Proficy* Logic Developer - PLC

Structured Text (ST) Language

Proficy Machine Edition 8.0

2013

Notice

GE Intelligent Platforms, Inc. reserves the right to make improvements to the products described in this publication at any time and without notice.

We want to hear from you. If you have any comments, questions, or suggestions about our documentation, send them to the following email address:
doc@ge.com

Disclaimer of Warranties and Liability

The information contained in this manual is believed to be accurate and reliable. However, GE Intelligent Platforms, Inc. assumes no responsibilities for any errors, omissions or inaccuracies whatsoever. Without limiting the foregoing, GE Intelligent Platforms, Inc. disclaims any and all warranties, expressed or implied, including the warranty of merchantability and fitness for a particular purpose, with respect to the information contained in this manual and the equipment or software described herein. The entire risk as to the quality and performance of such information, equipment and software, is upon the buyer or user. GE Intelligent Platforms, Inc. shall not be liable for any damages, including special or consequential damages, arising out of the use of such information, equipment and software, even if GE Intelligent Platforms, Inc. has been advised in advance of the possibility of such damages. The use of the information contained in the manual and the software described herein is subject to GE Intelligent Platforms, Inc. standard license agreement, which must be executed by the buyer or user before the use of such information, equipment or software.

Table Of Contents

ST Logic: an Overview

Structured Text logic is a subset of the ST programming language specified by the IEC 61131-3 standard.
ST logic is written as a series of statements. There are various kinds of statements:

- Expressions use operators to perform calculations on variables, parameters and/or constants, and then assign the results to variables.
- Call statements (function call, block call).
- Function Block invocation statements of UDFBs.
- Comments.
- Loop statements (repeat, while).
- Conditional statements (if).
- Control statements (exit, return).

For more information, see Statements .

Data Types

For ST operators, there is strict data type checking; that is, all operands in an ST statement must be the same data type.
For ST functions, if the Type Mismatch Warning option is set to "Show as Warning" and there is a data type mismatch, then upon validation, a warning displays in the Feedback Zone. Using mixed data types in a function can cause unexpected results. For example, if a REAL result is assigned to a DINT variable, the resulting value is copied into the DINT variable without being converted to a DINT data type. Because REAL data types are stored in a different format than DINT data types, the resulting value of the DINT variable may be unexpected.

Tip: If a REAL result is to be placed into a DINT variable, use the REAL_to_DINT conversion function.

In Logic Developer - PLC, ST supports the following data types:

- BOOL (boolean or discrete)
- BYTE (8-bit bit string)
- INT (16-bit integer)
- UINT (16-bit unsigned integer)
- WORD (16-bit bit string)
- DINT (32-bit integer)
- REAL (32-bit floating-point value)
- DWORD (32-bit bit string)
- LREAL (64-bit floating-point value)

Structure variables and arrays can also be used.

Parameters

The two types of parameters in ST logic are:

- ENO (ENable Output, also named Y0). This output parameter is available for every block, function, or function block that is called or invoked. ENO can be set

inside the logic of any block, function, or function block. It may be accessed during a block call, function call, or function block call if the formal call convention is used.

- Parameters from an ST parameterized block, ST user-defined function block (UDFB), or ST standard function block.

  Note: If you attempt to pass read-only memory to an operand that may be written to during the execution of the call, an error appears in the Feedback Zone during validation. This occurs when a system variable (in %S memory and the special case of #FST_EXE) or a UDFB output is used outside of its UDFB and is passed as an argument to such a parameter. A parameter is input or output passed by reference or by value result.

To use a parameter in ST logic, add the parameter to the UDFB, parameterized block, or standard function block being edited. The parameter or an element of the parameter (if its length is greater than 1) (see second bullet of the Notes below) can then be used in ST logic anywhere a variable is used.

Notes

- Parameters cannot use indirect references as operators.
- To access an element of a parameter, use the [*n*] notation, where *n* is a 0-based constant. An element can include elements of a parameter that is function block instance data, as well as a parameter that is an array, which you list here.
- If ENO is not set within a block, function, or function block, it defaults to 1 (True) when the block, function, or function block is called or invoked.

Bit References

Bit references can be used with variables, elements of arrays, elements of structure variables, and parameters if the variables or elements are BYTE, WORD, INT, UINT, DINT, or DWORD. For example:

- MyWordArray[5].X[2] references the third bit of the sixth element of the array MyWordArray if MyWordArray is a one-dimensional array of at least 6 elements.
- MyStructure.Counter.X[3] references the fourth bit of the Counter element of the structure variable MyStructure.
- MyParam[2].X[4] references the fifth bit of the second element of the parameter MyParam if MyParam has a length of at least two.

Note: Bits of variables and parameters are 0-based, for example, myVar.X[0] and myParam.X[0].

Order of Operations

The evaluation of an expression consists of applying the operators to the operands in a sequence defined by the operator precedence shown in the table below. The operator with highest precedence in an expression is applied first, followed by the operator of next lower precedence, and so on, until evaluation is complete. Operators of equal precedence are applied as written in the expression from left to right. For example, if A, B, C, and D are of data type INT with values 1, 2, 3, and 4, respectively, then:

```
A+B-C*D

// The above expression evaluates to -9

(A+B-C)*D

// The above expression evaluates to 0
```

When an operator has two operands, the leftmost operand is evaluated first. For example, in the expression

```
(A > B) OR (C < D)

// The expression (A > B) is evaluated first, followed by (C < D), followed by evaluation of the boolean OR.
```

Number	Operation	Symbol	Precedence
1	Parenthesization	(expression)	Highest
2	Negation	-	
3	Complement	not	
4	Exponentiation	**, ^	
5	Multiply	*	
6	Divide	/	
7	Modulo	mod	
8	Add	+	
9	Subtract	-	
10	Comparison	< , > , <= , >=	
11	Equality	=	
12	Inequality	<>, !=	
13	Boolean AND	&	
14	Boolean AND	and	
15	Boolean Exclusive OR	xor	
16	Boolean OR	or	Lowest

ST Instructions: Statements, Functions, Operators, and Standard Function Blocks

A Structured Text block is composed of ST statements. You insert statements in the ST editor to create simple executable units of Logic Developer - PLC logic. Each statement performs one or more operations on parameters, constants, and variables defined for the target the ST logic is associated with.

- Advanced Math Functions
- Advanced Math Operators
- Math Functions
- Math Operators
- Bitwise Operators
- Control Functions
- Conversion Functions
- Data Move Functions
- PACMotion Instructions and Function Blocks
- Relational Operators
- Statements
- Timers (Standard Function Blocks)

Communication instructions are described in the Addendum of this manual, after the index.

PACMotion instructions and function blocks are described at length in both GFK-2448 and online help.

Advanced Math Functions

(For PACSystems firmware version 2.50 and later only.)
ST advanced math *functions* perform advanced math operations on data. They support operations on DINT, INT, LREAL, and REAL variables. If the data type mismatch preference is set to "Show as Warning" and there is a data type mismatch, a warning displays in the Feedback Zone.
Using mixed data types in a function can cause unexpected results. For example, if a REAL result is assigned to a DINT variable, the resulting value is copied into the DINT variable without being converted to a DINT data type. Because REAL data types are stored in a different format than DINT data types, the resulting value of the DINT variable may be unexpected.

Tip: If a REAL result is to be placed into a DINT variable, use the REAL_to_DINT conversion function.

Function	Mnemonics	Examples
Exponential	expt	expt(IN1 := inReal1, IN2 := inReal2, Q => outReal, ENO => outBool);
	expt_real	expt_real(IN1 := inReal1, IN2 := inReal2, Q => outReal, ENO => outBool);
	expt_lreal	expt_lreal(IN1 := inLreal1, IN2 := inLreal2, Q => outLreal, ENO => outBool);
Inverse natural log	exp	exp(IN := inReal, Q => outReal, ENO => outBool);
	exp_real	exp_real(IN := inReal, Q => outReal, ENO => outBool);
	exp_lreal	exp_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
Inverse trigonometry	acos	acos(IN := inReal, Q => outReal, ENO => outBool);
	asin	asin(IN := inReal, Q => outReal, ENO => outBool);
	atan	atan(IN := inReal, Q => outReal, ENO => outBool);
	acos_real	cos_real(IN := inReal, Q => outReal, ENO => outBool);
	asin_real	sin_real(IN := inReal, Q => outReal, ENO => outBool);
	atan_real	tan_real(IN := inReal, Q => outReal, ENO => outBool);
	acos_lreal	cos_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
	asin_lreal	sin_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

	atan_lreal	tan_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
Logarithmic	log	log(IN := inReal, Q => outReal, ENO => outBool);
	ln	ln(IN := inReal, Q => outReal, ENO => outBool);
	log_real	log_real(IN := inReal, Q => outReal, ENO => outBool);
	ln_real	ln_real(IN := inReal, Q => outReal, ENO => outBool);
		log_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
	log_lreal	ln_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
	ln_lreal	
Square root	sqrt_dint	sqrt_dint(IN := inDint, Q => outDint, ENO => outBool);
	sqrt_int	sqrt_int(IN := inInt, Q => outInt, ENO => outBool);
	sqrt_lreal	sqrt_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
	sqrt_real	sqrt_real(IN := inReal, Q => outReal, ENO => outBool);
Trigonometry	cos	cos(IN := inReal, Q => outReal, ENO => outBool);
	sin	sin(IN := inReal, Q => outReal, ENO => outBool);
	tan	tan(IN := inReal, Q => outReal, ENO => outBool);
		cos_real(IN := inReal, Q => outReal, ENO => outBool);
	cos_real	sin_real(IN := inReal, Q => outReal, ENO => outBool);
	sin_real	tan_real(IN := inReal, Q => outReal, ENO => outBool);
	tan_real	cos_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
		sin_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
	cos_lreal	tan_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
	sin_lreal	
	tan_lreal	

Exponential

(PACSystems firmware version 2.50 or later; to process LREALs: 5.50 or later.)

Syntax

EXPT_REAL has replaced EXPT, which is supported for backward compatibility.
Formal convention:

```
expt(IN1 := inReal1, IN2 := inReal2, Q => outReal, ENO => outBool);

expt_real(IN1 := inReal1, IN2 := inReal2, Q => outReal, ENO => outBool);

expt_lreal(IN1 := inLreal1, IN2 := inLreal2, Q => outLreal, ENO => outBool);
```

Informal convention:

```
expt(inReal1, inReal2, outReal);

expt_real(inReal1, inReal2, outReal);

expt_lreal(inLreal1, inLreal2, outLreal);
```

Operation

The ST exponent *function* calculates the variable assigned to the input operand IN1 to the power of the variable assigned to the input operand IN2, and places the result ($IN1^{IN2}$) in the variable assigned to the output operand Q.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

Notes

- If the result of an expression is greater than the maximum or less than the minimum, then the result is platform dependent.
- For ST and LD functions that are similar, for example, the ST expt function and the LD expt function, an underflow or an overflow result may not be the same.

For details on valid operands, data types, memory areas, and examples, see the expt function in LD Help.

Note: The ** or ^ exponential operator does not support data types in the same way as the expt function.

Inverse Natural Log (exp)

(PACSystems firmware version 2.50 or later; to process LREALs: 5.50 or later.)

Syntax

EXP_REAL has replaced EXP, which is supported for backward compatibility.
Formal convention:

```
exp(IN := inReal, Q => outReal, ENO => outBool);

exp_real(IN := inReal, Q => outReal, ENO => outBool);

exp_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
```

Informal convention:

```
exp(inReal, outReal);

exp_real(inReal, outReal);

exp_lreal(inLreal, outLreal);
```

Operation

The ST inverse natural log (exp) *function* raises ***e*** to the power specified by the variable (*inReal* or *inLreal*) assigned to the input operand IN and places the result in the variable (*outReal* or *outLreal*) assigned to the output operand Q. Exp does not change the original REAL or LREAL data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the EXP instruction in LD Help.

Inverse Trigonometry

(PACSystems firmware version 2.50 or later; to process LREALs: 5.50 or later.)

Syntax

ACOS_REAL, ASIN_REAL, and ATAN_REAL have respectively replaced ACOS, ASIN, and ATAN, which are supported for backward compatibility.
Formal convention:

```
acos(IN := inReal, Q => outReal, ENO => outBool);

asin(IN := inReal, Q => outReal, ENO => outBool);

atan(IN := inReal, Q => outReal, ENO => outBool);


acos_real(IN := inReal, Q => outReal, ENO => outBool);

asin_real(IN := inReal, Q => outReal, ENO => outBool);

atan_real(IN := inReal, Q => outReal, ENO => outBool);


acos_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

asin_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

atan_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
```

Informal convention:

```
acos(inReal, outReal);

asin(inReal, outReal);

atan(inReal, outReal);


acos_real(inReal, outReal);

asin_real(inReal, outReal);

atan_real(inReal, outReal);


acos_lreal(inLreal, outLreal);

asin_lreal(inLreal, outLreal);

atan_lreal(inLreal, outLreal);
```

Operation

An ST inverse trigonometry *function* (acos, asin, and atan) respectively returns the inverse cosine, inverse sine, and inverse tangent of the input variable (*inReal* or *inLreal*) assigned to the input operand IN. The result in radians is stored in the variable (*outReal* or *outLreal*) assigned to the output operand Q. Acos, asin, and atan do not change the original REAL or LREAL data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	**LD function receives power**	**Equivalent ST called function**
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the Inverse Cosine, Inverse Sine, and Inverse Tangent in LD Help.

Logarithmic

(PACSystems firmware version 2.50 or later; to process LREALs: 5.50 or later.)

Syntax

LOG_REAL and LN_REAL have respectively replaced LOG and LN, which are supported for backward compatibility.
Formal convention:

```
log(IN := inReal, Q => outReal, ENO => outBool);

ln(IN := inReal, Q => outReal, ENO => outBool);


log_real(IN := inReal, Q => outReal, ENO => outBool);

ln_real(IN := inReal, Q => outReal, ENO => outBool);


log_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

ln_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
```

Informal convention:

```
log(inReal, outReal);

ln(inReal, outReal);


log_real(inReal, outReal);

ln_real(inReal, outReal);


log_lreal(inLreal, outLreal);

ln_lreal(inLreal, outLreal);
```

Operations

- The ST base 10 logarithm (log) *function* places the base 10 logarithm of the variable (*inReal* or *inLreal*) assigned to the input operand IN, in the variable (*outReal* or *outLreal*) assigned to the output operand Q.

- The ST natural logarithm (ln) function places the natural logarithm of the variable (*inReal* or *inLreal*) assigned to the input operand IN, in the variable (*outReal* or *outLreal*) assigned to the output operand Q.
- Log and ln do not change the original REAL or LREAL data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	**LD function receives power**	**Equivalent ST called function**
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the logarithmic instructions in LD Help.

Square Root

(PACSystems firmware version 2.50 or later; for Sqrt_LREAL: 5.50 or later.)

Syntax

Formal convention:

```
sqrt_dint(IN := inDint, Q => outDint, ENO => outBool);

sqrt_int(IN := inInt, Q => outInt, ENO => outBool);

sqrt_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

sqrt_real(IN := inReal, Q => outReal, ENO => outBool);
```

Informal convention:

```
sqrt_dint(inDint, outDint);

sqrt_int(inInt, outInt);

sqrt_lreal(inLreal, outLreal);

sqrt_real(inReal, outReal);
```

Operation

An ST square root *function* returns the square root of the input variable (*inDint*, *inInt*, *inLreal*, *inReal*) assigned to the input operand IN, and places the result in the output variable (*outDint*, *outInt*, *outLreal*, *outReal*) assigned to the output operand Q. Sqrt_dint, sqrt_int, sqrt_lreal, and sqrt_real respectively do not change the original DINT, INT, LREAL, or REAL data.

The output Q is valid except when:

- There is overflow.
- IN is negative.
- IN is a NaN (Not a Number).

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the SQRT instructions in LD Help.

Trigonometry

(PACSystems firmware version 2.50 or later; to process LREALs: 5.50 or later.)

Syntax

COS_REAL, SIN_REAL, and TAN_REAL have respectively replaced COS, SIN, and TAN, which are supported for backward compatibility.
Formal convention:

```
cos(IN := inReal, Q => outReal, ENO => outBool);

sin(IN := inReal, Q => outReal, ENO => outBool);

tan(IN := inReal, Q => outReal, ENO => outBool);


cos_real(IN := inReal, Q => outReal, ENO => outBool);

sin_real(IN := inReal, Q => outReal, ENO => outBool);

tan_real(IN := inReal, Q => outReal, ENO => outBool);


cos_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

sin_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

tan_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
```

Informal convention:

cos(inReal, outReal);

sin(inReal, outReal);

tan(inReal, outReal);

cos_real(inReal, outReal);

sin_real(inReal, outReal);

tan_real(inReal, outReal);

cos_lreal(inLreal, outLreal);

sin_lreal(inLreal, outLreal);

tan_lreal(inLreal, outLreal);

Note: *inReal* and *inLreal* are variables or constants ($-2^{63} <$ *inReal* [or *inLreal*] $< 2^{63}$).

Operation

An ST trigonometry *function* (cos, sin, and tan) respectively returns the cosine, sine, and tangent of the input variable (*inReal* or *inLreal*) assigned to the input operand IN. The result in radians is respectively placed in the variable *outReal* or *outLreal* assigned to the output operand Q. Cos, sin, and tan do not change the original REAL or LREAL data. ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	**LD function receives power**	**Equivalent ST called function**
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see Cosine, Sine, and Tangent in LD Help.

Math Functions

(For PACSystems firmware version 2.50 and later only.)
ST math *functions* perform basic math operations on data. They support operations on DINT, INT, LREAL, REAL, and UINT variables. If the data type mismatch preference is set to "Show as Warning" and there is a data type mismatch, a warning displays in the Feedback Zone.
Using mixed data types in a function can cause unexpected results. For example, if a REAL result is assigned to a DINT variable, the resulting value is copied into the DINT variable without being converted to a DINT data type. Because REAL data types are stored in a different format than DINT data types, the resulting value of the DINT variable may be unexpected.

Tip: If a REAL result is to be placed into a DINT variable, use the REAL_to_DINT conversion function.

Function	Mnemonics	Examples
Absolute Value	abs_dint	abs_dint(IN := inDint, Q => outDint, ENO => outBool);
	abs_int	abs_int(IN := inInt, Q => outInt, ENO => outBool);
	abs_lreal	abs_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
	abs_real	abs_real(IN := inReal, Q => outReal, ENO => outBool);
Scale	scale_dint	scale_dint(ihiDint, iloDint, ohiDint, oloDint, inDint, outDint);
	scale_int	scale_int(ihiInt, iloInt, ohiInt, oloInt, inInt, outInt);
	scale_uint	scale_uint(ihiUint, iloUint, ohiUint, oloUint, inUint, outUint);

Absolute Value

(PACSystems firmware version 2.50 or later; for Abs_LREAL: 5.50 or later.)

Syntax

Formal convention:

```
abs_dint(IN := inDint, Q => outDint, ENO => outBool);

abs_int(IN := inInt, Q => outInt, ENO => outBool);

abs_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

abs_real(IN := inReal, Q => outReal, ENO => outBool);
```

Informal convention:

```
abs_dint(inDint, outDint);

abs_int(inInt, outInt);

abs_lreal(inLreal, outLreal);

abs_real(inReal, outReal);
```

Operation

An ST absolute value *function* returns the unsigned magnitude of the input variable (*inDint, inInt, inLreal, inReal*) assigned to the input operand IN, placing the result in the output variable (*outDint*, *outInt*, *outLreal*, *outReal*) assigned to the output operand Q. Abs_dint, abs_int, abs_lreal and abs_real respectively do not change the original DINT, INT, LREAL, or REAL data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the ABS intructions in LD Help.

Scale

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
scale_dint(IHI := ihiDint, ILO := iloDint, OHI := ohiDint, OLO := oloDint, IN := inDint, OUT => outDint, ENO => outBool);

scale_int(IHI := ihiInt, ILO := iloInt, OHI := ohiInt, OLO := oloInt, IN := inInt, OUT => outInt, ENO => outBool);

scale_uint(IHI := ihiUint, ILO := iloUint, OHI := ohiUint, OLO := oloUint, IN := inUint, OUT => outUint, ENO => outBool);
```

Informal convention:

```
scale_dint(ihiDint, iloDint, ohiDint, oloDint, inDint, outDint);

scale_int(ihiInt, iloInt, ohiInt, oloInt, inInt, outInt);

scale_uint(ihiUint, iloUint, ohiUint, oloUint, inUint, outUint);
```

Operation

An ST scale *function* scales the value of the input variable (*inDint*, *inInt*, *inUint*) assigned to the input operand IN and places the result in the output variable (*outDint*, *outInt*, *outUint*) assigned to the output operand OUT. As in controller LD, the call is successful when the scale function is performed without overflow. Scale_dint, scale_int, and scale_uint respectively do not change the original DINT, INT, and UINT data. ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the SCALE functions in LD Help.

Control Functions

(For PACSystems firmware version 2.50 and later only.)
(For the Mask I/O Interrupt and the Suspend I/O Interrupt functions, PACSystems firmware version 3.50 and later only.)
ST Control *functions* limit program execution and change the way the CPU executes the application program. They support operations on BOOL, INT, and WORD variables. If the data type mismatch preference is set to "Show as Warning" and there is a data type mismatch, a warning displays in the Feedback Zone.

Function	Mnemonics	Examples
Do_I/O	do_io	do_io(ST := inWord, END := inWord2, ALT := inWord3, ENO => outBool);
Mask I/O Interrupt	mask_io_intr	mask_io_intr(MASK := inBool, IN1 := inWordorBool, ENO => outBool);
Service_Request	svc_req	svc_req(FNC := inInt, PRM := inWord, ENO => outBool);
Suspend_I/O	sus_io	sus_io(ENO => outBool);
Suspend I/O Interrupt	susp_io_intr	susp_io_intr(SUSP := inBool, IN1 := inWordorBool, ENO => outBool);
Switch_Position	switch_pos	switch_pos(POS => outWord1, MOD => outWord2, ENO => outBool);
Edge Detectors	r_trig f_trig	InstanceOfR_TRIG(CLK := inBool, Q => outBool, ENO => outBool); InstanceOfF_TRIG(CLK := inBool, Q => outBool, ENO => outBool);

Do I/O (do_io)

Syntax

Formal convention:

```
do_io(ST := inWord, END := inWord2, ALT := inWord3, ENO => outBool);
```

- or -

```
do_io(ST := inBool, END := inBool2, ALT := inWord, ENO => outBool);
```

Informal convention:

```
do_io(inWord, inWord2, inWord3);
```

- or -

```
do_io(inBool, inBool2, inWord);
```

Operation

The ST do_io *function* updates inputs or outputs for one scan while the program is running. You can also use do_io to update selected I/O during the program in addition to the normal I/O scan. Do_io does not change the original WORD and BOOL data.

Note: The ALT operand is optional and does not have to be included in a function call statement using the formal convention; the ALT operand **must** be included in an ST function call statement using the informal convention.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the DO_IO function in LD Help.

Edge Detectors - R_TRIG and F_TRIG

Operation

The IEC 61131-3 standard defines two edge detector function blocks. The *Rising Edge Trigger* R_TRIG and *Falling Edge Trigger* F_TRIG detect the changing state of a Boolean signal. The outputs of both function blocks produce a single pulse when an edge is detected.

Syntax for Rising Edge Trigger R_TRIG	**Syntax for Falling Edge Trigger F_TRIG**
Formal convention: InstanceOfR_TRIG(CLK := inBool, Q => outBool, ENO => outBool); Informal convention: InstanceOfR_TRIG(inBool, outBool, outBool);	Formal convention: InstanceOfF_TRIG(CLK := inBool, Q => outBool, ENO => outBool); Informal convention: InstanceOfF_TRIG(inBool, outBool, outBool);
You can declare instance variables of type R_TRIG and F_TRIG.	
The structure elements CLK and STATE are not accessible in logic. If they are used in logic, the compiler reports an error.	
All structure elements can be viewed in the Data Watch window and Data Monitor.	
Elements of an edge detector structure variable cannot be published. Publishing the elements would allow other applications to write to the instance data.	
The ENO parameter is the standard output parameter that indicates the function block executed correctly. ENO is set to Off when one of the other parameters is invalid.	
For R_TRIG, when the CLK input transitions from False to True, the output Q is set to True (Q = CLK and NOT STATE; STATE = CLK) for one function block instance execution. The output Q then remains False until a new rising edge is detected.	For F_TRIG, when the CLK input transitions from True to False, the output Q is set to True (Q = NOT CLK and NOT STATE; STATE = NOT CLK) for one function block instance execution. The output Q then remains False until a new falling edge is detected.
For R_TRIG, when the Controller transitions from stop to run mode and the CLK input is set to True, the output Q is set to True after the first execution of the function block instance. After the second execution, the output is set to False.	For F_TRIG, when the Controller transitions from stop to run mode and the CLK input is set to False, the output Q is set to True after the first execution of the function block instance. After the next execution, the output is set to False.

Operands

Operand	*Data Type*	*Memory Area*	*Description*
CLK	BOOL	I, Q, M, T, SA, SB, SC, G, symbolic	Edge detecting input
Q	Variable	I, Q, M, T, symbolic	Edge detecting output
STATE	BOOL		Internal Value

Notes

- C blocks cannot use edge detector function blocks.
- The Structure element Q is accessible in user logic. It is read-only.
- Instance variables reside in symbolic discrete memory, which is non-retentive.

CPU Support

R_TRIG and F_TRIG are supported for PACSystems CPUs with firmware version 5.00.

Mask I/O Interrupt (mask_io_intr)

Syntax

Formal convention:

```
mask_io_intr(MASK := inBool, IN1 := inWord, ENO => outBool);
```

- or -

```
mask_io_intr(MASK := inBool, IN1 := inBool2, ENO => outBool);
```

Informal convention:

```
mask_io_intr(inBool, inWord);
```

- or -

```
mask_io_intr(inBool, inBool2);
```

Operation

Use the *Mask I/O Interrupt function* to mask or unmask an interrupt from an input board when using I/O variables.

Note: When not using I/O variables, you can use SVC_REQ, where the value of the FNC operand must be 17.

When an interrupt is masked, the CPU does not execute the corresponding interrupt function when the input transitions and causes an interrupt.
Successful execution occurs unless:

- The I/O board is not a supported input module.
- The reference address specified does not correspond to a valid interrupt trigger reference.
- The specified channel does not have its interrupt enabled in the configuration.

Note: The ALT operand is optional and does not have to be included in a function call statement using the formal convention; the ALT operand **must** be included in an ST function call statement using the informal convention.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the MASK_IO_INTR instruction in LD Help.

SCAN SET I/O

Syntax

Formal convention:

```
scan_set_io (IN := inBool, OUT := inBool2, SET := inUint, ENO=> outBool);
```

Informal convention:

```
scan_set_io (inBool, inBool2, inUint);
```

Operation

A Scan Set I/O (SCAN_SET_IO) instruction scans the I/O of a specified scan set number. You can specify whether the inputs and outputs of the associated scan are scanned.

To assign less frequent scans:

1. In the Project tab of the Navigator, expand the Hardware Configuration node.
2. Expand the main rack.
3. Double-click the slot that contains the CPU.
   The Parameter editor displays the CPUs parameters.
4. In the Scan Sets tab, set the Number of Sweeps to a value greater than 1.

Notes

- Modules can be assigned to scan sets in hardware configuration.
- Execution of the SCAN_SET_IO instruction will not impact the normal scanning process of the corresponding scan set. For example, if the corresponding scan set is configured for non-default Number of Sweeps and/or Output delay settings, these are still in effect regardless of how many executions of the SCAN_SET_IO instruction occur in a sweep.
- The SCAN_SET_IO instruction will not scan any modules in the specified scan set if the modules do not support DO_IO scanning.

For details on valid operands, data types, memory areas, and examples, see also the SCAN_SET_IO instruction in LD Help.

Unsupported Modules

The following PACSystems modules do not support DO_IO scanning:

- IC693BEM331 90-30 Genius Bus Controller
- IC694BEM331 RX3i Genius Bus Controller
- IC693BEM341 90-30 2.5 GHz FIP Bus Controller
- IC693DNM200 90-30 Device Net Master
- IC695PBM300 RX3i ProfiBus Master
- IC695PBS301 RX3i ProfiBus Slave
- IC687BEM731 90-70 Genius Bus Controller
- IC697BEM731 90-70 Standard Width Genius Bus Controller

Suspend I/O (sus_io)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
sus_io(ENO => outBool);
```

Informal convention:

```
sus_io();
```

Operation

The ST sus_io *function* stops normal I/O scans from occurring for one CPU sweep. During the next output scan, all outputs are held at their current states. During the next input scan, the input references are not updated with data from inputs. However, during the input scan portion of the sweep, the CPU verifies that Genius bus controllers have completed their previous output updates.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the SUS_IO function in LD Help.

Suspend I/O Interrupt (susp_io_intr)

(For PACSystems firmware version 3.50 and later only.)

Syntax

Formal convention:

```
susp_io_intr(SUSP := inBool, IN1 := inWord, ENO => outBool);
```

- or -

```
susp_io_intr(SUSP := inBool, IN1 := inBool2, ENO => outBool);
```

Informal convention:

```
susp_io_intr(inBool, inWord);
```

- or -

```
susp_io_intr(inBool, inBool2);
```

Operation

Use the Suspend I/O Interrupt function with I/O variables to suspend a set of I/O interrupts and cause occurrences of these interrupts to be queued until these interrupts are resumed.

Note: When not using I/O variables, you can use SVC_REQ, where the value of the FNC operand must be 32.

The set of I/O interrupts are those that can be generated from the PACSystems High Speed Counter. The number of I/O interrupts that can be queued depends on the I/O module's capabilities. The CPU informs the I/O module that its interrupts are to be suspended or resumed. The I/O module's default is resumed. The Suspend applies to all I/O interrupts associated with the I/O module. Interrupts are suspended and resumed within a single scan.

Successful execution occurs unless:

- The I/O module associated with the specified address is not an appropriate module for this operation.
- Communication between the CPU and this I/O module has failed. (The board is not present, or it has experienced a fatal fault.)

Note: The ALT operand is optional and does not have to be included in a function call statement using the formal convention; the ALT operand **must** be included in an ST function call statement using the informal convention.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function

Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the SUSP_IO_INTR instruction in LD Help.

Service Request (svc_req)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
svc_req(FNC := inInt, PRM := inWord, ENO => outBool);
```

Informal convention:

```
svc_req(inInt, inWord);
```

Operation

The ST svc_req *function* requests the controller to perform the special controller service identified by the FNC operand.
Parameters for svc_req are located in the parameter block, which begins at the reference identified by the PRM operand. Svc_req does not change the original data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on each FNC operand, valid operands, data types, memory areas, and examples, see also the SVC_REQ function in LD Help.

Switch Position (switch_pos)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
switch_pos(POS => outWord1, MOD => outWord2, ENO => outBool);
```

Informal convention:

```
switch_pos(outWord2, outWord1);
```

Note: The operands in the statement above must be in U-shaped order using the informal convention.

Operation

The switch_pos *function* allows the ST logic to read the current position of the PACSystems switch, as well as the mode for which the switch is configured.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the SWITCH_POS function in LD Help.

Conversion Functions

(For PACSystems firmware version 2.50 and later only.)
ST data type conversion *functions* change a data item from one number format (data type) to another.

Note: ST math operators must be used with operands of the same data type. As a result, a data conversion is often required before using these operators.

Function	Mnemonic	Example
Convert Angles	DEG_TO_RAD	deg_to_rad(IN := inReal, Q => outReal, ENO => outBool);
	RAD_TO_DEG	rad_to_deg(IN := inReal, Q => outReal, ENO => outBool);
Convert BCD4 to INT	BCD4_TO_INT	bcd4_to_int(IN := inWord, Q => outInt, ENO => outBool);
Convert BCD4 to REAL	BCD4_TO_REAL	bcd4_to_real(IN := inWord, Q => outReal, ENO => outBool);
Convert BCD4 to UINT	BCD4_TO_UINT	bcd4_to_uint(IN := inWord, Q => outUint, ENO => outBool);
Convert BCD8 to DINT	BCD8_TO_DINT	bcd8_to_dint(IN := inDword, Q => outDint, ENO => outBool);
Convert BCD8 to REAL	BCD8_TO_REAL	bcd8_to_real(IN := inDword, Q => outReal, ENO => outBool);
Convert DINT to BCD8	DINT_TO_BCD8	dint_to_bcd8(IN := inDint, Q => outDword, ENO => outBool);
Convert DINT to DWORD	DINT_TO_DWORD	dint_to_dword(IN := inDint, Q => outDword, ENO => outBool);
Convert DINT to INT	DINT_TO_INT	dint_to_int(IN := inDint, Q => outInt, ENO => outBool);
Convert DINT to LREAL	DINT_TO_LREAL	dint_to_lreal(IN := inDint, Q => outLreal, ENO => outBool);
Convert DINT to REAL	DINT_TO_REAL	dint_to_real(IN := inDint, Q => outReal, ENO => outBool);
Convert DINT to UINT	DINT_TO_UINT	dint_to_uint(IN := inDint, Q => outUint, ENO => outBool);
Convert DWORD to DINT	DWORD_TO_DINT	dword_to_dint(IN := inDword, Q => outDint, ENO => outBool);
Convert INT to BCD4	INT_TO_BCD4	int_to_bcd4(IN := inInt, Q => outWord, ENO => outBool);
Convert INT to DINT	INT_TO_DINT	int_to_dint(IN := inInt, Q => outDint, ENO => outBool);

Convert INT to REAL	INT_TO_REAL	int_to_real(IN := inInt, Q => outReal, ENO => outBool);
Convert INT to UINT	INT_TO_UINT	int_to_uint(IN := inInt, Q => outUint, ENO => outBool);
Convert INT to WORD	INT_TO_WORD	int_to_word(IN := inInt, Q => outWord, ENO => outBool);
Convert LREAL to DINT	LREAL_TO_DINT	lreal_to_dint(IN := inLreal, Q => outDint, ENO => outBool);
Convert LREAL to REAL	LREAL_TO_REAL	lreal_to_real(IN := inLreal, Q => outReal, ENO => outBool);
Convert REAL to DINT	REAL_TO_DINT	real_to_dint(IN := inReal, Q => outDint, ENO => outBool);
Convert REAL to INT	REAL_TO_INT	real_to_int(IN := inReal, Q => outInt, ENO => outBool);
Convert REAL to LREAL	REAL_TO_LREAL	real_to_lreal(IN := inReal, Q => outLreal, ENO => outBool);
Convert REAL to UINT	REAL_TO_UINT	real_to_uint(IN := inReal, Q => outUint, ENO => outBool);
Truncate	TRUNC_DINT	trunc_dint(IN := inReal, Q => outDint, ENO => outBool);
	TRUNC_INT	trunc_int(IN := inReal, Q => outInt, ENO => outBool);
Convert UINT to BCD4	UINT_TO_BCD4	uint_to_bcd4(IN := inUint, Q => outWord, ENO => outBool);
Convert UINT to DINT	UINT_TO_DINT	uint_to_dint(IN := inUint, Q => outDint, ENO => outBool);
Convert UINT to INT	UINT_TO_INT	uint_to_dint(IN := inUint, Q => outInt, ENO => outBool);
Convert UINT to REAL	UINT_TO_REAL	uint_to_dint(IN := inUint, Q => outReal, ENO => outBool);
Convert UINT to WORD	UINT_TO_WORD	uint_to_word(IN := inUint, Q => outWord, ENO => outBool);
Convert WORD to INT	WORD_TO_INT	word_to_int(IN := inWord, Q => outInt, ENO => outBool);
Convert WORD to UINT	WORD_TO_UINT	word_to_uint(IN := inWord, Q => outUint, ENO => outBool);

Convert Angles

(PACSystems firmware version 2.50 or later; to process LREALs: 5.50 or later.)

Syntax

DEG_TO_RAD_REAL and RAD_TO_DEG_REAL have respectively replaced DEG_TO_RAD and RAD_TO_DEG, which are supported for backward compatibility.
Formal convention:

```
deg_to_rad(IN := inReal, Q => outReal, ENO => outBool);

rad_to_deg(IN := inReal, Q => outReal, ENO => outBool);


deg_to_rad_real(IN := inReal, Q => outReal, ENO => outBool);

rad_to_deg_real(IN := inReal, Q => outReal, ENO => outBool);


deg_to_rad_lreal(IN := inLreal, Q => outLreal, ENO => outBool);

rad_to_deg_lreal(IN := inLreal, Q => outLreal, ENO => outBool);
```

Informal convention:

```
deg_to_rad(inReal, outReal);

rad_to_deg(inReal, outReal);


deg_to_rad_real(inReal, outReal);

rad_to_deg_real(inReal, outReal);


deg_to_rad_lreal(inLreal, outLreal);

rad_to_deg_lreal(inLreal, outLreal);
```

Operation

An ST angle *function* converts data (in degrees or radians) specified by the  variable (*inReal* or *inLreal*) assigned to the input operand IN to data (in radians or degrees respectively), and places the result in the variable (*outReal* or *outLreal*) assigned to the output operand Q. Deg_to_rad and rad_to_deg do not change the original REAL or LREAL data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the Convert angle instructions in LD Help.

Convert BCD4 to INT (bcd4_to_int)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
bcd4_to_int(IN := inWord, Q => outInt, ENO => outBool);
```

Informal convention:

```
bcd4_to_int(inWord, outInt);
```

Operation

The ST bcd4_to_int *function* converts the 4-digit Binary-Coded-Decimal (BCD4) data specified by the variable (*inWord*) assigned to the input operand IN to the equivalent single-precision signed integer (INT) value, and places the result in the variable (*outInt*) assigned to the output operand Q. Bcd4_to_int does not change the original BCD4 data. ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the BCD4_TO_INT function in LD Help.

Convert BCD4 to REAL (bcd4_to_real)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
bcd4_to_real(IN := inWord, Q => outReal, ENO => outBool);
```

Informal convention:

```
bcd4_to_real(inWord, outReal);
```

Operation

The ST bcd4_to_real *function* converts the 4-digit Binary-Coded-Decimal (BCD4) data specified by the variable (*inWord*) assigned to the input operand IN to the equivalent floating-point (REAL) value, and places the result in the variable (*outReal*) assigned to the output operand Q. Bcd4_to_real does not change the original BCD4 data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the BCD4_TO_REAL function in LD Help.

Convert BCD4 to UINT (bcd4_to_uint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
bcd4_to_uint(IN := inWord, Q => outUint, ENO => outBool);
```

Informal convention:

```
bcd4_to_uint(inWord, outUint);
```

Operation

The ST bcd4_to_uint *function* converts the 4-digit Binary-Coded-Decimal (BCD4) data specified by the variable (*inWord*) assigned to the input operand IN to the equivalent single-precision unsigned integer (UINT) value, and places the result in the variable (*outUint*) assigned to the output operand Q. Bcd4_to_uint does not change the original BCD4 data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the BCD4_TO_UINT function in LD Help.

Convert BCD8 to DINT (bcd8_to_dint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
bcd8_to_dint(IN := inDword, Q => outDint, ENO => outBool);
```

Informal convention:

```
bcd8_to_dint(inDword, outDint);
```

Operation

The ST bcd8_to_dint *function* converts the 8-digit Binary-Coded-Decimal (BCD8) data specified by the  variable (*inDword*) assigned to the input operand IN to the equivalent double-precision signed integer (DINT) value, and places the result in the variable (*outDint*) assigned to the output operand Q. Bcd8_to_dint does not change the original BCD8 data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the BCD8_TO_DINT function in LD Help.

Convert BCD8 to REAL (bcd8_to_real)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
bcd8_to_real(IN := inDword, Q => outReal, ENO => outBool);
```

Informal convention:

```
bcd8_to_real(inDword, outReal);
```

Warning: Converting from BCD8 to REAL may result in the loss of significant digits.

Operation

The ST bcd8_to_real *function* converts the 8-digit Binary-Coded-Decimal (BCD8) data specified by the  variable (*inDword*) assigned to the input operand IN to the equivalent floating-point (REAL) value, and places the result in the variable (*outReal*) assigned to the output operand Q. Bcd8_to_real does not change the original BCD8 data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the BCD8_TO_REAL function in LD Help.

Convert DINT to BCD8 (dint_to_bcd8)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
dint_to_bcd8(IN := inDint, Q => outDword, ENO => outBool);
```

Informal convention:

```
dint_to_bcd8(inDint, outDword);
```

Operation

The ST dint_to_bcd8 *function* converts the double-precision signed integer (DINT) data specified by the variable (*inDint*) assigned to the input operand IN to the equivalent 8-digit Binary-Coded-Decimal (BCD8) value, and places the result in the variable (*outDword*) assigned to the output operand Q. Dint_to_bcd8 does not change the original DINT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the DINT_TO_BCD8 function in LD Help.

Convert DINT to DWORD (dint_to_dword)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
dint_to_dword(IN := inDint, Q => outDword, ENO => outBool);
```

Informal convention:

```
dint_to_dword(inDint, outDword);
```

Operation

The ST dint_to_dword *function* converts the double-precision signed integer (DINT) data specified by the variable (*inDint*) assigned to the input operand IN to the equivalent 4 byte, 32-bit bit string (DWORD) value, and places the result in the variable (*outDword*) assigned to the output operand Q. Dint_to_dword does not change the original DINT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	ENO effect
Successful	ENO set to 1 (True)
Failed	ENO set to 0 (False)

Convert DINT to INT (dint_to_int)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
dint_to_int(IN := inDint, Q => outInt, ENO => outBool);
```

Informal convention:

```
dint_to_int(inDint, outInt);
```

Operation

The ST dint_to_int *function* converts the double-precision signed integer (DINT) data specified by the  variable (*inDint*) assigned to the input operand IN to the equivalent single-precision signed integer (INT) value, and places the result in the variable (*outInt*) assigned to the output operand Q. Dint_to_int does not change the original DINT data. ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the DINT value exceeds the INT data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the DINT_TO_INT function in LD Help.

Convert DINT to LREAL (dint_to_lreal)

(For PACSystems firmware version 5.50 and later only.)

Syntax

Formal convention:

```
dint_to_lreal(IN := inDint, Q => outLreal, ENO => outBool);
```

Informal convention:

```
dint_to_lreal(inDint, outLreal);
```

Operation

The ST dint_to_lreal *function* converts the double-precision signed integer (DINT) data specified by the variable (*inDint*) assigned to the input operand IN to the equivalent double-precision floating-point (LREAL) value, and places the result in the variable (*outLreal*) assigned to the output operand Q. Dint_to_lreal does not change the original DINT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the DINT_TO_LREAL instruction in LD Help.

Convert DINT to REAL (dint_to_real)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
dint_to_real(IN := inDint, Q => outReal, ENO => outBool);
```

Informal convention:

```
dint_to_real(inDint, outReal);
```

Warning: Converting from DINT to REAL may result in the loss of significant digits.

Operation

The ST dint_to_real *function* converts the double-precision signed integer (DINT) data specified by the variable (*inDint*) assigned to the input operand IN to the equivalent floating-point (REAL) value, and places the result in the variable (*outReal*) assigned to the output operand Q. Dint_to_real does not change the original DINT data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the DINT_TO_REAL function in LD Help.

Convert DINT to UINT (dint_to_uint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
dint_to_uint(IN := inDint, Q => outUint, ENO => outBool);
```

Informal convention:

```
dint_to_uint(inDint, outUint);
```

Operation

The ST dint_to_uint *function* converts the double-precision signed integer (DINT) data specified by the  variable (*inDint*) assigned to the input operand IN to the equivalent single-precision unsigned integer (UINT) value, and places the result in the variable (*outUint*) assigned to the output operand Q. Dint_to_uint does not change the original DINT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the DINT value exceeds the UINT data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the DINT_TO_UINT function in LD Help.

Convert DWORD to DINT (dword_to_dint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
dword_to_dint(IN := inDword, Q => outDint, ENO => outBool);
```

Informal convention:

```
dword_to_dint(inDword, outDint);
```

Operation

The ST dword_to_dint *function* converts the 4 byte, 32-bit bit string (DWORD) data specified by the variable (*inDword*) assigned to the input operand IN to the equivalent double-precision signed integer (DINT) value, and places the result in the variable (*outDint*) assigned to the output operand Q. Dword_to_dint does not change the original DWORD data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	**ENO effect**
Successful	ENO set to 1 (True)
Failed	ENO set to 0 (False)

Convert INT to BCD4 (int_to_bcd4)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
int_to_bcd4(IN := inInt, Q => outWord, ENO => outBool);
```

Informal convention:

```
int_to_bcd4(inInt, outWord);
```

Operation

The ST int_to_bcd4 *function* converts the single-precision signed integer (INT) data specified by the variable (*inINT*) assigned to the input operand IN to the equivalent 4-digit Binary-Coded-Decimal (BCD4) value, and places the result in the variable (*outWord*) assigned to the output operand Q. Int_to_bcd4 does not change the original INT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the INT_TO_BCD4 function in LD Help.

Convert INT to DINT (int_to_dint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
int_to_dint(IN := inInt, Q => outDint, ENO => outBool);
```

Informal convention:

```
int_to_dint(inInt, outDint);
```

Operation

The ST int_to_dint *function* converts the single-precision signed integer (INT) data specified by the  variable (*inInt*) assigned to the input operand IN to the equivalent double-precision signed integer (DINT) value, and places the result in the variable (*outDint*) assigned to the output operand Q. Int_to_dint does not change the original INT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the INT_TO_DINT function in LD Help.

Convert INT to REAL (int_to_real)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
int_to_real(IN := inInt, Q => outReal, ENO => outBool);
```

Informal convention:

```
int_to_real(inInt, outReal);
```

Operation

The ST int_to_real *function* converts the single-precision signed integer (INT) data specified by the variable (*inInt*) assigned to the input operand IN to the equivalent floating-point (REAL) value, and places the result in the variable (*outReal*) assigned to the output operand Q. Int_to_real does not change the original INT data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the INT_TO_REAL function in LD Help.

Convert INT to UINT (int_to_uint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
int_to_uint(IN := inInt, Q => outUint, ENO => outBool);
```

Informal convention:

```
int_to_uint(inInt, outUint);
```

Operation

The ST int_to_uint *function* converts the single-precision signed integer (INT) data specified by the variable (*inInt*) assigned to the input operand IN to the equivalent single-precision unsigned integer (UINT) value, and places the result in the variable (*outUint*) assigned to the output operand Q. Int_to_uint does not change the original INT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the INT value exceeds the UINT data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the INT_TO_UINT function in LD Help.

Convert INT to WORD (int_to_word)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
int_to_word(IN := inInt, Q => outWord, ENO => outBool);
```

Informal convention:

```
int_to_word(inInt, outWord);
```

Operation

The ST int_to_word *function* converts the single-precision signed integer (INT) data specified by the variable (*inInt*) assigned to the input operand IN to the equivalent 2 byte, 16-bit bit string (WORD) value, and places the result in the variable (*outWord*) assigned to the output operand Q. Int_to_word does not change the original INT data. ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	ENO effect
Successful	ENO set to 1 (True)
Failed	ENO set to 0 (False)

Convert LREAL to DINT (lreal_to_dint)

(For PACSystems firmware version 5.50 and later only.)

Syntax

Formal convention:

```
lreal_to_dint(IN := inLreal, Q => outDint, ENO => outBool);
```

Informal convention:

```
lreal_to_dint(inLreal, outDint);
```

Warning: Converting from LREAL to DINT may result in overflow. For example, LREAL 5.7E120, which equals $5.7 * 10^{120}$, converts to DINT overflow.

Operation

The ST lreal_to_dint *function* converts the double-precision floating-point (LREAL) data specified by the  variable (*inLreal*) assigned to the input operand IN to the equivalent double-precision signed integer (DINT) value, and places the result in the variable (*outDint*) assigned to the output operand Q. Lreal_to_dint does not change the original LREAL data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the LREAL value exceeds the DINT data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the LREAL_TO_DINT instruction in LD Help.

Convert LREAL to REAL (lreal_to_real)

(For PACSystems firmware version 5.50 and later only.)

Syntax

Formal convention:

```
lreal_to_real(IN := inLreal, Q => outReal, ENO => outBool);
```

Informal convention:

```
lreal_to_real(inLreal, outReal);
```

Warning: Converting from LREAL to REAL may result in overflow. For example, LREAL 5.7E210, which equals $5.7 * 10^{210}$, converts to REAL overflow.

Operation

The ST lreal_to_real *function* converts the double-precision floating-point (LREAL) data specified by the  variable (*inLreal*) assigned to the input operand IN to the equivalent single-precision floating-point (REAL) value, and places the result in the variable (*outReal*) assigned to the output operand Q. Lreal_to_real does not change the original LREAL data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the LREAL value exceeds the REAL data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the LREAL_TO_REAL instruction in LD Help.

Convert REAL to DINT (real_to_dint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
real_to_dint(IN := inReal, Q => outDint, ENO => outBool);
```

Informal convention:

```
real_to_dint(inReal, outDint);
```

Warning: Converting from REAL to DINT may result in overflow. For example, REAL 5.7E20, which equals $5.7 * 10^{20}$, converts to DINT overflow.

Operation

The ST real_to_dint *function* converts the floating-point (REAL) data specified by the variable (*inReal*) assigned to the input operand IN to the equivalent double-precision signed integer (DINT) value, and places the result in the variable (*outDint*) assigned to the output operand Q. Real_to_dint does not change the original REAL data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the REAL value exceeds the DINT data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the REAL_TO_DINT function in LD Help.

Convert REAL to INT (real_to_int)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
real_to_int(IN := inReal, Q => outInt, ENO => outBool);
```

Informal convention:

```
real_to_int(inReal, outInt);
```

Warning: Converting from REAL to INT may result in overflow. For example, REAL 7.4E15, which equals $7.4 * 10^{15}$, converts to INT overflow.

Operation

The ST real_to_int *function* converts the floating-point (REAL) data specified by the variable (*inReal*) assigned to the input operand IN to the equivalent single-precision signed integer (INT) value, and places the result in the variable (*outInt*) assigned to the output operand Q. Real_to_int does not change the original REAL data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the REAL value exceeds the INT data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the REAL_TO_INT function in LD Help.

Convert REAL to LREAL (real_to_lreal)

(For PACSystems firmware version 5.50 and later only.)

Syntax

Formal convention:

```
real_to_lreal(IN := inReal, Q => outLreal, ENO => outBool);
```

Informal convention:

```
real_to_lreal(inReal, outLreal);
```

Operation

The ST real_to_lreal *function* converts the single-precision floating-point (REAL) data specified by the variable (*inReal*) assigned to the input operand IN to the equivalent double-precision floating-point (LREAL) value, and places the result in the variable (*outLreal*) assigned to the output operand Q. Real_to_lreal does not change the original REAL data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the REAL_TO_LREAL instruction in LD Help.

Convert REAL to UINT (real_to_uint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
real_to_uint(IN := inReal, Q => outUint, ENO => outBool);
```

Informal convention:

```
real_to_uint(inReal, outUint);
```

Warning: Converting from REAL to UINT may result in overflow. For example, REAL 7.2E17, which equals $7.2 * 10^{17}$, converts to UINT overflow.

Operation

The ST real_to_uint *function* converts the floating-point (REAL) data specified by the variable (*inReal*) assigned to the input operand IN to the equivalent single-precision unsigned integer (UINT) value, and places the result in the variable (*outUint*) assigned to the output operand Q. Real_to_uint does not change the original REAL data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the REAL value exceeds the UINT data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the REAL_TO_UINT function in LD Help.

Truncate

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
trunc_dint(IN := inReal, Q => outDint, ENO => outBool);

trunc_int(IN := inReal, Q => outInt, ENO => outBool);
```

Informal convention:

```
trunc_dint(inReal, outDint);

trunc_int(inReal, outInt);
```

Operation

The ST trunc_dint and trunc_int *functions* round a floating-point (REAL) value specified by the variable (*inReal*) assigned to the input operand IN down respectively to the nearest double-precision signed integer (DINT) or single-precision signed integer (INT) value and place the result in the variable (*outDint* or *outInt*) assigned to the output operand Q. Trunc_dint and trunc_int do not change the original REAL data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the truncate functions in LD Help.

Convert UINT to BCD4 (uint_to_bcd4)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
uint_to_bcd4(IN := inUint, Q => outWord, ENO => outBool);
```

Informal convention:

```
uint_to_bcd4(inUint, outWord);
```

Operation

The ST uint_to_bcd4 *function* converts the single-precision unsigned integer (UINT) data specified by the  variable (*inUint*) assigned to the input operand IN to the equivalent 4-digit Binary-Coded-Decimal (BCD4) value, and places the result in the variable (*outWord*) assigned to the output operand Q. Uint_to_bcd4 does not change the original UINT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the UINT_TO_BCD4 function in LD Help.

Convert UINT to DINT (uint_to_dint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
uint_to_dint(IN := inUint, Q => outDint, ENO => outBool);
```

Informal convention:

```
uint_to_dint(inUint, outDint);
```

Operation

The ST uint_to_dint *function* converts the single-precision unsigned integer (UINT) data specified by the variable (*inUint*) assigned to the input operand IN to the equivalent double-precision signed integer (DINT) value, and places the result in the variable (*outDint*) assigned to the output operand Q. Uint_to_dint does not change the original UINT data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the UINT_TO_DINT function in LD Help.

Convert UINT to INT (uint_to_int)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
uint_to_int(IN := inUint, Q => outInt, ENO => outBool);
```

Informal convention:

```
uint_to_int(inUint, outInt);
```

Operation

The ST uint_to_int *function* converts the single-precision unsigned integer (UINT) data specified by the variable (*inUint*) assigned to the input operand IN to the equivalent single-precision signed integer (INT) value, and places the result in the variable (*outInt*) assigned to the output operand Q. Uint_to_int does not change the original UINT data. ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed). The function call fails if the UINT value exceeds the INT data type range.

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the UINT_TO_INT function in LD Help.

Convert UINT to REAL (uint_to_real)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
uint_to_real(IN := inUint, Q => outReal, ENO => outBool);
```

Informal convention:

```
uint_to_real(inUint, outReal);
```

Operation

The ST uint_to_real function converts the single-precision unsigned integer (UINT) data specified by the variable (*inUint*) assigned to the input operand IN to the equivalent floating-point (REAL) value, and places the result in the variable (*outReal*) assigned to the output operand Q. Uint_to_real does not change the original UINT data.
ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see also the UINT_TO_REAL function in LD Help.

Convert UINT to WORD (uint_to_word)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
uint_to_word(IN := inUint, Q => outWord, ENO => outBool);
```

Informal convention:

```
uint_to_word(inUint, outWord);
```

Operation

The ST uint_to_word *function* converts the single-precision unsigned integer (UINT) data specified by the  variable (*inUint*) assigned to the input operand IN to the equivalent 2 byte, 16-bit bit string (WORD) value, and places the result in the variable (*outWord*) assigned to the output operand Q. Uint_to_word does not change the original UINT data. ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	ENO effect
Successful	ENO set to 1 (True)
Failed	ENO set to 0 (False)

Convert WORD to INT (word_to_int)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
word_to_int(IN := inWord, Q => outInt, ENO => outBool);
```

Informal convention:

```
word_to_int(inWord, outInt);
```

Operation

The ST word_to_int *function* converts the 2 byte, 16-bit bit string (WORD) data specified by the  variable (*inWord*) assigned to the input operand IN to the equivalent single-precision signed integer (INT) value, and places the result in the variable (*outInt*) assigned to the output operand Q. Word_to_int does not change the original WORD data. ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	**ENO effect**
Successful	ENO set to 1 (True)
Failed	ENO set to 0 (False)

Convert WORD to UINT (word_to_uint)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
word_to_uint(IN := inWord, Q => outUint, ENO => outBool);
```

Informal convention:

```
word_to_uint(inWord, outUint);
```

Operation

The ST word_to_uint *function* converts the 2 byte, 16-bit bit string (WORD) data specified by the  variable (*inWord*) assigned to the input operand IN to the equivalent single-precision unsigned integer (UINT) value, and places the result in the variable (*outUint*) assigned to the output operand Q. Word_to_uint does not change the original WORD data.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool* is set to 1 (function call was successful) or 0 (function call failed).

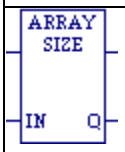
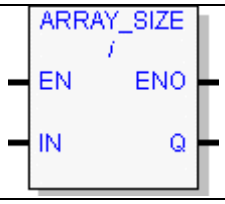
Function	ENO effect
Successful	ENO set to 1 (True)
Failed	ENO set to 0 (False)

Data Move Functions

(For PACSystems firmware version 2.50 and later only.)
ST Data Move *functions* provide basic data move capabilities. They support operations on BOOL, DWORD, and WORD variables. If the data type mismatch preference is set to "Show as Warning" and there is a data type mismatch, a warning displays in the Feedback Zone.

Function	Mnemonic	Example
Array Size	array_size	Q := array_size(IN := inArray);
Array Size Dimension 1	array_size_dim1	Q := array_size_dim1(IN := inArray);
Array Size Dimension 2	array_size_dim2	Q := array_size_dim2(IN := inArray);
Communication Request	comm_req	comm_req(IN := inWord, SYSID := inWord2, TASK := inDWORD, FT => FT_outBool, ENO => ENO_outBool);
Move Data Ex	move_data_ex	move_data_ex(DC := inBOOL, IN := inArray, Length := 6, Q => outArray);
Move from Flat	move_from_flat	This instruction is not currently available in ST and FBD. Use LD for the MOVE_FROM_FLAT operation.
Move to Flat	move_to_flat	move_to_flat(DC := inBOOL, IN := inUDTArray, Length := 10, Q => outBYTEArray);
Size of	size_of	Q := size_of(IN := inArray);

Array Size

LD	FBD	ST
		Formal convention: Q := ARRAY_SIZE(In := [input]);

Operation

ARRAY_SIZE counts the number of elements in the array assigned to input IN.

Output

- In LD and FBD, ARRAY_SIZE writes the number to output Q.
- In ST, the value is assigned to the variable to the left of the assignment operator, represented by variable Q in the ST code above.
- If a non-array variable is assigned to input IN, the value of Q is 1.
- In an array of n structure variables, the value n is written to Q: the elements in each structure variable are not counted.

Tip: If the array assigned to input IN of ARRAY_SIZE is passed to a parameterized C block for processing, also pass the value of output Q to the block. In the C block logic, use the value of output Q to ensure you process all the array elements without exceeding the end of the array. For a two-dimensional array, this method works only if all elements are treated identically, for example, all are initialized to the same value.

Operands

Input Operands

Operand	*Data Type*	*Memory Area*	*Description*
i	(FBD only.) The solve order for the instruction.		
power flow (LD only)			When set to On, ARRAY_SIZE executes. When set to Off, ARRAY_SIZE does not execute.
EN (FBD only)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	Enable input. When set to On, ARRAY_SIZE solves. When set to Off, ARRAY_SIZE does not solve.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, I/O variable	
IN	Array of any data type	data flow, I, Q, M, T, S, SA, SB, SC, G, R, P, L, AI, AQ, W, symbolic,	Array whose elements are counted

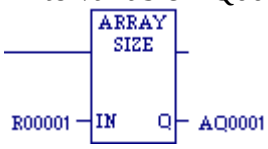
		I/O variable	

Output Operands

Operand	*Data Type*	*Memory Area*	*Description*
power flow (LD only; optional)			Set to On when ARRAY_SIZE has executed successfully.
ENO (FBD only; optional)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	ENO is set to On if EN is set to On.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, I/O variable	
Q	DINT or DWORD variable. ST also supports INT and WORD variables.	data flow, I, Q, M, T, G, R, P, L, AI, AQ, W, symbolic, I/O variable	Number of elements in the array assigned to input IN. In ST, the value of Q is assigned to the variable on the left side of the assignment operator.

Example

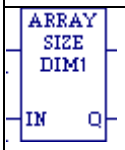
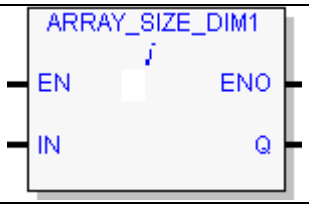
The two-dimensional array R00001 has its Array Dimension 1 property set to 4 and its Array Dimension 2 property set to 3. ARRAY_SIZE calculates 4 * 3 and writes the value 12 to variable AQ0001.



CPU Support

ARRAY_SIZE is supported for all PACSystems CPUs.

Array Size Dimension 1

LD	FBD	ST
		Formal convention: Q := ARRAY_SIZE_DIM1(In := [input]);

Operation

ARRAY_SIZE_DIM1 returns the value of the Array Dimension 1 property of an array.
Output

- In LD and FBD, ARRAY_SIZE_DIM1 writes the value to output Q.
- In ST, the value is assigned to the variable to the left of the assignment operator, represented by variable Q in the ST code above.
- If a non-array variable is assigned to input IN, the value of Q is 0.

You can use the value to ensure that a loop using a variable index to access array elements does not exceed the array's first dimension.

Operands

Input Operands

Operand	*Data Type*	*Memory Area*	*Description*
i	(FBD only.) The solve order for the instruction.		
power flow (LD only)			When set to On, ARRAY_SIZE_DIM1 executes. When set to Off, ARRAY_SIZE_DIM1 does not execute.
EN (FBD only)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	Enable input. When set to On, ARRAY_SIZE_DIM1 solves. When set to Off, ARRAY_SIZE_DIM1 does not solve.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, I/O variable	
IN	Array of any data type	data flow, I, Q, M, T, S, SA, SB, SC, G, R, P, L, AI, AQ, W, symbolic, I/O	The array whose Array Dimension 1 property value is returned in the Q output (LD, FBD) or assigned to the variable on the left side of the assignment statement

		variable	

Output Operands

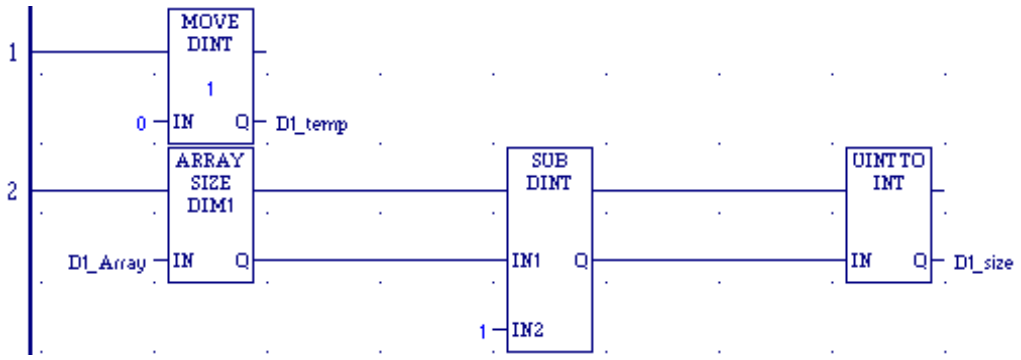
Operand	*Data Type*	*Memory Area*	*Description*
power flow (LD only; optional)			Set to On when ARRAY_SIZE_DIM1 has executed successfully.
ENO (FBD only; optional)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	ENO is set to On if EN is set to On.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, I/O variable	
Q	DINT or DWORD variable. ST also supports INT and WORD variables.	data flow, I, Q, M, T, S, SA, SB, SC, G, R, P, L, AI, AQ, W, symbolic, I/O variable	The value of the Array Dimension 1 property of the array assigned to input IN. The value is set to 0 if a non-array is assigned to IN. In ST, the value is assigned to the variable on the left side of the assignment operator. **Note:** Because the index of the first element of an array is zero, the index of the last element is one less than the value assigned to Q.

Examples

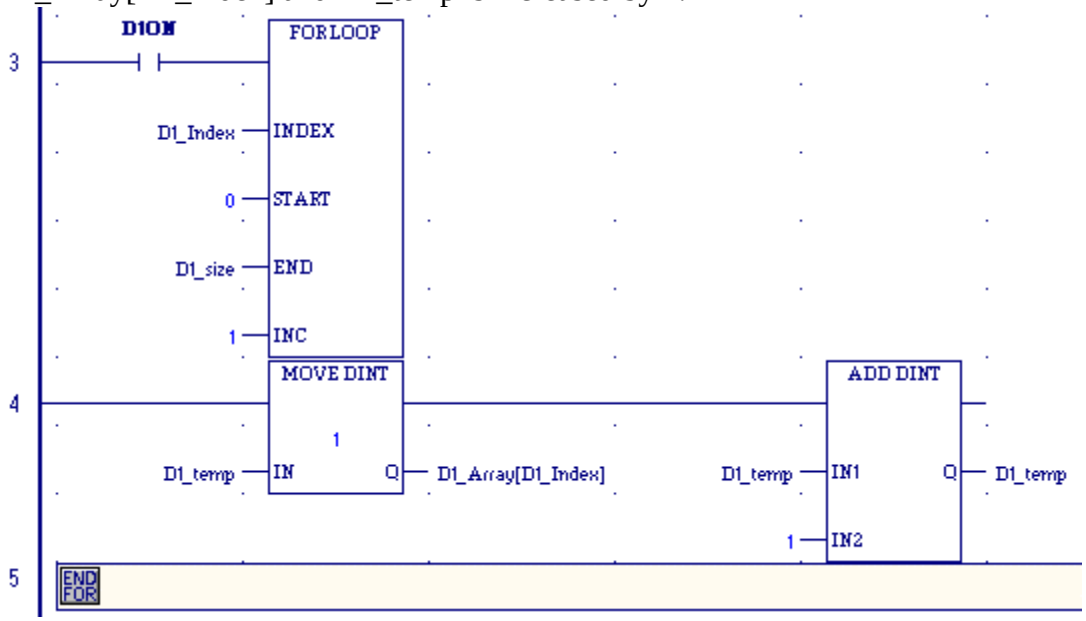
FOR_LOOP in LD logic

If you set up a FOR_LOOP that accesses array elements by means of a variable index, you must ensure that the FOR_LOOP does not iterate beyond the last element of the array.

In the following logic, MOVE_DINT initializes the variable D1_temp to 0. ARRAY_SIZE_DIM1 counts the number of elements of a one-dimensional array named D1_Array and outputs the result to output Q. Because the index of the first element of an array is zero, the loop must iterate (Q - 1) times. SUB_DINT performs the subtraction and the result is converted to an INT value and assigned to variable D1_size.



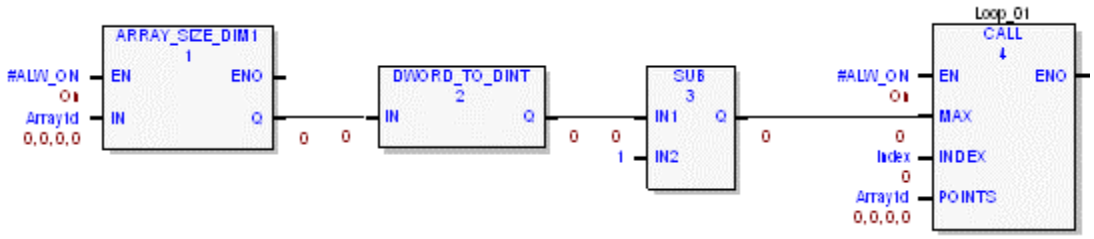
In the diagram below, the FOR_LOOP executes when D1ON is set to On. The variable index (D1_Index, see INDEX input) increments by 1 (see INC input) from 0 (see START input) through D1_size, the value calculated by ARRAY_SIZE_DIM1 and SUB_DINT (see END input). In each loop, the value of D1_temp is assigned to the element D1_Array[D1_Index] and D1_temp is increased by 1.



Loop initialization in FBD logic

FBD is not designed for loops that are completed in one scan. You can, however, use FBD to set up the parameters required for a loop and pass them to an LD or ST block that performs the loop.

In the following example, ARRAY_SIZE_DIM1 counts the number of elements of a one-dimensional array named Array1d and the value is converted from DWORD to DINT. Because an array element index is zero-based, the loop must iterate 1 less time than the number of elements. SUB performs the subtraction and passes the result to the MAX input of an LD or ST parameterized block named Loop_01. A value of 0 (zero) is assigned to the INDEX input and the array named Array1d is passed to the POINTS parameter.



In the non-displayed logic of Loop_01, a loop iterates by means of a variable index through all the elements of the POINTS formal parameter. The latter is set up as an 8-DWORD array, whereas the Array1d array passed to POINTS is a 4-DWORD array. Because the value assigned to MAX is 3, the loop iterates only four times, from 0 through 3, thus not exceeding the number of elements of the Array1d array used to populate the POINTS parameter. The Loop_01 block can thus be used to process the points of a 4-point or 8-point analog module.

Loop in ST logic

The following ST logic uses a For loop to initialize all elements of a one-dimensional REAL array, Array1d, to 0.0.
Dim1Size and i are INT variables. Array_Size_Dim1 is used to count the number of elements of Array1d. Because the lowest possible index of an array element is zero, the for loop iterates from 0 through a value 1 less than the number of elements.

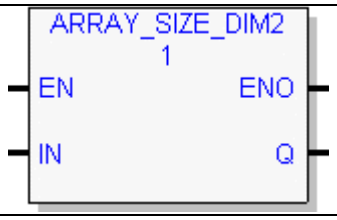
```
Dim1Size := Array_Size_Dim1(Array1d);

for i := 0 to (Dim1Size - 1) do

        Array1d[i] := 0.0

end_for;
```

If the above logic is used in an ST parameterized block or user-defined function block, any one-dimensional REAL array can be passed to it and have its elements initialized to 0.0.

CPU Support

ARRAY_SIZE_DIM1 is supported for all PACSystems CPUs.

Array Size Dimension 2

LD	FBD	ST
		Formal convention: Q := ARRAY_SIZE_DIM2(In := [input]);

Operation

ARRAY_SIZE_DIM2 returns the value of the Array Dimension 2 property of a two-dimensional array.

Output

- In LD and FBD, ARRAY_SIZE_DIM2 writes the value to output Q.
- In ST, the value is assigned to the variable to the left of the assignment operator, represented by variable Q in the ST code above.
- If a non-array variable is assigned to input IN, the value of Q is 0.

Using output Q as the upper limit of variable index loops in LD and ST

In an LD or ST block that is not a parameterized block or a User Defined Function Block (UDFB), you can use the output Q value to ensure that a loop using a variable index to access array elements does not exceed the array's second dimension.
Examples:

- LD logic: FOR_LOOP that iterates through an array's second dimension
- LD logic: FOR_LOOP that iterates through both dimensions of an array
- ST logic: FOR_LOOP that iterates through both dimensions of an array

Workaround to process two-dimensional arrays in parameterized blocks and UDFBs

Parameterized blocks and UDFBs do not support two-dimensional array parameters. All two-dimensional arrays passed into a parameterized block or UDFB are converted to flat, one-dimensional arrays. The total number of elements in the two-dimensional array input should equal the number of elements in the one-dimensional array parameter. If a two-dimensional array is passed into a parameterized block or UDFB, the ARRAY_SIZE_DIM1 and ARRAY_SIZE_DIM2 values should also be passed to determine which array element is being modified. The following example shows how a two-dimensional array is represented in one-dimensional format:

Two-dimensional array [3,2]	One-dimensional array [6]
Array[0,0]	Array[0]
Array[0,1]	Array[1]
Array[1,0]	Array[2]

Array[1,1]	Array[3]
Array[2,0]	Array[4]
Array[2,1]	Array[5]

The code to determine the one-dimensional parameter array index from the array indexes and array dimension 2 value of the two-dimensional array is

```
Flat_Index = (Dim1 * Array_Dim2) + Dim2
```

where

- Flat_Index is the index of the one-dimensional array element
- Dim1 is the one-dimensional index of the two-dimensional array element
- Array_Dim2 is the value of the Array Dimension 2 property of the two-dimensional array
- Dim2 is the two-dimensional index of the two-dimensional array element

Example: Passing a two-dimensional array from LD logic to an ST UDFB

No one-scan loop support in FBD

FBD is not designed for loops that are completed in one scan. You can use FBD to set up the loop parameters and pass them to an LD or ST block that performs the loop.

First dimension

To ensure that a variable index does not exceed the first dimension, ARRAY_SIZE_DIM1 is required.

Operands

Input Operands

Operand	*Data Type*	*Memory Area*	*Description*
i	(FBD only.) The solve order for the instruction.		
power flow (LD only)			When set to On, ARRAY_SIZE_DIM2 executes. When set to Off, ARRAY_SIZE_DIM2 does not execute.
EN (FBD only)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	Enable input. When set to On, ARRAY_SIZE_DIM2 solves. When set to Off, ARRAY_SIZE_DIM2 does not solve.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, I/O variable	
IN	Array of any data type	data flow, I, Q, M, T, S, SA, SB, SC,	The array whose Array Dimension 2 property value is returned in the Q output

		G, R, P, L, AI, AQ, W, symbolic, I/O variable	(LD, FBD) or assigned to the variable on the left side of the assignment statement

Output Operands

Operand	*Data Type*	*Memory Area*	*Description*
power flow (LD only; optional)			Set to On when ARRAY_SIZE_DIM2 has executed successfully.
ENO (FBD only; optional)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	ENO is set to On if EN is set to On.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, I/O variable	
Q	DINT or DWORD variable. ST also supports INT and WORD variables.	data flow, I, Q, M, T, S, SA, SB, SC, G, R, P, L, AI, AQ, W, symbolic, I/O variable	The value of the Array Dimension 2 property of the array assigned to the IN input. The value is set to 0 if a non-array is assigned to IN. In ST, the value is assigned to the variable on the left side of the assignment operator. **Note:** Because the index of the first element of an array is zero, the index of the last element is one less than the value assigned to Q.

Examples

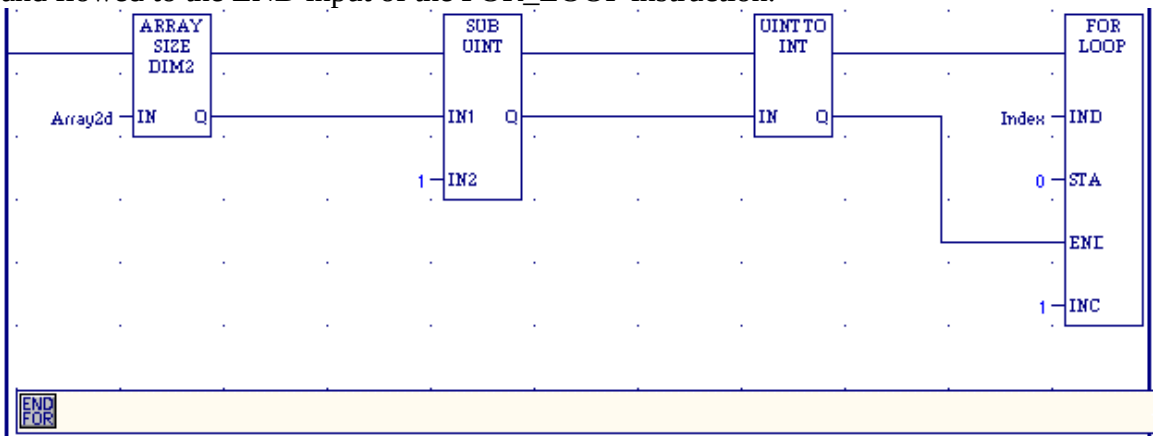
- LD logic: FOR_LOOP that iterates through an array's second dimension
- LD logic: FOR_LOOP that iterates through both dimensions of an array
- ST logic: FOR_LOOP that iterates through both dimensions of an array
- Passing a two-dimensional array from LD logic to an ST UDFB

LD logic: FOR_LOOP that iterates through an array's second dimension

In an LD block that is not a parameterized block or a User Defined Function Block (UDFB), if you set up a FOR_LOOP that accesses elements in the second dimension of an array by means of a variable index, you must ensure that the FOR_LOOP does not exceed the array's second dimension.

Note: Parameterized blocks and User Defined Function Blocks (UDFBs) do not support two-dimensional array parameters. See Workaround to process two-dimensional arrays in parameterized blocks and UDFBs.

In the following logic, ARRAY_SIZE_DIM2 counts the number of elements of a two-dimensional array named Array2d and outputs the result to output Q. Because the index of the first element of any dimension of an array is zero, the loop must iterate (Q - 1) times. SUB_UINT performs the subtraction and the result is converted to an INT value and flowed to the END input of the FOR_LOOP instruction.



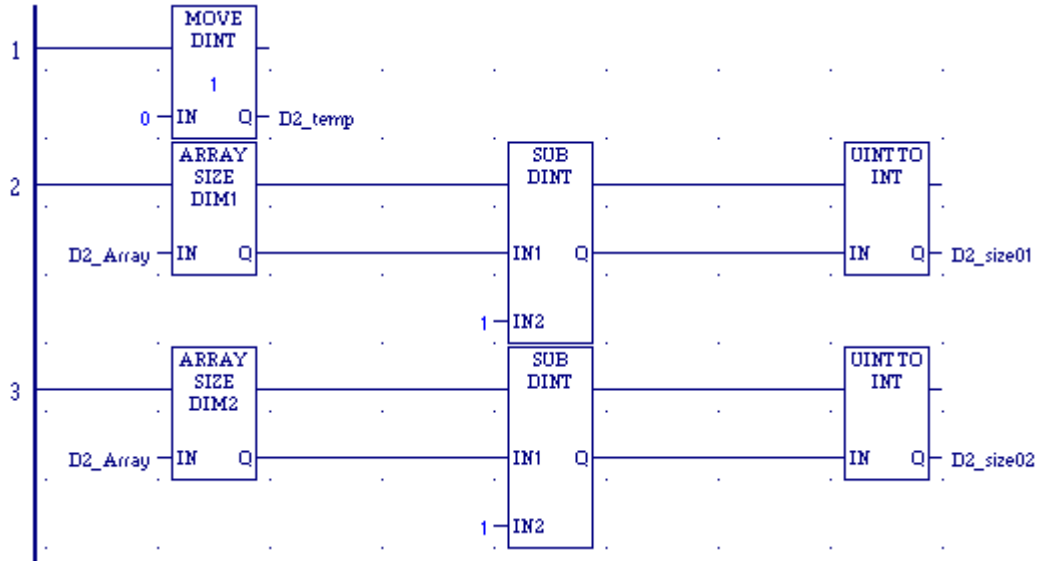
In the logic appearing between the FOR_LOOP and END_FOR instructions, various operations can take place with, for example, Array2d(1,Index), where Index increments by 1 (see INC input) from 0 (see START input) through the value calculated by ARRAY_SIZE_DIM2 and SUB_DINT (see END input).

LD logic: FOR_LOOP that iterates through both dimensions of an array

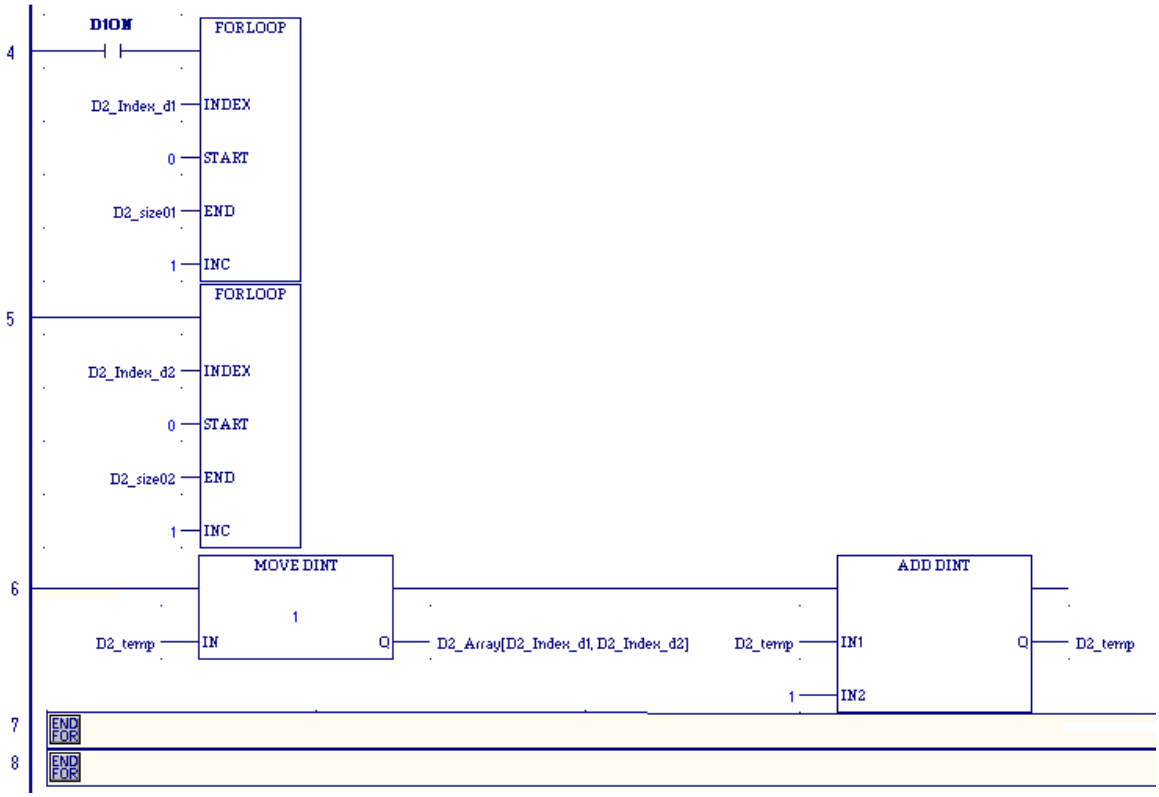
In an LD or ST block that is not a parameterized block or a User Defined Function Block (UDFB), you can use ARRAY_SIZE_DIM1, ARRAY_SIZE_DIM2, and nested FOR_LOOPs to ensure that operations on elements using two variable indexes each do not exceed either array dimension.

Note: Parameterized blocks and User Defined Function Blocks (UDFBs) do not support two-dimensional array parameters. See Workaround to process two-dimensional arrays in parameterized blocks and UDFBs.

In the following logic, MOVE_DINT initializes the variable D2_temp to 0.
ARRAY_SIZE_DIM1 counts the number of elements in the first dimension of a two-dimensional array named D2_Array. The value is placed in output Q. Because the index of the first element of an array is zero, the loop must iterate (Q - 1) times. SUB_DINT performs the subtraction and the result is converted to an INT value and assigned to variable D2_size01.
Likewise, ARRAY_SIZE_DIM2 counts the number of elements in the second dimension. After the subtraction and conversion, the value is assigned to variable D2_size02.



In the diagram below, the first FOR_LOOP executes when D1ON is set to ON. The variable index (D2_Index_d1, see INDEX input) increments by 1 (see INC input) from 0 (see START input) through D2_size01, the value calculated by ARRAY_SIZE_DIM1 and SUB_DINT (see END input). Within the first loop, the second FOR_LOOP executes. The variable index is D2_Index_d2, which increments by 1 from 0 through D2_size02, the value calculated by ARRAY_SIZE_DIM2 and SUB_DINT. Each time the second FOR_LOOP executes, the value of D2_temp is assigned to the element D2_Array[D2_Index_d1,D2_Index_d2] and D2_temp is incremented by 1.



ST logic: FOR_LOOP that iterates through both dimensions of an array

In an ST block that is not a parameterized block or a User Defined Function Block (UDFB), the following ST logic uses nested for loops to initialize all the elements of a two-dimensional REAL array, Array2d, to 0.0.

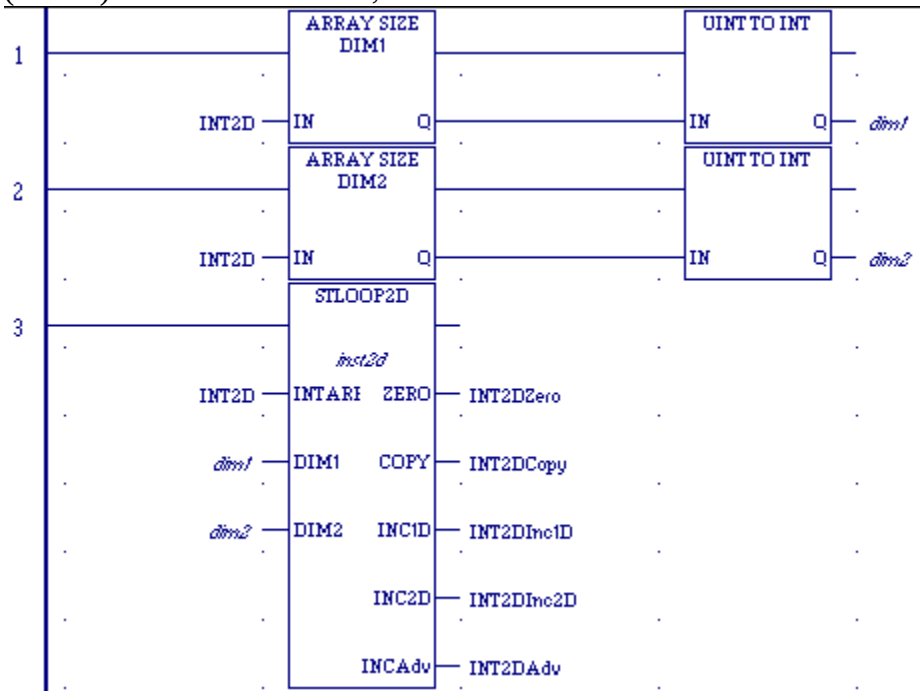
Note: Parameterized blocks and User Defined Function Blocks (UDFBs) do not support two-dimensional array parameters. See Workaround to process two-dimensional arrays in parameterized blocks and UDFBs.

Dim1Size, Dim2Size, i, and j are INT variables. Array_Size_Dim1 and Array_Size_Dim2 count the number of elements respectively in the first and second dimensions of Array2d. Because the lowest possible index of an array element is zero, the for loops iterate from 0 through a value 1 less than the number of elements in the respective dimension.

```
Dim1Size := Array_Size_Dim1(Array2d);

Dim2Size := Array_Size_Dim2(Array2d);

for i := 0 to (Dim1Size - 1) do

        for j := 0 to (Dim2Size - 1) do

                Array2d[i,j] := 0.0;

        end_for;

end_for;
```

Passing a two-dimensional array from LD logic to an ST UDFB

In the following LD logic, the two-dimensional array INT2D and its Array Dimension 1 and Array Dimension 2 values are passed to the ST User Defined Function Block (UDFB) named STLOOP2D, whose instance variable is inst2d.



INT2D is passed to the parameter INTARR, which is a one-dimensional array. One of the output parameters of STLOOP2D, INC1D, is a one-dimensional array; the other four are two-dimensional arrays. All five outputs are assigned to two-dimensional arrays for the calling LD logic to use.
In the following ST logic of UDFB STLOOP2D, five different operations on one-dimensional arrays are performed in such a way that the one-dimensional elements are correctly mapped to two-dimensional elements of the two-dimensional arrays assigned to the LD Call to STLOOP2D.

```
'Zero a 2D array
for i := 0 to (DIM1 - 1) do
   for j := 0 to (DIM2 - 1) do
      loopcount := (i*(DIM2) + j);
      ZERO[loopcount] := 0;
   end_for;
end_for;

'Copy the 2D array
for i := 0 to (DIM1 - 1) do
   for j := 0 to (DIM2 - 1) do
      loopcount := (i*(DIM2) + j);
      COPY[loopcount] := INTARR[loopcount];
   end_for;
end_for;

'Incrementing a 1D array
for i := 0 to (DIM1 - 1) do
   for j := 0 to (DIM2 - 1) do
      loopcount := (i*(DIM2) + j);
      INC1D[loopcount] := loopcount;
   end_for;
end_for;

'Incrementing a 2D array
for i := 0 to (DIM1 - 1) do
   for j := 0 to (DIM2 - 1) do
      loopcount := (i*(DIM2) + j);
      INC2D[loopcount] := j;
```

```
    end_for;
end_for;


'2D array value = DIM1 * 1000 + DIM2
'For example, 0, 1, 2, ... j, 1000, 1001, 1002, ...
for i := 0 to (DIM1 - 1) do
    for j := 0 to (DIM2 - 1) do
        loopcount := (i*(DIM2) + j);
        INCAdv[loopcount] := i*1000 + j;
    end_for;
end_for;
```

CPU Support

ARRAY_SIZE_DIM2 is supported for all PACSystems CPUs.

Communication Request (comm_req)

(For PACSystems firmware version 2.50 and later only.)

Syntax

Formal convention:

```
comm_req(IN := inWord, SYSID := inWord2, TASK := inDWORD, FT => FT_outBool, ENO =>
ENO_outBool);
```

Informal convention:

```
comm_req(inWord, inWord2, inDWORD, FT_outBool);
```

Operation

The ST Communication Request *function* communicates with a GE intelligent module, such as a Genius Communications Module, a Programmable Coprocessor Module, or a LAN Interface Module.

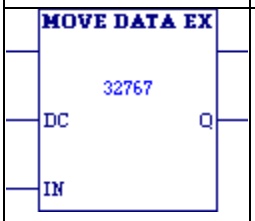
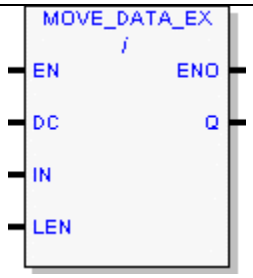
Note: The FT operand is optional and does not have to be included in an ST function call statement using the formal convention; the FT operand **must** be included in an ST function call statement using the informal convention.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *ENO_outBool* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on command blocks, valid operands, data types, memory areas, and examples, see also the COMM_REQ function in LD Help.

Move Data Explicit

LD	FBD	ST
		Formal convention: MOVE_DATA_EX(DC := , IN := , Length := , Q =>); - or - MOVE_DATA_EX(IN := , Length := , Q =>); **Tip:** Drag instruction from the Toolchest's LD Instructions or FBD Instructions drawer to your ST logic. The instruction with its input and output operand names are inserted in ST logic. Now assign a variable to the inputs and outputs, and a variable to the left of the assignment operator.

Operation

MOVE_DATA_EX provides optional data coherency by locking symbolic memory being written to during the copy operation. This enables a large number of bytes to be copied at one time.

Notes

- The input DC should be used only when using interrupt blocks and is required only when the same memory is used in more than one interrupt block, or in the main program and an interrupt block.
- If DC is True, an interrupt block cannot preempt the copy operation.
- If DC is False or not present, then interrupts can preempt the copy.
- Using DC can impact interrupt latency if the amount of data copied is large.

Operands

Notes

- All operands are required except for input DC (Data Coherency) and output Q.
- The constant value 0 is valid for DC.
- Input LEN (Length) must be a constant value.

Input Operands

Operand	*Data Type*	*Memory Area*	*Description*
i	(FBD only.) Solve order for the instruction.		

Power flow (LD)			When On, MOVE_DATA_EX executes. When Off, MOVE_DATA_EX does not execute.
EN (FBD)	BOOL variable or BOOL system variable	I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, discrete I/O variable	Enable input. When On, MOVE_DATA_EX solves. When Off, MOVE_DATA_EX does not solve.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, non-discrete I/O variable	
DC (Optional.)	▪ LD: data flow ▪ FBD: BOOL variable or BOOL system variable ▪ ST: BOOL variable, BOOL system variable, or BOOL constant	data flow, I, Q, M, T, G, symbolic, I/O variable	Data Coherency. ▪ If True, symbolic memory being written to is locked, enabling a coherent copying of symbolic data from one Controller memory area to another. ▪ If False (default), symbolic data is copied normally from one Controller memory area to another *without* data coherency. **Notes** ▪ DC should be used only when using interrupt blocks and is required only when the same memory is used in more than one interrupt block, or in the main program and an interrupt block. ▪ If DC is True, an interrupt block cannot preempt the copy operation. ▪ If DC is False or not present, then interrupts

			can preempt the copy.
IN	▪ Enumerated variable, array of enumerated variables, structure variable, or array of structure variables. ▪ (LD only.) Constant value 0.	I, Q, M, T, S, SA, SB, SC, G, R, P, L, AI, AQ, W, symbolic. data flow only if IN is connected to output Q of another MOVE_DATA_EX, MOVE_FROM_FLAT, or MOVE_TO_FLAT instruction.	Symbolic data to copy to the variable assigned to output Q as determined by the constant value assigned to LEN (Length). **Notes** ▪ The variable assigned to IN must be of the same data type as the variable assigned to output Q, except when the constant 0 (LD only) is assigned to IN. ▪ (LD only.) If the constant 0 is assigned to IN, then LEN (Length) is assigned the constant 1 and the variable or structure assigned to Q is set to its default (original) value.
▪ LEN (FBD) ▪ Length	INT constant in all languages	N/A	Length of IN; number of IN elements to copy. Valid range: 1 through 32,767. Default: 1. **Tip:** In the LD editor, double-click MOVE_DATA_EX to edit the LEN (Length) constant value.

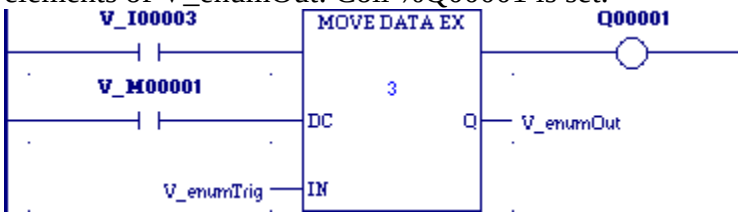
Output Operands

Operand	*Data Type*	*Memory Area*	*Description*
Power flow (LD; optional.)			On when MOVE_DATA_EX executed successfully
ENO (FBD; optional.)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, discrete I/O variable	ENO is set to On if EN is On.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, non-discrete I/O variable	
Q	Enumerated variable, array of enumerated variables, structure variable, or array of structure variables	I, Q, M, T, S, SA, SB, SC, G, R, P, L, AI, AQ, W, symbolic data flow only if Q is connected to input IN of another MOVE_DATA_EX, MOVE_FROM_FLAT, or MOVE_TO_FLAT instruction	Variable or array to which IN is copied. **Note:** The variable assigned to Q must be of the same data type as the variable assigned to input IN, except when the constant 0 (LD only) is assigned to IN.

Examples

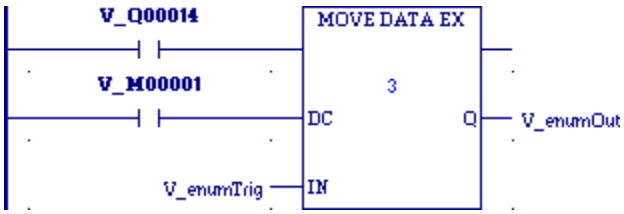
Example 1

When %I00003 is set, the first three elements of V_enumTrig are copied to the first three elements of V_enumOut. Coil %Q00001 is set.



Example 2

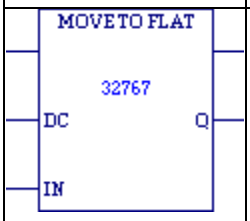
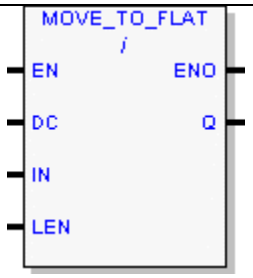
V_enumTrig and V_enumOut are both enumerated arrays of three elements in each array. Because Controllers do not recognize arrays, input Length should be 3, for the total number of enumerated elements to be copied. When the enabling input V_Q0014 is On, MOVE_DATA_EX copies three elements from memory location V_enumTrig to memory location V_enumOut.



CPU Support

MOVE_DATA_EX is supported for PACSystems with firmware version 6.00 or later.

MOVE_TO_FLAT

LD	FBD	ST
		Formal convention: MOVE_TO_FLAT(DC := , IN := , Length := , Q =>); - or - MOVE_TO_FLAT(IN := , Length := , Q =>); **Tip:** Drag instruction from Toolchest's LD Instructions or FBD Instructions drawer to your ST logic. The instruction with its input and output operand names are inserted in ST logic. Now assign a variable to the inputs and outputs, and a variable to the left of the assignment operator.

Operation

MOVE_TO_FLAT instruction copies data from symbolic or I/O variable memory to reference memory. MOVE_TO_FLAT copies across mismatching data types for an operation such as a Modbus transfer. MOVE_TO_FLAT provides optional data coherency by locking the reference memory being written to during the copy operation. This enables a large number of bytes to be copied at one time.

Notes

- The input DC should be used only when using interrupt blocks and is required only when the same memory is used in more than one interrupt block, or in the main program and an interrupt block.
- If DC is True, an interrupt block cannot preempt the copy operation.
- If DC is False or not present, then interrupts can preempt the copy.
- Using DC can impact interrupt latency if the amount of data copied is large.

Copying arrays and array elements

(LD, FBD.) The constant value assigned to input LEN (Length) determines the number of UDT array elements to be copied to the reference memory of the variable assigned to output Q.

Example: If the constant value 6 is assigned to LEN (length), then there should be a UDT array of at least six elements assigned to input IN. When logic executes, *n* bytes of data are copied from the UDT array elements to the reference memory of the variable assigned to output Q, where *n* is the length of the UDT array element (in bytes) times six.

Bounds check

If the number of bytes in the reference memory of the variable assigned to output Q is greater than the memory limit configured in Controller memory, an error appears in the  Feedback Zone and prevents downloading to the Controller.

Note: The constant value assigned to input LEN (Length) is not considered in this check.

Example: If the reference memory of the variable assigned to output Q is %R1 and the length (in bytes) of the UDT assigned to input IN is 6, then the highest used reference address is %R3. If the %R memory limit in the Controller is less than 3, then downloading to the Controller is prevented and an error appears in the Feedback Zone: Error 8038: %R memory usage exceeds limits in <block name>.
When MOVE_TO_FLAT receives power flow in LD, or executes in FBD or ST, the following occur:

- Data is copied from one location in Controller memory to another as determined by the constant value assigned to input LEN (Length).
- (LD.) The number inside MOVE_TO_FLAT is the number of UDT elements to be copied from. This number is the same as the input LEN value in FBD or ST. Valid range: 1 through 32,767.
- (Input Data Coherency optional.) If input DC is True, the reference memory of the variable assigned to output Q is locked, enabling a coherent copy of data from the UDT of input IN.
- If input DC is False (default), data is copied normally from the UDT to the reference memory of the variable assigned to Q without data coherency; that is, without locking the reference memory being copied to.

Operands

Input Operands

Operand	*Data Type*	*Memory Area*	*Description*
i	(FBD only.) Solve order for the instruction.		
Power flow (LD)			When On, MOVE_TO_FLAT executes. When Off, MOVE_TO_FLAT does not execute.
EN (FBD)	BOOL variable or BOOL system variable	I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, discrete I/O variable	Enable input. When On, MOVE_TO_FLAT solves. When Off, MOVE_TO_FLAT does not solve.

	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, non-discrete I/O variable	
DC (Optional.)	▪ LD: data flow ▪ FBD: BOOL variable or BOOL system variable ▪ ST: BOOL variable, BOOL system variable, or BOOL constant	data flow, I, Q, M, T, G, S, R, P, L, AI, AQ, W, symbolic, I/O variable	Data Coherency. ▪ If True, the reference memory being copied to is locked. This enables a coherent copy of a UDT to reference memory. ▪ If False (default), the UDT is copied normally to reference memory without locking the reference memory area being copied to; that is, *without* data coherency. **Notes** ▪ DC should be used only when using interrupt blocks and is required only when the same memory is used in more than one interrupt block, or in the main program and an interrupt block. ▪ If DC is True, an interrupt block cannot preempt the copy operation. ▪ If DC is False or not present, then interrupts can preempt the copy.
IN	User-defined Data Type (UDT) variable or UDT array	Discrete or non-discrete symbolic, discrete or non-discrete I/O variable	The data copied to the reference memory that the variable assigned to Q is mapped to. **Notes** ▪ Input LEN (Length) determines how many UDT variable elements to copy to Q. ▪ If an array head is assigned to input IN, the constant value assigned to LEN (Length) determines how many UDT array elements assigned to IN are copied to reference memory.

▪ LEN (FBD) ▪ Length	INT constant in all languages	N/A	Length of IN; number of UDT elements to copy to reference memory in output Q. Valid range: 1 through 32,767. Default: 1. **Tip:** In the LD editor, double-click MOVE_TO_FLAT to edit the LEN (Length) constant value.

Output Operands

Operand	*Data Type*	*Memory Area*	*Description*
Power flow (LD; optional.)			On when MOVE_TO_FLAT executed successfully
ENO (FBD; optional.)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, discrete I/O variable	ENO is set to On if EN is On
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, non-discrete I/O variable	
Q	BYTE, WORD, or array of BYTEs or WORDs	All memory areas except [%S, discrete symbolic, discrete I/O variable]	The variable, mapped to reference memory, that IN is copied to. The amount of data copied is determined by the constant value assigned to input LEN (Length). **Notes** ▪ Indirect referencing is available for all register references (%R, %P, %L, %W, %AI, and %AQ). ▪ BYTE arrays must be packed; that is, they must be in discrete memory.

Example

Consider the following:

- A UDT variable or UDT array is assigned to input IN.
- The constant value 8 is assigned to input LEN (Length).
- A DWORD variable mapped to %R1 is assigned to output Q.

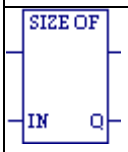
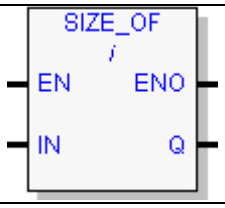
If the constant value 8 assigned is assigned to LEN (length), there should be a UDT array of at least eight elements assigned to IN. When MOVE_TO_FLAT executes, *n* bytes of

data are copied from the UDT variable or array to %R memory, starting at %R1 in the example, where *n* is the length of a UDT array element (in bytes) times eight.

CPU Support

MOVE_TO_FLAT is supported for PACSystems with firmware version 6.00 or later.

Size Of

LD	FBD	ST
		Formal convention: Q := SIZE_OF(In := [input]);

Operation

SIZE_OF counts the number of bits used by the variable assigned to input IN.

Output

- In LD and FBD, SIZE_OF writes the number of bits to output Q.
- In ST, the value is assigned to the variable to the left of the assignment operator, represented by variable Q in the ST code above.

A validation error occurs if you assign one of the following to input IN:

- A BYTE array in non-discrete memory
- A double-segment structure variable

Tip: If the variable assigned to input IN of SIZE_OF is passed to a parameterized C block for processing, also pass the value of output Q to the block. In C block logic, use the value of output Q to ensure that you process all the bits used by the variable without exceeding the end of the variable.

Operands

Input Operands

Operand	*Data Type*	*Memory Area*	*Description*
i	(FBD only.) The solve order for the instruction.		
power flow (LD only)			When set to On, SIZE_OF executes. When set to Off, SIZE_OF does not execute.
EN (FBD only)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	Enable input. When set to On, SIZE_OF solves. When set to Off, SIZE_OF does not solve.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, I/O variable	
IN	Variable of any data type except BYTE arrays in non-	data flow, I, Q, M, T, S, SA, SB, SC, G, R, P, L,	The variable whose size in bits is

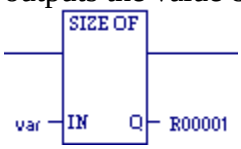
	discrete memory and double-segment structures	AI, AQ, W, symbolic, I/O variable	calculated

Output Operands

Operand	*Data Type*	*Memory Area*	*Description*
power flow (LD only; optional)			Set to On when SIZE_OF has executed successfully
ENO (FBD only; optional)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, I/O variable	ENO is set to On if EN is set to On.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, I/O variable	
Q	DINT or DWORD variable. ST also supports INT and WORD variables.	data flow, I, Q, M, T, G, R, P, L, AI, AQ, W, symbolic, I/O variable	The number of bits used by the variable assigned to input IN. In ST, the value is assigned to the variable on the left side of the assignment operator.

Example

The single-segment structure named var assigned to input IN contains eight BOOL elements (8 * 1 = 8 bits) and twelve WORD elements (12 * 16 = 192 bits). SIZE_OF outputs the value 8 + 192 = 200 to the variable R00001 assigned to output Q.



CPU Support

SIZE_OF is supported for all PACSystems CPUs.

Program Flow Functions

(For PACSystems firmware version 5.00 and later only.)
Use these functions to structure the execution flow of ST logic.

Function	Mnemonic	Example
Argument Present	arg_pres	arg_pres(IN := myIN, Q => outBOOL, ENO => outBool2);

Argument Present

(For PACSystems firmware version 5.00 and later only.)

Syntax

Formal convention:

```
arg_pres(IN := myIN, Q => outBOOL, ENO => outBool2);
```

Informal convention:

```
arg_pres(myIN, outBOOL);
```

Operation

An argument present (ARG_PRES) instruction determines if an input or output parameter value existed when the function block instance of the parameter was invoked. For example, a parameter may be optional (pass by value).

Note: This instruction must be called from a function block instance or from a parameterized block.

ENO is an optional BOOL output parameter. If ENO is present in a statement using the formal convention, then the state of *outBool2* is set to 1 (function call was successful) or 0 (function call failed).

Function	LD function receives power	Equivalent ST called function
Successful	Function passes power	ENO set to 1 (True)
Failed	Function does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the Arg_Pres instruction in LD Help.

Advanced Math Operators

(For PACSystems firmware version 2.50 and later only.)
ST advanced math operators perform advanced math operations on data.

Note: For information on the order of precedence of ST operators, see Order of Operations.

Operator	Symbol	Example
Exponential	** or ^	myRealResult := myReal ** myInt;

Exponential

(For PACSystems firmware version 2.50 or later; to process LREALs: 5.50 or later.)

Syntax

variable := variable1 ** variable2;

- or -

variable := variable1 ^ variable2;

Operation

The ST exponential operator calculates the value of *variable1* to the power of the value of *variable2* (*variable1*^{variable2}), placing the result in *variable*.

- *variable1* is the base.
- *variable2* is the exponent.

Operand Data Types

- *variable1* (the base) can be an LREAL or *REAL* variable, *expression*, parameter, or constant.
- The data types supported for the exponent vary depending on the data type used for the base.

  When the base is LREAL, *variable2* (the exponent) must be an LREAL variable, expression, parameter, or constant.

  When the base is REAL, the exponent can be a DINT, INT, REAL, or UINT variable, expression, parameter, or constant.
- The result is a variable of the same data type as the base.

Notes

- The ** and ^ operators do not support data types in the same way as the EXPT function.
- If the result of an expression is greater than the maximum or less than the minimum, then the result is platform dependent.
- For ST operators that have similar instructions, for example, the ST exponential (^) operator and the LD EXPT function, an underflow or an overflow result may not be the same.

Memory Areas

The operands are supported in the following memory areas: R, P, L, AI, AQ, W, I, Q, M, T, G, symbolic, I/O variable.

Examples

```
myRealResult := myReal ** myInt;

real1 := 2.0 ^ 2;

real2 := 3.0 ** 4.0;


dint1 := 2.0 ** 2.0; 'invalid: the result must be REAL or LREAL

rea12 := 2 ^ 2; 'invalid: the base must be REAL or LREAL

myLReal := myReal ^ 1.0; 'invalid: the base and the result must be of the same data type
```

Math Operators

(For PACSystems firmware version 2.50 and later only.)
ST math operators perform basic math operations on data. They support operations on DINT, INT, REAL, and UINT variables. When using operators, all operands in a single ST basic math statement must be the same data type, and the output operand must be the same data type as the input operands.

Note: The operators in the table below are in order of precedence. For more information, see Order of Operations.

Operator	Symbol	Example
Negation	-	myRealResult := -myReal;
Multiplication	*	myDintResult := myDint * myDint2;
Division	/	myRealResult := myReal / myReal2;
Modulo	mod	myUintResult := myUint mod myUint2;
Addition	+	myIntResult := myInt + myInt2;
Subtraction	-	myDintResult := myDint - myDint2;

Addition (+)

(PACSystems firmware version 2.50 or later; to add LREALs: 5.50 or later.)

Syntax

```
variable := variable1 + variable2;
```

Operation

The ST addition operator adds the values of *variable1* and *variable2*, placing the sum in *variable*.

Operand Data Types

- All the input and output operands must resolve to the same data type; they cannot be mixed.
- Input operands can be DINT, INT, LREAL, REAL, or UINT variables, expressions, parameters, or constants.
- The output operand must be a variable.
- If the result of an expression is greater than the maximum or less than the minimum, then the result is platform dependent.
- For ST operators and LD instructions that are similar, for example the ST addition (+) instruction and the LD ADD_INT instruction, an underflow or an overflow result may not be the same.

Examples

```
myIntResult := myInt + myInt2;

myINT := myINT + 1;

myREAL := myREAL + 1.0;


myUINT := 2.5 + 1.0; 'invalid: a REAL or LREAL result cannot be assigned to a UINT variable


myREAL := 2 + 1; 'invalid: a DINT, INT, or UINT result cannot be assigned to a REAL variable


myDINT := myDINT + 1.0;

myREAL := myREAL + 1;

myLREAL := myREAL + 1.0;

'The above three examples are invalid: all operands must be of the same data type
```

Division (/)

(PACSystems firmware version 2.50 or later; to divide LREALs: 5.50 or later.)

Syntax

```
variable := variable1 / variable2;
```

Operation

The ST division operator divides the value of *variable1* by the value of *variable2*, placing the quotient in *variable*.

Operand Data Types

- All the input and output operands must resolve to the same data type; they cannot be mixed.
- Input operands can be DINT, INT, LREAL, REAL, or UINT variables, expressions, parameters, or constants.
- The output operand must be a variable.
- A division by zero causes an application fault at runtime.
- If the operands are of data type INT, UINT, or DINT, and if the result of the division operation is not an integer, that is, not a whole number, then the decimal portion of the result is truncated. For example, if the division operation yields the value 0.99, the result is the value zero.
- For ST operators and LD instructions that are similar, for example the ST division (/) instruction and the LD DIV_INT instruction, an underflow or an overflow result may not be the same.

Examples

```
myRealResult := myReal / myReal2;

myREAL := myREAL / 2.0;

myUINT := myUINT / 2;
```

In the above example, if the result of the calculation is not an integer; that is, not a whole number, then the decimal portion of the result is truncated. In this case, if the result of the division operation is 2.99, the result is the UINT value 2.

```
myINT := 12.5 / 2.0; 'invalid: a REAL or LREAL result cannot be assigned to an INT variable

myREAL := 25 / 2; 'invalid: a DINT, INT, or UINT result cannot be assigned to a REAL variable


myDINT := myDINT / 2.0;

myREAL := myREAL / 2;

myLREAL := myREAL / 2.0;

'The above three examples are invalid: all operands must be of the same data type
```

Modulo (mod)

(For PACSystems firmware version 2.50 and later only.)

Syntax

variable := variable1 mod variable2;

Operation

The ST modulo operator divides the value of *variable1* by *variable2*, placing the remainder in *variable*.

Operand Data Types

- Operands can be DINT, INT, or UINT variables, expressions, parameters, or constants.
- For valid operand data types for the mod operator, click here.

Note: Both the input and output operands must resolve to the same data type; they can't be mixed.

Examples

myUintResult := myUint mod myUint2;

myDINT := myDINT mod 2;

myINT := 101 mod 3;

myDINT := myINT1 mod myINT2; ' invalid because an INT result can't be assigned to a DINT variable

Multiplication (*)

(PACSystems firmware version 2.50 or later; to multiply LREALs: 5.50 or later.)

Syntax

```
variable := variable1 * variable2;
```

Operation

The ST multiplication operator multiplies the value of *variable1* by the value of *variable2*, placing the product in *variable*.

Operand Data Types

- All the input and output operands must resolve to the same data type; they cannot be mixed.
- Input operands can be DINT, INT, LREAL, REAL, or UINT variables, expressions, parameters, or constants.
- The output operand must be a variable.
- If the result of an expression is greater than the maximum or less than the minimum, then the result is platform dependent.
- For ST operators and LD instructions that are similar, for example the ST multiplication (*) instruction and the LD MUL_INT instruction, an underflow or an overflow result may not be the same.

Examples

```
myDintResult := myDint * myDint2;

myDINT := myDINT * -1;

myREAL := myREAL * 2.0;


myUINT := 2.5 * 1.0; 'invalid: a REAL or LREAL result cannot be assigned to a UINT variable

myREAL := 2 * 2; 'invalid: a DINT, INT, or UINT result cannot be assigned to a REAL variable


myDINT := myDINT * -1.0;

myREAL := myREAL * 1;

myLREAL := myREAL * 1.0;


'All three above examples are invalid; all operands must be of the same data type.
```

Negation (-)

(For PACSystems firmware version 2.50 and later only; to negate LREALs: 5.50 or later)
Syntax

```
variable := -variable1;
```

Operation

The ST negation operator calculates the negative value of *variable1*, placing the result in *variable*.

- If *variable1* is a positive number, then *variable* becomes the negative value of *variable1*.
- If *variable1* is a negative number, then *variable* becomes the positive value of *variable1*.

Operand Data Types

- Both the input and output operands must resolve to the same data type; they cannot be mixed.
- The input operand can be a DINT, INT, LREAL, or REAL variable, expression, parameter, or constant.
- The output operand must be a variable.
- For ST operators and LD instructions that are similar, for example the ST negation (-) instruction and the LD SUB_INT instruction, an underflow or an overflow result may not be the same.

Examples

```
myRealResult := -myReal;
```

The above statement calculates the negative value of the REAL variable *myReal*, and the result is assigned to the REAL variable *myRealResult*.

```
myINT := -myINT2;

myREAL := -1.0;


myDINT := -1.0; 'invalid: the REAL constant -1.0 cannot be assigned to a DINT variable.

myREAL := -2; 'invalid: the INT constant -2 cannot be assigned to a REAL variable.
```

Subtraction (-)

(PACSystems firmware version 2.50 or later; to subtract LREALs: 5.50 or later.)
Syntax

```
variable := variable1 - variable2;
```

Operation

The ST subtraction operator subtracts the value of *variable2* from *variable1*, placing the difference in *variable*.

Operand Data Types

- All the input and output operands must resolve to the same data type; they cannot be mixed.
- Input operands can be DINT, INT, LREAL, REAL, or UINT variables, expressions, parameters, or constants.
- The output operand must be a variable.
- If the result of an expression is greater than the maximum or less than the minimum, then the result is platform dependent.
- For ST operators and LD instructions that are similar, for example the ST subtraction (-) instruction and the LD SUB_INT function, an underflow or an overflow result may not be the same.

Examples

```
myDintResult := myDint - myDint2;

myDINT := myDINT - 1;

myREAL := myREAL - 1.0;


myDINT := 2.5 - 1.0; 'invalid: a REAL or LREAL result cannot be assigned to a DINT variable

myREAL := 2 - 1; 'invalid: a DINT, INT, or UINT result cannot be assigned to a REAL variable

myUINT := myUINT - 1.0; 'invalid: operand1 and operand2 must be of the same data type

myREAL := myREAL - 1; 'invalid: operand1 and operand2 must be of the same data type

myLREAL := myREAL - 1.0; 'invalid: a REAL result cannot be assigned to an LREAL variable
```

Bitwise Operators

(For PACSystems firmware version 2.50 and later only.)
For ST bitwise operators:

- They perform basic bit or boolean operations on data.
- Operands can be BYTE, WORD, DWORD, or BOOL variables, expressions, parameters, or constants.
- All operands in a single ST statement with a bitwise operator must be the same data type.

Notes

- For the operators in the table below, if the input operand is a BOOL, then the operator is a boolean operator.
- The operators in the table below are in order of precedence. For more information, see Order of Operations.

Operator	Symbol	Example
Not		myByteResult := not myByte1;
And	&	myWordResult := myWord1 & myWord2;
Xor		myDwordResult := myDword1 xor myDword2;
Or		myByteResult := myByte1 or myByte2;

Bitwise NOT

(For PACSystems firmware version 2.50 and later only.)

Syntax

variable **:= not** variable1;

Note: If the input operand is BOOL, then the ST *not* operator is a boolean *not*.

Operation

The ST bitwise *not* operator sets the state of each bit of *variable* to 1 (True) if the state of the corresponding bit of *variable1* is 0 (False); otherwise, the state of the corresponding bit of *variable* is set to 0 (False).

Truth Table

not	**A**	=	**B**
	1		0
	0		1

Operand Data Types

- Operands can be BOOL, BYTE, DWORD, or WORD variables, expressions, parameters, or constants.
- For valid operand data types for the *not* operator, click here.

Note: All operands must be of the same data type.

Example

The following example shows the result of a *not* operation performed on a BYTE variable named *myByte1*. The result is stored in the variable named *myByteResult*. 0=False, 1=True.

myByteResult := not myByte1;

myByte1 = 0 1 0 0 0 1 0 0

myByteResult = 1 0 1 1 1 0 1 1

Tip: BYTEs, WORDs, and DWORDs used in a bitwise *NOT* should be displayed in = *4) BSPSPopupOnMouseOver(event)" class=BSSCPopup id=a3">binary format because other number formats may display misleading information.*

Bitwise AND (&)

(For PACSystems firmware version 2.50 and later only.)

Syntax

variable **:=** variable1 **and** variable2;

- or -

variable := variable1 **&** variable2;

Note: If the input operands are BOOL, then the ST *and* operator is a boolean *and*.

Operation

The ST bitwise *and* operator sets the state of each bit of *variable* to 1 (True) if the state of the corresponding bit of *variable1* is 1 (True) and the state of the corresponding bit of *variable2* is 1 (True); otherwise, the state of the corresponding bit of *variable* is set to 0 (False).

Truth Table

A	and	B	=	C
1		1		1
1		0		0
0		1		0
0		0		0

Operand Data Types

- Operands can be BOOL, BYTE, DWORD, or WORD variables, expressions, parameters, or constants.
- For valid operand data types for the *and* operator, click here.

Note: All operands must be the same data type.

Example

The following example shows the result of the *and* operation performed on two WORD variables named *myWord1* and *myWord2*. The result is stored in the variable named *myWordResult*. 0=False, 1=True.

myWordResult := myWord1 and myWord2;

- or -

```
myWordResult := myWord1 & myWord2;
```

```
myWord1       = 0 1 1 0 ... 0 0 0 0
myWord2       = 0 1 0 0 ... 0 1 0 0
-------------------------------------
myWordResult = 0 1 0 0 ... 0 0 0 0
```

Note: BYTEs, WORDs, and DWORDs used in a bitwise *AND* should be displayed in = *4) BSPSPopupOnMouseOver(event)" class=BSSCPopup id=a3">binary format because other number formats may display misleading information.*

Bitwise XOR

(For PACSystems firmware version 2.50 and later only.)

Syntax

variable **:=** variable1 **xor** variable2;

Note: If the input operands are BOOL, then the ST *xor* operator is a boolean *xor*.

Operation

The ST bitwise exclusive or (*xor*) operator sets the state of each bit of *variable* to 1 (True) if the state of the corresponding bit of *variable1* is set to 1 (True) or the state of the corresponding bit of the *variable2* is set to 1 (True), but not both; otherwise, the state of the corresponding bit of *variable* is set to 0 (False).

Truth Table

A	xor	B	=	C
1		1		0
1		0		1
0		1		1
0		0		0

Operand Data Types

- Operands can be BOOL, BYTE, DWORD, or WORD variables, expressions, parameters, or constants.
- For valid operand data types for the *xor* operator, click here.

Note: All operands must be of the same data type.

Example

The following example shows the result of the *xor* operation performed on two DWORD variables named *myDword1* and *myDword2*. The result is stored in the DWORD variable named *myDwordResult*. 0=OFF, 1=ON.

```
myDwordResult := myDword1 xor myDword2;
```

```
myDword1       = 0 1 1 0 ... 1 1 0 0
myDword2       = 1 1 0 0 ... 1 0 0 1
myDwordResult = 1 0 1 0 ... 0 1 0 1
```

Tip: BYTEs, WORDs, or DWORDs used in a bitwise OR should be displayed in = *4) BSPSPopupOnMouseOver(event)" class=BSSCPopup id=a3">binary format because other number formats may display misleading information.*

Bitwise OR

(For PACSystems firmware version 2.50 and later only.)

Syntax

variable **:=** variable1 **or** variable2;

Note: If the input operands are BOOL, then the ST *or* operator is a boolean *or*.

Operation

The ST bitwise *or* operator sets the state of each bit of *variable* to 1 (True) if the state of the corresponding bit of *variable1* is 1 (True) or the state of the corresponding bit of *variable2* is 1 (True); if both bits are set to 0 (False), the state of the corresponding bit of *variable* is set to 0 (False). If both bits are set to 1 (True), the state of the corresponding bit of *variable* is set to 1 (True).

Truth Table

A	or	B	=	C
1		1		1
1		0		1
0		1		1
0		0		0

Operand Data Types

- Operands can be BOOL, BYTE, DWORD, or WORD variables, expressions, parameters, or constants.
- For valid operand data types for the *and* operator, click here.

Note: All operands must be of the same data type.

Example

The following example shows the *or* operation performed on two BYTE variables named *myByte1* and *myByte2*. The result is stored in the variable named *myByteResult*. 0=OFF, 1=ON.

myByteResult := myByte1 or myByte2;

myByte1 = 0 1 1 0 1 1 0 0
myByte2 = 0 1 0 0 0 0 0 1
myByteResult = 0 1 1 0 1 1 0 1

Tip: BYTEs, WORDs, and DWORDs used in a bitwise *OR* should be displayed in = *4) BSPSPopupOnMouseOver(event)" class=BSSCPopup id=a3">binary format because other number formats may display misleading information.*

Relational Operators

(For PACSystems firmware version 2.50 and later only.)
For ST relational (comparison) operators:

- Operands can be INT, UINT, DINT, LREAL, REAL, BYTE, WORD, DWORD, or BOOL variables, expressions, parameters, or constants.
- Data types cannot be mixed in an expression.
- BOOL values cannot be compared, except by means of the equality and inequality operators.
- The result is always BOOL.

Note: For information on the order of precedence of ST operators, see Order of Operations.

Operator	Symbol	Example
Equality	=	If (myInt = 0) Then
Greater than or equal	>=	myBoolResult := myDint >= myMaxDint;
Greater than	>	If (myUint > 1) Then
Less than or equal	<=	myBoolResult := myInt <= myMinInt;
Less than	<	If (myDint < myDint2) Then
Inequality	<>, !=	myBoolResult := myUint <> myUint2;

Equality (=)

(PACSystems firmware version 2.50 or later; to compare LREALs: 5.50 or later; to compare STRUCTUREs and enumerated data types: 5.60 or later.)

Syntax

variable := variable1 = variable2;

Operation

The ST equality operator compares the values of *variable1* and *variable2*. If they are equal, then the state of *variable* is set to 1 (True); otherwise, the state of *variable* is set to 0 (False).

Operand Data Types

- *variable1* and *variable2* must be of the same data type; they can be BOOL, BYTE, DINT, DWORD, INT, LREAL, REAL, UINT, WORD, STRUCTURE, or enumerated variables, expressions, parameters, or constants.
- *variable*, the result, must be BOOL.
- -0.0 and 0.0 are equal.

Warning: Comparing REAL or LREAL values may produce unexpected results. For example, a calculation may result in 1.9999999, which is not equal to 2.0000000.

Note: For STRUCTURES, all elements must be in either discrete or analog memory; a mix is not allowed. TON, TOF, TP, and custom structures are not supported by the equality operator.

Examples

if (myInt = 0) then

If the value of the variable *myInt* is equal to zero, then the statements following the *If* statement are executed; otherwise, the statements after the next *Else*, *Elsif*, or *End_If* are executed.

myBoolResult := myWord = myWord2;

If the value of the variable *myWord* is equal to the value of the variable *myWord2*, then the state of the BOOL variable *myBoolResult* is set to 1 (True); otherwise, the state of *myBoolResult* is set to 0 (False).

Inequality (<> or !=)

(PACSystems firmware version 2.50 or later; to compare LREALs: 5.50 or later; to compare STRUCTUREs and enumerated data types: 5.60 or later.)

Syntax

variable := variable1 <> variable2;

- or -

variable := variable1 != variable2;

Operation

The ST inequality operator compares the values of *variable1* and *variable2*. If they are not equal, then the state of *variable* is set to 1 (True); otherwise, the state of *variable* is set to 0 (False).

Operand Data Types

- *variable1* and *variable2* must be of the same data type; they can be BOOL, BYTE, DINT, DWORD, INT, LREAL, REAL, UINT, WORD, STRUCTURE, or enumerated variables, expressions, parameters, or constants.
- *variable*, the result, must be BOOL.

Note: -0.0 and 0.0 are equal.

Warning: Comparing REAL or LREAL values may produce unexpected results. For example, a calculation may result in 1.9999999, which is not equal to 2.0000000.

Note: For STRUCTURES, all elements must be in either discrete or analog memory; a mix is not allowed. TON, TOF, TP, and custom structures are not supported by the inequality operator.

Examples

myBoolResult := myUint <> myUint2;

If the value of the variable *myUint* is not equal to the value of the variable *myUint2*, then the state of the BOOL variable *myBoolResult* is set to 1 (True); otherwise, the state of *myBoolResult* is set to 0 (False).

```
myInt_Index := 0;

while myInt_Index != 100 do

    myInt_Index := myInt_Index + 1;

end_while;
```

When the value of the variable *myInt_Index* is equal to 100, then the statements following the *end_while* statement are executed.
)" style="width: 40px; height: 20px; margin-top: 0px; margin-bottom: 0px; margin-left: 0px; margin-right: 0px; float: none; border-style: none; vertical-align: baseline; border-style: none; vertical-align: baseline;" width=40 height=20 align=bottom border=0>

Greater than (>)

(PACSystems firmware version 2.50 or later; to compare LREALs: 5.50 or later.)

Syntax

variable := variable1 > variable2;

Operation

The ST greater than operator compares the values of *variable1* and *variable2*. If the value of *variable1* is greater than the value of *variable2*, then the state of *variable* is set to 1 (True); otherwise, the state of *variable* is set to 0 (False).

Operand Data Types

- *variable1* and *variable2* must be of the same data type; they can be BYTE, DINT, DWORD, INT, LREAL, REAL, UINT, or WORD variables, expressions, parameters, or constants.
- *variable*, the result, must be BOOL.

Note: -0.0 and 0.0 are equal.

Examples

if (myUint > 1) then

If the value of the variable *myUint* is greater than one, then the statements following the *then* keyword are executed; otherwise, the statements after the next *Else, Elsif*, or *End_If* are executed.

myBoolResult := myReal > myReal2;

If the value of the variable *myReal* is greater than the value of the variable *myReal2*, then the state of the BOOL variable *myBoolResult* is set to 1 (True); otherwise, the state of *myBoolResult* is set to 0 (False).
)" style="width: 40px; height: 20px; margin-top: 0px; margin-bottom: 0px; margin-left: 0px; margin-right: 0px; float: none; border-style: none; vertical-align: baseline; border-style: none; vertical-align: baseline;" width=40 height=20 align=bottom border=0>

Greater than or Equal (>=)

(PACSystems firmware version 2.50 or later; to compare LREALs: 5.50 or later.)

Syntax

```
variable := variable1 >= variable2;
```

Operation

The ST greater than or equal operator compares the values of *variable1* and *variable2*. If the value of *variable1* is greater than or equal to the value of *variable2*, then the state of *variable* is set to 1 (True); otherwise, the state of *variable* is set to 0 (False).

Operand Data Types

- *variable1* and *variable2* must be of the same data type; they can be BYTE, DINT, DWORD, INT, LREAL, REAL, UINT, or WORD variables, expressions, parameters, or constants.
- *variable*, the result, must be BOOL.

Note: -0.0 and 0.0 are equal.

Examples

```
myBoolResult := myDint >= myMaxDint;
```

If the value of the variable *myDint* is greater than or equal to the value of the variable *myMaxDint*, then the state of the BOOL variable *myBoolResult* is set to 1 (True); otherwise, the state of *myBoolResult* is set to 0 (False).

```
repeat

    myUint_Index := myUint_Index + 1;

    until myUint_Index >= 100

end_repeat;
```

When the value of the variable *myUint_Index* is greater than or equal to 100, then the statements following the *end_repeat* statement are executed.
=)" style="width: 40px; height: 20px; margin-top: 0px; margin-bottom: 0px; margin-left: 0px; margin-right: 0px; float: none; border-style: none; vertical-align: baseline; border-style: none; vertical-align: baseline;" width=40 height=20 align=bottom border=0>

Less than (<)

(PACSystems firmware version 2.50 or later; to compare LREALs: 5.50 or later.)
Syntax

```
variable := variable1 < variable2;
```

Operation

The ST less than operator compares the values of *variable1* and *variable2*. If the value of *variable1* is less than the value of *variable2*, then the state of *variable* is set to 1 (True); otherwise, the state of *variable* is set to 0 (False).

Operand Data Types

- *variable1* and *variable2* must be of the same data type; they can be BYTE, DINT, DWORD, INT, LREAL, REAL, UINT, or WORD variables, expressions, parameters, or constants.
- *variable*, the result, must be BOOL.

Note: -0.0 and 0.0 are equal.

Examples

```
if (myDint < myDint2) then
```

If the value of the variable *myDint* is less than the value of the variable *myDint2*, then the statements following the *then* keyword are executed; otherwise, the statements after the next *Else*, *Elsif*, or *End_If* are executed.

```
myBoolResult := myReal < myReal2;
```

If the value of the variable *myReal* is less than the value of the variable *myReal2*, then the state of the BOOL variable *myBoolResult* is set to 1 (True); otherwise, the state of *myBoolResult* is set to 0 (False).

Less than or Equal (<=)

(PACSystems firmware version 2.50 or later; to compare LREALs: 5.50 or later.)

Syntax

```
variable := variable1 <= variable2;
```

Operation

The ST less than or equal operator compares the values of *variable1* and *variable2*. If the value of *variable1* is less than or equal to the value of *variable2*, then the state of *variable* is set to 1 (True); otherwise, the state of *variable* is set to 0 (False).

Operand Data Types

- *variable1* and *variable2* must be of the same data type; they can be BYTE, DINT, DWORD, INT, LREAL, REAL, UINT, or WORD variables, expressions, parameters, or constants.
- *variable*, the result, must be BOOL.

Note: -0.0 and 0.0 are equal.

Examples

```
myBoolResult := myInt <= myMinInt;
```

If the value of the variable *myInt* is less than or equal to value of the variable *myMinInt*, then the state of the BOOL variable *myBoolResult* is set to 1 (True); otherwise, the state of *myBoolResult* is set to 0 (False).

```
while myInt_Index <= 100 do

    myInt_Index := myInt_Index + 1;

end_while;
```

- While the value of the variable *myInt_Index* is less than or equal to the constant 100, the statements between the *do* keyword and the *end_while* statement are executed.
- When the value of the variable *myInt_Index* is greater than 100, then the statements following the *end_while* statement are executed.

Statements

(For PACSystems firmware version 2.50 and later only.)

ST blocks are composed of statements.

Statement	Symbol	Example
Assignment	:=	myDintBackup := myDintPrimary; **Note:** The result of a calculation on the right side of the assignment statement must be of the same data type as the destination variable on the left side of the assignment statement.

Block call		myBlock(myInput, myOutput);
		- or -
		myBlock(in := myInput, Q => myOutput, ENO => mySuccess);

Case		case myInt of
		1: myStatus := 1;
		myFlag := 50;
		2..6: myStatus := 2;
		7, 11, 19: myStatus := 3;
		20..26, 30, 32, 35..40: myStatus := 4;
		else
		myError := 1;
		myStatus := 0;
		end_case;

Comment	'	' This is a line comment
	//	// This is also a line comment
	(* *)	(* This is a block comment, which can span one or more lines. *)
Exit		exit;
For ... do		for myCurrent_col := 1 to 10 do

	<statement_list;> end_for;
Function block invocation	Instance(myInput, myOutput); - or - Instance(in := myInput, Q => myOutput, ENO => mySuccess);

Function call	abs_int(myInput, myOutput);
	- or -
	abs_int(in := myInput, Q => myOutput, ENO => mySuccess);

If ... then ... elsif	if (myDint = 0) then
	myTag := 1;
	elsif (myDint2 > 0) then
	myTag := 2;
	else
	myTag := 3;
	end_if;

Repeat ... until	repeat INT_temp := @INT_index; INT_index := INT_index + 1; @INT_index := INT_temp; until INT_index = 100 end_repeat;
Return	return;
While ... do	while INT_index < 100 do INT_temp := @INT_index; INT_index := INT_index + 1; @INT_index := INT_temp; end_while;

Assignment (:=)

(For PACSystems firmware version 2.50 and later only; to replace STRUCTUREs and enumerated data types: 5.60 or later.)

Syntax

variable **:=** expression;

Operation

The assignment statement replaces the current value of a variable with the result of an expression. An assignment statement consists of a variable reference on the left-hand side, followed by the assignment operator *:=*, followed by the expression to be evaluated. For example, the ST statement

```
A := B + C;
```

can be used to replace the single data value of variable A with the current value of the expression B + C.

Operand Data Types

- The expression must resolve to the same data type as the variable to which the value of the expression is assigned; data types cannot be mixed.
- Depending on the operator used in the expression, the following data types may be supported: BOOL, BYTE, DINT, DWORD, INT, LREAL, REAL, UINT, WORD, STRUCTURE, and enumerated variables.
- If the result of an expression is greater than the maximum or less than the minimum, then the result is platform dependent.
- For ST operators and LD instructions that are similar, for example the ST addition (+) instruction and the LD ADD_INT function, an underflow or an overflow result may not be the same.

Note: For STRUCTURES, all elements must be in either discrete or analog memory; a mix is not allowed. TON, TOF, TP, and custom structures are not supported in an assignment statement.

Examples

CV := CV + 1;

(* In the above example, the ST assignment operator := assigns the value of the input expression *CV + 1* using the same output variable *CV*. *)

myDINT := myDINT2; 'valid

myREAL5 := myREAL10; 'valid

myINT := 2.5; 'invalid: a REAL result can be assigned only to a REAL variable

myREAL := 100;

(* the above statement is invalid: an INT, UINT, DINT, BYTE, WORD, or DWORD result can be assigned only to an INT, UINT, DINT, BYTE, WORD, or DWORD variable respectively. *)

myINT := myUINT;

myUINT := myDINT;

'both of the above are invalid: the input variable or expression must resolve to the same data type as the output variable (result).

MyBOOL := myDINT = myDINT2; 'valid: relational operators always resolve to a BOOL

Block Call

(For PACSystems firmware version 2.50 and later only.)

Syntax

Block call: *Block_Name*(formal or informal argument list);

Tip: See also function call and function block invocation.

Operation

A Structured Text block call statement is a call to an ST block, parameterized ST block, LD block, parameterized LD block, C block, or parameterized C block, using either the formal or informal call convention. The block may then perform logical and arithmetic operations on the parameters as the block call statement is executed. After the called block execution is complete, the statement immediately following the block call statement is executed.

Notes

- The block call statement can be used in any ST block, including the _MAIN block and parameterized blocks.
- You can set up recursive subroutines by having a block call itself. A target's default stack size allows at least eight nested calls before a stack overflow fault is logged and the controller transitions to STOP/FAULT mode. To allow a deeper nesting of calls, increase the program stack size.
- The _MAIN block cannot be called.
- The maximum number of block calls is limited only by available memory.

Example

To call a block named *myBlock* in ST, with two inputs and one output, you can drag and drop *myBlock* from the Project tab of the Navigator to the ST editor. When you do this, an ST call statement with the formal call convention template is created as follows: Formal call convention:

```
myBlock(myInput1 := , myInput2 := , myOutput1 => , myOutput2 => );
```

After you complete the ST call statement by adding arguments, the ST statement may look as follows:

```
myBlock(myInput1 := 20, myInput2 := 30, myOutput1 => myResult1, myOutput2 => myResult2, Y0 =>
mySuccess);
```

Notes

- In a call or invocation statement using the formal call convention, the parameters can be in any order.
- The text *Y0 => mySuccess* is optional because Y0 *(also named ENO (ENable Output))* is an optional output parameter of type BOOL for all blocks.
- In ST, you can refer to Y0 or ENO in a call or invocation statement using the formal call convention only. Y0 and ENO refer to the same parameter and are interchangeable. Exception: using Y0 outside of its block. For example, in the ST statement mySuccess := myBlock.Y0;, you must use Y0. The ST statement mySuccess := myOtherBlock.ENO; is not valid.
- All required parameters of a block call, function call, or function block invocation statement must be present; optional parameters can be omitted.
- A single block call, function call, or function block invocation statement can use the informal call convention or the formal call convention, but not both.

Informal call convention:

```
myBlock(myInput1, myInput2, myOutput2, myOutput1);
```

Notes

- Input and output operands in a statement using the informal call convention must be specified in U-shaped order.
- For the informal call convention, all required and optional parameters in an ST statement must be present, except ENO (ENable Output, also named Y0), which can be used only in an ST statement using the formal call convention.

Function Call

(For PACSystems firmware version 2.50 and later only.)

Syntax

Function call: *Function_Name*(formal or informal argument list);

See also block call and function block invocation.

Operation

A Structured Text function call statement is a call to an *ST function*, using either the formal or informal call convention. The function then performs logical and arithmetic operations on the parameters as the function call statement is executed. After the called function execution is complete, the statement immediately following the call statement is executed.

Notes

- Function call statements can be used in any ST block, including the _MAIN block, parameterized blocks, and UDFBs.
- If a block or function block instance data has the same name as a function, compiling an ST block that contains a call to that name compiles the statement as a call to the block or function block invocation, and a warning displays in the Feedback Zone.
- The maximum number of function calls is limited only by available memory.

Examples

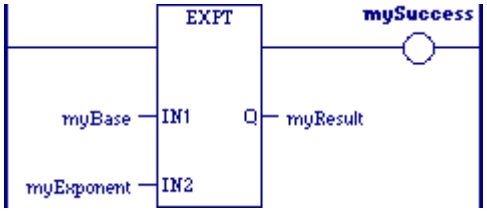
Example 1

```
abs_int(IN := myInput, Q => myOutput, ENO => mySuccess);
```

The ST statement above calls the ABS_INT function using the formal call convention.
IN := myInput specifies that the value of the input parameter variable named *myInput* is passed to the input parameter *IN* when the call is executed.
Q => myOutput specifies that the value of the output parameter *Q* is passed to the output variable parameter *myOutput* when the call is executed.
ENO => mySuccess (optional; can be used only with the formal call convention) specifies that the BOOL value of the output parameter *ENO (ENable Output, you can also use Y0)* is passed to the output variable parameter *mySuccess* when the call is executed.

Example 2

The following diagram displays LD logic that uses the LD exponent function named Expt.



To call the ST exponent function named *expt* in ST, the statements look like this:
Formal Call Convention:

```
expt(Q => myResult, IN1 := myBase, ENO => mySuccess, IN2 := myExponent);
```

Notes

- In a call or invocation statement using the formal call convention, the parameters can be in any order.
- The text *ENO => mySuccess* is optional because ENO *(ENable Output, also named Y0)* is an optional output parameter of type BOOL for all blocks, functions, and function blocks.
- In ST, you can refer to Y0 or ENO in a call or invocation statement using the formal call convention only. Y0 and ENO refer to the same parameter and are interchangeable. Exception: using Y0 outside of its UDFB. For example, in the ST statement mySuccess := myUDFB_Inst.Y0;, you must use Y0. The ST statement mySuccess := myUDFB_Inst.ENO; is not valid.
- All required parameters of a block call, function call, or function block invocation statement must be present; optional parameters can be omitted.
- A single block call, function call, or function block invocation statement can use the informal call convention or the formal call convention, but not both.

Informal call convention:

```
expt(myBase, myExponent, myResult);
```

Notes

- Input and output operands in a statement using the informal call convention must be specified in U-shaped order.
- For the informal call convention, all required and optional parameters in an ST statement must be present, except ENO (ENable Output, also named Y0), which can be used only in an ST statement using the formal call convention.

Example 3

```
abs_int(myInput, Q => myOutput);
```

The ST statement above is invalid because it mixes the call conventions: *myInput* uses the informal call convention; *Q => myOutput* uses the formal call convention.

Case ... of ... else

(For PACSystems firmware version 3.50 and later only.)

Syntax

case <integer_expression> **of**

<case 1>**:** <statement_list 1;>

<case 2>**:** <statement_list 2;>

...

<case n>**:** <statement_list n;>

else

<statement_list n+1;>

end_case;

<case 1> consists of

<N1, N2, ... Nm>

- and/or -

<L1..H1, L2..H2, ... , Ln..Hn>

- and/or -

both of the above.

where:

- **case**, **of**, **else**, and **end_case** are ST keywords.
- *integer_expression* is an ST expression that resolves to an INT, UINT, or DINT value.
- *<case 1>, <case 2>, ... , <case n>* contains one or more constants or ranges of constants of the same data type as *integer_expression*.
- *<statement_list 1;>, <statement_list 2;>, ... , <statement_list n;>* is a group of one or more ST statements that are executed only if integer_expression resolves to a value found in the corresponding *<case 1>, <case 2>, ... ,* or *<case n>*.
- *<statement_list n+1;>* is a group of one or more ST statements that are executed only if *integer_expression* resolves to a value that is not found in any of *<case 1>, <case 2>, ... , <case n>*.
- *N1, N2, ... , Nm* are corresponding INT, UINT, or DINT constants.
  - *N1* represents the first constant in the list.
  - *Nm* represents the last constant in the list.
- *L1..H1, L2..H2, ... , Ln..Hn,* are corresponding INT, UINT, or DINT ranges of constants.
  - *L1* represents the low constant of the first range in the list.

- .. (2 periods) denotes a range of constants.
- *H1* represents the high constant of the first range in the list.
- *Ln* represents the low constant of the last range in the list.
- .. (2 periods) denotes a range of constants.
- *Hn* represents the high constant of the last range in the list.

  Note: *H1* must be greater than *L1*; *Hn* must be greater than *Ln*.

Notes

- Data types in a *case ... of* construct cannot be mixed. The integer expression and all constants must be INT, UINT, or DINT.

- <*statement_list 1;*>, <*statement_list 2;*>, <*statement_list n;*>, <*statement_list n+1;*> may include any valid ST statements, including while, repeat, if, and nested *case ... of* constructs.

Operation

The ST *case ... of* construct statement specifies that statement groups are to be executed depending on the value contained in the associated INT, UINT, or DINT expression.
If the value of the INT, UINT, or DINT expression is contained in any of the INT, UINT, or DINT constants or constant ranges, then the associated statement group is executed.
(Optional.) If the value of the *INT, UINT, or DINT expression* is not contained in any of the constants and/or constant ranges, then the statement group following the *else* keyword <*statement_list 2;*> is executed.

Notes

- All constants must be of data type INT, UINT, or DINT, in decimal format.

- In an ST *case ... of* construct, there must be at least one ST statement after each colon (:); that is, after the last constant or constant range.

- There may be zero *else* conditions, in which case the **else** keyword is not present. However, it is good practice to include an *else* condition in a *case ... of* construct, for example, to detect an error condition.

- In an INT, UINT, or DINT constant range, the second constant of the range (the ending constant) must be at least 1 greater than the first constant of the range (the starting constant). For example, 0..1 is a valid range; 10..10 is not a valid range. All constants included in the range must **not** have been previously specified, either as a constant or in a constant range.

- INT, UINT, or DINT constants can be in any order, but they must not have been previously specified, either as a constant or in a constant range.

Operand Data Types

For valid operand data types for the *case ... of* construct, click here.

Limitations

- In a single *case ... of* construct, you can have a maximum of 1024 cases without the **else** keyword. If you need more than 1024 cases, insert an **else** keyword after

the last condition in the *case ... of* construct, and then insert another *case ... of* construct or a nested if... then... else statement after the **else** keyword.

- The number of constants and constant ranges in a single case statement is limited by the data type and the memory in your computer.
- The number of nested *case ... of* constructs and the number of levels of nested *case ... of* constructs is limited only by the memory in your computer.

Examples

A simple *case ... of* construct is the following:

```
case myInt of

0: myStatus := 1; // The value of myStatus becomes 1 if the value of myInt is zero.

end_case;
```

Let's add the *else* keyword to the above *case ... of* construct:

```
case myInt of

0: myStatus := 1; // The value of myStatus becomes 1 if the value of myInt is 0.

else

  myStatus := 2; // The value of myStatus becomes 2 if the value of myInt is not 0.

end_case;
```

Now let's replace the above 0 *case* statement with a range *case* statement:

```
case myInt of

0..10: myStatus := 0; // The value of myStatus becomes 0 if the value of myInt is between 0 and 10
inclusive.

else

  myStatus := 1; // The value of myStatus becomes 1 if the value of myInt is not between 0 and 10
inclusive.

end_case;
```

Finally, let's add some constants to the above *case* statement:

```
case myInt of

0..10, -1, -2: myStatus := 0; // The value of myStatus becomes 0 if the value of myInt is -1, -2, or
between 0 and 10 inclusive.

else

  myStatus := 1; // The value of myStatus becomes 1 if the value of myInt is not -1, -2, and not
between 0 and 10 inclusive.

end_case;
```

The *case ... of* construct example below is valid. The constants to the left of each colon (:) are compared with the value of the INT variable *TW*.

```
bcd4_to_int(IN := THUMBWHEEL, Q => TW);

TW_ERROR := 0;

case TW of

5,1:          DISPLAY := OVEN_TEMP;

2:            DISPLAY := MOTOR_SPEED;

3:            DISPLAY := GROSS - TARE;

6..10, 4:     DISPLAY := TW - 4;

else          DISPLAY := 0;

              TW_ERROR := 1;

end_case;

int_to_bcd4(IN := DISPLAY, Q => QW100);
```

If the value of *TW* is:	**ST statements executed:**
5 or 1	DISPLAY := OVEN_TEMP;
2	DISPLAY := MOTOR_SPEED;
3	DISPLAY := GROSS - TARE;
6 through 10 or 4	DISPLAY := TW - 4;
none of the above	DISPLAY := 0; TW_ERROR := 1;

The example below shows improper use of the *case ... of* construct.

```
bcd4_to_int(IN := THUMBWHEEL, Q => TW);

TW_ERROR := 0;

case TW of

5,1:        DISPLAY := OVEN_TEMP;

2..2:       DISPLAY := MOTOR_SPEED;

3:          DISPLAY := GROSS - TARE;

10..3,4:    DISPLAY := STATUS(TW - 4);

else        DISPLAY := 0;

            TW_ERROR := 1;

end_case;

int_to_bcd4(IN := DISPLAY, Q => QW100);
```

The above example is **invalid** because:

- 2..2: is not a valid range. The second constant of a range must be at least 1 greater than the first constant of the range.
- 10..3,4: is not valid for three reasons:
  - The constant 4 is already included in the range 3 through 10.
  - The constant 5 within the range 3 through 10 is already included in the first condition (5,1:).
  - The constant 3 in the range 10..3 is already specified as a constant in the statement immediately above the 10..3,4:... statement.

The following example shows the use of a nested *case* and a nested *if* statement.

```
bcd4_to_int(IN := THUMBWHEEL, Q => TW);
TW_ERROR := 0;
case TW of
0:                  DISPLAY := OVEN_TEMP;
2,1,-1,5:           DISPLAY := MOTOR_SPEED;
3:                  DISPLAY := GROSS - TARE;
9..10, 4:           DISPLAY := TW - 4;
else
    case DISPLAY of
    1,2,3:           TW := 1;
    4..10, 12, 14..20: TW := 2;
    else
        if TW > 1 then
            myStatus := 1;
        end_if;
    end_case;
    DISPLAY := 0;
    TW_ERROR := 1;
end_case;
int_to_bcd4(IN := DISPLAY, Q => QW100);
```

Comment

(For PACSystems firmware version 2.50 and later only.)

Syntax

```
(*
This block comment spans four lines
and is ignored by the compiler.
*)
```

- or -

```
myUINT := 10; 'This line comment is ignored by the compiler.
'myFinalResult := myInt2 + myInt4; 'This entire line is ignored.
```

- or -

```
myUINT := 10; //This line comment is ignored by the compiler.
//myFinalResult := myInt2 + myInt4; //This entire line is ignored.
```

Operation

Comments are permitted anywhere in the program where spaces are allowed. Comments are ignored by the ST compiler.

- Block comments are delimited at the beginning and end by the special character combinations (* and *), respectively.
- Line comments are delimited at the beginning by two forward slashes (//) or an apostrophe (').

Notes

- The use of nested block comments, for example, (* (* nested *) *) is not valid.
- The maximum number of characters allowed depends upon the memory in your computer.

Do

(PACSystems firmware version 2.50 and later only.)
The *do* keyword is required for the while ... do construct and the for ... do construct.

Syntaxes

while ... do ... construct

```
while boolean_expression do

<statement_list;>

end_while;
```

for ... do ... construct

```
for myControl_var := myStart_value to myEnd_value [by myStep_value] do

<statement_list;>

end_for;
```

Operation

In the ST *while ... do* construct statement, the <statement_list;> is executed repeatedly if the associated *boolean_expression* evaluates to 1 (True). When the associated *boolean_expression* evaluates to 0 (False), the *while ... do* construct has completed execution, and the first statement following the *end_while* keyword executes. If the state of the boolean expression is initially set to 0 (False), then the <statement_list;> is not executed.
In the ST *for ... do* construct statement, the <statement_list;> is executed repeatedly while *myControl_var* is iterated from the value of *myStart_value* to the value of *myEnd_value*. The values of *myStart_value*, *myEnd_value*, and *myStep_value* are calculated once only, at the beginning of the *for* loop. On the first iteration, the value of *myControl_var* is set to the value of *myStart_value*. The value of *myControl_var* is then incremented by the value of *myStep_value* at the end of each iteration. At the beginning of each iteration, the termination condition is evaluated. If it evaluates to 1 (True), then execution of the *for* loop ends, and the first statement following the *end_for* keyword executes.

Note: [**by** *myStep_value*] is optional; the value of *myStep_value* defaults to 1 if not specified.

Else

(For the *if ... then ...* construct, PACSystems firmware version 2.50 and later only.)
(For the *case* construct, PACSystems firmware version 3.50 and later only.)
The *else* keyword is optional for the if... then... else construct and the case ... of ... else construct.

Syntaxes

if ... then ... construct

```
if boolean_expression then
    <statement_list 1;>
else
    <statement_list 2;>
end_if;
```

case construct

```
case <integer_expression> of
<case 1>: <statement_list 1;>
<case 2>: <statement_list 2;>
...
<case n>: <statement_list n;>
else
<statement_list n+1;>
end_case;
```

Operation

In the ST *if ... then ...* construct statement, if the associated boolean expression evaluates to 0 (False), and there is an *else* keyword, then the statement group following the *else* is executed (<*statement_list 2;*>). If there is no *else* keyword, then the first statement following the *end_if* keyword is executed.
In the ST *case ... of* construct statement, if the associated value of the *INT, UINT, or DINT* expression is not contained in any of the *selector values* and *selector ranges*, then the statement group following the *else* keyword is executed (<*statement_list n+1;*>). If there is no *else* keyword, then the first statement following the *end_case* keyword is executed.

Exit

(For PACSystems firmware version 2.50 and later only.)

Syntax

exit;

Operation

The ST *exit* statement can be used only within a while, repeat, or for loop construct and enables execution to leave the loop prematurely. When an *exit* statement is reached, execution continues immediately from the end of the while, repeat, or for loop; no further part of the loop is executed.

Example

```
INT_index := 1;


while #ALW_ON do
    if INT_index = 100 then
        exit;
    end_if;

    INT_temp := @INT_index;
    INT_index := INT_index + 1;
    @INT_index := INT_temp;
end_while;


INT_Array[0] := INT;
```

In the above example, when the index is incremented to the last element of the array (*INT_Index* = 100), the state of the *if* condition is set to True(1), and then the exit statement is executed. This causes the execution to break out of the while loop construct; the statement after the *end_while* statement is then executed.

For ... Do

(For PACSystems firmware version 2.50 and later only.)
Syntax

for *myControl_var* := *myStart_value* **to** *myEnd_value* [**by** *myStep_value*] **do**

<*statement_list;*>

end_for;

where:

- *myControl_var* is the control variable or parameter.
- *myStart_value* is the start constant, variable, parameter, or expression.
- *myEnd_value* is the end constant, variable, parameter, or expression.
- *myStep_value* is the step constant, variable, parameter, or expression. *myStep_value* specifies the integer increment or decrement value for each iteration of the for loop.
- **for**, **to**, **by**, **do**, **end_for** are keywords.
- <*statement_list*;> is one or more ST statements; this may include if, while, repeat, case, and nested *for* constructs.

Note: [**by** *myStep_value*] is optional; the value of *myStep_value* defaults to 1 if not specified.

Operation

The values of *myStart_value*, *myEnd_value*, and *myStep_value* are calculated once only, at the beginning of the *for* loop. On the first iteration, the value of the control variable *myControl_var* is set to the value of *myStart_value*. The value of *myControl_var* is then incremented by the value of *myStep_value* (optional, defaults to 1) at the end of each iteration. At the beginning of each iteration, the termination condition is evaluated; if it evaluates to 1 (True), then execution of the *for* loop ends and the first statement after the *end_for* keyword executes.
The termination condition of a *for* loop depends on the value of *myStep_value*.

Value of myStep_value	**Termination condition**
Positive	*myControl_var* > *myEnd_value*
Negative	*myControl_var* < *myEnd_value*
Zero	None; that is, a zero causes an infinite loop, because a termination condition is never reached.

As with the other ST iteration statements (while, repeat), loop execution can be prematurely halted by an exit statement. **This is necessary if myStep_value is set to 0 (zero); otherwise, the watchdog timer expires and only power cycling can restart the controller.**

To prevent endless or unpredictable *for* loops, we recommend the following:

- Do **not** use the control variable or control parameter outside the for loop.
- Ensure that the <*statement_list;*> within a *for* loop does **not** modify the value of the control variable.

The ST compiler supports the nesting of *for* loops to a maximum of 10 levels.

Operands Data Types

- *myStart_value*, *myEnd_value*, and *myStep_value* can be DINT, INT, or UINT variables, expressions, parameters, or constants.
- *myControl_var* can be a DINT, INT, or UINT variable or parameter.
- The data types cannot be mixed; that is, *myControl_var*, *myStart_value*, *myEnd_value*, and *myStep_value* must all be DINT, INT, or UINT.

Example

```
/*
/* Load zero into each position of a two dimensional INT
/* or UINT array named index with 10 rows and 10 columns.
/* The array index is mapped to memory beginning at
/* location %R200.
/*
base := 200;
max_rows := 10;
max_cols := 10;
for myRows := 0 to (max_rows - 1) do /* loop through each of the 10 rows of the array index
for myCols := 0 to (max_cols - 1) do /* loop through each of the 10 columns
    index := myRows * max_cols + myCols + base; /* calculate index position
  @index := 0; /* load 0 to the memory location of the value in @index
                    /* @index performs an indirect reference operation on the variable named
                    index
                /* the above two statements are performed 100 times: 10 rows times 10 columns
end_for; /* the inner loop for the 10 columns for 1 row is complete
end_for; /* the outer loop for the 10 rows is complete
```

The following events occur during the execution of the above example:

1. At the beginning of the outer *for* loop, the value of *myRows* is set to 0.

2. The value of *myRows* is compared to the value of *max_rows - 1*. If the value of *myRows* is less than or equal to the value of *max_rows - 1*, execution proceeds to the inner *for* loop.
3. At the beginning of the inner *for* loop, the value of *myCols* is set to 0.
4. The value of *myCols* is compared to the value of *max_cols - 1*. If the value of *myCols* is less than or equal to the value of *max_cols - 1*, the following occurs:
   - The array *index* is calculated.
   - The indirect reference at memory location *@index* is set to zero.
   - The value of *myCols* is increased by 1 (because the *by* keyword has been omitted, the default increment is 1).
   - Execution proceeds to the beginning of step 4 of this example.

   If the value of *myCols* is greater than the value of *max_cols - 1*, execution returns to the outer *for* loop.
5. The value of *myRows* is increased by 1 (because the *by* keyword has been omitted, the default increment is 1), and the value of *myRows* is compared to the value of *max_rows - 1*.
6. If the value of *myRows* is less than or equal to the value of *max_rows - 1*, execution proceeds at step 3 of this example.
   If the value of *myRows* is greater than the value of *max_rows - 1*, both *for* loops have completed execution. The first statement after the second *end_for* keyword is executed, if it exists.

Formal Call Convention

(For PACSystems firmware version 2.50 and later only.)

Syntax

Name(IN1 := inVar_1, IN2 := inVar_2, ... , IN*n* := inVar_*n*, OUT1 => outVar_1, OUT2 => outVar_2, ... , OUT*n* => outVar_*n*[, ENO => mySuccess]);

Notes

- The text inside the square brackets [ ] is optional and does not have to be included.
- In a call or invocation statement using the formal call convention, the parameters can be in any order.

Operation

A call or invocation statement using the formal call convention can be used in a block call, function call, or function block call or invocation statement.

Syntax Definition

The following explanations refer to the above syntax example:

- *Name* is the name of the block or *function* being called, or the name of the *instance data* assigned to the instance of the function block being invoked.
- *IN1*, *IN2*, *... , INn* are the input parameter identifiers of the block call, function call, or function block invocation statement using the formal call convention. Input parameter identifiers specify which parameters receive the input values when the call or invocation is executed.
- *:=* (in a block call or invocation statement using the formal call convention) is the assignment operator to assign the value of an argument to an input parameter.
- *inVar_1*, *inVar_2*, *... , inVar_n* are arguments, which can be variables, parameters, or constants. They contain the values being passed into the block call or invocation statement using the formal call convention.
- *OUT1*, *OUT2*, *... , OUTn* are the output parameter identifiers of the block call or invocation statement, using the formal call convention. Output parameter identifiers specify which parameters the assigned output arguments are associated with when the call or invocation is executed.
- *=>* (in a block call or invocation statement using the formal call convention), is the assignment operator to assign the value of an output parameter to its associated argument.
- *outVar_1, outVar_2, ... , outVar_n* are the output arguments that receive the values of the assigned output parameters when the call or invocation statement is executed.
- *ENO => mySuccess* (optional) specifies that the BOOL value of the output parameter ENO (ENable Output, also named Y0) for a block call, function call, or

function block invocation statement using the formal call convention, is passed to the variable or parameter *mySuccess* when the call or invocation statement is executed.

Notes

- ENO *(ENable Output, also named Y0)* is an optional output parameter of type BOOL for all blocks, functions, and function blocks.
- In ST, you can refer to Y0 or ENO in a call or invocation statement using the formal call convention only. Y0 and ENO refer to the same parameter and are interchangeable. Exception: using Y0 outside of its function block. For example, in the ST statement mySuccess := myFunctionBlock_Inst.Y0;, you must use Y0. The ST statement mySuccess := myFunctionBlock_Inst.ENO; is not valid.
- In a call or invocation statement using the formal call convention, the parameters can be in any order.
- All required parameters of a block call, function call, or function block invocation statement must be present; optional parameters can be omitted.
- A single block call, function call, or function block invocation statement can use the informal call convention or the formal call convention, but not both.

Examples

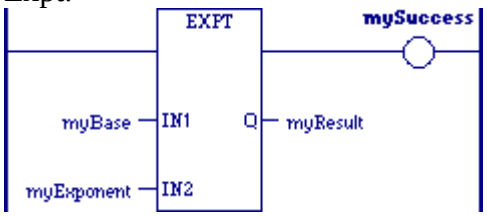
Example 1

```
abs_int(IN := myInput, Q => myOutput, ENO => mySuccess);
```

The ST statement above calls the ABS_INT function using the formal call convention.
IN := myInput specifies that the value of the input parameter variable named *myInput* is passed to the input parameter *IN* when the call is executed.
Q => myOutput specifies that the value of the output parameter *Q* is passed to the output variable parameter *myOutput* when the call is executed.
ENO => mySuccess (optional; can be used only with the formal call convention) specifies that the BOOL value of the output parameter *ENO (ENable Output, you can also use Y0)* is passed to the output variable parameter *mySuccess* when the call is executed.

Example 2

The following diagram displays LD logic that uses the LD exponent function named Expt.



To call the ST exponent function named *expt* in ST with a call statement using the formal call convention, the statement may look like this:

```
expt(Q => myResult, IN1 := myBase, ENO => mySuccess, IN2 := myExponent);
```

Example 3

```
abs_int(myInput, Q => myOutput);
```

The ST statement above is invalid because it mixes the call conventions: *myInput* uses the informal call convention; *Q => myOutput* uses the formal call convention.

Function Block Invocation Statement

ST supports invocation statements for the following kinds of function blocks:

- Standard function blocks
- Specialty function blocks
- User-defined function blocks (UDFBs)

Note: ST does not support built-in function blocks.

Syntax

function block invocation: *Instance_Variable_Name*(formal or informal argument list);

Tip: See also block call and function call.

Operation

A Structured Text function block invocation statement invokes a function block by means of an instance variable; that is, you must specify the instance variable in the invocation statement, not the function block name. The invocation can use either the formal or informal call convention. The invoked function block instance then performs logical or arithmetic operations on the parameters as the statement is executed. After the invoked function block has executed, the statement immediately following the invocation statement is executed.

The function block instance data persists in the instance variable associated with the function block instance until an invocation statement that uses the same instance data executes.

To complete a standard function block invocation statement:

Create an instance of the standard function block in the ST editor.

To complete a user-defined function block (UDFB) invocation statement:

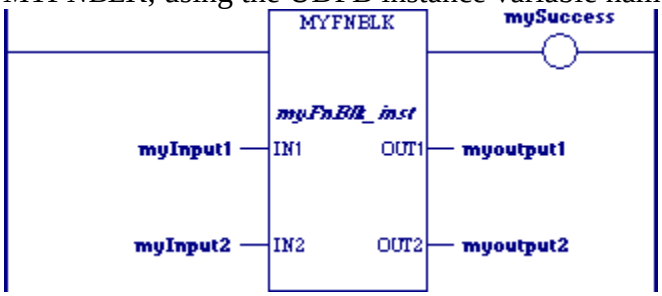
1. Create a UDFB.
2. Set the UDFB's input and output parameters and member variables.
3. Create an instance of the UDFB in the ST editor.

Notes

- An ST function block invocation statement can be used in any ST block, including the _MAIN block, parameterized blocks, and UDFBs.
- The maximum number of invocation statements is limited only by available memory.
- If you select a function block *instance* variable in an ST block, the following takes place:
  - If the Inspector is open, it displays the properties of the function block instance variable. To open the Inspector, press SHIFT+F7.
  - If the Feedback Zone is open, its References tab displays the location of all instances of the function block being invoked. To open the Feedback Zone, press SHIFT+F6.

UDFB invocation statement example

The following diagram displays an LD UDFB instance of the UDFB named *MYFNBLK*, using the UDFB instance variable named *myFnBlk_inst*.



To invoke the same UDFB named *MYFNBLK* in ST, use the UDFB instance variable named *myFnBlk_inst* as follows:

Formal call convention:

myFnBlk_inst(IN1 := myInput1, IN2 := myInput2, OUT1 => myOutput1, OUT2 => myOutput2, ENO => mySuccess);

Notes

- For UDFBs, the text *Y0 => mySuccess* is optional (for UDFBs, you must used Y0, not ENO) because Y0 *(also named ENO)* is an optional output parameter of type BOOL for all blocks, functions, and function blocks.
- For standard function blocks, the text ENO => mySuccess is optional (for standard function blocks, you must used ENO, not y0) because ENO (ENable Output, also named y0) is an optional output parameter of type BOOL for all blocks, functions, and function blocks.

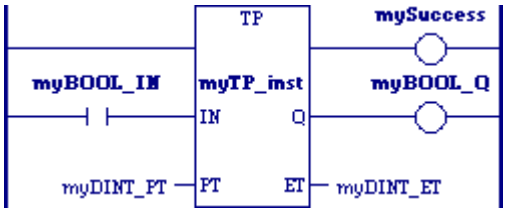
Informal call convention:

myFnBlk_inst(myInput1, myInput2, myOutput2, myOutput1);

Note: Input and output operands in a statement using the informal call convention must be specified in U-shaped order.

Standard function block invocation statement example

The following diagram displays an instance of a TP timer standard function block. The instance uses the instance variable named *myTP_inst*.



Note: (LD only.) Both the input, IN, and the output, Q, must be power flow.

To invoke the same TP timer standard function block in ST by using the instance variable named *myTP_inst*, the statement looks like this:

Formal call convention

myTP_inst(IN := myBOOL_IN, PT := myDINT_PT, ET => myDINT_ET, ENO => mySuccess, Q => myBOOL_Q);

Note: The text *ENO => mySuccess* is optional because ENO *(ENable Output, also named Y0)* is an optional output parameter of type BOOL for all blocks, functions, and function blocks.

Informal call convention

myTP_inst(myBOOL_IN, myDINT_PT, myDINT_ET, myBOOL_Q);

Note: Input and output operands in a statement that uses the informal call convention must be specified in U-shaped order.

If ... Then ... Else

(For PACSystems firmware version 2.50 and later only.)

Syntax

if boolean_expression 1 **then**

    <statement_list 1;>

elsif boolean_expression 2 **then**

    <statement_list 2;>

else

    <statement_list 3;>

end_if;

where:

- **if**, **then**, **elsif**, **else**, and **end_if** are ST keywords. **elseif** can also be used in place of **elsif**.
- *boolean_expression 1* and *boolean_expression 2* are ST expressions that resolve to boolean values: 1 (True) or 0 (False).
- *statement_list 1*, *statement_list 2*, and *statement_list 3* are one or more ST statements.

Note: <*statement_list 1;*>, <*statement_list 2;*>, and <*statement_list 3;*> may be one or more ST statements; this may include while, repeat, and nested *if* constructs.

Operation

The ST *if* statement specifies that a group of statements is to be executed only if the associated state of the Boolean expression evaluates to 1 (True). If the state of the boolean condition evaluates to 0 (False), then either no statement is executed, or the statement group following the *else* keyword (or the *elsif* keyword if its associated Boolean condition evaluates to 1 (True)) is executed.

Notes

- There may be zero or more *elsif* conditions in a single *if* statement.
- There may be zero *else* conditions.
- If no *else* statement exists in an *if ... then ...* construct, and all of the *if* or *elsif* conditions evaluate to 0 (False), then the statement bodies are not executed.
- The number of levels of nested *if* statements in an ST block is limited only by the memory on your computer.

Example

```
if (myDint = 0) then

    myTag := 1;

elsif (myDint2 > 0) then

    myTag := 2;

else

    myTag := 3;

end_if;
```

In the above example, the value of the DINT variable *myDint* is compared to zero. If they are equal, the value 1 is assigned to the variable *myTag*. If the value of *myDINT* is not zero, then the value of the DINT variable *myDint2* is compared to zero. If the value of *myDint2* is greater than zero, then the value 2 is assigned to *myTag*. If the value of *myDINT* is not zero and if the value of *myDINT2* is not greater than zero, then the value 3 is assigned to *myTag*.

Example

```
if SwitchPosition = 0 then
    ConveyorSpeed := 0; ' Conveyor is stopped
elsif ((SwitchPosition >= 1) AND (SwitchPosition <= 2)) then
    ConveyorSpeed := 2 * SwitchPosition; ' Run at slow speeds
elsif ((SwitchPosition >= 3) AND (SwitchPosition <= 5)) then
    ConveyorSpeed := 5 * SwitchPosition; ' Run at high speeds
else
    ConveyorSpeed :=0; ' Stop the conveyor on any bad input
    ConveyorFault := #ALW_ON;
end_if;
```

Informal Call Convention

U-shaped order
(For PACSystems firmware version 2.50 and later only.)

Syntax

Name(Input_Param_1, Input_Param_2, ... , Input_Param_n, Output_Param_n, Output_Param_n-1, Output_Param_n-2, ... , Output_Param_1);

Operation

A call or invocation statement using the formal call convention can be used in a block, function, or function block.

Syntax Definition

The following explanations refer to the above syntax example:

- *Name* is the name of the block or function being called, or the name of the instance data assigned to the instance of the function block being invoked.
- *Input_Param_1, Input_Param_2, ... , Input_Param_n* are the input variables or parameters containing the values being passed to the input parameters of the block, function, or function block when the call or invocation statement is executed.
- *Output_Param_n, Output_Param_n-1, Output_Param_n-2, ... , Output_Param_1* are the output variables or parameters containing the values being passed to the output parameters of the block, function, or function block when the call or invocation statement is executed.

Notes

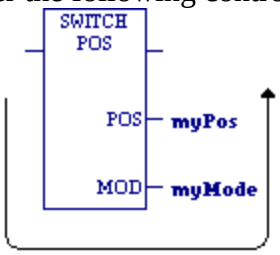
- A single ST statement can use the informal call convention or the formal call convention, but not both.
- For the informal call convention, all required and optional parameters in an ST statement must be present, except ENO (ENable Output, also named Y0), which can be used only in an ST statement using the formal call convention.

U-shaped Order

Input and output operands in a statement using the informal call convention must be specified in **U-shaped** order. The **U-shaped** order begins with the first operand (in an LD call, the top left input operand), then proceeds to the second operand (in an LD call, the bottom or next lower left input operand), then proceeds counter-clockwise to the next operand (in an LD call, the bottom right output operand), and then proceeds to the next operand (in an LD call, the top or the next higher right output operand). See Example 1 below.

Example 1

Consider the following controller LD function named Switch_Pos.



To call the ST function *Switch_Pos* using the informal call convention, the ST statement looks like this:

```
switch_Pos(myMode, myPos);
```

Note the U-shaped order of the parameters: *myMode* must be first; *myPos* must be last.

Example 2

```
abs_int(myInput, myOutput);
```

The ST statement above calls the ST *function* abs_int using the informal call convention.

- *myInput* is the input argument.
- *myOutput* is the output argument.

```
my_UDFB_inst_name(myInput, myOutput);
```

The ST statement above invokes a UDFB using the UDFB instance variable named *my_UDFB_inst_name*, using the informal call convention.

Repeat ... Until

(For PACSystems firmware version 2.50 and later only.)

Syntax

repeat

<statement_list;>

until <boolean_expression>

end_repeat;

Note: <*statement_list;*> may be one or more ST statements; this may include if, while, and nested *repeat* constructs.

Operation

The *repeat ... until* construct statement causes the sequence of statements up to the *until* keyword to execute repeatedly, and at least once, until the state of the associated Boolean *expression* evaluates to 1 (True).

Notes

- Unlike a *repeat ... until* construct, in a while ... do construct, the <*statement_list;*> group of statements may not be executed.
- The number of levels of nested *repeat* statements in an ST block is limited only by the memory on your computer.

Example

```
INT_index := 1;


repeat
    INT_temp := @INT_index;
    INT_index := INT_index + 1;
    @INT_index := INT_temp;
    until INT_index > 99
end_repeat;


INT_Array[0] := INT;
```

In the above example, each time through the *repeat ... until* loop, the variable *INT_index* is incremented by 1. When *INT_index* receives the value 100, the *end_repeat* statement is executed, and the *repeat ... until* loop is terminated.

Return

(For PACSystems firmware version 2.50 and later only.)

Syntax

return;

Operation

The *return* statement provides an early exit from a block or a user-defined function block (UDFB), for example, as the result of the evaluation of an if statement.

Example

```
if my_int > 100 then

        return;

end_if;

my_int := my_int + 1;
```

When the  ST block containing the above *if* statement is called, and when the value of the  variable *my_int* is greater than 100, then the following statements are executed:

- The *return* statement
- The statement after the block call statement in the calling block or invoking UDFB

Likewise, when the  ST UDFB containing the above *if* statement is invoked, and when the value of the variable *my_int* is greater than 100, then the following statements are executed:

- The *return* statement
- The statement after the invocation statement in the calling block or invoking UDFB

While ... Do

(For PACSystems firmware version 2.50 and later only.)
Syntax

while boolean_expression **do**

    <statement_list;>

end_while;

Note: <*statement_list;*> can be one or more ST statements; this may include if, repeat, and nested *while* constructs.

Operation

The *while ... do* construct statement causes the sequence of statements up to the *end_while* keyword to execute repeatedly until the state of the associated Boolean *expression* evaluates to 0 (False). If the state of the boolean expression is initially set to 0 (False), then the <*statement_list;*> group of statements is not executed.

Notes

- Unlike a *while ... do* loop construct, in a repeat ... until loop construct, the statements are executed at least once.
- The number of levels of nested *while ... do* statements in an ST block is limited only by the memory on your computer.

Example

```
INT_index := 1;

while INT_index < 100 do

    INT_temp := @INT_index;

    INT_index := INT_index + 1;

    @INT_index := INT_temp;

end_while;

INT_Array[0] := INT;
```

- While the value of the variable *Int_Index* is less than the constant 100, the statements between the *do* keyword and the *end_while* statement are executed.
- When the value of the variable *Int_Index* is equal to 100, then the statements following the *end_while* statement are executed.

Timers

Working with standard function blocks

Standard Function Block	Example
Timer Off Delay (TOF)	myTOF_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO => outBool);
Timer On Delay (TON)	myTON_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO => outBool);
Timer Pulse (TP)	myTP_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO => outBool);

Notes

- Forcing or writing values to the IN, PT, Q, ET, ENO, or TI timer standard function block instance data elements may cause indeterminate results.
- TOF requires instance data of data type TOF; TON requires instance data of data type TON; TP requires instance data of data type TP.

TOF, TON, TP Timer Standard Function Blocks

Syntax

Formal convention:

```
myTOF_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO =>
outBoolSuccess);

myTON_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO =>
outBoolSuccess);

myTP_Instance_Data(IN := inBool, PT := inDINT, ET => outDINT, Q => outBool, ENO => outBoolSuccess);
```

Informal convention:

```
myTOF_Instance_Data(inBool, inDINT, outDINT, outBool);

myTON_Instance_Data(inBool, inDINT, outDINT, outBool);

myTP_Instance_Data(inBool, inDINT, outDINT, outBool);
```

Note: The operands in the statement above must be in U-shaped order when using the informal convention.

Operation

For details, see operation of timer standard function blocks.
ENO is an optional BOOL output parameter. If ENO is present in a statement that uses the formal convention, then the state of *outBoolSuccess* is set to 1 (call was successful) or 0 (call failed).

Instruction or statement	**LD instruction receives power**	**Equivalent ST called statement**
Successful	Instruction passes power	ENO set to 1 (True)
Failed	Instruction does not pass power	ENO set to 0 (False)

For details on valid operands, data types, memory areas, and examples, see the timer standard function blocks.

Notes

- Forcing or writing values to the IN, PT, Q, ET, ENO, or TI timer standard function block instance data elements may cause indeterminate results.
- TOF requires instance data of data type TOF; TON requires instance data of data type TON; TP requires instance data of data type TP.
- Instance data can be a parameter of the current user-defined function block (UDFB) or parameterized block, or it can be a variable.

Other Languages

For details on the timer standard function block syntax in other languages, see LD and FBD.

Index

Addendum

Communication

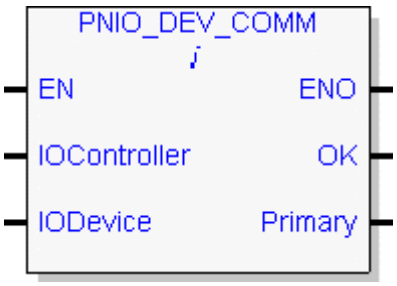
(For PACSystems firmware version 7.00 and later.)

Instruction	*Mnemonics*	*Description*
PNIO_DEV_COMM	PNIO_DEV_COMM	Verifies whether a Profinet Controller (PNC) is communicating with a Profinet IO-device and, in Hot Standby redundancy systems, whether the PNC is the device's primary IO-Controller.

PNIO_DEV_COMM

LD

FBD



ST

Formal convention:

```
PNIO_DEV_COMM(IOController := [input], IODevice := [input], OK => [output], Primary => [output]);
```

Tip: To spare yourself some typing, drag the instruction from the Toolchest's LD Instructions or FBD Instructions drawer to your ST logic. All the input and output names are inserted into your ST logic. You need only to supply the variable names or constants and remove the optional inputs or outputs you don't want to use.

Operation

PNIO_DEV_COMM verifies whether the Profinet Controller (PNC) identified by input IOController and the Profinet IO-device identified by input IODevice are communicating with one another. If so, output OK is set to On; otherwise, it is set to Off.

When PNIO_DEV_COMM is used in a Hot Standby (HSB) redundancy system, if OK is set to On, PNIO_DEV_COMM verifies whether the PNC is the device's Primary IO-Controller. If so, the Primary output is set to On; otherwise, the output is set to Off.

When PNIO_DEV_COMM is not used in an HSB redundancy system, Primary is set to the same value as OK.

Operands

Input Operands

Operand	*Data Type*	*Memory Area*	*Description*

i	(FBD only.) Solve order for the instruction.		
Power flow (LD)			When On, PNIO_DEV_COMM executes. When Off, PNIO_DEV_COMM does not execute.
EN (FBD)	BOOL variable or BOOL system variable	I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, discrete I/O variable	Enable input. When On, PNIO_DEV_COMM solves. When Off, PNIO_DEV_COMM does not solve.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, non-discrete I/O variable	
IOController	PNIO_CONTROLLER_REF variable	symbolic, I/O variable. LD and FBD also support data flow.	Variables used to identify respectively the Profinet Controller and Profinet device whose intercommunication status is to be reported
IODevice	PNIO_DEVICE_REF variable	symbolic, I/O variable. LD and FBD also support data flow.	

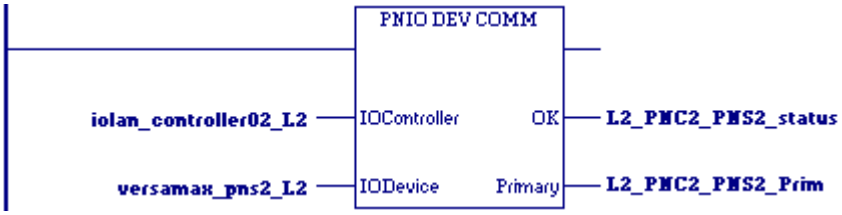
Output Operands

Operand	*Data Type*	*Memory Area*	*Description*
Power flow (LD; optional.)			There is power flow when PNIO_DEV_COMM executes successfully.
ENO (FBD; optional.)	BOOL variable	data flow, I, Q, M, T, S, SA, SB, SC, G, discrete symbolic, discrete I/O variable	ENO is set to On if EN is On.
	Bit reference in non-BOOL variable	R, P, L, AI, AQ, W, I, Q, M, T, G, non-discrete symbolic, non-discrete I/O variable	
OK	▪ LD: Power flow		(Optional.) Set to On if the specified IO Controller and IO device are currently communicating.

	▪ FBD and ST: BOOL variable		
Primary	▪ LD: Power flow ▪ FBD and ST: BOOL variable		(Optional.) Set to On if the PNC is the Primary IO Controller of the device in an HSB redundancy system. Requires communications with the device (OK is set to On). In a simplex system, Primary is set to On if OK is set to On.

Example

PNIO_DEV_COMM is used in an HSB redundancy system. It verifies whether the Profinet Controller (PNC) identified by the variable iolan_controller02_L2 and the Profinet device identified by the variable versamax_pns2_L2 are currently communicating with one another. If so, the BOOL variable L2_PNC2_PNS2_Prim is set to On. PNIO_DEV_COMM also verifies whether the identified PNC is the primary IO-Controller of the device; if so, the BOOL variable L2_PNC2_PNS2_status is set to On.



CPU Support

PNIO_DEV_COMM is supported for PACSystems CPUs with firmware version 7.00 or later.