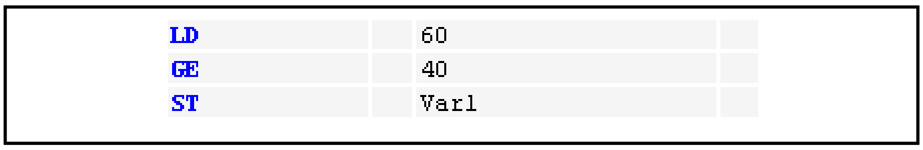
GE

IEC relational operator: Greater than or equal to

A Boolean operator with the result TRUE if the first operand is greater than or equal to the second operand.

The operands can be any of the following types:

- BOOL, BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL,
- TIME, LTIME, DATE, TIME_OF_DAY, DATE_AND_TIME and
- STRING.



```
LD    60
GE    40
ST    Var1
```

Fig.5-72: Operator GE in IL

Result is TRUE

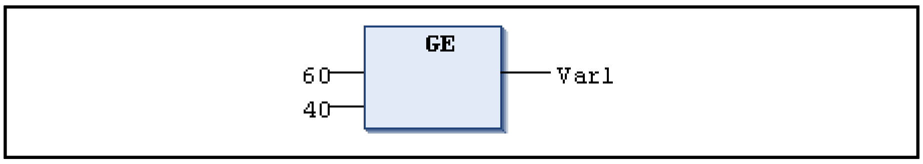


Fig.5-73: Operator GE in FBD

Operator GE in ST

```
VAR1 := 60 >= 40;  // TRUE
```

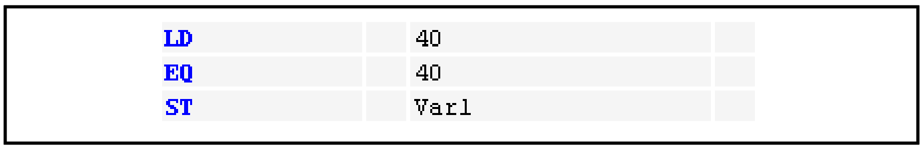
EQ

IEC relational operator: Equality

A Boolean operator with the result TRUE if the operands are equal.

The operands can be any of the following types:

- BOOL, BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL,
- TIME, LTIME, DATE, TIME_OF_DAY, DATE_AND_TIME and
- STRING.



```
LD    40
EQ    40
ST    Var1
```

Fig.5-74: Operator EQ in IL

Result is TRUE

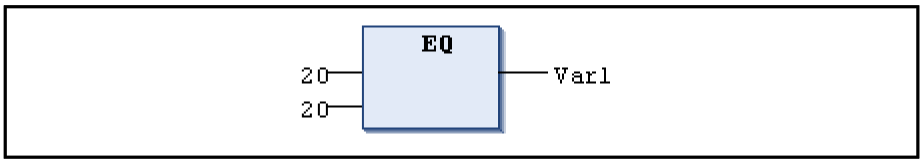


Fig.5-75: Operator EQ in FBD

Operator EQ in ST

```
VAR1 := 40 = 40; // TRUE
```

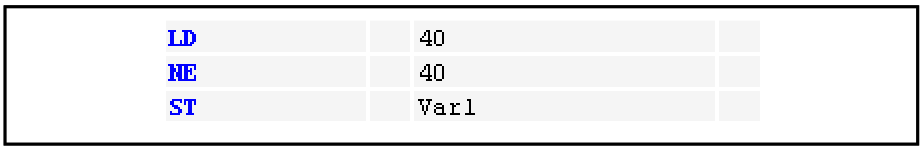
NE

IEC relational operator: Inequality

A Boolean operator with the result TRUE if the operands are not equal.

The operands can be any of the following types:

- BOOL, BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL,
- TIME, LTIME, DATE, TIME_OF_DAY, DATE_AND_TIME and
- STRING.



```
LD   40
NE   40
ST   Var1
```

Fig.5-76: Operator NE in IL

Result is FALSE

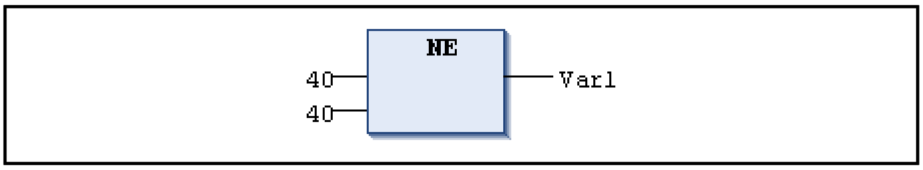


Fig.5-77: Operator NE in FBD

Operator NE in ST

```
VAR1 := 40 <> 40; // FALSE
```

5.3.7 Address Operators

Address Operators, Overview

In IndraLogic 2G, the following as address operators are available in the extension of the IEC 61131-3 standard:

-
-
-

ADR

This is not described in the IEC 61131-3 standard.

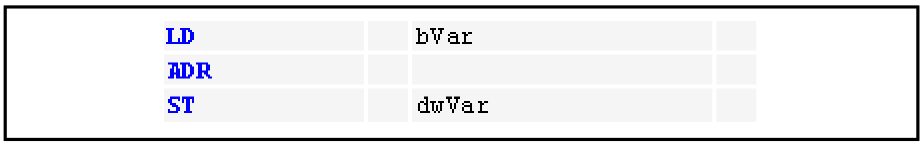
of its argument in a DWORD. This address can be sent to the vendor functions and treated there like a pointer or it can be assigned to a in the project.

in contrast to IndraLogic1.x, the ADR operator can now be used with function, program, function block and method names and thus replaces the INDEXOF operator.

In this context, note the following as well: '
'.

Always remember that function pointers can be forwarded to external libraries, but that **a function pointer cannot be called in IndraLogic!**

To enable a system call (runtime system), the corresponding object property ("Compile" tab) has to be set for the function tab.



LD	bVar
ADR	
ST	dwVar

Fig.5-78: Operator ADR in IL

Operator ADR in ST

```
dwVar:=ADR(bVAR);
```

If an   is applied, the contents of addresses can be shifted.

Thus, POINTER variables can point to an invalid memory space. To avoid problems, ensure that the pointer is not memorized but updated in each cycle.

"Pointer TO Variables" of functions and methods should not be returned to the user or assigned to the global variables.

Content Operators

This   is not described in the IEC61131-3 standard.

It allows a   to be dereferenced.

It is attached to the pointer identifier as "**^**".

Content operator in ST:

```
VAR
 pt:POINTER TO INT;
 var_int1:INT;
 var_int2:INT;
END_VAR
 pt := ADR(var_int1);
 var_int2:=pt^;
```

is applied, the contents of addresses can be shifted. Note when using pointers to point to addresses.

BITADR

This   is not described in the IEC 61131-3 standard.

BITADR returns the bit offset within a segment in a DWORD.

Note that the offset depends on whether the byte addressing option is enabled in the target system settings.

The current version of the implementation includes the following special features:

1. The most significant byte of the double word contains the additional information whether it is dealt with a flag variable (16#40), input variable (16#80) or output variable (16#C0).

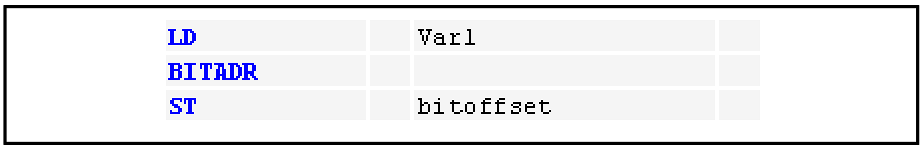
   If necessary, this part has to be masked out.
2. If the address was assigned with the complete path specification

   `<ApplicationName> <BlockName>.<VariableName>`

   , the compiler returns an error message.
3. If the address of a global variable was directly assigned, the compiler functions without an error.

Declaration:

```
VAR
 var1 AT %IX2.3:BOOL;
 bitoffset: DWORD;
END_VAR
```



```
LD      Var1
BITADR
ST      bitoffset
```

Fig.5-79: Operator BITADR in IL

Operator BITADR in ST

```
bitoffset:=BITADR(var1);
    (* Result if byte addressing=TRUE: 16x80000013,
    if byte addressing=FALSE: 16x80000023 *)
```

is applied, the contents of addresses can be shifted. Note when using pointers to point to addresses.

5.3.8 Call Operators

CAL, CALC, CALCN

IEC operator for calling a function block.

In IL, the or an is called with CAL.

In addition to the "CAL" operator, the modifications

- CALC; call is made only if the preceding expression has the value 'TRUE'
- CALCN; call is made only if the preceding expression has the value 'FALSE'

are permitted.

Following the name of the instance of a function block in parentheses is the assignment of the input variables of the function block and the value output.

Calling the instance "Inst" of a function block with the assignment of the input variables Par1, Par2 to 0 or TRUE.

Operator CAL in IL:

```
CAL            ZAB(         // Invocation of instance ZAB of FB TON
          IN:= FALSE,       // Input parameters
PT:= WAITINGTIME,
ET=> ETVar)       // Result
```

5.3.9 Type Conversion Operators

Type Conversion Operators, Overview

IndraLogic 2G allows operands of different data types to be connected, although an automatic **implicit** type conversion is attempted first.

It is not permitted to implicitly convert from a "higher" type to a "smaller" type (for example from INT to BYTE or from DINT to WORD). If this should be done, use special type conversions. In principle, every elementary type can be converted into any other elementary type.

Implicit type conversion, BYTE ⇒ USINT

```
VAR
 byte1:BYTE;
 usint1:USINT;
END_VAR
 usint1:=usint1+byte1;
```

The compiler converts **BYTE** into **USINT** without a warning or error message.

Implicit type conversion, BYTE ⇒ SINT

```
VAR
 byte1:BYTE;
 sint1:SINT;
END_VAR
 sint1:=sint1+byte1;
```

The compiler converts **BYTE** into **SINT** with the **warning**:

Implicit conversion of the unsigned data type "BYTE" to the signed data type "SINT": Possible loss of sign;

If the difference between the respective data types increases, the compiler returns an **error message.**

The **explicit** type conversion can also be used.

In principle, every elementary type can be converted into any other elementary type.

Syntax:

```
<elem.Typ1>_TO_<elem.Typ2>
```

The following type conversions are supported:

-
-
-
-
-
-
-

New compared to IndraLogic 1.x:

-
-
-
-

☞ For ...TO_STRING conversions, the string (character string) is generated left-justified. If it has been defined too short, it is cut off from the right.

BOOL_TO Conversions

Conversion from data type BOOL to another type.

Syntax:

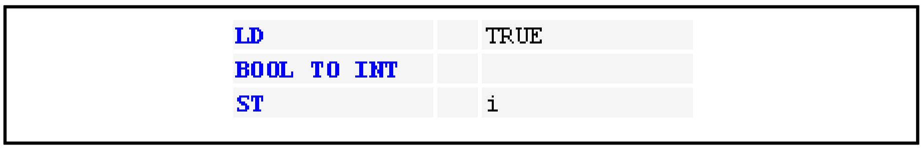
```
BOOL_TO_<data type>
```

For number types, the result is 1 if the operand is TRUE and 0 if the operand is FALSE.

For STRING types, the result is TRUE or FALSE.

Example:

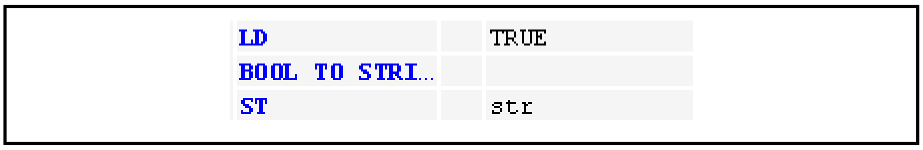
Operator BOOL_TO in IL



```
LD            TRUE
BOOL_TO_INT
ST            i
```

Fig.5-80: BOOL TO INTEGER

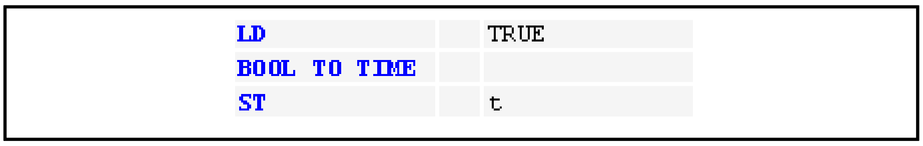
Result is 1



```
LD            TRUE
BOOL_TO_STRI...
ST            str
```

Fig.5-81: BOOL TO STRING

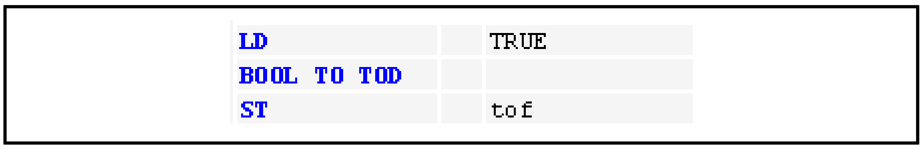
RESULT is TRUE



```
LD            TRUE
BOOL_TO_TIME
ST            t
```

Fig.5-82: BOOL TO TIME

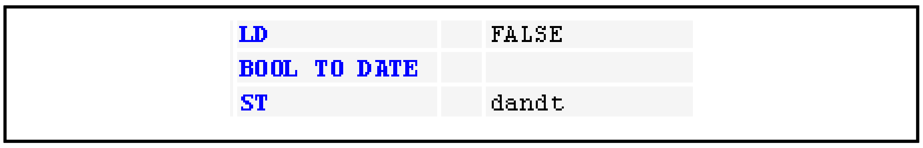
Result is T#1ms



```
LD            TRUE
BOOL_TO_TOD
ST            tof
```

Fig.5-83: BOOL TO TOD

Result is TOD#00:00:00.001



```
LD            FALSE
BOOL_TO_DATE
ST            dandt
```

Fig.5-84: BOOL TO DATE

Result is D#1970-01-01

```
LD          TRUE
BOOL_TO_DT
ST          dandt
```

Fig.5-85: BOOL TO DT

Result is DT#1970-01-01-00:00:01

Operator BOOL_TO_ in ST

```
(*BOOL TO INTEGER:*)
i:=BOOL_TO_INT(TRUE); (*result is 1*)

(*BOOL TO STRING:*)
str:=BOOL_TO_STRING(TRUE); (*result is "TRUE"*)

(* BOOL TO TIME: *)
t:=BOOL_TO_TIME(TRUE); (*result is T#1ms*)

(*BOOL TO TIME OF DAY:*)
tof:=BOOL_TO_TOD(TRUE); (*result is TOD#00:00:00.001*)

(*BOOL TO DATE:*)
dat:=BOOL_TO_DATE(FALSE); (*result is D#1970*)

(*BOOL TO DATE AND TIME:*)
dandt:=BOOL_TO_DT(TRUE); (*result is DT#1970-01-01-00:00:01*)
```

Example:

Operator BOOL TO in FBD:

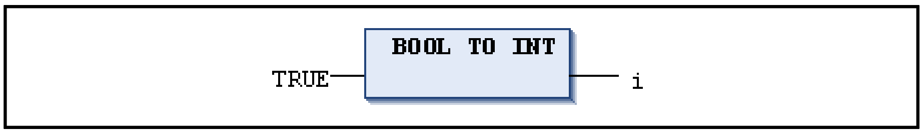


Fig.5-86: Operator BOOL TO Integer

Result is 1

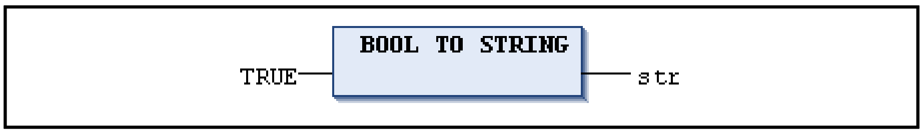


Fig.5-87: Operator BOOL TO STRING

Result is TRUE

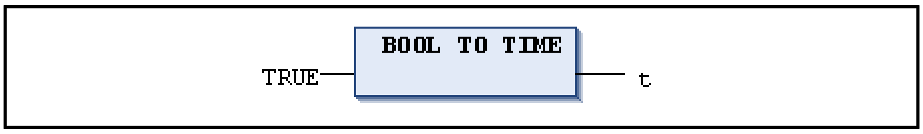


Fig.5-88: Operator BOOL TO TIME

Result is T#1ms

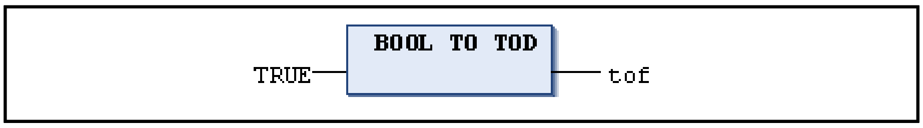


Fig.5-89: Operator BOOL TO TOD

Result is TOD#00:00:00.001

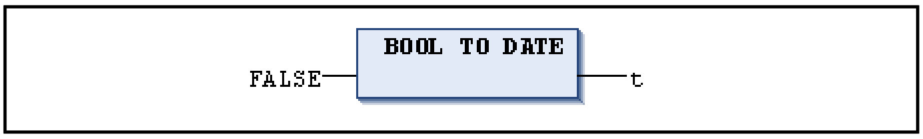


Fig.5-90: Operator BOOL TO DATE

Result is D#1970-01-01

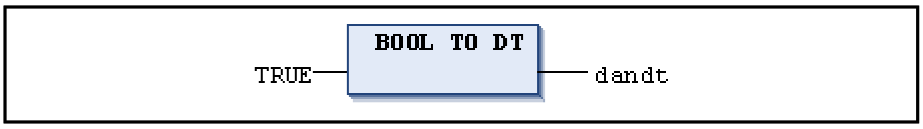


Fig.5-91: Operator BOOL TO DATE AND TIME

Result is DT#1970-01-01-00:00:01

TO_BOOL Conversions

Conversion from any data type to BOOL.

Syntax:

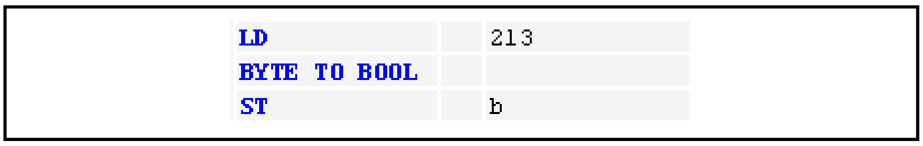
```
<data type>_TO_BOOL
```

The result is TRUE if the operand is not equal to 0. The result is FALSE if the operand is equal to 0.

For STRING types, the result is TRUE if the operand is 'TRUE'. Otherwise, the result is FALSE.

Example:

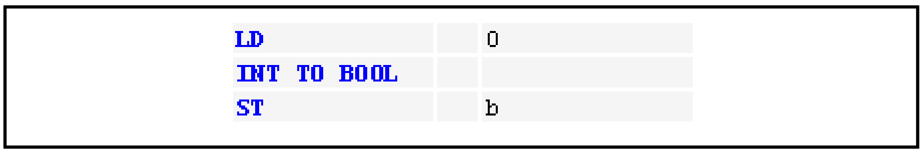
Operator TO BOOL in IL:



```
LD              213
BYTE TO BOOL
ST              b
```

Fig.5-92: Operator BYTE TO BOOL

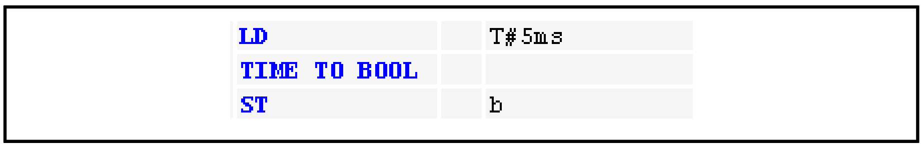
Result is TRUE



```
LD              0
INT TO BOOL
ST              b
```

Fig.5-93: Operator INTEGER TO BOOL

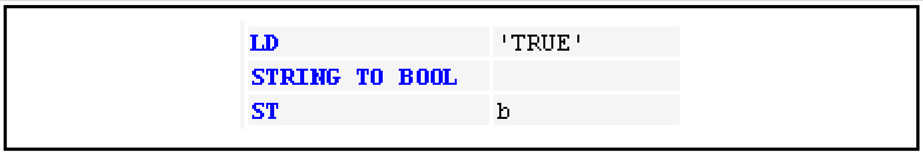
Result is FALSE



```
LD              T#5ms
TIME TO BOOL
ST              b
```

Fig.5-94: Operator TIME TO BOOL

Result is TRUE



```
LD                'TRUE'
STRING TO BOOL
ST                b
```

Fig.5-95: Operator STRING TO BOOL

Result is TRUE

Operator TO_BOOL in ST

```
(*BYTE TO BOOL:*)
b := BYTE_TO_BOOL(2#11010101);(*result is TRUE*)

(*INTEGER TO BOOL:*)
b := INT_TO_BOOL(0); (*result is FALSE*)

(*TIME TO BOOL:*)
b := TIME_TO_BOOL(T#5ms); (*result is TRUE *)

(*STRING TO BOOL:*)
b := STRING_TO_BOOL('TRUE'); (*result is TRUE*)
```

Example:

Operator TO BOOL in FBD:

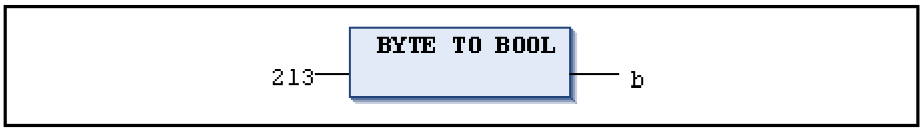


Fig.5-96: Operator BYTE TO BOOL

Result is TRUE

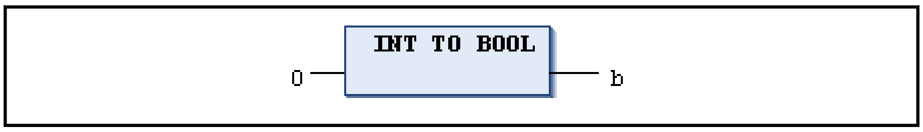


Fig.5-97: Operator INTEGER TO BOOL

Result is FALSE

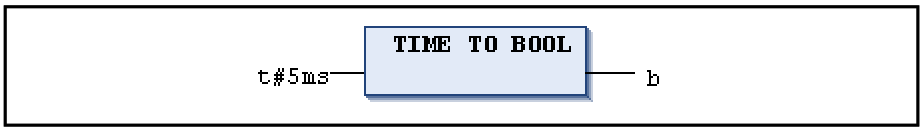


Fig.5-98: Operator TIME TO BOOL

Result is TRUE

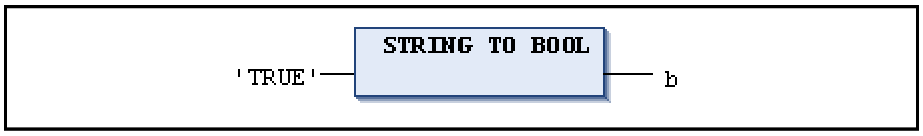


Fig.5-99: Operator STRING TO BOOL

Result is TRUE

Conversion of Integer Types

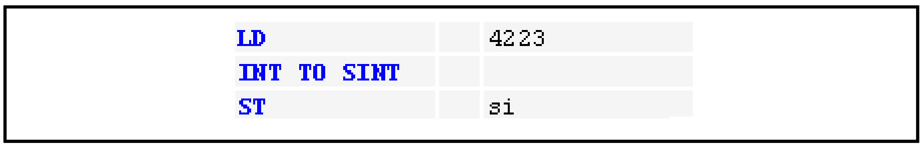
Conversion of an integer type to another number type:

Syntax:

```
<INT data type>_TO_<INT data type>
```

☞ When converting larger types to smaller types, information can be lost.

If the number to be converted exceeds the range limit, the initial bytes of the number are not considered.



```
LD            4223
INT TO SINT
ST            si
```

Fig.5-100: Operator INT TO SINT in IL

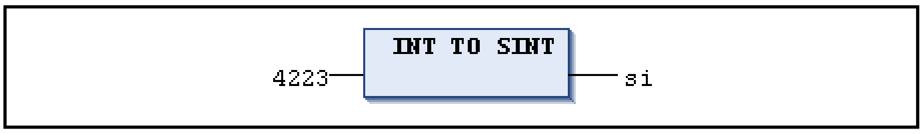


Fig.5-101: Operator INT TO SINT in FBD

☞ Save the integer 4223 (16#107f in hexadecimal) to a SINT variable. The, it gets the number 127 (16#7f in hexadecimal).

Operator INTEGER TO SIGNED INTEGER in ST

```
si := INT_TO_SINT(4223); (*result is 127*)
```

REAL_TO/LREAL_TO Conversions

Conversion from REAL or LREAL type to another type.

It is rounded up or down to an integer value and converted into the respective type. Exceptions are the types STRING, BOOL, REAL and LREAL.

☞ If REAL or LREAL is converted into SINT, USINT, INT, UINT, DINT, UDINT, LINT or ULINT and the value of the REAL/LREAL number is outside the value range of the integer, the result is undefined and depends on the target system.

An exception is then possible!

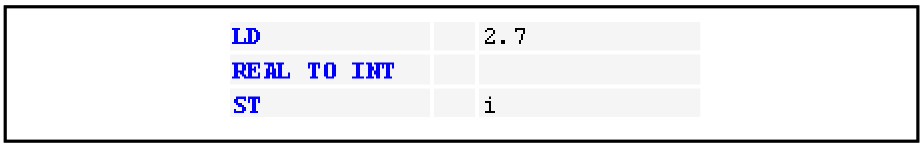
To get a target system-independent code, value range exceedances have to be intercepted via the application.

If the REAL/LREAL number is within the range, the conversion runs equally on all systems.

Note that for conversions into STRING type, the number of decimal places is limited to 16. IF the (L)REAL number contains more places, the 16th place is rounded and displayed in the STRING.

If the STRING defined is too short for the number, the number is shortened from the right.

☞ When converting larger types to smaller types, information can be lost.



```
LD            2.7
REAL TO INT
ST            i
```

Fig.5-102: Operator REAL TO INT in IL

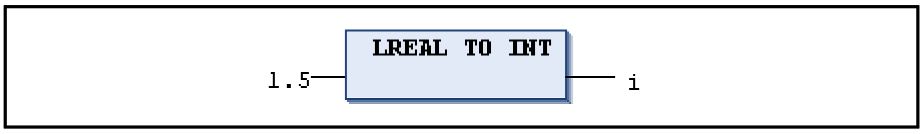


Fig.5-103: Operator LREAL TO INT

Operator REAL TO INTEGER in ST

```
i := REAL_TO_INT(1.5); (*result is 2*)
j := REAL_TO_INT(1.4); (*result is 1*)
k := REAL_TO_INT(-1.5); (*result is -2*)
l := REAL_TO_INT(-1.4); (*result is -1*)
```

TIME_TO/TIME_OF_DAY Conversions

Conversion from TIME or TIME_OF_DAY type to another type.

Syntax:

```
<TIME-data type>_TO_<data type>
```

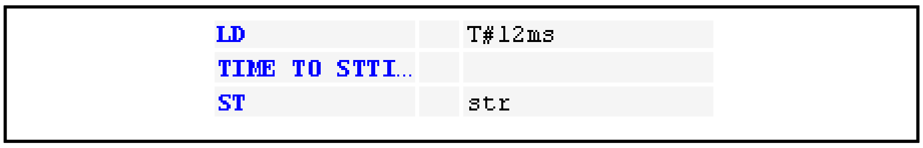
The time is internally saved into a DWORD in milliseconds (for TIME_OF_DAY starting at 00:00 o'clock). This value is converted.

For STRING types, the result is the time constant.

☞ When converting larger types to smaller types, information can be lost.

Example:

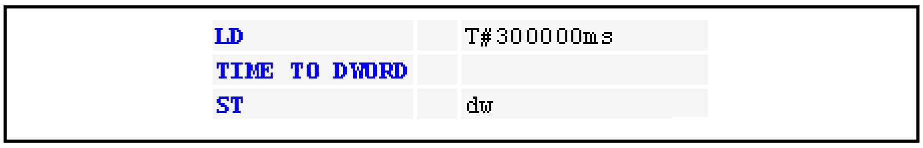
Operator TIME TO in IL



```
LD            T#12ms
TIME TO STTI…
ST            str
```

Fig.5-104: Operator TIME TO STRING

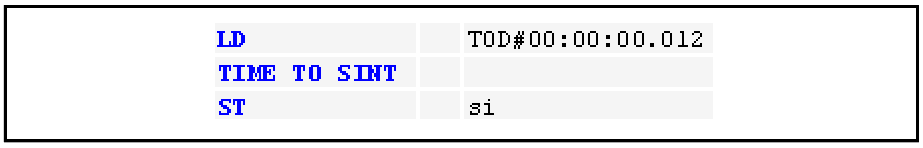
Result is T#12ms



```
LD            T#300000ms
TIME TO DWORD
ST            dw
```

Fig.5-105: Operator TIME TO DWORD

Result is 300000



```
LD              TOD#00:00:00.012
TIME TO SINT
ST              si
```

Fig.5-106: Operator TIME TO SINT

Result is 12

Operator TIME TO in ST

```
(*TIME TO STRING:*)
str:=TIME_TO_STRING(T#12ms); (*result is T#12ms*)
(*TIME TO DWORD:*)
dw:=TIME_TO_DWORD(T#5m); (*result is 300000*)
(*TIME TO SINT:*)
si:=TOD_TO_SINT(TOD#00:00:00.012); (*result is 12*)
```

Example:

Operator TIME TO in FBD

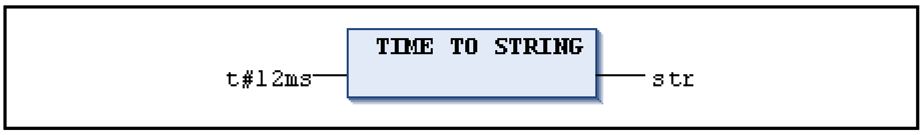


Fig.5-107: Operator TIME TO STRING

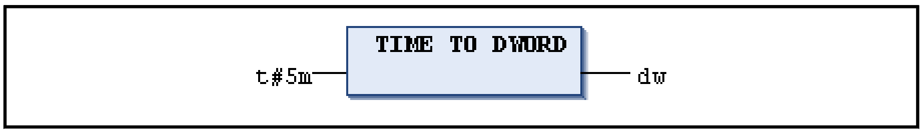


Fig.5-108: Operator TIME TO DWORD

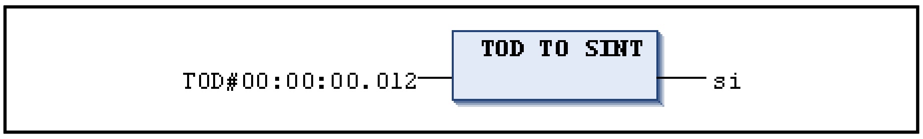


Fig.5-109: Operator TOD TO SINT

DATE_TO/DT_TO Conversions

IEC operator: from DATE or DATE_AND_TIME type to another type.

Syntax:

```
<DATE data type>_TO_<data type>
```

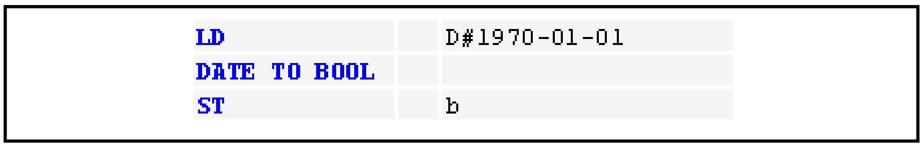
The date is internally saved in seconds starting on January 1st 1970. This value is converted.

For STRING types, the result is the date constant.

☞ When converting larger types to smaller types, information can be lost.

Example:

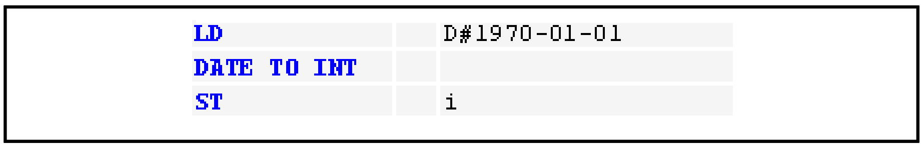
Operator DATE TO in IL



```
LD              D#1970-01-01
DATE TO BOOL
ST              b
```

Fig.5-110: Operator DATE TO BOOL

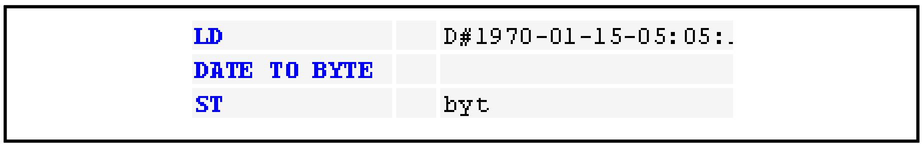
Result is FALSE



```
LD              D#1970-01-01
DATE TO INT
ST              i
```

Fig.5-111: Operator DATE TO INT

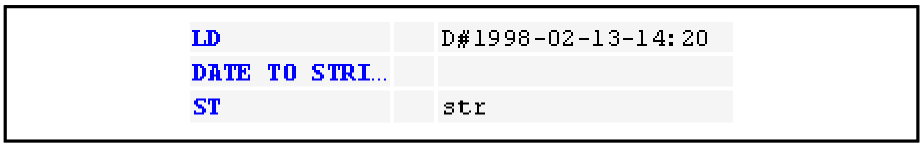
Result is 29952



```
LD              D#1970-01-15-05:05:.
DATE TO BYTE
ST              byt
```

Fig.5-112: Operator DATE TO BYTE

Result is 129



```
LD              D#1998-02-13-14:20
DATE TO STRI...
ST              str
```

Fig.5-113: Operator DATE TO STRING

Result is DT#1998-02-13-14:20

Operator DATE TO in ST:

```
(*Operator DATE_TO_BOOL:*)
b:=DATE_TO_BOOL(D#1970-01-01); (*result is FALSE*)

(*Operator DATE_TO_INT:*)
i :=DATE_TO_INT(D#1970-01-15); (*result is 29952*)

(*Operator DATE_TO_BYTE:*)
byt:=DT_TO_BYTE(DT#1970-01-15-05:05:05); (*result is 129*)

(*Operator DATE_TO_STRING:*)
str:=DT_TO_STRING(DT#1998-02-13-14:20); (*result is )
                                          'DT#1998-02-13-14:20'*)
```

Example:

Operator DATE TO in FBD

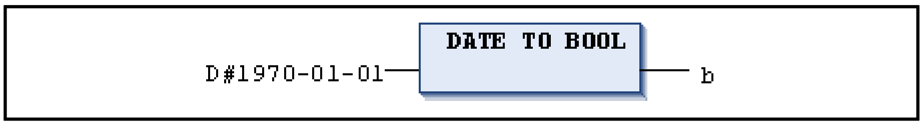


Fig.5-114: Operator DATE TO BOOL

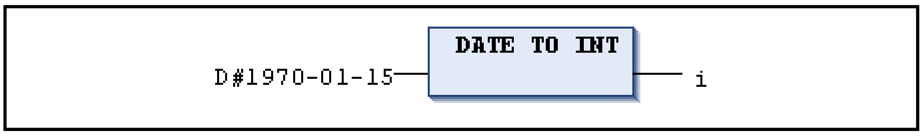


Fig.5-115: Operator DATE TO INT

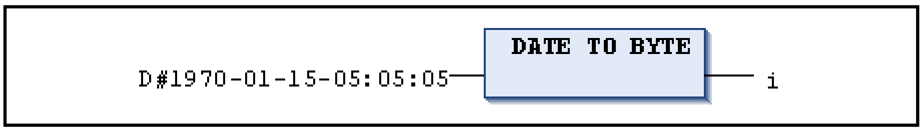


Fig.5-116: Operator DATE TO BYTE

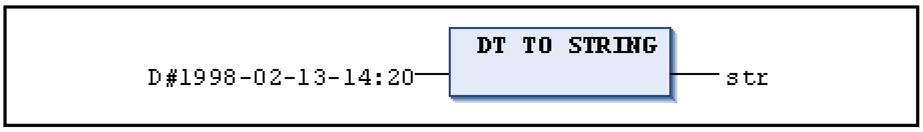


Fig.5-117: Operator DT TO STRING:

STRING_TO Conversions

IEC operator: from STRING type to another type.

The conversion runs according to standard C: STRING to INT and then INT to BYTE. The higher byte is cut, i.e. only values from 0 - 255 result.

Syntax:

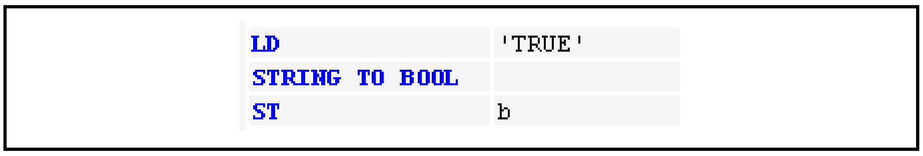
```
STRING_TO_<data type>
```

The operand of type STRING has to have a valid value of the target type. Otherwise, the result is 0.

☞ When converting larger types to smaller types, information can be lost.

Example:

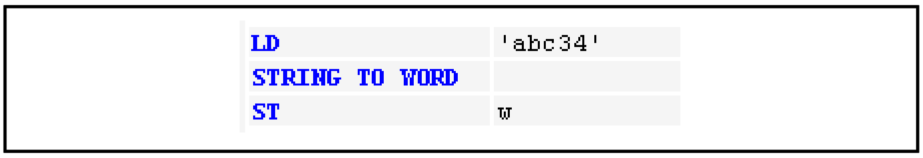
Operator STRING_TO in IL



```
LD              'TRUE'
STRING TO BOOL
ST              b
```

Fig.5-118: Operator STRING TO BOOL

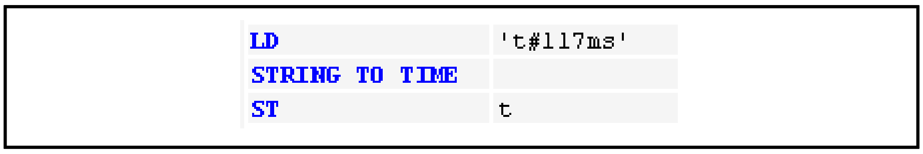
Result is TRUE



```
LD              'abc34'
STRING TO WORD
ST              w
```

Fig.5-119: Operator STRING TO WORD

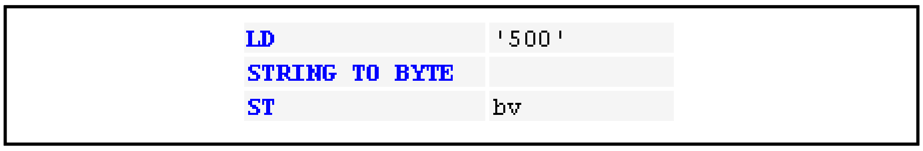
Result is 0



```
LD              't#117ms'
STRING TO TIME
ST              t
```

Fig.5-120: Operator STRING TO TIME

Result is T#117ms



```
LD              '500'
STRING TO BYTE
ST              bv
```

Fig.5-121: Operator STRING TO BYTE

Result is 244

Operator STRING_TO in ST

```
(*Operator STRING_TO_BOOL:*)
b:=STRING_TO_BOOL('TRUE'); (*result is TRUE*)

(*Operator STRING_TO_WORD:*)
w:=STRING_TO_WORD('abc34'); (*result is 0*)

(*Operator STRING_TO_TIME:*)
t:=STRING_TO_TIME('T#127ms'); (*result is T#127ms*)

(*Operator STRING_TO_BYTE:*)
bv:=STRING:TO_BYTE('500'); (*result is 244*)
```

Example:

Operator STRING TO in FBD

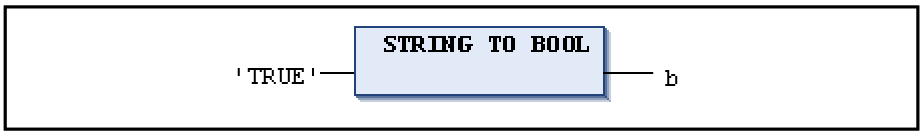


Fig.5-122: Operator STRING TO BOOL

Result is TRUE

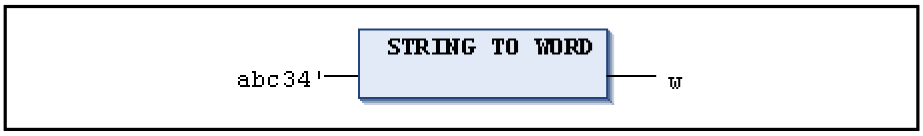


Fig.5-123: Operator STRING TO WORD

Result is 0

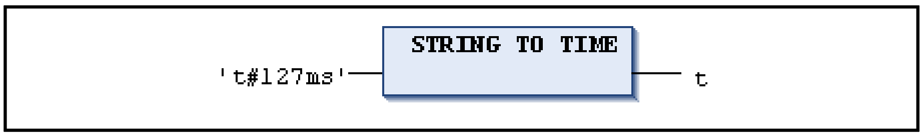


Fig.5-124: Operator STRING TO TIME

Result is T#127ms

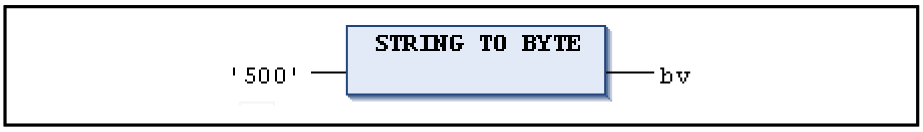


Fig.5-125: Operator STRING TO BYTE

Result is 244

TRUNC

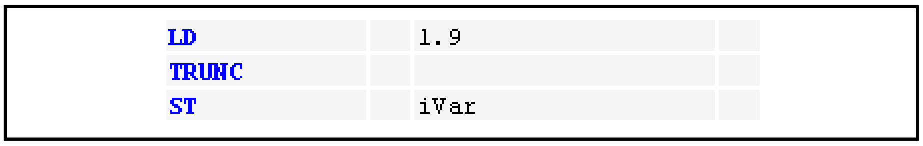
Conversion from REAL type to **DINT** type. Only the **value of the integer part of the number** is used.

In IndraLogic 1.x, the TRUNC operator converts from REAL to INT.

If a 1.x project is imported, TRUNC is automatically replaced by TRUNC_INT.

Operator TRUNC in ST

```
i:=TRUNC(1.9); (*result is 1*)
j:=TRUNC(-1.4); (*result is -1*)
```



```
LD      1.9
TRUNC
ST      iVar
```

Fig.5-126: Operator TRUNC in IL

TRUNC_INT

Conversion from REAL type to **INT** type. Only the value of the integer part of the number is used.

Note: TRUNC_INT corresponds to the TRUNC operator in IndraLogic 1.x and when 1.x projects are imported, it is automatically used at its place. Note the different function of TRUNC in IndraLogic 2G.

Operator TRUNC_INT in IL

```
LD                  1.9
TRUNC_INT
ST                  iVar
```

Operator TRUNC_INT in ST

```
i:=TRUNC_INT(1.9);  (*result is 1*)
i:=TRUNC_INT(-1.4); (*result is -1*)
```

ANY_TO Conversions

Conversion of any data type ANY or a specific numeric data type ANY_NUM into another type. Reasonable selection of data types is required.

Syntax:

```
ANY_NUM_TO_<numeric data type>
ANY_TO_<any data type>
```

Operator ANY_..._TO in ST

```
VAR
  re: REAL := 1.234;
  i:  INT;
END_VAR
  i:= ANY_TO_INT(re)// REAL to INT
```

5.3.10 Numeric Operators

Numeric Operators, Overview

The following numeric operators described in the IEC 61131-3 standard are supported:

-
-
-
-
-
-
-
-
-
-
-
-

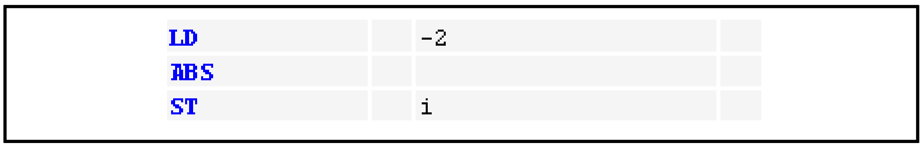
ABS

Numeric   : Returns the absolute value of a number.

The following type combinations for IN and OUT are possible:

IN	OUT
SINT	SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, REAL, LREAL, BYTE, WORD, DWORD, LWORD
INT	INT, DINT, LINT, UINT, UDINT, ULINT, REAL, LREAL, WORD, DWORD, LWORD
DINT	DINT, LINT, UDINT, ULINT, REAL, LREAL, DWORD, LWORD
LINT	LINT, ULINT, REAL, LREAL, LWORD
USINT	USINT, UINT, UDINT, ULINT, REAL, LREAL, BYTE, WORD, DWORD, LWORD
UINT	UINT, UDINT, ULINT, REAL, LREAL, WORD, DWORD, LWORD
UDINT	UDINT, ULINT, REAL, LREAL, DWORD, LWORD
ULINT	ULINT, REAL, LREAL, LWORD
REAL	REAL, LREAL
LREAL	LREAL
BYTE	USINT, UINT, UDINT, ULINT, REAL, LREAL, BYTE, WORD, DWORD, LWORD
WORD	UINT, UDINT, ULINT, REAL, LREAL, WORD, DWORD, LWORD

DWORD	UDINT, ULINT, LREAL, DWORD, LWORD
LWORD	ULINT, LREAL, LWORD



```
LD      -2
ABS
ST      i
```

Fig.5-127: Operator ABS in IL

Result in i is 2

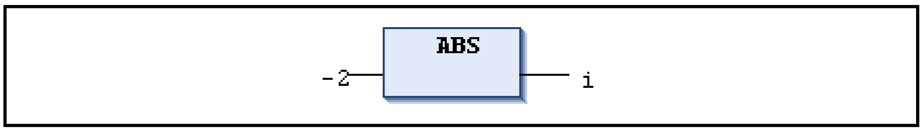


Fig.5-128: Operator ABS in FBD

Operator ABS in ST

```
i:=ABS(-2);
```

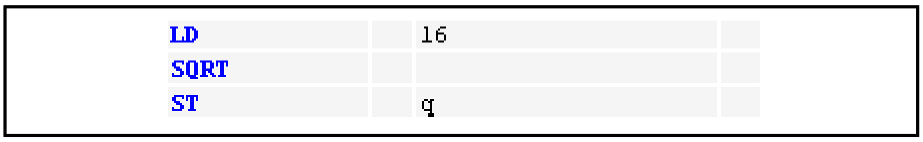
SQRT

Numeric : Returns the square root of a number.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD      16
SQRT
ST      q
```

Fig.5-129: Operator SQRT in IL

Result in q is 4

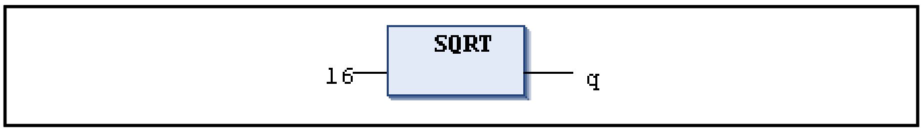


Fig.5-130: Operator SQRT in FBD

Operator SQRT in ST

```
q:=SQRT(16);
```

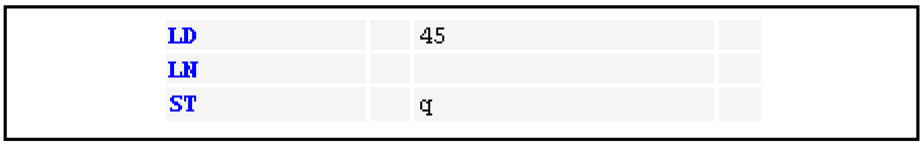
LN

Numeric : Returns the natural logarithm of a number.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD    45
LN
ST    q
```

Fig.5-131: Operator LN in IL

Result in q is 3.80666

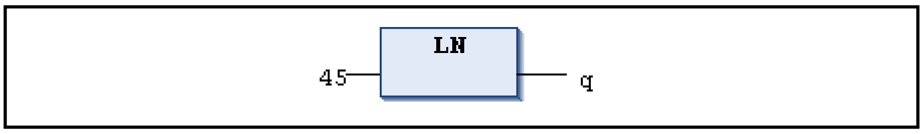


Fig.5-132: Operator LN in FBD

Operator LN in ST:

```
q:=LN(45);
```

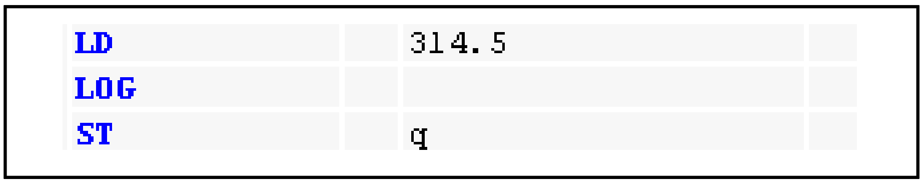
LOG

Numeric : Returns the logarithm to the basis of 10 to a number.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD    314.5
LOG
ST    q
```

Fig.5-133: Operator LOG in IL

Result in q is 2.49762

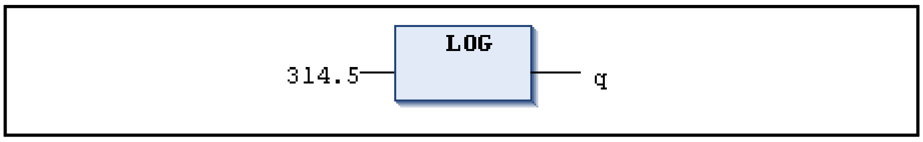


Fig.5-134: Operator LOG in FBD

Operator LOG in ST

```
q:=LOG(314.5);
```

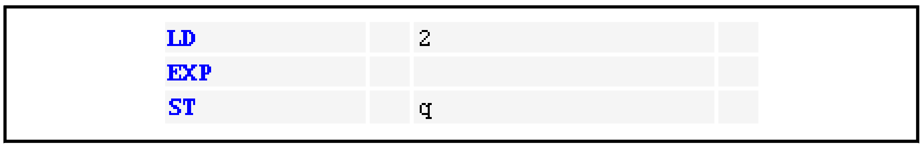
EXP

Numeric : Returns the exponential function.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD      2
EXP
ST      q
```

Fig.5-135: Operator EXP in IL

Result in q is 7.389056099

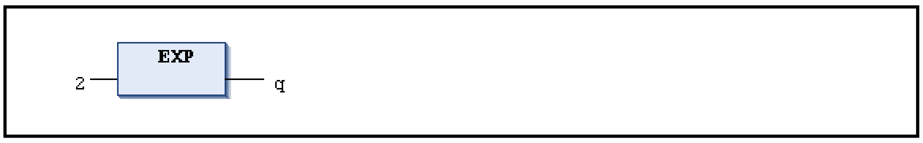


Fig.5-136: Operator EXP in FBD

Operator EXP in ST

```
q:=EXP(2);
```

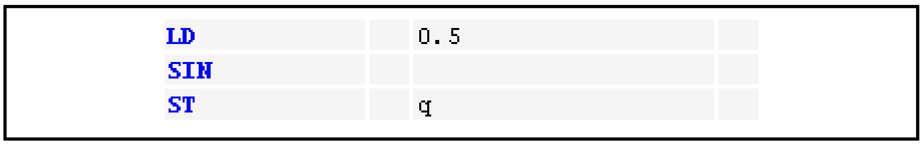
SIN

: Returns the sine value of a number. The value is calculated in radians.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD      0.5
SIN
ST      q
```

Fig.5-137: Operator SIN in IL

Result in q is 0.479426

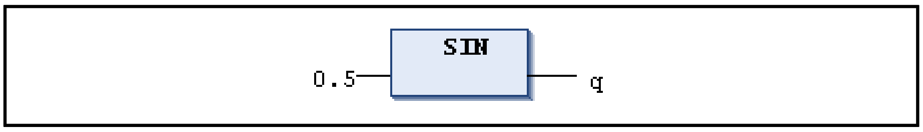


Fig.5-138: Operator SIN in FBD

Operator SIN in ST

```
q:=SIN(0.5);
```

COS

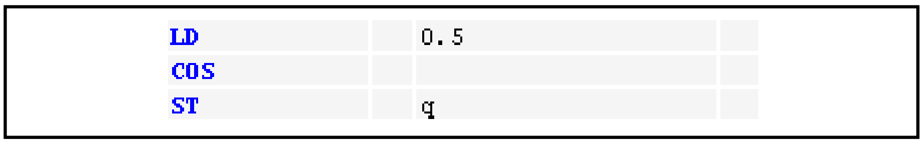
: Returns the cosine value of a number. The value is calculated in radians.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.

Programming Reference



```
LD    0.5
COS
ST    q
```

Fig.5-139: Operator COS in IL

Result in q is 0.877583

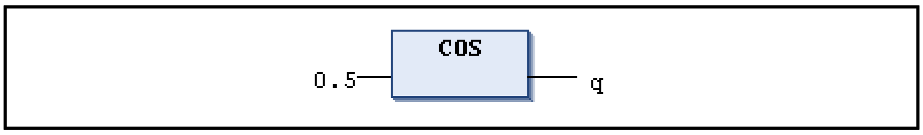


Fig.5-140: Operator COS in FBD

Operator COS in ST

```
q:=COS(0.5);
```

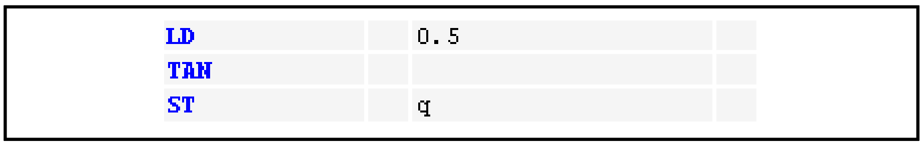
TAN

: Returns the tangent value of a number. The value is calculated in radians.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD    0.5
TAN
ST    q
```

Fig.5-141: Operator TAN in IL

Result in q is 0.546302

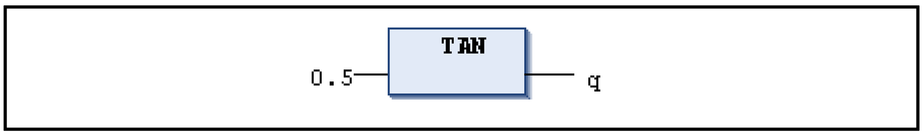


Fig.5-142: Operator TAN in FBD

Operator TAN in ST

```
q:=TAN(0.5);
```

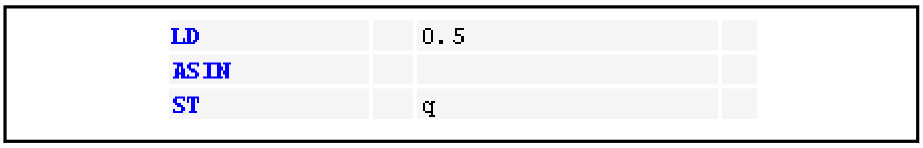
ASIN

: Returns the arc sine value (inverse function of sine) of a number. The value is calculated in radians.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD      0.5
ASIN
ST      q
```

Fig.5-143: Operator ASIN in IL

Result in q is 0.523599

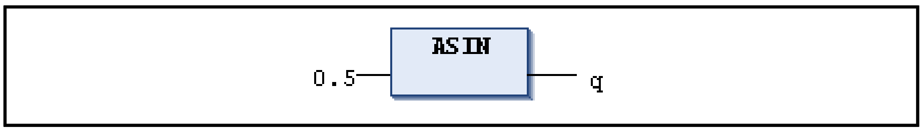


Fig.5-144: Operator ASIN in FBD

Operator ASIN in ST

```
q:=ASIN(0.5);
```

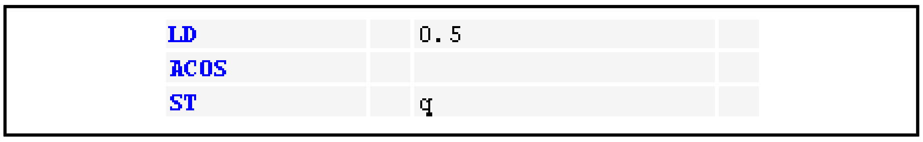
ACOS

: Returns the arc cosine value (inverse function of cosine) of a number. The value is calculated in radians.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD      0.5
ACOS
ST      q
```

Fig.5-145: Operator ACOS in IL

Result in q is 1.0472

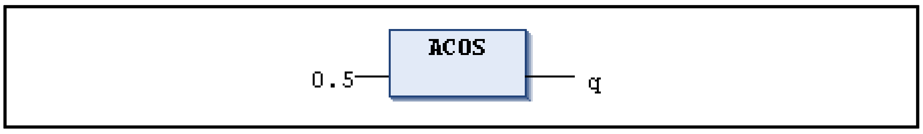


Fig.5-146: Operator ACOS in FBD

Operator ACOS in ST

```
q:=ACOS(0.5);
```

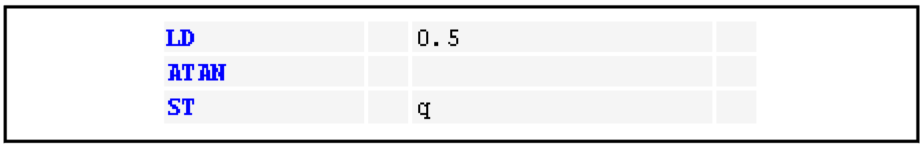
ATAN

: Returns the arc tangent value (inverse function of tangent) of a number. The value is calculated in radians.

IN can be one of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD      0.5
ATAN
ST      q
```

Fig.5-147: Operator ATAN in IL

Result in q is 0.463648

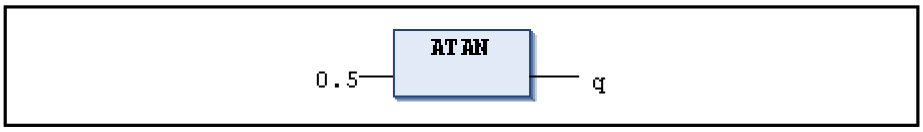


Fig.5-148: Operator ATAN in FBD

Operator ATAN in ST

```
q:=ATAN(0.5);
```

EXPT

Numeric : Exponentiation of one variable with another.

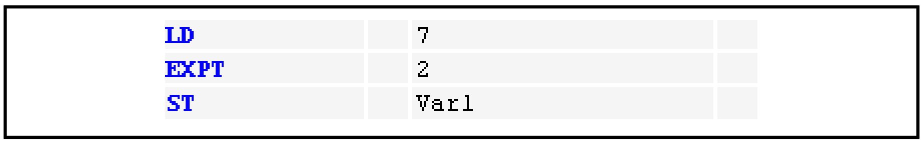
$$OUT := IN1^{IN2}$$

Fig.5-149: Exponentiation of the variable IN1 with IN2

IN1 and IN2 can be any of the following types:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL, LREAL.

OUT has to be of type REAL or LREAL.



```
LD      7
EXPT    2
ST      Var1
```

Fig.5-150: Operator EXPT in IL

Result is 49

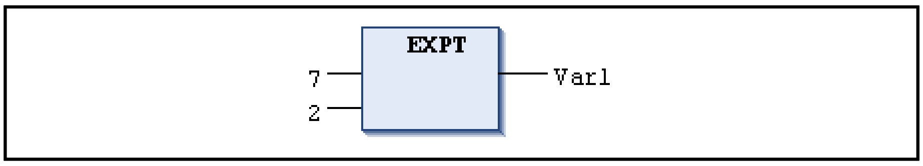


Fig.5-151: Operator EXPT in FBD

Operator EXPT in ST

```
var1 := EXPT(7,2);
```

5.3.11 Namespace Operators

Namespace Operators, Overview

An extension of the IEC 61131-3 standard operators, there are a few ways to design unique access to variables or modules even if the same variable or module name is used several times in the project.

The following namespace operators can be used to define the respectively valid namespace:

- .
- .
- .
- .

Global Namespace Operators

Namespace operator: Extension of the EN 61131-3 standard.

An instance path that begins with a dot "." always opens a global namespace.

So if there is a local variable with the same name as a global variable, "<varname>", ".<varname>" is used to address the global variable.

Namespace for Global Variable Lists

Namespace operator: Extension of the EN 61131-3 standard.

The name of a global variable list (GVL) can be used as a namespace identifier for the variables defined in the list. Thus, variables with the same name in different global variable lists can be used which still have unique access to a specific variable by attaching the name of the list to the front of the variable name separated by a dot "."

Syntax:

```
<global variable list name>.<variable>
```

Example:

The global variable lists **globlist1** and **globlist2** contain a variable **varx** each.

In the following line of code, **varx** is copied from the list **globlist2** to **varx** in the list **globlist1**:

Example:

```
globlist1.varx := globlist2.varx;
```

☞ If a variable that is declared in several global variable lists is referenced without using the list name prefix, an error message is output.

Library Namespace

Namespace operator: Extension of the EN 61131-3 standard.

The library name can be used in order to uniquely address a library function block.

Syntax:

```
<namespace>.<module name>
```

Example:

If a library integrated into a project contains a function block "fun1" and there is also a local function block "fun1" in the project, the library name can be attached to the front of the function block name separated by a dot "." in order to create unique access.

e.g. "lib1.fun1".

By default, the "namespace" of a library has the same name as the library. However, a different identifier can be used for it.

This can already be entered in the project information when the is created or later on in the of an integrated library.

Example:

There is a function **fun1** in the library **lib** and a function **fun1** declared locally in the project. By default, the namespace of the library is called "lib":

Example:

```
  res1 := fun(in := 12); // call of the project function fun
  res2 := lib.fun(in := 12); // call of the library function fun
```

Enumeration Namespace

Namespace operator: Extension of the EN 61131-3 standard.

The TYPE name of an can be used for a unique access to an enumeration constant.

This way, constants with the same name can be used in several enumerations.

The enumeration name is attached to the front of the constant name separated by a dot ".".

Syntax:

```
<enumeration name>.<constant name>
```

Example:

The constant **Blue** is a component of the enumeration **Colors** and the enumeration **Feelings**.

Example:

```
     // Access to enum value Blue
 color := Colors.Blue;     // in type Colors
 feeling := Feelings.Blue;// in type Feelings
```

5.3.12 IEC-Extending Operands

IEC-Extending Operands, Overview

In addition to the operators described in the IEC standard, IndraLogic supports the following operators:

- , releases the memory of instances generated with _NEW dynamic.
- , checks whether a reference refers to a valid value.
- allocates memory for instances of function blocks or arrays of standard data types.
- , enables the type conversion of one interface reference to another at runtime.
- enables the type conversion of an interface reference of a function block to a pointer at runtime.

__DELETE

This operator is not described in the IEC 61131-3 standard.

☞ Due to compatibility reasons, the compiler version has to be >= 3.3.2.0.

The operator __DELETE releases the memory of instances generated with _NEW dynamic.

__DELETE has no return value and after this operation, the operand is reset to 0.

Memory is reserved with __NEW.

Syntax:

```
__DELETE (<Pointer>)
```

If <Pointer> points to a function block, the associated method FB_EXIT is called before the pointer is set to 0.

Example with function block:

```
FUNCTION_BLOCK FBDynamic
VAR_INPUT
   in1, in2 : INT;
END_VAR
VAR_OUTPUT
   out : INT;
END_VAR
VAR
   test1 : INT := 1234;
   _inc : INT := 0;
   _dut : POINTER TO DUT;
   neu : BOOL;
END_VAR
out := in1 + in2;

METHOD FB_Exit : BOOL
VAR_INPUT
   bInCopyCode : BOOL;
END_VAR
__Delete(_dut);

METHOD FB_Init : BOOL
VAR_INPUT
   bInitRetains : BOOL;
   bInCopyCode : BOOL;
END_VAR
_dut := __NEW(DUT);

METHOD INC : INT
VAR_INPUT
END_VAR
_inc := _inc + 1;
INC := _inc;
// ------------------------------------
PLC_PRG(PRG)
VAR
   pFB : POINTER TO FBDynamic;
   bInit: BOOL := TRUE;
   bDelete: BOOL;
   loc : INT;
END_VAR
IF (bInit) THEN
  pFB := __NEW(FBDynamic);
  bInit := FALSE;
END_IF
IF (pFB <> 0) THEN
  pFB^(in1 := 1, in2 := loc, out => loc);
  pFB^.INC();
END_IF
IF (bDelete) THEN
  __DELETE(pFB);
END_IF
```

__ISVALIDREF

This operator is not described in the IEC 61131-3 standard.

The operator "__ISVALIDREF" can be used to check if a refers to a valid value, i.e. a value not equal to 0.

Syntax:

```
<boolean variable>:= __ISVALIDREF
 (with REFERENCE TO <data_type>);
```

<Boolean Variable> becomes TRUE if the reference points to a valid value. Otherwise, it becomes FALSE.

Declaration:

```
 ivar : INT;
 ref_int : REFERENCE TO INT;
 ref_int0: REFERENCE TO INT;
 testref: BOOL := FALSE;
 testref0: BOOL := FALSE;
```

Implementation:

```
 ivar := ivar +1;
 ref_int REF= ivar;
 ref_int0 REF= 0;
 testref := __ISVALIDREF(ref_int);
  // TRUE, because ref_int points to ivar, with value <> 0
 testref0 := __ISVALIDREF(ref_int0);
  // FALSE, because ref_int0 is set to 0
```

__NEW

This operator is not described in the IEC 61131-3 standard.

☞ Due to compatibility reasons, the compiler version has to be >= 3.3.2.0.

__NEW, allocates memory for instances of function blocks or arrays of standard data types. The operator returns a pointer to the correctly typed object which is not equal to 0. If the operator is not used in any assignment, an error message is output.

If the attempt to reserve memory fails, __NEW returns 0.

__DELETE releases the memory again.

Syntax:

```
__NEW (<Type>, [<Size>])
```

The operator generates an instance on the specified type <Type> and returns a pointer to this instance. Then, the initialization of the instance is called.

If <Type> is a scalar type, the optional operand <Size> is set. Then, the operator generates an array of type <Type> and of size <Size>.

Example:

```
pScalarType := __New(ScalarType, length);
```

☞ For dynamically generated instances, copy code methods are not possible in online mode!

Due to this reason, function blocks from libraries that cannot be changed and function blocks with the attribute "enable_dynamic_creation" can be applied to the __NEW operator. If a function block with this identifier is to be changed so that copy code is required, an error message is output.

☞ The code for the memory allocation has to be able to be introduced again.

A semaphore (SysSemEnter) can be used to prevent two tasks from attempting to allocate memory simultaneously. As a result, comprehensive use of __NEW causes greater jitter.

Example with a scalar type:

```
TYPE DUT :
STRUCT
a,b,c,d,e,f : INT;
END_STRUCT
END_TYPE
// ------------------------------------
PLC_PRG (PRG)
VAR
  pDut : POINTER TO DUT;
  bInit: BOOL := TRUE;
  bDelete: BOOL;
END_VAR
IF (bInit) THEN
   pDut := __NEW(DUT);
   bInit := FALSE;
END_IF
IF (bDelete) THEN
   __DELETE(pDut);
END_IF
```

Example with a function block:

```
FBDynamic(FP)
{attribute 'enable_dynamic_creation'}
FUNCTION_BLOCK FBDynamic
VAR_INPUT
   in1, in2 : INT;
END_VAR
VAR_OUTPUT
   out : INT;
END_VAR
VAR
   test1 : INT := 1234;
   _inc : INT := 0;
   _dut : POINTER TO DUT;
   neu : BOOL;
END_VAR
out := in1 + in2;
// ------------------------------------
PLC_PRG(PRG)
VAR
   pFB : POINTER TO FBDynamic;
   loc : INT;
   bInit: BOOL := TRUE;
   bDelete: BOOL;
END_VAR
IF (pFB <> 0) THEN
   pFB^(in1 := 1, in2 := loc, out => loc);
   pFB^.INC();
END_IF
IF (bDelete) THEN
   __DELETE(pFB);
END_IF
```

Example with an array:

```
PLC_PRG(PRG)
VAR
  bInit: BOOL := TRUE;
  bDelete: BOOL;
  pArrayBytes : POINTER TO BYTE;
  pArrayDuts : POINTER TO BYTE;
  test: INT;
  parr : POINTER TO BYTE;
END_VAR
IF (bInit) THEN
```

```
   pArrayBytes := __NEW(BYTE, 25);
   bInit := FALSE;
END_IF
IF (pArrayBytes <> 0) THEN
   pArrayBytes[24] := 125;
   test := pArrayBytes[24];
END_IF
IF (bDelete) THEN
   __DELETE(pArrayBytes);
END_IF
```

__QUERYINTERFACE

This operator is not described in the IEC 61131-3 standard.

__QUERYINTERFACE, enables type conversion of one interface reference to another at runtime. The operator returns a result of type BOOL. TRUE means that the conversion was performed successfully.

Due to compatibility reasons, the definition of the reference to be converted has to be an extension of the basic interface __SYSTEM.IQueryInterface and the compiler version has to be >= 3.3.0.10.

Syntax:

```
 __QUERYINTERFACE(<ITF_Source>, < ITF_Dest>)
```

As first operand, this operator receives an interface reference or a function blocks instance. As second operand, it receives an interface reference with the desired target types. After __QUERYINTERFACE is processed, ITF_Dest contains a reference on the desired interface if the object referenced by ITF_Source implements the interface. Then, the conversion was successful and the result of the operator returns TRUE. In all other cases, FALSE is returned.

The prerequisite for the explicit conversion is that ITF_Source and ITF_Dest are derived from Interface __System.IQueryInterface. This interface is implicitly available and does not require a library.

Example:

```
INTERFACE ItfBase EXTENDS __System.IQueryInterface
METHOD mbase : BOOL
END_METHOD

INTERFACE ItfDerived1 EXTENDS ItfBase
METHOD mderived1 : BOOL
END_METHOD

INTERFACE ItfDerived2 EXTENDS ItfBase
METHOD mderived2 : BOOL
END_METHOD
// ------------------------------------
 POU (PRG)
VAR
   itfderived1 : ItfDerived1;
   itfderived2 : ItfDerived2;
   bTest1, bTest2, xResult1, xResult2: BOOL;
END_VAR
xResult1 := __QUERYINTERFACE(itfbase, itfderived1);
      // variables of type ItfBase resp. ItfDerived1
xResult2 := __QUERYINTERFACE(itfbase, itfderived2);
      // variables of type ItfBase resp. ItfDerived2
IF (xResult1 = TRUE) THEN
   bTest1 := itfderived1.mderived1();
ELSIF xResult2 THEN
   bTest2 := itfderived2.mderived2();
END_IF
```

__QUERYPOINTER

This operator is not described in the IEC 61131-3 standard.

__QUERYPOINTER enables the type conversion of an interface reference of a function block to a pointer at runtime. The operator returns a result of type BOOL. TRUE means that the conversion was performed successfully.

☞ Due to compatibility reasons, the definition of the pointer to be converted has to be an extension of the basic interface __SYSTEM.IQueryInterface and the compiler version has to be >= 3.3.0.10.

Syntax:

```
__QUERYPOINTER (<ITF_Source>, < Pointer_Dest>)
```

As first operand, this operator receives an interface reference or a function block instance with the desired target types. As second operand, it receives a pointer. After __QUERYPOINTER has been processed, Pointer_Dest contains the pointer to the reference or instance of a function block to which the interface ITF_Source currently references. Pointer_Dest is not typed and can be cast to any desired type. The type has to be ensured by the programmer. For example, the interface could offer a method that returns a type code.

The prerequisite for the explicit conversion is that the ITF_Source is derived from Interface __System.IQueryInterface. This interface is implicitly available and does not require a library.

Example:

```
INTERFACE ItfBase EXTENDS __System.IQueryInterface
METHOD mbase : BOOL
END_METHOD

INTERFACE ItfDerived1 EXTENDS ItfBase
METHOD mderived : BOOL
END_METHOD
// ------------------------------------

FUNCTION_BLOCK FBVariante IMPLEMENTS ITFDerived
// ------------------------------------

PROGRAMM POU
VAR
  itfderived : ItfDerived;
  insV : FBVariante;
  xResult, xTest : BOOL;
  pVar: POINTER TO DWORD;
END_VAR
itfderived := insV;
xResult := __QUERYPOINTER(itfderived, pVar);
IF xResult THEN
     xTest := pVar.mderived();
END_IF
```

5.4 Operands

5.4.1 Operands, General Information

The following can be used in IndraLogic 2G as operands:

- *Constants*
  –
  –
  –

- –
- –
- –
- –
- –
- –
- –

- ●
- ●
- ●

5.4.2 Constants

BOOL Constants

BOOL constants are the truth values TRUE and FALSE.

Also refer to

- .

TIME Constants

TIME constants are especially used to operate the default timer module.

In addition to the time constant TIME with a size of 32 bits and which meets the IEC 61131-3 standard, LTIME is also supported as time basis for high resolution timers in the extension of the standard.

LTIME has a size of 64 bits and a resolution within nanoseconds range.

Syntax for a time constant:

```
t#<time declaration>
T#<time declaration>
time#<time declaration>
TIME#<time declaration>

LTIME#<time declaration>
```

Instead of "t#", the following spellings are also valid: "T#", "time#", "TIME#".

The time declaration can include the following time units. They have to be arranged in the following sequence, but it is not required to enter all units.

"d": Days

"h": Hours

"m": Minutes

"s": Seconds

"ms": Milliseconds

"us" : Microseconds (only for LTIME#<time declaration>)

"ns" : Nanoseconds (only for LTIME#<time declaration>)

Examples of correct time constants in an ST assignment:

TIME1 := T#14ms;	
TIME1 := T#100S12ms;	// Overflow is permitted in the highest component
TIME1 := t#12h34m15s;	

Examples of incorrect usage:

TIME1 := t#5m68s;	// Overflow at a low position
TIME1 := 15ms;	// T# missing
TIME1 := t#4ms13d;	// Incorrect time specification sequence

Also refer to

● .

DATE Constants

These constants can be used in order to enter date information.

Syntax:

```
d#<date declaration>
D#<date declaration>
date#<date declaration>
DATE#<date declaration>
```

Instead of "d#", the following spelling is also valid: "D#", "date", "DATE".

The date declaration has to be specified in the "<year-month-day>" format.

DATE (abbreviated D) values are internally treated as DWORD. The time is given in seconds and the calculations begin on January 1st 1970 at 00:00.

Examples

```
DATE#2011-05-06
d#2010-03-29
```

Also refer to

● .

TIME_OF_DAY Constants

Times can be specified with these kinds of constants.

Syntax:

```
tod#<time_of_day declaration>
TOD#<time_of_day declaration>
time_of_day#<time_of_day declaration>
TIME_OF_DAY#<time_of_day declaration>
```

Instead of "tod#", the following spelling is also valid: "TOD#", "time_of_day#", "TIME_OF_DAY#".

The time declaration has to be entered in the format "<hour:minute:second>".

Seconds can be specified as real numbers and fractions of seconds can also be specified.

TIME_OF_DAY (abbreviated TOD) values are internally treated as DWORD. The time is given in milliseconds and the calculations start at 00:00 o'clock.

Examples:

```
TIME_OF_DAY#15:36:30.123
tod#00:00:00
```

Also refer to

● .

DATE_AND_TIME Constants

DATE constants and TIME_OF_DAY constants can be combined to so-called DATE_AND_TIME constants.

Syntax:

```
dt#<Date_and_time declaration>
DT#<Date_and_time declaration>
date_and_time#<Date_and_time declaration>
DATE_AND_TIME#<Date_and_time declaration>
```

Instead of "dt#", the following spelling is also valid: "DT#", "date_and_time#", "DATE_AND_TIME#".

The date and time declaration have to be entered in the format "<year-month-day-hour:minute:second>".

Seconds can be specified as real numbers and fractions of seconds can also be specified.

DATE_AND_TIME (abbreviated DT) values are treated as DWORD internally. The time is given in seconds and the calculations begin on January 1st 1970 at 00:00.

Examples:

```
DATE_AND_TIME#2011-05-06-15:36:30
dt#2010-03-29-00:00:00
```

Also refer to

- .

Number Constants

Number values can appear as binary numbers, octal numbers, decimal numbers and hexadecimal numbers.

If an integer value is not a decimal number, its base has to be written in front of the integer constant followed by a hash (#).

In hexadecimal numbers, the numeric values for the numbers 10 to 15 are usually specified by the letters A-F.

Underscores are allowed within a number value.

14	(Decimal number)
2#1001_0011	(Binary number)
8#67	(Octal number)
16#A	(Hexadecimal number)

The type of these number values can be one of the following:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL or LREAL.

Implicit conversions of larger types to smaller types are not allowed, i.e. a DINT variable cannot be used as an INT variable. Use type conversions.

For bit character strings, displaying hexadecimal numbers has to begin with leading zeroes.

BYTE: 16#00 ... 16#FF

WORD: 16#0000 ... 16#FFFF etc.

REAL/LREAL Constants

REAL and LREAL constants can be specified as decimal fractions and with exponents.

To do this, use the American format with a dot.

7.4	instead of 7,4
1.64e+009	instead $1.64e^{+009}$

STRING Constants

A is any character string in ASCII code (8 bit).

STRING constants are enclosed by simple apostrophes before and after.

Spaces and umlauts can also be entered. They are treated as any other characters.

In character strings, the combination of the dollar sign ($) followed by two hexadecimal numbers is interpreted as the hexadecimal display of the eight bit character code. In addition, combinations of characters that start with a dollar sign are interpreted as follows:

Specified combination	**Interpretation**
'$<two hex. numbers>'	Hexadecimal display of the eight bit character code
' '	Space
'$$'	Dollar sign
'$''	Simple apostrophe
'$L' or '$l'	Line feed
'$N' or '$n'	New line
'$P' or '$p'	Page feed
'$R' or '$r'	Line break
'$T' or '$t'	Tab

Example:

Valid character strings:

'w1Wüß?'

'Abby and Craig'

':-)'

'costs ($$)'

WSTRING Constants

A is any character string in Unicode.

WSTRING constants are enclosed by quotation marks before and after.

Spaces and umlauts can also be entered. They are treated as any other characters.

In character strings, the combination of the dollar sign ($) followed by four hexadecimal numbers is interpreted as the hexadecimal display of the 16 bit character code. In addition, combinations of characters that start with a dollar sign are interpreted as follows:

Specified combination	Interpretation
'$<four hex. numbers>"	Hexadecimal display of the 16 bit character code
" "	Space
"$$"	Dollar sign
"$""	Simple quotation marks
"$L" or "$l"	Line feed
"$N" or "$n"	New line
"$P" or "$p"	Page feed
"$R" or "$r"	Line break
"$T" or "$t"	Tab

Example:

Valid character strings:

"w1Wüß?"

"Abby and Craig"

":-)"

"costs ($$)"

Typed Constants (Typed Literals)

Except REAL/LREAL constants (LREAL is always used here), the smallest possible data type is used when calculating with IEC constants. If a different data type is to be used, typed literals (typed constants) can be used without having to declare the constant explicitly. The constants are provided with a prefix that specifies the type:

Syntax:

```
<Type>#<Literal>
```

<Type> indicates the desired data type. Possible specifications are:

- BYTE, WORD, DWORD, LWORD,
- SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT,
- REAL or LREAL.

The type has to be written in upper case letters.

<Literal> indicates the constant. The input has to match the data types specified under <Type>.

Example:

```
var1:=DINT#34;
```

If the constants cannot be transferred to the target type without loss of data, an error message is output.

Typed constants can be used anywhere normal constants can be used.

5.4.3 Variables

Variables, General Information

Variables are either declared locally in the declaration part of a function block or in the global variable lists.

The available variables can be called using the .

Further information

- on the and
- on the

Access to Variables of Arrays, Structures and Function Blocks

The following access methods are possible:

Two-dimensional array components:

```
<ArrayName>[Index1, Index2]
```

Structure variables:

```
<StructureName>.<VariableName>
```

Function block and program variables:

```
<FunctionBlockInstanceName>.<VariableName>
```

Further information

- on
- on
- on and .

Bit Access to Variables

In integer variables, individual bits can be addressed. To do this, the index of the bit to be addressed is attached to the variable separated by a dot. The bit index can be indicated by any desired constant. The indexing is based on 0.

☞ Bit accesses used in visualizations that are executed using a only work if they contain literal bit index information (i.e. no defined constants).

Syntax:

```
<Variablename>.<Bitindex>
```

Example:

```
VAR
 a : INT;
 b : BOOL;
END_VAR

a.2 := b;
```

The **third** bit of the variable a is set to the value of variable b.

If the index is greater than the bit width of the variable, the following error is output:

Index <n> outside of the valid range for variable <var>!

Bit addressing is possible for the following variable types: SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, BYTE, WORD, DWORD, LWORD.

If the type of variable is not permitted, the following error message is output:

Illegal data type <Type> for direct indexing.

A bit access may not be assigned to a VAR_IN_OUT variable!

Bit access with a global constant:

If a global constant is declared that defines the bit number, this constant can be used for the bit access.

The variable **enable** defines which bit is to be accessed:

Declaration in a global variable list for both examples:

```
VAR_GLOBAL CONSTANT
  enable:INT:=2;
END_VAR
```

Example 1, bit access to an integer variable:

```
// Function block declaration
VAR
    xxx:INT;
END_VAR

// Bit access
xxx.enable:=true;
```

This sets the third bit in the variable xxx to TRUE.

Example 2, bit access to integer structure components:

```
// DUT declaration of stru1
TYPE stru1 :
    STRUCT
        bvar:BOOL;
        rvar:REAL;
        wvar:WORD;
        {bitaccess enable 42 'Start drive'}
    END_STRUCT
END_TYPE
// FB declaration
VAR
    x:stru1;
END_VAR

// Bit access
 x.enable:=true;
```

This sets the 42nd bit in the variable x to TRUE.

Since bvar contains 8 bits and rvar 32 bits, this access is on the second bit in the variable wvar which receives the a value of 4.

5.4.4 Addresses

Notes on addresses

☞ If the online change is used, the contents of addresses can be shifted. Remember this when using to !

Address

Individual memory cells are displayed directly using special character strings.

Syntax:

```
%<memory area prefix><size prefix><number|.number|.number...>
```

The following range prefixes are supported:

I	Input (input, physical inputs via input drivers, "sensors")
Q	Output (output, physical outputs via output driver, "actuators")
M	Flag memory space

The following prefixes are supported for the sizes:

X	Single bit
None	Single bit
B	Byte (8 bits)
W	Word (16 bits) [word (32 bits)]
D	Double word (32 bits)
L	Long word (64 bits)

Examples

%QX7.5 and %Q7.5	Output bit 7.5
%IW215	Input word 215
%QB7	Output byte 7
%MD48	Double word at memory location 48 in flag area
ivar AT %IW0 : WORD;	Example of a variable declaration with address specification.

Also refer to

- .

Ensure that the address is valid:

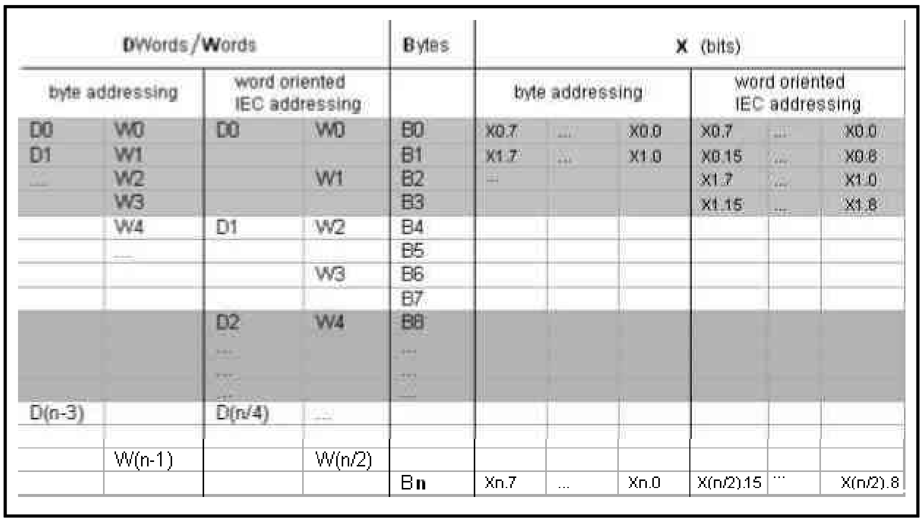
in an application, the requested position in the process image has to be known, that is the responsible **memory space**: Input (I), output (Q) or flag memory spaces (M), see above. The required **size** has to be specified as well: bit, byte, Word, DWord (see above: X, B, W, D)

Decisive is the **currently used device configuration** and its settings (hardware structure, device description, I/O settings). Note that there are differences in the address interpretation of devices with "byte addressing mode" and devices with word-oriented "IEC addressing mode".

Depending on the size and the addressing mode, different memory cells can be addressed with the same address specification.

The following table shows a comparison of **byte addressing** and **word-oriented IEC addressing** for bit, BYTE, WORD and DWORD. It also shows overlapping memory spaces that are also present for byte addressing (also refer to the example below the table).

With regard to the spelling, note that the IEC addressing mode is always word-oriented. That means that the word number is in front of the point and then the bit number.



DWords / Words				Bytes	X (bits)					
byte addressing		word oriented IEC addressing			byte addressing			word oriented IEC addressing		
D0	W0	D0	W0	B0	X0.7	...	X0.0	X0.7	...	X0.0
D1	W1			B1	X1.7	...	X1.0	X0.15	...	X0.8
...	W2		W1	B2	...			X1.7	...	X1.0
	W3			B3				X1.15	...	X1.8
	W4	D1	W2	B4						
	...			B5						
			W3	B6						
				B7						
		D2	W4	B8						
		...		...						
		...		...						
		...		...						
D(n-3)		D(n/4)	...							
	W(n-1)		W(n/2)							
				Bn	Xn.7	...	Xn.0	X(n/2).15	...	X(n/2).8

Fig.5-152: Comparison of byte-oriented or word-oriented addressing of the address formats D, W, B and X

n = byte number; word-oriented, addressing

Memory space overlapping in byte addressing mode, example:

D0 contains B0 - B3, W0 contains B0 and B1, W1 contains B1 and B2, W2 contains B2 and B3 ->

To avoid overlapping, W1 or D1, D2, D3 may not be used as addressing!

Boolean values are reserved byte by byte if a single bit address is not explicitly specified.

Example: A value change of varbool1 AT %QW0 affects the range from QX0.0 to QX0.7.

If the online change is used, the contents of addresses can be shifted. Remember this when using to !

5.4.5 Functions

Functions

In ST, a can also occur as operand.

Example:

```
Result := Fct(7) + 3;
```

TIME() function

This function provides the time in milliseconds passed since the system start.

The data type is TIME.

Example in IL:

```
  TIME
  ST     systime
```

Example in ST:

```
   systime:=TIME();
```

5.5 Visualization

5.5.1 Visualization, General Information

Depending on the application, IndraWorks provides two visualization tools implemented at different levels of performance:

- **WinStudio**; This tool should only be used for the visualization of applications that run on visualization devices (BTV / Vxx).

  See also "Rexroth WinStudio; Brief Description, DOK-CONTRL-WIS*PC**V06-KB..-EN-P".

  The Symbol configuration, page 306 object is used to transfer the data to be visualized.
- **Visualization onboard**; This tool described below should be used to support the commissioning of own applications.

5.5.2 Visualization in IndraLogic 2G

All visualizations in IndraLogic 2G have the following characteristics:

- Within a visualization, any desired expressions, even function calls, are permitted.
- A visualization can be created as object below an application (or in the "General module" folder).
- In contrast to IndraLogic 2.x, all of the visualization elements as well as the visualization objects themselves are implemented as IEC61131-3 function blocks. This way, visualizations can be instanced in other visualizations, i.e. they can be referenced and used to extend other function blocks.
- It includes a placeholder concept that treats a placeholder as an input variable for a visualization, see

  .
- Interfaces can be edited in an interface editor. The use of frame elements makes it easy to switch from a display of one visualization to another one within a visualization object.
- Visualizations in the Project Explorer are managed by a

  for each application. The manager defines common settings for all application-specific visualizations.
- Some general settings for all visualizations in a project (file paths, size adjustment, etc.) can be made in the .
- Each individual visualization has properties such as its planned usage (as "visualization", "Numpad/Keypad" or "dialog") or the display size.

  In this context, ensure that a visualization that can be created explicitly to be used as user input dialog in other visualizations. As an implicit option, there is also a prefabricated numpad and keypad mask for this purpose. The use of such numpad/keypad templates and dialogs can be defined in the configuration of a visualization element.
- which provides the available visualization elements and a properties editor to configure the added elements.

  It is easy to arrange and group the visualization elements. The standard elements are provided in the corresponding visualization libraries in the project, see . IndraLogic 1.x visualizations can be imported.

- using IndraLogic project variables that are entered directly or using expressions, i.e. combinations of the variables with operators and constants. This allows variable values to be scaled for example so that the variable values can be used in the visualization.
- Language selection (ANSI or UNICODE) in a visualization is possible by using , see .
- Variable values of an application can be modified and displayed using a . The formatting for these inputs and output is based on the standard function sprintf (C library).
- can be used.
- A can be assigned to each visualization element.
- In addition to the zoom function, automatic is also possible (visualization adjusts to the screen size).
- Visualizations can be saved in and thus provided for other projects.
- Visualization element repositories (### in preparation ###) are used to manage visualization elements on the local system.

☞ The visualization works with an integrated runtime system. Thus, start and download messages appear when work is being done in the editor.

5.5.3 Prerequisites

A project with visualization requires the same basic structure as a project without visualization. A device with the application and task configuration as well as an application program have to be present in the project tree (Project Explorer).

The IndraLogic 2G visualization is realized according to the IEC 61131-3 standard. Thus, certain have to be integrated into the project.

To select the visualization libraries and to exactly define the selection of elements to be provided in the visualization editor, are used. Thus, each visualization project has to be based on such as profile.

5.5.4 Options

In the "Options" dialog, standard entries for the project settings can be made in the 'Visualization' category such as visualization directories for language and image files that have to be available to configure visualizations.

Therefore, click on **Tools** ▸ **Options** ▸ **IndraLogic 2G** ▸ **Visualizations** in the main menu to specify the visualization options.

For more information, refer to: .

5.5.5 Creating a Visualization Object

A visualization can be added to the "General module" folder (global object pool) or below an "Application" depending on whether the visualization is to be used globally or only for a specific application.

To create a visualization object in the project, use **Add** ▸ **Visualization...** in the context menu.

Alternatively, the visualization object is available in the "PLC objects" library in the "VI logic objects" folder and can be added to the project via drag&drop.

As soon as the first visualization object is added to the project, the libraries defined in the currently active visualization profile are automatically integrated into the Library Manager.

As soon as the first visualization is added below an application, the is also automatically added.

The newly created visualization object is automatically opened in the visualization editor

5.5.6 Editing a Visualization

The **visualization editor** to create visualizations works together with the toolbox which provides the visualization elements from the integrated element libraries and with the **properties editor** to configure visualization elements.

If necessary, call both editors manually in the main menu under **View ▸ Other windows**.

To open a visualization object, double-click on the visualization object in the Project Explorer.

☞ See also the following chapters on editing visualizations:

-
- (available visualization elements)
- (configuring the elements)

5.5.7 Visualization Manager

When a visualization is added below an application in the Project Explorer, a visualization manager object is also automatically added. The "Visualization Manager" defines common settings for all visualizations in the application, e.g. start visualization.

For more information, refer to: .

5.5.8 Start Visualization

The "start visualization", i.e. the visualization object to be displayed first after logging into the control with the application, has to be added below the respective application object in the Project Explorer.

If only one visualization is assigned to the application, it is automatically used as start visualization.

If several visualizations are assigned to the application, the start visualization . Afterwards, this option is automatically disabled in all other visualization objects.

☞ The start visualizations below an application switches to online mode with the application.

Since visualizations in the POUs do not have a direct reference to an application, these do not switch to the online mode.

If a change in visualization to a visualization from the POUs is configured in a visualization below the application, it can thus be reached in online mode.

5.5.9 Frames, References, Interfaces, Placeholders

The concept of visualization references and placeholders in IndraLogic 1.x is replaced by a similar concept in IndraLogic 2G.

- In principle, a visualization can be added to another visualization and thus referenced. The "Frame" element which can include one or more "visualization references" is used for this purpose.

  The visualizations available depend on their positions relative to the visualization just edited in the Project Explorer.

- Each visualization is treated as a function block and has an "interface" in which the input variables for a function block can be defined ( ).

  These input parameters function as "placeholders".

  In an instance of the visualization, they have to be replaced by values or expressions for the specific usage in the local object. The replacement has to be be carried out in the "Property" dialog of the "Frame" element that integrates the visualization instances. Note that the input variables of a visualization instance allocated to a specific application have to be assigned to valid variables for this application.

- **Switching visualizations:**

  If a "frame" element includes several visualization references ( ), user inputs for another visualization element can be configured so that it causes the display of these references to switch in the frame.

  To do this, the input configuration provides the "Switch visualization" option.

Example:

A visualization contains three buttons and a frame to which the visualizations Visu1, Visu2 and Visu3 are assigned.

The buttons are configured (input, OnMouse actions, visualization switch) so that a user input to a button calls the respective visualization in the frame.

Thus - in contrast to IndraLogic 1.x - different visualizations can be displayed in alternation in one visualization. In other words, a switchboard can be visualized.

- In addition, note the following: A visualization is treated as a function block and can be used to extend another function block, see .

5.5.10 Text and Language in Visualizations

A text can be assigned to a visualization element in the respective fields of the "Properties" editor.

"Text lists" can be used to manage texts. A text list is a POU that contains text strings that can be referenced uniquely within a project by combining a text list name with a list-internal text ID. Different language versions of a text (local country language) can be defined in a text list. Thus, each text is also identified by a language code. At least the default language has to be defined for every text.

Detailed information on the usage of text lists can be found in .

☞ Directories that provide text lists for usage in visualizations can be specified in the visualization options. Click on **Tools** ▸ **Options** ▸ **IndraLogic 2G** ▸ **Visualizations** in the main menu to enter visualization options, see .

Enter the text list name and text ID (string format) in the text properties of a visualization element in order to define which text is displayed in the element in online mode. In addition, use formatting sequences to affect how the text is displayed, see . To enable **dynamic switching among texts** a variable to specify the text ID has to be used as well.

The language set in IndraLogic determines the local country language version of the text used. A **language selection** in the visualization can also be forced during online mode if a corresponding input configuration is present for a visualization element.

For example, users can then switch to another language via mouse click (configuration in the "Input" category in the "Properties" editor), see .

This allows dynamic display and language selection for visualization texts.

There are two types of text in a visualization:

1. **Static texts:**

   Static text cannot be changed in online mode. It is configured in the "Texts" category in the visualization element properties and is managed in an automatically generated text list "GlobalTextList".

2. **Dynamic texts:**

   Dynamic text can be modified in online mode by the user input or by a variable of the application. It is configured in the "Dynamic texts" category in the visualization element properties. The name of the dynamic text lists and the text ID for the desired text have to be specified.

Formatting text

In addition to entering the pure text in the element configuration, formatting specifications can also be edited to format the text display in online mode. A formatting specification always begins with a "%" followed by a character that determines the type of formatting. Use formatting by itself or in combination with the actual text.

If a **%s** is included in the text specification, in online mode, it is replaced by the value of the variable specified in the "Text variable" properties field in the "Text variables" category.

To display the instance name of a variable that was transferred to the visualization function block as input parameter, use the .

Note that any formatting specification in accordance with the standard C library function "sprintf" can be used if it matches the data type of the variable used.

The following table contains some examples of formatting specifications.

Characters following "%"	Argument/output as
d,i	Decimal number
o	Unsigned octal number (without a preceding zero)
x	Unsigned hexadecimal number (without a preceding "0x")
u	Unsigned decimal number

Characters following "%"	Argument/output as
c	Single character
s	Character string: In online mode, this is replaced by the value of the variables specified in the "Text variable" properties field in the "Text variables" category.
f	REAL values **Syntax:** `%|<Alignment><LowestWidth><Accuracy>|f` Alignment is defined by a minus sign (left-justified) or plus sign (right-justified, default). Accuracy is defined by the number of places after the decimal (preset value: 6). See the following example.

Fig.5-153: Formatting specifications

Example:

Entry in the "Text" properties field: Fill level %2.5f mm

Entry in the "Text variable" field (REAL variable to specify the fill level), e.g.: Plc_Main.fvar1

⇒ Output in online mode, e.g.

```
Fill level 32.8999 mm
```

If a "percent sign %" is to be displayed together with one of the previously described formatting specifications, "%%" has to be entered.

Example:

Enter "Rate in %%: %s" to display the following in the online mode:

```
Rate in %: 12
```

(if the value of the text variable is currently "12").

To process all character sequences in the visualization in Unicode format, select the option in the Visualization Manager of your application.

System time output

A combination of "%t" and the following special characters in square brackets is replaced by the actual system time in online mode. The placeholders define the display format, refer to the table below.

Import of IndraLogic 1.x projects:

The time format %t used in IndraLogic 1.x is automatically converted into the new %t[] format when an old project is imported. However, the following placeholders are no longer supported: %U, %W, %z, %Z.

Valid placeholders:

ddd	Name of weekday, abbreviated, e.g. "Wed"
dddd	Name of weekday, e.g. "Wednesday"
ddddd	Weekday as number (0 - 6; Sunday is 0)
MMM	Name of month, abbreviated, e.g. "Feb"
MMMM	Name of month, e.g. "February"
d	Day in month as number (1 – 31), e.g. "8"
dd	Day in month as number (01 – 31), e.g. "08"
M	Month as number (1 – 12), e.g. "4"
MM	Month as number (01 – 12), e.g. "04"
yyy	Day in year as number (01 – -366), e.g. "067"
y	Year without century indication (0-99), e.g. "9"
yy	Year without century indication (00-99), e.g. "09"
yyy	Year with century indication, e.g. "2009"
HH	Hour, 24 hour format (01-24), e.g. "16"
hh	Hour, 12 hour format (01-12), e.g. "8" for 4 PM
m	Minutes (0-59), without preceding zero, e.g. "6"
mm	Minutes (00-59), without preceding zero, e.g. "06"
s	Seconds (0-59), without preceding zero, e.g. "6"
ss	Seconds (00-59), without preceding zero, e.g. "06"
ms	Milliseconds (0-999), without preceding zero, e.g. "322"
t	Identifier for the display in 12 hour format: A (hours <12) or P (hours >12), e.g. "A" if it is 9 a.m.
tt	Identifier for the display in 12 hour format: AM (hours <12) or PM (hours >12), e.g. "AM" if it is 9 o'clock in the morning
' '	Character strings that contain one of the placeholders listed above have to be enclosed in single apostrophes. All other texts in the format string can remain without apostrophes: e.g. "update", since it contains a "d" and a "t"

Fig.5-154: Valid placeholders in the new format

Example:

%t['Last update:' ddd MMM dd.MM.yy 'at' HH:mm:ss] ⇒

Display in online mode:

Last update: Wed Aug 28.08.02 at 16:32:45

Font The font used in online operation can also be defined in the visualization element properties. See the categories 'Text properties' and 'Font variables'.

Alignment The horizontal and vertical alignment of the element text can be defined in the element properties in the "Text properties" category.

☞ When configuring visualization elements (e.g. text fields, tooltips, alignment, font), also refer to the description in .

5.5.11 Images in a Visualization

The "Image" element can be used to add an image from an external image file to a visualization.

☞ Directories that provide image files for usage in visualizations can be specified in the visualization options. Click on **Tools** ▸ **Options** ▸ **IndraLogic 2G** ▸ **Visualizations** in the main menu to enter visualization options; see .

The image files are managed in the project in .

Individual images can be uniquely referenced by a combination of image pool name and ID specified for each image file in the pool.

This ID and (for unique access) the image pool name can be specified in the element properties of an image element in order to determine the image displayed in online mode. The image reference can also be specified using a variable, which enables **dynamic switching among images**.

In addition, a background image for a visualization can be defined in the main menu with **VI logic visualization** ▸ **Background.**

Alternatively, **Background** can also be selected via the respective context menu.

☞ When configuring visualization elements (e.g. text fields, tooltips, alignment, font), also refer to the description in the .

5.5.12 External Data Sources

External (remote) data sources can also be used in visualizations in order to set up a point to point connection between an application (on the "local" device) and a remote data source.

For this purpose, create a data server for the local application which manages the related data sources; see .

5.5.13 Visualization Libraries

The libraries necessary for creating and executing a visualization are automatically added to the library manager in the global "General module" folder as soon as a visualization is created in the project.

Since the visualization elements in IndraLogic 2G are created as function blocks, the basic elements are provided by libraries.

If a visualization object is added to the project, certain visualization libraries are integrated into the Library Manager. The names and versions of these libraries are defined in the visualization profile currently used. The profile also specifies exactly which elements from these libraries are available in the visualization editor toolbox.

A visualization library is always created as a special type of a "placeholder library". The result is that the exact version of the library to be used is not specified as long as it is not integrated into a project. The current visualization profile specifies which version is actually needed. Note that this library type differs from the device-specific placeholder libraries at which the placeholders are resolved from the device description.

See the following basic libraries that are integrated by default as soon as a visualization element is added to a standard project. They reference other libraries that are not listed here:

- **VisuElems.library**, (basic set of visualization elements)
- **VisuElemMeter.library**, (meter and bar display elements)
- **VisuElemWinControls.library**, (table, text field, scrollbar elements)
- **VisuElemTrace.library**, (trace element)
- **VisuInputs.library**, (handling inputs on a visualization)

5.5.14 Visualization Profiles

Each visualization project, that is a project that contains at least one visualization object, has to be based on a visualization profile. This profile defines the following:

- that are automatically integrated into the project as soon as a visualization object is .
- a selection of the visualization elements from the integrated libraries provided in the

The **profile configuration** is completed outside the project in the visualization element repository (### in preparation ###).

Several profiles can be defined and stored on the local system.

The **profile that is used by default in the project** is defined in the project settings (visualization profile).

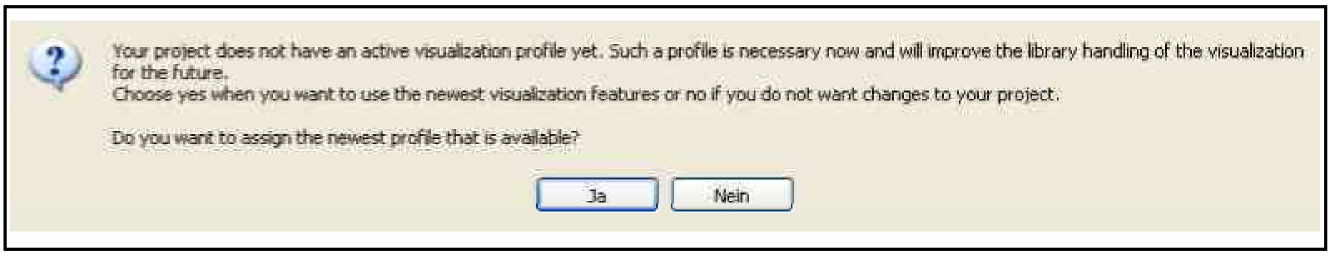
Switching to another profile is possible at any time. Note that the selected profile applies to all devices and applications.

In case of a profile switch, a message appears that indicates that the switch could prevent logging in without an online change or download.

Switching profiles causes an automatic update in the library manager with respect to the required libraries. These do not have to be adjusted manually.

A visualization profile has to define at least one **"main" visualization library**, i.e. a library that defines the replacements for the placeholders used in other visualization libraries. In a standard installation, this library is **VisuElems.library**.

When opening an **old project** that was not created with a visualization profile, you are prompted whether you want to assign the newest profile available.


Your project does not have an active visualization profile yet. Such a profile is necessary now and will improve the library handling of the visualization for the future.
Choose yes when you want to use the newest visualization features or no if you do not want changes to your project.

Do you want to assign the newest profile that is available?

Fig.5-155: Message box if there is no profile

If "Yes", the "Newest profile" is used. If "No", the oldest available profile is entered in the project settings (it is called "**Compatibility profile**"), but no further changes are made in the project.

5.5.15 Visualization Running in the Programming System

For diagnostic purposes, it can be desirable to let one of the visualization(s) belonging to an application only run in the programming system without having to load visualization code to the control.

This"**Diagnostic visualization**" available **from V3.3.2.0** is **automatically** used if no client object "TargetVisualization" is attached below the Visualization Manager in the Project Explorer. Then, no **visualization code** is generated and loaded to the control. However, a few limitations result which are listed in the following.

If you do not want to remove the TargetVisualization objects below the Visualization Manager to obtain the diagnostic mode of the visualization, exclude the client objects from compilation by selecting the setting "Exclude from compilation" in the respective object properties.

☞ If a fixed compiler version was set in a project that was originally created with a version < V3.3.2, the project acts according to the possibilities available in that version!

For versions lower than V3.3.2.0, the setting "Do not use the PLC for visualization" has to be selected so that a visualization only runs in the programming system.

Numerical values displayed in an element in a diagnostic visualization ("%s" in the "Text" property) are displayed in accordance with the currently set display format (binary, decimal, hexadecimal).

☞ In the diagnostic visualization, the VAR_INPUT variables act like VAR_IN_OUT variables.

Restrictions

Expressions, monitoring

The diagnostic visualization supports only those expressions that can be handled by the monitoring mechanism of the programming system. These are:

- **Normal variable accesses such as PlcProg.myPou.nCounter**
- *Complex accesses as listed in the following. Note that a runtime system of V3.3 SP2 or later is required:*
  - Access to an array of scalar data types in which one variable is used as index (a[i])
  - Access to an array of complex data types (structures, function blocks, arrays) in which one variable is used as index (a[i].x)
  - Access to a multidimensional array by all types of data types with one or more variable indices (a[i, 1, j].x)
  - Access to an array with a constant index (a[3])
  - Accesses as described above in which simple operators are used for the calculations within the index brackets (a[i+3])
  - Nested combinations of the complex expressions listed above (a[i + 4 * j].aInner[j * 3].x)
- Operators supported in index calculations: +, -, *, /, MOD
- Pointer monitoring such as p^.x
- Methods or function calls are not supported except for the following: All standard text functions, all type conversion functions such as INT_TO_DWORD, all operators such as SEL, MIN, ...

Inputs:

Within the input action "Execute ST code" only a list of assignments is supported.

Example:

```
PlcProg.n := 20 * PlcProg.m;
```

Not permitted:

```
  IF PlcProg.n < MAX_COUNT THEN
     PlcProg.n := PlcProg.n + 1;
  END_IF
```

Use the following instead:

```
PlcProg.n := MIN(MAX_COUNT, PlcProg.n + 1);
```

If a list of assignments is used, the value on the left side is assigned in the next cycle. Immediate processing in the next line is not possible.

Visualization interface:

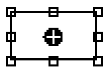
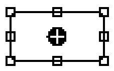
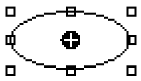
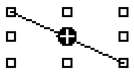
Within the interface definition of a visualization, the type "interface" ("INTERFACE") may not be used.

DataServer:

The diagnostic visualization (visualization runs in the programming system) is only for observation purposes of the current application. It does not display the current values of variables transmitted from another data source via the data server. Only the initial values or the most recently transferred values are displayed.

5.5.16 Visualization Elements of Type "Visualization"

The category "Visualization" of the "ToolBox" tab in the visualization editor contains the following elements:

Rectangle		Rectangle, rounded rectangle and ellipse are the same type of element. Each shape can be converted into the other by changing the type property.
Rounded Rectangle		
Ellipse		
Line		Line: Adjust the slope of a "line" element between "upper left - lower right" and "lower left - upper right".

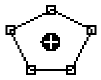
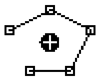
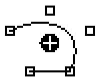
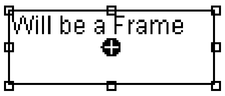
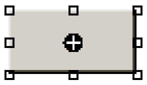
Polygon		Polygon, polyline and curve are the same type of element. Each shape can be converted into the other by changing the type property. An additional position point that specifies the segments and the shape of an element can be inserted by clicking on an existing point (small rectangle on the frame line) while holding down the <Ctrl> key. A point can be deleted by clicking on the point while pressing and holding down the <Shift> + <Ctrl> keys. In a curve element, the points next to the line are used as "grips" to modify the curve shape.
Polyline		
Curve		
Frame		Frame: A frame defines a section of the current visualization to which one or several other visualizations are assigned of which one is displayed in online mode. In principle a frame can be configured as a rectangle element. With the correct configuration, a user input can be used to switch among the visualizations displayed. With the correct configuration, a user input can be used to switch among the visualizations displayed. Assign visualizations to a frame using the "Configuration of frame visualizations" which opens via the command). All available visualizations are provided for selection. For information about configuring a visualization switch by user inputs, refer to the possible .
Button		An image and a "height" (for the relief view) can be assigned to a button element.
Image		An image element is filled by an image file defined by an ID and name of the in which it is managed. The file specification can also be dynamically configured, i.e. via a project variable.

Fig.5-156: Standard elements in the VisuElems.library

☞ When configuring visualization elements (e.g. text fields, tooltips, alignment, font), also refer to the description in .

5.5.17 Visualization Elements of Type "Complex Controls Visualization"

Elements overview

The category "Complex Controls" in the in the visualization editor contains the following elements.

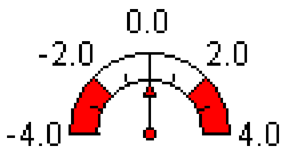
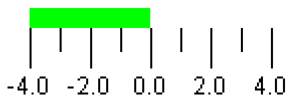
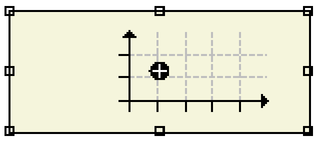
Element name	Element	Description
meter		The "meter" element inserts a tachometer into the visualization for which a minimum and maximum display value can be entered. The needle position indicates the current value of the associated input variable. Specific background colors can be defined for certain value ranges.
Bar display		The "bar display" element inserts a bar graph into the visualization for which a minimum and maximum display value can be entered. The length of a the bar indicates the current value of the associated input variable. Specific colors can be defined for certain value ranges. A horizontal bar display is preset. After the element has been inserted, the orientation can be changed to vertical in the element properties.
Trace		element allows a trace to be integrated into a visualization. The name of the trace to be displayed is input in the element properties. However, the recorded variables are configured in the dialog.

Fig.5-157: Elements in the "Complex Controls" category

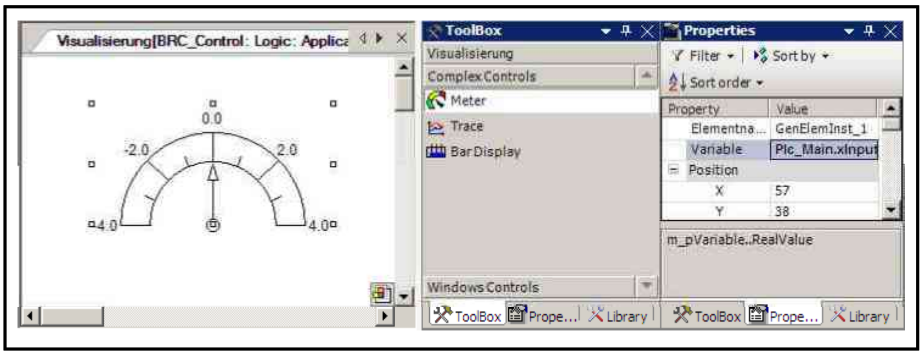
An explanation using examples can be found under:

-
-
-

Configuring the Meter Element

To configure the "meter" element, carry out the following steps:

1. The "meter" visualization element looks like a tachometer and allows the current value of the associated input variable to be displayed. Insert the "meter" visualization element into a visualization by dragging it from the and dropping it in the visualization.



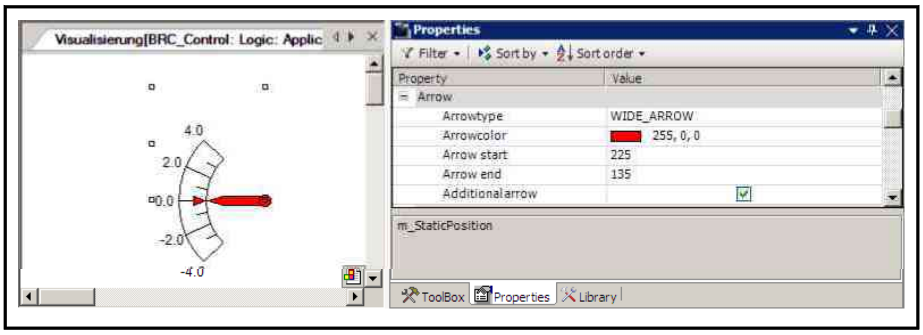
Property	Value
Elementna...	GenElemInst_1
Variable	Plc_Main.xInput
Position	
X	57
Y	38

m_pVariable..RealValue

Fig.5-158: Inserting the meter element and link to the input variables

In the element properties for the "meter" element, enter the associated input variable (e. g. "xInput") into the "Value" column. After clicking into the input field, the ... button is available. It can be used to search through the project for the input variable. Ensure to specify the input variable by entering its complete path in the Project Explorer.

2. The orientation and size or the scale display as well as the color and appearance of the arrow can be specified in the "Arrow" section in the element properties.



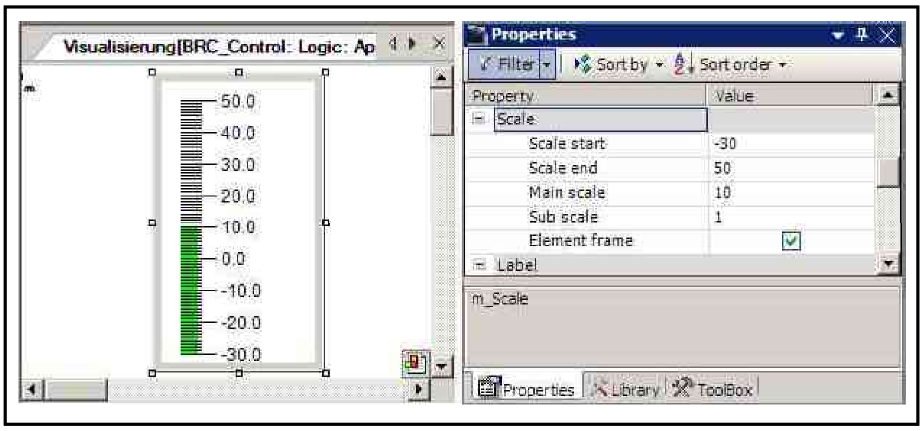
Property	Value
Arrow	
Arrowtype	WIDE_ARROW
Arrowcolor	🟥 255, 0, 0
Arrow start	225
Arrow end	135
Additionalarrow	✓

m_StaticPosition

Fig.5-159: Defining the appearance of the scale display and the arrow

The number in the "Arrow start" and "Arrow end" input field specifies the angle (in degrees) spanned by the left/right margins of the scale. This angle is mathematically oriented, i.e. counterclockwise. Thus, the value in the "Arrow start" field always has to be greater than the value in "Arrow end"! The pair of starting and end value is periodic with 360 degrees.

3. In the "Scale" section, specify the numerical range for the scale as well as rough and precision scaling.



Property	Value
Scale	
Scale start	-30
Scale end	50
Main scale	10
Sub scale	1
Element frame	✓
Label	

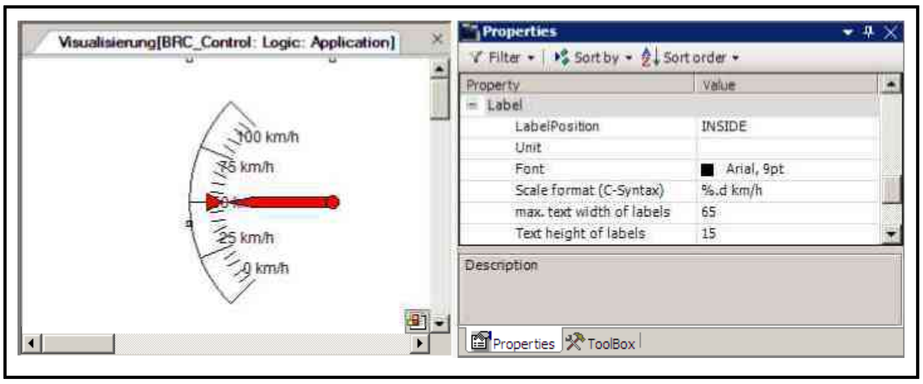
m_Scale

Fig.5-160: Configuration within the "Scale" section in the element properties

Enter the starting point for the scale in "Scale start". The "Scale start" value has to be lower than the "Scale end" value inserted at the end of the scale.

In contrast to rough scaling, precision scaling ("Sub scale") can be omitted by setting the value for the spacing to 0. In this case, precision scaling lines are not displayed. In the example, the "Frame inside" checkbox is deselected so that the inner arc of the scale is hidden.

4. Format the scale label in the "Label" section.



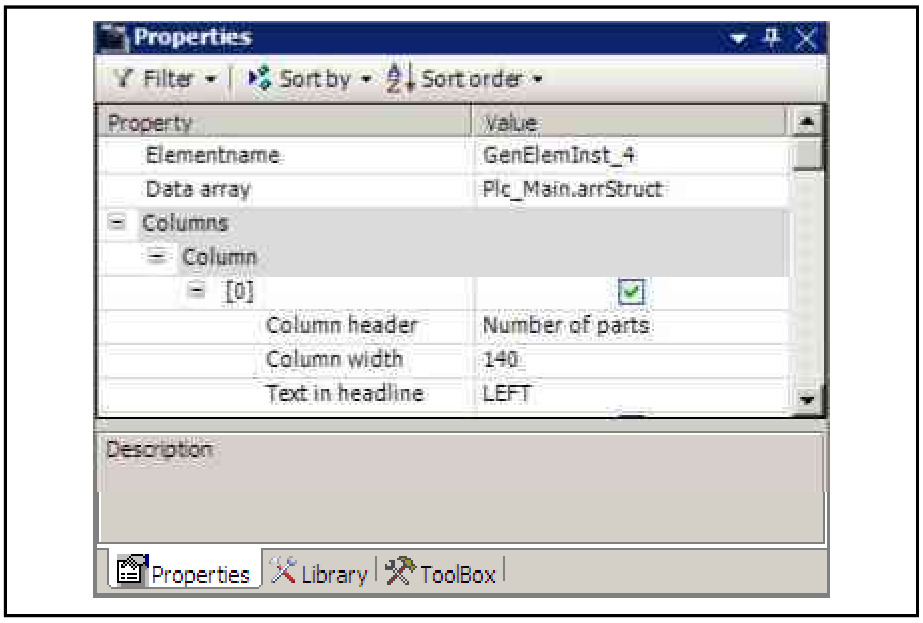
Property	Value
Label	
LabelPosition	INSIDE
Unit	
Font	■ Arial, 9pt
Scale format (C-Syntax)	%.d km/h
max. text width of labels	65
Text height of labels	15

Fig.5-161: Configuration within the "Label" section in the element properties

By changing "LabelPosition" from OUTSIDE to INSIDE, move the scale labels to the inside of the arc. The entry in the "Unit" field appears below the lowest point of the arrow. In the "Font" line, use the ... button to define the font of the display.

In the "Scale format (C syntax)" line, adjust the formatting of the scale labels. The numerical value of the scale has to be formatted using the syntax of the programming language C (use %d for integers and %.Xf for floating point numbers where X can be replaced by the desired number of digits following the decimal point); see . The values in the lines "Max. text width of labels" and "Text height of labels" are automatically entered based on your previous settings in this section. These values have to be changed only if the automatic adjustment does not lead to the desired result.

5. In the "Colors" section, select colors for specific scale ranges by creating "Color areas" with the Create new button. The color areas are numbered in ascending order. Each color area has its own input fields within the element properties.



Property	Value
Elementname	GenElemInst_4
Data array	Plc_Main.arrStruct
Columns	
Column	
[0]	✓
Column header	Number of parts
Column width	140
Text in headline	LEFT

Fig.5-162: Configuration within the "Colors - Color areas" in the element properties

In the "Begin of area" and "End of area" fields, specify the area that is to be filled with color. Select the color from the pop-up menu in the "Colors" line.

Use the ❌ Delete button to delete existing color areas.

The effect of the "Durable color areas" checkbox is only visible in online mode.

The right "meter" element only displays the color area where the arrow is currently located. The left "meter" element displays all of the color areas because there is a check in the "Durable color areas" checkbox.

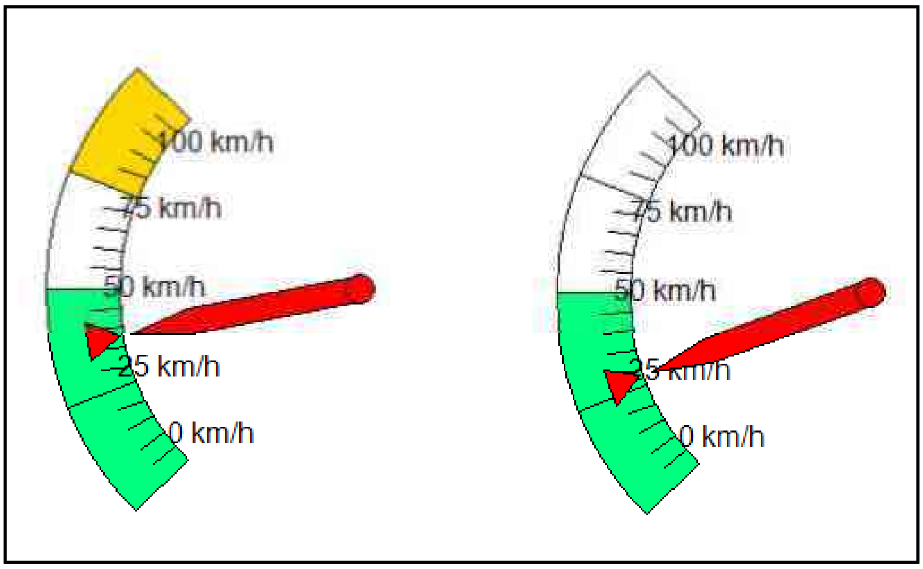
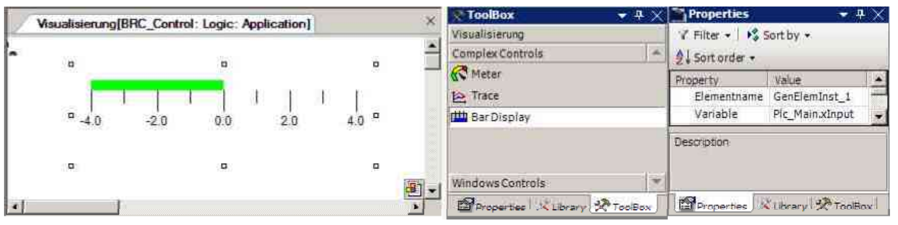


Fig.5-163: Durable color areas option

Configuring the bar display element

To configure the "bar display" element, carry out the following steps:

1. Insert the "bar display" visualization element into a visualization by drag- and dropping it in the visualization. It looks like a thermometer and allows the current value of the associated input variable to be displayed.



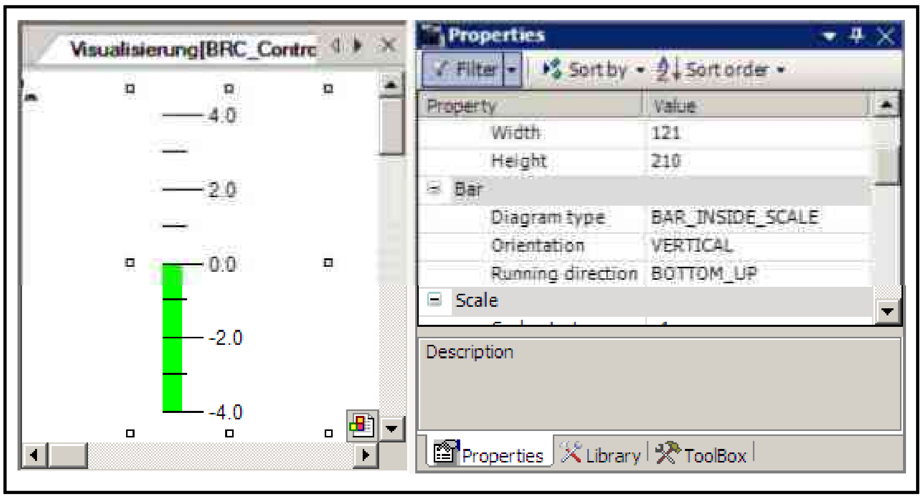
Visualisierung[BRC_Control: Logic: Application]

Property	Value
Elementname	GenElemInst_1
Variable	Plc_Main.xInput

Fig.5-164: Inserting the bar display element and linking to the input variables

In the element properties for the "bar display" element enter the associated input variable (e. g. "xInput") in the "Variable" line. After clicking into the input field, the … button is available. It can be used to search through the project for the input variable. Ensure to specify the input variable by entering its complete path in the Project Explorer.

2. The orientation and placement of the bar with respect to the scale can be set in the "Bar" section of the element properties. The position of bar with respect to the scale can be specified in the "Diagram type" line. The selection field offers SCALE_BESIDE_BAR, SCALE_INSIDE_BAR and BAR_INSIDE_SCALE.



Property	Value
Width	121
Height	210
⊟ Bar	
Diagram type	BAR_INSIDE_SCALE
Orientation	VERTICAL
Running direction	BOTTOM_UP
⊟ Scale	

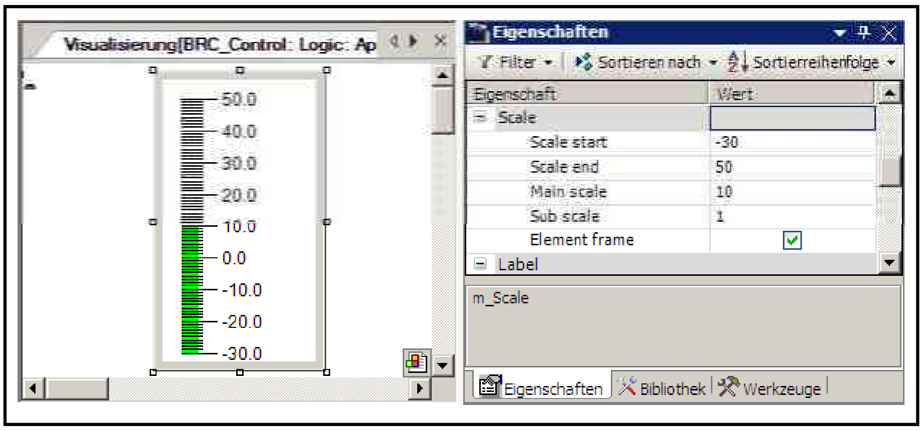
Fig.5-165: Configuration within the "Bar" section in the element properties

In the example, the orientation of the bar has been changed from HORIZONTAL to VERTICAL in the "Orientation" line. This setting also changes the possible entries for the "Running direction" line.

Orientation	Running direction	Description
HORIZONTAL		
	LEFT_RIGHT	The bar runs from left to right
	RIGHT_LEFT	The bar runs from right to left
VERTICAL		
	BOTTOM_UP	The bar runs from top to bottom
	TOP_DOWN	The bar runs from bottom to top

Fig.5-166: Settings for the "bar" property

3. In the "Scale" section, specify the value range of the scale and the subdivisions for rough and precision scaling.



Eigenschaft	Wert
⊟ Scale	
Scale start	-30
Scale end	50
Main scale	10
Sub scale	1
Element frame	✓
⊟ Label	

Fig.5-167: Configuration within the "Scale" section in the element properties

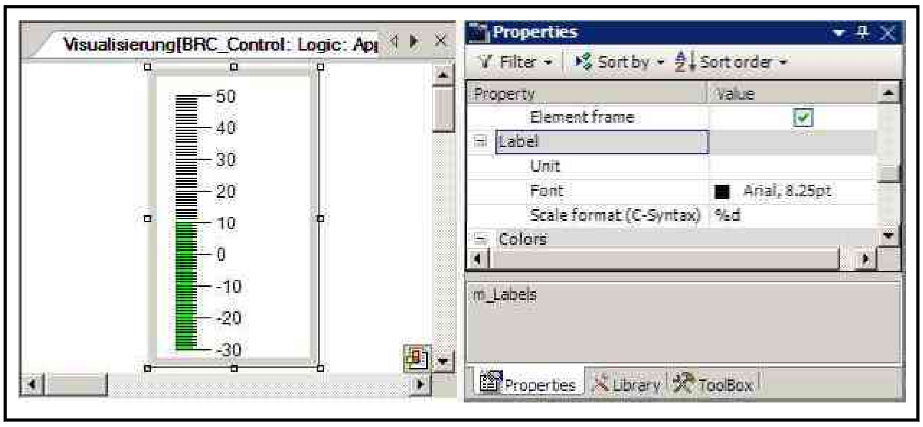
Limit the value range of the scale starting at the bottom using the value in "Scale start" and from the top using the value in "Scale end". The value for "Scale start" has to be lower than that for "Scale end".

The entry for spacing in "Main scale" specifies the marking for rough scaling. With the entry for spacing in "Sub scale" YOU specify the marking for precision scaling. The entries for spacing in "Main scale" and

"Sub scale" can be set to 0 to switch off the display of scaling marks. If the value for rough scaling (Main scale) is set to 0, no scaling marks are displayed, regardless of the value set for precision scaling. If precision scaling (Sub scale) is set to 0, only the scaling marks for rough scaling are displayed.

Place a check in the "Element frame" checkbox to place a frame around the "bar display" element.

4. Format the bar label in the "Label" section.



Property	Value
Element frame	✓
⊟ Label	
Unit	
Font	■ Arial, 8.25pt
Scale format (C-Syntax)	%d
⊟ Colors	

m_Labels

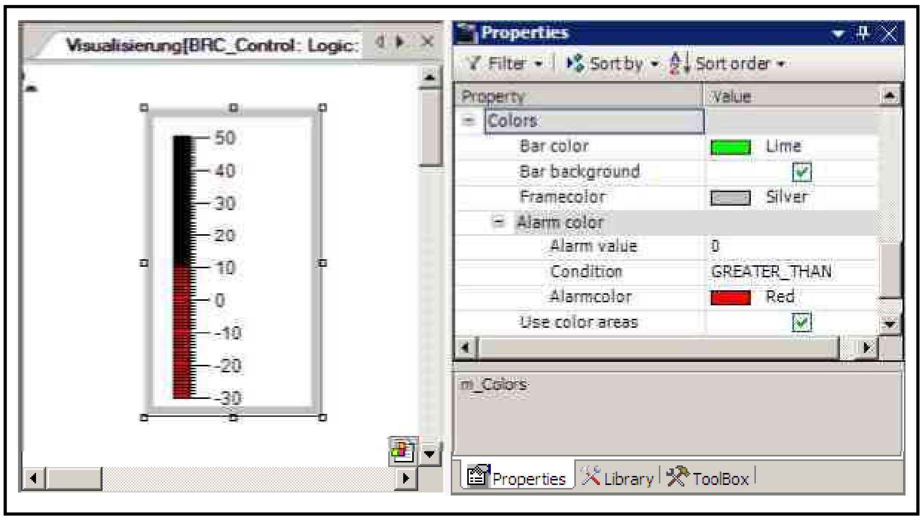
Fig.5-168: Configuration within the "Label" section in the element properties

In the "Unit" input field, specify the unit for the display and it is centered below the bar. In the "Font" line, use the ... button to define the font of the display.

In the "Scale format (C syntax)" line, adjust the formatting of the scale labels. The numerical value of the scale has to be formatted using the syntax of the programming language C (use %d for integers and %.Xf for floating point numbers where X can be replaced by the desired number of digits following the decimal point); see

.

5. In the "Colors" section, specify the coloring of the element.



Property	Value
⊟ Colors	
Bar color	🟩 Lime
Bar background	✓
Framecolor	▨ Silver
⊟ Alarm color	
Alarm value	0
Condition	GREATER_THAN
Alarmcolor	🟥 Red
Use color areas	✓

m_Colors

Fig.5-169: Configuration within the "Color" section in the element properties

In the "Bar color" line, specify the color of the bar.

The default background for the bar that is not currently filled in is white.

Use the "Bar background" checkbox to change the background color to black.

In the "Frame color" line, specify the color of the frame.

In the "Alarm color" subsection, enter an alarm color in the "Alarm color" line. The bar is displayed in this color as soon as the current value of the input variable meets the requirements for the alarm state. The alarm color is determined by the definition of the alarm value in the "Alarm value" line.

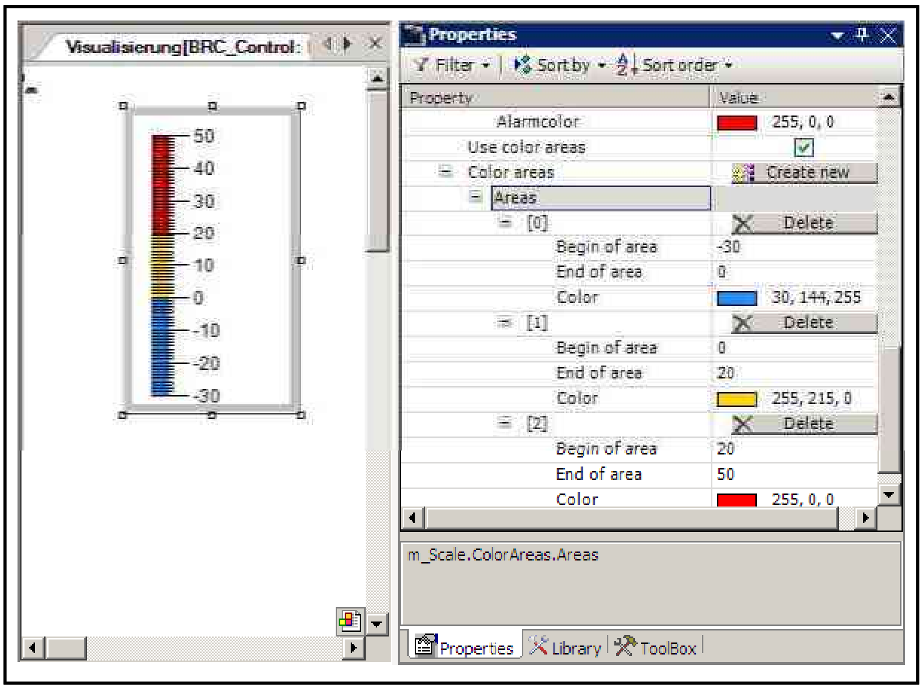
Values may either not exceed or not be less than the "Alarm value" entered. This setting can be made in the "Condition" line with the options GREATER_THAN or LESS_THAN.

By selecting the checkbox "Use color areas", define the color areas for the "bar display" element.

Subsections of the scale can be assigned a certain color by creating a color area with the Create new button. The color areas are numbered in ascending order. Each color area has its own input fields within the element properties.

In the "Begin of area" and "End of area" fields, specify the area that is to be filled with color. Select the color from the pop-up menu in the "Colors" line.

Use the Delete button to delete existing color areas.



Property	Value
Alarmcolor	🟥 255, 0, 0
Use color areas	✓
Color areas	Create new
Areas	
[0]	Delete
Begin of area	-30
End of area	0
Color	🟦 30, 144, 255
[1]	Delete
Begin of area	0
End of area	20
Color	🟨 255, 215, 0
[2]	Delete
Begin of area	20
End of area	50
Color	🟥 255, 0, 0

m_Scale.ColorAreas.Areas

Fig.5-170: Configuration within the "Colors - Color areas" section in the element properties

Configuring the Trace Element

To display traced variables within a visualization, insert a trace object into the project first. In the trace object, configure the variables to be recorded, a trigger variable (as needed) and the trace (axes, update interval, etc.).

Activating the option "Create trace block for visualization" within the implicitly generates the "<TraceName>_<RecordName>_VISU" function block. That is the prerequisite for using the trace object as an input for the trace visualization element.

Creating the trace object

The presence of the trace object is the prerequisite for its used in the visualization.

In the context menu of the application node click on **Add ▸ Trace** to add the "trace" object.

Open the trace object by double clicking on it.

In the trace editor, click on "Configuration" and activate the option "Create trace block for visualization".

Activating the option "Create trace block for visualization" within the trace configuration implicitly generates the "<TraceName>_<RecordName>_VISU" function block in the trace editor, see .

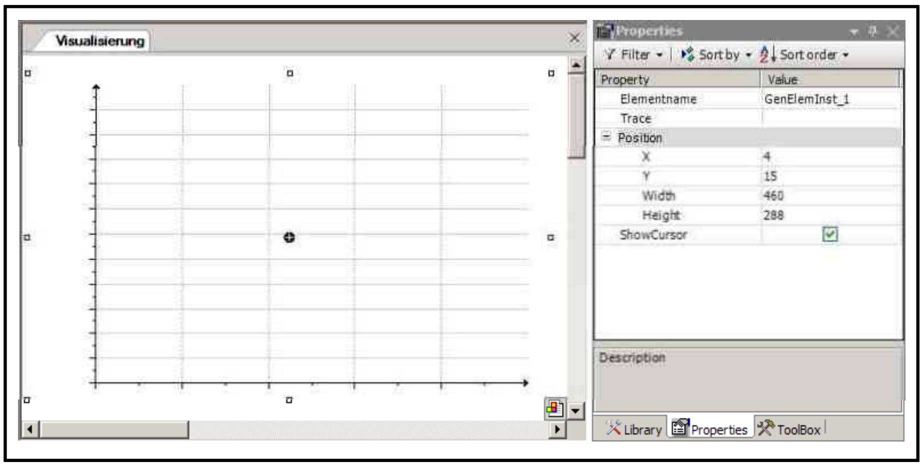
That is the prerequisite for using the trace object as an input for the trace visualization element.

Further information on the "trace" object can be found in and .

Using the trace object in the visualization

To configure the "trace" element, carry out the following steps:

1. Insert the "trace" visualization element into a visualization by dragging it from the and dropping it in the visualization.



Property	Value
Elementname	GenElemInst_1
Trace	
Position	
X	4
Y	15
Width	460
Height	288
ShowCursor	✓

Fig.5-171: Inserting and configuring the trace visualization element

The related trace object is defined within the element properties. In the "Trace" line enter the complete name of the trace, including its path in the project tree (for sequence tracking)! Click the ... button for input assistance.

2. To affect how the "trace" is displayed in online mode, add further visualization elements. The following template includes four buttons for managing trace behavior in the visualization.

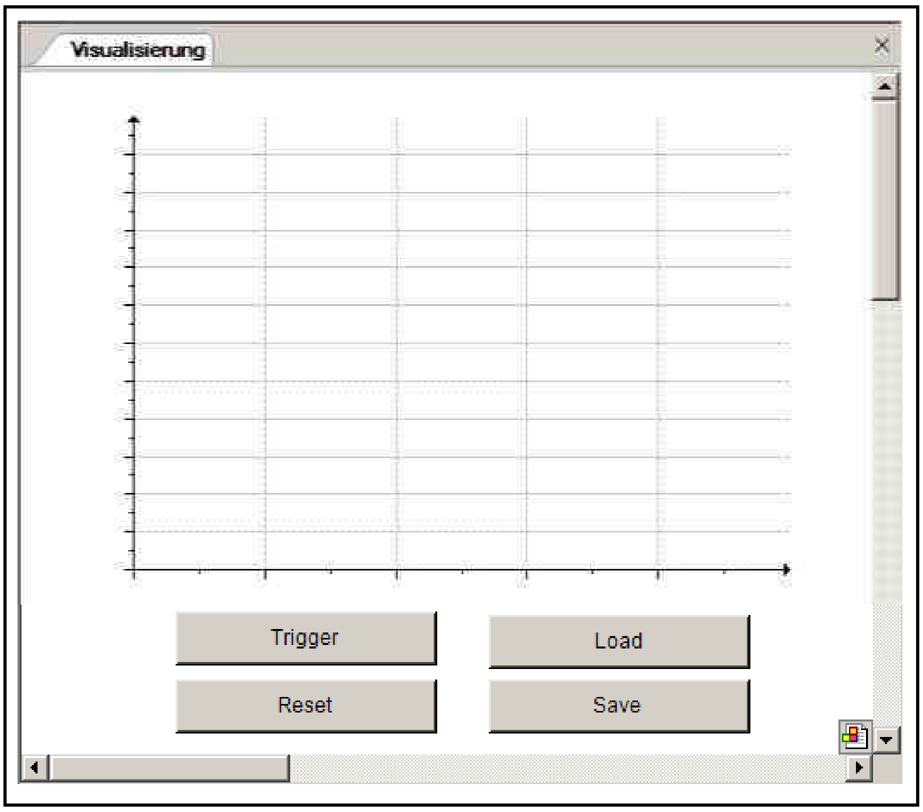
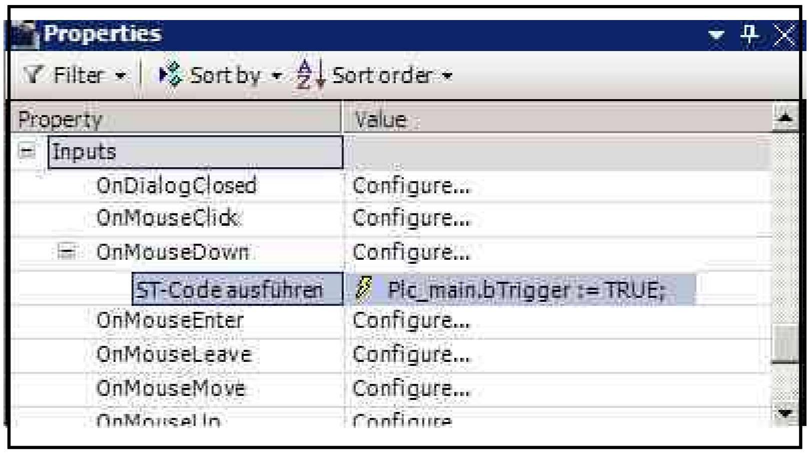


Fig.5-172: Additional elements for managing the trace

- **Trigger**

  The button labeled "Trigger" triggers the trace, i. e. the value "TRUE" is assigned to the associated trigger variable. This can be implemented in the element properties of the button. The "bTrigger" trigger variable has to be announced in the declarations of the main program. To function, the trigger button requires the corresponding instructions in the statements of the main program.

  The special "OnMouseDown" action is connected with the execution of ST code that sets the associated trigger variable to TRUE ("Plc_Main.bTrigger:=TRUE;").



Property	Value
Inputs	
OnDialogClosed	Configure...
OnMouseClick	Configure...
OnMouseDown	Configure...
ST-Code ausführen	Plc_main.bTrigger := TRUE;
OnMouseEnter	Configure...
OnMouseLeave	Configure...
OnMouseMove	Configure...
OnMouseUp	Configure

Fig.5-173: Element properties dialog

- **Reset**

  The "Reset" button should reset the trace, for example, by setting the trigger variable back to FALSE ("Plc_Main.bTrigger:=FALSE;")

and the associated reset variable is set to TRUE (e. g. "Plc_Main.bReset:=TRUE;"). The "bReset" reset variable has to be made known in the declarations of the main program.

To function, the reset button requires the corresponding instructions in the statements of the main program.

- **Load**

  The button labeled "Load" allows the displayed trace to be modified without requiring a logout from the related application.

  To do this, ST code that sets the trigger variable to FALSE is attached and has to be executed. ("Plc_Main.bTrigger:=FALSE;").

  Furthermore, in the code, the "bLoad" variable is set to TRUE, for example with "Trace_PlcTask_VISU.bLoad:=TRUE;".

  This Boolean variable "bLoad" is part of the implicitly generated "<TraceName>_<RecordName>_<_VISU>" block and it monitors the loading of a trace configuration file with the suffix .tcg. The trace visualization (and of course, the trace itself) are adapted for the configuration specified in this file.

  A detailed description of the configuration file can be found in

  .

- **Save**

  The "Save" button allows the trace data currently displayed in the visualization to be saved. To do this, the attached ST code sets the trigger variable to FALSE and the bSave variable to TRUE (for example with

  ```
  Plc_Main.bTrigger:=FALSE;
  Trace_PlcTask_VISU.bSave:=TRUE;
  ```

  This Boolean variable "Trace_PlcTask_VISU.bSave" is part of the implicitly generated block

  ```
  <TraceName>_<RecordName>_VISU
  ```

  block and it requires that a

  ```
  <TraceName>_<RecordName>.dta
  ```

  data file be created in the runtime system. This file contains the currently displayed values of the recorded variables and if necessary it can be read into the trace again at a later time.

The following figure shows the visualization of a trace in online mode after the "Trigger" button has been clicked:

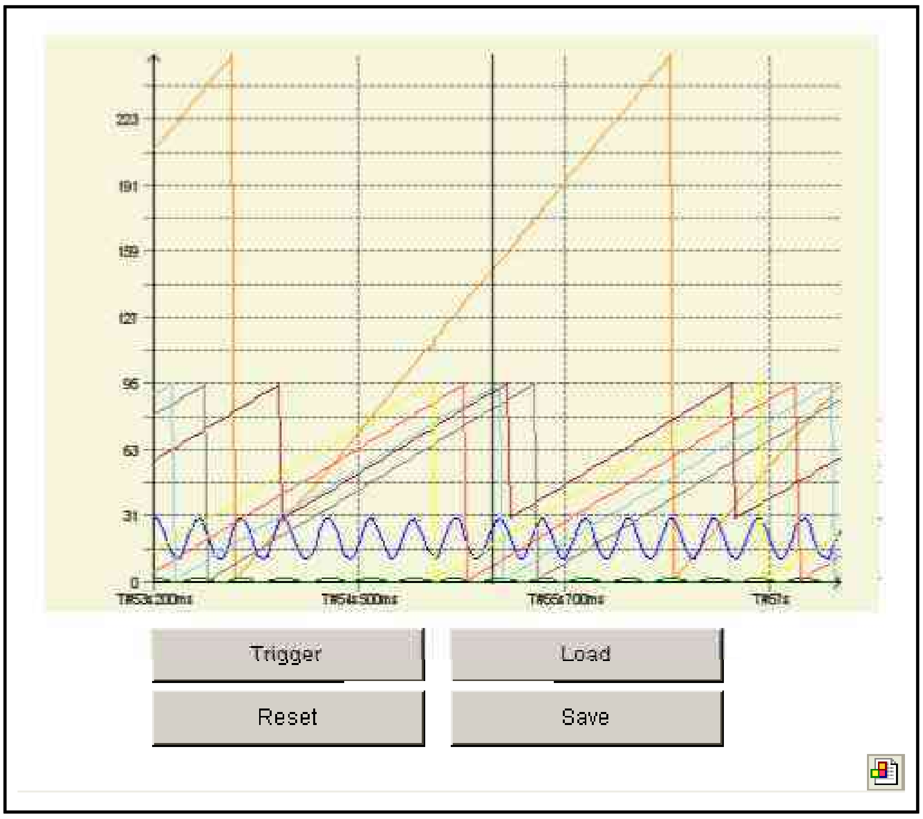


Fig.5-174: Visualization of a trace in online mode

A black vertical line marks the edge of the trigger variables. The recording of the variables continues up to a specified percentage value (TriggerPosition%) of the entire trace interval; then the visualization is stopped.

Activating the "ShowCursor" option in the element properties of the trace visualization element causes a tooltip to display the traced and trigger variables related to the position highlighted by the cursor in the trace visualization element.

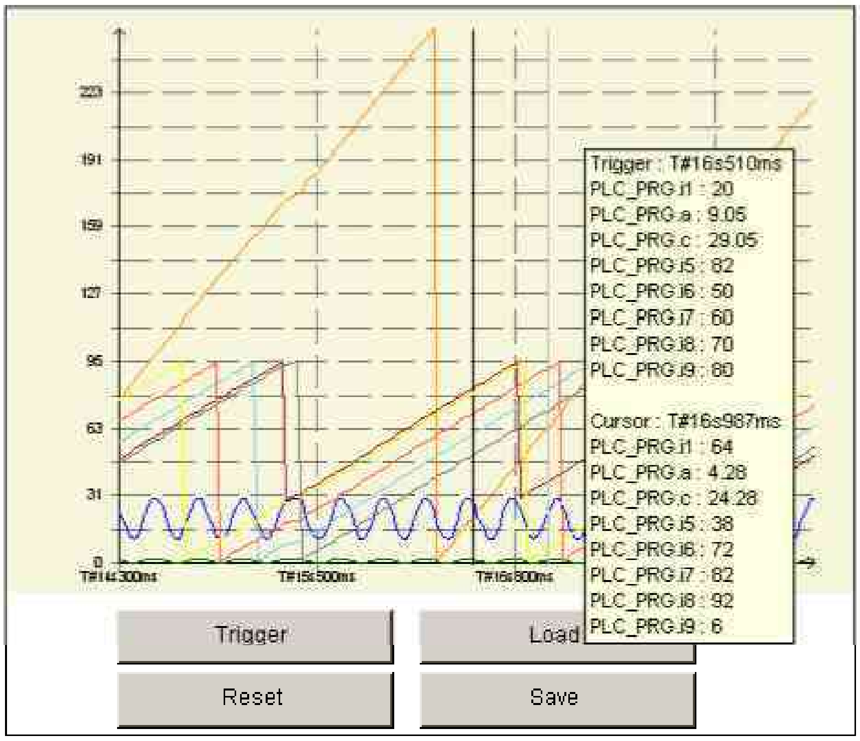
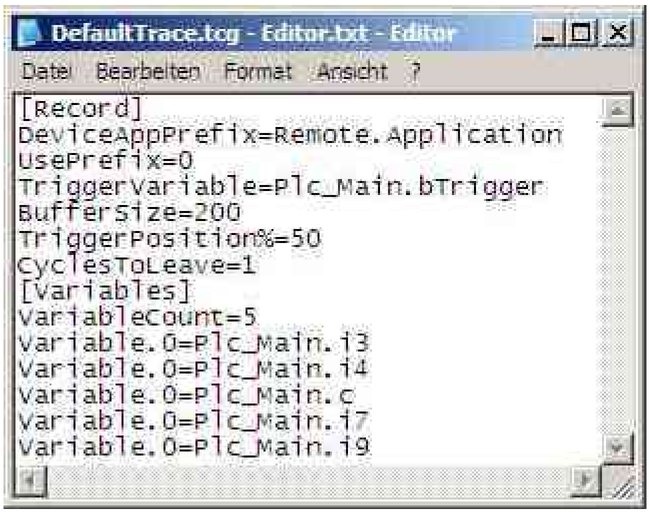


Fig.5-175: Visualization of a trace in online mode with the ShowCursor option

Trace configuration file

The trace configuration file "<TraceName>_<RecordName>.tcg" has to be stored in the directory of the runtime system, for example in "LW:\Programs \Rexroth\IndraWorks\GatewayPLC".


DefaultTrace.tcg - Editor.txt - Editor
Datei Bearbeiten Format Ansicht ?

```
[Record]
DeviceAppPrefix=Remote.Application
UsePrefix=0
TriggerVariable=Plc_Main.bTrigger
BufferSize=200
TriggerPosition%=50
CyclesToLeave=1
[Variables]
VariableCount=5
Variable.0=Plc_Main.i3
Variable.0=Plc_Main.i4
Variable.0=Plc_Main.c
Variable.0=Plc_Main.i7
Variable.0=Plc_Main.i9
```

Fig.5-176: Trace configuration file

- **[Record] section:**

  To avoid having to enter each respective complete name for individual trace variables, set the "DeviceAppPrefix" to the same path percentage as the variables to be traced. This prefix is also displayed in the tooltip if the "UsePrefix" parameter is set to 1. If "UsePrefix" is set to 0, the prefix is hidden in the tooltip.

A detailed description of the TriggerVariable, BufferSize and TriggerPosition% entries can be found in trace editor, configuration, page 436.

The "CyclesToLeave" parameter corresponds with the "Record at every <x> – th cycle" parameter in the trace configuration.

- **[Variables] section:**

  The number of recorded variables can be specified in "VariableCount". Here, the number in the original trace may be maintained or decreased, but it may not be exceeded. All variables to be recorded have be made known with a symbol configuration. Alternatively, declare all of the variables to be recorded by using a data source located in the device that contains the trace.

5.5.18 Visualization Elements of Type "Windows Controls"

Elements overview

The "Windows Controls" category on the "ToolBox" tab contains the following elements:

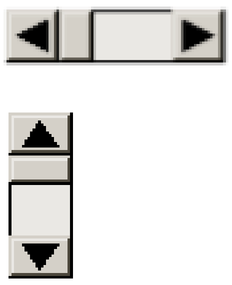
Element name	Element	Description
Table		The "table" element is used to display the values of an array, a structure or the local values of a function block. A column or line is used to display the elements of a one-dimensional array or a structure or a POU. Two-dimensional arrays and arrays that contain structures or POUs as elements are displayed in a matrix structure (columns and lines). A basic type variable can be understood as a one-dimensional array with an element and so it can also be displayed with the table elements (as a table with a single entry).
Text field		The "text field" element is used to display text that is either entered directly into the element properties or that comes from a text input variable. In contrast to the normal rectangle, the frame can be displayed as shadowed.
Scrollbar		The "Scrollbar" element generates a scrollbar for which minimum and maximum values can be defined. The position of the controller is then linked to the value of the input variables. If an output variable is defined, its value can be written by manually positioning the controller. By default, the scrollbar is horizontal, but if the shape of the element changes (by dragging with the mouse) after the scrollbar has been inserted such that the height of the element is greater than its width, the scrollbar switches to vertical.

Fig.5-177: Elements in the "Windows Control" category

Explanations on the table element can be found in examples under the following link: .

A detailed description can be found under .

Configuring the table element

A table element can be inserted into a visualization to visualize one- or two-dimensional arrays, structures or the local variables of a POU.

The table is configured in the element properties window.

Insert the "table" visualization element into a visualization by dragging it from the and dropping it in the visualization.

First, enter the variable to be visualized with the table element in the "Data array" line.

Its structure determines the number of lines and columns in the table.

For an ARRAY:

- The lines represent the first dimension of the array entered.
- If the array is two-dimensional, the columns represent the second index.

For a structure: the individual components represent the structure.

For a POU: they represent the local variables.

A one-dimensional array of elements of the basic type is represented by a single column, whereas

an array of structure type elements is displayed in a table in which the number of lines corresponds with the number of array elements and the number of columns corresponds with the number of structure components.

If the dimensions of the linked variables are changed later, e. g. if the array boundaries are changed or if structure components are added or removed, the table can also be updated by clicking the <Return> key in a table field.

The configuration of the valid components remains.

A **table column can be duplicated** by holding down the <Ctrl> key while left-clicking in the title cell of the column to be duplicated.

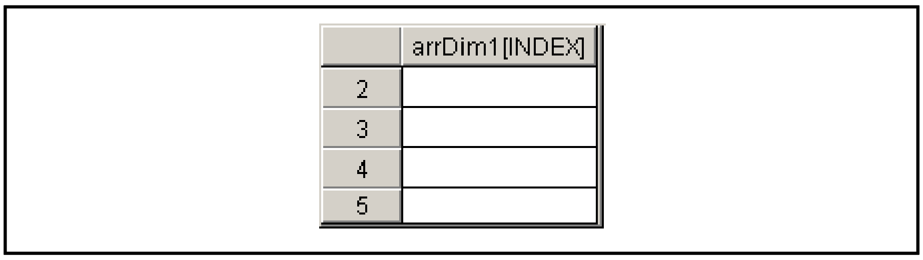
A **table column can be deleted** by holding down the <Ctrl> + <Shift> keys while left-clicking in the title cell of the column to be deleted.

☞ If the entry in the "Data array" field in the element properties is changed, all of the other settings for the element properties are reset to the default settings!

Display of table elements

- **Declaration of a one-dimensional field:**

```
arrDim1: ARRAY [2..5] OF INT;
```



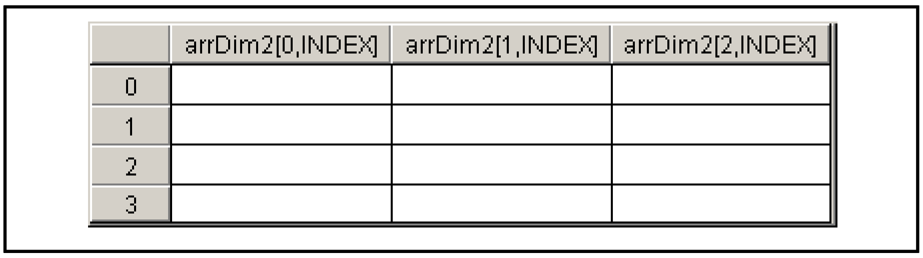
	arrDim1[INDEX]
2	
3	
4	
5	

Fig.5-178: "Table" visualization element as a one-dimensional array

Note that the indexing in the line labeling matches the indexes of the first dimension of the array; for this reason, the line numbering in this example starts with "2"!

- **Declaration of a two-dimensional field:**

```
arrDim2: ARRAY [0..2, 0..3] OF INT;
```

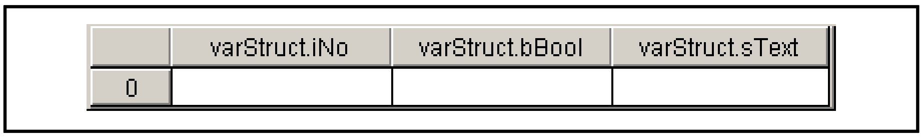


	arrDim2[0,INDEX]	arrDim2[1,INDEX]	arrDim2[2,INDEX]
0			
1			
2			
3			

Fig.5-179: "Table" visualization element as a two-dimensional array

- **Type declaration for the structure "MyStruct" and its use:**

```
TYPE MyStruct :
STRUCT
  iNo: INT;
  bBool: BOOL;
  sText: STRING;
END_STRUCT
END_TYPE
VAR
  varStruct: MyStruct;
END_VAR.
```

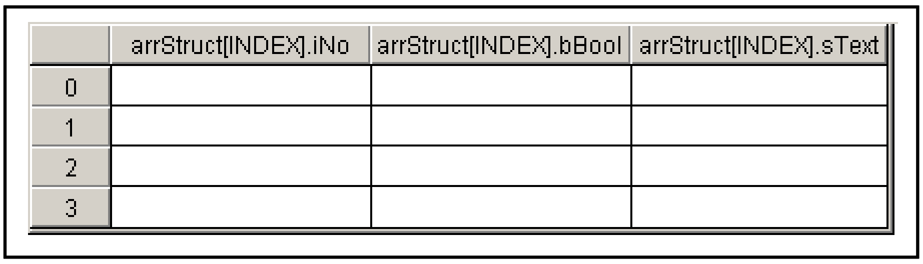


	varStruct.iNo	varStruct.bBool	varStruct.sText
0			

Fig.5-180: "Table" visualization element as a structure with a line

- **Declaration of a field based on "MyStruct":**

```
VAR
  arrStruct: ARRAY[0..3] OF MyStruct;
END_VAR.
```

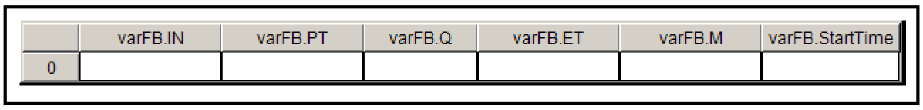


	arrStruct[INDEX].iNo	arrStruct[INDEX].bBool	arrStruct[INDEX].sText
0			
1			
2			
3			

Fig.5-181: "Table" visualization element as an ARRAY OF STRUCT with several lines

- **Declaration of an FB instance:**

```
VAR
  varFB: TON;
END_VAR.
```



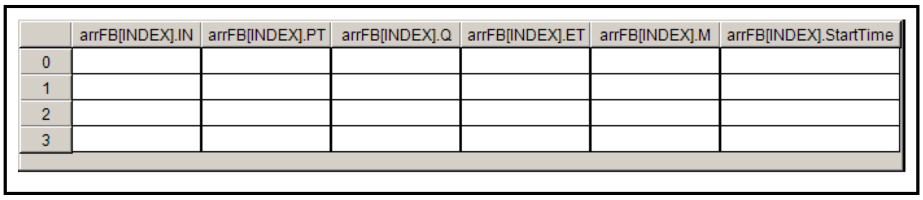
	varFB.IN	varFB.PT	varFB.Q	varFB.ET	varFB.M	varFB.StartTime
0						

Fig.5-182: "Table" visualization element as a one-dimensional function block

When calling a function block, all of the inputs and outputs have their own columns.

- **Declaration of a field of FB instances:**

```
VAR
  arrFB: ARRAY[0..3] OF TON;
END_VAR.
```



	arrFB[INDEX].IN	arrFB[INDEX].PT	arrFB[INDEX].Q	arrFB[INDEX].ET	arrFB[INDEX].M	arrFB[INDEX].StartTime
0						
1						
2						
3						

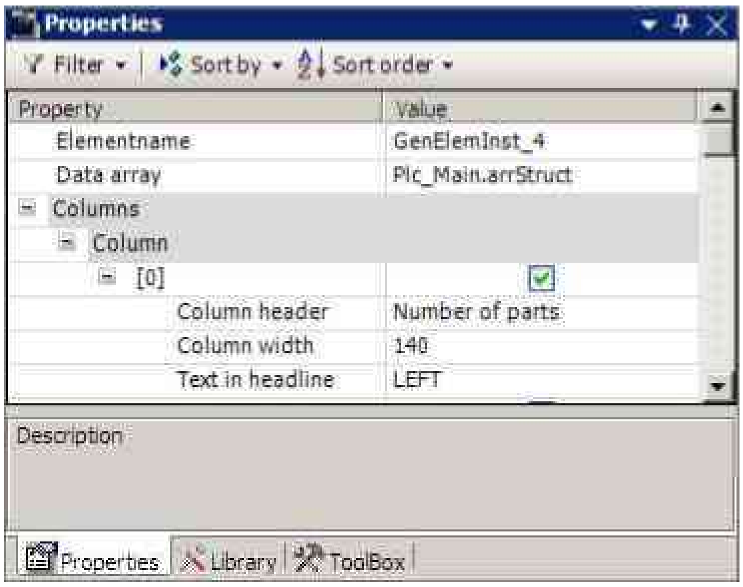
Fig.5-183: "Table" visualization element as a two-dimensional function block

In addition to a few basic properties, e. g. size, position or line width in the table, a wide variety of specific properties can be set for the table. A selection of these properties are explained further here using a step-by-step continuation of the "arrStruct" example.

Properties of a table elements

A selection of these properties are explained further here using a step-by-step continuation of the "arrStruct" example.

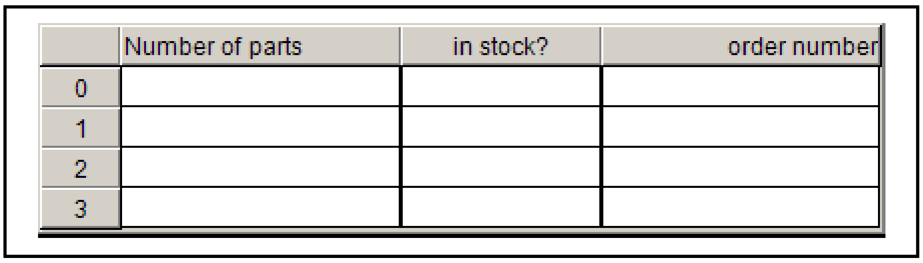
1. In the element properties, change the default entries in the "Column header" column to more meaningful titles, adjust the width of the columns and specify the alignment of the heading.



Property	Value
Elementname	GenElemInst_4
Data array	Plc_Main.arrStruct
Columns	
Column	
[0]	✓
Column header	Number of parts
Column width	140
Text in headline	LEFT

Fig.5-184: Specifying column width, heading and alignment

The effects of these changes:

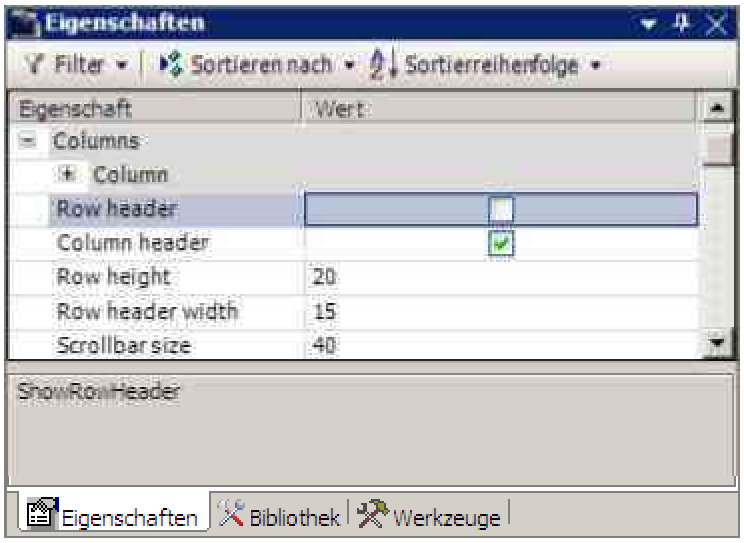


	Number of parts	in stock?	order number
0			
1			
2			
3			

Fig.5-185: Column heading as defined

2. Specify the height of the lines as 20 pixels in the "Row height" line. Switch off the line labeling by removing the checkmark from the "Row header" line, so that now the first column that contains the array indexes is missing.

   The following figure also shows the wider scrollbars (line: "Scrollbar size"), which are inserted in the table as soon as its size decreases so that it is less than the comprehensive line heights or column widths.

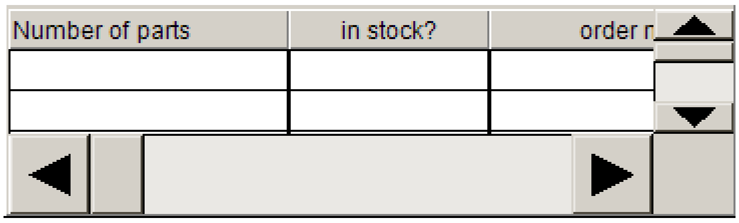


Eigenschaft	Wert
Columns	
Column	
Row header	☐
Column header	✓
Row height	20
Row header width	15
Scrollbar size	40

ShowRowHeader

Fig.5-186: Specifying column height and scrollbars; switching off line labeling

The effects of these changes:



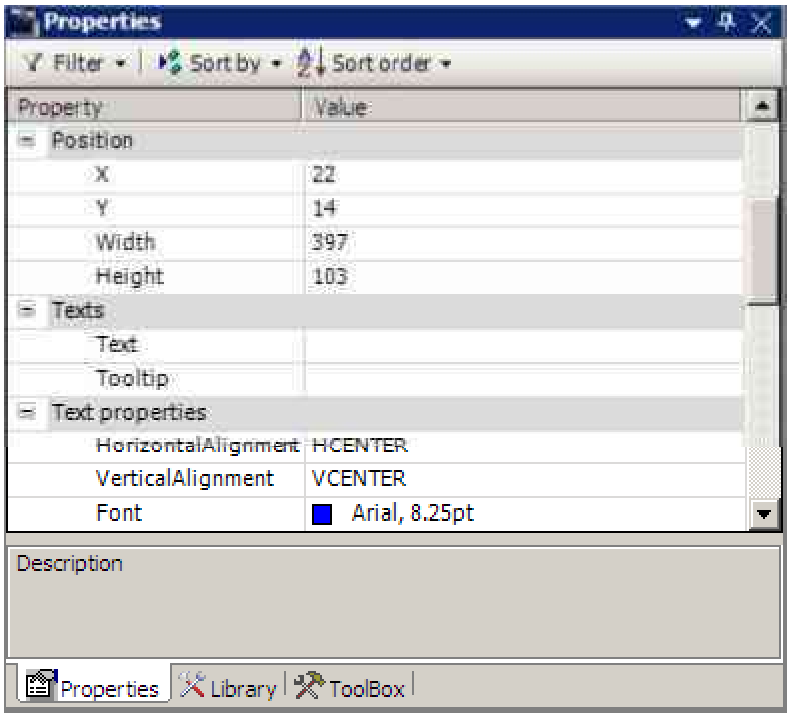
Number of parts	in stock?	order n

Fig.5-187: Expanded column options

3. The position of the "table" element is specified in the "Position" element property in the X and Y lines or by dragging with the mouse to the desired position.

   The scrollbars can be hidden with the "WITH" and "High" lines by increasing the default values. Their height is determined by the line height multiplied by the number of lines. The width of the table is the sum of the individual column widths and the width of the line labeling (if it is active).

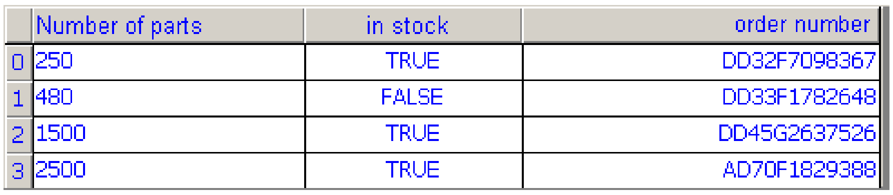
The text display is specified in the "Text properties" element property. In the "Font" line, use the ... button to define the font of the display. The horizontal text alignment in the "HorizontalAlignment" line only applies for line labeling. The horizontal text alignment in the "VerticalAlignment" line only applies for column labeling. The alignment of the texts in individual table cells is based on the settings made in the column properties.



Property	Value
Position	
X	22
Y	14
Width	397
Height	103
Texts	
Text	
Tooltip	
Text properties	
HorizontalAlignment	HCENTER
VerticalAlignment	VCENTER
Font	🟦 Arial, 8.25pt

Fig.5-188: Hiding scrollbars; text options

The appearance of the (online) display of the table has now changed.



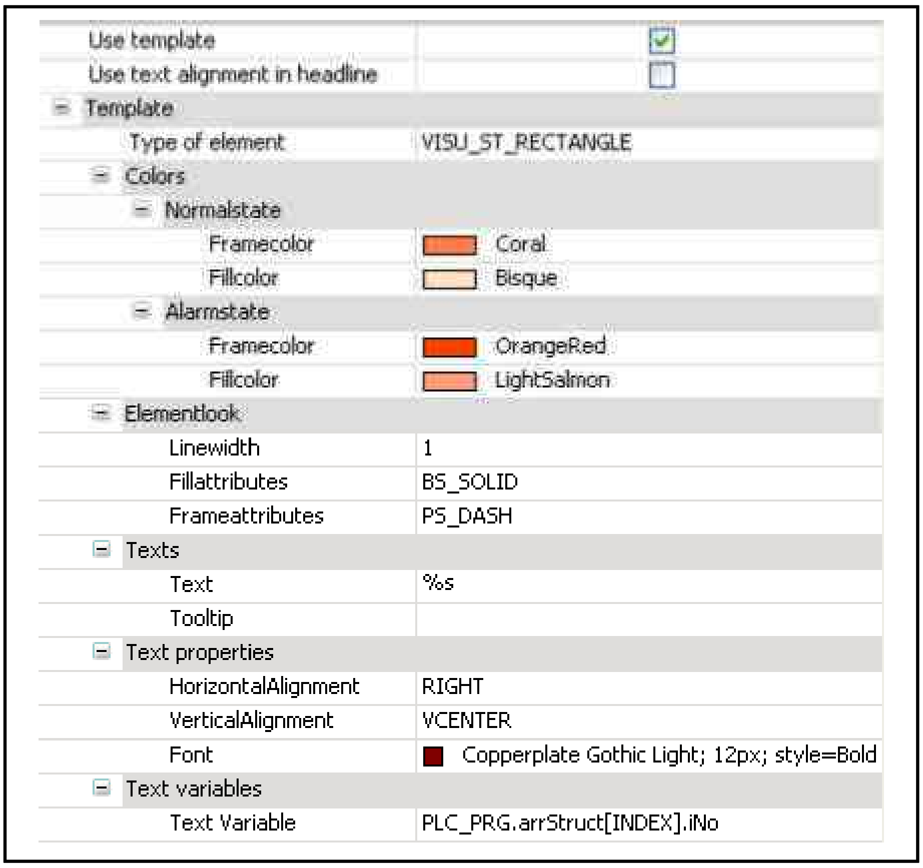
	Number of parts	in stock	order number
0	250	TRUE	DD32F7098367
1	480	FALSE	DD33F1782648
2	1500	TRUE	DD45G2637526
3	2500	TRUE	AD70F1829388

Fig.5-189: Online display

Define the values in the columns and lines in your program.

Note that the alignment of the entries in the table cells corresponds with the setting for the individual columns.

4. Table fields can be edited line by line. For the column selected, select the "Use template" checkbox: Then, another "Template" point is inserted and expands in the column properties.



Use template	✓
Use text alignment in headline	☐
⊟ Template	
Type of element	VISU_ST_RECTANGLE
⊟ Colors	
⊟ Normalstate	
Framecolor	🟧 Coral
Fillcolor	▭ Bisque
⊟ Alarmstate	
Framecolor	🟥 OrangeRed
Fillcolor	🟧 LightSalmon
⊟ Elementlook	
Linewidth	1
Fillattributes	BS_SOLID
Frameattributes	PS_DASH
⊟ Texts	
Text	%s
Tooltip	
⊟ Text properties	
HorizontalAlignment	RIGHT
VerticalAlignment	VCENTER
Font	🟥 Copperplate Gothic Light; 12px; style=Bold
⊟ Text variables	
Text Variable	PLC_PRG.arrStruct[INDEX].iNo

Fig.5-190: Expanded options for editing table fields line by line

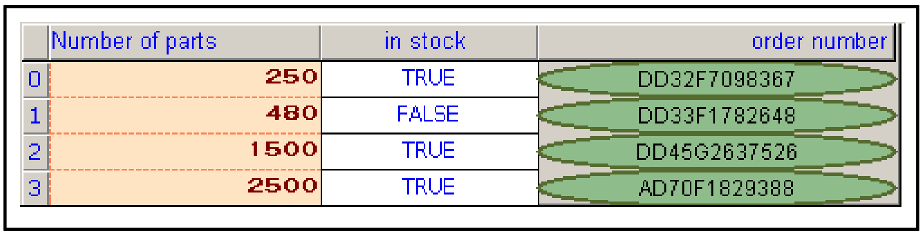
In the "Type of element" element property, three types are available in the menu for displaying the cells:

- VISU_ST_RECTANGLE (rectangle)
- VISU_ST_ROUNDRECT (rectangle with rounded corners)
- VISU_ST_CIRCLE (circular)

The related configuration possibilities show the element properties for the selected template element. Select the fill and frame colors for the normal and alarm states. They can also be toggled with a variable. The alignment of the entries in the column cells is now specified by the template setting (with the exception of the heading).

Instead of the array components entered by default, any other variable in the project can be entered in the "Text variables" field. In this way it is possible to display the array elements in the table in a different order.

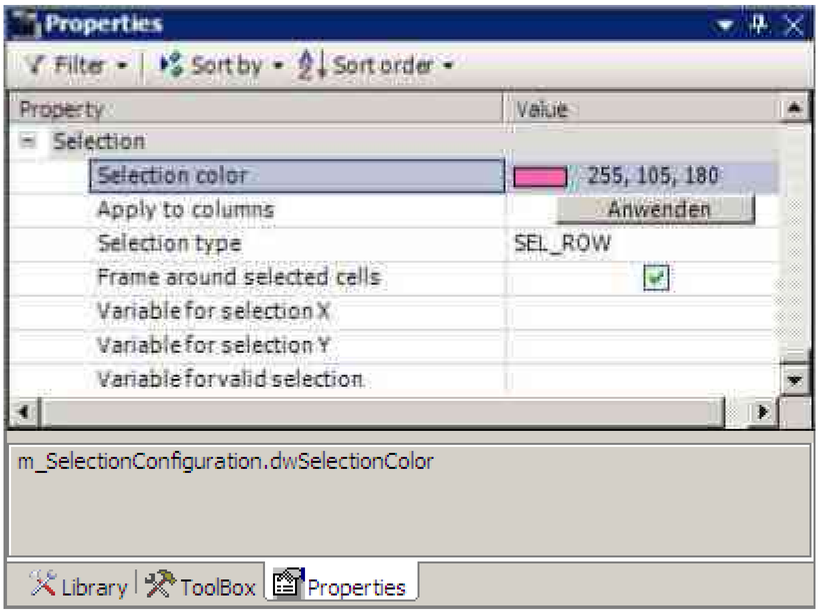
In the example, a template was used for the first and third columns.



	Number of parts	in stock	order number
0	250	TRUE	DD32F7098367
1	480	FALSE	DD33F1782648
2	1500	TRUE	DD45G2637526
3	2500	TRUE	AD70F1829388

Fig.5-191: Example configuration with templates

5. The alarm state cannot only be triggered by a variable value, but also by highlighting a table cell by clicking on it in online mode. The behavior of the selected region can be specified in the "Selection" section.



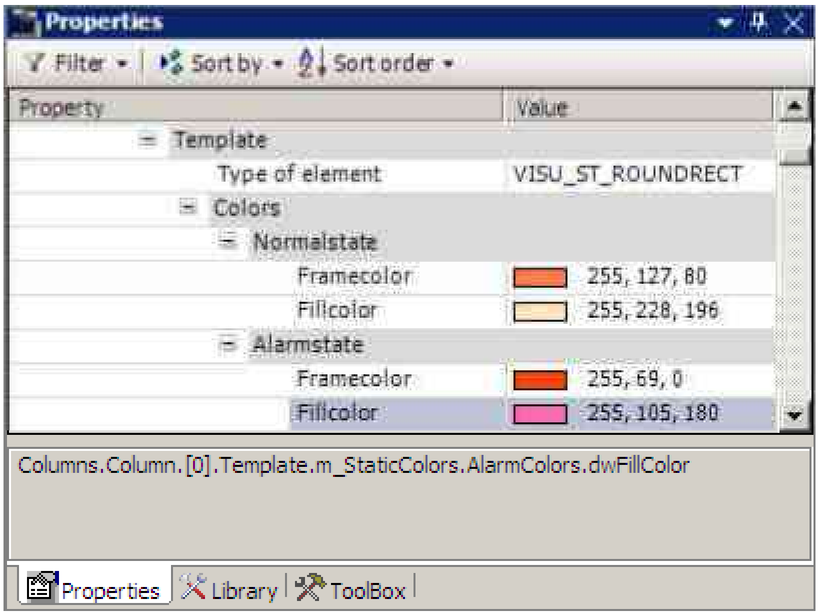
Property	Value
Selection	
Selection color	🟪 255, 105, 180
Apply to columns	Anwenden
Selection type	SEL_ROW
Frame around selected cells	✓
Variable for selection X	
Variable for selection Y	
Variable for valid selection	

m_SelectionConfiguration.dwSelectionColor

Fig.5-192: Defining alarm color in online mode

The specified background color in the "Selection color" line for highlighted elements applies for all highlighted elements for which no alarm color was specified (within a template). So in our example, a highlighted cell in the second column is filled with hot pink, while a highlighted cell in the first column is filled with light salmon, the alarm color of the associated template.

After selecting the color, click on <Apply> to replace the alarm colors in the specific template settings with the selection made in the "Selection color" field. As a result, the following entries are made in the corresponding fields in the template for the first column of our example:



Property	Value
Template	
Type of element	VISU_ST_ROUNDRECT
Colors	
Normalstate	
Framecolor	🟧 255, 127, 80
Fillcolor	🟧 255, 228, 196
Alarmstate	
Framecolor	🟥 255, 69, 0
Fillcolor	🟪 255, 105, 180

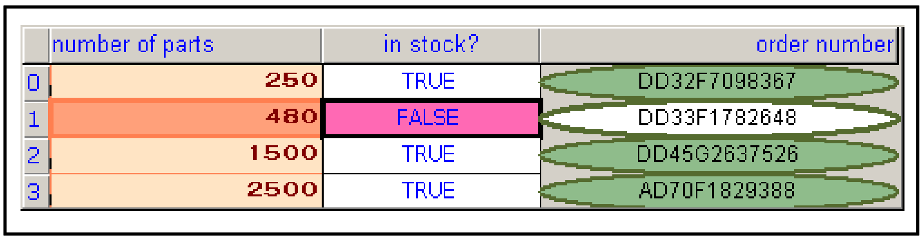
Columns.Column.[0].Template.m_StaticColors.AlarmColors.dwFillColor

Fig.5-193: Alarm colors accepted into the template

The "Selection type" field specifies which elements are to be highlighted by clicking on a single table cell. In this example, the SEL_ROW type was selected, and so all of the cells that are in the same line as the cell that was clicked are highlighted and framed.

If no columns of the display in the table were excluded, the position of the selected cells within the table represents the indexes of the visualized array associated with the entry. These indexes can be saved in variables that are entered in the last three lines of the "Selection" section for this purpose. The third variable is Boolean and is set to TRUE as long as the selection is valid.

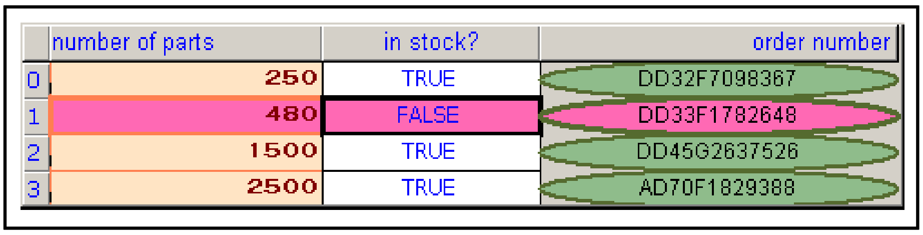
After the cell with the entry "FALSE" is clicked, all of the cells in the second line are highlighted. Only the highlighted cell in the second column has the selected color because the color for the alarm state in the other columns is set by the templates.



	number of parts	in stock?	order number
0	250	TRUE	DD32F7098367
1	480	FALSE	DD33F1782648
2	1500	TRUE	DD45G2637526
3	2500	TRUE	AD70F1829388

Fig.5-194: Selected color for alarm state

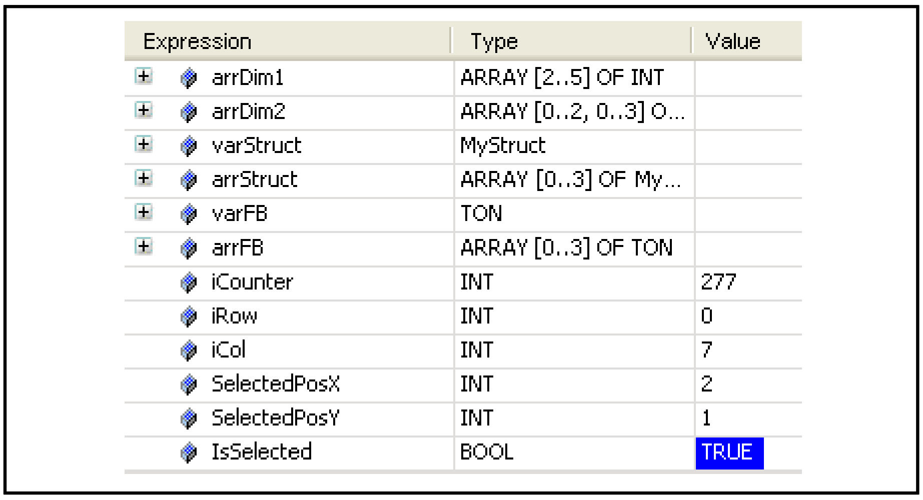
After applying the selected color for the templates as well, the highlighting looks different.



	number of parts	in stock?	order number
0	250	TRUE	DD32F7098367
1	480	FALSE	DD33F1782648
2	1500	TRUE	DD45G2637526
3	2500	TRUE	AD70F1829388

Fig.5-195: Identical colors for the alarm state in all of the columns

The array index related to the selected cell is assigned to the associated variables as a value.



Expression	Type	Value
+ arrDim1	ARRAY [2..5] OF INT	
+ arrDim2	ARRAY [0..2, 0..3] O...	
+ varStruct	MyStruct	
+ arrStruct	ARRAY [0..3] OF My...	
+ varFB	TON	
+ arrFB	ARRAY [0..3] OF TON	
iCounter	INT	277
iRow	INT	0
iCol	INT	7
SelectedPosX	INT	2
SelectedPosY	INT	1
IsSelected	BOOL	TRUE

Fig.5-196: Variables in the online view

5.5.19 Keyboard Operation in Online Mode

Some standard shortcuts are supported by every device for keyboard operation in a visualization in online mode. In addition, depending on the device, other key (combinations) can be used.

Default shortcuts:

Key(s)	Action
<Tab>	The next element according to the chronological sequence of insertion is selected; in this case, within a table, each individual cell is selected; if a frame element is selected, the selection is forwarded to the individual elements contained
<Shift>+ <Tab>ƆRƇ	Previous element is selected; reverse order than with <Tab>
<Enter>	Input action executed for the selected element
<Arrow keys>	Next element in the direction indicated by the arrow key is selected

Fig.5-197: Shortcuts

☞ keyboard operation can be activated and deactivated explicitly using the command . This can be desirable because as long as keyboard operation is activated for the visualization, other commands that are given using shortcuts are not executed.

Device-specific keyboard operation:

In addition to the standard keys, it depends on the device which other key (combinations) can be used for the visualizations that run on the device. See and .

☞ There is an option for handling in the application code.

5.5.20 Note Events and Input Actions

Overview

Using methods provided by the visualization libraries, the application can take notice of specific events triggered by user inputs in a visualization:

- ,
- .

Note the Closing of an Edit Controls (Writing a Variable)

The application can determine when an "Edit Control" is closed in an associated visualization, i.e. when a variable value is written due to user input in an input field. This can be useful if another action is to be executed after an Edit Control is closed.

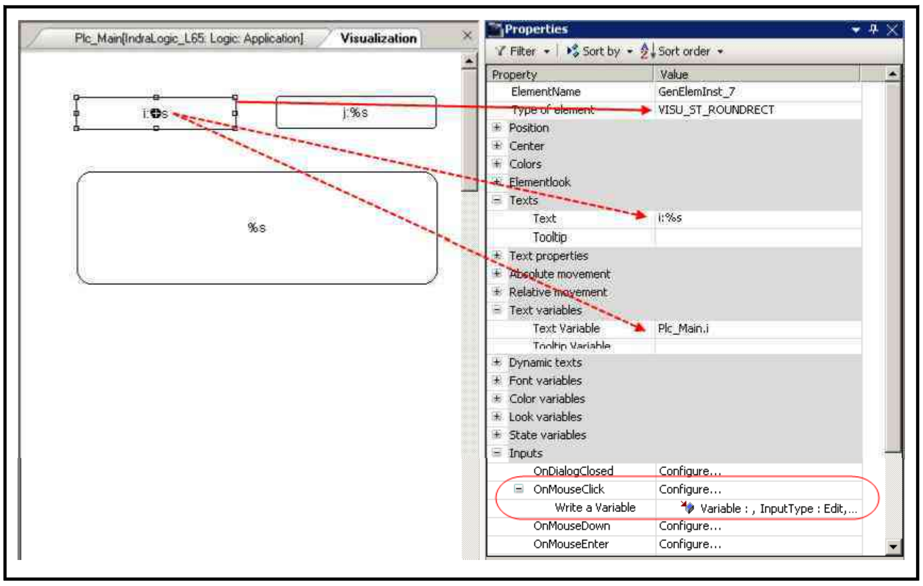
To receive information about the closing, a function block instance (that the interface VisuElems.IEditBoxInputHandler from the VisuElemBase.library implements) can be assigned to the global instance VisuElems.Visu_Globals.g_VisuEventManager; in this case, the method SetEditBoxEventHandler of this instance is used as shown in the example below.

VisuElems.IEditBoxInputHandler requires the method VariableWritten, which is called after a variable value is written due to user input in a visualization.

Example:

There is a visualization with the following elements:

1. two rectangles, each with an Edit Control field (input "Write variable") in which the user can input a value for variable i or j.



Plc_Main[IndraLogic_L65: Logic: Application] Visualization

i:%s j:%s %s

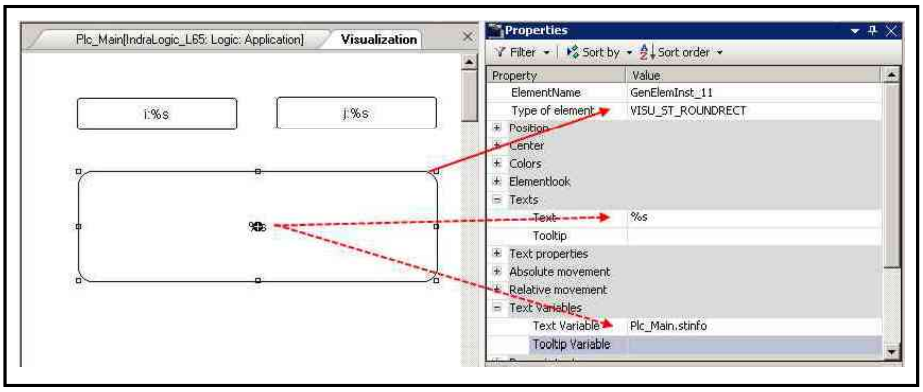
Properties

Filter Sort by Sort order

Property	Value
ElementName	GenElemInst_7
Type of element	VISU_ST_ROUNDRECT
Position	
Center	
Colors	
Elementlook	
Texts	
Text	i:%s
Tooltip	
Text properties	
Absolute movement	
Relative movement	
Text variables	
Text Variable	Plc_Main.i
Tooltip Variable	
Dynamic texts	
Font variables	
Color variables	
Look variables	
State variables	
Inputs	
OnDialogClosed	Configure...
OnMouseClick	Configure...
Write a Variable	Variable : , InputType : Edit,...
OnMouseDown	Configure...
OnMouseEnter	Configure...

Fig.5-198: Input elements

2. Another rectangle in which a text (text variable stInfo) with its own information is shown if one of the input fields in either of the two upper rectangles is closed, i.e. if a value is written.



Plc_Main[IndraLogic_L65: Logic: Application] Visualization

i:%s j:%s %s

Properties

Filter Sort by Sort order

Property	Value
ElementName	GenElemInst_11
Type of element	VISU_ST_ROUNDRECT
Position	
Center	
Colors	
Elementlook	
Texts	
Text	%s
Tooltip	
Text properties	
Absolute movement	
Relative movement	
Text Variables	
Text Variable	Plc_Main.stinfo
Tooltip Variable	

Fig.5-199: Text output element

The application contains the following blocks:

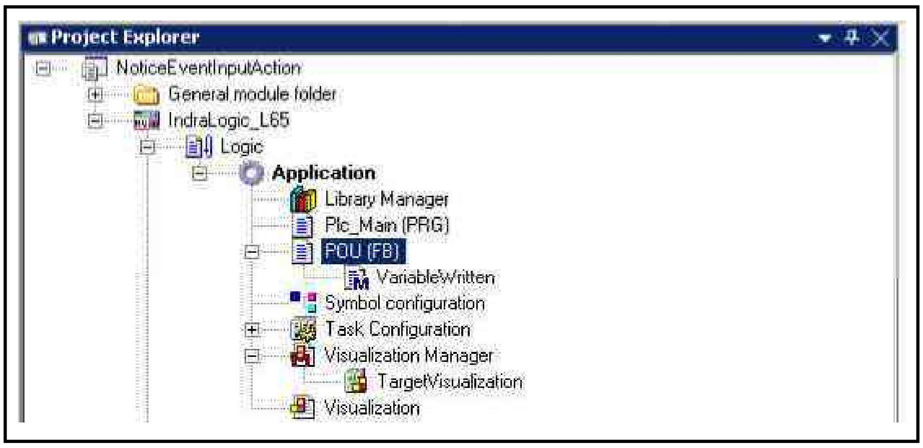


Fig.5-200: Application objects in the Project Explorer

PROGRAM Plc_Main

```
PROGRAM Plc_Main
VAR_INPUT
  i:INT;           // variable to be written by user input in visualization
  j:REAL;          // variable to be written by user input in visualization
  stInfo : STRING; (* information on the user input via the edit control field;
                      string gets composed by method VariableWritten;
                      result is displayed in the lower rectangle of the visualization *)
END_VAR
VAR
 inst : POU;
 bFirst : BOOL := TRUE;
END_VAR

 IF bFirst THEN
 bFirst := FALSE;
 VisuElems.Visu_Globals.g_VisuEventManager.SetEditBoxEventHandler(inst);
                  // Call of method VariableWritten
END_IF
```

Function block POU

```
FUNCTION_BLOCK POU IMPLEMENTS VisuElems.IEditBoxInputHandler
```

(no further declarations, no implementation code)

The method "VariableWritten" provides information when an Edit Control in the visualization is closed, i.e. when a variable is written based on user input in one of the upper rectangles, to the text variable "stinfo", which is to be displayed in the lower rectangle.

The text variable is combined.

Method VariableWritten, part of POU

```
METHOD VariableWritten : BOOL (* provides some information
       always when an edit control field is closed in the visualization,
       i.e. a variable gets written by user input in one of the upper rectangles *)
VAR_INPUT
 pVar : POINTER TO BYTE;
 varType : VisuElems.Visu_Types;
 iMaxSize : INT;
 pClient : POINTER TO VisuElems.VisuStructClientData;
END_VAR
// String stinfo, which will be displayed in the lower rectangle, is composed here
Plc_Main.stInfo := 'Variable written; type: ';
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, INT_TO_STRING(varType));
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, ', adr: ');
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, DWORD_TO_STRING(pVar));
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, ', by: ');
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, SEL(pClient^.globaldata.clienttype =
 VisuElems.Visu_ClientType.Targetvisualization, 'other visu', 'targetvisu'));
```

Note keyboard events

The application can specify key actions (keyboard event) in an associated visualization, i.e. it notices when the user **presses and releases a key** when the visualization is active.

☞ Note the configuration possibilities for keyboard operation in visualizations.

To receive information about such events, a function block (that the interface VisuElems.IVisuUserEventManager from the VisuElemBase.library implements) can be assigned to the global instance VisuElems.Visu_Globals.g_VisuEventManager and the method **SetKeyEventHandler** of this instance is used as shown in the following example:

Example: There is a visualization with a visualization element that displays the value Plc_Main.stInfo as text. That means that in online operation, the following information is output in this element on the currently activated key action:

"KeyEvent up: <dwKey>,

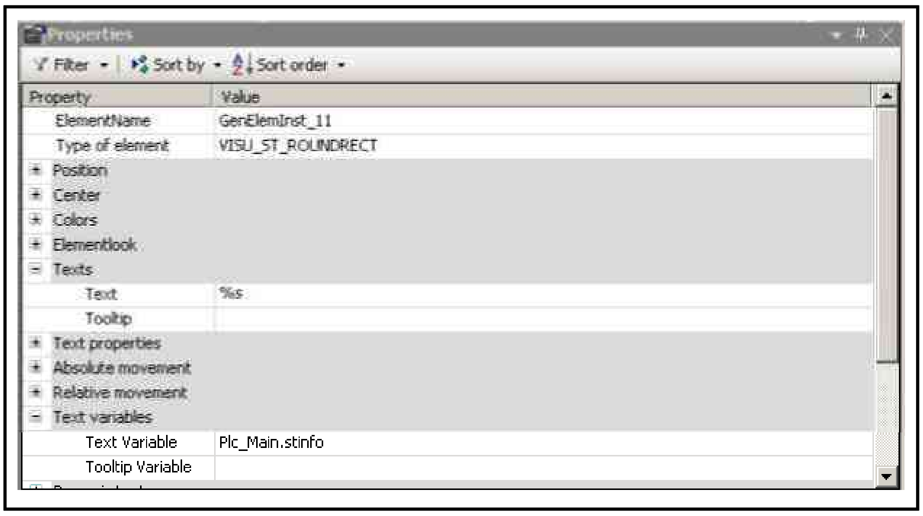
key: <key id>,

modifier: <dwModifiers>,

by: <pClient^.globaldata.clienttype.....>"

.

Example: "KeyEvent up: TRUE, key: 75, modifier: 0, by: targetvisu".



Property	Value
ElementName	GenElemInst_11
Type of element	VISU_ST_ROUNDRECT
+ Position	
+ Center	
+ Colors	
+ Elementlook	
− Texts	
Text	%s
Tooltip	
+ Text properties	
+ Absolute movement	
+ Relative movement	
− Text variables	
Text Variable	Plc_Main.stinfo
Tooltip Variable	

Fig.5-201: Text configuration of the visualization element

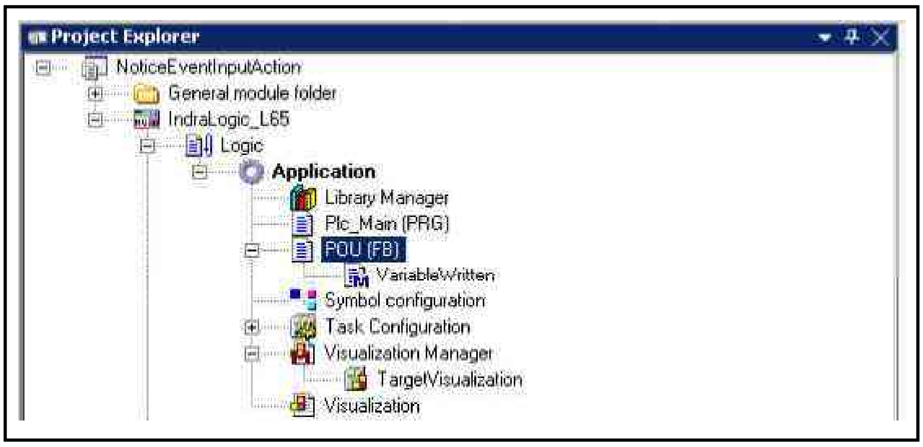


Fig.5-202: Application objects in the Project Explorer

PROGRAM Plc_Main

```
PROGRAM Plc_Main
VAR_INPUT
  stInfo : STRING;
END_VAR
VAR_OUTPUT
     g_VisuEventManager: INT;
END_VAR
VAR
 inst : POU;
 bFirst : BOOL := TRUE;
END_VAR
 IF bFirst THEN
 bFirst := FALSE;
 VisuElems.Visu_Globals.g_VisuEventManager.SetKeyEventHandler(inst);
END_IF
```

This makes it possible to determine when a keyboard event is triggered.

Function block POU

```
FUNCTION_BLOCK POU IMPLEMENTS VisuElems.IKeyEventHandler
```

(no further declarations, no implementation code)

The interface VisuElems.IVisuUserEventManager requires a method with the following signature that is called with all of the available information:

Method HandleKeyEvent, assigned to FB POU

```
   // This method will be called after a key event is released.
   // RETURN:
   // TRUE -  When the handler has handled this event and it should not be handled by someone else
   // FALSE - When the event is not handled by this handler
METHOD HandleKeyEvent : BOOL
VAR_INPUT
   // Event type. The value is true if a key-up event was released.
 bKeyUpEvent : BOOL;
   // Key code
 dwKey : DWORD;
   // Modifier. Values:
   // VISU_KEYMOD_SHIFT : DWORD := 1;
   // VISU_KEYMOD_ALT : DWORD := 2;
   // VISU_KEYMOD_CTRL : DWORD := 4;
 dwModifiers : DWORD;
   // Pointer to the client structure were the event was released
 pClient : POINTER TO VisuStructClientData;
END_VAR
VAR
END_VAR
Plc_Main.stInfo := 'KeyEvent up: ';
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, BOOL_TO_STRING(bKeyUpEvent));
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, ', key: ');
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, DWORD_TO_STRING(dwKey));
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, ', modifier: ');
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, DWORD_TO_STRING(dwModifiers));
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, ', by: ');
Plc_Main.stInfo := CONCAT(Plc_Main.stInfo, SEL(pClient^.globaldata.clienttype =
 VisuElems.Visu_ClientType.Targetvisualization, 'other visu', 'targetvisu'));
```

5.5.21 Input Dialogs

A visualization can be created as an input dialog and defined in its object properties as a "dialog" (visualization object, context menu**Properties...** ▸ **Visualization**, "Visualization used as:" dialog).

Dialog visualizations can then be used in other visualizations to provide the user with an input mask. Dialog visualizations are offered when configuring an "OpenDialog" or "OnDialogClosed" property of a visualization element, for example.

In addition to the dialog visualizations created by yourself, the standard login dialog visualization provided with the **VisuDialogs.library** library can always be used.

The parameters defined in the interface of a dialog visualization are automatically written in a

`<DialogName>_VISU_STRUCT`

("DialogName" = Name of the visualization) structure; see

.

This structure can then be used by the application, e. g. using a function that is called if there is user input for a visualization element (configuration of the visualization element). In this way, opening a dialog in a visualization and the reaction to user input in this dialog can be programmed.

DialogManager

All visualizations that are configured as dialogs are automatically instanced and managed by the internal DialogManager.

This manager can be addressed by the VisuManager, which is also internal, by calling the **GetDialogManager()** method.

The DialogManager provides some methods for the application to handle a visualization dialog:

The following provides an example for configuring a login dialog:

Configuration of a login dialog

In this example, a button element in a visualization is configured so that if the "Login" button is clicked, a login dialog opens, in which a user name and a password can be entered. The default login dialog is provided by the "Visu-Dialog.library", but a self-created dialog could be handled in the same way:

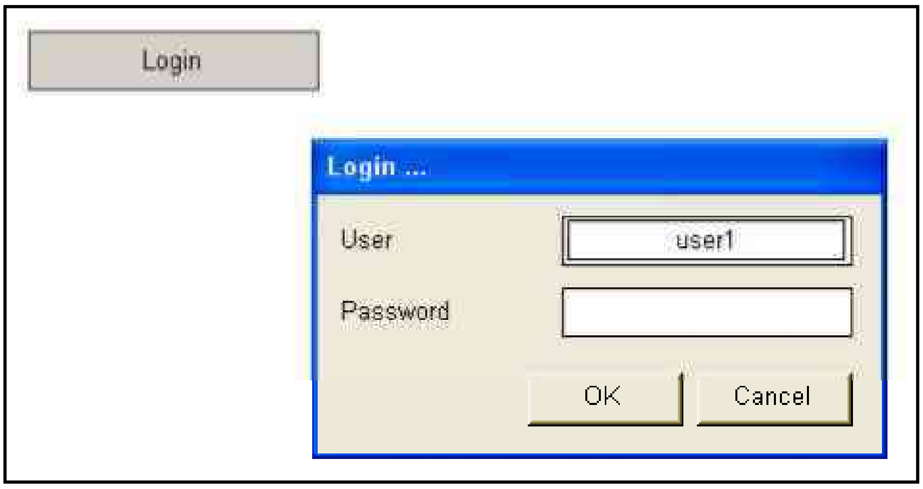


Fig.5-203: Login dialog

A button element is added to a visualization. It is configured with the text property "Login" and the following two input properties:

Configuring the "Login" button:

1. In your project, add a visualization to an application.

   Then add a function to the global "General module" folder and name the function "OpenLoginDialog".

   Add a second function "OnLoginDialogClosed".

   The following is now displayed in Project Explorer:

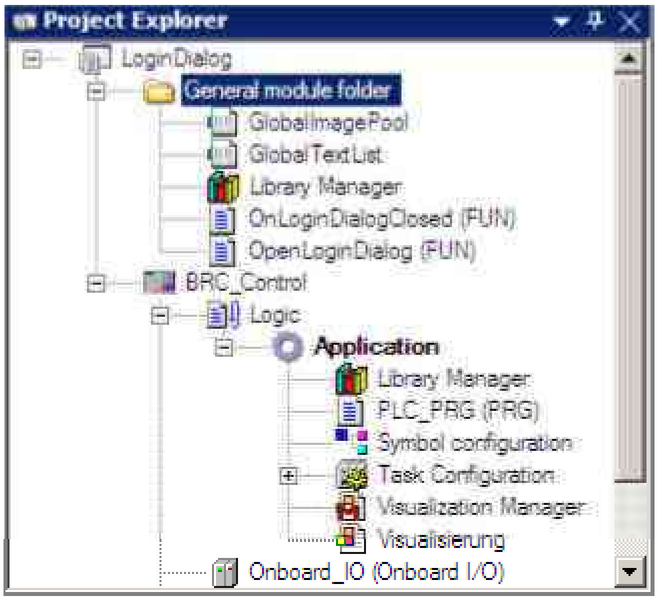


Fig.5-204: Project LoginDialog in Project Explorer

- The function "OpenLoginDialog" is called when the Login dialog opens, e. g. during an OnMouseDown action on the button element. This function reads the parameter of the open Login dialog.

  An example for a possible implementation can be seen in

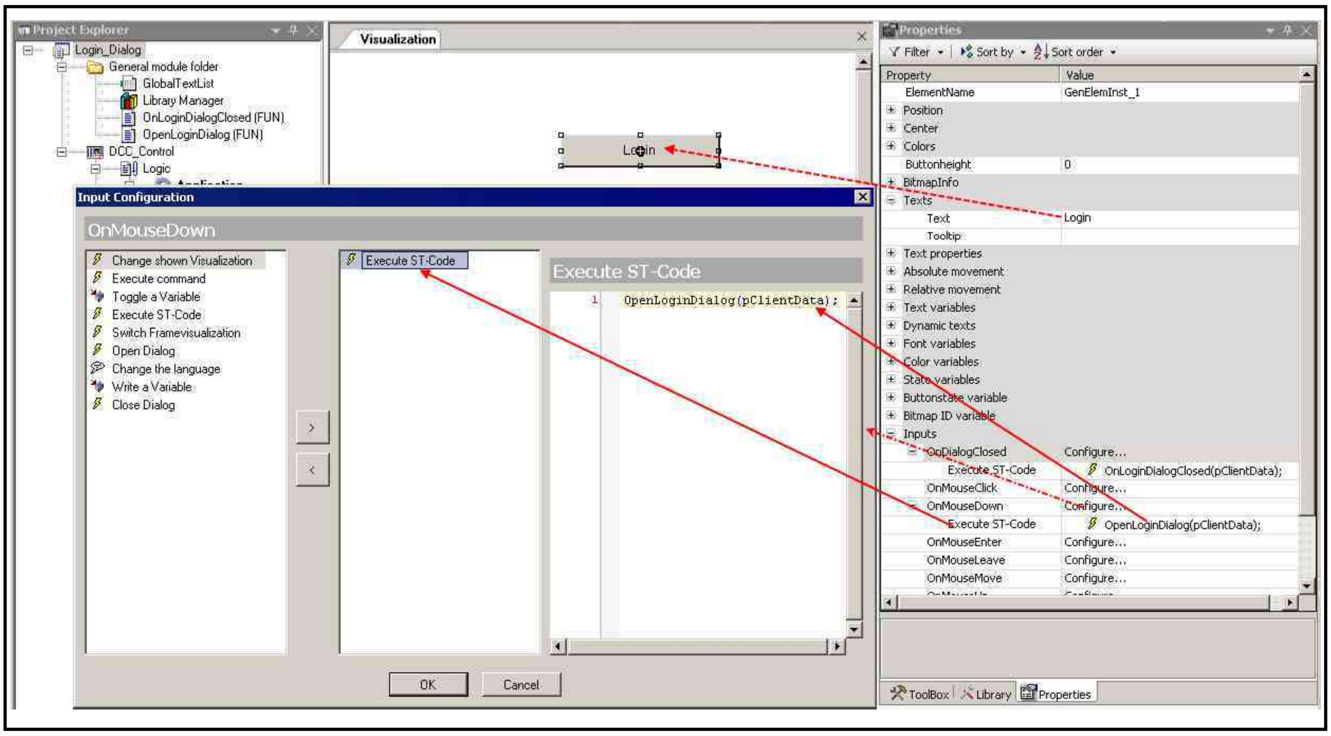
  .

- The function "OnLoginDialogClosed" defines the reaction to the closing of the Login dialog.

  An example for a possible implementation can be seen in

  .

2. Add the "Button" element to a visualization. Name the "Button" element "Login" in the "Text" property. In the "Inputs" property, configure the following inputs.



```
OpenLoginDialog(pClientData);
```

Property	Value
ElementName	GenElemInst_1
Position	
Center	
Colors	
Buttonheight	0
BitmapInfo	
Texts	
Text	Login
Tooltip	
Text properties	
Absolute movement	
Relative movement	
Text variables	
Dynamic texts	
Font variables	
Color variables	
State variables	
Buttonstate variable	
Bitmap ID variable	
Inputs	
OnDialogClosed	Configure...
Execute ST-Code	OnLoginDialogClosed(pClientData);
OnMouseClick	Configure...
OnMouseDown	Configure...
Execute ST-Code	OpenLoginDialog(pClientData);
OnMouseEnter	Configure...
OnMouseLeave	Configure...
OnMouseMove	Configure...

Fig.5-205: Configuring the "Login" button

2.1 OnMouseDown / Execute ST code:

Function call '**OpenLoginDialog(pClientData)**'.

The variable 'pClientData' has to be transferred to a respective 'OpenLoginDialog()' function existing in the project.

2.2 OnDialogClosed / Close dialog / Execute ST code:

Function call '**OnLoginDialogClosed(pClientData)**'.

The respective functions 'OpenLoginDialog()' and 'OnLoginDialogClosed()' are present as function blocks in the project.

See a possible implementation as follows:

Implementation example:

The function 'OpenLoginDialog()' is called when the dialog opens, e.g. during an OnMouseDown action on the button element. This function reads the parameter of the open dialog.

Function OpenLoginDialog

```
FUNCTION OpenLoginDialog : BOOL
VAR_INPUT
 pClientData : POINTER TO VisuStructClientData;
END_VAR
VAR
 dialogMan : IDialogManager;
 loginDialog : IVisualisationDialog;
 pLoginInfo : POINTER TO Login_VISU_STRUCT;
 (* Login_VISU_STRUCT contains the parameters that are
   defined in the interface of the visualisation "Login" *)
 result : Visu_DialogResult;
 stTitle : STRING := 'Login ...';
 stPasswordLabelText: STRING;
 stUserLabelText: STRING;
 stUsername: STRING;
END_VAR
dialogMan := g_VisuManager.GetDialogManager();
 (* The DialogManager is provided by the VisuManager that is
   implicitly available *)
```

```
IF dialogMan <> 0 AND pClientData <> 0 THEN
  loginDialog := dialogMan.GetDialog('VisuDialogs.Login'); (* The dialog to be opened is indicated *)
  IF loginDialog <> 0 THEN
    pLoginInfo := dialogMan.GetClientInterface(loginDialog, pClientData);
    IF pLoginInfo <> 0 THEN      (* Now the parameters of the Login dialog of
   Login_VISU_STRUCT are read *)
      pLoginInfo^.stTitle := stTitle;
      pLoginInfo^.stPasswordLabelTxt := stPasswordLabelText;
      pLoginInfo^.stUserLabelTxt := stUserLabelText;
      dialogMan.OpenDialog(loginDialog, pClientData, TRUE, 0);
    END_IF
  END_IF
END_IF
```

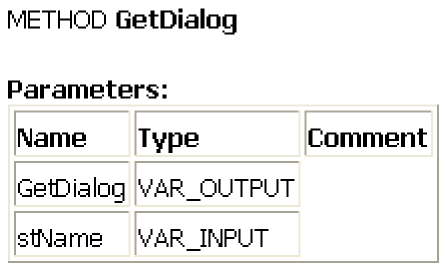
The function "OnLoginDialogClosed()" defines the reaction to the closing of the Login dialog.

Function OnLoginDialogClosed

```
FUNCTION OnLoginDialogClosed : BOOL
VAR_INPUT
 pClientData : POINTER TO VisuStructClientData;
END_VAR
VAR
 dialogMan : IDialogManager;
 loginDialog : IVisualisationDialog;
 pLoginInfo : POINTER TO Login_VISU_STRUCT;
 result : Visu_DialogResult;
 stPassword: STRING;
 stUsername: STRING;
END_VAR
dialogMan := g_VisuManager.GetDialogManager();      (* The DialogManager is provided by the VisuManager
   that is implicitly available *)
IF dialogMan <> 0 AND pVisuClient <> 0 THEN
  loginDialog := dialogMan.GetDialog('VisuDialogs.Login'); (* recognizes the Login dialog *)
  IF loginDialog <> 0 THEN
  result := loginDialog.GetResult(); (* recognizes the result (OK, Cancel) of the dialog *)
  IF result = Visu_DialogResult.OK THEN
    loginDialog.SetResult(Visu_DialogResult.None); (* Reset to Default (none) *)
    pLoginInfo := dialogMan.GetClientInterface(loginDialog, pVisuClient);
           (* Structure Login_VISU_STRUCT is read; in the following
   the structure parameters can be set *)
    IF pLoginInfo <> 0 THEN
      stPassword := pLoginInfo^.stPassword;
      pLoginInfo^.stPassword := ''; (* Password is reset *)
      stUsername := pLoginInfo^.stUsername;
    END_IF
    ELSIF result = Visu_DialogResult.Cancel THEN
      loginDialog.SetResult(Visu_DialogResult.None);
      (* Response to "Cancel" ("Abbrechen") *)
    ELSE
      (* nothing to do here *)
    END_IF
  END_IF
END_IF
```

Methods of the DialogManager

- **GetDialog(STRING stName):** Provides the dialog currently affected by an event as an IVisualizationDialog.



METHOD **GetDialog**

Parameters:

Name	**Type**	**Comment**
GetDialog	VAR_OUTPUT	
stName	VAR_INPUT	

- **GetClientInterface():** Provides a pointer to the dialog structure.

METHOD **GetClientInterface**

Parameters:

Name	**Type**	**Comment**
GetClientInterface	VAR_OUTPUT	
dialog	VAR_INPUT	
pClient	VAR_INPUT	

- **OpenDialog():** Opens the dialog for the client.

  METHOD **OpenDialog**

  Parameters:

Name	**Type**	**Comment**
OpenDialog	VAR_OUTPUT	
dialog	VAR_INPUT	
pClient	VAR_INPUT	
bModal	VAR_INPUT	
pRect	VAR_INPUT	

- **CloseDialog():** Closes the dialog for the client.

  METHOD **CloseDialog**

  Parameters:

Name	**Type**	**Comment**
CloseDialog	VAR_OUTPUT	
dialog	VAR_INPUT	
pClient	VAR_INPUT	