Logic Developer - PC: ST Instructions

ST instructions are the building blocks Structured Text is composed of. Instructions and operators can be inserted into expressions to create statements, which are simple executable units of logic. Each instruction performs an operation on variables or constants defined for the target the ST block is associated with.

In the following list, ST logic instructions are grouped functionally, according to the type of operation performed:

- Advanced math
- Basic ST instructions
- Bit Shift Rotate
- Conditionals
- Data type conversions
- Iterations
- Strings
- Timer

See some of these instructions in ladder and FBD.

Basic Instructions (ST)

Logical | Bitwise | Conversion | Math | Relational

Most of the basic ST instructions support operations on BOOL, DINT and LREAL variables.

Note: An instance of a UDFB can be used in ST logic.

Logical Operators (ST)

NOT

Operand can be BOOL variable, expression or constant. The state of *Operand* is toggled.

Example

```
myBOOL := NOT(myBOOL1);
```

AND

Operands can be BOOL variables, expressions or constants. A logical AND is performed.

Example

```
myBOOL := myBOOL1 AND myBOOL2;
```

OR

Operands can be BOOL variables, expressions or constants. A logical OR is performed.

Example

```
myBOOL := myBOOL1 OR myBOOL2;
```

XOR

Operands can be BOOL variables, expressions or constants. A logical XOR is performed.

Example

```
myBOOL := myBOOL1 XOR myBOOL2;
```

Note: For logical instructions, all operands must be BOOL variables, constants, or resolve to a BOOL value.

Bitwise Operators (ST)

- NOT
- AND
- OR
- XOR

Note: All of the above operands must be DINT values or resolve to DINT values.

Conversion Operators (ST)

- BOOL_TO_INT
- INT_TO_BOOL
- INT_TO_REAL
- REAL_TO_INT

Math Operators (ST)

+

Operands can be DINT or LREAL variables, expressions or constants. Adds both operands together.

Note: Both operands must be either DINT or LREAL; they cannot be mixed.

Examples

```
myDINT := myDINT + 1; 'valid
myLREAL := myLREAL + 1.0; 'valid

myDINT := 2.5 + 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 2 + 1;
'the above is invalid, attempting to assign a DINT result to an LREAL
variable

myDINT := myDINT + 1.0;
myLREAL := myLREAL + 1;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

-

Negation or subtraction.

Operands can be BOOL, DINT or LREAL variables, expressions or constants. If only one operand is supplied, the operator is negation. If two operands are supplied, the operator is subtraction.

Negation

If Operand is BOOL, the state of Operand is toggled.

Examples

```
myBOOL := -#ALW_OFF;
'the above is valid because a BOOL is assigned to a BOOL. In this case,
the state of myBOOL is turned ON(1).

myDINT := -myDINT2; 'valid statement because a DINT is assigned to a
DINT.
myLREAL := -1.0; 'valid statement because an LREAL is assigned to an
```

```
LREAL.

myDINT := -1.0;
'The above is invalid because -1.0 is an LREAL constant being assigned to
a DINT variable.

myLREAL := -2;
'The above statement is invalid because -2 is a DINT constant being
assigned to an LREAL variable.
```

Subtraction

Subtracts *operand2* from *operand1*.

Note: Both operands must be either DINT or LREAL variables, expressions or constants; they cannot be mixed.

Examples

```
myDINT := myDINT - 1; 'valid
myLREAL := myLREAL - 1.0; 'valid

myDINT := 2.5 - 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 2 - 1;
'the above is invalid, attempting to assign a DINT result to an LREAL
variable

myDINT := myDINT - 1.0;
myLREAL := myLREAL - 1;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

*

Operands can be DINT or LREAL variables, expressions, or constants. Multiplies *Operand1* by *Operand2*.

Note: Both operands must be either DINT or LREAL; they cannot be mixed.

Examples

```
myDINT := myDINT * -1; 'valid
myLREAL := myLREAL * 2.0; 'valid
```

```
myDINT := 2.5 * 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 2 * 2;
'the above is invalid, attempting to assign a DINT result to an LREAL
variable

myDINT := myDINT * -1.0;
myLREAL := myLREAL * 1;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

Operand1 / Operand2

Operands can be DINT or LREAL variables, expressions or constants. Divides *Operand1* by *Operand2*.

Notes

- Both operands must be either DINT or LREAL; they cannot be mixed.
- Dividing 2 DINTs results in the fraction being lost.
- #Overflow is set to ON (1) on division by zero.

Examples

```
myDINT := 5 / 2; 'valid; the result is a DINT 2
myLREAL := 5.0 / 2.0; 'valid; the result is LREAL 2.0

myDINT := 2.5 / 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 4 / 2;
'the above is invalid, attempting to assign a DINT result to an LREAL
variable

myDINT := myDINT / 2.0;
myLREAL := myLREAL / 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

MOD

Operands can be DINT or LREAL variables, expressions, or constants. Returns the remainder when *Operand1* is divided by *Operand2*.

Note: Both operands must be either DINT or LREAL; they cannot be mixed.

Examples

```
myDINT := 5 MOD 3; 'valid; the result is a DINT 2
myLREAL := 5.0 MOD 3.0; 'valid; the result is LREAL 2.0

myDINT := 2.5 MOD 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 4 MOD 2;
'the above is invalid, attempting to assign a DINT result to an LREAL
variable

myDINT := myDINT MOD 2.0;
myLREAL := myLREAL MOD 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

Relational Operators (ST)

For all relational operators:

- Operands must both be DINT or LREAL; you cannot mix data types in an expression.
- Operands can be variables, expressions or constants.
- You cannot compare two BOOL values.
- The result is always BOOL.

>

If *Operand1* is greater than *Operand2*, the logical expression resolves to True. If *Operand1* is less than or equal to *Operand2*, the logical expression resolves to False.

Examples

```
myBOOL := 5 > 3; 'valid; the result is On(1)
myBOOL := 2.9 > 3.0; 'valid; the result is Off(0)

myDINT := 2.5 > 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT > 2.0;
myBOOL := myLREAL > 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

>=

If *Operand1* is greater than or equal to *Operand2*, the BOOL expression resolves to True. If *Operand1* is less than *Operand2*, the BOOL expression resolves to False.

Examples

```
myBOOL := 5 >= 3; 'valid; the result is On(1)
myBOOL := 2.9 >= 3.0; 'valid; the result is Off(0)

myDINT := 2.5 >= 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT >= 2.0;
myBOOL := myLREAL >= 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

=

If *Operand1* is exactly equal to *Operand2*, the BOOL expression resolves to True. If *Operand1* is not equal to *Operand2*, the BOOL expression resolves to False.

Warning: Comparing LREAL values may produce unexpected results. For example, a calculation may result in 1.99999999999, which is not equal to 2.00000000000.

Examples

```
myBOOL := 5 = 3; 'valid; the result is Off(0)
```

```
myBOOL := 5.0 = 5.0; 'valid; the result is On(1)

myDINT := 2.5 = 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT = 2.0;
myBOOL := myLREAL = 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

<> or !=

If *Operand1* is not equal to *Operand2*, the BOOL expression resolves to True. If *Operand1* is exactly equal to *Operand2*, the BOOL expression resolves to False.

Warning: Comparing LREAL values may produce unexpected results. For example, a calculation may result in 1.99999999999, which is not equal to 2.00000000000.

Examples

```
myBOOL := 5 <> 3; 'valid; the result is On(1)
myBOOL := 5.0 <> 5.0; 'valid; the result is Off(0)

myDINT := 2.5 <> 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT <> 2.0;
myBOOL := myLREAL <> 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

<=

If *Operand1* is less than or equal to *Operand2*, the BOOL expression resolves to True. If *Operand1* is greater than *Operand2*, the BOOL expression resolves to False.

Examples

```
myBOOL := 5 <= 3; 'valid; the result is Off(0)
myBOOL := 2.9 <= 3.0; 'valid; the result is On(1)
```

```
myDINT := 2.5 <= 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT <= 2.0;
myBOOL := myLREAL <= 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

<

If *Operand1* is less than *Operand2*, the logical expression resolves to True. If *Operand1* is greater than or equal to *Operand2*, the logical expression resolves to False.

Examples

```
myBOOL := 5 < 3; 'valid; the result is Off(0)
myBOOL := 2.9 < 3.0; 'valid; the result is On(1)

myDINT := 2.5 < 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT < 2.0;
myBOOL := myLREAL < 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

Note: In an ST Relational expression, both operands must be the same data type.

Advanced Math Instructions (ST)

These ST instructions perform trigonometric, exponential and logarithmic operations on data. Only LREAL values are allowed for all advanced math functions except EXPT or ABS; these can be either DINT or LREAL values. The return value of all advanced math functions are LREAL except for ABS and EXPT. These two functions return the data type of the input operands.

See also Ladder Advanced Math Instructions. The functionality, operands, and CPU support for ST instructions is the same as for LD functions.

Absolute Value	ABS
Square Root	SQRT
Cosine	COS
Sine	SIN
Tangent	TAN
Arc Cosine	ACOS
Arc Sine	ASIN
Arc Tangent	ATAN
Arc Tangent (Operand1 / Operand2)	ATAN2
Hyperbolic Cosine	COSH
Hyperbolic Sine	SINH
Hyperbolic Tangent	TANH
Degrees to Radians	DTOR
Radians to Degrees	RTOD
Log	LOG

Natural Log	LN
Natural Exponent	EXP
Exponent (operand1 ^{operand2})	EXPT
Timer	Timer

Bit Shift/Rotate Instructions (ST)

These instructions are used to reposition the bits in DINT variables or in single elements of DINT arrays.

Instruction	Example
Rotate Left (ROL)	mydint := ROL(mydint1, mydint2);
Rotate Right (ROR)	mydint := ROR(mydintArray[1], mydint3);
Shift Left (SHL)	mydint := SHL(mydint4, mydintArray[7]);
Shift Right (SHR)	mydint := SHR(mydintArray[2], mydintArray[3]);

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Absolute Value Instruction (ST)

ABS(*Operand*)

Operand: DINT or LREAL variable, constant or expression that resolves to these data types.

Returns the unsigned magnitude of *Operand*.

Note: This instruction returns a value of the same data type as the operand.

See also ABS (Ladder).

Logarithmic Instructions (ST)

LN | LOG

LN(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the natural (base *e*) logarithm of *Operand*.

Notes

- An *Operand* of zero is invalid.
- The result of this instruction is LREAL.
- If the *Operand* is too large, the result will be zero. The valid range of an LREAL variable is ±2.225x10-308 through ±1.79x10308.

See also LN (Ladder).

LOG(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the base 10 logarithm of *Operand*.

Notes

- The result of this instruction is LREAL.
- If the *Operand* is too large, the result will be zero. The valid range of an LREAL variable is ±2.225x10-308 through ±1.79x10308.

See also LOG (Ladder).

Bitwise AND (ST)

Example

```
mydint := mydintArray[3] AND mydint2;
```

Note: If BOOL operands are used, the AND instruction is a boolean AND.

The bitwise AND instruction turns each bit in **mydint** ON(1) if the corresponding bit in **mydintArray[3]** is ON(1) and the corresponding bit in **mydint2** is ON(1). Otherwise, the bit is turned OFF(0).

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Truth Table

A	AND	**B**	=	**C**
1		1		1
1		0		0
0		1		0
0		0		0

Variations

The following variations apply to the bitwise AND instruction:

- A simple 32-bit AND is always performed.
- **A**, **B** and **C** may be DINT arrays, however in each case only a single element of the array can be specified. In this case, the single element specified in the DINT array **A** is ANDed with the single element specified in the DINT array **B**. The result is placed in the single element specified in the DINT array **C**.

Example

The following example shows the result of two DINTs being ANDed together.

0=OFF, 1=ON.

A=0110...0100
B=0100...0000
C=0100...0000

Tip: DINTs used in a bitwise AND should be displayed in *binary* format as other number formats may present misleading information.

Bitwise NOT (ST)

Example

```
myResult := NOT myDINTArray[1];
```

Note: If a BOOL operand is used, the NOT instruction is a boolean NOT.

If **myDINTArray** is DINT, the NOT instruction turns each bit in **myResult** ON(1) if the corresponding bit in **myDINTArray[1]** is OFF(0), and vice versa.

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Truth Table

A NOT =	**C**
1	0
0	1

Variations

The following variations apply to the bitwise NOT instruction:

- A simple 32-bit NOT is always performed.
- **A** and **C** may be DINT arrays, however in each case only a single element of the array can be specified. In this case, the single element specified in the DINT array **A** is NOTed. The result is placed in the single element specified in the DINT array **C**.

Example

The following example shows the result when a DINT is NOTed. 0=OFF, 1=ON

A = 0 1 1 0 ... 1 1 0 0
C = 1 0 0 1 ... 0 0 1 1

Tip: DINTs used in a bitwise NOT should be displayed in binary format as other number formats may present misleading information.

Bitwise OR (ST)

Example

```
mydint := mydint3 OR mydintArray[8];
```

Note: If BOOL operands are used, the OR instruction is a boolean OR.

The bitwise OR instruction turns each bit in **mydint** ON(1) if the corresponding bit in **mydint3** is ON(1) or the corresponding bit in **mydintArray[8]** is ON(1). Otherwise, the bit is turned OFF(0).

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Truth Table

A	OR	**B**	=	**C**
1		1		1
1		0		1
0		1		1
0		0		0

Variations

The following variations apply to the bitwise OR instruction:

- A simple 32-bit OR is always performed.
- **A**, **B** and **C** may be DINT arrays, however in each case only a single element of the array can be specified. In this case, the single element specified in the DINT array **A** is ORed with the single element specified in the DINT array **B**. The result is placed in the single element specified in the DINT array **C**.

Example

The following example shows the result of two DINTs being ORed together.

1=ON, 0=OFF

A = 0 1 1 0 ... 1 1 0 0
B = 1 1 0 0 ... 0 0 0 1
C = 1 1 1 0 ... 1 1 0 1

Tip: DINTs used in a bitwise OR should be displayed in *binary* format as other number formats may present misleading information.

Bitwise XOR (ST)

Example

```
mydint := mydintArray[1] XOR mydintArray[8];
```

Note: If BOOL operands are used, the XOR instruction is a boolean XOR.

The bitwise exclusive OR (XOR) instruction turns each bit in **mydint** ON(1) if the corresponding bit in **mydintArray[1]** is ON(1), or the corresponding bit in **mydintArray[8]** is ON(1), but not both. Otherwise, the bit in **mydint** is turned OFF(0).

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Truth Table

A	XOR	**B**	=	**C**
1		1		0
1		0		1
0		1		1
0		0		0

Variations

The following variations apply to the bitwise XOR instruction:

- A simple 32-bit XOR is always performed.
- **A**, **B** and **C** may be DINT arrays, however in each case only a single element of the DINT array can be specified. In this case, the single element specified in the DINT array **A** is XORed with the single element specified in the DINT array **B**. The result is placed in the single element specified in the DINT array **C**.

Example

The following example shows the result of two DINTs being exclusive ORed together. 0=OFF, 1=ON

A = 0 1 1 0 ... 1 1 0 0
B = 1 1 0 0 ... 1 0 0 1
C = 1 0 1 0 ... 0 1 0 1

Tip: DINTs used in a bitwise XOR should be displayed in *binary* format as other number formats may present misleading information.

Bit Rotate Left (ROL) (ST)

Example

```
mydint := ROL(mydint1, mydint2);
```

Rotate left. The ROL instruction shifts the bits in **mydint1** left **mydint2** positions. Bits shifted off the left end (most significant bit) are rotated back into the right end (least significant bit). The result is placed in **mydint**. Each scan causes **mydint1** to be rotated left **mydint2** times.

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Variations

The following variations apply to the bitwise ROL instruction:

- A simple 32-bit rotation to the left is always performed. **mydint2** should contain a value from 0 to 31.

  Note: #Overflow is turned ON(1) if **mydint2** is out of range, and the result will be incorrect.

- **mydint**, **mydint1** and **mydint2** may be DINT arrays, however, in each case only a single element of the array may be specified.

  Example

  ```
  myArray[0] := ROL(myArray[1], myArray[2]);
  ```

  The above example is valid. The bits of the value in **myArray[1]** will be rotated left by myArray[2] bits. The result is placed in **myArray[0]**. **myArray[2]** should contain a value from 0 to 31.

  Note: #Overflow is turned ON(1) if **myArray[2]** is out of range, and the result will be incorrect.

  Example

```
myArray := ROL(myArray, myArray[2]);
```

The above example is not valid. Each operand must be a simple DINT variable or a single element of a DINT array.

Bit Rotate Right (ROR) (ST)

Example

```
mydint := ROR(mydintArray[1], mydint3);
```

Rotate right. The ROR instruction shifts the bits of the value in **mydintArray[1]** right **mydint3** positions. Bits shifted off the right end (least significant bit) are rotated back into the left end (most significant bit). The result is placed in **mydint**. Each scan causes **mydintArray[1]** to be rotated right **mydint3** times.

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Variations

The following variations apply to the bitwise ROR instruction:

- A simple 32-bit rotation to the right is always performed. **mydint3** should contain a value from 0 to 31.

  Note: #Overflow is turned ON(1) if **mydint3** is out of range, and the result will be incorrect.

- **mydint**, **mydintArray[1]** and **mydint3** may be DINT arrays, however, in each case only a single element of the array may be specified.

  Example

  ```
  myArray[0] := ROR(myArray[1], myArray[2]);
  ```

  The above example is valid. The bits of the value in **myArray[1]** will be rotated right by myArray[2] bits. The result is placed in **myArray[0]**. **myArray[2]** should contain a value from 0 to 31.

  Note: #Overflow is turned ON(1) if **myArray[2]** is out of range, and the result will be incorrect.

Example

```
myArray := ROR(myArray, myArray[2]);
```

The above example is not valid. Each operand must be a simple DINT variable or a single element of a DINT array.

Bit Shift Left (SHL) (ST)

Example

```
mydint := SHL(mydint4, mydintArray[7]);
```

Shift left. The SHL instruction shifts the bits in **mydint4** left **mydintArray[7]** positions. Bits shifted off the left end (most significant bit) are lost, and the now-empty positions at the right end (least significant bit) are turned OFF(0). The result is placed in **mydint**. Each scan causes **mydint4** to be shifted **mydintArray[7]** times.

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Variations

The following variations apply to the bitwise SHL instruction:

- A simple 32-bit shift to the left is always performed. **mydintArray[7]** should contain a value from 0 to 31.

  Note: #Overflow is turned ON(1) if **mydintArray[7]** is out of range, and the result will be incorrect.

- Any operand in an ST bit expression may be a DINT array, however, in each case only a single element of the array may be specified.

  Example

  ```
  myArray[0] := SHL(myArray[1], myArray[2]);
  ```

  The above example is valid. The bits of the value in **myArray[1]** will be shifted left by myArray[2] bits. The result is placed in **myArray[0]**. **myArray[2]** should contain a value from 0 to 31.

  Note: #Overflow is turned ON(1) if **myArray[2]** is out of range, and the result will be incorrect.

Example

```
myArray := SHL(myArray, myArray[2]);
```

The above example is not valid. Each operand must be a simple DINT variable or a single element of a DINT array.

Bit Shift Right (SHR) (ST)

Example

```
mydint := SHR(mydintArray[2], mydintArray[3]);
```

Shift right. The SHR instruction shifts the bits in **mydintArray[2]** right **mydintArray[3]** positions. Bits shifted off the right end (most significant bit) are lost, and the now-empty positions at the left end (most significant bit) are turned OFF(0). The result is placed in **mydint**. Each scan causes **mydintArray[2]** to be shifted **mydintArray[3]** times.

Note: An operand in an ST bit expression, including the result variable (the variable to the left of the assignment operator ":=") must be a simple DINT variable or a single element of a DINT array.

Variations

The following variations apply to the bitwise SHR instruction:

- A simple 32-bit shift to the right is always performed. **mydintArray[3]** should contain a value from 0 to 31.

  Note: #Overflow is turned ON(1) if **mydintArray[3]** is out of range, and the result will be incorrect.

- Any operand in an ST bit expression may be a DINT array, however, in each case only a single element of the array may be specified.

  Example

  ```
  myArray[0] := SHR(myArray[1], myArray[2]);
  ```

  The above example is valid. The bits of the value in **myArray[1]** will be shifted right by myArray[2] bits. The result is placed in **myArray[0]**. **myArray[2]** should contain a value from 0 to 31.

  Note: #Overflow is turned ON(1) if **myArray[2]** is out of range, and the result will be incorrect.

Example

```
myArray := SHR(myArray, myArray[2]);
```

The above example is not valid. Each operand must be a simple DINT variable or a single element of a DINT array.

BOOL, DINT, LREAL Conversions (ST)

BOOL_TO_INT | INT_TO_BOOL | INT_TO_REAL | REAL_TO_INT

BOOL_TO_INT(*Operand*)

Operand: BOOL variable, constant or expression that resolves to a BOOL.

Converts BOOL *Operand* to a DINT value.

Example

```
myDINT:= BOOL_TO_INT(myBOOL);
```

INT_TO_BOOL(*Operand*)

Operand: DINT variable, constant or expression that resolves to a DINT.

Converts DINT *Operand* to a BOOL value.

Tip: If *Operand* is not zero, the result is always ON(1).

Example

```
myBOOL:= INT_TO_BOOL(myDINT);
```

INT_TO_REAL(*Operand*)

Operand: DINT variable, constant or expression that resolves to a DINT.

Converts the DINT *Operand* to an LREAL value.

Example

```
myLREAL := INT_TO_REAL(myDINT);
```

REAL_TO_INT(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Converts LREAL *Operand* to a DINT value.

Example

```
myDINT:= REAL_TO_INT(myLREAL);
```

Note: The decimal or fraction portion of the LREAL value is lost. For example, the LREAL value 2.99 becomes the DINT value 2 after the conversion.

Conditional Statements (ST)

IF ... THEN ... ELSE | CASE

IF ... THEN ... ELSE

Conditional Statements

Conditional Statements allow selected statements will be executed when certain conditions exist.

```
IF ... THEN ... ELSE
```

Selected ST statements can be evaluated depending on the value returned by a boolean expression:

```
IF <boolean expression> THEN
     <statements...>
END_IF;
```

A boolean expression always resolves to a boolean value (TRUE (ON) or FALSE (OFF)).

Alternative statements can be executed using the general forms:

```
IF <boolean expression> THEN
     <statements...>
ELSE
     <statements...>
END_IF;

- or -

IF <boolean expression> THEN
     <statements...>
ELSEIF <boolean expression> THEN
     <statements...>
END_IF;

- or -
```

```
IF <boolean expression> THEN
     <statements...>
ELSEIF <boolean expression> THEN
     <statements...>
ELSE
     <statements...>
END_IF;
```

Any number of additional ELSEIF sections can be added to the IF ... THEN construct.

Example

```
IF SwitchPosition = 0 THEN
     ConveyorSpeed := 0; ' Conveyor is stopped
ELSEIF ((SwitchPosition >= 1) AND (SwitchPosition <= 2)) THEN
     ConveyorSpeed := 2 * SwitchPosition; ' Run at slow speeds
ELSEIF ((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;
```

CASE

The CASE conditional statement will cause selected statements to execute depending on the value of an expression that returns a DINT result, for example the value of a single DINT variable or the DINT value resolved from a complex expression.

The set of statements which have a DINT selector value that matches the value of the DINT expression are executed. If no match is found, the statements after ELSE are executed.

The CASE construct has the general form:

```
CASE <DINT expression> OF
     <DINT selector value1> : <statements...>
     <DINT selector value2> : <statements...>
     ...
```

```
        ELSE
        <statements...>
    END_CASE;
```

Example

```
CASE SwitchPosition OF
    0    : ConveyorSpeed := 0; ' Conveyor is stopped
    1,2  : ConveyorSpeed := 2 * SwitchPosition; ' Run at slow speeds
    3..5 : ConveyorSpeed := 5 * SwitchPosition; ' Run at high speeds
ELSE
    ConveyorSpeed :=0; ' Stop the conveyor on any bad input
    ConveyorFault := #ALW_ON;
END_CASE;
```

Angle Conversion Instructions (ST)

DTOR | RTOD

DTOR(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Converts *Operand* (angle in degrees) to radians.

Note: The result of this instruction is LREAL.

See also DTOR (LD).

RTOD(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Converts *Operand* (angle in radians) to degrees.

Note: The result of this instruction is LREAL.

See also RTOD (LD).

Exponential Instructions (ST)

EXP | EXPT

EXP(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the natural exponent of *Operand* or $e^{Operand}$.

Notes

- The result of this instruction is LREAL.
- If the *Operand* is too large, the result will be zero. The valid range of an LREAL variable is ±2.225x10-308 through ±1.79x10308.

See also EXP (Ladder).

EXPT(*Operand1*,*Operand2*)

Operands: DINT or LREAL variables, constants or expressions.

This can also be expressed as *Operand1 ^ Operand2*.

- Or -

*Operand1 ** Operand2.*

Calculates $Operand1^{Operand2}$.

Note: Both operands must be either DINT or LREAL; they cannot be mixed.

Examples

```
dint1 := EXPT(2,2); 'valid, result is a DINT 4
lreal2 := 2.0 ^ 3.0; 'valid, result is an LREAL 8.0
dint2 := 3 ** 4; 'valid, result is a DINT 81
```

```
dint1 := EXPT(2.0,2.0);
'the above is not valid, attempting to assign an LREAL result to a DINT
variable

lrea12 := 2 ^ 2;
'the above is not valid, attempting to assign a DINT result to an LREAL
variable

dint2 := 2.0 ** 2.0;
'the above is not valid, attempting to assign an LREAL result to a DINT
variable

dint3 := EXPT(2.0,2);
lreal3 := 3 ^ 9.0;
'both of the above are not valid, operand1 and operand2 must be the same
data type
```

Notes

- This instruction returns a value of the same data type as the operands.
- If the result is too large, the result will be zero. The valid range of an LREAL variable is ±2.225x10-308 through ±1.79x10308.

See also EXPT (Ladder).

Hyperbolic Instructions (ST)

COSH | SINH | TANH

COSH(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the hyperbolic cosine of *Operand* (in radians). The angle is returned in radians.

Note: The result of this instruction is LREAL.

See also COSH (Ladder).

SINH(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the hyperbolic sine of *Operand* (in radians). The angle is returned in radians.

Note: The result of this instruction is LREAL.

See also SINH (Ladder).

TANH(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the hyperbolic tangent of *Operand* (in radians).

Operand: LREAL variable or constant.

Note: The result of this instruction is LREAL.

See also TANH (Ladder).

Inverse Trigonometric Instructions (ST)

ACOS | ASIN | ATAN | ATAN2

ACOS(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL and where (-1 ≤ *Operand* ≤ 1).

Calculates the inverse cosine of *Operand* (in radians). The angle is returned in radians, in the range 0 to pi.

Notes

- The result of this instruction is LREAL.
- An *Operand* outside of the range (-1 ≤ operand ≤ 1) causes the result to be zero.

See also ACOS (Ladder).

ASIN(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL and where (-1 ≤ *Operand* ≤ 1).

Calculates the inverse sine of *Operand* (in radians). The angle is returned in radians, in the range -pi/2 to +pi/2.

Notes

- The result of this instruction is LREAL.
- An *Operand* outside of the range (-1 ≤ operand ≤ 1) causes the result to be zero.

See also ASIN (Ladder).

ATAN(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the inverse tangent of *Operand* (in radians). The angle is returned in radians, in the range -pi/2 to +pi/2.

Note: The result of this instruction is LREAL.

See also ATAN (Ladder).

ATAN2(*Operand1*,*Operand2*)

Operands: LREAL variables, constants or expressions that resolve to LREAL values.

Calculates the inverse tangent of (*Operand2*/*Operand1*). The angle is returned in radians, in the range -pi to +pi.

Notes

- The result of this instruction is LREAL.
- If *Operand2* is zero, the result will be zero.
- If *Operand1* is zero, the result will **NOT** be correct.

See also ATAN2 (Ladder).

Iteration Statements (ST)

FOR...DO | WHILE...DO | REPEAT...UNTIL | EXIT | RETURN

With iteration statements, you can repeat one or more statements a number of times depending on the state of a particular variable or condition.

Note: Iteration statements should be constructed carefully to avoid endless loops. Iteration statements may also significantly increase the time to execute some software elements, such as function blocks.

FOR ... DO Statement

The FOR ... DO construct allows a set of statements to be repeated depending on the value of a DINT iteration variable. This construct takes the general form:

```
FOR <initialise iteration variable>
     TO <final value expression>
     [BY <increment expression>] DO
     <statements...>
END_FOR;
```

The FOR ... DO construct can be used to count iterations counting up or down and using any size increment until a final value is reached. If the BY keyword is omitted, the iteration variable will increase by 1.

The test to check whether the iteration has reached the final value is made before executing the statements inside the FOR ... DO construct. It is, therefore, possible for the final increment to cause the value of the iteration variable to exceed the final value, in which case the statements inside the FOR ... DO construct will not be executed.

Notes

- It is a best practise to not use the iteration variable outside the FOR loop.
- The statements within a FOR ... DO construct should not modify variables that will affect the expressions for the final and increment values.

Example

```
Sum := 0;
' Calculate the sum of the array elements
FOR Index := 0 TO 9 BY 1 DO
     Sum := Sum + DintArray[Index];
END_FOR;
```

WHILE ... DO Statement

The WHILE ... DO construct can be used to execute one or more statements while a particular boolean expression remains TRUE (ON). The boolean expression is tested prior to executing the statements, the statements within the WHILE ... DO are executed only while it is TRUE (ON). This construct takes the general form:

```
WHILE <boolean expression> DO
     <statements...>
END_WHILE;
```

Note: The WHILE <BOOL expression> should not use a value that must be read at the beginning of the scan. Such a value would not change during the execution part of the scan and would cause an endless loop that would fault the watchdog timer. If you want to refer to a value read at the beginning of the scan, use an IF statement to test the value and take action once based on the test; every scan would effectively repeat the test, which would cause the action to continue or stop.

Example

```
Index := 0;
Sum := 0;
'Another sum of an array
WHILE (Index <= 9)  DO
     Sum := Sum + DintArray[Index];
Index := Index +1;
END_WHILE;
```

REPEAT ... UNTIL Statement

The REPEAT ... UNTIL construct can be used to execute one or more statements while a particular boolean expression remains TRUE (ON). The boolean expression is tested after executing the statements, the statements within the REPEAT ... UNTIL are executed again after it becomes FALSE (OFF). This construct takes the general form:

```
REPEAT
     <statements...>
UNTIL <boolean expression>
END_REPEAT;
```

Note: Make sure the UNTIL <boolean expression> does not refer to, or depend on, a value that must be read at the beginning of the scan. Such a value would not change during the execution part of the scan and would cause an endless loop that would fault the watchdog timer. If you want to refer to a value read at the beginning of the scan, use an IF statement to test the value and take action once based on the test; every scan would effectively repeat the test, which would cause the action to continue or stop.

Example

```
Index := 0;
Sum := 0;
'Another way to sum an array
REPEAT
     Sum := Sum + DintArray[Index];
     Index := Index +1;
UNTIL Index < 10
END_REPEAT;
```

EXIT Statement

The EXIT statement can only be used within a WHILE, REPEAT, or FOR loop and allows it to leave the loop prematurely. When an EXIT statement is reached, execution continues immediately from the end of the iteration construct; no further part of the iteration construct is executed.

RETURN Statement

The RETURN statement can be used only within function and program block

bodies and is used to return prematurely from the code body. The RETURN statement causes execution to continue from the end of the function or function block body.

Trigonometric Instructions (ST)

COS | SIN | TAN

COS(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the cosine of *Operand*. The *Operand* is returned with the angle in radians.

Note: The result of this instruction is LREAL.

See also COS (Ladder).

SIN(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the sine of *Operand.* The *Operand* is returned with the angle in radians.

Note: The result of this instruction is LREAL.

See also SIN (Ladder).

TAN(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the tangent of *Operand*. The *Operand* is returned with the angle in radians.

Note: The result of this instruction is LREAL.

See also TAN (Ladder).

Square Root (ST)

SQRT(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Calculates the square root of *Operand*.

Notes

- The result of this instruction is LREAL.
- A negative *Operand* causes the result to be zero.

See also SQRT (Ladder).

String Instructions (ST)

These instructions are used to manipulate STRING variables.

Note: A STRING operand in an ST expression must be a simple STRING variable; STRING constants and STRING arrays are invalid.

Tips

- In the  Inspector, assign a value to a STRING variable with its Initial Value property.
- The maximum length of a string in a STRING variable is 255 characters; the minimum length is 1 (default is 32).

String Instruction	Example
assign	`myStringResult := myString;`
bool_to_string	`myStringResult := bool_to_string(myBool);`
concat	`myStringResult := concat(myString, myString2);`
delete	`myStringResult := delete(myString, myDintSize, myDintStartPosition);`
equal	`myBoolResult := myString = myString2;`
find	`myDintResult := find(myString, mySearchString);`
getat	`myStringResult := getat(myString, myDintPosition);`
greater than	`myBoolResult := myString > myString2;`
greater than or equal	`myBoolResult := myString >= myString2;`
insert	`myStringResult := insert(myString, myString2, myDintPosition);`
int_to_string	`myStringResult := int_to_string(myDint);`
left	`myStringResult := left(myString, myDintSize);`
length	`myDintResult := length(myString);`
less than	`myBoolResult := myString < myString2;`
less than or equal	`myBoolResult := myString <= myString2;`

makelower	*myStringResult* := makelower(*myString*);
makeupper	*myStringResult* := makeupper(*myString*);
mid	*myStringResult* := mid(*myString*, *myDintSize*, *myDintStartPosition*);
not equal	*myBoolResult* := *myString* <> *myString2*;
real_to_string	*myStringResult* := real_to_string(*myLreal*);
replace	*myStringResult* := replace(*myString*, *myString2*, *myDintSize*, *myDintStartPosition*);
reverse	*myStringResult* := reverse(*myString*);
reversefind	*myDintResult* := reversefind(*myString*, *myString2*);
right	*myStringResult* := right(*myString*, *myDintSize*);
string_to_bool	*myBoolResult* := string_to_bool(*myString*);
string_to_int	*myDintResult* := string_to_int(*myString*);
string_to_real	*myLrealResult* := string_to_real(*myString*);

STRING ASCII and Char Instructions (ST)

ASCII | Char

ascii (*Operand1, Operand2*)

Example

```
myDintResult := ascii(myString, myDintPosition);
```

Calculate the ASCII value of the character of the STRING variable *myString* at position *myDintPosition*, and store the result in the DINT variable *myDintResult*.

Variations

The following variations apply to the ASCII instruction:

- *myDintResult* and *myDintPosition* may be DINT arrays, however, only a single element of an array may be specified.
- *myString* may be a STRING array, however, only a single element of the array may be specified.

Note: If the value of *myString* is empty, or the value of *myDintPosition* is not a valid position in *myString*, *myDintResult* will contain ????.

Example

```
myDintResult := ascii(myString, myDintPosition);
```

myString	***myDintPosition***	***myDintResult***
0	0	0
abcde	4	ascii for e
abcde	5	-1???
1234567890	0	ascii for 1

STRING Char (ST)

Example

```
myStringResult := char(myDint);
```

Calculate the character corresponding to the ASCII value of the DINT variable *myDint*, and store the result in the STRING variable *myStringResult*.

Variations

The following variations apply to the Char instruction:

- *myStringResult* may be a STRING array, however, only a single element of the array may be specified.
- *myDint* may be a DINT array, however, only a single element of the array may be specified.

Note: If the value of *myDINT* isn't a valid ASCII value, *myStringResult* will contain ????.

Example

```
myStringResult := char(myDint);
```

myDint	***myStringResult***
0	null
100	
128	
255	
256	-1???

String Case Instructions (ST)

makelower | makeupper

makelower (*Operand*)

Operand: Simple STRING variable.

Convert all alphabetic, uppercase characters of *Operand* to lowercase. Makelower affects uppercase, alphabetic characters only.

Example

```
myStringResult := makelower(myString);
```

myString	Length of ***myStringResult***	***myStringResult***
ALL_UPPER_CASE	14	all_upper_case
ALL_UPPER_CASE	7	all_upp
all_lower_case	14	all_lower_case
1234567890	10	1234567890
1234567890	6	123456

Note: If *myStringResult* isn't large enough to contain the value of *myString*, the right-most characters of the result are truncated.

makeupper (*Operand*)

Operand: Simple STRING variable.

Convert all alphabetic, lowercase characters of *Operand* to uppercase. Makeupper affects lowercase, alphabetic characters only.

Example

```
myStringResult := makeupper(myString);
```

myString	Length of ***myStringResult***	***myStringResult***

all_lower_case	14	ALL_LOWER_CASE
all_lower_case	7	ALL_LOW
ALL_UPPER_CASE	14	ALL_UPPER_CASE
1234567890	10	1234567890
1234567890	6	123456

Note: If *myStringResult* isn't large enough to contain the value of *myString*, the right-most characters of the result are truncated.

String Concatenation Instruction (ST)

*c*oncat (*Operand1*, *Operand2*)

Operand1, *Operand2:* Simple STRING variables.

Concatenate *Operand1* and *Operand2*.

Example

```
myStringResult := concat(myString, myString2);
```

myString	***myString2***	***myStringResult*** Length of	***myStringResult***
ConcatString	ConcatString2	32	ConcatStringConcatString2
ConcatString	ConcatString2	21	ConcatStringConcatStr
12345	67890	10	1234567890
12345	67890	1	1
null	*null*	32	*null*

Notes

- *myStringResult* will contain the characters of *myString* followed by the characters of *myString2*.
- If *myStringResult* isn't large enough to contain the result of the **concat** instruction, the right-most characters of the result are truncated.

String Delete Instruction (ST)

delete(*Operand1, Operand2, Operand3*)

Operand1: Simple STRING variable.

Operand2, *Operand3*: DINT constants, variables or arrays.

From the left, delete the first occurrence of the substring stored in *Operand1*, for the length of *Operand2*, at position *Operand3*.

Example

```
myStringResult := delete(myString, myDintSize, myDintStartPosition);
```

myString	*myDintSize*	*myDintStartPosition*	Length of *myStringResult*	*myStringResult*
abc_def_ghi	5	3	32	abcghi
abc_def_ghi	5	3	4	abcg
abcde	6	0	32	*null*
12345	0	6	32	*null*
1234567890	6	5	32	*null*

Notes

- If *myStringResult* isn't large enough to contain the result of the **delete** instruction, the right-most characters of the result are truncated.
- The following will cause *myStringResult* to be set to null:
  - *myDintSize* is > than the length of *myString.*
  - *myDintStartPosition* is > than the length of *myString*.
  - The sum of *myDintSize* and *myDintStartPosition* is > the length of *myString.*

String IsEmpty Instruction (ST)

Operand: Simple STRING variable.

If the value of the STRING variable *myString* is:

- Empty, the value of the BOOL variable *myBoolResult* is set to True (1).
- Not empty, the value of *myBoolResult* is set to False (0).

Tip: *myString* is considered empty if it contains no characters or blanks; that is, it contains only null values (represented by the ASCII code zero).

Variations

The following variations apply to the Isempty instruction:

- *myBoolResult* may be a BOOL array, however, only a single element of the array may be specified.
- *myString* may be a STRING array, however, only a single element of the array may be specified.

Example

```
myBoolResult := isempty(myString);
```

myString	***myBoolResult***
null	1
blank	0
0	0
123	0
abc	0

String Extraction Instructions (ST)

GetAt | Left | Mid | Right

getat (*Operand1, Operand2*)

Operand1: Simple STRING variable.

Operand2: DINT constant, variable or array.

Retrieve the character stored in *Operand1* at position *Operand2*.

Example

```
myStringResult := getat(myString, myDintPosition);
```

myString	***myDintPosition***	***myStringResult***
abcdef	0	a
~!@#$%^	3	#
1234567890	7	8
1234567890	10	*null*
null	0	*null*

Note: If *myDintPosition* is > than the length of *myString*, *myStringResult* is set to null.

left (*Operand1, Operand2*)

Operand1: Simple STRING variable.

Operand2: DINT constant, variable or array.

Retrieve a substring of the string stored in *Operand1*, for the length of *Operand2*, starting with the left-most character.

Example

```
myStringResult := left(myString, myDintSize);
```

myString	***myDintsize***	***myStringResult***	Length of ***myStringResult***

abcdef	1	32	a
abcdef	6	32	abcdef
abcdef	7	32	*null*
12345	4	3	123
12345	0	32	*null*
12345	-1	32	*null*

Notes

- If *myStringResult* isn't large enough to contain the result of the **left** instruction, the right-most characters of the result are truncated.
- If *myDintSize* is > than the length of *myString,* or, if *myDintSize* is < 0, *myStringResult* is set to null.

mid (*Operand1*, *Operand2*, *Operand3*)

Operand1: Simple STRING variable.

Operand2, *Operand3*: DINT constants, variables or arrays.

Retrieve a substring of the string stored in *Operand1*, for the length of *Operand2*, at position *Operand3*.

Example

```
myStringResult := mid(myString, myDintSize, myDintStartPosition);
```

myString	myDintsize	myDintStartPosition	Length of myStringResult	myStringResult
abcdef	2	3	32	de
abcdef	5	0	32	abcde
abcdef	1	5	32	f
abcdef	5	2	32	*null*
abcdef	4	2	1	c
12345	5	0	32	12345
12345	6	0	32	*null*
12345	2	6	32	*null*
12345	-1	0	32	*null*

12345	5	-1	32	*null*

Notes

- If *myStringResult* isn't large enough to contain the result of the Mid instruction, the right-most characters of the result are truncated.
- The following will cause *myStringResult* to be set to null:
  - *myDintSize* is > than the length of *myString* or *myDintSize* is < zero.
  - *myDintStartPosition* is > than the length of *myString*.
  - *myDintStartPosition* is < zero.
  - The sum of *myDintSize* and *myDintStartPosition* is > than the length of *myString.*

right (*Operand1, Operand2*)

Operand1: Simple STRING variable.

Operand2: DINT constant, variable or array.

Retrieve a substring of the string stored in *Operand1*, for the length of *Operand2*, starting with the right-most character.

Example

```
myStringResult := right(myString, myDintSize);
```

myString	***myDintsize***	Length of ***myStringResult***	***myStringResult***
abcdef	5	32	bcdef
abcdef	7	32	*null*
abcdef	6	1	a
12345	4	3	234
12345	0	32	*null*
12345	-1	32	*null*

Notes

- If *myStringResult* isn't large enough to contain the result of the Right instruction, the right-most characters of the result are truncated.
- If *myDintSize* is > than the length of *myString,* or, if *myDintSize* is < 0, *myStringResult* is set to

null.

String Find and ReverseFind Instructions (ST)

Find | ReverseFind

find (*Operand1*, *Operand2*)

Operand1, *Operand2:* Simple STRING variables.

In *Operand1*, starting from the left, calculate the position of the first occurrence of *Operand2*.

Example

```
myDintResult := find(myString, mySearchString);
```

myString	***mySearchString***	***myDintResult***
123456	345	2
12341234	1	0
1245	3	-1
FindString	String	4
String_String	string	-1
null	*null*	-1

Notes

- If the value of *mySearchString* is not found in *myString*, -1 is assigned to *myDintResult*.
- Uppercase and lowercase alphabetic characters are not equal, since they are represented by different ASCII numbers. For example, 'A' is not equal to 'a'.

reversefind (*Operand1*, *Operand2*)

Operand1, *Operand2:* Simple STRING variables.

In *Operand1*, starting from the right, calculate the position of the first occurrence of *Operand2*.

Example

```
myDintResult := reversefind(myString, mySearchString);
```

myString	*mySearchString*	*myDintResult*
123456	345	2
12341234	1	4
1245	3	-1
FindString	String	4
String_String	String	7
String_String	string	-1
null	*null*	-1

Note: If the value of *mySearchString* is not found in *myString*, -1 is assigned to *myDintResult*.

String Insert Instruction (ST)

insert(*Operand1, Operand2, Operand3*)

Operand1, *Operand2:* Simple STRING variables.

Operand3: DINT constant, variable or array.

In *Operand1*, insert the string in *Operand2*, at position *Operand3*.

Example

```
myStringResult := insert(myString, myString2, myDintPosition);
```

myString	***myString2***	***myDintPosition***	Length of ***myStringResult***	***myStringResult***
defghi	abc	0	32	abcdefghi
abc	def	3	32	abcdef
defghi	abc	3	6	defabc
123	456	3	4	1234
null	123	0	32	123
null	123	1	32	*null*

Note: If *myStringResult* isn't large enough to contain the result of the **insert** instruction, the right-most characters of the result are truncated.

String Length Instruction (ST)

length(*Operand*)

Operand: Simple STRING variable.

Calculate the length of *Operand*.

Example

```
myDintResult := length(myString);
```

myString	***myDintResult***
abcdef	6
$	1
-2.56	5
null	0

Note: The result of the **Length** instruction is always a DINT value.

Tip: The maximum length of a string in a STRING variable is 255 characters; the minimum length is 1 (default is 32).

String Replace Instruction (ST)

replace(*Operand1, Operand2, Operand3, Operand4*)

Operand1, *Operand2:* Simple STRING variables.

Operand3, *Operand4*: DINT constants, variables or arrays.

Replace a substring of *Operand1* with *Operand2*, for the length of *Operand3*, at position *Operand4*.

Example

```
myStringResult := replace(myString, myString2, myDintSize, myDintStartPosition);
```

myString	*myString2*	*myDintSize*	*myDintStartPosition*	Length of *myStringResult*	*myStringResult*
old_string	new	3	0	32	new_string
old_string	new_	0	0	32	new_old_string
my_string_old	new	3	10	32	my_string_new
string	add	7	0	32	*null*
string	add	0	7	32	*null*
my_string	add	-1	0	32	*null*
my_string	add	3	-1	32	*null*
my_string	add	-1	-1	32	*null*
123	456	3	2	32	*null*
123	456	0	3	32	123456
123	456	0	3	4	1234
null	123	3	0	32	*null*
null	123	0	0	32	123
123	*null*	3	0	32	*null*
123	*null*	2	0	32	3

Notes

- If *myStringResult* isn't large enough to contain the result of the **replace** instruction, the right-most characters of the result are truncated.
- The following will cause *myStringResult* to be set to null:
  - *myDintSize* is > than the length of *myString.*
  - *myDintStartPosition* is > than the length of *myString*.
  - The sum of *myDintSize* and *myDintStartPosition* is > than the length of *myString.*
  - *myDintSize* is < 0.
  - *myDintStartPosition* is < 0.

String Reverse (ST)

reverse(*Operand*)

Operand: Simple STRING variable.

Reverse the characters of *Operand*.

Example

```
myStringResult := reverse(myString);
```

myString	***myStringResult*** Length of	***myStringResult***
gnirtS_esreveR	32	Reverse_String
gnirtS_esreveR	10	Reverse_St
012345	32	543210
null	32	*null*

Note: If *myStringResult* isn't large enough to contain the result of the **reverse** instruction, the right-most characters of the result are truncated.

String Type Conversion Instructions (ST)

bool_to_string | int_to_string | real_to_string | string_to_bool | string_to_int | string_to_real

bool_to_string (*Operand*)

Operand: BOOL variable or array.

Convert the value in *Operand* to a STRING.

The ST ADD instruction has the same functionality, operands, and CPU support as in  ladder and  ST.

Example

```
myStringResult := bool_to_string(myBool);
```

myBool	***myStringResult***
0	0
1	1

int_to_string (*Operand*)

Operand: DINT constant, variable or array.

Convert the value in *Operand* to a STRING.

Example

```
myStringResult := int_to_string(myDint);
```

myDint	length of ***myStringResult***	***myStringResult***
0	32	0
2147483647	32	2147483647
-2147483648	7	-214748

Note: If *myStringResult* isn't large enough to contain the result of the **int_to_string** instruction, the right-most characters of the result are truncated.

real_to_string (*Operand*)

Operand: LREAL constant, variable or array.

Convert the value in *Operand* to a STRING.

Example

```
myStringResult := real_to_string(myLreal);
```

myLreal	length of ***myStringResult***	***myStringResult***
0.0	32	0.0
2.2250738507201E-308	32	2.2250738507201E-308
-2.2250738507201E-308	10	-2.2250738
1.7976931348623157E+308	32	1.7976931348623157E+308
-1.7976931348623157E+308	5	-1.79

Note: If *myStringResult* isn't large enough to contain the result of the **real_to_string** instruction, the right-most characters of the result are truncated.

string_to_bool (*Operand*)

Operand: Simple STRING variable.

Convert the value in *Operand* to a BOOL.

Example

```
myBoolResult := string_to_bool(myString);
```

myString	***myBoolResult***
0	Off (0)
1	On (1)
-1	On (1)
2	On (1)
a	Off (0)
null	Off (0)

Note: If *myString* contains a:

- Numeric value other than zero, *myBoolResult* is set to True (1).
- Non-numeric value, *myBoolResult* is set to False (0).

string_to_int (*Operand*)

Operand: Simple STRING variable.

Convert the value in *Operand* to a DINT.

Example

```
myDintResult := string_to_int(myString);
```

myString	***myDintResult***
0	0
2147483647	2147483647
-2147483648	-2147483648
a	0
null	0

Note: If *myString* doesn't contain a valid DINT value, *myDintResult* is set to zero.

string_to_real (*Operand*)

Operand: Simple STRING variable.

Convert the value in *Operand* to an LREAL.

Example

```
myLrealResult := string_to_real(myString);
```

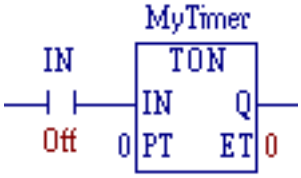
myString	***myLrealResult***
0	0.0
-2.2250738507201	-2.2250738507201
a	0.0
null	0.0

Note: If *myString* doesn't contain a valid LREAL value, *myLrealResult* is set to 0.0.

Timer (ST)

TIMER structure variable | Syntax | Detailed Operation | Examples

The ST language supports only one type of Timer, which is the equivalent of a Ladder Timer On Delay (TON) preceded by a Normally Open contact IN:



To use the ST Timer, create a TIMER structure variable and use its name in logic with the proper syntax:

```
MyTimer(PT := Operand1, IN := Operand2);
```

When the Timer's **IN** parameter is ON:

- The Timer initializes and begins to count.
- Program execution passes to the next statement.

When a TIMER variable is encountered in logic, the Timer checks whether the elapsed time (**ET**) has reached the preset time (**PT**). When **ET** reaches **PT**, the Timer turns ON its **Q** bit to indicate the time has elapsed.

References

- TIMER structure variable.
- Detailed ST Timer operation.

Note: The ST Timer can be used to build logic equivalent to a Ladder Timer Off Delay (TOF) and to a Ladder Timer Pulse (TP).

ST Timer Syntax

```
MyTimer(PT := Operand1, IN := Operand2);
```

where

- **`MyTimer`** is the name of a TIMER variable.
- **`PT`** (required) is the MyTimer.PT element.
- *`Operand1`* is a DINT constant or variable; it cannot be an expression. *`Operand1`* is assigned to the MyTimer.PT element.
- **IN** (required) is the equivalent of a Normally Open contact in Ladder logic.
- *`Operand2`* is a BOOL variable or constant; it cannot be an expression. *`Operand2`* is assigned to the Mytimer.TI element.

Detailed Operation

When the Timer is invoked, and MyTimer.TI is OFF, and the IN parameter is ON:

- The timing bit MyTimer.TI is turned ON.
- The elapsed time MyTimer.ET begins counting upward (in milliseconds).
- The output enable bit MyTimer.Q is turned OFF.

When the Timer is running (MyTimer.TI is turned ON) and referenced in ST logic:

- The elapsed time MyTimer.ET is incremented.

When the Timer is running (MyTimer.TI is turned ON) and the elapsed time (**ET**) is equal to or greater than the preset time (MyTimer.ET >= MyTimer.PT) after incrementing:

- The elapsed time MyTimer.ET is set to the preset value and is no longer incremented.
- The timing bit MyTimer.TI is turned OFF.
- The output enable bit MyTimer.Q is turned ON.

When the Timer is invoked and the IN parameter is OFF:

- The elapsed time MyTimer.ET is set to zero.
- The timing bit MyTimer.TI is set to OFF.
- The output enable bit MyTimer.Q is set to OFF.

Note: MyTimer.TI must be ON to initialize the Timer or keep it running. MyTimer.TI must be OFF to prevent the Timer from initializing or stop it from running.

Example 1

```
MyTimer(PT := 5, IN := #Alw_On);
```

When the above statement is encountered in ST logic, and MyTimer.TI is OFF, and the IN parameter is ON (#Alw_On is a BOOL system variable with the value ON):

- The timing bit MyTimer.TI is turned ON.
- The elapsed time MyTimer.ET begins counting upward (in milliseconds).
- The output enable bit MyTimer.Q is turned OFF.

When the Timer is running (MyTimer.TI is turned ON) and referenced in ST:

- The elapsed time MyTimer.ET is incremented.

When the Timer is running and the elapsed time is equal to or greater than the preset time (MyTimer.ET >= 5) after incrementing:

- The elapsed time MyTimer.ET is set to the preset value (5) and is no longer incremented.

- The timing bit MyTimer.TI is turned OFF.
- The output enable bit MyTimer.Q is turned ON.

Note: MyTimer.TI must be ON to initialize the Timer or keep it running. MyTimer.TI must be OFF to prevent the Timer from initializing or stop it from running.

Example 2

```
MyTimer(PT := 5, IN := #Alw_Off);
```

When the above statement is encountered in ST logic, and the IN parameter is OFF (#Alw_Off is a BOOL system variable with the value OFF):

- The elapsed time MyTimer.ET is set to zero.
- The timing bit MyTimer.TI is set to OFF.
- The output enable bit MyTimer.Q is set to OFF.

Note: MyTimer.TI must be ON to initialize the Timer or keep it running. MyTimer.TI must be OFF to prevent the Timer from initializing or stop it from running.

ST Language: an Overview

ST Language | Assignments | Expressions | Operators | Calling function blocks | Data types | Conditional statements | Iteration statements

Note: Instances of UDFBs can be used in ST logic.

About the Structured Text Language

Structured Text (ST) is one of five programming languages specified by the IEC 61131-3 standard.

Assignments

Assignment statements can change the value stored in a variable or the value returned by a function. An assignment statement has the following general format:

```
Y := X;
```

where **X** represents a variable, expression, or constant that produces a new value for the variable **Y** when the assignment statement is evaluated.

An expression can be simple (a literal constant) or complex (many nested expressions).

Parameter data types

Valid parameter data types for the COPY instruction are as follows:

If **X** is:	then **Y** is:
BOOL	BOOL
DINT	DINT
DINT constant	DINT
LREAL	LREAL
LREAL constant	LREAL
STRING	STRING
Custom structure	Custom structure

Notes

- There is no space between the colon (:) and the equal sign (=).
- The result (variable Y) must be of the same data type as the variable or constant X.
- X and Y can be single elements of BOOL, DINT, or LREAL arrays.
- X and Y can also be custom structures or single elements of custom structures.

See also the the COPY and MOV instructions in ladder and the MOV instruction in FBD.

Expressions

Expressions calculate values from other variables and constants. Structured Text expressions always produce a value of a particular data type, which can be BOOL, DINT, LREAL, STRING, or a user-defined data type. An expression can involve one or more operators, variables, and functions.

Composite expressions can be created by nesting simpler expressions.

Example

```
ConveyorSpeed := 0; ' this is a simple expression
```

Notes

- Expressions must always produce values that match the data type of the variable being assigned.
- Expressions are evaluated in order of the precedence of the operators and other subexpressions. Parenthesized expressions have the highest precedence, followed by functions. Operators of the highest precedence are evaluated next, followed by lower precedence operators, down to the lowest. Where operators have the same precedence, they are evaluated left to right.

Operators

ST defines a range of operators for arithmetic and boolean operations. For details on any of these operators, please see ST Language operators. These are (each category is in order of highest to lowest precedence):

- parentheses

- Function(...)

Math Operators

- ** (exponent)
- * (multiply)
- / (division)
- MOD (remainder)
- + (add)
- - (subtract)

Boolean Operators

- - (negate)
- NOT
- < (less than)
- > (greater than)
- <= (less than or equal)
- >= (greater than or equal)
- = (equal)
- <> (not equal)
- AND (&)
- XOR (exclusive OR)
- OR

Calling Function Blocks

ST logic can call other ST blocks, ladder logic, FBDs, and IL blocks

as subroutines or functions. The function block statement has the following general forms:

```
BlockName();
```

- Or -

```
BlockName([Parameter1 := Calling_Parameter1, Parameter2 :=
Calling_Parameter2, ... , Parameter n := Calling_Parameter n]);
```

- Or -

```
BlockName([Parameter1 := Calling_Parameter1, Parameter2 :=
Calling_Parameter2, ... , Parameter n := Calling_Parameter n OUT
Out_Calling_Parameter1 := Out_Parameter1, Out_Calling_Parameter2 :=
Out_Parameter2, ... , Out_Calling_Parameter n := Out_Parameter n]);
```

where:

- Statements inside [ ] are optional.
- *n* is any integer.
- Parameter*n* and Out_Parameter*n* are variables used inside the called block.
- Calling_Parameter*n* and Out_Calling_Parameter*n* are variables defined in your project.
- Parameter*n* must be of the same data type as Calling_Parameter*n*.
- Out_Calling_Parameter*n* must be of the same data type as Out_Parameter*n*.
- The values can be produced by expressions of any complexity.

For an ST block, the following steps are performed:

1. When an ST block is called, prior to its execution, the assignment operations inside the parenthesis of the Call statement are executed until the **OUT** keyword is encountered.
2. The block is executed.
3. After the block has executed, the assignment operations following the **OUT** keyword are performed.

Typically, Parameter*n* and Out_Parameter*n* are global variables, and are used inside the called block, while Calling_Parameter*n* and Out_Calling_Parameter*n* are variables used in the ST calling statement.

Example 1

The following ST statement can be used when calling a ladder or ST UDFB block:

```
myBlock(myDINT_Param := myDINT_Calling_Param, myBOOL_Param :=
myBOOL_Calling_Param);
```

Note: When calling a ladder UDFB or an ST UDFB, you can't use the three ST statements shown below. You must refer to the input and output variables in the UDFB by using the single ST statement shown above.

The above ST statement is equivalent to the following three ST statements:

```
myDINT_Param := myDINT_Calling_Param;
myBOOL_Param := myBOOL_Calling_Param;
myBlock();
```

Example 2

The following ST statement can be used when calling a ladder UDFB or an ST UDFB:

```
myBlock(myLREAL_Param1 := myLREAL_Calling_Param1, myBOOL_Param2 :=
myBOOL_Calling_Param2 OUT myDINTOut_Calling_Param3 := myDINT_Param3,
myLREALOut_Calling_Param4 := myLREAL_Param4);
```

Note: When calling an ST or ladder UDFB block, you cannot use the five ST statements shown below. You must refer to the input and output variables in the UDFB by using the single ST statement shown above.

The above ST statement is equivalent to the following five ST statements:

```
myLREAL_Param1 := myLREAL_Calling_Param1;
myBOOL_Param2 := myBOOL_Calling_Param2;
myBlock();
myDINTOut_Calling_Param3 := myDINT_Param3;
myLREALOut_Calling_Param4 := myLREAL_Param4;
```

Notes

- If you enter a name for a variable that doesn't yet exist in your project you must create the variable before downloading.
- Universal variables are prefixed with a '$'. In ST, they are automatically created only under certain conditions.

Data types and Variable types

Supported data types are BOOL, DINT, LREAL, STRING, TIMER, and STRUCTURE variable types.

Basic Instructions (ST)

These ST instructions perform basic math operations on data. Most of the basic ST instructions support operations on BOOL, DINT and LREAL variables. However, all parameters in a single ST basic math instruction must be the same data type, and the return value must be the same data type as the parameters.

Instruction	Symbol or Mnemonic	Example
Addition	+	`myDintResult := myDint + myDint2;`
Negation or subtraction	-	`myLrealResult := myLreal - myLreal2;`
Multiplication	*	`myDintResult := myDint * myDint2;`
Division	/	`myLrealResult := myLreal / myLreal2;`
Modulus	mod	`myDintResult := mod(myDint, myDint2);`

Logical Operators (ST)

not

Operand can be BOOL variable, expression or constant. The state of *Operand* is toggled.

Example

```
myBOOL := NOT(myBOOL1);
```
See also NOT instructions in  ladder and  FBD.

and

Operands can be BOOL variables, expressions or constants. A logical AND is performed.

Example

```
myBOOL := myBOOL1 AND myBOOL2;
```
See also AND instructions in  ladder and  FBD.

or

Operands can be BOOL variables, expressions or constants. A logical OR is performed.

Example

```
myBOOL := myBOOL1 OR myBOOL2;
```
See also OR instructions in  ladder and  FBD.

xor

Operands can be BOOL variables, expressions or constants. A logical XOR is performed.

Example

```
myBOOL := myBOOL1 XOR myBOOL2;
```
See also XOR instructions in  ladder and  FBD.

Note: For logical instructions, all operands must be BOOL variables, constants, or expressions which resolve to a BOOL value.

Bitwise Operators (ST)

not | and | or | xor

Note: All operands in ST Bitwise instructions must be DINT variables or be expressions which resolve to DINT values.

Conversion Operators (ST)

Math Operators (ST)

+ (Addition)

Operands can be DINT or LREAL variables, expressions or constants. Adds both operands together.

Note: Both operands must be either DINT or LREAL; they cannot be mixed.

Examples

```
myDINT := myDINT + 1; 'valid
myLREAL := myLREAL + 1.0; 'valid

myDINT := 2.5 + 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT variable

myLREAL := 2 + 1;
'the above is invalid, attempting to assign a DINT result to an LREAL variable

myDINT := myDINT + 1.0;
myLREAL := myLREAL + 1;
'both of the above are invalid, operand1 and operand2 must be the same data type
```

See also ADD instructions in ladder and FBD.

- (Negation or Subtraction)

Operands can be BOOL, DINT or LREAL variables, expressions or constants. If only one operand is supplied, the operator is negation. If two operands are supplied, the operator is subtraction.

Negation

If Operand is BOOL, the state of Operand is toggled.

Examples

```
myBOOL := -#ALW_OFF;
```

```
'the above is valid because a BOOL is assigned to a BOOL. In this case,
the state of myBOOL is turned ON(1).

myDINT := -myDINT2; 'valid statement because a DINT is assigned to a
DINT.
myLREAL := -1.0; 'valid statement because an LREAL is assigned to an
LREAL.

myDINT := -1.0;
'The above is invalid because -1.0 is an LREAL constant being assigned to
a DINT variable.

myLREAL := -2;
'The above statement is invalid because -2 is a DINT constant being
assigned to an LREAL variable.
```

Subtraction

Subtracts *operand2* from *operand1*.

Note: Both operands must be either DINT or LREAL variables, expressions or constants; they cannot be mixed.

Examples

```
myDINT := myDINT - 1; 'valid
myLREAL := myLREAL - 1.0; 'valid

myDINT := 2.5 - 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 2 - 1;
'the above is invalid, attempting to assign a DINT result to an LREAL
variable

myDINT := myDINT - 1.0;
myLREAL := myLREAL - 1;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

See also SUB instructions in  ladder and  FBD.

* (Multiplication)

Operands can be DINT or LREAL variables, expressions, or constants.

Multiplies *Operand1* by *Operand2*.

Note: Both operands must be either DINT or LREAL; they cannot be mixed.

Examples

```
myDINT := myDINT * -1; 'valid
myLREAL := myLREAL * 2.0; 'valid

myDINT := 2.5 * 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 2 * 2;
'the above is invalid, attempting to assign a DINT result to an LREAL
variable

myDINT := myDINT * -1.0;
myLREAL := myLREAL * 1;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

See also MUL instructions in  ladder and  FBD.

/ (Division)

Operands can be DINT or LREAL variables, expressions or constants. Divides *Operand1* by *Operand2*.

Notes

- Both operands must be either DINT or LREAL; they cannot be mixed.
- Dividing 2 DINTs results in the fraction being lost.
- On division by zero, #Overflow is set to ON (1).

Examples

```
myDINT := 5 / 2; 'valid; the result is a DINT 2
myLREAL := 5.0 / 2.0; 'valid; the result is LREAL 2.0

myDINT := 2.5 / 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 4 / 2;
'the above is invalid, attempting to assign a DINT result to an LREAL
```

```
variable

myDINT := myDINT / 2.0;
myLREAL := myLREAL / 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

See also DIV instructions in  ladder and  FBD.

mod (Modulus)

Operands can be DINT or LREAL variables, expressions, or constants. Returns the remainder when *Operand1* is divided by *Operand2*. If both operands are LREAL, the following occurs:

1. Both operands are converted to DINT values.
2. The **mod** instruction is performed on the DINT operands.
3. The DINT result is converted to an LREAL.

Notes

Both operands must be either DINT or LREAL; they cannot be mixed.

Examples

```
myDINT := 5 MOD 3; 'valid; the result is a DINT 2
myLREAL := 5.0 MOD 3.0; 'valid; the result is LREAL 2.0
myLREAL := 124.45 MOD 3.432; 'valid; the result is LREAL 1.0

myDINT := 2.5 MOD 1.0;
'the above is invalid, attempting to assign an LREAL result to a DINT
variable

myLREAL := 4 MOD 2;
'the above is invalid, attempting to assign a DINT result to an LREAL
variable

myDINT := myDINT MOD 2.0;
myLREAL := myLREAL MOD 2;
'both of the above are invalid, operand1 and operand2 must be the same
data type
```

See also MOD instructions in  ladder and  FBD.

Relational Operators (ST)

For all relational operators:

- Operands must both be DINT, LREAL or STRING; you cannot mix data types in an expression.
- Operands can be variables, expressions or constants.
- STRING constants are invalid.
- You cannot compare two BOOL values.
- The result is always BOOL.

> (Greater than)

If *Operand1* is greater than *Operand2*, the logical expression resolves to True. If *Operand1* is less than or equal to *Operand2*, the logical expression resolves to False.

Examples

```
myBOOL := 5 > 3; 'valid; the result is set to True (1)
myBOOL := 2.9 > 3.0; 'valid; the result is set to False (0)
myBOOL := myString > myString2; 'valid if myString and myString2 are both STRING variables

myDINT := 2.5 > 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT variable

myBOOL := myDINT > 2.0;
myBOOL := myLREAL > 2;
myBOOL := mySTRING > '2';
myBOOL := 'a' > 'b';
'all of the above are invalid; operand1 and operand2 must be the same data type, and STRING constants are invalid
```

See also GT instructions in ladder and FBD.

>= (Greater than or Equal)

If *Operand1* is greater than or equal to *Operand2*, the BOOL expression resolves to True. If *Operand1* is less than *Operand2*, the BOOL expression resolves to False.

Examples

```
myBOOL := 5 >= 3; 'valid; the result is set to True (1)
myBOOL := 2.9 >= 3.0; 'valid; the result is set to False (0)
myBOOL := myString >= myString2; 'valid if myString and myString2 are
both STRING variables

myDINT := 2.5 >= 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT >= 2.0;
myBOOL := myLREAL >= 2;
myBOOL := mySTRING >= '2';
myBOOL := 'a' >= 'b';
'all of the above are invalid; operand1 and operand2 must be the same
data type, and STRING constants are invalid
```

See also GE instructions in  ladder and  FBD.

= (Equal)

If *Operand1* is exactly equal to *Operand2*, the BOOL expression resolves to True. If *Operand1* is not equal to *Operand2*, the BOOL expression resolves to False.

Warning: Comparing LREAL values may produce unexpected results. For example, a calculation may result in 1.99999999999, which is not equal to 2.00000000000.

Examples

```
myBOOL := 5 = 3; 'valid; the result is set to False (0)
myBOOL := 5.0 = 5.0; 'valid; the result is set to True (1)
myBOOL := myString = myString2; 'valid if myString and myString2 are both
STRING variables

myDINT := 2.5 = 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT = 2.0;
myBOOL := myLREAL = 2;
myBOOL := mySTRING = '2';
myBOOL := 'a' = 'a';
'all of the above are invalid; operand1 and operand2 must be the same
```

data type, and STRING constants are invalid

See also EQ instructions in ladder and FBD.

<> or != (Not Equal)

If *Operand1* is not equal to *Operand2*, the BOOL expression resolves to True. If *Operand1* is exactly equal to *Operand2*, the BOOL expression resolves to False.

Warning: Comparing LREAL values may produce unexpected results. For example, a calculation may result in 1.99999999999, which is not equal to 2.00000000000.

Examples

```
myBOOL := 5 <> 3; 'valid; the result is set to True (1)
myBOOL := 5.0 <> 5.0; 'valid; the result is set to False (0)
myBOOL := myString <> myString2; 'valid if myString and myString2 are both STRING variables

myDINT := 2.5 <> 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT variable

myBOOL := myDINT <> 2.0;
myBOOL := myLREAL <> 2;
myBOOL := mySTRING <> '2';
myBOOL := 'a' <> 'b';
'all of the above are invalid; operand1 and operand2 must be the same data type, and STRING constants are invalid
```

See also NE instructions in ladder and FBD.

< (Less than)

If *Operand1* is less than *Operand2*, the logical expression resolves to True. If *Operand1* is greater than or equal to *Operand2*, the logical expression resolves to False.

Examples

```
myBOOL := 5 < 3; 'valid; the result is set to False (0)
myBOOL := 2.9 < 3.0; 'valid; the result is set to True (1)
myBOOL := myString < myString2; 'valid if myString and myString2 are both STRING variables

myDINT := 2.5 < 1.0;
```

```
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT < 2.0;
myBOOL := myLREAL < 2;
myBOOL := mySTRING < '2';
myBOOL := 'a' < 'b';
'all of the above are invalid; operand1 and operand2 must be the same
data type, and STRING constants are invalid
```

Note: In an ST relational expression, both operands must be the same data type.

See also LT instructions in  ladder and  FBD.

<= (Less than or Equal)

If *Operand1* is less than or equal to *Operand2*, the BOOL expression resolves to True. If *Operand1* is greater than *Operand2*, the BOOL expression resolves to False.

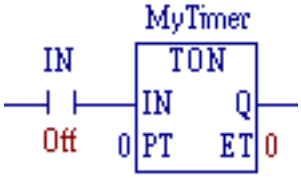
Examples

```
myBOOL := 5 <= 3; 'valid; the result is set to False (0)
myBOOL := 2.9 <= 3.0; 'valid; the result is set to True (1)
myBOOL := myString <= myString2; 'valid if myString and myString2 are
both STRING variables

myDINT := 2.5 <= 1.0;
'the above is invalid, attempting to assign a BOOL result to a DINT
variable

myBOOL := myDINT <= 2.0;
myBOOL := myLREAL <= 2;
myBOOL := mySTRING <= '2';
myBOOL := 'a' <= 'b';
'all of the above are invalid; operand1 and operand2 must be the same
data type, and STRING constants are invalid
```

See also LE instructions in  ladder and  FBD.

Timer (ST)

The ST language supports only one type of Timer, which is the equivalent of the ladder Timer On Delay (TON) preceded by a Normally Open contact IN:



To use the ST Timer, create a TIMER structure variable and use its name in logic with the proper syntax:

```
MyTimer(PT := Operand1, IN := Operand2);
```

When the Timer's **IN** parameter is ON:

- The Timer initializes and begins to count.
- Program execution passes to the next statement.

When a TIMER variable is encountered in logic, the Timer checks whether the elapsed time (**ET**) has reached the preset time (**PT**). When **ET** reaches **PT**, the Timer turns ON its **Q** bit to indicate the time has elapsed.

References

- TIMER structure variable.
- Detailed ST Timer operation.

Note: The ST Timer can be used to build logic equivalent to a ladder Timer Off Delay (TOF) and to a ladder Timer Pulse (TP).

ST Timer Syntax

```
MyTimer(PT := Operand1, IN := Operand2);
```

where

- **MyTimer** is the name of a TIMER variable.
- **PT** (required) is the MyTimer.PT element.
- *Operand1* is a DINT constant or variable; it cannot be an expression. *Operand1* is assigned to the MyTimer.PT element.
- **IN** (required) is the equivalent of a Normally Open contact in ladder logic.
- *Operand2* is a BOOL variable or constant; it cannot be an expression. *Operand2* is assigned to the Mytimer.TI element.

Detailed Operation

When the Timer is invoked, and MyTimer.TI is OFF, and the IN parameter is ON:

- The timing bit MyTimer.TI is turned ON.
- The elapsed time MyTimer.ET begins counting upward (in milliseconds).
- The output enable bit MyTimer.Q is turned OFF.

When the Timer is running (MyTimer.TI is turned ON) and referenced in ST logic:

- The elapsed time MyTimer.ET is incremented.

When the Timer is running (MyTimer.TI is turned ON) and the elapsed time (**ET**) is equal to or greater than the preset time (MyTimer.ET >= MyTimer.PT) after incrementing:

- The elapsed time MyTimer.ET is set to the preset value and is no longer incremented.
- The timing bit MyTimer.TI is turned OFF.
- The output enable bit MyTimer.Q is turned ON.

When the Timer is invoked and the IN parameter is OFF:

- The elapsed time MyTimer.ET is set to zero.
- The timing bit MyTimer.TI is set to OFF.
- The output enable bit MyTimer.Q is set to OFF.

Note: MyTimer.TI must be ON to initialize the Timer or keep it running. MyTimer.TI must be OFF to prevent the Timer from initializing or stop it from running.

Example 1

```
MyTimer(PT := 5, IN := #Alw_On);
```

When the above statement is encountered in ST logic, and MyTimer.TI is OFF, and the IN parameter is ON (#Alw_On is a BOOL system variable with the value ON):

- The timing bit MyTimer.TI is turned ON.
- The elapsed time MyTimer.ET begins counting upward (in milliseconds).
- The output enable bit MyTimer.Q is turned OFF.

When the Timer is running (MyTimer.TI is turned ON) and referenced in ST:

- The elapsed time MyTimer.ET is incremented.

When the Timer is running and the elapsed time is equal to or greater than the preset time (MyTimer.ET >= 5) after incrementing:

- The elapsed time MyTimer.ET is set to the preset value (5) and is no longer incremented.

- The timing bit MyTimer.TI is turned OFF.
- The output enable bit MyTimer.Q is turned ON.

Note: MyTimer.TI must be ON to initialize the Timer or keep it running. MyTimer.TI must be OFF to prevent the Timer from initializing or stop it from running.

Example 2

```
MyTimer(PT := 5, IN := #Alw_Off);
```

When the above statement is encountered in ST logic, and the IN parameter is OFF (#Alw_Off is a BOOL system variable with the value OFF):

- The elapsed time MyTimer.ET is set to zero.
- The timing bit MyTimer.TI is set to OFF.
- The output enable bit MyTimer.Q is set to OFF.

Note: MyTimer.TI must be ON to initialize the Timer or keep it running. MyTimer.TI must be OFF to prevent the Timer from initializing or stop it from running.

See also TON instructions in ladder and FBD.

Advanced Math Instructions (ST)

These ST instructions perform trigonometric, exponential, and logarithmic operations on data. Only LREAL values are allowed for all advanced math instructions except expt and abs; these can be DINT or LREAL values. The return value of all advanced math instructions are LREAL except for expt and abs. These two instructions return the data type of the input operands.

Instruction	Mnemonic	Example
Absolute Value	abs	`myLrealResult := abs(myLreal);`
Square Root	sqrt	`myLrealResult := sqrt(myLreal);`
Cosine	cos	`myLrealResult := cos(myLreal);`
Sine	sin	`myLrealResult := sin(myLreal);`
Tangent	tan	`myLrealResult := tan(myLreal);`
Arccosine	acos	`myLrealResult := acos(myLreal);`
Arcsine	asin	`myLrealResult := asin(myLreal);`
Arctangent	atan	`myLrealResult := atan(myLreal);`
Arctangent2 (Operand1 / Operand2)	atan2	`myLrealResult := atan2(myLreal,myLreal2);`
Hyperbolic Cosine	cosh	`myLrealResult := cosh(myLreal);`
Hyperbolic Sine	sinh	`myLrealResult := sinh(myLreal);`
Hyperbolic Tangent	tanh	`myLrealResult := tanh(myLreal);`

Degrees to Radians	dtor	`myLrealRadians := dtor(myLrealDegrees);`
Radians to Degrees	rtod	`myLrealDegrees := rtod(myLrealRadians);`
Log	log	`myLrealResult := log(myLreal);`
Natural Log	ln	`myLrealResult := ln(myLreal);`
Natural Exponent	exp	`myLrealResult := exp(myLreal);`
Exponent (operand1 operand2)	expt	`myLrealResult := myLrealBase ** myLrealPower;`

See also advanced math instructions in  ladder and  FBD.

Bit Shift/Rotate Instructions (ST)

These ST instructions are used to reposition the bits in DINT variables or in single elements of DINT arrays.

Instruction	Mnemonic	Example
Rotate Left	rol	`myDintResult := ROL(100, 4);`
Rotate Right	ror	`myDintResult := ROR(myDint1, 30);`
Shift Left	shl	`myDintResult := SHL(64, myDint2);`
Shift Right	shr	`myDintResult := SHR(myDint3, myDint4);`

See also bit shift/rotate instructions in  ladder and  FBD.

Conditional Statements (ST)

if ... then ... else | case

if ... then ... else

Conditional Statements

Conditional Statements allow selected statements will be executed when certain conditions exist.

```
if ... then ... else
```

Selected ST statements can be evaluated depending on the value returned by a boolean expression:

```
if <boolean expression> then
     <statements...>
end_if;
```

A boolean expression always resolves to a boolean value: True (1) or False (0).

Alternative statements can be executed using the general forms:

```
if <boolean expression> then
     <statements...>
else
     <statements...>
end_if;

- or -

if <boolean expression> then
     <statements...>
elseif <boolean expression> then
     <statements...>
end_if;

- or -
```

```
if <boolean expression> then
     <statements...>
elseif <boolean expression> then
     <statements...>
else
     <statements...>
end_if;
```

Any number of additional elseif sections can be added to the if ... then construct.

Example

```
if SwitchPosition = 0 then
     ConveyorSpeed := 0; ' Conveyor is stopped
elseif ((SwitchPosition >= 1) AND (SwitchPosition <= 2)) then
     ConveyorSpeed := 2 * SwitchPosition; ' Run at slow speeds
elseif ((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;
```

case

The **case** conditional statement will cause selected statements to execute depending on the value of an expression that returns a DINT result, for example the value of a single DINT variable or the DINT value resolved from a complex expression.

The set of statements which have a DINT selector value that matches the value of the DINT expression are executed. If no match is found, the statements after the **else** are executed.

The **case** construct has the general form:

```
case <DINT expression> of
     <DINT selector value1> : <statements...>
     <DINT selector value2> : <statements...>
     ...
```

```
          else
          <statements...>
     end_case;
```

Example

```
case SwitchPosition of
     0    : ConveyorSpeed := 0; ' Conveyor is stopped
     1,2  : ConveyorSpeed := 2 * SwitchPosition; ' Run at slow speeds
     3..5 : ConveyorSpeed := 5 * SwitchPosition; ' Run at high speeds
else
     ConveyorSpeed :=0; ' Stop the conveyor on any bad input
     ConveyorFault := #ALW_ON;
end_case;
```

BOOL, DINT, LREAL Conversions (ST)

bool_to_int | int_to_bool | int_to_real | real_to_int

bool_to_int(*Operand*)

Operand: BOOL variable, constant or expression that resolves to a BOOL.

Converts BOOL *Operand* to a DINT value.

Example

```
myDINT:= bool_to_int(myBOOL);
```

See also BOOL_TO_INT instructions in  ladder and  FBD.

int_to_bool(*Operand*)

Operand: DINT variable, constant or expression that resolves to a DINT.

Converts DINT *Operand* to a BOOL value.

Tip: If *Operand* is not zero, the result is always ON(1).

Example

```
myBOOL:= int_to_bool(myDINT);
```

See also INT_TO_BOOL instructions in  ladder and  FBD.

int_to_real(*Operand*)

Operand: DINT variable, constant or expression that resolves to a DINT.

Converts the DINT *Operand* to an LREAL value.

Example

```
myLREAL := int_to_real(myDINT);
```

See also the INT_TO_REAL instruction in FBD.

real_to_int(*Operand*)

Operand: LREAL variable, constant or expression that resolves to an LREAL.

Converts LREAL *Operand* to a DINT value.

Example

```
myDINT:= real_to_int(myLREAL);
```

Note: The decimal or fraction portion of the LREAL value is lost. For example, the LREAL value 2.99 becomes the DINT value 2 after the conversion.

See also the REAL_TO_INT instruction in  FBD.

Iteration Statements (ST)

for...do | while...do | repeat...until | exit | return

With iteration statements, you can repeat one or more statements a number of times depending on the state of a particular variable or condition.

Note: Iteration statements should be constructed carefully to avoid endless loops. Iteration statements may also significantly increase the time to execute some software elements, such as function blocks.

for ... do Statement

The **for** ... **do** construct allows a set of statements to be repeated depending on the value of a DINT iteration variable. This construct takes the general form:

```
for <initialise iteration variable>
     to <final value expression>
     [by <increment expression>] do
     <statements...>
end_for;
```

The **for** ... **do** construct can be used to count iterations counting up or down and using any size increment until a final value is reached. If the **by** keyword is omitted, the iteration variable will increase by 1.

The test to check whether the iteration has reached the final value is made before executing the statements inside the **for** ... **do** construct. It is, therefore, possible for the final increment to cause the value of the iteration variable to exceed the final value, in which case the statements inside the **for** ... **do** construct will not be executed.

Notes

- It is a best practise to not use the iteration variable outside the for loop.
- The statements within a **for** ... **do** construct should not modify variables that will affect the expressions for the final and increment values.

Example

```
Sum := 0;
' Calculate the sum of the array elements
for Index := 0 to 9 by 1 do
     Sum := Sum + DintArray[Index];
end_for;
```

while ... do Statement

The **while** ... **do** construct can be used to execute one or more statements while a particular boolean expression remains True (1). The boolean expression is tested prior to executing the statements, the statements within the **while** ... **do** are executed only while it is set to True (1). This construct takes the general form:

```
while <boolean expression> do
     <statements...>
end_while;
```

Note: The **while** <BOOL expression> should not use a value that must be read at the beginning of the scan. Such a value would not change during the execution part of the scan and would cause an endless loop that would fault the watchdog timer. If you want to refer to a value read at the beginning of the scan, use an if statement to test the value and take action once based on the test; every scan would effectively repeat the test, which would cause the action to continue or stop.

Example

```
Index := 0;
Sum := 0;
'Another sum of an array
while (Index <= 9)  do
     Sum := Sum + DintArray[Index];
Index := Index +1;
end_while;
```

repeat ... until Statement

The **repeat** ... **until** construct can be used to execute one or more statements while a particular boolean expression remains True (1). The boolean expression is tested after executing the statements, the statements within the **repeat** ... **until** are executed again after it becomes False (0). This construct takes the general form:

```
repeat
     <statements...>
until <boolean expression>
end_repeat;
```

Note: Make sure the **until** <boolean expression> does not refer to, or depend on, a value that must be read at the beginning of the scan. Such a value would not change during the execution part of the scan and would cause an endless loop that would fault the watchdog timer. If you want to refer to a value read at the beginning of the scan, use an if statement to test the value and take action once based on the test; every scan would effectively repeat the test, which would cause the action to continue or stop.

Example

```
Index := 0;
Sum := 0;
'Another way to sum an array
repeat
     Sum := Sum + DintArray[Index];
     Index := Index +1;
until Index < 10
end_repeat;
```

exit Statement

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

return Statement

The **return** statement:

- Can be used only within function and program block bodies and is used to return prematurely from the code body.
- Causes execution to continue from the end of the function or function block body.

A Return instruction exists also in  ladder.