.6.8 IL Language:

```
Op1: FFLD  RESET
   R  Q   (* Q is forced to FALSE if RESET is TRUE *)
        (* Q is unchanged if RESET is FALSE *)
```

See also

S RS SR

2.2.7 RS

Function Block - Reset dominant bistable.

2.2.7.1 Inputs

`SET : BOOL` Condition for forcing to TRUE
`RESET1 : BOOL` Condition for forcing to FALSE (highest priority command)

2.2.7.2 Outputs

`Q1` : `BOOL` Output to be forced

2.2.7.3 Truth table

SET	RESET1	Q1 *prev*	Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

2.2.7.4 Remarks

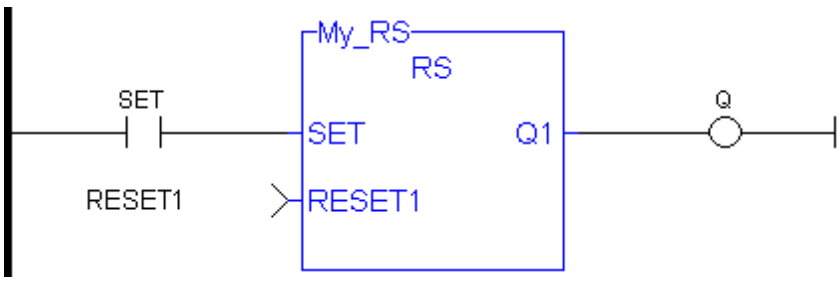
The output is unchanged when both inputs are FALSE. When both inputs are TRUE, the output is forced to FALSE (reset dominant).

2.2.7.5 ST Language

```
(* MyRS is declared as an instance of RS function block *)
MyRS (SET, RESET1);
Q1 := MyRS.Q1;
```

2.2.7.6 FBD Language

2.2.7.7 FFLD Language



2.2.7.8 IL Language:

```
(* MyRS is declared as an instance of RS function block *)
Op1: CAL MyRS (SET, RESET1)
    FFLD  MyRS.Q1
    ST  Q1
```

See also

R S SR

2.2.8 R_TRIG

Function Block - Rising pulse detection

2.2.8.1 Inputs

`CLK : BOOL` Boolean signal

2.2.8.2 Outputs

`Q : BOOL` TRUE when the input changes from FALSE to TRUE

2.2.8.3 Truth table

CLK	CLK *prev*	Q
0	0	0
0	1	0
1	0	1
1	1	0

2.2.8.4 Remarks

Although]P[and]N[contacts can be used in FFLD language, it is recommended to use declared instances of R_TRIG or F_TRIG function blocks in order to avoid contingencies during an Online Change.

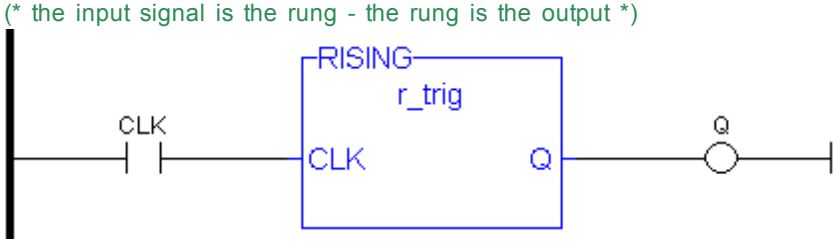
2.2.8.5 ST Language

```
(* MyTrigger is declared as an instance of R_TRIG function block *)
MyTrigger (CLK);
Q := MyTrigger.Q;
```

2.2.8.6 FBD Language

2.2.8.7 FFLD Language

```
(* the input signal is the rung - the rung is the output *)
```



2.2.8.8 IL Language:

```
(* MyTrigger is declared as an instance of R_TRIG function block *)
Op1: CAL MyTrigger (CLK)
FFLD MyTrigger.Q
ST Q
```

See also

F_TRIG

2.2.9 S

Operator - Force a boolean output to TRUE.

2.2.9.1 Inputs

`SET : BOOL` Condition

2.2.9.2 Outputs

`Q : BOOL` Output to be forced

2.2.9.3 Truth table

SET	Q *prev*	Q
0	0	0
0	1	1
1	0	1
1	1	1

2.2.9.4 Remarks

S and R operators are available as standard instructions in the IL language. In FFLD languages they are represented by (S) and (R) coils In FBD language, you can use (S) and (R) coils, but you must prefer RS and SR function blocks. Set and reset operations are not available in ST language.

2.2.9.5 ST Language

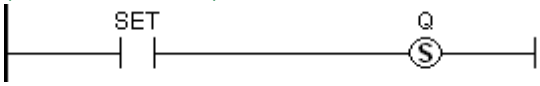
Not available.

2.2.9.6 FBD Language

Not available. Use RS or SR function blocks.

2.2.9.7 FFLD Language

```
(* use of "S" coil *)
```



2.2.9.8 IL Language:

```
Op1: FFLD  SET
   S  Q   (* Q is forced to TRUE if SET is TRUE *)
        (* Q is unchanged if SET is FALSE *)
```

See also

R RS SR

2.2.10 SEMA

Function Block - Semaphore.

2.2.10.1 Inputs

`CLAIM : BOOL` Takes the semaphore
`RELEASE : BOOL` Releases the semaphore

2.2.10.2 Outputs

`BUSY : BOOL` True if semaphore is busy

2.2.10.3 Remarks

The function block implements the following algorithm:

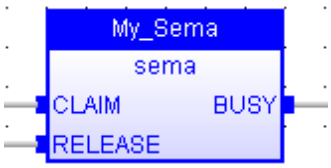
```
BUSY := mem;
if CLAIM then
   mem := TRUE;
else if RELEASE then
   BUSY := FALSE;
   mem := FALSE;
end_if;
```

In FFLD language, the input rung is the CLAIM command. The output rung is the BUSY output signal.

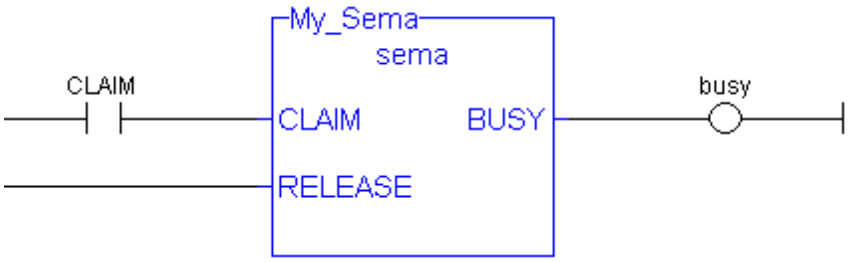
2.2.10.4 ST Language

```
(* MySema is a declared instance of SEMA function block *)
MySema (CLAIM, RELEASE);
BUSY := MyBlinker.BUSY;
```

2.2.10.5 FBD Language



2.2.10.6 FFLD Language



2.2.10.7 IL Language:

```
(* MySema is a declared instance of SEMA function block *)
Op1: CAL MySema (CLAIM, RELEASE)
   FFLD MyBlinker.BUSY
   ST BUSY
```

2.2.11 SR

Function Block - Set dominant bistable.

2.2.11.1 Inputs

`SET1 : BOOL` Condition for forcing to TRUE (highest priority command)
`RESET : BOOL` Condition for forcing to FALSE

2.2.11.2 Outputs

`Q1` : `BOOL` Output to be forced

2.2.11.3 Truth table

SET1	RESET	Q1 *prev*	Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

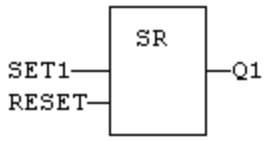
2.2.11.4 Remarks

The output is unchanged when both inputs are FALSE. When both inputs are TRUE, the output is forced to TRUE (set dominant).

2.2.11.5 ST Language

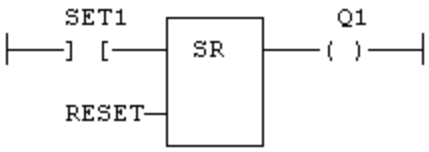
```
(* MySR is declared as an instance of SR function block *)
MySR (SET1, RESET);
Q1 := MySR.Q1;
```

2.2.11.6 FBD Language



2.2.11.7 FFLD Language

```
(* the SET1 command is the rung - the rung is the output *)
```



2.2.11.8 IL Language:

```
(* MySR is declared as an instance of SR function block *)
Op1: CAL MySR (SET1, RESET)
    FFLD  MySR.Q1
    ST  Q1
```

See also

R S RS

2.2.12 XOR XORN

Operator - Performs an exclusive OR of all inputs.

2.2.12.1 Inputs

IN1 : BOOL First boolean input
IN2 : BOOL Second boolean input

2.2.12.2 Outputs

Q : BOOL Exclusive OR of all inputs

2.2.12.3 Truth table

IN1	IN2	Q
0	0	0
0	1	1
1	0	1
1	1	0

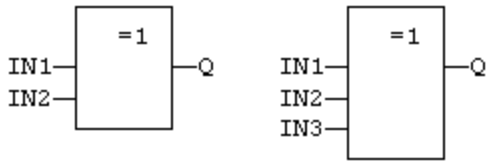
2.2.12.4 Remarks

The block is called "=1" in FBD and FFLD languages. In IL language, the XOR instruction performs an exclusive OR between the current result and the operand. The current result must be boolean. The XORN instruction performs an exclusive between the current result and the boolean negation of the operand.

2.2.12.5 ST Language

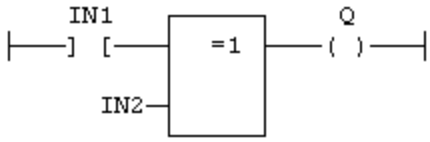
```
Q := IN1 XOR IN2;
Q := IN1 XOR IN2 XOR IN3;
```

2.2.12.6 FBD Language



2.2.12.7 FFLD Language

```
(* First input is the rung. The rung is the output *)
```



2.2.12.8 IL Language

```
Op1: FFLD   IN1
     XOR  IN2
     ST   Q   (* Q is equal to: IN1 XOR IN2 *)
Op2: FFLD   IN1
     XORN IN2
     ST   Q   (* Q is equal to: IN1 XOR (NOT IN2) *)
```

See also

AND OR NOT

2.3 Arithmetic operations

Below are the standard operators that perform arithmetic operations:

+	addition
-	subtraction
*	multiplication
/	division
- (NEG)	integer negation (unary operator)

Below are the standard functions that perform arithmetic operations:

MIN	get the minimum of two integers or an ANY
MAX	get the maximum of two integers or an ANY
LIMIT	bound an integer to low and high limits or an ANY
MOD	modulo
ODD	test if an integer is odd
SetWithin	Force a value when within an interval

2.3.1 + ADD

Operator - Performs an addition of all inputs.

2.3.1.1 Inputs

`IN1 : ANY` First input
`IN2 : ANY` Second input

2.3.1.2 Outputs

`Q : ANY` Result: IN1 + IN2

2.3.1.3 Remarks

All inputs and the output must have the same type. In FBD language, the block can have up to 16 inputs. In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the ADD instruction performs an addition between the current result and the operand. The current result and the operand must have the same type.

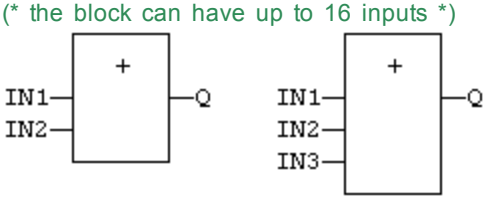
The addition can be used with strings. The result is the concatenation of the input strings.

2.3.1.4 ST Language

```
Q := IN1 + IN2;
MyString := 'He' + 'll ' + 'o';  (* MyString is equal to 'Hello' *)
```

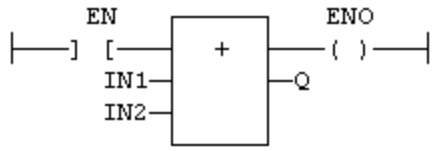
2.3.1.5 FBD Language

```
(* the block can have up to 16 inputs *)
```



2.3.1.6 FFLD Language

(* The addition is executed only if EN is TRUE *)
(* ENO is equal to EN *)



2.3.1.7 IL Language:

```
Op1: FFLD  IN1
     ADD IN2
     ST  Q   (* Q is equal to: IN1 + IN2 *)
Op2: FFLD  IN1
     ADD IN2
     ADD IN3
     ST  Q   (* Q is equal to: IN1 + IN2 + IN3 *)
```

See also

- * /

2.3.2 / DIV

Operator - Performs a division of inputs.

2.3.2.1 Inputs

IN1	:	ANY_NUM	First input
IN2	:	ANY_NUM	Second input

2.3.2.2 Outputs

Q	:	ANY_NUM	Result: IN1 / IN2

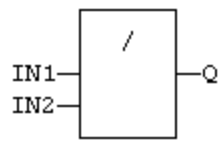
2.3.2.3 Remarks

All inputs and the output must have the same type. In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the DIV instruction performs a division between the current result and the operand. The current result and the operand must have the same type.

2.3.2.4 ST Language

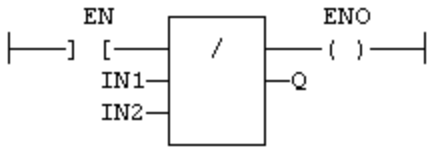
```
Q := IN1 / IN2;
```

2.3.2.5 FBD Language



2.3.2.6 FFLD Language

(* The division is executed only if EN is TRUE *)
(* ENO is equal to EN *)



2.3.2.7 IL Language:

```
Op1: FFLD  IN1
   DIV IN2
   ST  Q   (* Q is equal to: IN1 / IN2 *)
Op2: FFLD  IN1
   DIV IN2
   DIV IN3
   ST  Q   (* Q is equal to: IN1 / IN2 / IN3 *)
```

See also

+ - *

2.3.3 NEG -

Operator - Performs an integer negation of the input.

2.3.3.1 Inputs

`IN : DINT` Integer value

2.3.3.2 Outputs

`Q : DINT` Integer negation of the input

2.3.3.3 Truth table (examples)

IN	Q
0	0
1	-1
-123	123

2.3.3.4 Remarks

In FBD and FFLD language, the block "NEG" can be used.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

This feature is not available in IL language. In ST language, "-" can be followed by a complex boolean expression between parentheses.

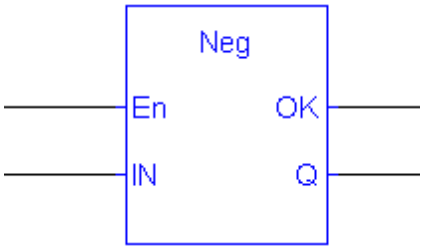
2.3.3.5 ST Language

```
Q := -IN;
Q := - (IN1 + IN2);
```

2.3.3.6 FBD Language

2.3.3.7 FFLD Language

```
(* The negation is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.3.3.8 IL Language

Not available

2.3.4 LIMIT

Function - Bounds an integer between low and high limits.

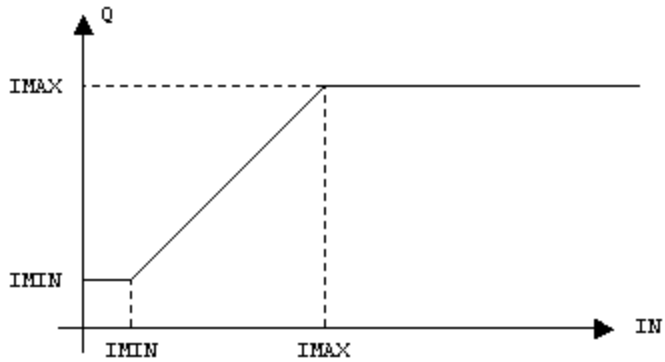
2.3.4.1 Inputs

`IMIN : DINT` Low bound
`IN : DINT` Inputvalue
`IMAX : DINT` High bound

2.3.4.2 Outputs

`Q : DINT` IMIN if IN < IMIN; IMAX if IN > IMAX; IN otherwise

2.3.4.3 Function diagram



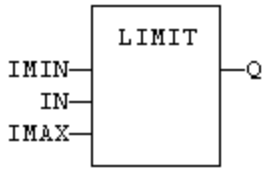
2.3.4.4 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. Other inputs are operands of the function, separated by a coma.

2.3.4.5 ST Language

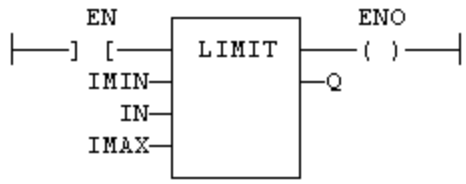
```
Q := LIMIT (IMIN, IN, IMAX);
```

2.3.4.6 FBD Language



2.3.4.7 FFLD Language

(* The comparison is executed only if EN is TRUE *)
(* ENO has the same value as EN *)



2.3.4.8 IL Language:

```
Op1: FFLD   IMIN
   LIMIT IN, IMAX
   ST   Q
```

See also

MIN MAX MOD ODD

2.3.5 MAX

Function - Get the maximum of two integers.

2.3.5.1 Inputs

`IN1 : DINT` First input
`IN2 : DINT` Second input

2.3.5.2 Outputs

`Q : DINT` IN1 if IN1 > IN2; IN2 otherwise

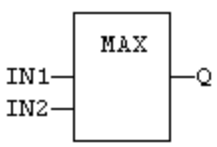
2.3.5.3 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.3.5.4 ST Language

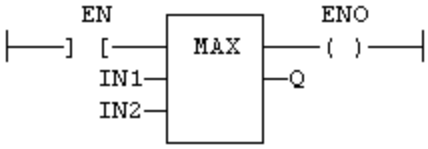
```
Q := MAX (IN1, IN2);
```

2.3.5.5 FBD Language



2.3.5.6 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.3.5.7 IL Language:

```
Op1: FFLD  IN1
     MAX IN2
     ST  Q   (* Q is the maximum of IN1 and IN2 *)
```

See also

MIN LIMIT MOD ODD

2.3.6 MIN

Function - Get the minimum of two integers.

2.3.6.1 Inputs

IN1	:	DINT	First input
IN2	:	DINT	Second input

2.3.6.2 Outputs

Q	:	DINT	IN1 if IN1 < IN2; IN2 otherwise

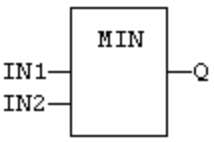
2.3.6.3 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.3.6.4 ST Language

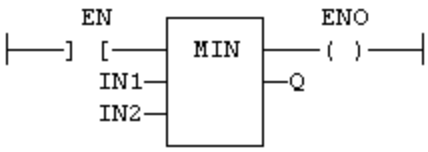
```
Q := MIN (IN1, IN2);
```

2.3.6.5 FBD Language



2.3.6.6 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.3.6.7 IL Language:

```
Op1: FFLD  IN1
     MIN IN2
     ST  Q   (* Q is the minimum of IN1 and IN2 *)
```

See also

MAX LIMIT MOD ODD

2.3.7 MOD / MODR / MODLR

Function - Calculation of modulo.

Inputs	Function MOD	MODR	MODLR	Description
IN	DINT	REAL	LREAL	Input value
BASE	DINT	REAL	LREAL	Base of the modulo

Output	Function MOD	MODR	MODLR	Description
Q	DINT	REAL	LREAL	Modulo: rest of the integer division (IN / BASE)

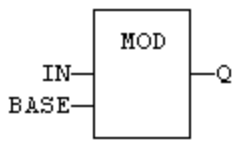
2.3.7.1 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.3.7.2 ST Language

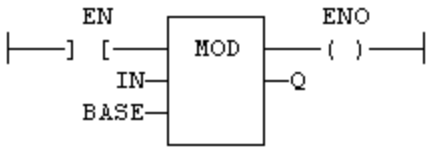
```
Q := MOD (IN, BASE);
```

2.3.7.3 FBD Language



2.3.7.4 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.3.7.5 IL Language:

```
Op1: FFLD  IN
     MOD BASE
     ST  Q   (* Q is the rest of integer division: IN / BASE *)
```

See also

MIN MAX LIMIT ODD

2.3.8 * MUL

Operator - Performs a multiplication of all inputs.

2.3.8.1 Inputs

IN1	:	ANY_NUM	First input
IN2	:	ANY_NUM	Second input

2.3.8.2 Outputs

Q	:	ANY_NUM	Result: IN1 * IN2

2.3.8.3 Remarks

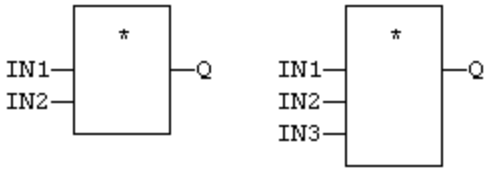
All inputs and the output must have the same type. In FBD language, the block can have up to 16 inputs. In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the MUL instruction performs a multiplication between the current result and the operand. The current result and the operand must have the same type.

2.3.8.4 ST Language

```
Q := IN1 * IN2;
```

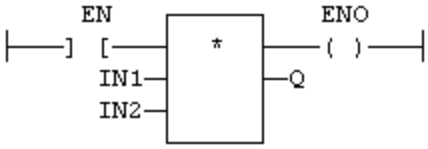
2.3.8.5 FBD Language

```
(* the block can have up to 16 inputs *)
```



2.3.8.6 FFLD Language

```
(* The multiplication is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```



2.3.8.7 IL Language:

```
Op1: FFLD  IN1
     MUL  IN2
     ST  Q   (* Q is equal to: IN1 * IN2 *)
Op2: FFLD  IN1
     MUL  IN2
     MUL  IN3
     ST  Q   (* Q is equal to: IN1 * IN2 * IN3 *)
```

See also

+ - /

2.3.9 ODD

Function - Test if an integer is odd

2.3.9.1 Inputs

`IN : DINT` Input value

2.3.9.2 Outputs

`Q : BOOL` TRUE if IN is odd. FALSE if IN is even.

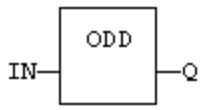
2.3.9.3 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung is the result of the function. In IL language, the input must be loaded before the function call.

2.3.9.4 ST Language

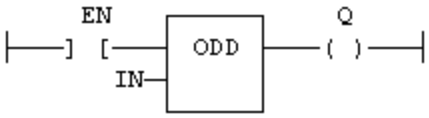
```
Q := ODD (IN);
```

2.3.9.5 FBD Language



2.3.9.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
```



2.3.9.7 IL Language:

```
Op1: FFLD  IN
     ODD
     ST  Q   (* Q is TRUE if IN is odd *)
```

See also

MIN MAX LIMIT MOD

2.3.10 - SUB

Operator - Performs a subtraction of inputs.

2.3.10.1 Inputs

`IN1 : ANY_NUM / TIME` First input
`IN2 : ANY_NUM / TIME` Second input

2.3.10.2 Outputs

`Q : ANY_NUM / TIME` Result: IN1 - IN2

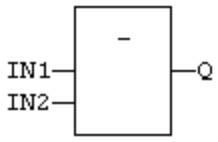
2.3.10.3 Remarks

All inputs and the output must have the same type. In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the SUB instruction performs a subtraction between the current result and the operand. The current result and the operand must have the same type.

2.3.10.4 ST Language

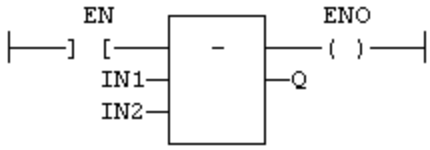
```
Q := IN1 - IN2;
```

2.3.10.5 FBD Language



2.3.10.6 FFLD Language

```
(* The subtraction is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```



2.3.10.7 IL Language:

```
Op1: FFLD  IN1
     SUB IN2
     ST  Q   (* Q is equal to: IN1 - IN2 *)
Op2: FFLD  IN1
     SUB IN2
     SUB IN3
     ST  Q   (* Q is equal to: IN1 - IN2 - IN3 *)
```

See also

+ * /

2.4 Comparison operations

Below are the standard operators and blocks that perform comparisons:

<	less than
>	greater than
<=	less or equal
>=	greater or equal
=	is equal
<>	is not equal
CMP	detailed comparison

2.4.1 CMP

Function Block - Comparison with detailed outputs for integer inputs

2.4.1.1 Inputs

```
IN1 : DINT   First value
IN2 : DINT   Second value
```

2.4.1.2 Outputs

```
LT : BOOL   TRUE if IN1 < IN2
EQ : BOOL   TRUE if IN1 = IN2
GT : BOOL   TRUE if IN1 > IN2
```

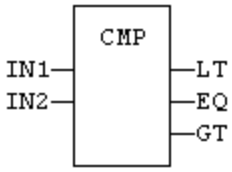
2.4.1.3 Remarks

In FFLD language, the rung input (EN) validates the operation. The rung output is the result of "LT" (lower than) comparison).

2.4.1.4 ST Language

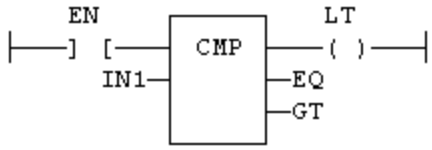
```
(* MyCmp is declared as an instance of CMP function block *)
MyCMP (IN1, IN2);
bLT := MyCmp.LT;
bEQ := MyCmp.EQ;
bGT := MyCmp.GT;
```

2.4.1.5 FBD Language



2.4.1.6 FFLD Language

```
(* the comparison is performed only if EN is TRUE  *)
```



2.4.1.7 IL Language:

```
(* MyCmp is declared as an instance of CMP function block *)
Op1: CAL MyCmp (IN1, IN2)
   FFLD  MyCmp.LT
   ST  bLT
   FFLD  MyCmp.EQ
   ST  bEQ
   FFLD  MyCmp.GT
   ST  bGT
```

See also

> < >= <= = <>

2.4.2 >= GE

Operator - Test if first input is greater than or equal to second input.

2.4.2.1 Inputs

```
IN1 : ANY   First input
IN2 : ANY   Second input
```

2.4.2.2 Outputs

`Q : BOOL` TRUE if IN1 >= IN2

2.4.2.3 Remarks

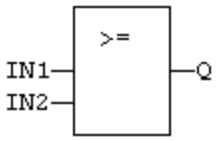
Both inputs must have the same type. In FFLD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the GE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".

2.4.2.4 ST Language

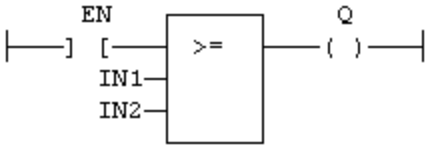
```
Q := IN1 >= IN2;
```

2.4.2.5 FBD Language



2.4.2.6 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
```



2.4.2.7 IL Language:

```
Op1: FFLD  IN1
     GE  IN2
     ST  Q   (* Q is true if IN1 >= IN2 *)
```

See also

> < <= = <> CMP

2.4.3 > GT

Operator - Test if first input is greater than second input.

2.4.3.1 Inputs

`IN1 : ANY` First input
`IN2 : ANY` Second input

2.4.3.2 Outputs

`Q : BOOL` TRUE if IN1 > IN2

2.4.3.3 Remarks

Both inputs must have the same type. In FFLD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the

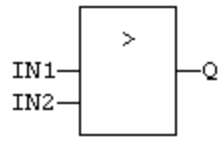
GT instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".

2.4.3.4 ST Language

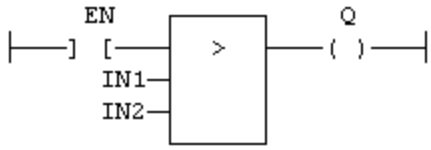
```
Q := IN1 > IN2;
```

2.4.3.5 FBD Language



2.4.3.6 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
```



2.4.3.7 IL Language:

```
Op1: FFLD  IN1
     GT  IN2
     ST  Q   (* Q is true if IN1 > IN2 *)
```

See also

< >= <= = <> CMP

2.4.4 = EQ

Operator - Test if first input is equal to second input.

2.4.4.1 Inputs

IN1 : ANY		First input
IN2 : ANY		Second input

2.4.4.2 Outputs

Q : BOOL	TRUE if IN1 = IN2

2.4.4.3 Remarks

Both inputs must have the same type. In FFLD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the EQ instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

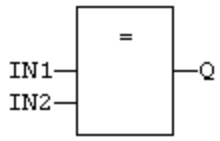
Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".

Equality comparisons cannot be used with TIME variables. The reason is that the timer actually has the resolution of the target cycle and test can be unsafe as some values can never be reached.

2.4.4.4 ST Language

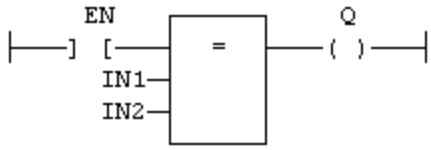
```
Q := IN1 = IN2;
```

2.4.4.5 FBD Language



2.4.4.6 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
```



2.4.4.7 IL Language:

```
Op1: FFLD  IN1
     EQ  IN2
     ST  Q   (* Q is true if IN1 = IN2 *)
```

See also

> < >= <= <> CMP

2.4.5 <> NE

Operator - Test if first input is not equal to second input.

2.4.5.1 Inputs

IN1 : ANY	First input
IN2 : ANY	Second input

2.4.5.2 Outputs

Q : BOOL	TRUE if IN1 is not equal to IN2

2.4.5.3 Remarks

Both inputs must have the same type. In FFLD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the NE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

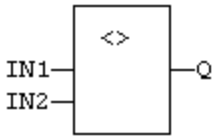
Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".

Equality comparisons cannot be used with TIME variables. The reason is that the timer actually has the resolution of the target cycle and test can be unsafe as some values can never be reached

2.4.5.4 ST Language

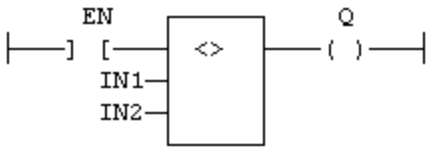
```
Q := IN1 <> IN2;
```

2.4.5.5 FBD Language



2.4.5.6 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
```



2.4.5.7 IL Language:

```
Op1: FFLD  IN1
     NE  IN2
     ST  Q   (* Q is true if IN1 is not equal to IN2 *)
```

See also

> < >= <= = CMP

2.4.6 <= LE

Operator - Test if first input is less than or equal to second input.

2.4.6.1 Inputs

IN1 : ANY		First input
IN2 : ANY		Second input

2.4.6.2 Outputs

Q : BOOL	TRUE if IN1 <= IN2

2.4.6.3 Remarks

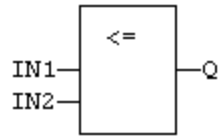
Both inputs must have the same type. In FFLD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the LE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".

2.4.6.4 ST Language

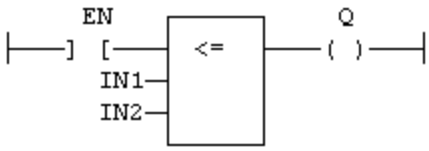
```
Q := IN1 <= IN2;
```

2.4.6.5 FBD Language



2.4.6.6 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
```



2.4.6.7 IL Language:

```
Op1: FFLD  IN1
     LE  IN2
     ST  Q   (* Q is true if IN1 <= IN2 *)
```

See also

> < >= = <> CMP

2.4.7 < LT

Operator - Test if first input is less than second input.

2.4.7.1 Inputs

IN1 : ANY		First input
IN2 : ANY		Second input

2.4.7.2 Outputs

Q : BOOL	TRUE if IN1 < IN2

2.4.7.3 Remarks

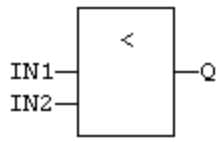
Both inputs must have the same type. In FFLD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the LT instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".

2.4.7.4 ST Language

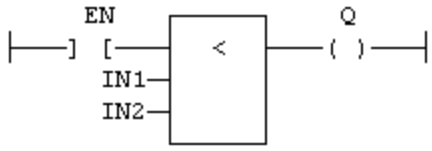
```
Q := IN1 < IN2;
```

2.4.7.5 FBD Language



2.4.7.6 FFLD Language

```
(* The comparison is executed only if EN is TRUE *)
```



2.4.7.7 IL Language:

```
Op1: FFLD  IN1
   LT  IN2
   ST  Q   (* Q is true if IN1 < IN2 *)
```

See also

> >= <= = <> CMP

2.5 Type conversion functions

Below are the standard functions for converting a data into another data type:

ANY_TO_BOOL	converts to boolean
ANY_TO_SINT / ANY_TO_USINT	converts to small (8 bit) integer
ANY_TO_INT / ANY_TO_UINT	converts to 16 bit integer
ANY_TO_DINT / ANY_TO_UDINT	converts to integer (32 bit - default)
ANY_TO_LINT / ANY_TO_ULINT	converts to long (64 bit) integer
ANY_TO_REAL	converts to real
ANY_TO_LREAL	converts to double precision real
ANY_TO_TIME	converts to time
ANY_TO_STRING	converts to character string

Below are the standard functions performing conversions in BCD format (*):

BIN_TO_BCD	converts a binary value to a BCD value
BCD_TO_BIN	converts a BCD value to a binary value

(*) BCD conversion functions may not be supported by all targets.

2.5.1 ANY_TO_BOOL

Operator - Converts the input into boolean value.

2.5.1.1 Inputs

`IN : ANY` Input value

2.5.1.2 Outputs

`Q : BOOL` Value converted to boolean

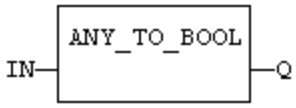
2.5.1.3 Remarks

For DINT, REAL and TIME input data types, the result is FALSE if the input is 0. The result is TRUE in all other cases. For STRING inputs, the output is TRUE if the input string is not empty, and FALSE if the string is empty. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung is the result of the conversion. In IL Language, the ANY_TO_BOOL function converts the current result.

2.5.1.4 ST Language

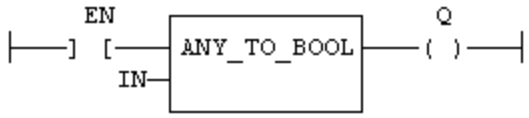
```
Q := ANY_TO_BOOL (IN);
```

2.5.1.5 FBD Language



2.5.1.6 FFLD Language

```
(* The conversion is executed only if EN is TRUE *)
(* The output rung is the result of the conversion *)
(* The output rung is FALSE if the EN is FALSE *)
```



2.5.1.7 IL Language:

```
Op1: FFLD  IN
   ANY_TO_BOOL
   ST  Q
```

2.5.1.8 See also

ANY_TO_SINT ANY_TO_INT ANY_TO_DINT ANY_TO_LINT ANY_TO_REAL ANY_TO_LREAL ANY_TO_TIME ANY_TO_STRING

2.5.2 ANY_TO_DINT / ANY_TO_UDINT

Operator - Converts the input into integer value (can be unsigned with ANY_TO_UDINT).

2.5.2.1 Inputs

`IN : ANY` Input value

2.5.2.2 Outputs

`Q : DINT` Value converted to integer

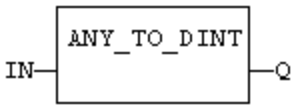
2.5.2.3 Remarks

For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_DINT function converts the current result.

2.5.2.4 ST Language

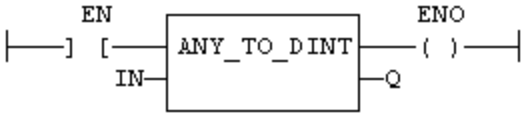
```
Q := ANY_TO_DINT (IN);
```

2.5.2.5 FBD Language



2.5.2.6 FFLD Language

```
(* The conversion is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.5.2.7 IL Language:

```
Op1: FFLD  IN
     ANY_TO_DINT
     ST  Q
```

2.5.2.8 See also

ANY_TO_BOOL ANY_TO_SINT ANY_TO_INT ANY_TO_LINT ANY_TO_REAL ANY_TO_LREAL ANY_TO_TIME ANY_TO_STRING

2.5.3 ANY_TO_INT / ANY_TO_UINT

Operator - Converts the input into 16 bit integer value (can be unsigned with ANY_TO_UINT).

2.5.3.1 Inputs

`IN : ANY` Input value

2.5.3.2 Outputs

`Q : INT` Value converted to 16 bit integer

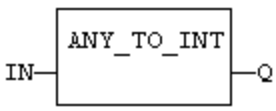
2.5.3.3 Remarks

For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_INT function converts the current result.

2.5.3.4 ST Language

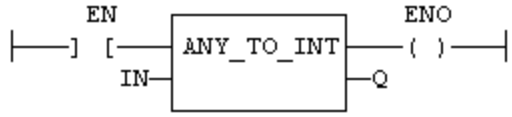
```
Q := ANY_TO_INT (IN);
```

2.5.3.5 FBD Language



2.5.3.6 FFLD Language

(* The conversion is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.5.3.7 IL Language:

```
Op1: FFLD  IN
   ANY_TO_INT
   ST  Q
```

2.5.3.8 See also

ANY_TO_BOOL ANY_TO_SINT ANY_TO_DINT ANY_TO_LINT ANY_TO_REAL ANY_TO_LREAL ANY_TO_TIME ANY_TO_STRING

2.5.4 ANY_TO_LINT / ANY_TO_ULINT

Operator - Converts the input into long (64 bit) integer value (can be unsigned with ANY_TO_ULINT).

2.5.4.1 Inputs

`IN : ANY` Input value

2.5.4.2 Outputs

`Q : LINT` Value converted to long (64 bit) integer

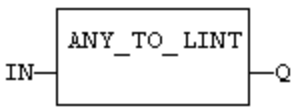
2.5.4.3 Remarks

For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_LINT function converts the current result.

2.5.4.4 ST Language

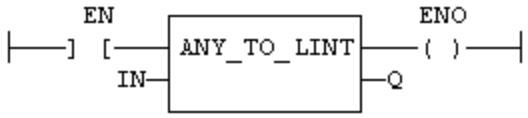
```
Q := ANY_TO_LINT (IN);
```

2.5.4.5 FBD Language



2.5.4.6 FFLD Language

(* The conversion is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.5.4.7 IL Language:

```
Op1: FFLD  IN
   ANY_TO_LINT
   ST  Q
```

2.5.4.8 See also

ANY_TO_BOOL ANY_TO_SINT ANY_TO_INT ANY_TO_DINT ANY_TO_REAL ANY_TO_LREAL ANY_TO_TIME ANY_TO_STRING

2.5.5 ANY_TO_LREAL

Operator - Converts the input into double precision real value.

2.5.5.1 Inputs

`IN : ANY` Input value

2.5.5.2 Outputs

`Q : LREAL` Value converted to double precision real

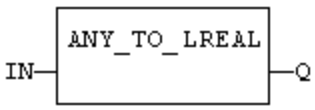
2.5.5.3 Remarks

For BOOL input data types, the output is 0.0 or 1.0. For DINT input data type, the output is the same number. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_LREAL function converts the current result.

2.5.5.4 ST Language

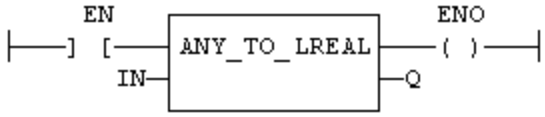
```
Q := ANY_TO_LREAL (IN);
```

2.5.5.5 FBD Language



2.5.5.6 FFLD Language

```
(* The conversion is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.5.5.7 IL Language:

```
Op1: FFLD  IN
   ANY_TO_LREAL
```

```
   ST  Q
```

2.5.5.8 See also

ANY_TO_BOOL ANY_TO_SINT ANY_TO_INT ANY_TO_DINT ANY_TO_LINT ANY_TO_REAL ANY_TO_TIME ANY_TO_STRING

2.5.6 ANY_TO_REAL

Operator - Converts the input into real value.

2.5.6.1 Inputs

`IN : ANY` Input value

2.5.6.2 Outputs

`Q : REAL` Value converted to real

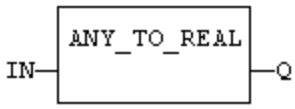
2.5.6.3 Remarks

For BOOL input data types, the output is 0.0 or 1.0. For DINT input data type, the output is the same number. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_REAL function converts the current result.

2.5.6.4 ST Language

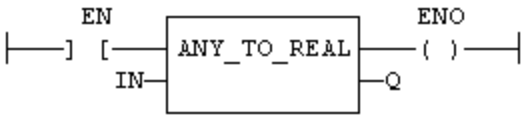
```
Q := ANY_TO_REAL (IN);
```

2.5.6.5 FBD Language



2.5.6.6 FFLD Language

```
(* The conversion is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.5.6.7 IL Language:

```
Op1: FFLD  IN
   ANY_TO_REAL
   ST  Q
```

2.5.6.8 See also

ANY_TO_BOOL ANY_TO_SINT ANY_TO_INT ANY_TO_DINT ANY_TO_LINT ANY_TO_LREAL ANY_TO_TIME ANY_TO_STRING

2.5.7 ANY_TO_TIME

Operator - Converts the input into time value.

2.5.7.1 Inputs

`IN : ANY` Input value

2.5.7.2 Outputs

`Q : TIME` Value converted to time

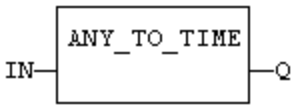
2.5.7.3 Remarks

For BOOL input data types, the output is t#0 ms or t#1 ms. For DINT or REAL input data type, the output is the time represented by the input number as a number of milliseconds. For STRING inputs, the output is the time represented by the string, or t#0 ms if the string does not represent a valid time. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_TIME function converts the current result.

2.5.7.4 ST Language

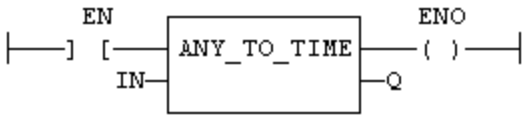
```
Q := ANY_TO_TIME (IN);
```

2.5.7.5 FBD Language



2.5.7.6 FFLD Language

```
(* The conversion is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.5.7.7 IL Language:

```
Op1: FFLD  IN
ANY_TO_TIME
ST  Q
```

2.5.7.8 See also

ANY_TO_BOOL ANY_TO_SINT ANY_TO_INT ANY_TO_DINT ANY_TO_LINT ANY_TO_REAL ANY_TO_LREAL ANY_TO_STRING

2.5.8 ANY_TO_SINT / ANY_TO_USINT

Operator - Converts the input into a small (8 bit) integer value (can be unsigned with ANY_TO_USINT).

2.5.8.1 Inputs

`IN : ANY` Input value

2.5.8.2 Outputs

`Q : SINT` Value converted to a small (8 bit) integer

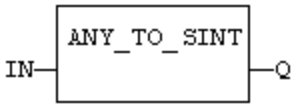
2.5.8.3 Remarks

For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_SINT function converts the current result.

2.5.8.4 ST Language

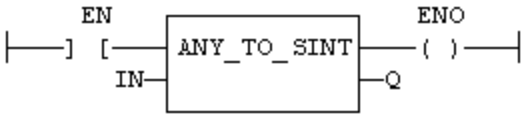
```
Q := ANY_TO_SINT (IN);
```

2.5.8.5 FBD Language



2.5.8.6 FFLD Language

```
(* The conversion is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.5.8.7 IL Language

```
Op1: FFLD  IN
   ANY_TO_SINT
   ST  Q
```

2.5.8.8 See also

ANY_TO_BOOL ANY_TO_INT ANY_TO_DINT ANY_TO_LINT ANY_TO_REAL ANY_TO_LREAL ANY_TO_TIME ANY_TO_STRING

2.5.9 ANY_TO_STRING

Operator - Converts the input into string value.

2.5.9.1 Inputs

`IN : ANY` Input value

2.5.9.2 Outputs

`Q : STRING` Value converted to string

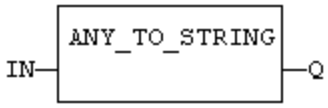
2.5.9.3 Remarks

For BOOL input data types, the output is '1' or '0' for TRUE and FALSE respectively. For DINT, REAL or TIME input data types, the output is the string representation of the input number. It is a number of milliseconds for TIME inputs. In FFLD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL language, the ANY_TO_STRING function converts the current result.

2.5.9.4 ST Language

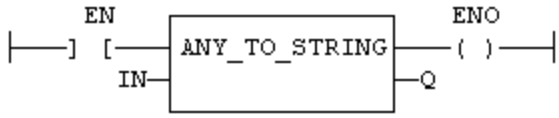
```
Q := ANY_TO_STRING (IN);
```

2.5.9.5 FBD Language



2.5.9.6 FFLD Language

```
(* The conversion is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.5.9.7 IL Language:

```
Op1: FFLD  IN
   ANY_TO_STRING
   ST  Q
```

2.5.9.8 See also

ANY_TO_BOOL ANY_TO_SINT ANY_TO_INT ANY_TO_DINT ANY_TO_LINT ANY_TO_REAL ANY_TO_LREAL ANY_TO_TIME

2.5.10 NUM_TO_STRING

Function- Converts a number into string value.

2.5.10.1 Inputs

IN : ANY Input number.
WIDTH : DINT Wished length for the output string (see remarks)
DIGITS : DINT Number of digits after decimal point

2.5.10.2 Outputs

Q : STRING Value converted to string.

2.5.10.3 Remarks

This function converts any numerical value to a string. Unlike the ANY_TO_STRING function, it allows you to specify a wished length and a number of digits after the decimal points.

If WIDTH is 0, the string is formatted with the necessary length.

If WIDTH is greater than 0, the string is completed with heading blank characters in order to match the value of WIDTH.

If WIDTH is greater than 0, the string is completed with trailing blank characters in order to match the absolute value of WIDTH.

If DIGITS is 0 then neither decimal part nor point are added.

If DIGITS is greater than 0, the corresponding number of decimal digits are added. '0' digits are added if necessary

If the value is too long for the specified width, then the string is filled with '*' characters.

2.5.10.4 Examples

```
Q := NUM_TO_STRING (123.4, 8, 2); (* Q is ' 123.40' *)
Q := NUM_TO_STRING (123.4, -8, 2); (* Q is '123.40 ' *)
Q := NUM_TO_STRING (1.333333, 0, 2); (* Q is '1.33' *)
Q := NUM_TO_STRING (1234, 3, 0); (* Q is '***' *)
```

2.5.11 BCD_TO_BIN

Function - Converts a BCD (Binary Coded Decimal) value to a binary value

2.5.11.1 Inputs

`IN : DINT` Integer value in BCD

2.5.11.2 Outputs

`Q : DINT` Value converted to integer
or 0 if IN is not a valid positive BCD value

2.5.11.3 Truth table (examples)

IN	Q
-2	0 (invalid)
0	0
16 (16#10)	10
15 (16#0F)	0 (invalid)

2.5.11.4 Remarks

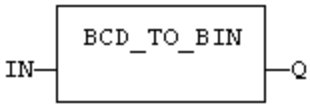
The input must be positive and must represent a valid BCD value. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.5.11.5 ST Language

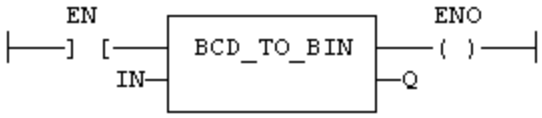
```
Q := BCD_TO_BIN (IN);
```

2.5.11.6 FBD Language



2.5.11.7 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.5.11.8 IL Language:

```
Op1: FFLD       IN
   BCD_TO_BIN
   ST       Q
```

See also

BIN_TO_BCD

2.5.12 BIN_TO_BCD

Function - Converts a binary value to a BCD (Binary Coded Decimal) value

2.5.12.1 Inputs

`IN : DINT` Integer value

2.5.12.2 Outputs

`Q : DINT` Value converted to BCD or 0 if IN is less than 0

2.5.12.3 Truth table (examples)

IN	Q
-2	0 (invalid)
0	0
10	16 (16#10)
22	34 (16#22)

2.5.12.4 Remarks

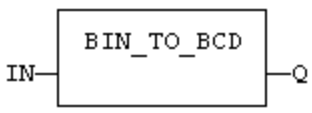
The input must be positive. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.5.12.5 ST Language

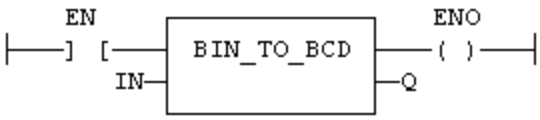
```
Q := BIN_TO_BCD (IN);
```

2.5.12.6 FBD Language



2.5.12.7 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.5.12.8 IL Language:

```
Op1: FFLD       IN
   BIN_TO_BCD
   ST       Q
```

See also

BCD_TO_BIN

2.6 Selectors

Below are the standard functions that perform data selection:

SEL	2 integer inputs
MUX4	4 integer inputs
MUX8	8 integer inputs

2.6.1 MUX4

Function - Select one of the inputs - 4 inputs.

2.6.1.1 Inputs

```
SELECT  :  DINT  Selection command
IN1     :  ANY   First input
IN2     :  ANY   Second input
...     :
IN4     :  ANY   Last input
```

2.6.1.2 Outputs

```
Q  :  ANY    IN1 or IN2 ... or IN4 depending on SELECT (see truth table)
```

2.6.1.3 Truth table

SELECT	Q
0	IN1
1	IN2
2	IN3
3	IN4
other	0

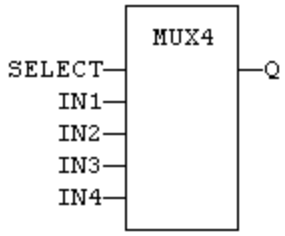
2.6.1.4 Remarks

In FFLD language, the input rung (EN) enables the selection. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by comas.

2.6.1.5 ST Language

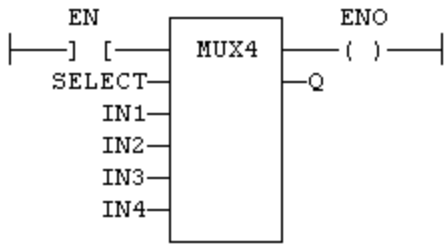
```
Q := MUX4 (SELECT, IN1, IN2, IN3, IN4);
```

2.6.1.6 FBD Language



2.6.1.7 FFLD Language

```
(* the selection is performed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.6.1.8 IL Language

```
Op1: FFLD   SELECT
     MUX4 IN1, IN2, IN3, IN4
     ST   Q
```

See also

SEL MUX8

2.6.2 MUX8

Function - Select one of the inputs - 8 inputs.

2.6.2.1 Inputs

SELECT	:	DINT	Selection command
IN1	:	ANY	First input
IN2	:	ANY	Second input
...	:		
IN8	:	ANY	Last input

2.6.2.2 Outputs

Q : ANY IN1 or IN2 ... or IN8 depending on SELECT (see truth table)

2.6.2.3 Truth table

SELECT	Q
0	IN1
1	IN2
2	IN3
3	IN4
4	IN5
5	IN6
6	IN7
7	IN8
other	0

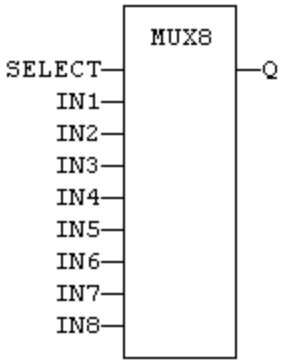
2.6.2.4 Remarks

In FFLD language, the input rung (EN) enables the selection. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by comas.

2.6.2.5 ST Language

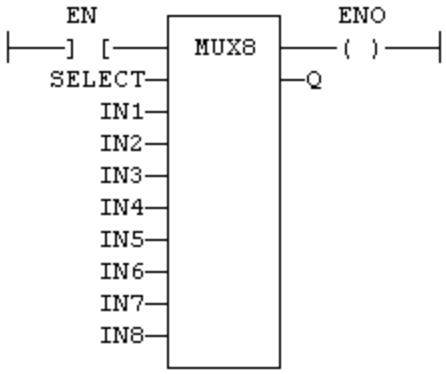
```
Q := MUX8 (SELECT, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8);
```

2.6.2.6 FBD Language



2.6.2.7 FFLD Language

```
(* the selection is performed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.6.2.8 IL Language

Not available

Op1: FFLD SELECT
MUX8 IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8
ST Q

See also

SEL MUX4

2.6.3 SEL

Function - Select one of the inputs - 2 inputs.

2.6.3.1 Inputs

SELECT	:	BOOL	Selection command
IN1	:	ANY	First input
IN2	:	ANY	Second input

2.6.3.2 Outputs

Q : ANY IN1 if SELECT is FALSE; IN2 if SELECT is TRUE

2.6.3.3 Truth table

SELECT	Q
0	IN1
1	IN2

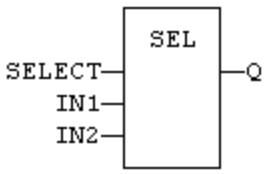
2.6.3.4 Remarks

In FFLD language, the selector command is the input rung. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by comas.

2.6.3.5 ST Language

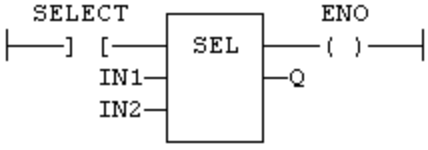
Q := SEL (SELECT, IN1, IN2);

2.6.3.6 FBD Language



2.6.3.7 FFLD Language

(* the input rung is the selector *)
(* ENO has the same value as SELECT *)



2.6.3.8 IL Language

Op1: FFLD SELECT
SEL IN1, IN2
ST Q

See also

MUX4 MUX8

2.7 Registers

Below are the standard functions for managing 8 bit to 32 bit registers:

SHL	shift left
SHR	shift right
ROL	rotation left
ROR	rotation right

Below are advanced functions for register manipulation:

MBShift	multibyte shift / rotate

The following functions enable bit to bit operations on a 8 bit to 32 bit integers:

AND_MASK	boolean AND
OR_MASK	boolean OR
XOR_MASK	exclusive OR
NOT_MASK	boolean negation

The following functions enable to pack/unpack 8, 16 and 32 bit registers

LOBYTE	Get the lowest byte of a word
HIBYTE	Get the highest byte of a word
LOWORD	Get the lowest word of a double word
HIWORD	Get the highest word of a double word
MAKEWORD	Pack bytes to a word
MAKEDWORD	Pack words to a double word
PACK8	Pack bits in a byte
UNPACK8	Extract bits from a byte

The following functions provide bit access in 8 bit to 32 bit integers:

SETBIT	Set a bit in a register
TESTBIT	Test a bit of a register

The following functions have been deprecated. They are available for backwards compatibility only. The functions listed above should be used for all current and future development.

AND_WORD	AND_BYTE
OR_WORD	OR_BYTE
NOT_WORD	NOT_BYTE
XOR_WORD	XOR_BYTE
ROLW	RORW
ROLB	RORB
SHLW	SHRW
SHLB	SHRB

2.7.1 AND_MASK

Function - Performs a bit to bit AND between two integer values

2.7.1.1 Inputs

`IN` : `ANY` First input
`MSK` : `ANY` Second input (AND mask)

2.7.1.2 Outputs

`Q : ANY` AND mask between IN and MSK inputs

2.7.1.3 Remarks

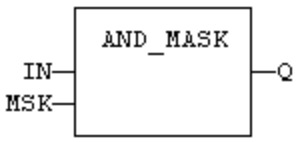
Arguments can be signed or unsigned integers from 8 to 32 bits.

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

2.7.1.4 ST Language

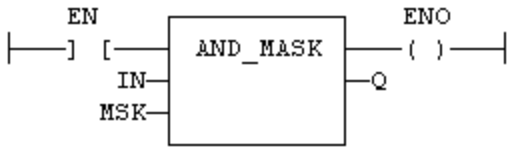
```
Q := AND_MASK (IN, MSK);
```

2.7.1.5 FBD Language



2.7.1.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```



2.7.1.7 IL Language:

```
Op1: FFLD     IN
   AND_MASK MSK
   ST     Q
```

See also

OR_MASK XOR_MASK NOT_MASK

2.7.2 HIBYTE

Function - Get the most significant byte of a word

2.7.2.1 Inputs

`IN : UINT` 16 bit register

2.7.2.2 Outputs

`Q : USINT` Most significant byte

2.7.2.3 Remarks

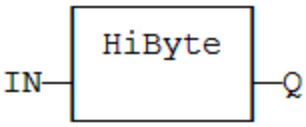
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.7.2.4 ST Language

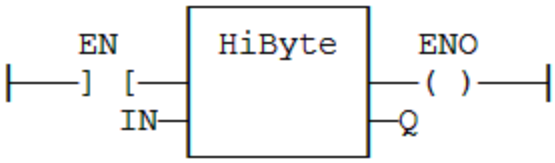
Q := HIBYTE (IN);

2.7.2.5 FBD Language



2.7.2.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.7.2.7 IL Language:

Op1: FFLD IN
HIBYTE
ST Q

See also

LOBYTE LOWORD HIWORD MAKEWORD MAKEDWORD

2.7.3 LOBYTE

Function - Get the less significant byte of a word

2.7.3.1 Inputs

`IN : UINT` 16 bit register

2.7.3.2 Outputs

`Q : USINT` Lowest significant byte

2.7.3.3 Remarks

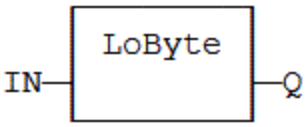
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.7.3.4 ST Language

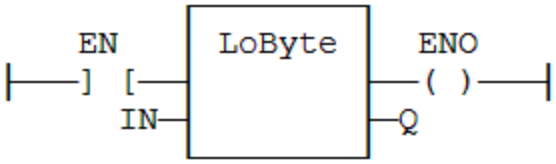
Q := LOBYTE (IN);

2.7.3.5 FBD Language



2.7.3.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.7.3.7 IL Language:

```
Op1: FFLD  IN
   LOBYTE
   ST  Q
```

See also

HIBYTE LOWORD HIWORD MAKEWORD MAKEDWORD

2.7.4 HIWORD

Function - Get the most significant word of a double word

2.7.4.1 Inputs

`IN : UDINT` 32 bit register

2.7.4.2 Outputs

`Q : UINT` Most significant word

2.7.4.3 Remarks

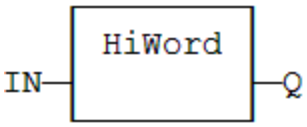
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.7.4.4 ST Language

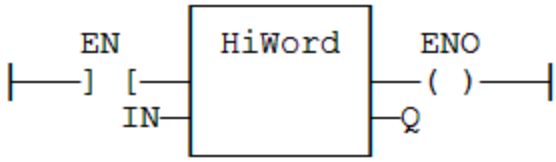
```
Q := HIWORD (IN);
```

2.7.4.5 FBD Language



2.7.4.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.7.4.7 IL Language:

```
Op1: FFLD  IN
   HIWORD
   ST  Q
```

See also

LOBYTE HIBYTE LOWORD MAKEWORD MAKEDWORD

2.7.5 LOWORD

Function - Get the less significant word of a double word

2.7.5.1 Inputs

`IN : UDINT` 32 bit register

2.7.5.2 Outputs

`Q : UINT` Lowest significant word

2.7.5.3 Remarks

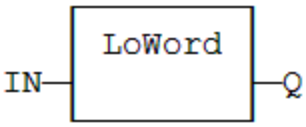
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.7.5.4 ST Language

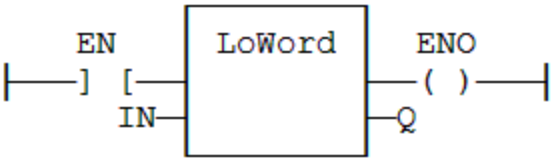
```
Q := LOWORD (IN);
```

2.7.5.5 FBD Language



2.7.5.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.7.5.7 IL Language:

```
Op1: FFLD  IN
   LOWORD
   ST  Q
```

See also

LOBYTE HIBYTE HIWORD MAKEWORD MAKEDWORD

2.7.6 MAKEDWORD

Function - Builds a double word as the concatenation of two words

2.7.6.1 Inputs

`HI : USINT` Highest significant word
`LO : USINT` Lowest significant word

2.7.6.2 Outputs

`Q : UINT` 32 bit register

2.7.6.3 Remarks

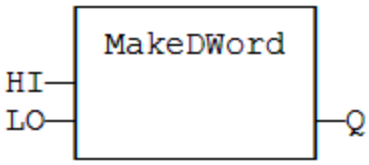
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input must be loaded in the current result before calling the function.

2.7.6.4 ST Language

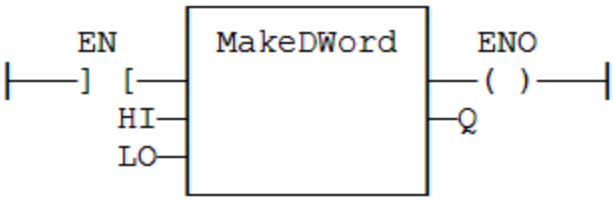
```
Q := MAKEDWORD (HI, LO);
```

2.7.6.5 FBD Language



2.7.6.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.7.6.7 IL Language:

```
Op1: FFLD       HI
   MAKEDWORD LO
   ST       Q
```

See also

LOBYTE HIBYTE LOWORD HIWORD MAKEWORD

2.7.7 MAKEWORD

Function - Builds a word as the concatenation of two bytes

2.7.7.1 Inputs

`HI : USINT` Highest significant byte
`LO : USINT` Lowest significant byte

2.7.7.2 Outputs

`Q : UINT` 16 bit register

2.7.7.3 Remarks

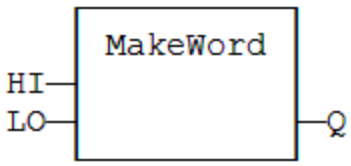
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input must be loaded in the current result before calling the function.

2.7.7.4 ST Language

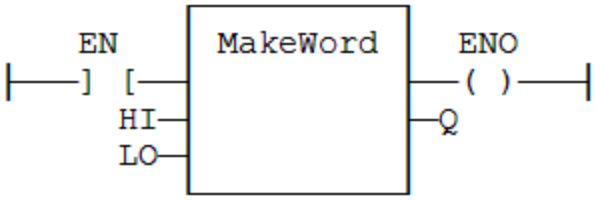
```
Q := MAKEWORD (HI, LO);
```

2.7.7.5 FBD Language



2.7.7.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.7.7.7 IL Language:

```
Op1: FFLD     HI
     MAKEWORD LO
     ST       Q
```

See also

LOBYTE HIBYTE LOWORD HIWORD MAKEDWORD

2.7.8 MBSHIFT

Function - Multibyte shift / rotate

2.7.8.1 Inputs

`Buffer : SINT/USINT`	Array of bytes
`Pos : DINT`	Base position in the array
`NbByte : DINT`	Number of bytes to be shifted/rotated
`NbShift : DINT`	Number of shifts or rotations
`ToRight : BOOL`	TRUE for right / FALSE for left
`Rotate: BOOL`	TRUE for rotate / FALSE for shift
`InBit : BOOL`	Bit to be introduced in a shift

2.7.8.2 Outputs

`Q : BOOL`	TRUE if successful

2.7.8.3 Remarks

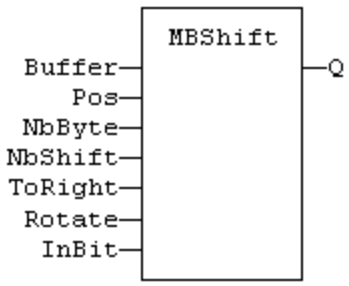
Use the "ToRight" argument to specify a shift to the left (FALSE) or to the right (TRUE). Use the "Rotate" argument to specify either a shift (FALSE) or a rotation (TRUE). In case of a shift, the "InBit" argument specifies the value of the bit that replaces the last shifted bit.

In FFLD language, the rung input (EN) validates the operation. The rung output is the result ("Q").

2.7.8.4 ST Language

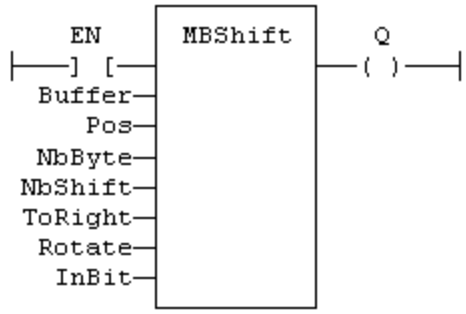
```
Q := MBShift (Buffer, Pos, NbByte, NbShift, ToRight, Rotate, InBit);
```

2.7.8.5 FBD Language



2.7.8.6 FFLD Language

```
(* the function is called only if EN is TRUE  *)
```



2.7.8.7 IL Language:

Not available

2.7.9 NOT_MASK

Function - Performs a bit to bit negation of an integer value

2.7.9.1 Inputs

IN : ANY Integer input

2.7.9.2 Outputs

Q : ANY Bit to bit negation of the input

2.7.9.3 Remarks

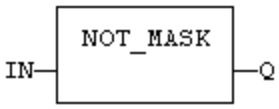
Arguments can be signed or unsigned integers from 8 to 32 bits.

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the parameter (IN) must be loaded in the current result before calling the function.

2.7.9.4 ST Language

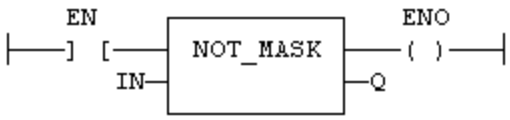
```
Q := NOT_MASK (IN);
```

2.7.9.5 FBD Language



2.7.9.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```



2.7.9.7 IL Language:

```
Op1: FFLD     IN
   NOT_MASK
   ST     Q
```

See also

AND_MASK OR_MASK XOR_MASK

2.7.10 OR_MASK

Function - Performs a bit to bit OR between two integer values

2.7.10.1 Inputs

IN : ANY First input
MSK : ANY Second input (OR mask)

2.7.10.2 Outputs

Q : ANY OR mask between IN and MSK inputs

2.7.10.3 Remarks

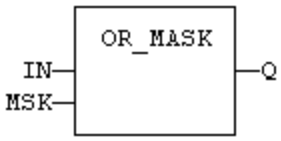
Arguments can be signed or unsigned integers from 8 to 32 bits.

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

2.7.10.4 ST Language

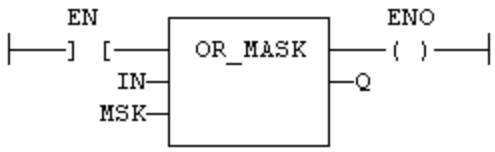
Q := OR_MASK (IN, MSK);

2.7.10.5 FBD Language



2.7.10.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```



2.7.10.7 IL Language:

```
Op1: FFLD    IN
   OR_MASK MSK
   ST    Q
```

See also

AND_MASK XOR_MASK NOT_MASK

2.7.11 PACK8

Function - Builds a byte with bits

2.7.11.1 Inputs

IN0 : BOOL		Less significant bit
...		
IN7 : BOOL		Most significant bit

2.7.11.2 Outputs

Q : USINT	Byte built with input bits

2.7.11.3 Remarks

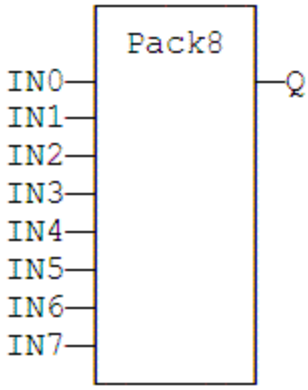
In FFLD language, the input rung is the IN0 input. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.7.11.4 ST Language

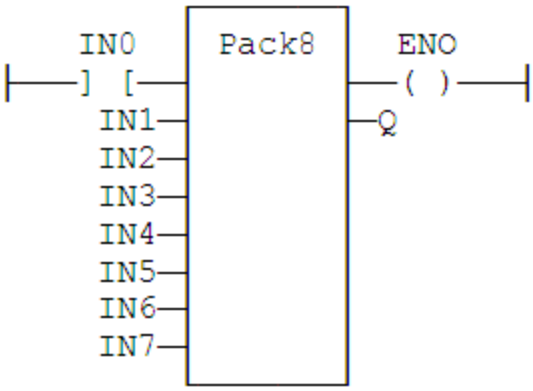
Q := PACK8 (IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7);

2.7.11.5 FBD Language



2.7.11.6 FFLD Language

```
(* ENO keeps the same value as EN *)
```



2.7.11.7 IL Language

```
Op1: FFLD   IN0
     PACK8 IN1, IN2, IN3, IN4, IN5, IN6, IN7
     ST   Q
```

See also

UNPACK8

2.7.12 ROL

Function - Rotate bits of a register to the left.

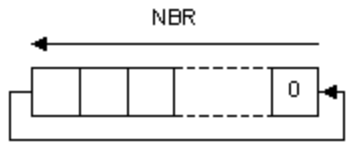
2.7.12.1 Inputs

`IN : ANY` register
`NBR : DINT` Number of rotations (each rotation is 1 bit)

2.7.12.2 Outputs

`Q : ANY` Rotated register

2.7.12.3 Diagram



2.7.12.4 Remarks

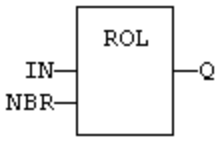
Arguments can be signed or unsigned integers from 8 to 32 bits.

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.7.12.5 ST Language

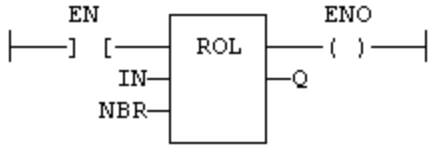
```
Q := ROL (IN, NBR);
```

2.7.12.6 FBD Language



2.7.12.7 FFLD Language

```
(* The rotation is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.7.12.8 IL Language:

```
Op1: FFLD  IN
     ROL NBR
     ST  Q
```

See also

SHL SHR ROR SHLb SHRb ROLb RORb SHLw SHRw ROLw RORw

2.7.13 ROR

Function - Rotate bits of a register to the right.

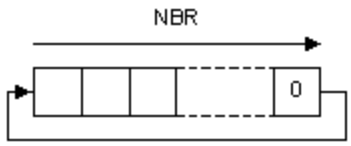
2.7.13.1 Inputs

`IN : ANY` register
`NBR : ANY` Number of rotations (each rotation is 1 bit)

2.7.13.2 Outputs

`Q : ANY` Rotated register

2.7.13.3 Diagram



2.7.13.4 Remarks

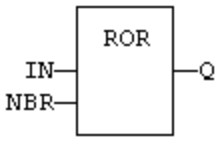
Arguments can be signed or unsigned integers from 8 to 32 bits.

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.7.13.5 ST Language

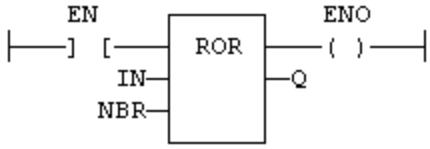
```
Q := ROR (IN, NBR);
```

2.7.13.6 FBD Language



2.7.13.7 FFLD Language

```
(* The rotation is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.7.13.8 IL Language:

```
Op1: FFLD  IN
     ROR NBR
     ST  Q
```

See also

SHL SHR ROL SHLb SHRb ROLb RORb SHLw SHRw ROLw RORw

2.7.14 RORb / ROR_SINT / ROR_USINT / ROR_BYTE

Function - Rotate bits of a register to the right.

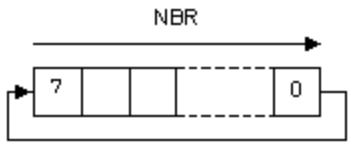
2.7.14.1 Inputs

`IN : SINT` 8 bit register
`NBR : SINT` Number of rotations (each rotation is 1 bit)

2.7.14.2 Outputs

`Q : SINT` Rotated register

2.7.14.3 Diagram



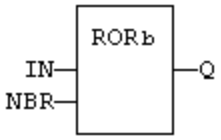
2.7.14.4 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.7.14.5 ST Language

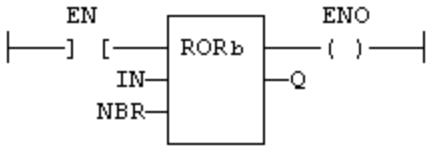
```
Q := RORb (IN, NBR);
```

2.7.14.6 FBD Language



2.7.14.7 FFLD Language

```
(* The rotation is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.7.14.8 IL Language:

```
Op1: FFLD   IN
     RORb NBR
     ST   Q
```

2.7.14.9 See also

SHL SHR ROL ROR SHLb SHRb ROLb SHLw SHRw ROLw RORw

2.7.15 RORw / ROR_INT / ROR_UINT / ROR_WORD

Function - Rotate bits of a register to the right.

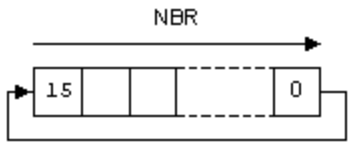
2.7.15.1 Inputs

`IN : INT`	16 bit register
`NBR : INT`	Number of rotations (each rotation is 1 bit)

2.7.15.2 Outputs

`Q : INT`	Rotated register

2.7.15.3 Diagram



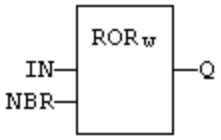
2.7.15.4 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.7.15.5 ST Language

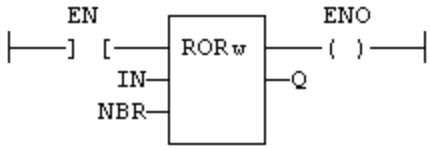
```
Q := RORw (IN, NBR);
```

2.7.15.6 FBD Language



2.7.15.7 FFLD Language

```
(* The rotation is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.7.15.8 IL Language:

```
Op1: FFLD  IN
     RORw NBR
     ST   Q
```

2.7.15.9 See also

SHL SHR ROL ROR SHLb SHRb ROLb RORb SHLw SHRw ROLw

2.7.16 SETBIT

Function - Set a bit in an integer register.

2.7.16.1 Inputs

IN : ANY	8 to 32 bit integer register
BIT : DINT	Bit number (0 = less significant bit)
VAL : BOOL	Bit value to apply

2.7.16.2 Outputs

Q : ANY	Modified register

2.7.16.3 Remarks

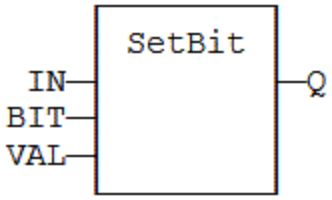
Types LINT, REAL, LREAL, TIME and STRING are not supported for IN and Q. IN and Q must have the same type. In case of invalid arguments (bad bit number or invalid input type) the function returns the value of IN without modification.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

2.7.16.4 ST Language

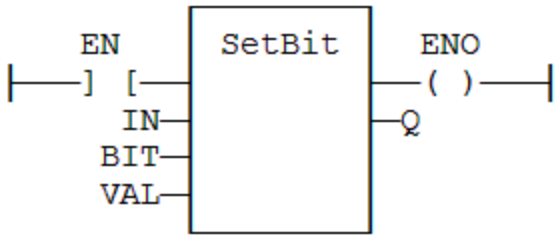
```
Q := SETBIT (IN, BIT, VAL);
```

2.7.16.5 FBD Language



2.7.16.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.7.16.7 IL Language

Not available

See also

TESTBIT

2.7.17 SHL

Function - Shift bits of a register to the left.

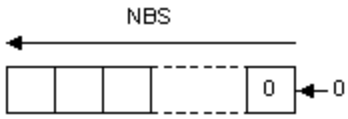
2.7.17.1 Inputs

`IN : ANY` register
`NBS : ANY` Number of shifts (each shift is 1 bit)

2.7.17.2 Outputs

`Q : ANY` Shifted register

2.7.17.3 Diagram



2.7.17.4 Remarks

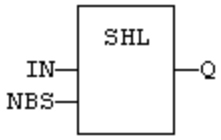
Arguments can be signed or unsigned integers from 8 to 32 bits.

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.7.17.5 ST Language

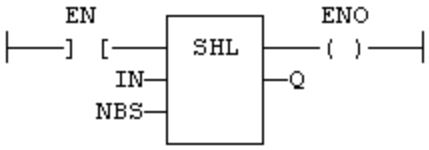
```
Q := SHL (IN, NBS);
```

2.7.17.6 FBD Language



2.7.17.7 FFLD Language

```
(* The shift is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.7.17.8 IL Language:

```
Op1: FFLD  IN
     SHL NBS
     ST  Q
```

See also

SHR ROL ROR SHLb SHRb ROLb RORb SHLw SHRw ROLw RORw

2.7.18 SHR

Function - Shift bits of a register to the right.

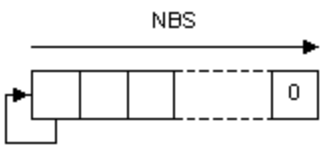
2.7.18.1 Inputs

`IN : ANY` register
`NBS : ANY` Number of shifts (each shift is 1 bit)

2.7.18.2 Outputs

`Q : ANY` Shifted register

2.7.18.3 Diagram



2.7.18.4 Remarks

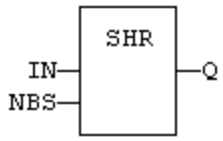
Arguments can be signed or unsigned integers from 8 to 32 bits.

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

2.7.18.5 ST Language

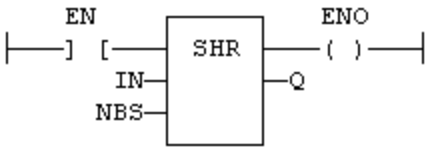
```
Q := SHR (IN, NBS);
```

2.7.18.6 FBD Language



2.7.18.7 FFLD Language

```
(* The shift is executed only if EN is TRUE *)
(* ENO has the same value as EN *)
```



2.7.18.8 IL Language:

```
Op1: FFLD  IN
     SHR NBS
     ST  Q
```

See also

SHL ROL ROR SHLb SHRb ROLb RORb SHLw SHRw ROLw RORw

2.7.19 TESTBIT

Function - Test a bit of an integer register.

2.7.19.1 Inputs

`IN : ANY`	8 to 32 bit integer register
`BIT : DINT`	Bit number (0 = less significant bit)

2.7.19.2 Outputs

`Q : BOOL`	Bit value

2.7.19.3 Remarks

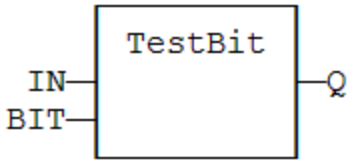
Types LINT, REAL, LREAL, TIME and STRING are not supported for IN and Q. IN and Q must have the same type. In case of invalid arguments (bad bit number or invalid input type) the function returns FALSE.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung is the output of the function.

2.7.19.4 ST Language

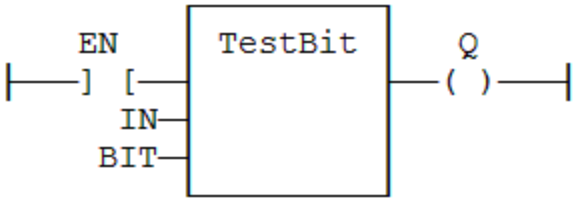
```
Q := TESTBIT (IN, BIT);
```

2.7.19.5 FBD Language



2.7.19.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
```



2.7.19.7 IL Language

Not available

See also

SETBIT

2.7.20 UNPACK8

Function block - Extract bits of a byte

2.7.20.1 Inputs

`IN : USINT` 8 bit register

2.7.20.2 Outputs

`Q0 : BOOL` Less significant bit
...
`Q7 : BOOL` Most significant bit

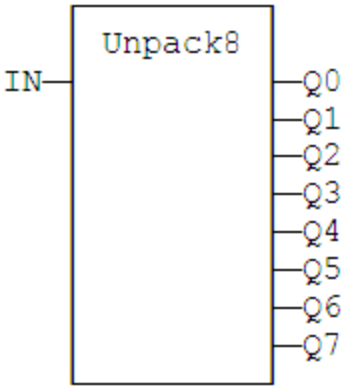
2.7.20.3 Remarks

In FFLD language, the output rung is the Q0 output. The operation is executed only in the input rung (EN) is TRUE.

2.7.20.4 ST Language

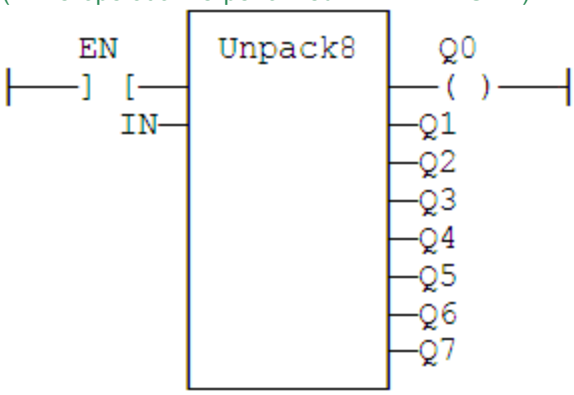
```
(* MyUnpack is a declared instance of the UNPACK8 function block *)
MyUnpack (IN);
Q0 := MyUnpack.Q0;
Q1 := MyUnpack.Q1;
Q2 := MyUnpack.Q2;
Q3 := MyUnpack.Q3;
Q4 := MyUnpack.Q4;
Q5 := MyUnpack.Q5;
Q6 := MyUnpack.Q6;
Q7 := MyUnpack.Q7;
```

2.7.20.5 FBD Language



2.7.20.6 FFLD Language

(* The operation is performed if EN = TRUE *)



2.7.20.7 IL Language:

```
(* MyUnpack is a declared instance of the UNPACK8 function block *)
Op1: CAL MyUnpack (IN)
   FFLD  MyUnpack.Q0
   ST  Q0
   (* ... *)
   FFLD  MyUnpack.Q7
   ST  Q7
```

See also

PACK8

2.7.21 XOR_MASK

Function - Performs a bit to bit exclusive OR between two integer values

2.7.21.1 Inputs

`IN` : `ANY` First input
`MSK` : `ANY` Second input (XOR mask)

2.7.21.2 Outputs

`Q` : `ANY` Exclusive OR mask between IN and MSK inputs

2.7.21.3 Remarks

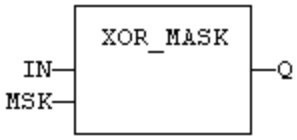
Arguments can be signed or unsigned integers from 8 to 32 bits.

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

2.7.21.4 ST Language

```
Q := XOR_MASK (IN, MSK);
```

2.7.21.5 FBD Language



2.7.21.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```

2.7.21.7 IL Language:

```
Op1: FFLD      IN
   XOR_MASK MSK
   ST      Q
```

See also

AND_MASK OR_MASK NOT_MASK

2.8 Counters

Below are the standard blocks for managing counters:

CTU	Up counter
CTD	Down Counter
CTUD	Up / Down Counter

2.8.1 CTD / CTDr

Function Block - Down counter.

2.8.1.1 Inputs

`CD : BOOL` Enable counting. Counter is decreased on each call when CD is TRUE

`LOAD : BOOL` Re-load command. Counter is set to PV when called with LOAD to TRUE

`PV : DINT` Programmed maximum value

2.8.1.2 Outputs

`Q : BOOL` TRUE when counter is empty, i.e. when CV = 0
`CV : DINT` Current value of the counter

2.8.1.3 Remarks

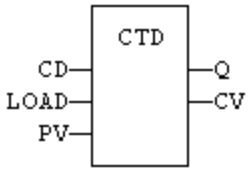
The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CD input. Use R_TRIG or F_TRIG function block for counting pulses of CD input signal. In FFLD language, CD is the input rung. The output rung is the Q output.

CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all boolean inputs (CU, CD, RESET, LOAD) have an implicit rising edge detection included. Note that these counters can be not supported on some target systems.

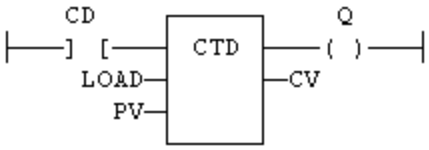
2.8.1.4 ST Language

```
(* MyCounter is a declared instance of CTD function block *)
MyCounter (CD, LOAD, PV);
Q := MyCounter.Q;
CV := MyCounter.CV;
```

2.8.1.5 FBD Language



2.8.1.6 FFLD Language



2.8.1.7 IL Language:

```
(* MyCounter is a declared instance of CTD function block *)
Op1: CAL   MyCounter (CD, LOAD, PV)
FFLD   MyCounter.Q
ST   Q
FFLD   MyCounter.CV
ST   CV
```

See also

CTU CTUD

2.8.2 CTU / CTUr

Function Block - Up counter.

2.8.2.1 Inputs

`CU : BOOL` Enable counting. Counter is increased on each call when CU is TRUE
`RESET : BOOL` Reset command. Counter is reset to 0 when called with RESET to

TRUE
`PV : DINT` Programmed maximum value

2.8.2.2 Outputs

`Q : BOOL` TRUE when counter is full, i.e. when CV = PV
`CV : DINT` Current value of the counter

2.8.2.3 Remarks

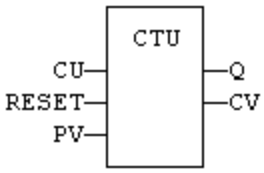
The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CU input. Use R_TRIG or F_TRIG function block for counting pulses of CU input signal. In FFLD language, CU is the input rung. The output rung is the Q output.

CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all boolean inputs (CU, CD, RESET, LOAD) have an implicit rising edge detection included. Note that these counters can be not supported on some target systems.

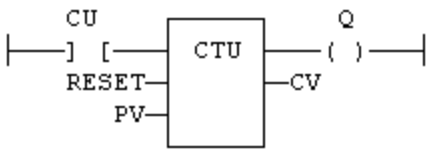
2.8.2.4 ST Language

```
(* MyCounter is a declared instance of CTU function block *)
MyCounter (CU, RESET, PV);
Q := MyCounter.Q;
CV := MyCounter.CV;
```

2.8.2.5 FBD Language



2.8.2.6 FFLD Language



2.8.2.7 IL Language:

```
(* MyCounter is a declared instance of CTU function block *)
Op1: CAL   MyCounter (CU, RESET, PV)
FFLD   MyCounter.Q
ST   Q
FFLD   MyCounter.CV
ST   CV
```

See also

CTD CTUD

2.8.3 CTUD / CTUDr

Function Block - Up/down counter.

2.8.3.1 Inputs

`CU : BOOL` Enable counting. Counter is increased on each call when CU is TRUE
`CD : BOOL` Enable counting. Counter is decreased on each call when CD is TRUE
`RESET : BOOL` Reset command. Counter is reset to 0 called with RESET to TRUE
`LOAD : BOOL` Re-load command. Counter is set to PV when called with LOAD to TRUE
`PV : DINT` Programmed maximum value

2.8.3.2 Outputs

`QU : BOOL` TRUE when counter is full, i.e. when CV = PV
`QD : BOOL` TRUE when counter is empty, i.e. when CV = 0
`CV : DINT` Current value of the counter

2.8.3.3 Remarks

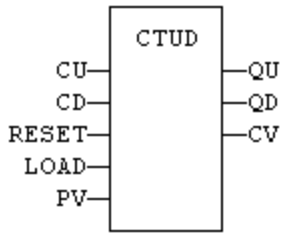
The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CU and CD inputs. Use R_TRIG or F_TRIG function blocks for counting pulses of CU or CD input signals. In FFLD language, CU is the input rung. The output rung is the QU output.

CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all boolean inputs (CU, CD, RESET, LOAD) have an implicit rising edge detection included. Note that these counters can be not supported on some target systems.

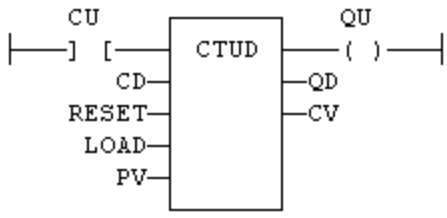
2.8.3.4 ST Language

```
(* MyCounter is a declared instance of CTUD function block *)
MyCounter (CU, CD, RESET, LOAD, PV);
QU := MyCounter.QU;
QD := MyCounter.QD;
CV := MyCounter.CV;
```

2.8.3.5 FBD Language



2.8.3.6 FFLD Language



2.8.3.7 IL Language:

```
(* MyCounter is a declared instance of CTUD function block *)
Op1: CAL   MyCounter (CU, CD, RESET, LOAD, PV)
FFLD    MyCounter.QU
ST    QU
FFLD    MyCounter.QD
ST    QD
FFLD    MyCounter.CV
ST    CV
```

See also

CTU CTD

2.9 Timers

Below are the standard functions for managing timers:

TON	On timer
TOF	Off timer
TP	Pulse timer
BLINK	Blinker
BLINKA	Asymmetric blinker
PLS	Pulse signal generator
TMU	Up-counting stop watch
TMUsec	Up-counting stop watch (seconds)
TMD	Down-counting stop watch

2.9.1 BLINK

Function Block - Blinker.

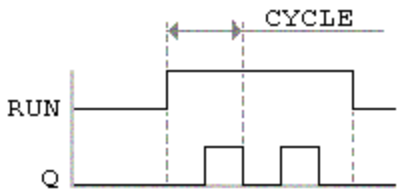
2.9.1.1 Inputs

`RUN : BOOL` Enabling command
`CYCLE : TIME` Blinking period

2.9.1.2 Outputs

`Q : BOOL` Output blinking signal

2.9.1.3 Time diagram



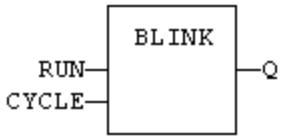
2.9.1.4 Remarks

The output signal is FALSE when the RUN input is FALSE. The CYCLE input is the complete period of the blinking signal. In FFLD language, the input rung is the IN command. The output rung is the Q output signal.

2.9.1.5 ST Language

```
(* MyBlinker is a declared instance of BLINK function block *)
MyBlinker (RUN, CYCLE);
Q := MyBlinker.Q;
```

2.9.1.6 FBD Language



2.9.1.7 FFLD Language

2.9.1.8 IL Language

```
(* MyBlinker is a declared instance of BLINK function block *)
Op1: CAL MyBlinker (RUN, CYCLE)
   FFLD  MyBlinker.Q
   ST  Q
```

See also

TON TOF TP

2.9.2 BLINKA

Function Block - Asymmetric blinker.

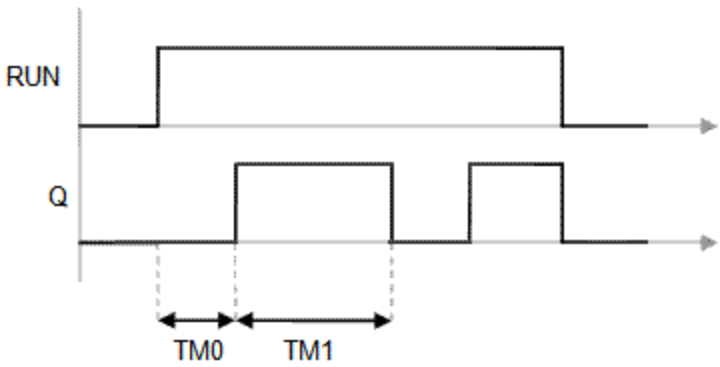
2.9.2.1 Inputs

`RUN : BOOL`	Enabling command
`TM0 : TIME`	Duration of FALSE state on output
`TM1 : TIME`	Duration of TRUE state on output

2.9.2.2 Outputs

`Q : BOOL`	Output blinking signal

2.9.2.3 Time diagram



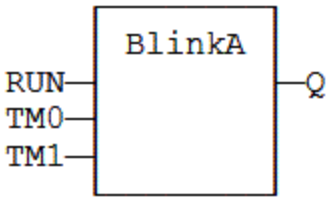
2.9.2.4 Remarks

The output signal is FALSE when the RUN input is FALSE. In FFLD language, the input rung is the IN command. The output rung is the Q output signal.

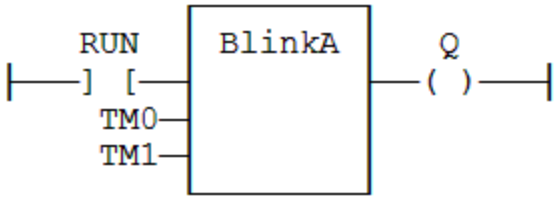
2.9.2.5 ST Language

```
(* MyBlinker is a declared instance of BLINKA function block *)
MyBlinker (RUN, TM0, TM1);
Q := MyBlinker.Q;
```

2.9.2.6 FBD Language



2.9.2.7 FFLD Language



2.9.2.8 IL Language:

```
(* MyBlinker is a declared instance of BLINKA function block *)
Op1: CAL MyBlinker (RUN, TM0, TM1)
   FFLD  MyBlinker.Q
   ST  Q
```

See also

TON TOF TP

2.9.3 PLS

Function Block - Pulse signal generator

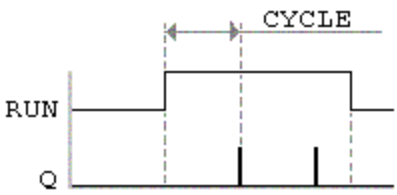
2.9.3.1 Inputs

`RUN : BOOL` Enabling command
`CYCLE : TIME` Signal period

2.9.3.2 Outputs

`Q : BOOL` Output pulse signal

2.9.3.3 Time diagram



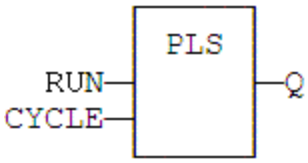
2.9.3.4 Remarks

On every period, the output is set to TRUE during one cycle only. In FFLD language, the input rung is the IN command. The output rung is the Q output signal.

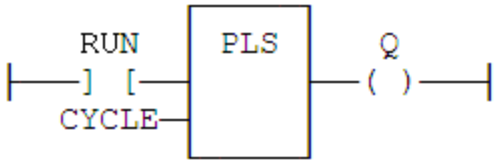
2.9.3.5 ST Language

```
(* MyPLS is a declared instance of PLS function block *)
MyPLS (RUN, CYCLE);
Q := MyPLS.Q;
```

2.9.3.6 FBD Language



2.9.3.7 FFLD Language



2.9.3.8 IL Language

```
(* MyPLS is a declared instance of PLS function block *)
Op1: CAL MyPLS (RUN, CYCLE)
   FFLD  MyPLS.Q
   ST  Q
```

See also

TON TOF TP

2.9.4 Sig_Gen

Function Block - Generator of pseudo-analogical Signal

2.9.4.1 Inputs

`RUN : BOOL` Enabling command

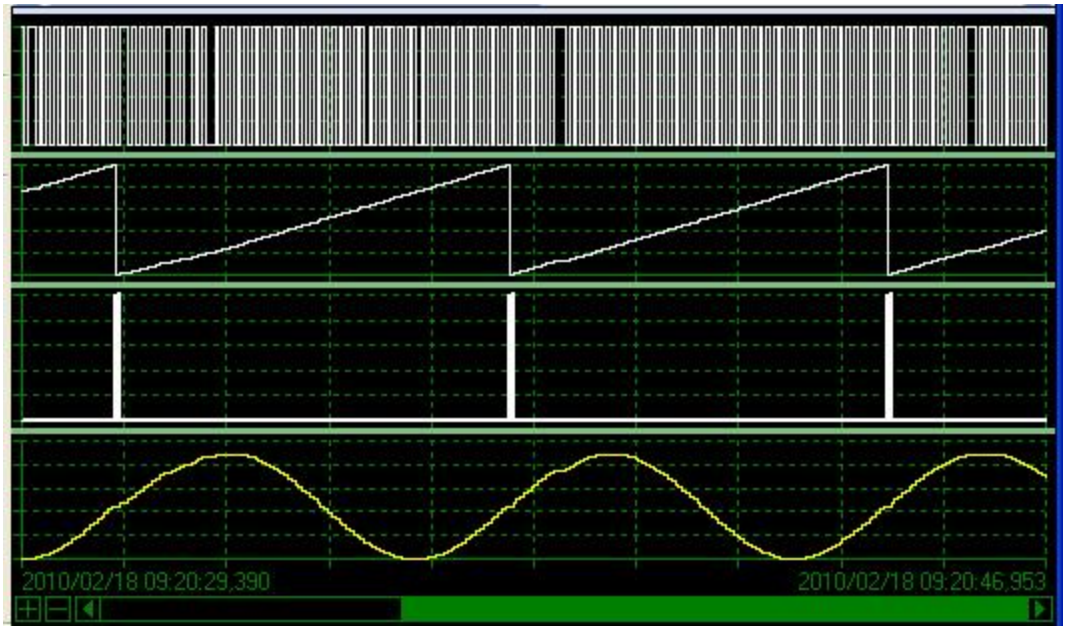
`PERIOD : TIME` Signal period

`MAXIMUM : DINT` Maximum growth during the signal period

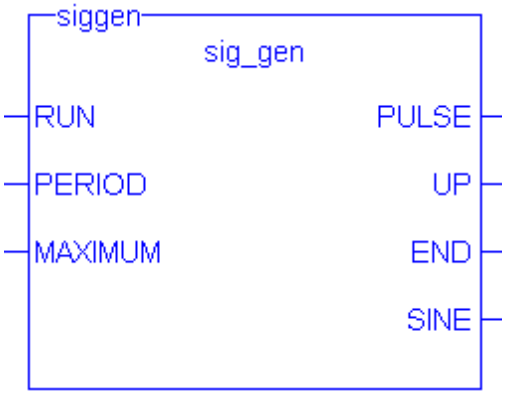
2.9.4.2 Outputs

This FB generates signals of the four following types:

- `PULSE`: blinking at each period
- `UP` : growing according max * period
- `END` : pulse after max * period
- `SINE` : sine curve



2.9.4.3 FFLD Language



2.9.5 TMD

Function Block - Down-counting stop watch.

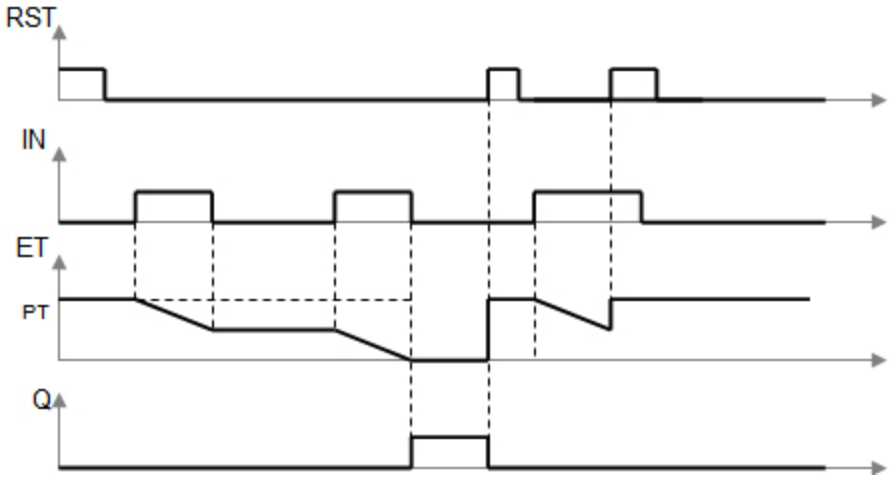
2.9.5.1 Inputs

`IN : BOOL` The time counts when this input is TRUE
`RST : BOOL` Timer is reset to PT when this input is TRUE
`PT : TIME` Programmed time

2.9.5.2 Outputs

`Q : BOOL` Timer elapsed output signal
`ET : TIME` Elapsed time

2.9.5.3 Time diagram



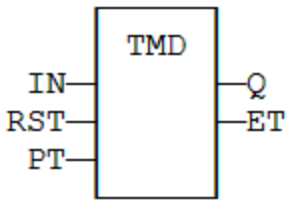
2.9.5.4 Remarks

The timer counts up when the IN input is TRUE. It stops when the programmed time is elapsed. The timer is reset when the RST input is TRUE. It is not reset when IN is false.

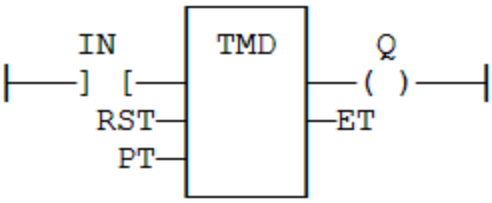
2.9.5.5 ST Language

```
(* MyTimer is a declared instance of TMD function block *)
MyTimer (IN, RST, PT);
Q := MyTimer.Q;
ET := MyTimer.ET;
```

2.9.5.6 FBD Language



2.9.5.7 FFLD Language



2.9.5.8 IL Language

```
(* MyTimer is a declared instance of TMD function block *)
Op1: CAL MyTimer (IN, RST, PT)
 FFLD: MyTimer.Q
 ST: Q
 FFLD: MyTimer.ET
 ST: ET
```

See also

TMU

2.9.6 TMU / TMUsec

Function Block - Up-counting stop watch. TMUsec is identical to TMU except that the parameter is a number of seconds.

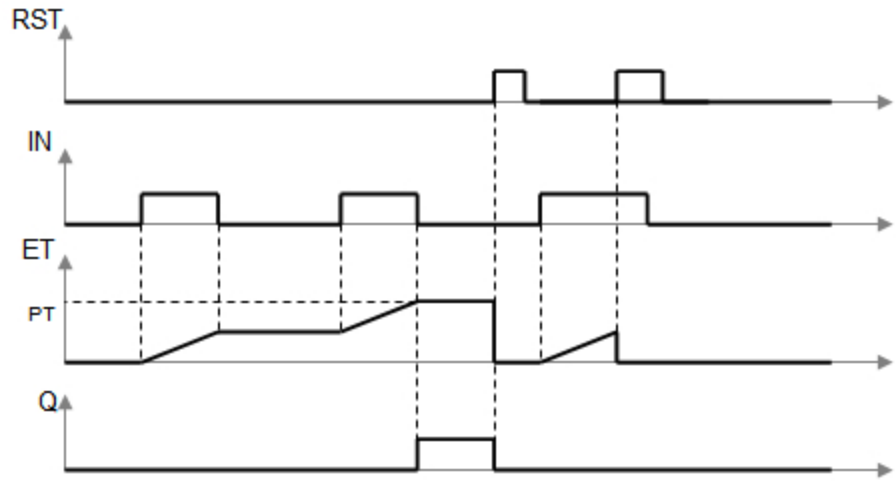
2.9.6.1 Inputs

`IN : BOOL` The time counts when this input is TRUE
`RST : BOOL` Timer is reset to 0 when this input is TRUE
`PT : TIME` Programmed time

2.9.6.2 Outputs

`Q : BOOL` Timer elapsed output signal
`ET : TIME` Elapsed time

2.9.6.3 Time diagram



2.9.6.4 Remarks

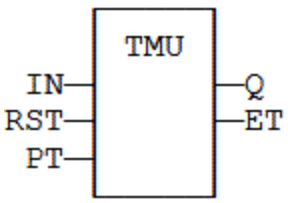
The timer counts up when the IN input is TRUE. It stops when the programmed time is elapsed. The timer is reset when the RST input is TRUE. It is not reset when IN is false.

2.9.6.5 ST Language

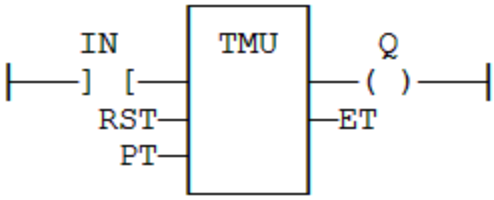
```
(* MyTimer is a declared instance of TMU function block *)

   MyTimer (IN, RST, PT);
   Q := MyTimer.Q;
   ET := MyTimer.ET;
```

2.9.6.6 FBD Language



2.9.6.7 FFLD Language



2.9.6.8 IL Language:

```
(* MyTimer is a declared instance of TMU function block *)

  Op1: CAL   MyTimer (IN, RST, PT)
       FFLD  MyTimer.Q
       ST    Q
       FFLD  MyTimer.ET
       ST    ET
```

See also

TMD

2.9.7 TOF / TOFR

Function Block - Off timer.

2.9.7.1 Inputs

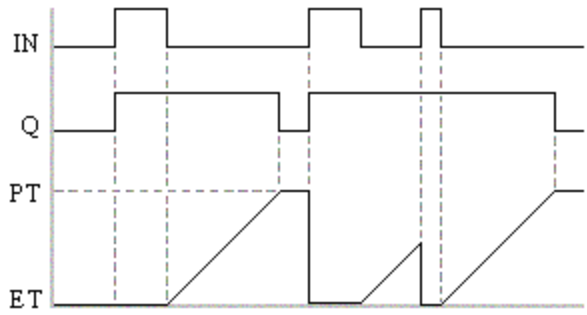
`IN : BOOL` Timer command
`PT : TIME` Programmed time

`RST : BOOL` Reset (TOFR only)

2.9.7.2 Outputs

`Q : BOOL` Timer elapsed output signal
`ET : TIME` Elapsed time

2.9.7.3 Time diagram



2.9.7.4 Remarks

The timer starts on a falling pulse of IN input. It stops when the elapsed time is equal to the programmed time. A rising pulse of IN input resets the timer to 0. The output signal is set to TRUE on when the IN input rises to TRUE, reset to FALSE when programmed time is elapsed..

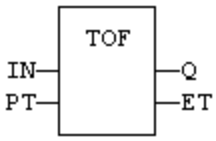
TOFR is same as TOF but has an extra input for resetting the timer

In FFLD language, the input rung is the IN command. The output rung is Q the output signal.

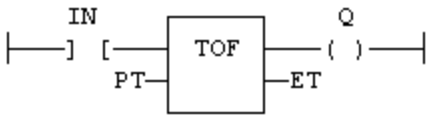
2.9.7.5 ST Language

```
(* MyTimer is a declared instance of TOF function block *)
MyTimer (IN, PT);
Q := MyTimer.Q;
ET := MyTimer.ET;
```

2.9.7.6 FBD Language



2.9.7.7 FFLD Language



2.9.7.8 IL Language:

```
(* MyTimer is a declared instance of TOF function block *)
Op1: CAL MyTimer (IN, PT)
   FFLD  MyTimer.Q
   ST  Q
   FFLD  MyTimer.ET
   ST  ET
```

See also

TON TP BLINK

2.9.8 TON

Function Block - On timer.

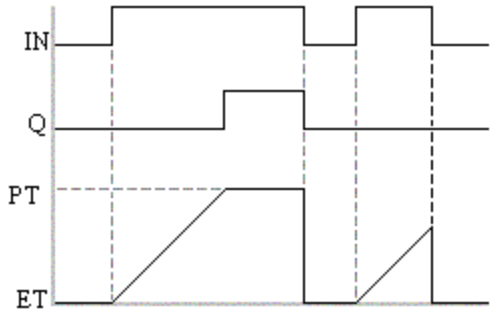
2.9.8.1 Inputs

`IN : BOOL`	Timer command
`PT : TIME`	Programmed time

2.9.8.2 Outputs

`Q : BOOL`	Timer elapsed output signal
`ET : TIME`	Elapsed time

2.9.8.3 Time diagram



2.9.8.4 Remarks

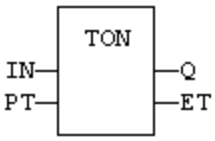
The timer starts on a rising pulse of IN input. It stops when the elapsed time is equal to the programmed time. A falling pulse of IN input resets the timer to 0. The output signal is set to TRUE when programmed time is elapsed, and reset to FALSE when the input command falls.

In FFLD language, the input rung is the IN command. The output rung is Q the output signal.

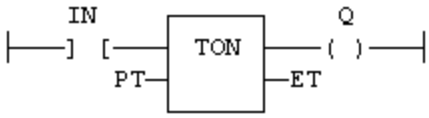
2.9.8.5 ST Language

```
(* Inst_TON is a declared instance of TON function block *)
Inst_TON( FALSE, T#2s );
Q := Inst_TON.Q;
ET := Inst_TON.ET;
```

2.9.8.6 FBD Language



2.9.8.7 FFLD Language



2.9.8.8 IL Language:

```
(* MyTimer is a declared instance of TON function block *)
Op1: CAL MyTimer (IN, PT)
    FFLD  MyTimer.Q
    ST  Q
    FFLD  MyTimer.ET
    ST  ET
```

See also

TOF TP BLINK

2.9.9 TP / TPR

Function Block - Pulse timer.

2.9.9.1 Inputs

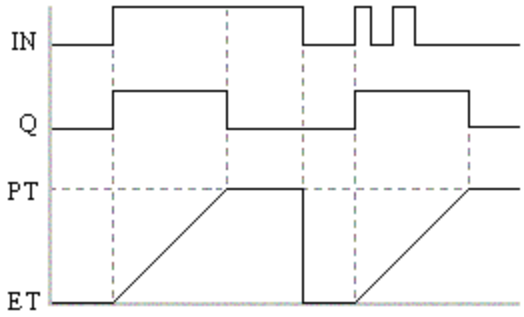
`IN : BOOL` Timer command
`PT : TIME` Programmed time

`RST : BOOL` Reset (TPR only)

2.9.9.2 Outputs

`Q : BOOL` Timer elapsed output signal
`ET : TIME` Elapsed time

2.9.9.3 Time diagram



2.9.9.4 Remarks

The timer starts on a rising pulse of IN input. It stops when the elapsed time is equal to the programmed time. A falling pulse of IN input resets the timer to 0, only if the programmed time is elapsed. All pulses of IN while the timer is running are ignored. The output signal is set to TRUE while the timer is running.

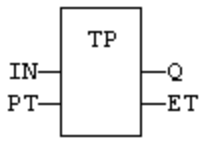
TPR is same as TP but has an extra input for resetting the timer

In FFLD language, the input rung is the IN command. The output rung is Q the output signal.

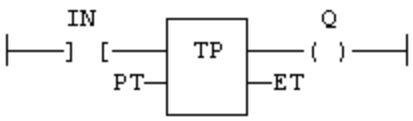
2.9.9.5 ST Language

```
(* MyTimer is a declared instance of TP function block *)
MyTimer (IN, PT);
Q := MyTimer.Q;
ET := MyTimer.ET;
```

2.9.9.6 FBD Language



2.9.9.7 FFLD Language



2.9.9.8 IL Language:

```
(* MyTimer is a declared instance of TP function block *)
Op1: CAL MyTimer (IN, PT)
```

```
FFLD  MyTimer.Q
ST  Q
FFLD  MyTimer.ET
ST  ET
```

See also

TON TOF BLINK

2.10 Mathematic operations

Below are the standard functions that perform mathematic calculation:

ABS	absolute value
TRUNC	integer part
LOG, LN / LNL	logarithm, natural logarithm
POW,EXPT, EXP / EXPL	power
SQRT, ROOT	square root, root extraction
SCALELIN	scaling - linear conversion

2.10.1 ABS / ABSL

Function - Returns the absolute value of the input.

2.10.1.1 Inputs

`IN : REAL/LREAL` ANY value

2.10.1.2 Outputs

`Q : REAL/LREAL` Result: absolute value of IN

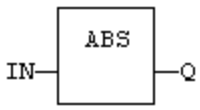
2.10.1.3 Remarks

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

2.10.1.4 ST Language

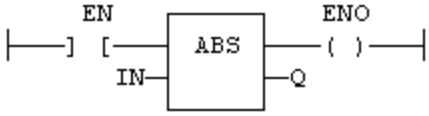
```
Q := ABS (IN);
```

2.10.1.5 FBD Language



2.10.1.6 FFLD Language

The function is executed only if EN is TRUE.
ENO keeps the same value as EN.



2.10.1.7 IL Language

```
Op1: FFLD  IN
   ABS
```

```
   ST  Q   (* Q is: ABS (IN) *)
```

See also

TRUNC LOG POW SQRT

2.10.2 EXPT

Function - Calculates a power.

2.10.2.1 Inputs

`IN : REAL`	Real value
`EXP : DINT`	Exponent

2.10.2.2 Outputs

`Q : REAL`	Result: IN at the 'EXP' power

2.10.2.3 Remarks

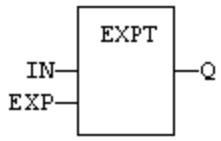
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function. The exponent (second input of the function) must be the operand of the function.

2.10.2.4 ST Language

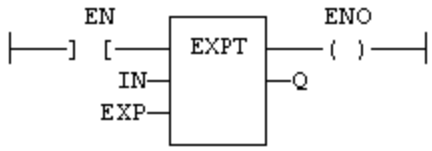
```
Q := EXPT (IN, EXP);
```

2.10.2.5 FBD Language



2.10.2.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.10.2.7 IL Language:

```
Op1: FFLD   IN
   EXPT EXP
   ST   Q   (* Q is: (IN ** EXP) *)
```

See also

ABS TRUNC LOG SQRT

2.10.3 LOG

Function - Calculates the logarithm (base 10) of the input.

2.10.3.1 Inputs

`IN : REAL` Real value

2.10.3.2 Outputs

`Q : REAL` Result: logarithm (base 10) of IN

2.10.3.3 Remarks

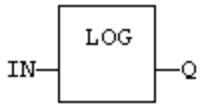
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.10.3.4 ST Language

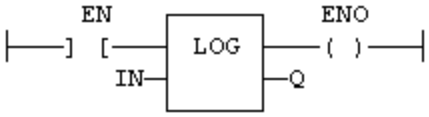
```
Q := LOG (IN);
```

2.10.3.5 FBD Language



2.10.3.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.10.3.7 IL Language:

```
Op1: FFLD  IN
     LOG
     ST  Q   (* Q is: LOG (IN) *)
```

See also

ABS TRUNC POW SQRT

2.10.4 POW ** POWL

Function - Calculates a power.

2.10.4.1 Inputs

`IN : REAL/LREAL` Real value
`EXP : REAL/LREAL` Exponent

2.10.4.2 Outputs

`Q : REAL/LREAL` Result: IN at the 'EXP' power

2.10.4.3 Remarks

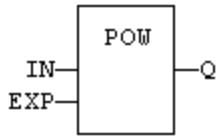
Alternatively, in ST language, the "**" operator can be used. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function. The exponent (second input of the function) must be the operand of the function.

2.10.4.4 ST Language

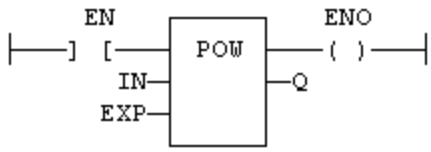
```
Q := POW (IN, EXP);
Q := IN ** EXP;
```

2.10.4.5 FBD Language



2.10.4.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.10.4.7 IL Language:

```
Op1: FFLD  IN
     POW EXP
     ST  Q   (* Q is: (IN ** EXP) *)
```

See also

ABS TRUNC LOG SQRT

2.10.5 ScaleLin

Function - Scaling - linear conversion.

2.10.5.1 Inputs

IN	:	REAL	Real value
IMIN	:	REAL	Minimum input value
IMAX	:	REAL	Maximum input value
OMIN	:	REAL	Minimum output value
OMAX	:	REAL	Maximum output value

2.10.5.2 Outputs

OUT	:	REAL	Result: OMIN + IN * (OMAX - OMIN) / (IMAX - IMIN)

2.10.5.3 Truth table

inputs	OUT
IMIN >= IMAX	= IN
IN < IMIN	= IMIN
IN > IMAX	= IMAX
other	= OMIN + IN * (OMAX - OMIN) / (IMAX - IMIN)

2.10.5.4 Remarks

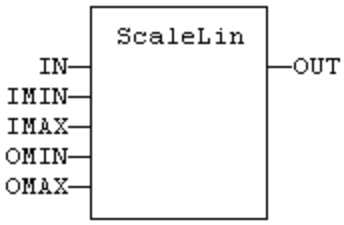
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.10.5.5 ST Language

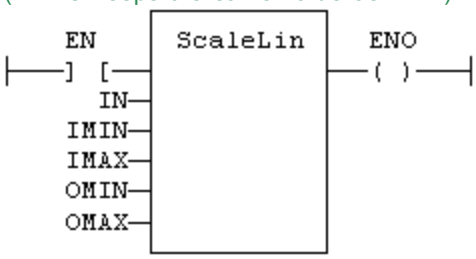
```
OUT := ScaleLin (IN, IMIN, IMAX, OMIN, OMAX);
```

2.10.5.6 FBD Language



2.10.5.7 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.10.5.8 IL Language

```
Op1: FFLD      IN
     ScaleLin IMAX, IMIN, OMAX, OMIN
     ST       OUT
```

2.10.6 SQRT / SQRTL

Function - Calculates the square root of the input.

2.10.6.1 Inputs

`IN : REAL/LREAL` Real value

2.10.6.2 Outputs

`Q : REAL/LREAL` Result: square root of IN

2.10.6.3 Remarks

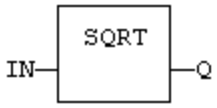
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.10.6.4 ST Language

```
Q := SQRT (IN);
```

2.10.6.5 FBD Language



2.10.6.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```

2.10.6.7 IL Language:

```
Op1: FFLD  IN
     SQRT
     ST  Q   (* Q is: SQRT (IN) *)
```

See also

ABS TRUNC LOG POW

2.10.7 TRUNC / TRUNCL

Function - Truncates the decimal part of the input.

2.10.7.1 Inputs

`IN : REAL/LREAL` Real value

2.10.7.2 Outputs

`Q : REAL/LREAL` Result: integer part of IN

2.10.7.3 Remarks

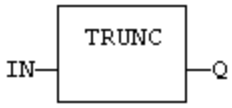
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.10.7.4 ST Language

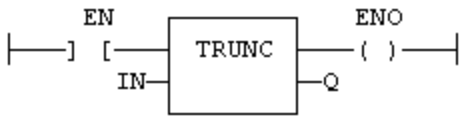
```
Q := TRUNC (IN);
```

2.10.7.5 FBD Language



2.10.7.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.10.7.7 IL Language:

```
Op1: FFLD  IN
     TRUNC
     ST  Q   (* Q is the integer part of IN *)
```

See also

ABS LOG POW SQRT

2.11 Trigonometric functions

Below are the standard functions for trigonometric calculation:

SIN	sine
COS	cosine
TAN	tangent
ASIN	arc-sine
ACOS	arc-cosine
ATAN	arc-tangent
ATAN2	arc-tangent of Y / X

See Also:

UseDegrees

2.11.1 ACOS / ACOSL

Function - Calculate an arc-cosine.

2.11.1.1 Inputs

`IN : REAL/LREAL` Real value

2.11.1.2 Outputs

`Q : REAL/LREAL` Result: arc-cosine of IN

2.11.1.3 Remarks

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

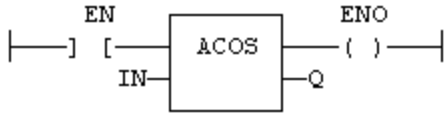
2.11.1.4 ST Language

```
Q := ACOS (IN);
```

2.11.1.5 FBD Language

2.11.1.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.11.1.7 IL Language:

```
Op1: FFLD  IN
     ACOS
     ST  Q   (* Q is: ACOS (IN) *)
```

See also

SIN COS TAN ASIN ATAN ATAN2

2.11.2 ASIN / ASINL

Function - Calculate an arc-sine.

2.11.2.1 Inputs

`IN : REAL/LREAL` Real value

2.11.2.2 Outputs

`Q : REAL/LREAL` Result: arc-sine of IN

2.11.2.3 Remarks

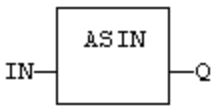
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.11.2.4 ST Language

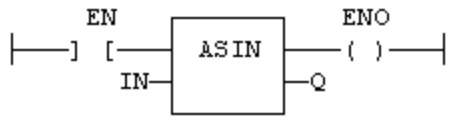
```
Q := ASIN (IN);
```

2.11.2.5 FBD Language



2.11.2.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.11.2.7 IL Language:

```
Op1: FFLD  IN
     ASIN
```

```
   ST  Q   (* Q is: ASIN (IN) *)
```

See also

SIN COS TAN ACOS ATAN ATAN2

2.11.3 ATAN / ATANL

Function - Calculate an arc-tangent.

2.11.3.1 Inputs

`IN : REAL/LREAL` Real value

2.11.3.2 Outputs

`Q : REAL/LREAL` Result: arc-tangent of IN

2.11.3.3 Remarks

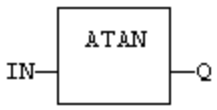
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.11.3.4 ST Language

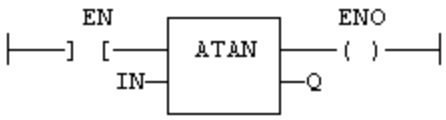
```
Q := ATAN (IN);
```

2.11.3.5 FBD Language



2.11.3.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.11.3.7 IL Language:

```
Op1: FFLD  IN
   ATAN
   ST  Q   (* Q is: ATAN (IN) *)
```

See also

SIN COS TAN ASIN ACOS ATAN2

2.11.4 ATAN2 / ATAN2L

Function - Calculate arc-tangent of Y/X

2.11.4.1 Inputs

`Y : REAL/LREAL` Real value
`X : REAL/LREAL` Real value

2.11.4.2 Outputs

`Q : REAL/LREAL` Result: arc-tangent of Y / X

2.11.4.3 Remarks

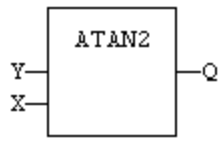
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.11.4.4 ST Language

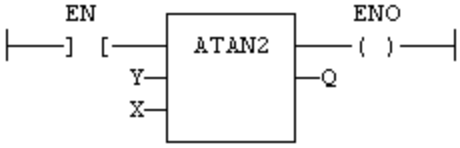
```
Q := ATAN2 (IN);
```

2.11.4.5 FBD Language



2.11.4.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.11.4.7 IL Language

```
Op1: FFLD  Y
     ATAN2 X
     ST  Q   (* Q is: ATAN2 (Y / X) *)
```

See also

SIN COS TAN ASIN ACOS ATAN

2.11.5 COS / COSL

Function - Calculate a cosine.

2.11.5.1 Inputs

`IN : REAL/LREAL` Real value

2.11.5.2 Outputs

`Q : REAL/LREAL` Result: cosine of IN

2.11.5.3 Remarks

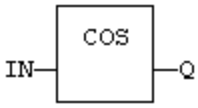
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.11.5.4 ST Language

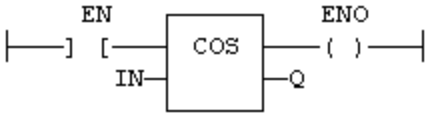
```
Q := COS (IN);
```

2.11.5.5 FBD Language



2.11.5.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.11.5.7 IL Language:

```
Op1: FFLD  IN
     COS
     ST  Q   (* Q is: COS (IN) *)
```

See also

SIN TAN ASIN ACOS ATAN ATAN2

2.11.6 SIN / SINL

Function - Calculate a sine.

2.11.6.1 Inputs

`IN : REAL/LREAL` Real value

2.11.6.2 Outputs

`Q : REAL/LREAL` Result: sine of IN

2.11.6.3 Remarks

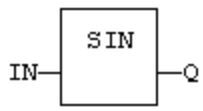
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.11.6.4 ST Language

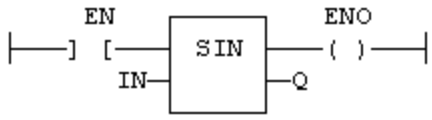
```
Q := SIN (IN);
```

2.11.6.5 FBD Language



2.11.6.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.11.6.7 IL Language:

```
Op1: FFLD  IN
     SIN
     ST  Q   (* Q is: SIN (IN) *)
```

See also

COS TAN ASIN ACOS ATAN ATAN2

2.11.7 TAN / TANL

Function - Calculate a tangent.

2.11.7.1 Inputs

`IN : REAL/LREAL` Real value

2.11.7.2 Outputs

`Q : REAL/LREAL` Result: tangent of IN

2.11.7.3 Remarks

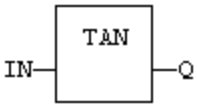
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.11.7.4 ST Language

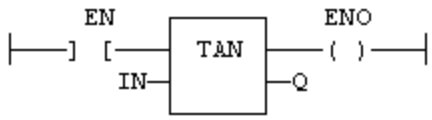
```
Q := TAN (IN);
```

2.11.7.5 FBD Language



2.11.7.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.11.7.7 IL Language:

```
Op1: FFLD  IN
     TAN
```

```
    ST  Q   (* Q is: TAN (IN) *)
```

See also

SIN COS ASIN ACOS ATAN ATAN2

2.11.8 UseDegrees

Function - Sets the unit for angles in all trigonometric functions.

2.11.8.1 Inputs

`IN : BOOL` If TRUE, turn all trigonometric functions to use degrees
If FALSE, turn all trigonometric functions to use radians (default)

2.11.8.2 Outputs

`Q : BOOL` TRUE if functions use degrees before the call

2.11.8.3 Remarks

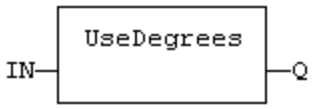
This function sets the working unit for the following functions:

SIN	sine
COS	cosine
TAN	tangent
ASIN	arc-sine
ACOS	arc-cosine
ATAN	arc-tangent
ATAN2	arc-tangent of Y / X

2.11.8.4 ST Language

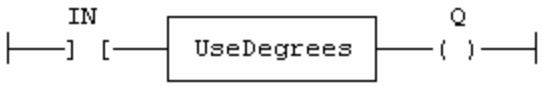
```
Q := UseDegrees (IN);
```

2.11.8.5 FBD Language



2.11.8.6 FFLD Language

```
(* Input is the rung. The rung is the output *)
```



2.11.8.7 IL Language

```
Op1: FFLD   IN
     UseDegrees
     ST   Q
```

2.12 String operations

Below are the standard operators and functions that manage character strings:

+	concatenation of strings
CONCAT	concatenation of strings
MLEN	get string length
DELETE	delete characters in a string
INSERT	insert characters in a string
FIND	find characters in a string
REPLACE	replace characters in a string
LEFT	extract a part of a string on the left
RIGHT	extract a part of a string on the right
MID	extract a part of a string
CHAR	build a single character string
ASCII	get the ASCII code of a character within a string
ATOH	converts string to integer using hexadecimal basis
HTOA	converts integer to string using hexadecimal basis
CRC16	CRC16 calculation
ArrayToString	copies elements of an SINT array to a STRING
StringToArray	copies characters of a STRING to an SINT array

Other functions are available for managing string tables as resources:

StringTable	Select the active string table resource
LoadString	Load a string from the active string table

2.12.1 ArrayToString / ArrayToStringU

Function - Copy an array of SINT to a STRING.

2.12.1.1 Inputs

`SRC : SINT` Source array of SINT small integers (USINT for ArrayToStringU)
`DST : STRING` Destination STRING
`COUNT : DINT` Numbers of characters to be copied

2.12.1.2 Outputs

`Q : DINT` Number of characters copied

2.12.1.3 Remarks

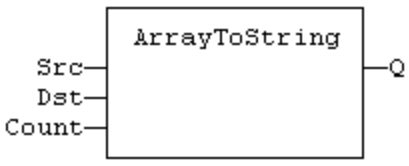
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

This function copies the COUNT first elements of the SRC array to the characters of the DST string. The function checks the maximum size of the destination string and adjust the COUNT number if necessary.

2.12.1.4 ST Language

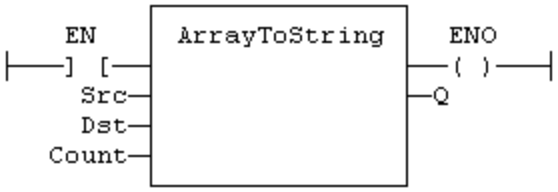
```
Q := ArrayToString (SRC, DST, COUNT);
```

2.12.1.5 FBD Language



2.12.1.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.12.1.7 IL Language

Not available

See also

StringToArray

2.12.2 ASCII

Function - Get the ASCII code of a character within a string

2.12.2.1 Inputs

`IN` : `STRING` Input string
`POS` : `DINT` Position of the character within the string
(The first valid position is 1)

2.12.2.2 Outputs

`CODE` : `DINT` ASCII code of the selected character
or 0 if position is invalid

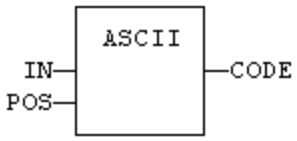
2.12.2.3 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operand of the function.

2.12.2.4 ST Language

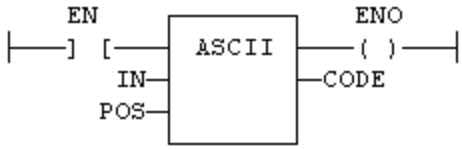
```
CODE := ASCII (IN, POS);
```

2.12.2.5 FBD Language



2.12.2.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```



2.12.2.7 IL Language:

```
Op1: FFLD      IN
   AND_MASK MSK
   ST      CODE
```

See also

CHAR

2.12.3 ATOH

Function - Converts string to integer using hexadecimal basis

2.12.3.1 Inputs

`IN : STRING` String representing an integer in hexadecimal format

2.12.3.2 Outputs

`Q : DINT` Integer represented by the string

2.12.3.3 Truth table (examples)

IN	Q
''	0
'12'	18
'a0'	160
'A0zzz'	160

2.12.3.4 Remarks

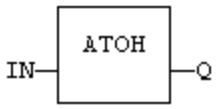
The function is case insensitive. The result is 0 for an empty string. The conversion stops before the first invalid character. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.12.3.5 ST Language

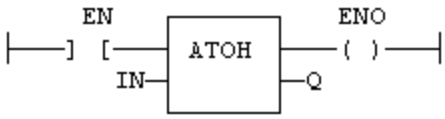
```
Q := ATOH (IN);
```

2.12.3.6 FBD Language



2.12.3.7 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.12.3.8 IL Language:

```
Op1: FFLD  IN
   ATOH
   ST  Q
```

See also

HTOA

2.12.4 CHAR

Function - Builds a single character string

2.12.4.1 Inputs

`CODE : DINT` ASCII code of the wished character

2.12.4.2 Outputs

`Q : STRING` STRING containing only the specified character

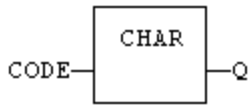
2.12.4.3 Remarks

In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the input parameter (CODE) must be loaded in the current result before calling the function.

2.12.4.4 ST Language

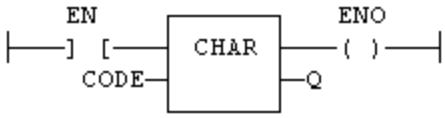
```
Q := CHAR (CODE);
```

2.12.4.5 FBD Language



2.12.4.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```



2.12.4.7 IL Language:

```
Op1: FFLD  CODE
   CHAR
   ST  Q
```

See also

ASCII

2.12.5 CONCAT

Function - Concatenate strings.

2.12.5.1 Inputs

`IN_1 : STRING` Any string variable or constant expression
...
`IN_N : STRING` Any string variable or constant expression

2.12.5.2 Outputs

`Q : STRING` Concatenation of all inputs

2.12.5.3 Remarks

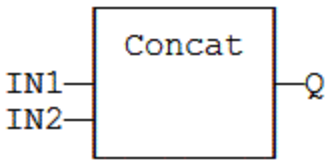
In FBD or FFLD language, the block can have up to 16 inputs. In IL or ST, the function accepts a variable number of inputs (at least 2).

Note that you also can use the "**+**" operator to concatenate strings.

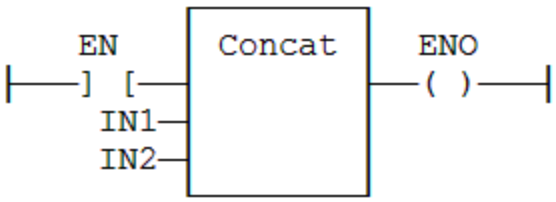
2.12.5.4 ST Language

```
Q := CONCAT ('AB', 'CD', 'E');
(* now Q is 'ABCDE' *)
```

2.12.5.5 FBD Language



2.12.5.6 FFLD Language



2.12.5.7 IL Language

```
Op1: FFLD    'AB'
   CONCAT 'CD', 'E'
   ST  Q   (* Q is now 'ABCDE' *)
```

2.12.6 CRC16

Function - calculates a CRC16 on the characters of a string

2.12.6.1 Inputs

`IN : STRING` character string

2.12.6.2 Outputs

`Q : INT` CRC16 calculated on all the characters of the string.

2.12.6.3 Remarks

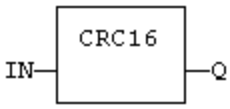
In FFLD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the input parameter (IN) must be loaded in the current result before calling the function.

The function calculates a MODBUS CRC16, initialized at 16#FFFF value.

2.12.6.4 ST Language

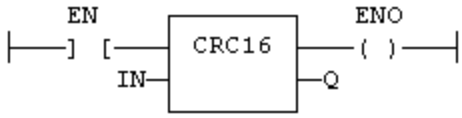
```
Q := CRC16 (IN);
```

2.12.6.5 FBD Language



2.12.6.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO is equal to EN *)
```



2.12.6.7 IL Language:

```
Op1: FFLD   IN
     CRC16
     ST   Q
```

2.12.7 DELETE

Function - Delete characters in a string.

2.12.7.1 Inputs

`IN : STRING`	Character string
`NBC : DINT`	Number of characters to be deleted
`POS : DINT`	Position of the first deleted character (first character position is 1)

2.12.7.2 Outputs

`Q : STRING`	Modified string.

2.12.7.3 Remarks

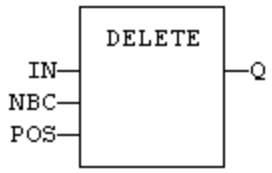
The first valid character position is 1. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by comas.

2.12.7.4 ST Language

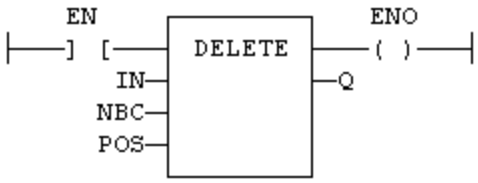
```
Q := DELETE (IN, NBC, POS);
```

2.12.7.5 FBD Language



2.12.7.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.12.7.7 IL Language:

```
Op1: FFLD    IN
   DELETE NBC, POS
   ST    Q
```

See also

\+ MLEN INSERT FIND REPLACE LEFT RIGHT MID

2.12.8 FIND

Function - Find position of characters in a string.

2.12.8.1 Inputs

`IN : STRING` Character string
`STR : STRING` String containing searched characters

2.12.8.2 Outputs

`POS : DINT` Position of the first character of STR in IN, or 0 if not found

2.12.8.3 Remarks

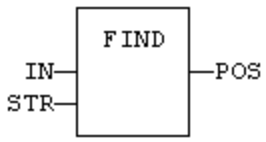
The first valid character position is 1. A return value of 0 means that the STR string has not been found. Search is case sensitive. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.

2.12.8.4 ST Language

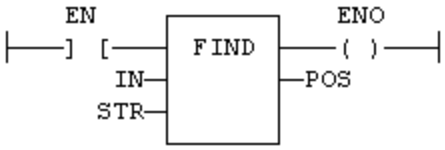
```
POS := FIND (IN, STR);
```

2.12.8.5 FBD Language



2.12.8.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.12.8.7 IL Language:

```
Op1: FFLD    IN
   FIND   STR
   ST    POS
```

See also

\+ MLEN DELETE INSERT REPLACE LEFT RIGHT MID

2.12.9 HTOA

Function - Converts integer to string using hexadecimal basis

2.12.9.1 Inputs

`IN : DINT` Integer value

2.12.9.2 Outputs

`Q : STRING` String representing the integer in hexadecimal format

2.12.9.3 Truth table (examples)

IN	Q
0	'0'
18	'12'
160	'A0'

2.12.9.4 Remarks

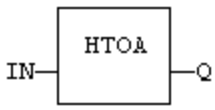
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.12.9.5 ST Language

```
Q := HTOA (IN);
```

2.12.9.6 FBD Language



2.12.9.7 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```

2.12.9.8 IL Language:

```
Op1: FFLD  IN
     HTOA
     ST  Q
```

See also

ATOH

2.12.10 INSERT

Function - Insert characters in a string.

2.12.10.1 Inputs

`IN : STRING`	Character string
`STR : STRING`	String containing characters to be inserted
`POS : DINT`	Position of the first inserted character (first character position is 1)

2.12.10.2 Outputs

`Q : STRING`	Modified string.

2.12.10.3 Remarks

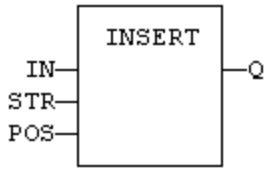
The first valid character position is 1. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by comas.

2.12.10.4 ST Language

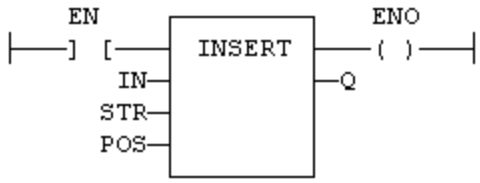
```
Q := INSERT (IN, STR, POS);
```

2.12.10.5 FBD Language



2.12.10.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.12.10.7 IL Language:

```
Op1: FFLD    IN
   INSERT STR, POS
   ST    Q
```

See also

\+ MLEN DELETE FIND REPLACE LEFT RIGHT MID

2.12.11 LEFT

Function - Extract characters of a string on the left.

2.12.11.1 Inputs

`IN : STRING` Character string
`NBC : DINT` Number of characters to extract

2.12.11.2 Outputs

`Q : STRING` String containing the first NBC characters of IN.

2.12.11.3 Remarks

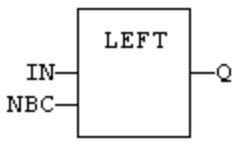
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.

2.12.11.4 ST Language

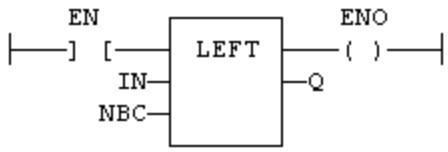
```
Q := LEFT (IN, NBC);
```

2.12.11.5 FBD Language



2.12.11.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.12.11.7 IL Language:

```
Op1: FFLD    IN
   LEFT  NBC
   ST    Q
```

See also

\+ MLEN DELETE INSERT FIND REPLACE RIGHT MID

2.12.12 LoadString

Function - Load a string from the active string table.

2.12.12.1 Inputs

`ID: DINT` ID of the string as declared in the string table

2.12.12.2 Outputs

`Q : STRING` Loaded string or empty string in case of error

2.12.12.3 Remarks

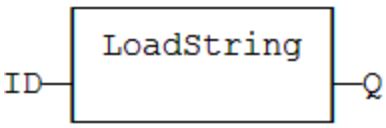
This function loads a string from the active string table and stores it into a STRING variable. The StringTable() function is used for selecting the active string table.

The "ID" input (the string item identifier) is an identifier such as declared within the string table resource. You don't need to "define" again this identifier. The system does it for you.

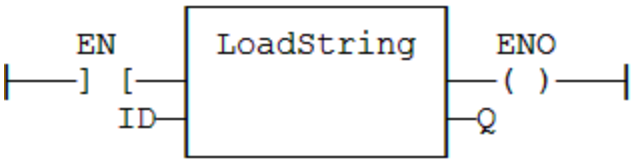
2.12.12.4 ST Language

```
Q := LoadString (ID);
```

2.12.12.5 FBD Language



2.12.12.6 FFLD Language



2.12.12.7 IL Language:

```
Op1: FFLD    ID
   LoadString
   ST    Q
```

See also

StringTable String tables

2.12.13 MID

Function - Extract characters of a string at any position.

2.12.13.1 Inputs

`IN : STRING` Character string
`NBC : DINT` Number of characters to extract
`POS : DINT` Position of the first character to extract (first character of IN is at position 1)

2.12.13.2 Outputs

`Q : STRING` String containing the first NBC characters of IN.

2.12.13.3 Remarks

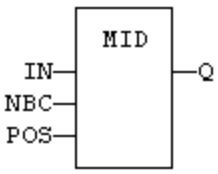
The first valid position is 1. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by comas.

2.12.13.4 ST Language

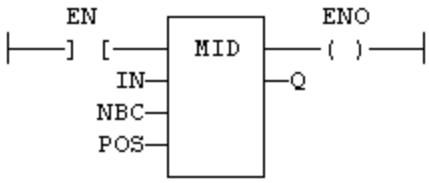
```
Q := MID (IN, NBC, POS);
```

2.12.13.5 FBD Language



2.12.13.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.12.13.7 IL Language:

```
Op1: FFLD   IN
     MID  NBC, POS
     ST   Q
```

See also

+ MLEN DELETE INSERT FIND REPLACE LEFT RIGHT

2.12.14 MLEN

Function - Get the number of characters in a string.

2.12.14.1 Inputs

`IN : STRING` Character string

2.12.14.2 Outputs

`NBC : DINT` Number of characters currently in the string. 0 if string is empty.

2.12.14.3 Remarks

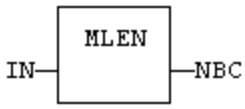
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

2.12.14.4 ST Language

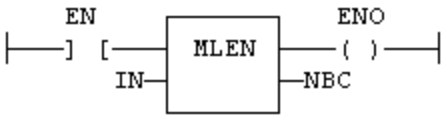
```
NBC := MLEN (IN);
```

2.12.14.5 FBD Language



2.12.14.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.12.14.7 IL Language:

```
Op1: FFLD   IN
     MLEN
     ST   NBC
```

See also

\+ DELETE INSERT FIND REPLACE LEFT RIGHT MID

2.12.15 REPLACE

Function - Replace characters in a string.

2.12.15.1 Inputs

`IN : STRING` Character string
`STR : STRING` String containing the characters to be inserted in place of NDEL removed characters
`NDEL : DINT` Number of characters to be deleted before insertion of STR
`POS : DINT` Position where characters are replaced (first character position is 1)

2.12.15.2 Outputs

`Q : STRING` Modified string.

2.12.15.3 Remarks

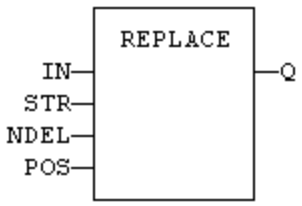
The first valid character position is 1. In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by comas.

2.12.15.4 ST Language

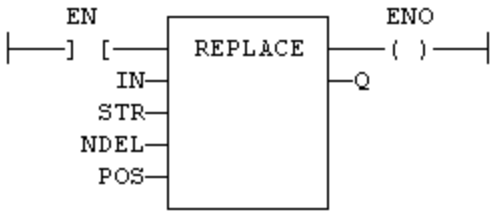
```
Q := REPLACE (IN, STR, NDEL, POS);
```

2.12.15.5 FBD Language



2.12.15.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.12.15.7 IL Language:

```
Op1: FFLD    IN
   REPLACE STR, NDEL, POS
   ST    Q
```

See also

\+ MLEN DELETE INSERT FIND LEFT RIGHT MID

2.12.16 RIGHT

Function - Extract characters of a string on the right.

2.12.16.1 Inputs

`IN : STRING` Character string
`NBC : DINT` Number of characters to extract

2.12.16.2 Outputs

`Q : STRING` String containing the last NBC characters of IN.

2.12.16.3 Remarks

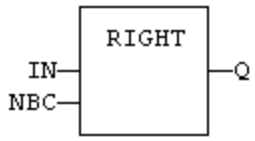
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.

2.12.16.4 ST Language

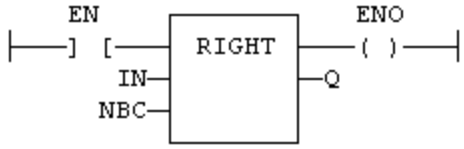
```
Q := RIGHT (IN, NBC);
```

2.12.16.5 FBD Language



2.12.16.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



2.12.16.7 IL Language:

```
Op1: FFLD    IN
   RIGHT  NBC
   ST    Q
```

See also

\+ MLEN DELETE INSERT FIND REPLACE LEFT MID

2.12.17 StringTable

Function - Selects the active string table.

2.12.17.1 Inputs

`TABLE : STRING`	Name of the Sting Table resource -	*must be a constant*
`COL : STRING`	Name of the column in the table -	*must be a constant*

2.12.17.2 Outputs

`OK : BOOL` TRUE if OK

2.12.17.3 Remarks

This function selects a column of a valid String Table resource to become the active string table. The LoadString() function always refers to the active string table.

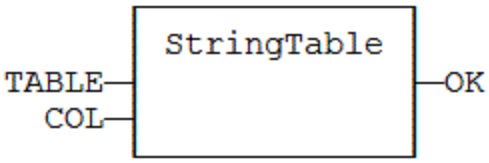
Arguments must be constant string expressions and must fit to a declared string table and a valid column name within this table.

If you have only one string table with only one column defined in your project, you do not need to call this function as it will be the default string table anyway.

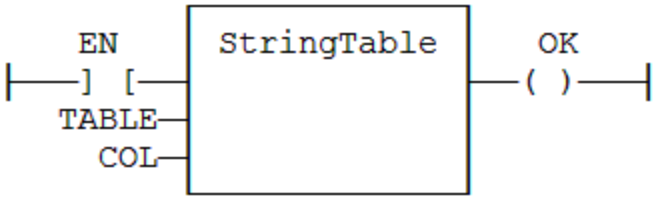
2.12.17.4 ST Language

```
OK := StringTable ('MyTable', 'FirstColumn");
```

2.12.17.5 FBD Language



2.12.17.6 FFLD Language



2.12.17.7 IL Language:

```
Op1: FFLD       'MyTable'
StringTable 'First Column'
ST       OK
```

See also

LoadString String tables

2.12.18 StringToArray / StringToArrayU

Function - Copies the characters of a STRING to an array of SINT.

2.12.18.1 Inputs

`SRC : STRING` Source STRING
`DST : SINT` Destination array of SINT small integers (USINT for StringToArrayU)

2.12.18.2 Outputs

`Q : DINT` Number of characters copied

2.12.18.3 Remarks

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

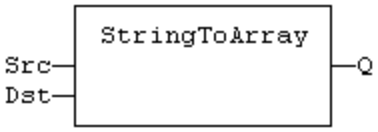
In IL, the input must be loaded in the current result before calling the function.

This function copies the characters of the SRC string to the first characters of the DST array. The function checks the maximum size destination arrays and reduces the number of copied characters if necessary.

2.12.18.4 ST Language

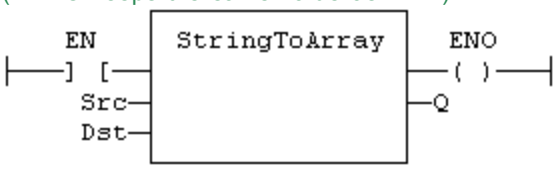
```
Q := StringToArray (SRC, DST);
```

2.12.18.5 FBD Language



2.12.18.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



2.12.18.7 IL Language:

Op1: FFLD SRC
StringToArray DST
ST Q

See also

ArrayToString

3 Advanced operations

Below are the standard blocks that perform advanced operations.

Analog signal processing:

Average / AverageL	Calculate the average of signal samples
Integral	Calculate the integral of a signal
Derivate	Derivate a signal
PID	PID loop
Ramp	Ramp signal
Rand	Give a Random value modulo the input value
Lim_Alrm	Low / High level detection
Hyster	Hysterisis calculation
SigPlay	Play an analog signal from a resource
SigScale	Get a point from a signal resource
CurveLin	Linear interpolation on a curve
SurfLin	Linear interpolation on a surface

Alarm management:

Lim_Alrm	Low / High level detection
Alarm_M	Alarm with manual reset
Alarm_A	Alarm with automatic reset

Data collections and serialization:

StackInt	Stack of integers
FIFO	"First in / first out" list
LIFO	"Last in / first out" stack
SerializeIn	Extract data from a binary frame
SerializeOut	Write data to a binary frame
SerGetString	Extract a string from a binary frame
SerPutString	Copies a string to a binary frame

Data Logging:

LogFileCSV	Log values of variables to a CSV file

Special operations:

GetSysInfo	Get system information
Printf	Trace messages
CycleStop	Sets the application in cycle stepping mode
FatalStop	Breaks the cycle and stop with fatal error
EnableEvents	Enable / disable produced events for binding
ApplyRecipeColumn	Apply the values of a column from a recipe file
VLID	Get the ID of an embedded list of variables
SigID	Get the ID of a signal resource

Communication:

SERIO: serial communication
AS-interface
TCP-IP management functions
UDP management functions
MQTT protocol handling
MBSlaveRTU MBSlaveUDP

Others:

Dynamic memory allocation functions
Real Time Clock
Variable size text buffers manipulation
XML writing and parsing

3.1 ALARM_A

Function Block - Alarm with automatic reset

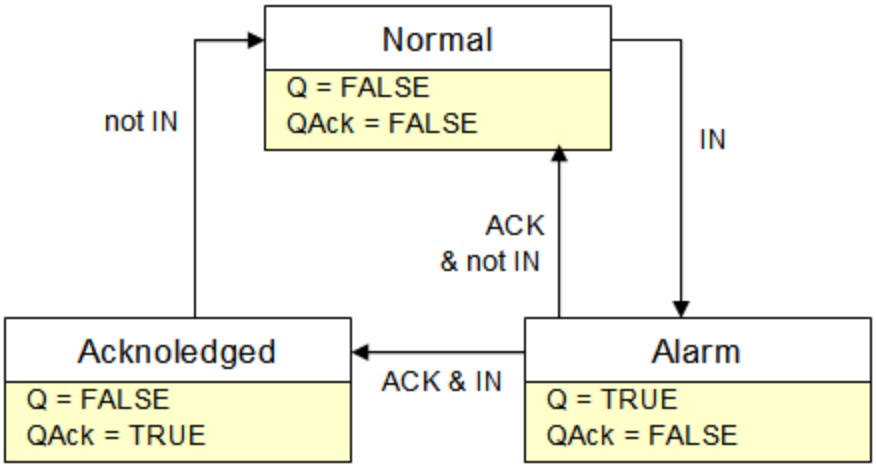
3.1.1 Inputs

IN	:	BOOL	Process signal
ACK	:	BOOL	Acknowledge command

3.1.2 Outputs

Q	:	BOOL	TRUE if alarm is active
QACK	:	BOOL	TRUE if alarm is acknowledged

3.1.3 Sequence



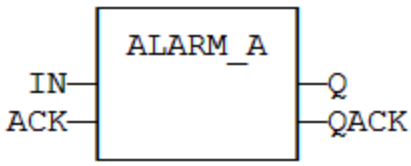
3.1.4 Remarks

Combine this block with the LIM_ALRM block for managing analog alarms.

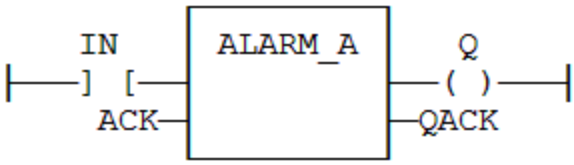
3.1.5 ST Language

```
(* MyALARM is declared as an instance of ALARM_A function block *)
MyALARM (IN, ACK);
Q := MyALARM.Q;
QACK := MyALARM.QACK;
```

3.1.6 FBD Language



3.1.7 FFLD Language



3.1.8 IL Language

```
(* MyALARM is declared as an instance of ALARM_A function block *)
Op1: CAL MyALARM (IN, ACK)
FFLD MyALARM.Q
ST Q
FFLD MyALARM.QACK
ST QACK
```

See also

ALARM_M LIM_ALRM

3.2 ALARM_M

Function Block - Alarm with manual reset

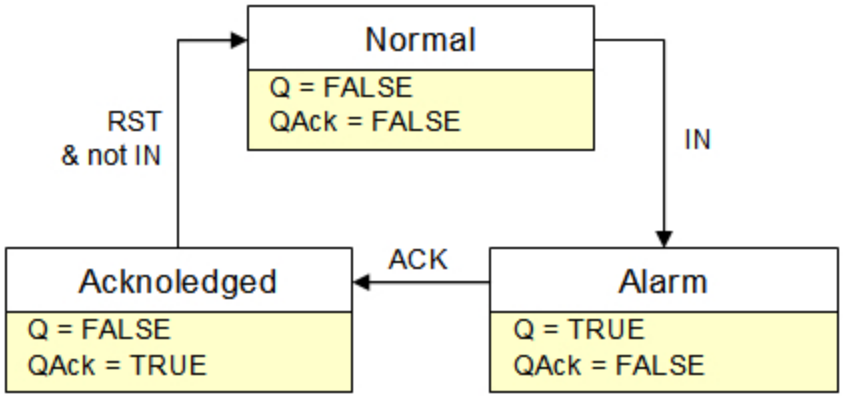
3.2.1 Inputs

IN	:	BOOL	Process signal
ACK	:	BOOL	Acknowledge command
RST	:	BOOL	Reset command

3.2.2 Outputs

Q	:	BOOL	TRUE if alarm is active
QACK	:	BOOL	TRUE if alarm is acknowledged

3.2.3 Sequence



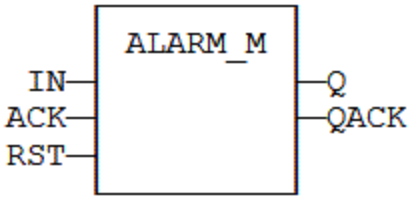
3.2.4 Remarks

Combine this block with the LIM_ALRM block for managing analog alarms.

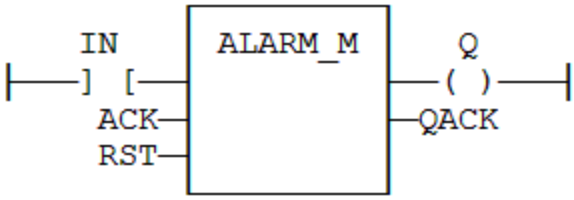
3.2.5 ST Language

```
(* MyALARM is declared as an instance of ALARM_M function block *)
MyALARM (IN, ACK, RST);
Q := MyALARM.Q;
QACK := MyALARM.QACK;
```

3.2.6 FBD Language



3.2.7 FFLD Language



3.2.8 IL Language

```
(* MyALARM is declared as an instance of ALARM_M function block *)
Op1: CAL MyALARM (IN, ACK, RST)
FFLD MyALARM.Q
ST Q
FFLD MyALARM.QACK
ST QACK
```

See also

ALARM_A LIM_ALRM

3.3 ApplyRecipeColumn

Function - Apply the values of a column from a recipe file

3.3.1 Inputs

`FILE : STRING`	Path name of the recipe file (.RCP or .CSV) - *must be a constant value!*
`COL : DINT`	Index of the column in the recipe (0 based)

See an example of RCP file

```
@COLNAME=Col3 Col4

@SIZECOL1=100

@SIZECOL2=100

@SIZECOL3=100

@SIZECOL4=100
```

```
bCommand
tPerio
bFast
Blink1
test_var
bOut
@EXPANDED=Blink1
```

See an example of CSV file

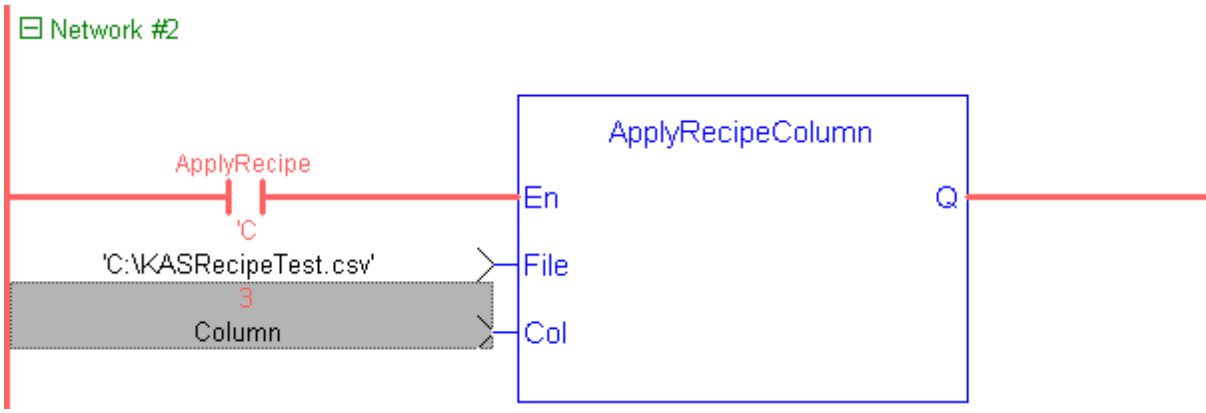
Example of CSV file with five variables and five set of values

```
comment lines here
TravelSpeed;100;200;300;400;500
MasterAbsPos;0;45;90;135;180
MasterDeltaPos;0;90;180;270;360
MachineSpeed;50;100;150;200;250
MachineState;0;0;1;1;2
```

Note

For your CSV file to be valid, ensure the data are separated with **semicolons** (and not commas).

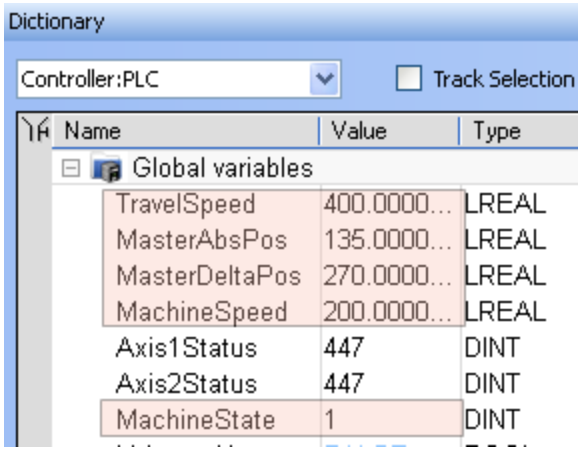
Usage in a FFLD program where column 3 is selected



Column 3 corresponds to column E in the Excel sheet because this parameter is 0 based

	A	B	C	D	E	F
1	comment lines here					
2	TravelSpeed	100	200	300	400	500
3	MasterAbsPos	0	45	90	135	180
4	MasterDeltaPos	0	90	180	270	360
5	MachineSpeed	50	100	150	200	250
6	MachineState	0	0	1	1	2

Result displayed in the Dictionary when the application is running



3.3.2 Outputs

`OK : BOOL` TRUE if OK - FALSE if parameters are invalid

3.3.3 Remarks

The 'FILE' input is a constant string expression specifying the path name of a valid .RCP or .CSV file. If no path is specified, the file is assumed to be located in the project folder. RCP files are created using an external recipe editor. CSV files can be created using EXCEL or NOTEPAD.

In CSV files, the first line must contain column headers, and is ignored during compiling. There is one variable per line. The first column contains the symbol of the variable. Other columns are values.

If a cell is empty, it is assumed to be the same value as the previous (left side) cell. If it is the first cell of a row, it is assumed to be null (0 or FALSE or empty string).

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung is the result of the function.

Warning

Recipe files are read at compiling time and are embedded into the downloaded application code. This implies that a modification performed in the recipe file after downloading is not taken into account by the application.

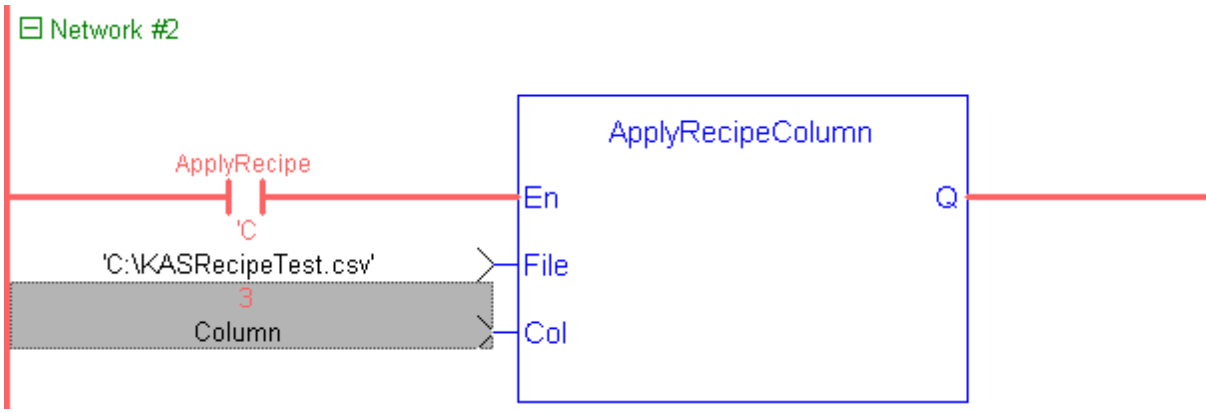
3.3.4 ST Language

```
OK := ApplyRecipeColumn ('MyFile.rcp', COL);
```

3.3.5 FBD Language

3.3.6 FFLD Language

```
(* The function is executed only if ApplyRecipe is TRUE *)
```



3.3.7 IL Language

```
Op1: FFLD          'MyFile.rcp'
ApplyRecipeColumn COL
ST          OK
```

3.4 AS-interface functions

The following functions enable special operation on AS-i networks:

`ASiReadPP`	read permanent parameters of an AS-i slave
`ASiWritePP`	write permanent parameters of an AS-i slave
`ASiSendParam`	send parameters to an AS-i slave
`ASiReadPI`	read actual parameters of an AS-i slave
`ASiStorePI`	store actual parameters as permanent parameters

Warning

AS-i networking may be not available on some targets. Please refer to OEM instructions for further details about available features.

Interface

```
Params := ASiReadPP (Master, Slave);
bOK := ASiWritePP (Master, Slave, Params);
bOK := ASiSendParam (Master, Slave, Params);
Params := ASiReadPI (Master, Slave);
bOK := ASiStorePI (Master);
```

Arguments

`Master : DINT`	Index of the AS-i master (1..N) such as shown in configuration
`Slave : DINT`	Address of the AS-i slave (1..32 / 33..63)
`Params : DINT`	Value of AS-i parameters
`bOK : BOOL`	TRUE if successful

3.5 AVERAGE / AVERAGEL

Function Block - Calculates the average of signal samples.

3.5.1 Inputs

`RUN : BOOL`	Enabling command
`XIN : REAL`	Input signal
`N : DINT`	Number of samples stored for average calculation - Cannot exceed 128

3.5.2 Outputs

`XOUT : REAL` Average of the stored samples (*)

(*) AVERAGEL has LREAL arguments.

3.5.3 Remarks

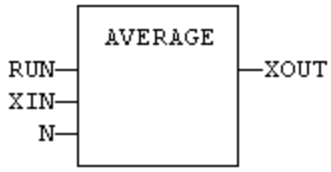
The average is calculated according to the number of stored samples, that can be less that N when the block is enabled. In FFLD language, the input rung is the RUN command. The output rung keeps the state of the input rung.

The "N" input is take into account only when the RUN input is FALSE. So the "RUN" needs to be reset after a change.

3.5.4 ST Language

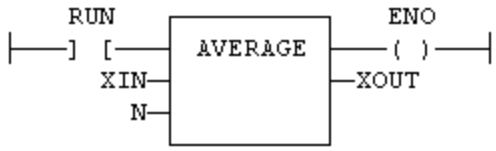
```
(* MyAve is a declared instance of AVERAGE function block *)
MyAve (RUN, XIN, N);
XOUT := MyAve.XOUT;
```

3.5.5 FBD Language



3.5.6 FFLD Language

```
(* ENO has the same state as RUN *)
```



3.5.7 IL Language:

```
(* MyAve is a declared instance of AVERAGE function block *)
Op1: CAL MyAve (RUN, XIN, N)
   FFLD  MyAve.XOUT
   ST  XOUT
```

See also

INTEGRAL DERIVATE LIM_ALRM HYSTER STACKINT

3.6 CurveLin

Function block- Linear interpolation on a curve.

3.6.1 Inputs

X : REAL X coordinate of the point to be interpolated.

XAxis : REAL[] X coordinates of the known points of the X axis.

YVal : REAL[] Y coordinate of the points defined on the X axis.

3.6.2 Outputs

Y : REAL Interpolated Y value corresponding to the X input

OK : BOOL TRUE if successful.

ERR : DINT Error code if failed - 0 if OK.

3.6.3 Remarks

This function performs linear interpolation in between a list of points defined in the XAxis single dimension array. The output Y value is an interpolation of the Y values of the two rounding points defined in the X axis. Y values of defined points are passed in the YVal single dimension array.

Values in XAxis must be sorted from the smallest to the biggest. There must be at least two points defined in the X axis. YVal and XAxis input arrays must have the same dimension.

In case the X input is less than the smallest defined X point, the Y output takes the first value defined in YVal and an error is reported. In case the X input is greater than the biggest defined X point, the Y output takes the last value defined in YVal and an error is reported.

The ERR output gives the cause of the error if the function fails:

Error Code	Meaning
0	OK
1	Invalid dimension of input arrays
2	Invalid points for the X axis
4	X is out of the defined X axis

3.7 CycleStop

Function - Sets the application in cycle stepping mode.

3.7.1 Inputs

`IN : BOOL` Condition

3.7.2 Outputs

`Q : BOOL` TRUE if performed

3.7.3 Remarks

This function turns the Virtual Machine in "Cycle Stepping" mode. Restarting normal execution will be performed using the debugger.

The VM is set in "cycle stepping" mode only if the IN argument is TRUE.

The current main program and all possibly called sub-programs or UDFBs are normally performed up the end. Other programs of the cycle are not executed.

3.8 DERIVATE

Function Block - Derivates a signal.

3.8.1 Inputs

`RUN : BOOL` Run command: TRUE=derivate / FALSE=hold
`XIN : REAL` Input signal
`CYCLE : TIME` Sampling period (must not be less than the target cycle timing)

3.8.2 Outputs

`XOUT : REAL` Output signal

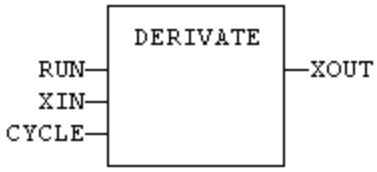
3.8.3 Remarks

In FFLD language, the input rung is the RUN command. The output rung keeps the state of the input rung.

3.8.4 ST Language

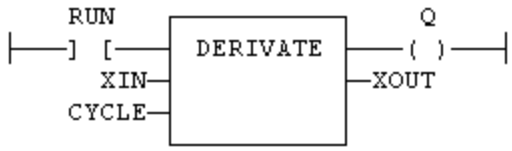
```
(* MyDerv is a declared instance of DERIVATE function block *)
MyDerv (RUN, XIN, CYCLE);
XOUT := MyDerv.XOUT;
```

3.8.5 FBD Language



3.8.6 FFLD Language

```
(* ENO has the same state as RUN *)
```



3.8.7 IL Language:

```
(* MyDerv is a declared instance of DERIVATE function block *)
Op1: CAL MyDerv (RUN, XIN, CYCLE)
FFLD  MyDerv.XOUT
ST  XOUT
```

See also

AVERAGE INTEGRAL LIM_ALRM HYSTERSTACKINT

3.9 Dynamic memory allocation functions

The following functions enable the dynamic allocation of arrays for storing DINT integer values:

`ARCREATE` allocates an array of DINT integers
`ARREAD` read a DINT integer in an array allocated by `ARCREATE`
`ARWRITE` write a DINT integer in an array allocated by `ARCREATE`

Warning

• The memory used for those arrays is allocated directly by the Operating System of the target. There is no insurance that the required memory space will be available.
• Allocating large arrays may cause the Operating System to be instable or slow down the performances of the target system.
• Dynamic memory allocation may be unsuccessful (not enough memory available on the target). Your application should process such error cases in a safe way.
• Dynamic memory allocation may be not available on some targets. Please refer to OEM instructions for further details about available features.

ARCREATE: array allocation

```
OK := ARCREATE (ID, SIZE);
```

```
ID : DINT    integer ID to be assigned to the array (first possible ID is 0)
SIZE : DINT  wished number of DINT values to be stored in the array
OK : DINT    return check
```

Return values

```
1    OK - array is allocated and ready for read / write operations
2    the specified ID is invalid or already used for another array
3    the specified size is invalid
4    not enough memory (action denied by the Operating System)
```

The memory allocated by ARCREATE will be released when the application stops.

ARREAD: read array element

```
VAL := ARREAD (ID, POS);
```

```
ID : DINT    integer ID of the array
POS : DINT   index of the element in the array (first valid index is 0)
VAL : DINT   value of the specified item or 0 if arguments are invalid
```

ARWRITE: write array element

```
OK := ARWRITE (ID, POS, VAL);
```

```
ID : DINT    integer ID of the array
POS : DINT   index of the element in the array (first valid index is 0)
VAL : DINT    value to be assigned to the element
OK : DINT    return check
```

Return values

```
1    OK - element was forced successfully
2    the specified ID is invalid (not an allocated array)
3    the specified index is invalid (out of array bounds)
```

3.10 EnableEvents

Function - Enable or disable the production of events for binding(runtime to runtime variable exchange)

3.10.1 Inputs

```
EN : BOOL   TRUE to enable events / FALSE to disable events
```

3.10.2 Outputs

`ENO : BOOL` Echo of EN input

3.10.3 Remarks

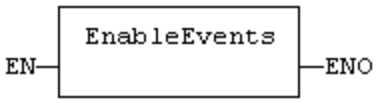
Production is enabled when the application starts. The first production will be operated after the first cycle. So to disable events since the beginning, you must call `EnableEvents (FALSE)` in the very first cycle.

In FFLD language, the input rung (EN) enables the event production, and the output rung keeps the state of the input rung. In IL language, the input must be loaded before the function call.

3.10.4 ST Language

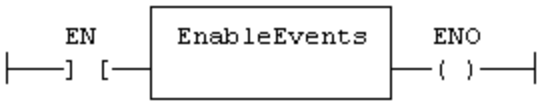
```
ENO := EnableEvents (EN);
```

3.10.5 FBD Language



3.10.6 FFLD Language

```
(* Events are enables if EN is TRUE *)
(* ENO has the same value as EN *)
```



3.10.7 IL Language:

```
Op1: FFLD  EN
     EnableEvents
     ST  ENO
```

3.11 FatalStop

Function - Breaks the application in fatal error.

3.11.1 Inputs

`IN : BOOL` Condition

3.11.2 Outputs

`Q : BOOL` TRUE if performed

3.11.3 Remarks

This function breaks the current cycle and sets the Virtual Machine in "ERROR" mode. Restarting normal execution will be performed using the debugger.

The VM is stopped only if the IN argument is TRUE. The end of the current cycle is then not performed.

3.12 FIFO

Function block - Manages a "first in / first out" list

3.12.1 Inputs

`PUSH : BOOL`	Push a new value (on rising edge)
`POP : BOOL`	Pop a new value (on rising edge)
`RST : BOOL`	Reset the list
`NEXTIN : ANY`	Value to be pushed
`NEXTOUT : ANY`	Value of the oldest pushed value - *updated after call!*
`BUFFER : ANY`	Array for storing values

3.12.2 Outputs

`EMPTY : BOOL`	TRUE if the list is empty
`OFLO : BOOL`	TRUE if overflow on a PUSH command
`COUNT : DINT`	Number of values in the list
`PREAD : DINT`	Index in the buffer of the oldest pushed value
`PWRITE : DINT`	Index in the buffer of the next push position

3.12.3 Remarks

NEXTIN, NEXTOUT and BUFFER must have the same data type *and cannot be STRING*.

The NEXTOUT argument specifies a variable which is filled with the oldest push value after the block is called.

Values are stored in the "BUFFER" array. Data is arranged as a roll over buffer and is never shifted or reset. Only read and write pointers and pushed values are updated. The maximum size of the list is the dimension of the array.

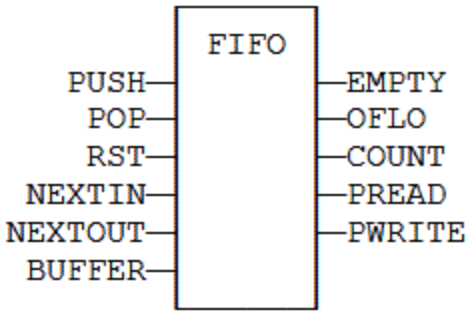
The first time the block is called, it remembers on which array it should work. If you call later the same instance with another BUFFER input, the call is considered as invalid and makes nothing. Outputs reports an empty list in this case.

In FFLD language, input rung is the PUSH input. The output rung is the EMPTY output.

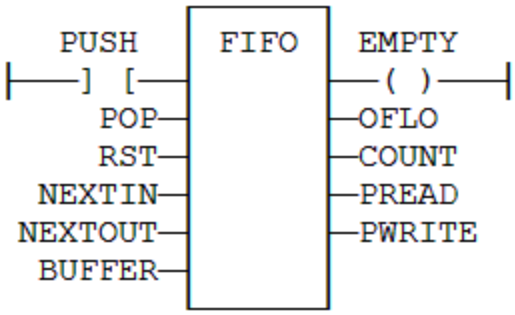
3.12.4 ST Language

```
(* MyFIFO is a declared instance of FIFO function block *)
MyFIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER);
EMPTY := MyFIFO.EMPTY;
OFLO := MyFIFO.OFLO;
COUNT := MyFIFO.COUNT;
PREAD := MyFIFO.PREAD;
PWRITE := MyFIFO.PWRITE;
```

3.12.5 FBD Language



3.12.6 FFLD Language



3.12.7 IL Language

```
(* MyFIFO is a declared instance of FIFO function block *)
Op1: CAL MyFIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER)
FFLD MyFIFO.EMPTY
ST EMPTY
FFLD MyFIFO.OFLO
ST OFLO
FFLD MyFIFO.COUNT
ST COUNT
FFLD MyFIFO.PREAD
ST PREAD
FFLD MyFIFO.PWRITE
ST PWRITE
```

See also

LIFO

3.13 File management functions

The following functions enable sequential read / write operations in disk files:

Function	Use
"F_AOPEN" (see page 185)	Create or open a file in append mode
"F_CLOSE" (see page 185)	Close an open file
"F_COPY" (see page 185)	Copy a file
"F_DELETE" (see page 185)	Remove a file
"F_EOF" (see page 186)	Test if the end of the file is reached in a file that is open for reading

Function	Use
"F_EXIST" (see page 186)	Test if a file exists
F_GETSIZE	Get the size of a file
"F_RENAME" (see page 186)	Rename a file
"F_ROPEN" (see page 186)	Open a file for reading
"F_WOPEN" (see page 187)	Create or reset a file and open it for writing
"FA_READ" (see page 187)	Read a DINT integer from a binary file
"FA_WRITE" (see page 187)	Write a DINT integer to a binary file
"FB_READ" (see page 187)	Read binary data from a file
"FB_WRITE" (see page 187)	Write binary data to a file
"FM_READ" (see page 188)	Read a string value from a text file
"FM_WRITE" (see page 188)	Write a string value to a text file
"SD_MOUNT" (see page 188)	Mount an SD card
"SD_UNMOUNT" (see page 188)	Unmount an SD card
"SD_ISREADY" (see page 189)	Check that the SD card is ready for read/write

Related function blocks:

`LogFileCSV` log values of variables to a CSV file

Each file is identified in the application by a unique handle manipulated as a DINT value. The file handles are allocated by the target system. Handles are returned by the Open functions and used in all other calls for identifying the file.

Warnings

• These functions can have a serious impact on CPU load and the life expectancy of a flash drive. **It is highly recommended that these be used on an event basis, and not at every PLC cycle.**
• Files are opened and closed directly by the Operating System of the target. Opening some files can be dangerous for system safety and integrity. **The number of open files may be is limited to only ONE file by the target system**.

Notes

• Opening a file can be unsuccessful (invalid path or file name, too many open files...) Your application must process such error cases in a safe way.
• File management may be unavailable on some targets.
• Memory on the SD card is available in addition to the existing flash memory.
• Valid paths for storing files depend on the target implementation.
• Error messages are logged in the Controller log section of KAS Run Time where there is a failure in any related function block.
• Using the KAS Simulator, all pathnames are ignored, and files are stored in a reserved directory. Only the file name passed to the Open functions is taken into account.
• PAC and AKD PDMM binary files are not identical. AKD PDMM files are big endian, meaning the data structures between the files are different.

3.13.1 SD Card Access

Files may be written to and read from an SD card. This is typically used for storing a firmware image for Recovery Mode.

To use an SD card on the PDMM:

1. Ensure that the SD card is inserted
2. Mount the card using "SD_MOUNT" (see page 188)

3. Ensure the card is accessible using "SD_ISREADY" (see page 189) before performing a read or write action
4. Unmount the card, if desired, using "SD_UNMOUNT" (see page 188) after performing read/write actions

3.13.2 System Conventions

Depending upon the system used, paths to file locations may be defined as either absolute (`C://dir1/file1`) or relative paths (`/dir1/file1`). Not all systems handle all options, and the paths will vary depending upon the system.

System	Absolute Paths	Relative Paths
PAC	X	X
Simulator	X	X
AKD PDMM		X

3.13.2.1 PAC Path Conventions

When a relative path is provided to the function blocks, the path is appended with the default userdata folder, which is:
`<Installed Directory>Kollmorgen/Kollmorgen Automation Suite/Sinope Runtime/Application/userdata`

3.13.2.2 Simulator Path Conventions

When a relative path is provided to the function blocks, the path is appended with the default userdata folder, which is:
`<Installed Directory>Kollmorgen/Kollmorgen Automation Suite/Sinope Simulator/Application/userdata`

3.13.2.3 AKD PDMM Path Conventions

AKD PDMM only allows for relative paths and there is no support for creating directories on the AKD PDMM. Any path provided to these function blocks, `file1` for example, will be appended with the default userdata folder which is:
`/mount/flash/userdata`

3.13.2.4 SD Card Path Conventions

To access the SD card memory a valid SD card label must be used at the beginning of the path, followed by the relative path to the SD card. *(Valid SD Card Label)/(Relative Path)*

A valid SD card relative path starts with `//`, `/`, `\\`, or `\`. This is immediately followed by `SDCard` which is followed by `\` or `/`. Please note that this path label is case insensitive.

Valid Paths	Notes
//SDCard/file1	
\Sdcard/dir1/file1	dir1 must have been already created
/sdcard/dir1/file1	dir1 must have been already created
//sdCard\file1	

Invalid Paths	Reason for being invalid
///SDCard/file1	Started with more than two forward or backward slashes
/\Sdcard/dir1/file1	Started with one forward and one backward slash
/sdcarddir1/file1	No forward or backward slash
/sdcard1/dir1/file1	Invalid label

In order to maintain compatibility with a PAC or Simulator, the `SDCard` folder is created inside the userdata folder . File access points to `userdata/SDCard` when a PDMM SDCard path is used on a PAC or Simulator.

3.13.2.5 File Name Warning - Limitations

File names in the PAC flash storage are case-insensitive. File names in the PDMM flash storage are case-sensitive and the SD card (FAT16 or FAT32) are not case-sensitive.

Storage	File System	Case-Sensitive
PAC compact flash	NTFS	No
PDMM embedded flash	FFS3 (POSIX-like)	Yes
PDMM SD card	FAT16 or FAT32	No

For example, two files (`MyFile.txt` and `myfile.txt`) can exist in the same directory of the PDMM flash, but cannot exist in the same directory on a PAC or the PDMM's SD card. If you copy two files (via backup operation or function) with the same name, but different upper/lower case letters, from the PDMM flash to the SD card, one of the files will be lost. **To prevent conflicts and to keep your application compatible across all platforms, use unique filenames and do not rely on case-sensitive filenames.**

3.13.3 F_AOPEN

Open a file in "append" mode

```
ID := F_AOPEN (PATH);
```

PATH : STRING Name of the file. Can include a path name according to target system conventions.
ID : DINT ID of the open file or NULL if the file can't be read

If the file does not exist, it is created. If the file already exists, it is opened at the end for appending.

3.13.4 F_CLOSE

Close an open file

```
OK := F_CLOSE (ID);
```

ID : DINT ID of the open file
OK : BOOL return check; TRUE if successful

3.13.5 F_COPY

Copy source file contents to a destination file. Please note that large files will take a noticeable amount of time to complete. For example, a 1000KB file takes approximately 0.6 seconds. The output status is set after the file copy operation is complete.

```
OK := F_COPY (SRC, DST);
```

SRC : STRING Name of the source file (must exist). Can include a pathname according to target system conventions.
DST : STRING Name of the destination file. Can include a pathname according to target system conventions.
OK : BOOL TRUE is successful

3.13.6 F_DELETE

Remove a file

```
OK := F_DELETE (PATH);
```

PATH : STRING Name of the file (must exist). Can include a pathname according to target system conventions.
OK : BOOL TRUE if successful

3.13.7 F_EOF

Test if the end of a file is encountered

```
OK := F_EOF (ID);
```

ID : DINT ID of the open file
OK : BOOL TRUE if the end of the file has been encountered

F_EOF must be used only for files open in read mode by the `F_ROPEN` function.

3.13.8 F_EXIST

Test if file exists

```
OK := F_EXIST (PATH);
```

PATH : STRING Name of the file, can include a path name according to target system conventions.
OK : BOOL TRUE if the file exists

3.13.9 F_GETSIZE

Get the size of a file. Note that this function block returns `0` when the file size is zero or if the file is not present.

```
SIZE := F_GETSIZE (PATH);
```

PATH : STRING Name of the file, can include a path name according to target system conventions
SIZE : DINT Size of the file in bytes

3.13.10 F_RENAME

Rename a file

```
OK := F_RENAME (PATH, NEWNAME);
```

PATH : STRING Name of the file (must exist). Can include a pathname according to target system conventions.
NEWNAME : STRING New name for the file
OK : BOOL TRUE if successful

3.13.11 F_ROPEN

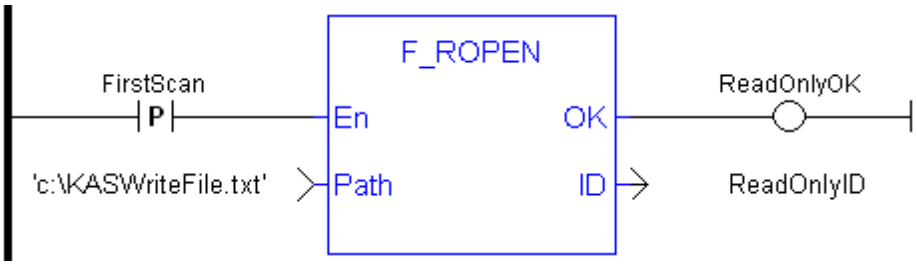
Open a file for reading

3.13.11.1 Example

Structured Text

```
ID := F_ROPEN( PATH ) ;
```

Ladder Diagram



Note

The positive transition on each file operation FB prevents to open the file every time the program runs (each cycle).

PATH : STRING Name of the file; the file must exist. Can include a path name according to target system conventions.
ID : DINT ID of the open filer NULL if the file can't be read

3.13.12 F_WOPEN

Open a file for writing

```
ID := F_WOPEN (PATH);
```

PATH : STRING Name of the file. Can include a path name according to target system conventions.
ID : DINT ID of the open file or NULL if the file can't be read

If the file does not exist, it is created. If the file already exists, its contents are cleared.

3.13.13 FA_READ

Read a DINT value from a file

```
Q := FA_READ (ID);
```

ID: DINT ID of a file open for reading
Q : DINT read value or 0 in case of error

Integer values read by `FA_READ` must have been written by the `FA_WRITE` function. Integers are stored in binary format in the file, using memory conventions of the target system.

3.13.14 FA_WRITE

Write a DINT value to a file

```
OK := FA_WRITE (ID, IN);
```

ID : DINT ID of a file open for writing
IN : DINT integer value to be written
OK : BOOL return check; TRUE if successful

Integers are stored in binary format in the file, using memory conventions of the target system.

3.13.15 FB_READ

Read binary data from a file

```
OK := FB_READ (ID, V);
```

ID : DINT ID of a file open for writing
V : ANY variable to be read; cannot be a string.
OK : BOOL return check; TRUE if successful

Variables are stored in binary format in the file, using memory conventions of the target system.

3.13.16 FB_WRITE

Write binary data to a file

```
OK := FB_WRITE (ID, V);
```

ID : DINT	ID of a file open for writing
V : ANY	variable to be written; cannot be a string.
OK : BOOL	return check; TRUE if successful

Variables are stored in binary format in the file, using memory conventions of the target system.

3.13.17 FM_READ

Read a string value from a file

```
Q := FM_READ (ID);
```

ID : DINT	ID of a file open for reading
Q : STRING	read value or empty string in case of error

This function is intended to read a text line in the file. Reading stops when end of line character is encountered. Reading stops when the maximum length declared for the return variable is reached.

3.13.18 FM_WRITE

Write a string value to a file

```
OK := FM_WRITE (ID, IN);
```

ID : DINT	ID of a file open for writing
IN : STRING	string value to be written
OK : BOOL	return check; TRUE if successful

This function writes a text line in the file. End of line character is systematically written after the input string.

3.13.19 SD_MOUNT

Mount the SDCard on the PDMM. This will not perform any action, and always return TRUE with a PAC or Simulator.

```
OK := SD_MOUNT();
```

OK : BOOL	TRUE if mounting SDCard is successful

NOTE:

Before performing, make sure the SDCard is inserted.

TIP:

It is recommended that SD_MOUNT be used only when motion is not started.

3.13.20 SD_UNMOUNT

Un-mount the SDCard from the PDMM. This will not perform any action, and always return TRUE with a PAC or Simulator.

```
OK := SD_UNMOUNT();
```

OK : BOOL	TRUE if un-mounting SDCard is successful

TIP:

It is recommended that SD_UNMOUNT be used only when motion is not started.

3.13.21 SD_ISREADY

Verify if the SDCard is mounted on the PDMM. This will verify if the SDCard folder is available inside the userdata folder when using a PAC or Simulator.

```
OK := SD_ISREADY();
```

OK : BOOL — TRUE if the SDCard is mounted (PDMM) or if the SDCard folder is avalilable (PAC)

3.14 GETSYSINFO

Function - Returns system information.

3.14.1 Inputs

`INFO : DINT` Identifier of the requested information

3.14.2 Outputs

`Q : DINT` Value of the requested information or 0 if error

3.14.3 Remarks

The INFO parameter can be one of the following predefined values:

_SYSINFO_TRIGGER_MICROS	programmed cycle time in micro-seconds
_SYSINFO_TRIGGER_MS	programmed cycle time in milliseconds
_SYSINFO_CYCLETIME_MICROS	duration of the previous cycle in micro-seconds
_SYSINFO_CYCLETIME_MS	duration of the previous cycle in milliseconds
_SYSINFO_CYCLEMAX_MICROS	maximum detected cycle time in micro-seconds
_SYSINFO_CYCLEMAX_MS	maximum detected cycle time in milliseconds
_SYSINFO_CYCLESTAMP_MS	time stamp of the current cycle in milliseconds (OEM dependent)
_SYSINFO_CYCLEOVERFLOWS	number of detected cycle time overflows
_SYSINFO_CYCLECOUNT	counter of cycles
_SYSINFO_APPVERSION	version number of the application
_SYSINFO_APPSTAMP	compiling date stamp of the application
_SYSINFO_CODECRC	CRC of the application code
_SYSINFO_DATACRC	CRC of the application symbols

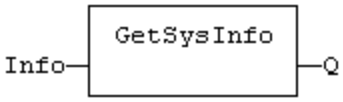
In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

In IL, the input must be loaded in the current result before calling the function.

3.14.4 ST Language

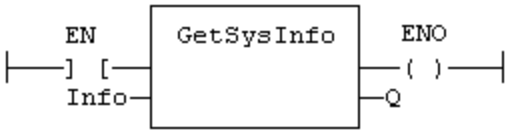
```
Q := GETSYSINFO (INFO);
```

3.14.5 FBD Language



3.14.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



3.14.7 IL Language:

```
Op1: FFLD  INFO
     GETSYSINFO
     ST  Q
```

3.15 HYSTER

Function Block - Hysteresis detection.

3.15.1 Inputs

`XIN1 : REAL`	First input signal
`XIN2 : REAL`	Second input signal
`EPS : REAL`	Hysterisis

3.15.2 Outputs

`Q : BOOL` Detected hysteresis: TRUE if XIN1 becomes greater than XIN2+EPS and is not yet below XIN2-EPS

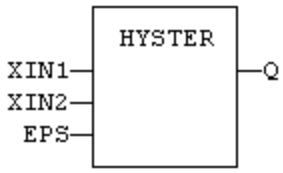
3.15.3 Remarks

The hysteresis is detected on the difference of XIN1 and XIN2 signals. In FFLD language, the input rung (EN) is used for enabling the block. The output rung is the Q output.

3.15.4 ST Language

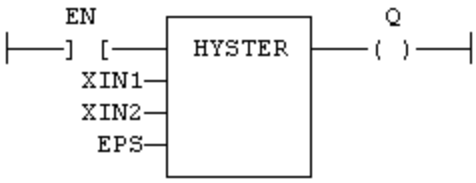
```
(* MyHyst is a declared instance of HYSTER function block *)
MyHyst (XIN1, XIN2, EPS);
Q := MyHyst.Q;
```

3.15.5 FBD Language



3.15.6 FFLD Language

```
(* The block is not called if EN is FALSE *)
```



3.15.7 IL Language:

```
(* MyHyst is a declared instance of HYSTER function block *)
Op1: CAL MyHyst (XIN1, XIN2, EPS)
FFLD MyHyst.Q
ST Q
```

See also

AVERAGE INTEGRAL DERIVATE LIM_ALRM STACKINT

3.16 INTEGRAL

Function Block - Calculates the integral of a signal.

3.16.1 Inputs

`RUN : BOOL`	Run command: TRUE=integrate / FALSE=hold
`R1 : BOOL`	Overriding reset
`XIN : REAL`	Input signal
`X0 : REAL`	Initial value
`CYCLE : TIME`	Sampling period (must not be less than the target cycle timing)

3.16.2 Outputs

`Q : DINT`	Running mode report: NOT (R1)
`XOUT : REAL`	Output signal

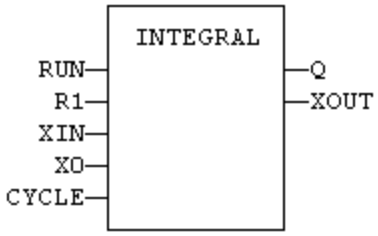
3.16.3 Remarks

In FFLD language, the input rung is the RUN command. The output rung is the Q report status.

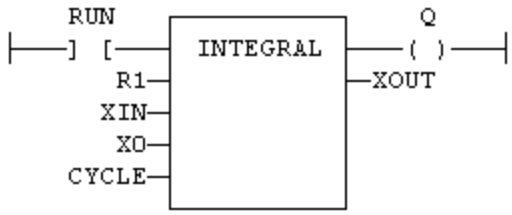
3.16.4 ST Language

```
(* MyIntg is a declared instance of INTEGRAL function block *)
MyIntg (RUN, R1, XIN, X0, CYCLE);
Q := MyIntg.Q;
XOUT := MyIntg.XOUT;
```

3.16.5 FBD Language



3.16.6 FFLD Language



3.16.7 IL Language:

```
(* MyIntg is a declared instance of INTEGRAL function block *)
Op1: CAL MyIntg (RUN, R1, XIN, X0, CYCLE)
   FFLD  MyIntg.Q
   ST  Q
   FFLD  MyIntg.XOUT
   ST  XOUT
```

See also

AVERAGE DERIVATE LIM_ALRM HYSTER STACKINT

3.17 LIFO

Function block - Manages a "last in / first out" stack

3.17.1 Inputs

PUSH : BOOL	Push a new value (on rising edge)
POP : BOOL	Pop a new value (on rising edge)
RST : BOOL	Reset the list
NEXTIN : ANY	Value to be pushed
NEXTOUT : ANY	Value at the top of the stack - *updated after call!*
BUFFER : ANY	Array for storing values

3.17.2 Outputs

EMPTY : BOOL	TRUE if the stack is empty
OFLO : BOOL	TRUE if overflow on a PUSH command
COUNT : DINT	Number of values in the stack
PREAD : DINT	Index in the buffer of the top of the stack
PWRITE : DINT	Index in the buffer of the next push position

3.17.3 Remarks

NEXTIN, NEXTOUT and BUFFER must have the same data type *and cannot be STRING*.

The NEXTOUT argument specifies a variable which is filled with the value at the top of the stack after the block is called.

Values are stored in the "BUFFER" array. Data is never shifted or reset. Only read and write pointers and pushed values are updated. The maximum size of the stack is the dimension of the array.

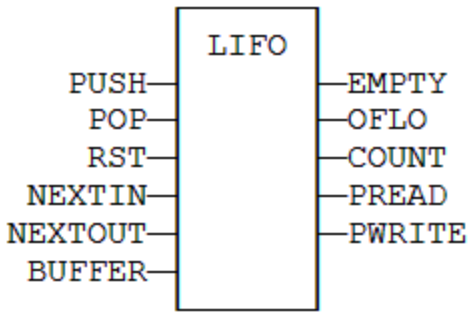
The first time the block is called, it remembers on which array it should work. If you call later the same instance with another BUFFER input, the call is considered as invalid and makes nothing. Outputs reports an empty stack in this case.

In FFLD language, input rung is the PUSH input. The output rung is the EMPTY output.

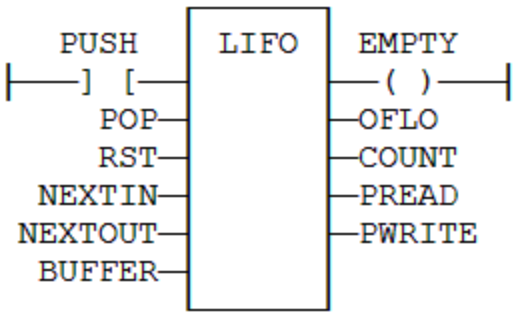
3.17.4 ST Language

```
(* MyLIFO is a declared instance of LIFO function block *)
MyLIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER);
EMPTY := MyLIFO.EMPTY;
OFLO := MyLIFO.OFLO;
COUNT := MyLIFO.COUNT;
PREAD := MyLIFO.PREAD;
PWRITE := MyLIFO.PWRITE;
```

3.17.5 FBD Language



3.17.6 FFLD Language



3.17.7 IL Language

```
(* MyLIFO is a declared instance of LIFO function block *)
Op1: CAL MyLIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER)
FFLD  MyLIFO.EMPTY
ST  EMPTY
FFLD  MyLIFO.OFLO
ST  OFLO
FFLD  MyLIFO.COUNT
ST  COUNT
FFLD  MyLIFO.PREAD
ST  PREAD
FFLD  MyLIFO.PWRITE
ST  PWRITE
```

See also

FIFO

3.18 LIM_ALRM

Function Block - Detects High and Low limits of a signal with hysteresis.

3.18.1 Inputs

`H : REAL`	Value of the High limit
`X : REAL`	Input signal
`L : REAL`	Value of the Low limit
`EPS : REAL`	Value of the hysteresis

3.18.2 Outputs

`QH : BOOL`	TRUE if the signal exceeds the High limit
`Q : BOOL`	TRUE if the signal exceeds one of the limits (equals to QH OR QL)
`QL : BOOL`	TRUE if the signal exceeds the Low limit

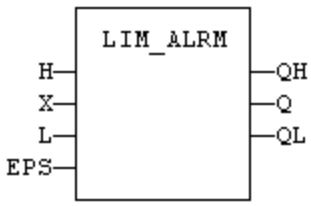
3.18.3 Remarks

In FFLD language, the input rung (EN) is used for enabling the block. The output rung is the QH output.

3.18.4 ST Language

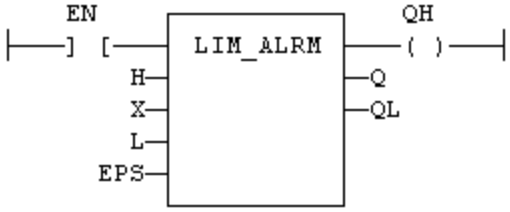
```
(* MyAlarm is a declared instance of LIM_ALRM function block *)
MyAlarm (H, X, L, EPS);
QH := MyAlarm.QH;
Q := MyAlarm.Q;
QL := MyAlarm.QL;
```

3.18.5 FBD Language



3.18.6 FFLD Language

(* The block is not called if EN is FALSE *)



3.18.7 IL Language:

```
(* MyAlarm is a declared instance of LIM_ALRM function block *)
Op1: CAL MyAlarm (H, X, L, EPS)
   FFLD  MyAlarm.QH
   ST  QH
   FFLD  MyAlarm.Q
   ST  Q
   FFLD  MyAlarm.QL
   ST  QL
```

See also

ALARM_A ALARM_M

3.19 LogFileCSV

Function block - Generate a log file in CSV format for a list of variables

3.19.1 Inputs

`LOG : BOOL`	Variables are saved on any rising edge of this input
`RST : BOOL`	Reset the contents of the CSV file
`LIST : DINT`	ID of the list of variables to log (use VLID function)
`PATH : STRING`	Path name of the CSV file

3.19.2 Outputs

`Q : BOOL`	TRUE if the requested operation has been performed without error
`ERR : DINT`	Error report for the last requested operation (0 is OK)

Warning

Calling this function leads to miss several PLC cycles.
File are opened and closed directly by the Operating System of the target.
Opening some files may be dangerous for system safety and integrity. The number of open files may be limited by the target system.

Note

• Opening a file may be unsuccessful (invalid path or file name, too many open files...) Your application has to process such error cases in a safe way.
• File management may be not available on some targets. Please refer to OEM instructions for further details about available features.
• Valid paths for storing files depend on the target implementation. Please refer to OEM instructions for further details about available paths.

3.19.3 Remarks

This function enables to log values of a list of variables in a CSV file. On each rising edge of the LOG input, one more line of values is added to the file. There is one column for each variable, as they are defined in the list.

The list of variables is prepared using the KAS IDE or a text editor. Use the VLID function to get the identifier of the list.

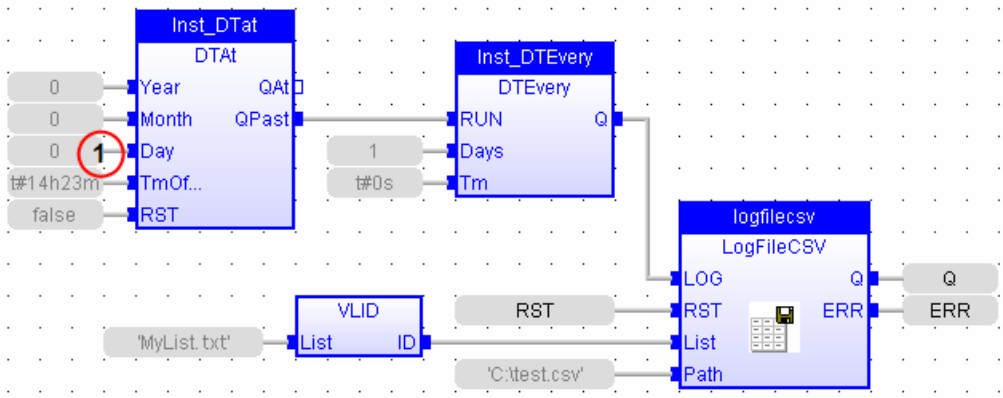
On a rising edge of the RST command, the file is emptied.

When a LOG or RST command is requested, the Q output is set to TRUE if successful.

In case of error, a report is given in the ERR output. Possible error values are:

1 = Cannot reset file on a RST command
2 = Cannot open file for data storing on a LOG command
3 = Embedded lists are not supported by the runtime
4 = Invalid list ID
5 = Error while writing to file

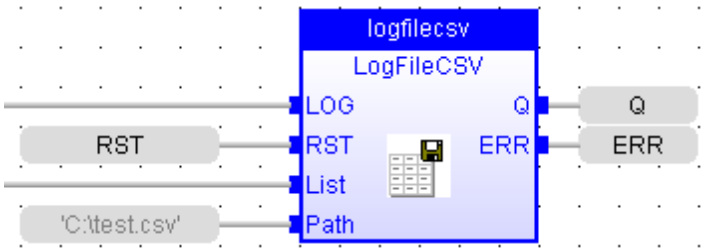
Combined with real time clock management functions, this block provides a very easy way to generate a periodical log file. The following example shows a list and a program that log values everyday at 14h23m (2:23 pm) (see call out 1)



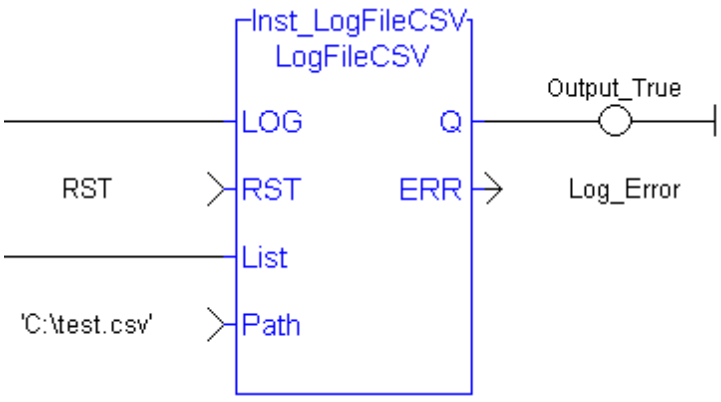
3.19.4 ST Language

```
(* MyLOG is a declared instance of LogFileCSV function block *)
MyLOG (b_LOG, RST, LIST, PATH);
Q := MyLOG.Q;
ERR := MyLog.ERR;
```

3.19.5 FBD Language



3.19.6 FFLD Language



3.19.7 IL Language

```
(* MyLOG is a declared instance of LogFileCSV function block *)
Op1: CAL MyLOG (b_LOG, RST, LIST, PATH);
FFLD MyLOG.Q
ST Q
FFLD MyLog.ERR
ST ERR
```

See also

VLID

3.20 MBSlaveRTU

Function Block - MODBUS RTU Slave protocol on serial port.

3.20.1 Inputs

`IN : BOOL`	Enabling command: the port is open when this input is TRUE
`PORT : STRING`	Settings string for the serial port (e.g. 'COM1:9600,N,8,1')
`SLV : DINT`	MODBUS slave number

3.20.2 Outputs

`Q : BOOL`	TRUE if the port is successfully open

3.20.3 Remarks

When active, this function block manages the MODBUS RTU Slave protocol on the specified serial communication port. The configuration of the MODBUS Slave map (designing MODBUS addresses) is done using the MODBUS configuration tool, from the Fieldbus Configurator.

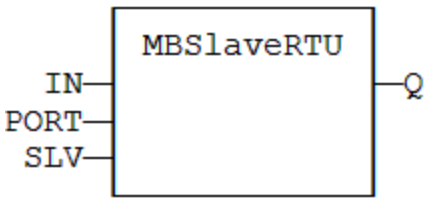
There can be several instances of the MBSlaveRTU working simultaneously on different serial ports. Other MODBUS Slave connections (TCP server, UDP) can also be active at the same time.

The slave number entered in the MODBUS Slave configuration tool is ignored when MODBUS Slave protocol is handled by this function block. Instead, the SLV input specifies the MODBUS slave number.

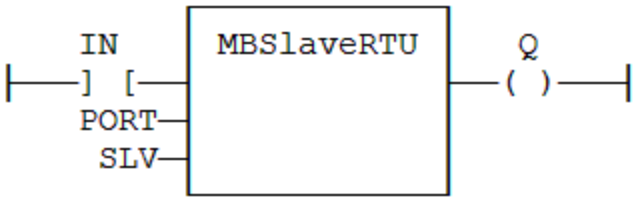
3.20.4 ST Language

```
(* MySlave is a declared instance of MBSlaveRTU function block *)
MySlave (IN, PORT, SLV);
Q := MySlave.Q;
```

3.20.5 FBD Language



3.20.6 FFLD Language



3.20.7 IL Language:

```
(* MySlave is a declared instance of MBSlaveRTU function block *)
Op1: CAL   MySlave (IN, PORT, SLV)
   FFLD    MySlave.Q
   ST   Q
   FFLD    MyCounter.CV
   ST   CV
```

See also

MBSlaveUDP MODBUS configuration

3.21 MBSlaveUDP

Function Block - MODBUS UDP Slave protocol on ETHERNET.

3.21.1 Inputs

`IN : BOOL`	Enabling command: the port is open when this input is TRUE
`PORT : DINT`	ETHERNET port number
`SLV : DINT`	MODBUS slave number
`RTU : BOOL`	Protocol: TRUE = MODBUS RTU / FALSE = Open MODBUS

3.21.2 Outputs

`Q : BOOL`	TRUE if the port is successfully open

3.21.3 Remarks

When active, this function block manages the MODBUS UDP Slave protocol on the specified ETHERNETport. The configuration of the MODBUS Slave map (designing MODBUS addresses) is done using the MODBUS configuration tool, from the Fieldbus Configurator.

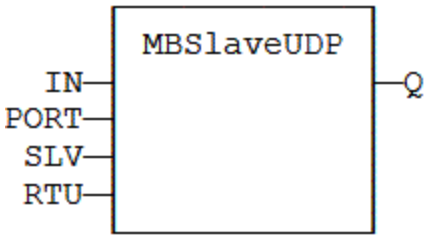
There can be several instances of the MBSlaveUDP working simultaneously on different serial ports. Other MODBUS Slave connections (TCP server, serial) can also be active at the same time.

The slave number entered in the MODBUS Slave configuration tool is ignored when MODBUS Slave protocol is handled by this function block. Instead, the SLV input specifies the MODBUS slave number. If SLV is 0 then the default port number from the MODBUS configuration is used.

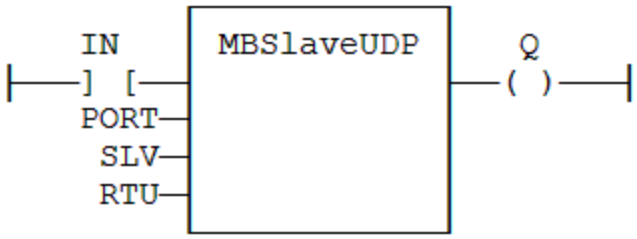
3.21.4 ST Language

```
(* MySlave is a declared instance of MBSlaveUDP function block *)
MySlave (IN, PORT, SLV, RTU);
Q := MySlave.Q;
```

3.21.5 FBD Language



3.21.6 FFLD Language



3.21.7 IL Language:

```
(* MySlave is a declared instance of MBSlaveUDP function block *)
Op1: CAL    MySlave (IN, PORT, SLV, RTU)
     FFLD    MySlave.Q
     ST    Q
     FFLD    MyCounter.CV
     ST    CV
```

See also

MBSlaveRTU MODBUS configuration

3.22 PID

Function Block - PID loop

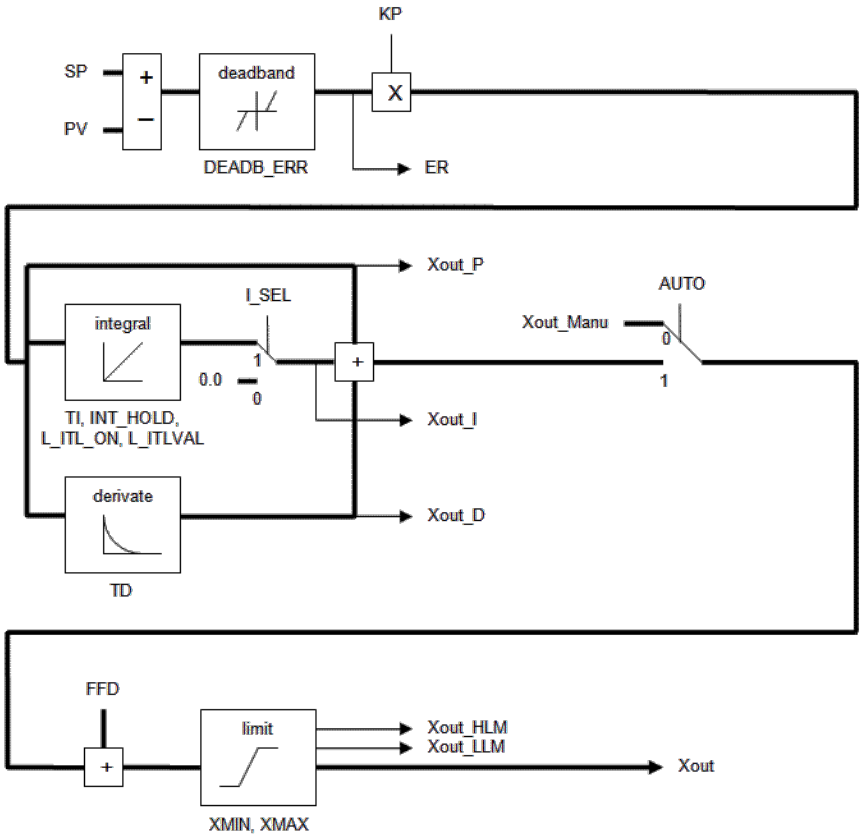
3.22.1 Inputs

Input	Type	Description
AUTO	BOOL	TRUE = normal mode - FALSE = manual mode.
PV	REAL	Process value.
SP	REAL	Set point.
Xout_Manu	REAL	Output value in manual mode.
KP	REAL	Gain.
TI	REAL	Integration time.
TD	REAL	Derivation time.
TS	TIME	Sampling period.
XMIN	REAL	Minimum allowed output value.
XMAX	REAL	Maximum output value.
I_SEL	BOOL	If FALSE, the integrated value is ignored.
INT_HOLD	BOOL	If TRUE, the integrated value is frozen.
I_ITL_ON	BOOL	If TRUE, the integrated value is reset to I_ITLVAL.
I_ITLVAL	REAL	Reset value for integration when I_ITL_ON is TRUE.
DEADB_ERR	REAL	Hysteresis on PV. PV will be considered as unchanged if greater than (PVprev - DEADBAND_W) and less that (PRprev + DEADBAND_W).
FFD	REAL	Disturbance value on output.

3.22.2 Outputs

Output	Type	Description
Xout	REAL	Output command value.
ER	REAL	Last calculated error.
Xout_P	REAL	Last calculated proportional value.
Xout_I	REAL	Last calculated integrated value.
Xout_D	REAL	Last calculated derivated value.
Xout_HLM	BOOL	TRUE if the output valie is saturated to XMIN.
Xout_LLM	BOOL	TRUE if the output value is saturated to XMAX.

3.22.3 Diagram



3.22.4 Remarks

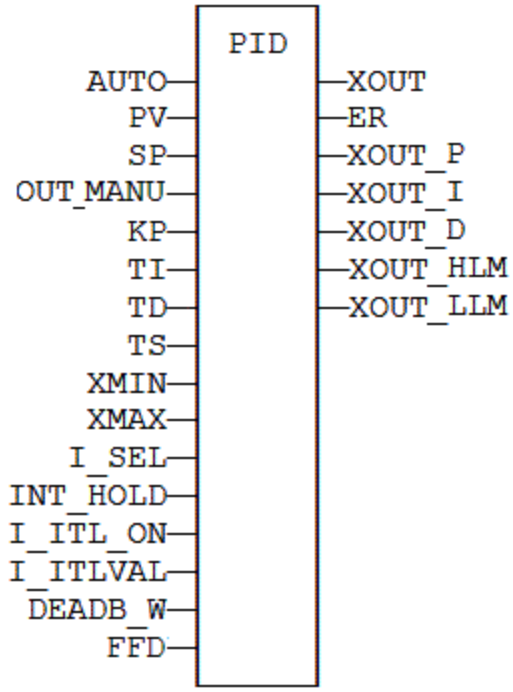
It is important for the stability of the control that the TS sampling period is much bigger than the cycle time.

In FFLD language, the output rung has the same value as the AUTO input, corresponding to the input rung.

3.22.5 ST Language

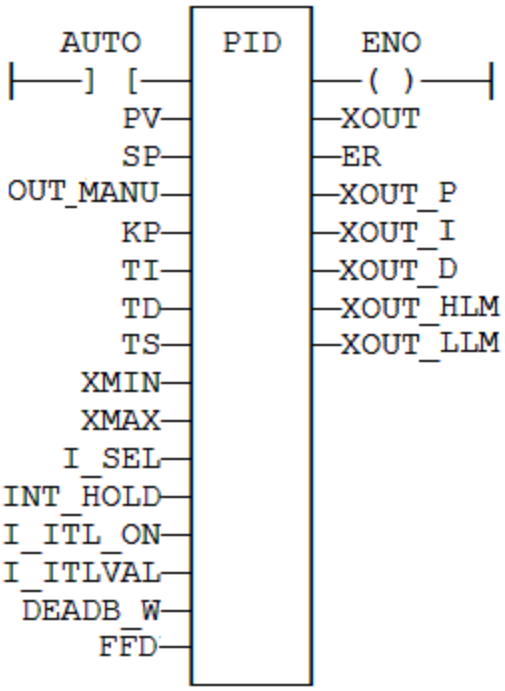
```
(* MyPID is a declared instance of PID function block *)
MyPID (AUTO, PV, SP, XOUT_MANU, KP, TI, TD, TS, XMIN, XMAX,
I_SEL, I_ITL_ON, I_ITLVAL, DEADB_ERR, FFD);
XOUT := MyPID.XOUT;
ER := MyPID.ER;
XOUT_P := MyPID.XOUT_P;
XOUT_I := MyPID.XOUT_I;
XOUT_D := MyPID.XOUT_D;
XOUT_HLM := MyPID.XOUT_HLM;
XOUT_LLM := MyPID.XOUT_LLM;
```

3.22.6 FBD Language



3.22.7 FFLD Language

(* ENO has the same state as the input rung *)



3.22.8 IL Language

(* MyPID is a declared instance of PID function block *)
Op1: CAL MyPID (AUTO, PV, SP, XOUT_MANU, KP, TI, TD, TS,
XMIN, XMAX, I_SEL, I_ITL_ON, I_ITLVAL,
DEADB_ERR, FFD)

```
FFLD  MyPID.XOUT
ST  XOUT
FFLD  MyPID.ER
ST  ER
FFLD  MyPID.XOUT_P
ST  XOUT_P
FFLD  MyPID.XOUT_I
ST  XOUT_I
FFLD  MyPID.XOUT_D
ST  XOUT_D
FFLD  MyPID.XOUT_HLM
ST  XOUT_HLM
FFLD  MyPID.XOUT_LLM
ST  XOUT_LLM
```

3.23 PID Functions

3.23.1 JS_DeadTime - analog delay

call:

INPUT : signal input
N : number of samples (200 max)
DeadTime : delay (seconds)

return:

OUT : output signal

Notes

Allows to put a delay on an analog signal

Warning

The dead time divided by the number of samples must be very greater than cycle time.
If N = 0 the function understands 1
If N > 200 the function understands 200

3.23.2 JS_LeadLag - signal lead / lag

call:

Input : input signal
Lead : lead value
Lag : lag value
Ts : sampling period

return:

Out : output signal

function:

Qn = Qn-1 + Ts/Ti.(Mn-1 - Qn-1) + Td/Ti.(Mn - Mn-1)

Qn : Output at t
Qn-1 : Output at t-1
Mn : Input at t
Mn-1 : Input at t-1
Ts : Sampling period
Ti : Lag
Td : Lead

Notes

If Lag = 0 the function is not executed
Sampling period must be very greater than cycle time

3.23.3 JS_PID - PID loop setpoint balance

call:

LSL : Loop Scale Lo
LSH : Loop Scale Hi
Auto : automatic or manual mode
Pv : Process output value
Sp : Set point value
Ramp : Setpoint Ramp (Unit per Minute)
Balance : Auto/Manu Setpoint balancing
Action : Output action Direct or Reverse
Mixt : Interactive or Non-Interactive PID
Deriv : Derivative action
Feedback : external PID feedback for output manipulations
X0 : Adjustment value: In manual mode, output pid regulator equal to X0
You must connect obligatory a variable (not a constant and no computation on this variable
Kp : Proportionality constant
Ti : Integral time constant in minute
Td : derivative time constant in minute
Ts : Sampling period
Xmin : Minimum limit on output command value
Xmax : Maximum limit on output command value

return:

SPcur : Current Setpoint
Xout : Command

Notes

Automatic mode must be set to false at init.
Balance = 0 Non-balancing Setpoint ; =1 Balancing
Xout is limited inside specified Xmin ,Xmax range.
Xmax should be greater than Xmin.
The integral term is held when Xout reaches the limits
The Ts parameter should be greater (>>>) than the kernel cycle time.
Action :
- Direct (0) Output increase if PV-SP positive and decrease if PV-SP negative
- Reverse (1) Output decrease if PV-SP positive and increase if PV-SP negative
Mixt :
- Interactive PID (0) : The I and D parameters are multipied by KP
- Non-Interactive (1) : The P,I,D parameters are independant
Feedback :
- The range must be 0-100
- Feeback=0 : internal feedback

3.23.4 JS_Ramp - Limit variation speed

call:

- INPUT : input signal
- Rampe : maximum variation speed (Unit/mn)
- Cycle : application cycle time

return:

OUT : output signal

Notes

- The variation of speed is expressed as units per minute
- The Cycle input must be the application cycle time
  Use GetSysInfo (_SYSINFO_CYCLETIME_MS)

3.24 printf

Function - Display a trace output.

3.24.1 Inputs

`FMT : STRING`	Trace message
`ARG1..ARG4 : DINT`	Numerical arguments to be included in the trace

3.24.2 Outputs

`Q : BOOL`	Return check

3.24.3 Remarks

This function works as the famous "printf" function of the "C" language, with up to 4 integer arguments. You can use the following pragmas in the `FMT` trace message to represent the arguments according to their left to the right order:

%ld signed value in decimal
%lu unsigned value in decimal
%lx value in hexadecimal

The trace message is displayed in the LOG window with runtime messages. Trace is supported by the KAS Simulator.

Warning

Your target platform can support trace functions or not. Please refer to OEM instructions for further details on available features.

3.24.4 Example

```
(* i1, i2, i3, i4 are declared as DINT *)
i1 := 1;
i2 := 2;
i3 := 3;
i4 := 4;
printf ('i1=%ld; i2=%ld; i3=%ld; i4=%ld', i1, i2, i3, i4);
```

Output message:

```
i1=1; i2=2; i3=3; i4=4;
```

3.25 RAMP

Function block - Limit the ascendance or descendance of a signal

3.25.1 Inputs

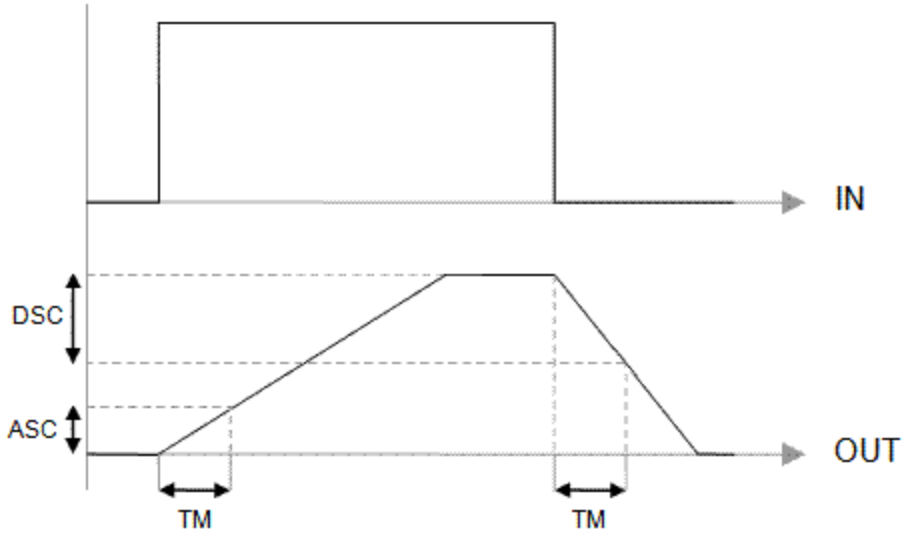
`IN : REAL`	Input signal
`ASC : REAL`	Maximum ascendance during time base
`DSC : REAL`	Maximum descendance during time base

```
TM  : TIME      Time base
RST : BOOL      Reset
```

3.25.2 Outputs

```
OUT : REAL      Ramp signal
```

3.25.3 Time diagram



3.25.4 Remarks

Parameters are not updated constantly. They are taken into account when only:
- the first time the block is called
- when the reset input (RST) is TRUE

In these two situations, the output is set to the value of IN input.

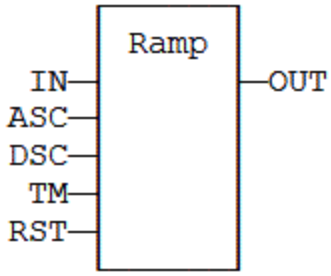
ASC and DSC give the maximum ascendant and descendant growth during the TB time base.
Both must be expressed as positive numbers.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

3.25.5 ST Language

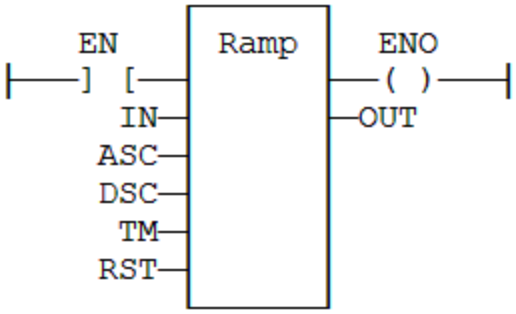
```
(* MyRamp is a declared instance of RAMP function block *)
MyRamp (IN, ASC, DSC, TM, RST);
OUT := MyBlinker.OUT;
```

3.25.6 FBD Language



3.25.7 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



3.25.8 IL Language

```
(* MyRamp is a declared instance of RAMP function block *)
Op1: CAL MyRamp (IN, ASC, DSC, TM, RST)
FFLD  MyBlinker.OUT
ST  OUT
```

3.26 Real Time clock management functions

The following functions read the real time clock of the target system:

`DTCurDate`	Get current date stamp
`DTCurTime`	Get current time stamp
`DTDay`	Get day from date stamp
`DTMonth`	Get month from date stamp
`DTYear`	Get year from date stamp
`DTSec`	Get seconds from time stamp
`DTMin`	Get minutes from time stamp
`DTHour`	Get hours from time stamp
`DTMs`	Get milliseconds from time stamp

The following functions format the current date/time to a string:

`DAY_TIME`	With predefined format
`DTFORMAT`	With custom format

The following functions are used for triggering operations:

`DTAt`	Pulse signal at the given date/time
`DTEvery`	Pulse signal with long period

Warning

The real-time clock may not be available on all controller hardware models. Please consult the controller hardware specifications for real-time clock availability.

DAY_TIME: get current date or time

```
Q := DAY_TIME (SEL);
```

`SEL : DINT`	specifies the wished information (see below)
`Q : STRING`	wished information formatted on a string

Possible values of SEL input

```
1             current time - format: 'HH:MM:SS'
2             day of the week
0 (default)   current date - format: 'YYYY/MM/DD'
```

DTCURDATE: get current date stamp

```
Q := DTCurDate ();
```

```
Q : DINT      numerical stamp representing the current date
```

DTCURTIME: get current time stamp

```
Q := DTCurTime ();
```

```
Q : DINT      numerical stamp representing the current time of the day
```

DTYEAR: extract the year from a date stamp

```
Q := DTYear (iDate);
```

```
IDATE : DINT  numerical stamp representing a date
Q : DINT      year of the date (ex: 2004)
```

DTMONTH: extract the month from a date stamp

```
Q := DTMonth (iDate);
```

```
IDATE : DINT  numerical stamp representing a date
Q : DINT      month of the date (1..12)
```

DTDAY: extract the day of the month from a date stamp

```
Q := DTDay (iDate);
```

```
IDATE : DINT  numerical stamp representing a date
Q : DINT      day of the month of the date (1..31)
```

DTHOUR: extract the hours from a time stamp

```
Q := DTHour (iTime);
```

```
ITIME : DINT  numerical stamp representing a time
Q : DINT      Hours of the time (0..23)
```

DTMIN: extract the minutes from a time stamp

```
Q := DTMin (iTime);
```

```
ITIME : DINT  numerical stamp representing a time
Q : DINT      Minutes of the time (0..59)
```

DTSEC: extract the seconds from a time stamp

```
Q := DTSec (iTime);
```

```
ITIME : DINT  numerical stamp representing a time
Q : DINT      Seconds of the time (0..59)
```

DTMS: extract the milliseconds from a time stamp

```
Q := DTMs (iTime);
```

```
ITIME : DINT  numerical stamp representing a time
Q : DINT      Milliseconds of the time (0..999)
```

3.26.1 DAY_TIME

Function - Format the current date/time to a string.

3.26.1.1 Inputs

```
SEL : DINT    Format selector
```

3.26.1.2 Outputs

`Q : STRING` String containing formatted date or time

Warning

The real-time clock may not be available on all controller hardware models. Please consult the controller hardware specifications for real-time clock availability.

3.26.1.3 Remarks

Possible values of the SEL input are:

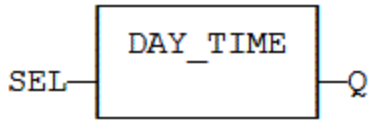
`1` current time - format: 'HH:MM:SS'
`2` day of the week
`0 (default)` current date - format: 'YYYY/MM/DD'

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

3.26.1.4 ST Language

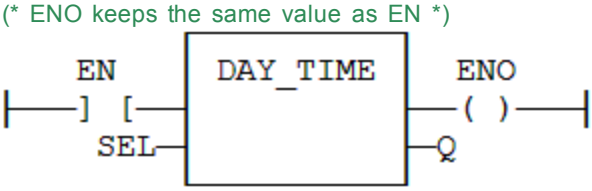
```
Q := DAY_TIME (SEL);
```

3.26.1.5 FBD Language



3.26.1.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



3.26.1.7 IL Language

```
Op1: FFLD SEL
DAY_TIME
ST Q
```

See also

DTFORMAT

3.26.2 DTFORMAT

Function - Format the current date/time to a string with a custom format.

3.26.2.1 Inputs

`FMT: STRING` Format string

3.26.2.2 Outputs

`Q : STRING` String containing formatted date or time

Warning

The real-time clock may not be available on all controller hardware models. Please consult the controller hardware specifications for real-time clock availability.

3.26.2.3 Remarks

The format string may contain any character. Some special markers beginning with the '%' character indicates a date/time information:

%Y Year including century (e.g. 2006)
%y Year without century (e.g. 06)
%m Month (1..12)
%d Day of the month (1..31)
%H Hours (0..23)
%M Minutes (0..59)
%S Seconds (0..59)

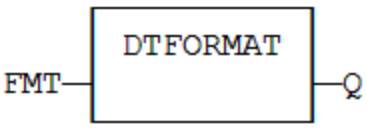
Example

```
(* let's say we are at July 04th 2006, 18:45:20 *)
   Q := DTFORMAT ('Today is %Y/%m/%d - %H:%M:%S');
(* Q is 'Today is 2006/07/04 - 18:45:20 *)
```

3.26.2.4 ST Language

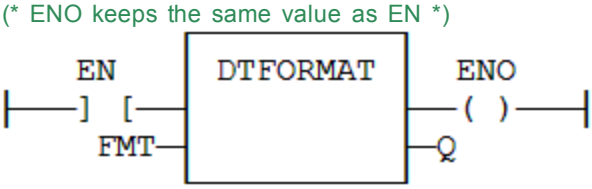
Q := DTFORMAT (FMT);

3.26.2.5 FBD Language



3.26.2.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



3.26.2.7 IL Language

Op1: FFLD FMT
DTFORMAT
ST Q

See also

DAY_TIME

3.26.3 DTAT

Function Block - Generate a pulse at given date and time

3.26.3.1 Inputs

YEAR : DINT	Wished year (e.g. 2006)
MONTH : DINT	Wished month (1 = January)

DAY : DINT	Wished day (1 to 31)
TMOFDAY : TIME	Wished time
RST : BOOL	Reset command

3.26.3.2 Outputs

QAT : BOOL	Pulse signal
QPAST : BOOL	True if elapsed

Warning

The real-time clock may not be available on all controller hardware models. Please consult the controller hardware specifications for real-time clock availability.

3.26.3.3 Remarks

Parameters are not updated constantly. They are taken into account when only:
- the first time the block is called
- when the reset input (RST) is TRUE

In these two situations, the outputs are reset to FALSE.

The first time the block is called with RST=FALSE and the specified date/stamp is passed, the output QPAST is set to TRUE, and the output QAT is set to TRUE for one cycle only (pulse signal).

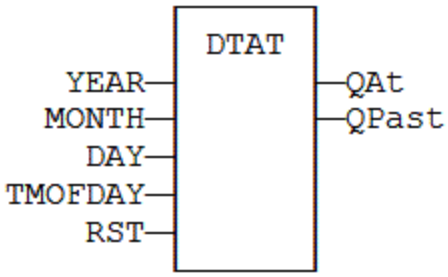
Highest units are ignored if set to 0. For instance, if arguments are "year=0, month=0, day = 3, tmofday=t#10h" then the block will trigger on the next 3rd day of the month at 10h.

In FFLD language, the block is activated only if the input rung is TRUE..

3.26.3.4 ST Language

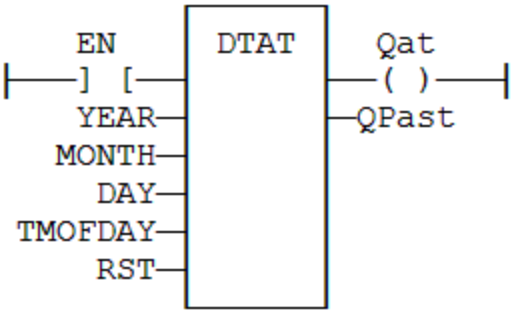
```
(* MyDTAT is a declared instance of DTAT function block *)
MyDTAT (YEAR, MONTH, DAY, TMOFDAY, RST);
QAT := MyDTAT.QAT;
QPAST := MyDTATA.QPAST;
```

3.26.3.5 FBD Language



3.26.3.6 FFLD Language

(* Called only if EN is TRUE *)



3.26.3.7 IL Language:

```
(* MyDTAT is a declared instance of DTAT function block *)
Op1: CAL MyDTAT (YEAR, MONTH, DAY, TMOFDAY, RST)
FFLD  MyDTAT.QAT
ST  QAT
FFLD  MyDTATA.QPAST
ST  QPAST
```

See also

DTEVERY Real time clock functions

3.26.4 DTEVERY

Function Block - Generate a pulse signal with long period

3.26.4.1 Inputs

`RUN : DINT`	Enabling command
`DAYS : DINT`	Period : number of days
`TM : TIME`	Rest of the period (if not a multiple of 24h)

3.26.4.2 Outputs

`Q : BOOL`	Pulse signal

3.26.4.3 Remarks

This block provides a pulse signal with a period of more than 24h. The period is expressed as:

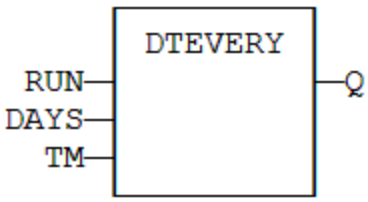
DAYS * 24h + TM

For instance, specifying DAYS=1 and TM=6h means a period of 30 hours.

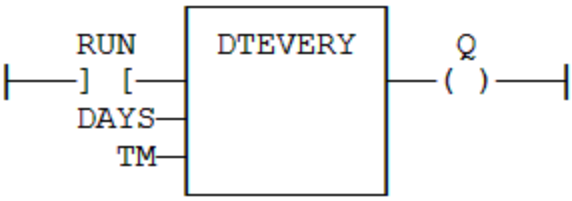
3.26.4.4 ST Language

```
(* MyDTEVERY is a declared instance of DTEVERY function block *)
MyDTEVERY (RUN DAYS, TM);
Q := MyDTEVERY.Q;
```

3.26.4.5 FBD Language



3.26.4.6 FFLD Language



3.26.4.7 IL Language:

```
(* MyDTEVERY is a declared instance of DTEVERY function block *)
Op1: CAL MyDTEVERY (RUN DAYS, TM)
   FFLD  MyDTEVERY.Q
   ST  Q
```

See also

DTAT Real time clock functions

3.27 SERIALIZEIN

Function - Extract the value of a variable from a binary frame

3.27.1 Inputs

`FRAME : USINT`	Source buffer - must be an array
`DATA : ANY(*)`	Destination variable to be copied
`POS : DINT`	Position in the source buffer
`BIGENDIAN : BOOL`	TRUE if the frame is encoded with Big Endian format

(*) `DATA`cannot be a STRING

3.27.2 Outputs

`NEXTPOS : DINT`	Position in the source buffer after the extracted data 0 in case or error (invalid position / buffer size)

3.27.3 Remarks

This function is commonly used for extracting data from a communication frame in binary format.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language

.

The `FRAME` input must fit the input position and data size. If the value cannot be safely extracted, the function returns 0.

The `DATA` input must be directly connected to a variable, and cannot be a constant or complex expression. This variable will be forced with the extracted value.

The function extracts the following number of bytes from the source frame:

1 byte for BOOL, SINT, USINT and BYTE variables
2 bytes for INT, UINT and WORD variables
4 bytes for DINT, UDINT, DWORD and REAL variables
8 bytes for LINT and LREAL variables

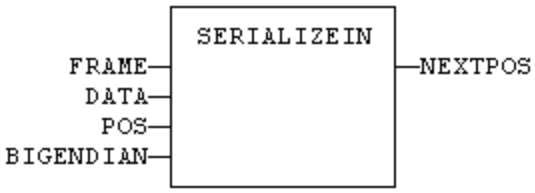
The function cannot be used to serialize STRING variables.

The function returns the position in the source frame, after the extracted data. Thus the return value can be used as a position for the next serialization.

3.27.4 ST Language

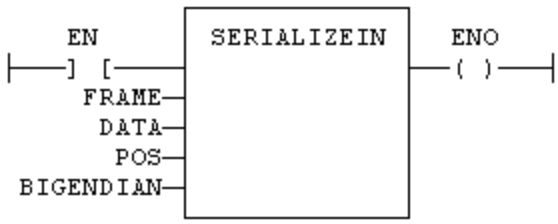
```
Q := SERIALIZEIN (FRAME, DATA, POS, BIGENDIAN);
```

3.27.5 FBD Language



3.27.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



3.27.7 IL Language:

Not available

See also

SERIALIZEOUT

3.28 SERIALIZEOUT

Function - Copy the value of a variable to a binary frame

3.28.1 Inputs

`FRAME : USINT`	Destination buffer - must be an array
`DATA : ANY(*)`	Source variable to be copied
`POS : DINT`	Position in the destination buffer
`BIGENDIAN : BOOL`	TRUE if the frame is encoded with Big Endian format

(*) `DATA`cannot be a STRING

3.28.2 Outputs

`NEXTPOS : DINT`	Position in the destination buffer after the copied data 0 in case or error (invalid position / buffer size)

3.28.3 Remarks

This function is commonly used for building a communication frame in binary format.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language

.

The `FRAME` input must be an array large enough to receive the data. If the data cannot be safely copied to the destination buffer, the function returns 0.

The function copies the following number of bytes to the destination frame:

1 byte for BOOL, SINT, USINT and BYTE variables
2 bytes for INT, UINT and WORD variables
4 bytes for DINT, UDINT, DWORD and REAL variables
8 bytes for LINT and LREAL variables

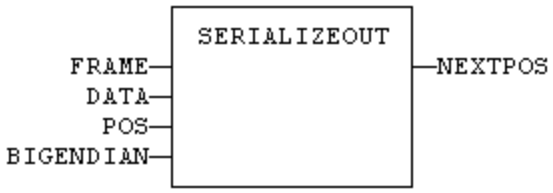
The function cannot be used to serialize STRING variables.

The function returns the position in the destination frame, after the copied data. Thus the return value can be used as a position for the next serialization.

3.28.4 ST Language

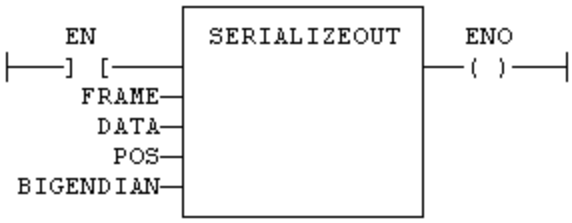
```
Q := SERIALIZEOUT (FRAME, DATA, POS, BIGENDIAN);
```

3.28.5 FBD Language



3.28.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



3.28.7 IL Language:

Not available

See also

SERIALIZEIN

3.29 SerGetString

Function - Extract a string from a binary frame

3.29.1 Inputs

`FRAME : USINT`	Source buffer - must be an array
`DST : STRING`	Destination variable to be copied
`POS : DINT`	Position in the source buffer
`MAXLEN : DINT`	Specifies a fixed length string
`EOS : BOOL`	Specifies a null terminated string
`HEAD : BOOL`	Specifies a string headed with its length

3.29.2 Outputs

NEXTPOS : DINT Position in the source buffer after the extracted data
0 in case or error (invalid position / buffer size)

3.29.3 Remarks

This function is commonly used for extracting data from a communication frame in binary format.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language

.

The FRAME input must fit the input position and data size. If the value cannot be safely extracted, the function returns 0.

The DST input must be directly connected to a variable, and cannot be a constant or complex expression. This variable will be forced with the extracted value.

The function extracts the following bytes from the source frame:

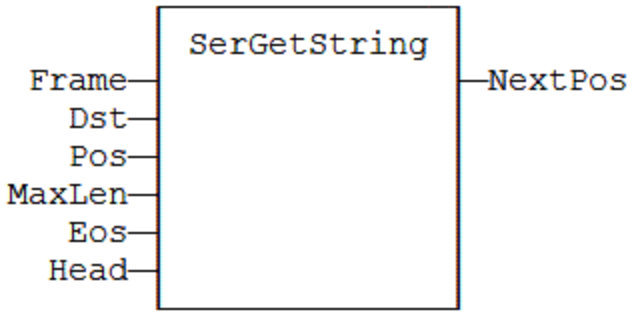
MAXLEN	EOS	HEAD	description
<> 0	any	any	The string is stored on a fixed length specified by MAXLEN. If the string is actually smaller, the space is completed with null bytes.
= 0	TRUE	any	The string is stored with its actual length and terminated by a null byte.
= 0	FALSE	TRUE	The string is stored with its actual length and preceded by its length stored on one byte
=0	FALSE	FALSE	invalid call

The function returns the position in the source frame, after the extracted data. Thus the return value can be used as a position for the next serialization.

3.29.4 ST Language

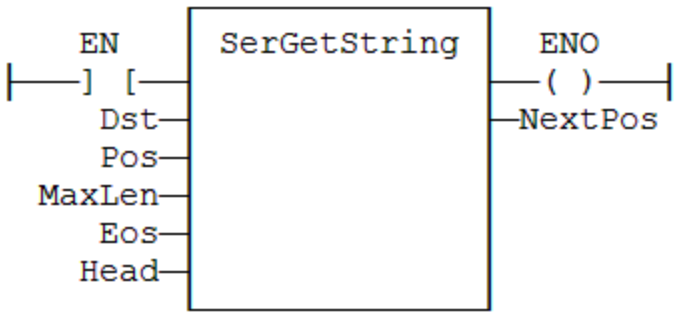
```
Q := SerGetString (FRAME, DSR, POS, MAXLEN, EOS, HEAD);
```

3.29.5 FBD Language



3.29.6 FFLD Language

```
(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)
```



3.29.7 IL Language

Not available

3.30 SerPutString

Function - Copies a string to a binary frame

3.30.1 Inputs

`FRAME : USINT`	Destination buffer - must be an array
`DST : STRING`	Source variable to be copied
`POS : DINT`	Position in the source buffer
`MAXLEN : DINT`	Specifies a fixed length string
`EOS : BOOL`	Specifies a null terminated string
`HEAD : BOOL`	Specifies a string headed with its length

3.30.2 Outputs

`NEXTPOS : DINT`	Position in the destination buffer after the copied data 0 in case or error (invalid position / buffer size)

3.30.3 Remarks

This function is commonly used for storing data to a communication frame.

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language

.

The `FRAME` input must fit the input position and data size. If the value cannot be safely copied, the function returns 0.

The function copies the following bytes to the frame:

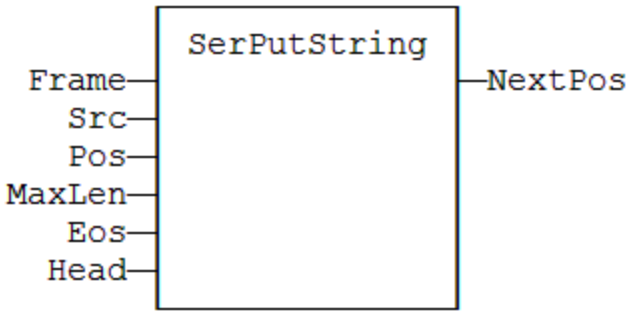
MAXLEN	EOS	HEAD	description
<>0	any	any	The string is stored on a fixed length specified by MAXLEN. If the string is actually smaller, the space is completed with null bytes. If the string is longer, it is truncated.
=0	TRUE	any	The string is stored with its actual length and terminated by a null byte.
=0	FALSE	TRUE	The string is stored with its actual length and preceded by its length stored on one byte
=0	FALSE	FALSE	invalid call

The function returns the position in the source frame, after the stored data. Thus the return value can be used as a position for the next serialization.

3.30.4 ST Language

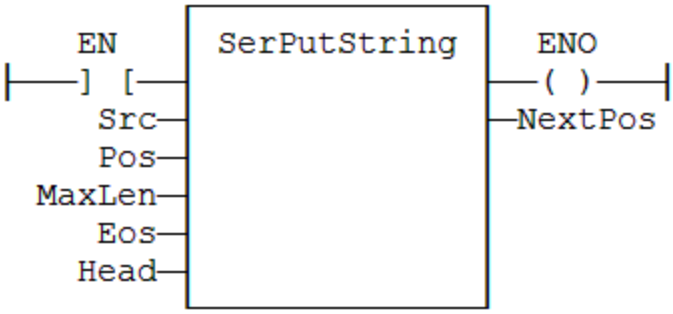
```
Q := SerPutString (FRAME, DSR, POS, MAXLEN, EOS, HEAD);
```

3.30.5 FBD Language



3.30.6 FFLD Language

(* The function is executed only if EN is TRUE *)
(* ENO keeps the same value as EN *)



3.30.7 IL Language:

Not available

3.31 SERIO

Function Block - Serial communication.

3.31.1 Inputs

RUN : BOOL	Enable communication (opens the comm port)
SND : BOOL	TRUE if data has to be sent
CONF : STRING	Configuration of the communication port
DATASND : STRING	Data to send

3.31.2 Outputs

OPEN : BOOL	TRUE if the communication port is open
RCV : BOOL	TRUE if data has been received
ERR : BOOL	TRUE if error detected during sending data
DATARCV : STRING	Received data

3.31.3 Remarks

The RUN input does not include an edge detection. The block tries to open the port on each call if RUN is TRUE and if the port is still not successfully open. The CONF input is used for settings when opening the port. Please refer to your OEM instructions for further details about possible parameters.

The SND input does not include an edge detection. Characters are sent on each call if SND is TRUE and DATASND is not empty.

The DATARCV string is erased on each cycle with received data (if any). Your application is responsible for analyzing or storing received character immediately after the call to SERIO block.

SERIO is available during simulation. In that case, the CONF input defines the communication port according to the syntax of the "MODE" command. For example:

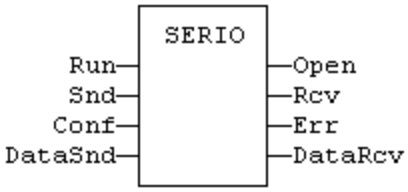
```
'COM1:9600,N,8,1'
```

The SERIO block may not be supported on some targets. Refer to your OEM instructions for further details.

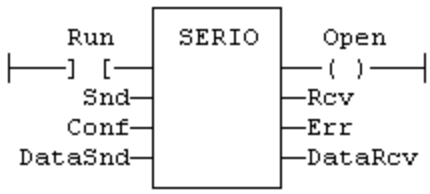
3.31.4 ST Language

```
(* MySer is a declared instance of SERIO function block *)
MySer (RUN, SND, CONF, DATASND);
OPEN := MySer.OPEN;
RCV := MySer.RCV;
ERR := MySer.ERR;
DATARCV := MySer.DATARCV;
```

3.31.5 FBD Language



3.31.6 FFLD Language



3.31.7 IL Language:

```
(* MySer is a declared instance of SERIO function block *)
Op1: CAL   MySer (RUN, SND, CONF, DATASND)
FFLD   MySer.OPEN
ST   OPEN
FFLD   MySer.RCV
ST   RCV
FFLD   MySer.ERR
ST   ERR
FFLD   MySer.DATARCV
ST   DATARCV
```

3.32 SigID

Function - Get the identifier of a "Signal" resource

3.32.1 Inputs

`SIGNAL : STRING` Name of the signal resource - *must be a constant value!*
`COL : STRING` Name of the column within the signal resource - *must be a constant value!*

3.32.2 Outputs

`ID : DINT` ID of the signal - to be passed to other blocks

3.32.3 Remarks

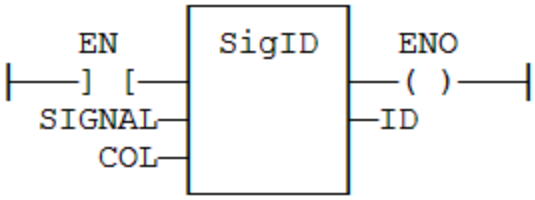
Some blocks have arguments that refer to a "signal" resource. For all these blocks, the signal argument is materialized by a numerical identifier. This function enables you to get the identifier of a signal defined as a resource.

3.32.4 ST Language

```
ID := SigID ('MySignal', 'FirstColumn');
```

3.32.5 FBD Language

3.32.6 FFLD Language



3.32.7 IL Language

```
Op1: FFLD    'MySignal'
   SigID 'FirstColumn'
   ST   ID
```

See also

SigPlay SigScale

3.33 SigPlay

Function block - Generate a signal defined in a resource

3.33.1 Inputs

`IN : BOOL` Triggering command
`ID : DINT` ID of the signal resource, provided by SigID function
`RST : BOOL` Reset command
`TM : TIME` Minimum time in between two changes of the output

3.33.2 Outputs

Q : BOOL	TRUE when the signal is finished
OUT : REAL	Generated signal
ET : TIME	Elapsed time

3.33.3 Remarks

The "ID" argument is the identifier of the "signal" resource. Use the SigID function to get this value.

The "IN" argument is used as a "Play / Pause" command to play the signal. The signal is not reset to the beginning when IN becomes FALSE. Instead, use the "RST" input that resets the signal and forces the OUT output to 0.

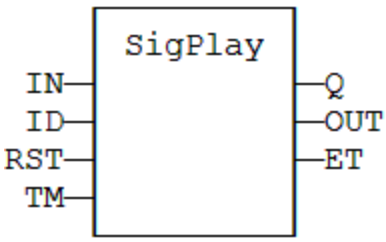
The "TM" input specifies the minimum amount of time in between two changes of the output signal. This parameter is ignored if less than the cycle scan time.

This function block includes its own timer. Alternatively, you can use the SigScale function if you want to trigger the signal using a specific timer.

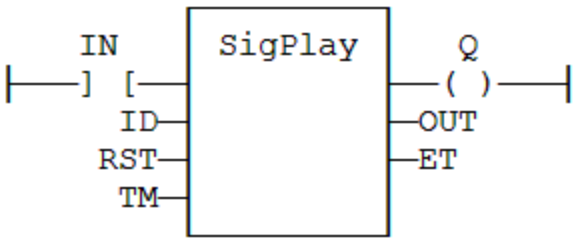
3.33.4 ST Language

```
Q := SigScale (ID, IN);
```

3.33.5 FBD Language



3.33.6 FFLD Language



3.33.7 IL Language

```
Op1: FFLD      IN
SigScale ID
ST     Q
```

See also

SigScale SigID

3.34 SigScale

Function - Get a point from a "Signal" resource

3.34.1 Inputs

`ID : DINT` ID of the signal resource, provided by SigID function
`IN : TIME` Time (X) coordinate of the wished point within the signal resource

3.34.2 Outputs

`Q : REAL` Value (Y) coordinate of the point in the signal

3.34.3 Remarks

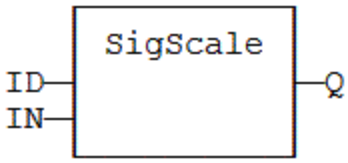
The "ID" argument is the identifier of the "signal" resource. Use the SigID function to get this value.

This function converts a time value to a analog value such as defined in the signal resource. This function can be used instead of SigPlay function block if you want to trigger the signal using a specific timer.

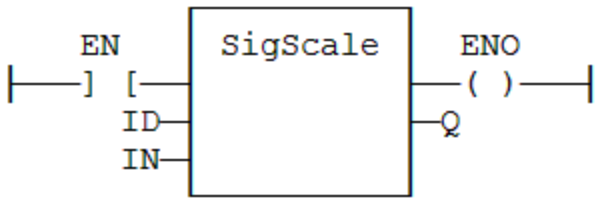
3.34.4 ST Language

Q := SigScale (ID, IN);

3.34.5 FBD Language



3.34.6 FFLD Language



3.34.7 IL Language

```
Op1: FFLD      IN
SigScale ID
ST     Q
```

See also

SigPlay SigID

3.35 STACKINT

Function Block - Manages a stack of DINT integers.

3.35.1 Inputs

`PUSH : BOOL` Command: when changing from FALSE to TRUE, the value of IN is pushed on the stack
`POP : BOOL` Pop command: when changing from FALSE to TRUE, deletes the top of the stack
`R1 : BOOL` Reset command: if TRUE, the stack is emptied and its size is set to N
`IN : DINT` Value to be pushed on a rising pulse of PUSH
`N : DINT` maximum stack size - cannot exceed 128

3.35.2 Outputs

`EMPTY : BOOL` TRUE if the stack is empty
`OFLO : BOOL` TRUE if the stack is full
`OUT : DINT` value at the top of the stack

3.35.3 Remarks

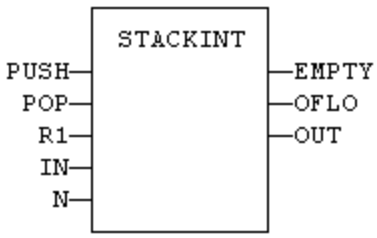
Push and pop operations are performed on rising pulse of PUSH and POP inputs. In FFLD language, the input rung is the PUSH command. The output rung is the EMPTY output.

The specified size (N) is taken into account only when the R1 (reset) input is TRUE.

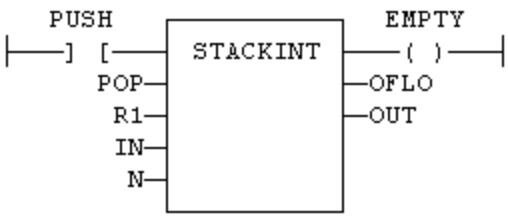
3.35.4 ST Language

```
(* MyStack is a declared instance of STACKINT function block *)
MyStack (PUSH, POP, R1, IN, N);
EMPTY := MyStack.EMPTY;
OFLO := MyStack.OFLO;
OUT := MyStack.OUT;
```

3.35.5 FBD Language



3.35.6 FFLD Language



3.35.7 IL Language

```
(* MyStack is a declared instance of STACKINT function block *)
Op1: CAL MyStack (PUSH, POP, R1, IN, N)
    FFLD  MyStack.EMPTY
    ST  EMPTY
    FFLD  MyStack.OFLO
    ST  OFLO
    FFLD  MyStack.OUT
    ST  OUT
```

See also

AVERAGE INTEGRAL DERIVATE LIM_ALRM HYSTER

3.36 SurfLin

Function block- Linear interpolation on a surface.

3.36.1 Inputs

X : REAL X coordinate of the point to be interpolated.

Y : REAL Y coordinate of the point to be interpolated.

XAxis : REAL[] X coordinates of the known points of the X axis.

YAxis : REAL[] Y coordinates of the known points of the Y axis.

ZVal : REAL[,] Z coordinate of the points defined by the axis.

3.36.2 Outputs

Z : REAL Interpolated Z value corresponding to the X,Y input point

OK : BOOL TRUE if successful.

ERR : DINT Error code if failed - 0 if OK.

3.36.3 Remarks

This function performs linear surface interpolation in between a list of points defined in XAxis and YAxis single dimension arrays. The output Z value is an interpolation of the Z values of the four rounding points defined in the axis. Z values of defined points are passed in the ZVal matrix (two dimension array).

ZVal dimensions must be understood as: ZVal [iX , iY]

Values in X and Y axis must be sorted from the smallest to the biggest. There must be at least two points defined in each axis. ZVal must fit the dimension of XAxis and YAxis arrays. For instance:

XAxis : ARRAY [0..2] of REAL;

YAxis : ARRAY [0.3] of REAL;

ZVal : ARRAY [0..2,0..3] of REAL;

In case the input point is outside the rectangle defined by XAxis and YAxis limits, the Z output is bound to the corresponding value and an error is reported.

The ERR output gives the cause of the error if the function fails:

Error Code	Meaning
0	OK

Error Code	Meaning
1	Invalid dimension of input arrays
2	Invalid points for the X axis
3	Invalid points for the Y axis
4	X,Y point is out of the defined axis

3.37 TCP-IP management functions

The following functions enable management of TCP-IP sockets for building client or server applications over ETHERNET network:

`tcpListen`	create a listening server socket
`tcpAccept`	accept client connections
`tcpConnect`	create a client socket and connect it to a server
`tcpIsConnected`	test if a client socket is connected
`tcpClose`	close a socket
`tcpSend`	send characters
`tcpReceive`	receive characters
`tcpIsValid`	test if a socket is valid

Each socket is identified in the application by a unique handle manipulated as a DINT value.

Warning

• Even though the system provides a simplified interface, you must be familiar with the socket interface such as existing in other programming languages such as "C".
• Socket management may be not available on some targets. Please refer to OEM instructions for further details about available features.

tcpListen: create a "listening" server socket

```
SOCK := tcpListen (PORT, MAXCNX);
```

`PORT : DINT`	TCP port number to be attached to the server socket
`MAXCNX : DINT`	maximum number of client sockets that can be accepted
`SOCK : DINT`	ID of the new server socket

This function creates a new socket performs the "bind" and "listen" operations using default TCP settings. You will have to call the `tcpClose` function to release the socket returned by this function.

tcpAccept: accept a new client connection

```
SOCK := tcpAccept (LSOCK);
```

`LSOCK : DINT`	ID of a server socket returned by the `tcpListen` function
`SOCK : DINT`	ID of a new client socket accepted, or invalid ID if no new connection

This functions performs the "accept" operation using default TCP settings. You will have to call the `tcpClose` function to release the socket returned by this function.

tcpConnect: create a client socket and connect it to a server

```
SOCK := tcpConnect (ADDRESS, PORT);
```

`ADDRESS : STRING`	IP address of the remote server
`PORT : DINT`	wished port number on the server
`SOCK : DINT`	ID of the new client socket

This function creates a new socket performs the "connect" operation using default TCP settings and specified server address and port. You will have to call the `tcpClose`function to release the socket returned by this function.

Warning

It is possible that the functions returns a valid socket ID even if the connection to the server is not yet actually performed. After calling this function, you must call `tcpIsConnected` function to know if the connection is ready.

tcpIsConnected: test if a client socket is connected

```
OK := tcpIsConnected (SOCK);
```

```
SOCK : DINT     ID of the client socket
 OK : BOOL      TRUE if connection is correctly established
```

Warning

It is possible that the socket becomes invalid after this function is called, if an error occurs in the TCP connection. You must call the `tcpIsValid`function after calling this function. If the socket is not valid anymore then you must close it by calling `tcpClose`.

tcpClose: release a socket

```
OK := tcpClose (SOCK);
```

```
SOCK : DINT     ID of any socket
 OK : BOOL      TRUE if successful
```

You are responsible for closing any socket created by `tcpListen`, `tcpAccept` or `tcpConnect` functions, even if they have become invalid.

tcpSend: send characters

```
NBSENT := tcpSend (SOCK, NBCHR, DATA);
```

```
SOCK : DINT       ID of a socket
 NBCHAR : DINT    number of characters to be sent
DATA : STRING     string containing characters to send
NBSENT : DINT     number of characters actually sent
```

It is possible that the number of characters actually sent is less than the number expected. In that case, you will have to call again the function on te next cycle to send the pending characters.

Warning

It is possible that the socket becomes invalid after this function is called, if an error occurs in the TCP connection. You must call the `tcpIsValid`function after calling this function. If the socket is not valid anymore then you must close it by calling `tcpClose`.

tcpReceive: receive characters

```
NBRCV := tcpReceive (SOCK, MAXCHR, DATA);
```

```
SOCK : DINT       ID of a socket
 MAXCHR : DINT    maximum number of characters wished
DATA : STRING     string where to store received characters
NBRCV : DINT      number of characters actually received
```

It is possible that the number of characters actually received is less than the number expected. In that case, you will have to call again the function on the next cycle to receive the pending characters.

Warning

It is possible that the socket becomes invalid after this function is called, if an error occurs in the TCP connection. You must call the `tcpIsValid`function after calling this function. If the socket is not valid anymore then you must close it by calling `tcpClose`.

tcpIsValid: test if a socket is valid

```
OK := tcpIsValid (SOCK);
```

```
SOCK : DINT     ID of the socket
 OK : BOOL      TRUE if specified socket is still valid
```

3.38 Text buffers manipulation

Strings are limited to 255 characters. Here is a set of functions and function blocks for working with not limited text buffers. Text buffers are dynamically allocated or re-allocated.

Warning

- There must be one instance of the TxbManager declared in your application for using these functions.
- The application should take care of releasing memory allocated for each buffer. Allocating buffers without freeing them will lead to memory leaks.

The application is responsible for freeing all allocated text buffers. However, all allocated buffers are automatically released when the application stops.

Below are the functions and function blocks for managing variable length text buffers:

Memory management / Miscellaneous:

TxbManager: main gatherer of text buffer data in memory

TxbLastError: get detailed error report about last call

Allocation / exchange with files:

TxbNew: Allocate a new empty buffer

TxbNewString: Allocate a new buffer initialized with string

TxbFree: Release a text buffer

TxbReadFile: Allocate a new buffer from file

TxbWriteFile: Store a text buffer to file

Data exchange:

TxbGetLength: Get length of a text buffer

TxbGetData: Store text contents to an array of characters

TxbGetString: Store text contents to a string

TxbSetData: Store an array of characters to a text buffer

TxbSetString: Store string to text buffer

TxbClear: Empty a text buffer

TxbCopy: Copy a text buffer

Sequential reading:

TxbRewind: Rewind sequential reading

TxbGetLine: Sequential read line by line

Sequential writing:

TxbAppend: Append variable value

TxbAppendLine: Append a text line

TxbAppendEol: Append end of line characters

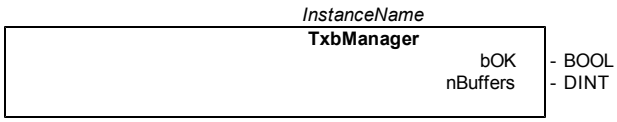
TxpAppendTxb: Append contents of another buffer

UNICODE conversions:

TxbAnsiToUtf8: Convert a text buffer to UNICODE

TxbUtf8ToAnsi: Converts a text buffer to ANSI

3.38.1 TxbManager



Description:

This function block is used for managing the memory allocated for text buffers. It takes care of releasing the corresponding memory when the application stops, and can be used for tracking memory leaks.

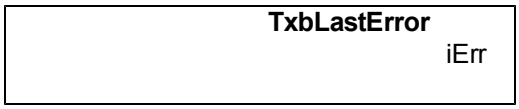
Warning

There must be one and only one instance of this block declared in the IEC application in order to use any other Txb... function.

Outputs:

bOK : BOOL	TRUE if the text buffers memory system is correctly initialized.
nBuffers : DINT	Number of text buffers currently allocated in memory.

TxbLastError



Description:

All TXB functions and blocks simply return a boolean information as a return value. This function can be called after any other function giving a FALSE return. It gives a detailed error code about the last detected error.

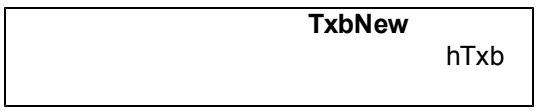
Outputs:

iErr : DINT	Error code reported by the last call: 0 = OK other = error (see below)

Below are possible error codes:

1 invalid instance of TXBManager - should be only one
2 manager already open - should be only one instance of TxbManager
3 manager not open - no instance of TxbManager declared
4 invalid handle
5 string has been truncated during copy
6 cannot read file
7 cannot write file
8 unsupported data type
9 too many text buffers allocated

TxbNew



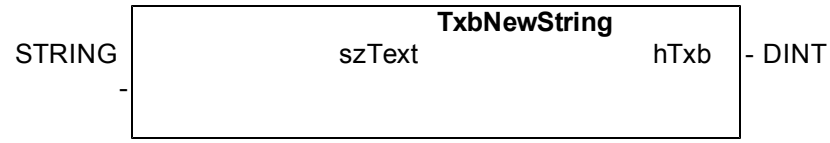
Description:

This function allocates a new text buffer initially empty. The application will be responsible for releasing the buffer by calling the TxbFree() function.

Outputs:

hTxb : DINT	Handle of the new buffer

TxbNewString



Description:

This function allocates a new text buffer initially filled with teh specified string. The application will be responsible for releasing the buffer by calling the TxbFree() function.

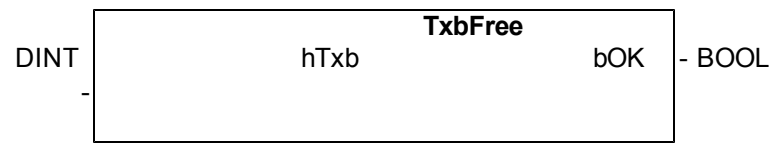
Inputs:

szText : STRING	Initial value of the text buffer

Outputs:

hTxb : DINT	Handle of the new buffer

TxbFree



Description:

This function releases a text buffer from memory.

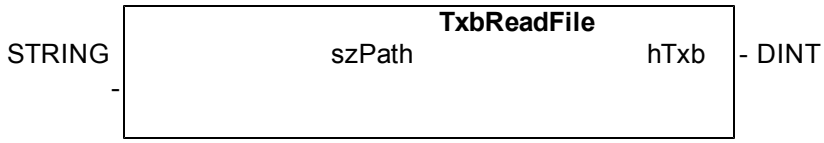
Inputs:

hTxb : DINT	Handle of a valid text buffer

Outputs:

bOK : BOOL	TRUE if successful

TxbReadFile



Description:

This function allocates a new text buffer and fills it with the contents of the specified file. The application will be responsible for releasing the buffer by calling the TxbFree() function.

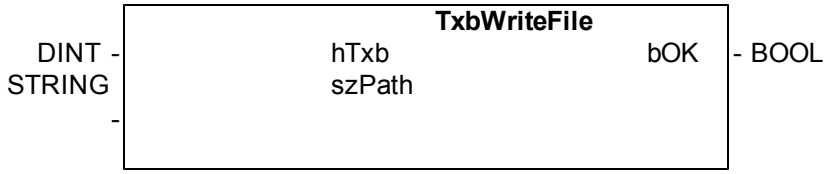
Inputs:

szPath : STRING	Full qualified path name of the file to be read

Outputs:

hTxb : DINT	Handle of the new buffer

TxbWriteFile



Description:

This function stores the contents of a text buffer to a file. The text buffer remains allocated in memory.

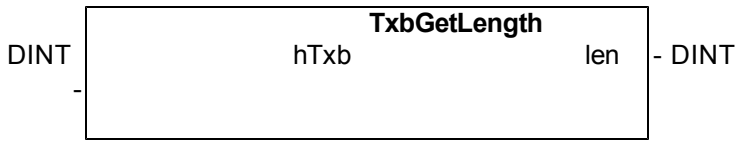
Inputs:

hTxb : DINT	Handle of the text buffer
szPath : STRING	Full qualified path name of the file to be created.

Outputs:

bOK : BOOL	TRUE if successful

TxbGetLength



Description:

This function returns the current length of a text buffer.

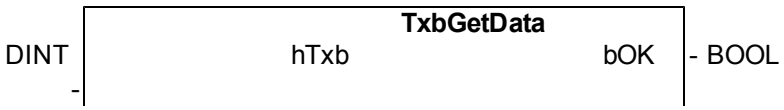
Inputs:

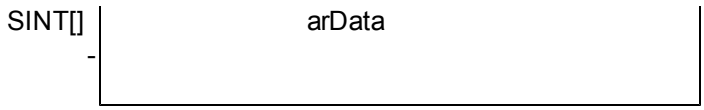
hTxb : DINT	Handle of the text buffer

Outputs:

len : DINT	Number of characters in the text buffer

TxbGetData





Description:

This function copies the contents of a text buffer to an array of characters.

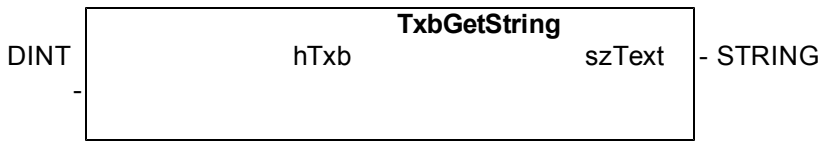
Inputs:

hTxb : DINT	Handle of the text buffer
arData : SINT[]	Array of characters to be filled with text

Outputs:

bOK : BOOL	TRUE if successful

TxbGetString



Description:

This function copies the contents of a text buffer to a string. The text is truncated if the string is not large enough.

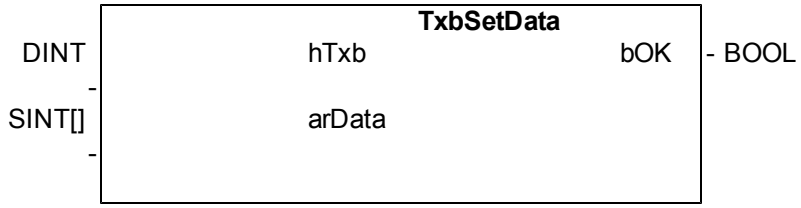
Inputs:

hTxb : DINT	Handle of the text buffer

Outputs:

szText : STRING	String to be filled with text

TxbSetData



Description:

This function copies an array of characters to a text buffer. All characters of the input array are copied.

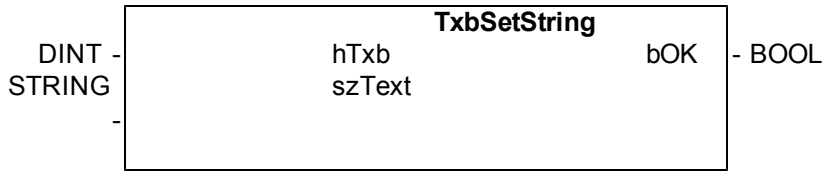
Inputs:

hTxb : DINT	Handle of the text buffer
arData : SINT[]	Array of characters to copy

Outputs:

bOK : BOOL	TRUE if successful

TxbSetString



Description:

This function copies the contents of a string to a text buffer.

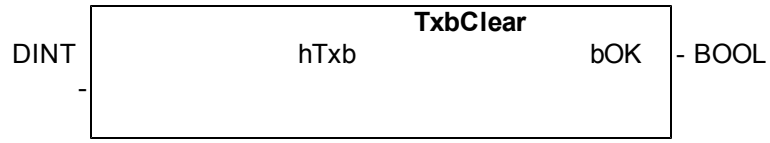
Inputs:

hTxb : DINT	Handle of the text buffer
szText : STRING	String to be copied

Outputs:

bOK : BOOL	TRUE if successful

TxbClear



Description:

This function empties a text buffer.

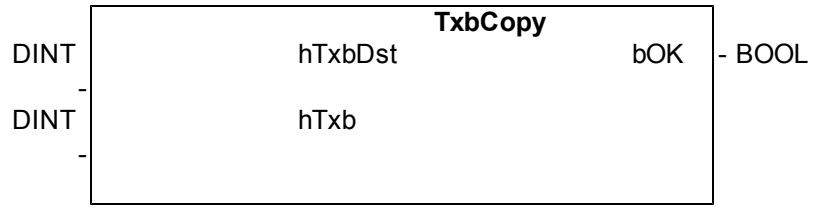
Inputs:

hTxb : DINT	Handle of the text buffer

Outputs:

bOK : BOOL	TRUE if successful

TxbCopy



Description:

This function copies the contents of the hTxb buffer the to hTxbDst buffer.

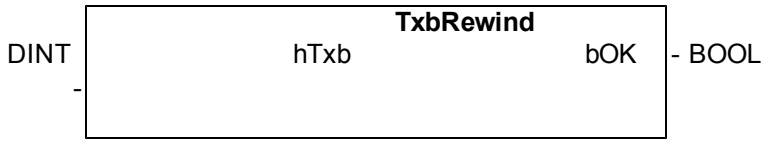
Inputs:

hTxbDst : DINT	Handle of the destination text buffer
hTxb : DINT	Handle of the source text buffer

Outputs:

bOK : BOOL	TRUE if successful

TxbRewind



Description:

This function resets the sequential reading of a text buffer (rewind to the beginning of the text).

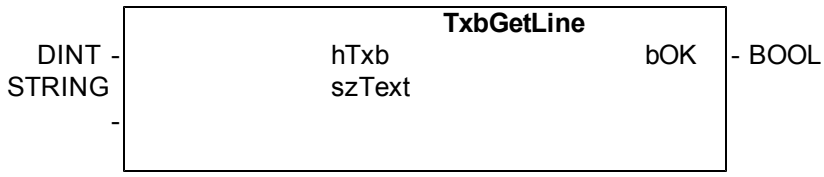
Inputs:

hTxb : DINT	Handle of the text buffer

Outputs:

bOK : BOOL	TRUE if successful

TxbGetLine



Description:

This function sequentially reads a line of text from a text buffer. End of line characters are not copied to the output string.

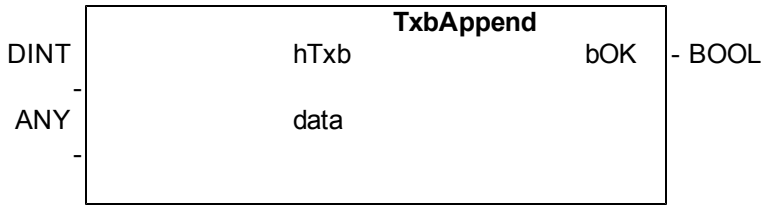
Inputs:

hTxb : DINT	Handle of the text buffer
szText : STRING	String to be filled with read line

Outputs:

bOK : BOOL	TRUE if successful

TxbAppend



Description:

This function adds the contents of a variable, formatted as text, to a text buffer. The specified variable can have any data type.

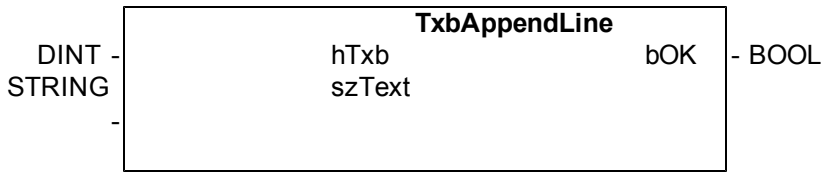
Inputs:

hTxb : DINT	Handle of the text buffer
data : ANY	Any variable

Outputs:

bOK : BOOL	TRUE if successful

TxbAppendLine



Description:

This function adds the contents of the specified string variable to a text buffer, plus end of line characters.

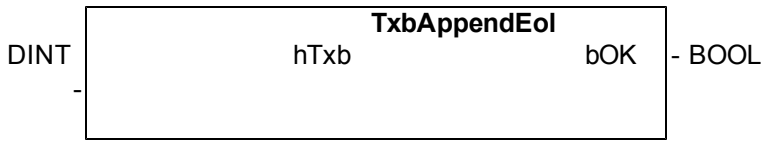
Inputs:

hTxb : DINT	Handle of the text buffer
szText : STRING	String to be added to the text

Outputs:

bOK : BOOL	TRUE if successful

TxbAppendEol



Description:

This function adds end of line characters to a text buffer.

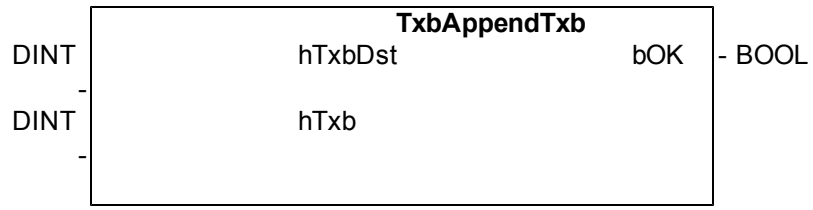
Inputs:

hTxb : DINT	Handle of the text buffer

Outputs:

bOK : BOOL	TRUE if successful

TxbAppendTxb



Description:

This function adds the contents of the "hTxb" text buffer to the "hTxbDst" text buffer.

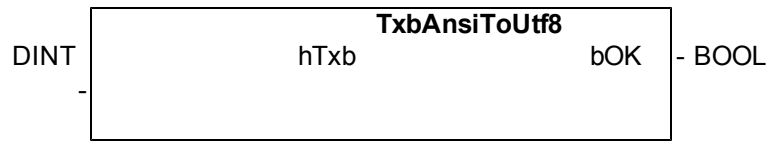
Inputs:

hTxbDst : DINT	Handle of the text buffer to be completed
hTxb : DINT	Handle of the text buffer to be added

Outputs:

bOK : BOOL	TRUE if successful

TxbAnsiToUtf8



Description:

This function converts the whole contents of a text buffer from ANSI to UNICODE UTF8 encoding.

Warning

This function may be time and memory consuming for large buffers.

Warning

UNICODE conversion may be not available on some operating systems

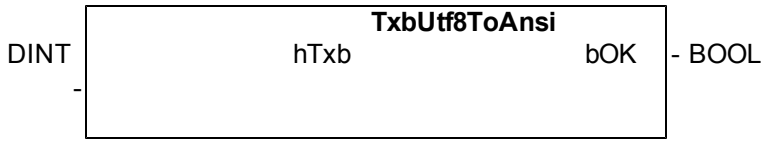
Inputs:

hTxb : DINT — Handle of the text buffer

Outputs:

bOK : BOOL — TRUE if successful

TxbUtf8ToAnsi



Description:

This function converts the whole contents of a text buffer from UNICODE UTF8 to ANSI encoding.

Warning

This function may be time and memory consuming for large buffers.

Warning

UNICODE conversion may be not available on some operating systems

Inputs:

hTxb : DINT — Handle of the text buffer

Outputs:

bOK : BOOL — TRUE if successful

3.39 UDP management functions

The following functions enable management of UDP sockets for building client or server applications over ETHERNET network:

`udpCreate`	create a UDP socket
`udpAddrMake`	build an address buffer for UDP functions
`updSendTo`	send a telegram
`udpRcvFrom`	receive a telegram
`udpClose`	close a socket
`udpIsValid`	test if a socket is valid

Each socket is identified in the application by a unique handle manipulated as a DINT value.

Warning

• Even though the system provides a simplified interface, you must be familiar with the socket interface such as existing in other programming languages such as "C".
• Socket management may be not available on some targets. Please refer to OEM instructions for further details about available features.

udpCreate: create a UDP socket

```
SOCK := udpCreate (PORT);
```

`PORT : DINT`	TCP port number to be attached to the server socket or 0 for a client socket
`SOCK : DINT`	ID of the new server socket

This function creates a new UDP socket. If the PORT argument is not 0, the socket is bound to the port and thus can be used as a server socket.

udpAddrMake: build an address buffer for UDP functions

```
OK := udpAddrMake (IPADDR, PORT, ADD);
```

`IPADDR : STRING`	IP address in form xxx.xxx.xxx.xxx
`PORT : DINT`	IP port number
`ADD : USINT[32]`	buffer where to store the UDP address (filled on output)
`OK : BOOL`	TRUE if successful

This functions is required for building a internal "UDP" address to be passed to the `udpSendTo`function in case of UDP client processing.

udpSendTo: send a UDP telegram

```
OK := udpSendTo (SOCK, NB, ADD, DATA);
```

`SOCK : DINT`	ID of the client socket
`NB : DINT`	number of characters to send
`ADD : USINT[32]`	buffer containing the UDP address (on input)
`DATA : STRING`	characters to send
`OK : BOOL`	TRUE if successful

The "ADD" buffer must contain a valid UDP address either constructed by the `udpAddrMake` function or returned by the `udpRcvFrom`function.

udpRcvFrom: receive a UDP telegram

```
OK := udpRcvFrom (SOCK, NB, ADD, DATA);
```

`SOCK : DINT`	ID of the client socket
`NB : DINT`	maximum number of characters received
`ADD : USINT[32]`	buffer containing the UDP address of the transmitter (filled on

output)

```
DATA : STRING     buffer where to store received characters
 Q : DINT           number of actually received characters
```

If characters are received, the function fills the `ADD` argument with the internal "UDP" of the sender. This buffer can then be passed to the `udpSendTo`function to send the answer.

udpClose: release a socket

```
OK := udpClose (SOCK);
```

```
SOCK : DINT     ID of any socket
 OK : BOOL       TRUE if successful
```

You are responsible for closing any socket created by `tcpListen`, `tcpAccept` or `tcpConnect` functions, even if they have become invalid.

udpIsValid: test if a socket is valid

```
OK := udpIsValid (SOCK);
```

```
SOCK : DINT     ID of the socket
 OK : BOOL       TRUE if specified socket is still valid
```

3.40 VLID

Function - Get the identifier of an embedded list of variables

3.40.1 Inputs

```
FILE : STRING     Path name of the .TXT list file - must be a constant value!
```

3.40.2 Outputs

```
ID : DINT         ID of the list - to be passed to other blocks
```

3.40.3 Remarks

Some blocks have arguments that refer to a list of variables. For all these blocks, the "list" argument is materialized by a numerical identifier. This function enables you to get the identifier of a list of variables.

Embedded lists of variables are simple ".TXT" text files with one variable name per line (note that you can only declare global variable).

Lists must contain single variables only. Items of arrays and structures must be specified one by one. The length of the list is not limited by the system.

Warning

List files are read at compiling time and are embedded into the downloaded application code. This implies that a modification performed in the list file after downloading will not be taken into account by the application.

3.40.4 ST Language

```
ID := VLID ('MyFile.txt');
```

3.40.5 FBD Language

3.40.6 FFLD Language

(* The function is executed only if EN is TRUE *)

3.40.7 IL Language

```
Op1: FFLD  'MyFile.txt'
VLID COL
ST   ID
```

This page intentionally left blank.

Global Support Contacts

Danaher Motion Assistance Center

Phone: 1-540-633-3400
Fax: 1-540-639-4162
Email: contactus@danahermotion.com

Danaher Motion
203A West Rock Road
Radford, VA 24141 USA

Europe Product Support

France

- Linear Units
- Ball- & Leadscrews
- Actuators
- Gearheads
- Rails & Components
- Servo Motors & Direct Drives
- Servo Drives & High Frequency Inverters
- Machine & Motion Controls

Tel.: +33 (0)243 5003-30
Fax: +33 (0)243 5003-39
Email: sales.france@tollo.com

Germany

- Gearheads
- Servo Motors & Direct Drives
- Servo Drives & High Frequency Inverters
- Machine & Motion Controls

Tel.: +49 (0)2102 9394-0
Fax: +49 (0)2102 - 9394-3155
Email: technik@kollmorgen.com

- Ball- & Leadscrews
- Linear Units
- Actuators
- Rails & Components

Tel.: +49 (0)70 22 504-0
Fax: +49 (0)70 22 54-168
Email: sales.wolfschlugen@danahermotion.com

Italy

- Ball- & Leadscrews
- Linear Units
- Actuators

- Rails & Components
- Servo Motors & Direct Drives
- Servo Drives & High Frequency Inverters
- Machine & Motion Controls

Tel.: +39 0362 5942-60
Fax: +39 0362 5942-63
Email: info@danahermotion.it

Sweden

- Ball- & Leadscrews
- Linear Units
- Actuators
- Gearheads
- Rails & Components
- Servo Motors & Direct Drives
- Servo Drives & High Frequency Inverters
- Machine & Motion Controls

Tel.: +46 (0)44 24 67-00
Fax: +46 (0)44 24 40-85
Email: helpdesk.kid@danahermotion.com

Switzerland

- Servo Motors & Direct Drives
- Servo Drives & High Frequency Inverters
- Machine & Motion Controls

Tel. : +41 (0)21 6313333
Fax: +41 (0)21 6360509
Email: info@danaher-motion.ch

- Miniature Motors

Tel.: +41 (0)32 9256-111
Fax: +41 (0)32 9256-596
Email: info@portescap.com

United Kingdom / Ireland

- Ball- & Leadscrews
- Linear Units
- Actuators
- Gearheads
- Rails & Components
- Servo Motors & Direct Drives
- Servo Drives & High Frequency Inverters
- Machine & Motion Controls

Tel.: +44 (0)1525 243-243
Fax: +44 (0)1525 243-244
Email: sales.uk@danahermotion.com