



## 5 Programming Reference

### 5.1 Declaration

#### 5.1.1 Variable Declaration

A variable can be declared in the statements for a POU using the or in a

The data type of the variables to be declared is specified by the

For example, the declaration of simple variables begins with "VAR" and ends with "END\_VAR".

*For other types, refer to*

- 
- 
- 
- 
- 
- 
- 
- 
- 

The variable type keywords can be supplemented by which also consist of keywords.

Example: "RETAIN" (VAR RETAIN).

A variable declaration has to follow these rules:

**Syntax:**

< > { <Address>}: <DataType> {:=<Initialization>};

The sections in curly brackets {} are optional.

**Identifier**

The identifier is the name of the variable. The points listed below have to be considered when assigning an identifier. For further recommendations for assigning an identifier, see

- An identifier may not contain any spaces or special characters.
- Case is not taken into consideration for identifiers, i.e. "VAR1", "Var1" and "var1" identify the same variable.
- Underscores are recognized (i.e. "A\_BCD" and "AB\_CD" are treated as two different identifiers), but a sequence of multiple underscores is not permitted.
- The length of an identifier and its significant parts are unlimited.
- Also note the rules for "Multiple use of identifiers (namespace, validity ranges)".

*Multiple use of identifiers (namespaces, validity ranges):*

- An identifier may not be locally used twice.
- An identifier may not be identical with a keyword.



## Programming Reference

- An identifier may be used globally more than once. Thus, a local variable may have the same name as a global variable. In this case, the local variable takes precedence locally.

- can have the same name as a variable defined in another GVL.

In this context, please note the following functionalities that extend the standard with regard to the **namespace**/validity range of variables that were not available in IndraLogic 1.x:

1. Operators that apply globally, "global scope":

An instance path that begins with "." always opens a global namespace. If a local variable (e.g. "ivar") has the same name as a global variable (e.g. ".ivar"), the global variable is addressed.

2. The name of a "global variable list" can provide a unique definition for the namespace for the variables it contains. Thus, variables with the same name can be declared in different global variable lists and still be addressed uniquely by using the list name as prefix.

Example:

```
globlist1.ivar := globlist2.ivar;  
//ivar from GVL globlist2 is copied to ivar  
in GVL globlist1
```

3. Variables defined in the global variable list of a library integrated into the project can be addressed uniquely based on the syntax

```
"<NameSpace library>.<GVLname>.<VariableName>"
```

The next point includes further information on the namespace for libraries.

Example:

```
globlist1.ivar := lib1.globlist1.ivar  
//ivar from GVL globlist1 in library lib1 is  
copied to ivar in GVL globlist1.
```

- When a library manager is added to a library, a is also defined.

Thus, a library manager or a library variable can be uniquely addressed with

```
"<NameSpace library>.<BlockName|VariableName>"
```

If libraries are nested, note that the namespaces for all participating libraries have to be entered in sequence.

Example: If Lib1 is referenced by Lib0, the function block "fun" in Lib1 is addressed using

```
"Lib0.Lib1.fun"
```

```
:
```

```
ivar := Lib1.fun(4, 5); //return value from fun is  
copied to the variable ivar in the project
```

If "Publish all IEC symbols in the referencing project" has been enabled of the referenced library "Lib1", fun can also be addressed directly using "Lib0.fun".



Programming Reference

AT <Address>: Using the keyword "AT", the variable can be linked directly to a

In function blocks, variables with incomplete address information can also be declared. To use such a variable in a local instance, a corresponding entry for this variable has to be in the "variable configuration".

Data type: valid optionally extended by an  
";=<



Note that is possible. To enter declarations more quickly, use the can be used to affect code generation in the declarations of an object.

### 5.1.2 Recommendations for Assigning an Identifier

#### Recommendations for Assigning an Identifier, General Information

Identifiers are assigned during the of variables ( and when POU's and visualizations are created (function blocks: functions, function blocks, programs).

, the following rule is recommended for achieving the greatest possible consistency when assigning names:

- \_\_\_\_\_
- 
- 
- 
- 
- 
- 

#### Variable Names <Identifiers>

The naming of variables should be related to the **Hungarian notation**.

A short, meaningful description should accompany each variable, the "basic name". The first letter of each respective word in a basic name should be written in upper case, the others in lower case (example: FileSize).

If necessary, the compilation file can also be generated in other languages.

"Prefix(es)" corresponding to the variable data type are attached to the front of the basic name in lower case letters.

Data type	Lower limit	Upper limit	Information content	Prefix	Comment
BOOL	FALSE	TRUE	1 bits	x <sup>1)</sup>	
				b	Reserved

1) For Boolean variables, x was purposely chosen as a prefix to provide a separation from BYTE on one hand, and on the other, to accommodate the perspective of the IEC programmer (cf. address %IX0.0).



Programming Reference

Data type	Lower limit	Upper limit	Information content	Prefix	Comment
BYTE	16#00	16#FF	8 bits	by	Bit string, not for arithmetic operations
WORD	16#0000	16#FFFF	16 bits	w	Bit string, not for arithmetic operations
DWORD	16#00000000	16#FFFFFFFF	32 bits	dw	Bit string, not for arithmetic operations
LWORD	16#00000000 00000000	16#FFFFFFFF FF FFFFFF	64 bits	lw	Bit string, not for arithmetic operations
SINT	-128	127	8 bits	si	
USINT	0	255	8 bits	usi	
INT	-32.768	32.767	16 bits	i	
UINT	0	65.535	16 bits	ui	
DINT	-2.147.483.648	2.147.483.647	32 bits	di	
UDINT	0	4.294.967.295	32 bits	udi	
LINT	-2 <sup>63</sup>	2 <sup>63</sup> - 1	64 bits	li	
ULINT	0	2 <sup>64</sup> - 1	64 bits	uli	
REAL			32 bits	r	
LREAL			64 bits	lr	
STRING				s	
WSTRING				ws	
TIME				tim	
TIME_OF_DAY				tod	
DATE_AND_TIME				dt	
DATE				date	
ENUM			16 bits/32 bits	e	0...32767
POINTER				p	
ARRAY				a	

*Example*

```
bySubIndex: BYTE;
sFileName:  STRING;
udiCounter: UDINT;
```



## Programming Reference

In case of **nested declarations**, the prefixes are attached to each other in the order of the declaration:

### Example

---

```
pabyTelegramData: POINTER TO ARRAY [0..7] OF BYTE;
```

---

**Function block instances** and **variables** of user-defined data types have a short identifier for the FB or data type names as prefix.

### Example

---

```
cansdoReceivedTelegram: CAN_SDOTelegram;  
  
TYPE CAN_SDOTelegram :      (* prefix: sdo *)  
STRUCT  
  wIndex:WORD;  
  bySubIndex:BYTE;  
  byLen:BYTE;  
  aby: ARRAY [0..3] OF BYTE;  
END_STRUCT  
END_TYPE
```

---

**Local constants** (c) begin with the constant prefix "c" and an additional underscore "\_", followed by a type prefix and the variable name.

### Example

---

```
VAR CONSTANT  
  c_uiSyncID: UINT := 16#80;  
END_VAR
```

---

For **global variables** (g) and **global constants** (gc), an additional prefix and underscore are added to the library prefix.

### Examples:

---

```
VAR_GLOBAL  
  CAN_g_iTest: INT;  
END_VAR  
VAR_GLOBAL CONSTANT  
  CAN_gc_dwExample: DWORD;  
END_VAR
```

---

## Variable Names in IndraLogic 2G Libraries

Variable names in IndraLogic 2G are formed as in the description above, except that global variables and constants do not require library prefixes, since this function is replaced by the namespace.

### Example

---

```
g_iTest: INT; // Declaration  
CAN.g_iTest // Usage, call in program
```

---

## User-Defined Data Types (DUT)

### Structures:

The name of each structure data type consists of the library prefix (in the example: "CAN"), an underscore and a short, meaningful description of the structure (in the example: "SDOTelegram").

The corresponding prefix for variables created with this structure should follow as a comment directly after the colon.

### Syntax:



Programming Reference

<Libraryprefix>\_<Identifier>

*Example*

---

```

TYPE CAN_SDOTelegram :      (* prefix: sdo *)
STRUCT
  wIndex:WORD;
  bySubIndex:BYTE;
  byLen:BYTE;
  abyData: ARRAY [0..3] OF BYTE;
END_STRUCT
END_TYPE

```

---

**Enumeration values:**

Begin with the library prefix (in the example: "CAL") followed by an underscore and the identifier in upper case letters.

**Syntax:**

<Libraryprefix>\_<Identifier>



In past versions of IndraLogic, ENUM values > 16#7FFF have led to errors, since they were not automatically converted to INT. For this reason, ENUMs should always be defined with the correct INT values.

*Example*

---

```

TYPE CAL_Day : (
  CAL_MONDAY,
  CAL_TUESDAY,
  CAL_WEDNESDAY,
  CAL_THURSDAY,
  CAL_FRIDAY,
  CAL_SATURDAY,
  CAL_SUNDAY);
END_TYPE

// Declaration:
VAR
  eToday: CAL_Day;
END_VAR

```

---

**User-Defined Data Types (DUT) in IndraLogic 2G Libraries**

The library prefix is not used for DUT names in IndraLogic 2G libraries, since its function is replaced by the "namespace".

Likewise, enumeration values are also defined without a library prefix:

Example (from a **library with namespace CAL**):

*Type definition*

---

```

TYPE Day : (
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
  SUNDAY);
END_TYPE

```

---

*Declaration:*

---

```

eToday: CAL.Day;

```

---

*Use in the application:*

---

```

IF eToday = CAL.Day.MONDAY THEN

```

---





## POUs (Functions, Function Blocks, Programs, Actions...)

Programming Reference

Functions, function blocks and programs consist of the library prefix (in the example: "CAN"), an underscore and a short, meaningful name of the POU (in the example: "SendTelegram").

As with the variables, the first letter of each respective word in a basic name should be written in upper case, the others in lower case. It is recommended to use a combination of a verb and a noun in the POU name.

### Example

---

```
FUNCTION_BLOCK CAN_SendTelegram //prefix: can
```

---

The declaration contains a **short description** of the function block as a comment.

In addition, all **inputs** and **outputs** have comments.

For function blocks, the corresponding prefix for created instances should come right after the name as a comment.

**Actions** do not contain a prefix; only actions to be called internally only by the function block itself begin with prv\_ (private).

For compatibility reasons to previous versions of IndraLogic, each **function** has to have at least one transfer parameter.

**External functions** may not use structures as return values.

## POUs in IndraLogic 2G Libraries

The library prefix is not used for POU names in IndraLogic 2G libraries, since its function is replaced by the namespace.

**Method** names are created as action names.

Possible inputs for methods have to include comments. Likewise, the declaration should contain a short description of the method.

For methods, there are no limitations with regard to their implementation among external and internal libraries.

Interfaces should begin with the letter "I", e.g. "ICANDevice"

## Identifiers for Visualizations



---

Note that a visualization does currently not have the same name as another function block in the project, since this would lead to problems when switching visualizations.

---

## 5.1.3 Variable Initialization

is 0. User-defined initialization values can also be entered in the declarations for each variable and each data type.

The user-defined initialization starts with the assignment operator "!=" and can consist of any valid ST expression refer to

Thus, the initial value can be defined using constants, other variables or functions.

The programmer has to ensure that a variable "x" used to initialize another variable, "y" is also declared.

*Examples for valid variable initializations:*

---

```
VAR  
var1:INT := 12; // Integer variable with initial value 12
```

## Programming Reference

```
x : INT := 13 + 8;           // Initial value is defined by a term of constants
y : INT := x + fun(4);      // Initial value is defined by a term
                             // that contains a function call;
                             // in these cases please observe the order!

z : POINTER TO INT := ADR(y); // Not described in the IEC61131-3
                             // standard: Initial value is defined by an address function;
                             // However, in this case the pointer is not initialized
                             // during an online change!

END_VAR
```

*A description on the initialization can be found under*

- 
- 
- 
- 
- 



Starting from compiler version 3.3.2.0, variables from the namespace are always initialized before the local variables of a POU.

## 5.1.4 Any Expressions for Variable Initialization

A variable can be initialized with any desired

Other variables from the same namespace as well as function calls can also be used.

If a variable is initialized with another variable, the other variable should also be initialized.

*Examples for valid variable initializations:*

```
VAR
x : INT := 13 + 8;
y : INT := x + fun(4);

//Warning: The pointer is not initialized in the case of an online change!
z : POINTER TO INT := ADR(y);
END_VAR
```

## 5.1.5 Declaration Editor

The declaration editor is a text editor for the variable

Usually, it is used with language editors.

*Also refer to*

- 

## 5.1.6 "Auto Declare" Function

Under **Tools ▶ Options ▶ IndraLogic2G ▶ Smart coding** a setting can be made to open the "Auto Declare" dialog automatically as soon as a variable that is not declared is entered in the statements into an editor and <Enter> is pressed.



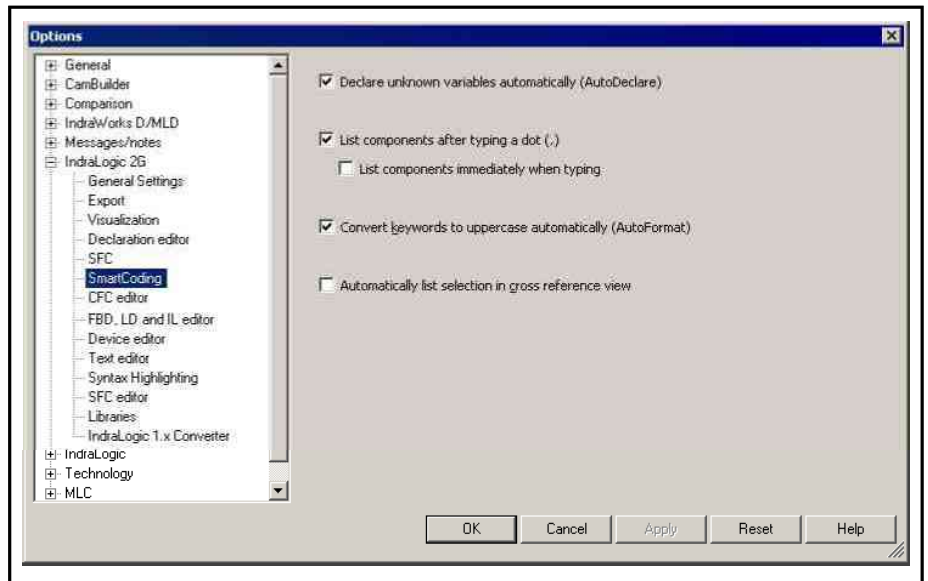


Fig.5-1: "SmartCoding" options

The "Auto Declare" dialog supports the variable

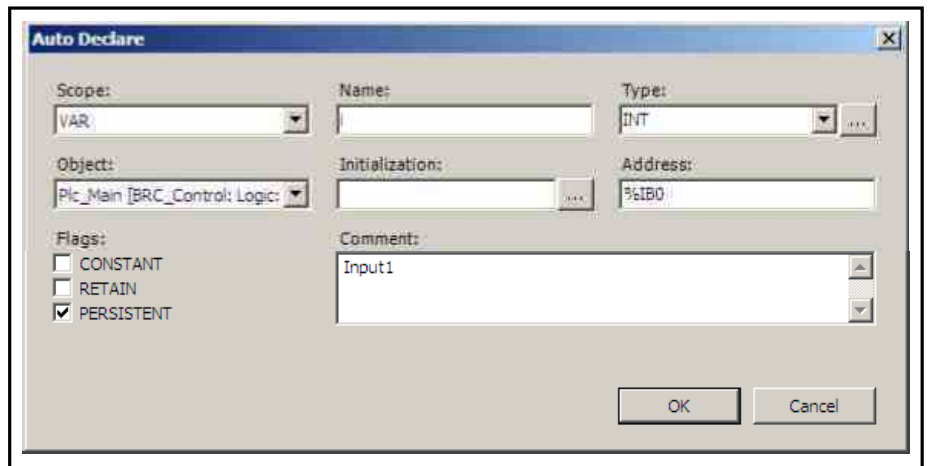


Fig.5-2: "Auto Declare" dialog

The dialog to declare a variable can also be opened explicitly using the command (usually found in the "Edit" menu).

This command or the shortcut <Shift>+<F2> also opens the dialog, even if a variable that is already declared is selected in the statements of an editor.

A description of automatic declaration is found under

### 5.1.7 Short Form Mode

In the declaration editor, as in other text editors for variable declaration, the "short form mode" for the input is supported.



The statement in the editor does not support the "short form mode".

This mode is enabled if a declaration line is completed with the shortcut <Ctrl>+<Enter>.

In the short form mode, use short forms to enter a declaration instead of typing in all the details.



### Programming Reference

The following short forms are supported:

- All identifiers except the last identifier in a line become variable identifiers of a declaration.
- The declaration data type is specified by the last identifier in the line. The following applies here:

---

B or BOOL	results in	BOOL
I or INT	results in	INT
R or REAL	results in	REAL
S or string	results in	STRING

- If a data type cannot be specified using these rules, BOOL is automatically used as the data type and the last identifier in the line is not used as data type (see below: example )1).
- Depending on the type of declaration, each constant entered becomes an initialization or a string length specification (see below, examples (2) and (3)).
- An address (as in %MD12) is automatically extended with the AT attribute (see below, example (4)).
- A text following a semicolon (;) becomes a comment (see below, example (4)).
- All other characters in the line are ignored (see below, exclamation point in example (5)).

Examples:

---

Short form	Resulting declaration
(1) O	A: BOOL;
(2) A B I 2	A, B: INT := 2;
(3) ST S 2; A string	ST:STRING(2); // A string
(4) X %MD12 R 5; Real Number	X AT %MD12: REAL := 5.0; // Real Number
(5) B !	B: BOOL;

## 5.1.8 AT Declaration

can be linked to a specific input, output or memory address of the control configured in the Project Explorer.

### Syntax:

<Identifier> AT <Address>: <DataType>;

that corresponds to the currently active control configuration in the Project Explorer.

This allows to give a meaningful name to the address. Modifications with regard to the incoming or outgoing signal may be carried out only at one position (e.g. in the declaration).

Note the following when assigning a variable to an address:

- Write access to variables to which an input was assigned is not possible.



## Programming Reference

- AT declarations can only be carried out for local and global variables, not for input and output function block variables.
- If AT declarations are used with structure or function block components, all instances use the same memory which is as using static variables in classic programming languages such as "C".
- The memory layout of structures depends on the target system.

### Examples:

---

```
counter_heat7 AT %QX0.0: BOOL;  
lightcabinetimpulse AT %IX7.2: BOOL;  
download AT %MX2.2: BOOL;
```

---



If Boolean variables are assigned to a BYTE, WORD or DWORD address, they assign TRUE or FALSE to an entire byte, not only to the first bit after the offset!

---



I/O modules of the same type on the respective slot and in the same sequence are required for automatic address generation resulting in the same assignment and thus to the connection of the correct inputs and outputs.

Please check this carefully if the control is replaced.

---

Note that a variable can also be assigned to an address when configuring the modules.

## 5.1.9 Keywords

Keywords can be entered in upper case, lower case and mixed.

The following character strings are reserved as keywords, i.e. they cannot be used as for variables or POU's:

ACTION (only used in export format)

BY



Programming Reference

CONTINUE, page 396

COS, page 605

Date, page 555

DINT, page 554

DIV, page 571

DO, page 394

DT, page 555

DWORD, page 554

ELSE, page 393

ELSIF, page 393

END\_ACTION (only used in export format)

END\_FUNCTION (only used in export format)

END\_FUNCTION\_BLOCK (only used in export format)

END\_PROGRAM (only used in export format)

END\_STRUCT

END\_TYPE

JMPC

JMPCN

LD

LDN



LN, page 603  
LOG, page 604  
LREAL, page 554  
LT, page 584  
LTIME, page 555  
LWORD, page 554  
MAX, page 582  
METHOD, page 45,

OF

R

READ\_ONLY  
READ\_WRITE

RET

RETC  
RETCN

S

SIZEOF



## Programming Reference

[SQRT, page 603](#)

ST

STN

TO

TYPE

VAR\_ACCESS (only used in special circumstances, depending on hardware)

XOR

XORN

In addition, all conversion operators as listed in the input assistance are treated as keywords.

### 5.1.10 Local Variables (VAR)

All local variables of a function block are between the keywords **VAR** and **END\_VAR**. External access to local variables is not possible.

**VAR** can be extended by an .



*Example*

---

```
VAR
  iLoc1: INT;
END_VAR
```

---

### 5.1.11 Input Variables (VAR\_INPUT)

Variables used as input variables for a function block are declared between the keywords **VAR\_INPUT** and **END\_VAR**.

This means that when calling the function block, a value can be transferred to these variables.

**VAR\_INPUT** can be extended by an **IN\_OUT**.

*Example*

---

```
VAR_INPUT
  iIn1: INT;
END_VAR
```

---

### 5.1.12 Output Variables (VAR\_OUTPUT)

All output variables of the function block are declared between the keywords **VAR\_OUTPUT** and **END\_VAR**.

This means that the values of these variables can be returned to the function block called. They can be queried and used there.

**VAR\_OUTPUT** can be extended by an **IN\_OUT**.

*Example*

---

```
VAR_OUTPUT
  iOut1: INT;
END_VAR
```

---

**Output variables in functions and methods:**

According to the IEC 61131-3 standard, functions and methods can have additional outputs. These have to be assigned when the function is called:

*Example*

---

```
fun(iIn1 := 1, iIn2 := 2, iOut1 => iLoc1, iOut2 => iLoc2);
```

---

The return value of the function "fun" is additionally calculated and transferred to its outputs.

### 5.1.13 Input/Output Variables (VAR\_IN\_OUT)

Variables used as input/output variables for a function block are declared between the keywords **VAR\_IN\_OUT** and **END\_VAR**.



For input/output variables, the value of the transferred variable is changed directly ("Transfer as pointer", Call by reference).

That means that the input value for such variables may not be a constant.

Thus, the VAR\_IN\_OUT variables for a function block cannot be read or described using

```
<FBinstance>.<InputOutputVariable>
```

from an external location.

---

## Programming Reference

*Example*

```
VAR_IN_OUT  
iInOut1: INT;  
END_VAR
```

## 5.1.14 Global Variables (VAR\_GLOBAL)

Variables, constants or remanent variables that should be known across the entire project can be declared as global variables.



A variable declared locally in a function block and with the same name as a global variable has priority in the function block.



Starting from compiler version 3.3.2.0, variables from thr are always initialized before the local variables of a POU.

The variables are declared locally between the keywords **VAR\_GLOBAL** and **END\_VAR**.

**VAR\_GLOBAL** can be extended hv an

A variable is detected as a global variable if a dot is placed in fron of the variable name, e.g. ".iGlobVar1".



For more detailed information on the multiple usage of variable names, on the operator for the global namespace "." and name-spaces, see

Global variable lists can be used to manage global variables in a project.

A "GVL" can be added as an object in the Project Explorer using the command.

## 5.1.15 Temporary Variables (VAR\_TEMP)

The temporary variable functionality is an extension with regard to the IEC 61131-3 standard.

Temporary variables **are re-initialized every time the function block is called.** **VAR\_TEMP** declarations can only be made in programs and function blocks.

The variables can only be accessed in the statement of the program or function block.

The variables have to be declared between the keywords **VAR\_TEMP** and **END\_VAR**.

## 5.1.16 Static Variables (VAR\_STAT)

The static variable functionality is an extension with regard to the IEC 61131-3 standard.

Static variables can be used in function blocks, functions and methods. They have to be declared between the keywords **VAR\_STAT** and **END\_VAR** and **are initialized the first time the respective function block is called.**

Static variables can only be accessed in the namespace in which they are they keep their value even after the function block is exited again. They can be used as counters for function calls for example.

**VAR\_STAT** can be extended by an

### 5.1.17 External Variables (VAR\_EXTERNAL)

Programming Reference

External variables are imported into a function block. The variables have to be declared locally between the keywords **VAR\_EXTERNAL** and **END\_VAR**.



If a **VAR\_EXTERNAL** is not declared in a GVL, an error message is output.



It is not required in IndraLogic to declare variables as external. The keyword is intended to ensure compliance with IEC 61131-3.

#### Example

```
VAR_EXTERNAL
iVarExt1: INT :=12;
END_VAR
```

### 5.1.18 Attribute Keywords for Variable Types

The following keywords can be used to add the corresponding attributes in the of variable types.

**RETAIN:** See of type **RETAIN**.

**PERSISTENT:** See of type **PERSISTENT**.

**CONSTANT:** See

### 5.1.19 Access Variables

### In preparation ###

### 5.1.20 Remanent Variables (VAR RETAIN, VAR PERSISTENT)

Remanent variables can retain their value for the entire program runtime. They are declared as pure "retain variables" or "persistent variable" or as a combination of retain and persistent.

Each has its own memory area used for management.

The type of declaration chosen specifies the degree of "resistance" a remanent variable has in case of a reset, download or computer reboot.

In practice, the combination of both types is most often requested (**PERSISTENT**).



When an **IndraLogic 1.x project** is opened, the declarations of retain variables remain effective and are not changed, but the declarations of persistent variables have to be revised or recreated.

An individual global variable list has to be created for the project! Refer to

For further information, refer to

- 
- 
- 

#### Retain variables

The management of variables declared as retain variables depends on the target system: Usually, their are managed in their **own memory space**. They are identified by the keyword **RETAIN** in a function block or in a global variable list (GVL) in the IndraLogic project.



Programming Reference

*Example*

```
VAR RETAIN
  iRem1 : INT;
END_VAR
```

Retain variables keep their value after an uncontrolled shutdown or in response to the online command as well as after switching the control off and on normally (reboot).

When the program restarts, the saved values are used for further processing. All other variables are re-initialized in this case, either with their initialized values or with default initializations.

Use case:

A counter in a production facility that is to continue counting after power failure.

However, retain variables are re-initialized at a a or a new program download.

The retain property can be combined with the persistent property. To do this, refer to the below.



If a local variable is declared as RETAIN in a **program**, **this exact variable** is saved in the retain area (like a global retain variable).

If a local variable is declared as RETAIN in a **function block**, the **entire instance** of this function block is saved in the retain area (all function block data), although only the declared retain variable is treated as such.

If a local variable is declared in a **function** as RETAIN, it has no effect! The variable is not saved in the retain area! If a local variable is declared as PERSISTENT in a function, it has also no effect!



The memory space for retain and persistent variables is limited to 127 KB each.

**Persistent variables**

Currently, persistent variables are treated as persistent retain variables. In this case, the values are kept even at a control reboot. See below.

Persistent variables are identified by the keyword "PERSISTENT" (**VAR\_GLOBAL PERSISTENT**). They are only re-initialized during a reboot or of the control.

In contrast to the retain variables, they keep their value after a download. A use case for "persistent retain variables" might be a counter for operating hours which is supposed to continue counting after a power failure or download. See below

Persistent variables are treated as follows and thus different as in IndraLogic1.x:

Persistent variables can ONLY be declared in a **special global variable list** of the object type that is part of an application. There is only "ONE" such list per application.



From V3.3.0.1, a declaration with "VAR\_GLOBAL PERSISTENT" has the same effect as a declaration with "VAR\_GLOBAL PERSISTENT RETAIN" or "VAR\_GLOBAL RETAIN PERSISTENT".

Programming Reference

Like retain variables, persistent variables are managed in their own memory space

*Example*

```
VAR GLOBAL PERSISTENT RETAIN
  iVarPers1 : DINT;
  bVarPers : BOOL;
END_VAR
```



Currently, only **global** persistent variables can be created

The target system has to provide one separate memory space per application for the persistent variable list.

Each time the **application is loaded**, the persistent variable list on the control is compared with that in the project. The variable list on the control is identified by the application name among others. In case of **inconsistencies**, the user is prompted to initialize all persistent variables before the download. Inconsistencies occur due to renaming, deletion or other modifications to existing persistent variable declarations.



Thus, carefully consider **each change in the declaration part** of the persistent variable list and the effects on a re-initialization subsequently asked.

**New declarations** can only be added to the end of the list, but they are identified as new while loading. Thus, re-initializing the entire list is not necessary.

Overview table on the behavior of remanent variables

After online command	VAR	VAR RETAIN	VAR PERSISTENT VAR RETAIN PERSISTENT VAR PERSISTENT RETAIN
Reset warm	-	x	x
Reset cold	-	-	x
Reset origin	-	-	-
Loading (= download)	-	-	x
Online change	x	x	x
Reboot control	-	x	x

x Value is kept  
- Value is re-initialized

Fig.5-3: Overview table on the behavior of remanent variables

**⚠ WARNING**

**Dangerous state due to PERSISTENT data in the remanent data memory**

The device description can define that the PERSISTENT data be mapped in the remanent data memory. In this case, the values are kept even at control reboot!

Make absolutely sure that the PERSISTENT data cannot cause damage at a reboot.

Programming Reference



The maximum memory space for retain and persistent variables is limited to 127 KB each.

## 5.1.21 Constants (VAR CONSTANT), Typed Constants

**CONSTANT** Constants are identified by the keyword **CONSTANT**. They can be declared locally or globally.

*Syntax:*

```
VAR CONSTANT  
<Identifier> : <Type> := <initialization>;  
END_VAR
```

*Example*

```
VAR CONSTANT  
  c_iCon1:INT:=12;  
END_VAR
```

In the description of the **CONSTANT**, a list of possible values can be found.

Typed constants can also be used:

### Typed constants (typed literals)

Normally, the smallest possible data type is used when calculating with IEC constants. If another data type is to be used, this can be done with constants of a specific data type (typed literals) to which a specific type is assigned.

In this case, the constants do not have to be declared explicitly as "VAR CONSTANT" in the declaration. The constants are provided with a prefix that specifies the type:

It is written as follows:

*Syntax:*

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

<Type> indicates the desired data type.

Possible inputs:

BOOL, SINT, USINT, BYTE, INT, UINT, WORD, DINT, UDINT, DWORD, LINT, ULINT, LWORD, REAL, 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*

```
iVar1:= DINT#34 ;
```

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

Constants of a specific data type can be used anywhere normal constants can be used.




Furthermore, the system is also provided with the following constants: TIME#, T#, DATE#, D#, TIME\_OF\_DAY#, TOD#, DATE\_AND\_TIME#, DT#

and corresponding constants CHAR#, WCHAR#, STRING#, WSTRING#.





**Constants in online mode** Programming Reference  
If the default setting "Replace constants" is selected, constants in online mode in the declaration or monitoring window are indicated by a preceding symbol  in the value column. In this case, they cannot be accessed by forcing or writing for example.



**Replace constants:** This option is selected by default. This of scalar type (not for strings, arrays and structures). In online mode, constants are labeled in the or in the by a symbol preceding the value. In this case, access via an ADR operator, forcing and writing is not possible. If the option is disabled, the constant can be accessed, but the computing time increases.

### 5.1.22 Variable Configuration (VAR\_CONFIG)

The variable configuration can be used to "map" function block variables to the process image, i.e. to assign the variables to the device I/Os without having at the variable declaration in the function block. In this case, the address is assigned centrally in a **global VAR\_CONFIG list** below the application for all function block instances of the application.

**Incomplete addresses** are assigned to the function block variables in the declaration between the keywords "VAR" and "END\_VAR" for this purpose.

These addresses are identified by a "".

*Syntax:*

```
<Name> AT %<I|Q>* : <Data type>;
```

*Example for the assignment of incompletely defined addresses:*

```
FUNCTION_BLOCK locio
VAR
  xLocIn AT %I*: BOOL := TRUE;
  xLocOut AT %Q*: BOOL;
END_VAR
```

Two local I/O variables - a local input variable (%I\*) and a local output variable (%Q\*) - are defined here.

The **final definition** of the addresses is then made in the "variable configuration" in a global variable list:



command to insert an object of the type "global variable list" (GVL) into the Project Explorer below the desired application.

In this GVL, enter the declarations of the instance variables with the exact addresses between the keywords **VAR\_CONFIG** and **END\_VAR**.

The keyword **VAR\_GLOBAL** is replaced by the keyword **VAR\_CONFIG**.

The instance variables have to be specified with the complete instance path in which the individual POU and instance names are both separated by a dot each.

## Programming Reference

whose class (input/output) matches with that of the incomplete specification (%I\*, %Q\*) in the function block. The data type has to match as well.

### Syntax:

---

```
<instance variable path> AT %<I|Q><location> : <data type>;
```

---

Configuration variables whose instance paths are invalid since the instance does not exist, are reported as errors.

But an error is also output if there is no address configuration present for an instance variable declared with an incomplete address.

### Example for a variable configuration:

The following usage of the function block "locio" exists in a program (see example above).

#### Declaration:

---

```
PROGRAM PlcProg
VAR
  locioVar1: locio;
  locioVar2: locio;
END_VAR
```

---

Then, a correct variable configuration would look as follows for example:

#### Configuration:

---

```
VAR_CONFIG
  PlcProg.locioVar1.xLocIn AT %IX1.0 : BOOL;
  PlcProg.locioVar1.xLocOut AT %QX0.0 : BOOL;
  PlcProg.locioVar2.xLocIn AT %IX1.0 : BOOL;
  PlcProg.locioVar2.xLocOut AT %QX0.3 : BOOL;
END_VAR
```

---



Modifications of variables directly assigned to the I/O addresses are displayed immediately in the process image, while modifications of variables "mapped" using a variable configuration are displayed when the task responsible is completed.

---

## 5.1.23 User-Defined Data Types

In addition to the standard data types, user-defined data types can also be used.

On declaration and initialization, refer to the description of (Data Unit Type).

## 5.1.24 Extendable Functions, PARAMS

### In preparation ###

## 5.1.25 FB\_init, FB\_reinit Methods

**FB\_init** The "FB\_init" method replaces the INI operator used in IndraLogic 1.x.

A method named FB\_init is a special for a function block.

It can be declared explicitly but is always created implicitly as well. Thus, it can be controlled for every function block.

The FB\_init method contains an **initialization code** for the function block based on the declarations in the declaration part of the function block.

## Programming Reference

If the init method is also explicitly declared, the implicit initialization code is added to the explicitly created method.

The programmer can then add further initialization code.



When the execution reaches the user-defined initialization code, the function block has already been completely initialized using the implicit initialization code.

The FB\_init method is called for all declared instances after a download. **Attention:** For online changes, the most recent values overwrite the initialization values.

To call in sequence in case of inheritance, refer to: See

Also refer to the option to call a function block method automatically after the initialization via FB\_init: Attribute

### Interface of the FB\_init method:

```
METHOD fb_init : BOOL
VAR_INPUT
  bInitRetains: BOOL; // if TRUE, the Retain variables are initialized
                        // (warm start / cold start)
  bInCopyCode : BOOL; // if TRUE, the instance is copied
                        // into the Copy-Code (Online Change)
END_VAR
```

The return value is not used.



An "fb\_exit" method and the resulting processing sequence can also be used. See

### User-defined input:

Additional inputs can be defined in an FB\_init method. These have to be assigned in the declaration of the function block instance.

#### Example of an init method for a function block called "serialdevice":

```
METHOD fb_init : BOOL
VAR_INPUT
  bInitRetains : BOOL; // Initialization of Retain variables
  bInCopyCode : BOOL; // Instance is copied into the Copy-Code
  nCOMnum : INT; // additional input: Number of the COM
                // interfaces that is leading
END_VAR
```

#### Declaration of the function block "serialdevice":

```
COM1 : serialdevice(nCOMnum:=1);
COM0 : serialdevice(nCOMnum:=0);
```

### FB\_reinit

If a method is declared with the name "FB\_reinit" for a function block instance, it is called if the **instance is copied** (for example at an online change after changes were made in the function block declaration).

The method re-initializes the instance module generated by the copy code. A re-initialization may be desirable, since after copying, the original instance data is written on the newly created instance, but the original values are the desired ones. Note that the FB\_reinit method **has to be explicitly declared** in contrast to the FB\_init method. If the basic implementation of the function



Programming Reference

block should be re-initialized, the FB\_reinit has to be called explicitly for this function block.

The FB\_reinit method has no inputs.

To call in sequence in case of inheritance, refer to:

### 5.1.26 FB\_exit Method

The method named "FB\_exit" is a special method for a function block. It has to be declared explicitly. There is no implicit declaration.

The "exit" method - if available - is called for all declared instances of the function block before a new download or during online changes for all new or deleted instances.

Interface of the FB\_exit method: There is only one obligatory parameter:

*Interface of the FB\_exit method:*

---

```

METHOD fb_exit : BOOL
VAR_INPUT
  bInCopyCode : BOOL; // if TRUE, the exit method is called
                    // to leave the instance, which is copied
                    // afterwards (Online Change).
END_VAR

```

---

See also the [FB\\_reinit](#) and the following execution sequence:

1. exit method: exit old instance

```
old_inst.fb_exit(bInCopyCode := TRUE);
```

2. init method: initialize new instance:

```
new_inst.fb_init(bInitRetains:= FALSE, bInCopyCode:= TRUE);
```

3. Copying the function block values (copy code):

```
copy_fub(&old_inst, &new_inst);
```

In case of inheritance, the following call sequence applies additionally (the following is assumed for the POU's named in this list as example).

SubFB EXTENDS MainFB and SubSubFB EXTENDS SubFB):

*Call sequence:*

---

```

fbSubSubFb.FB_Exit(...);
fbSubFb.FB_Exit(...);
fbMainFb.FB_Exit(...);
fbMainFb.FB_Init(...);
fbSubFb.FB_Init(...);
fbSubSubFb.FB_Init(...);

```

---

*for FB\_reinit:*

---

```

fbMainFb.FB_reinit(...);
fbSubFb.FB_reinit(...);
fbSubSubFb.FB_Init(...);

```

---

### 5.1.27 Pragma Statements

#### Pragma Statements, Overview

A pragma statement is used to affect the properties of one or more variables with regard to compiling or precompiling (preprocessor). That means that a pragma affects the code generation.



## Programming Reference

For example, a pragma statement can specify if a variable is initialized, displayed in online mode, added to the manager. During the compilation, that specify how a variable is to be interpreted under certain conditions can be used. These conditional pragmas can also be entered as "compiler definitions" in the

or kept invisible in the library manager. During the compilation, can be forced. that specify how a variable is to be interpreted under certain conditions can be used. These conditional pragmas can also be entered as "compiler definitions" in the of an object. A pragma can be inserted in a separate line or together with the code in an implementation or declaration line. In the FBD/LD/IL editor, call the "Add jump label" command first and then the "Label:" entry in the label text field has to be replaced by the corresponding pragma statement.



1. If a label and a pragma should be used, enter the pragma first and then the jump label.
2. A pragma statement is given in curly brackets.
3. **Lower-case letters** are currently required for pragma statements.
4. If the compiler cannot interpret the statement text, the entire pragma is treated as a comment and is not read. However, a warning is output.

### *Syntax:*

```
{ <instruction text> }
```

The opening bracket may be placed directly after a variable name. Opening and closing brackets always have to be placed in the same line.

Depending on the type and content of a pragma, it affects either:

- the line in which it is located
- or all following lines until it is canceled by a corresponding pragma or
- until the same pragma is executed with other parameters or
- the end of the file is reached.

A file is a: declaration part, statement part, global variable list, type declaration.

Possible pragma types:

### In preparation ###: Pragmas as in IndraLogic 1.x.

## Message Pragmas

Message pragmas can be used to force messages to be output in the message window during the compilation (project build).

The pragma statement can be inserted into a separate or an existing line in the text editor of a POU and considered when the project is compiled.

There are four types of message pragmas:

Programming Reference

Pragma	Message type
{text 'textstring'}	<b>Text</b> ⓘ: The text specified, "Textstring", is output.
{info 'textstring'}	<b>Information</b> ⓘ: The text specified, "Textstring", is output.
{warning digit 'text-string'}	<b>Warning</b> ⚠: The text specified, "Textstring", is output. <b>###</b> In preparation <b>###</b> . The specified number indicates the warning level (between 1 and 5). In contrast to a "data type global" _____, the warning is locally defined for the current position.
{error 'textstring'}	<b>Error</b> ❌: The text specified, "Textstring", is output.

☞ For the message types "information", "warning" or "error message" to move to the source position of the message, i.e. the position at which the pragma is positioned in the POU.

*Declaration and implementation in the ST editor:*

```

VAR
  ivar : INT; {info 'TODO: should get another name'}
  bvar : BOOL;
  arrTest : ARRAY [0..10] OF INT;
  i: INT;
END_VAR

arrTest[i] := arrTest[i]+1;
ivar:=ivar+1;
{warning 'This is a warning'}
{text 'Part xy has been compiled completely'}
  
```

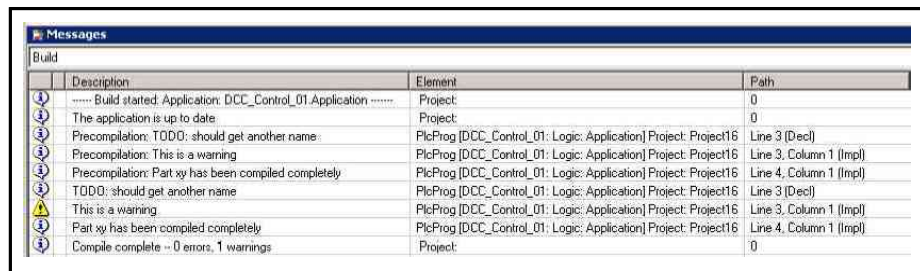


Fig.5-4: Example for an output in the message window

**Attribute Pragma**

**Attribute Pragma, General Information**

Attribute pragmas can be added to a signature to influence the compilation or precompilation. All variables in the declaration is one signature.

There are user-defined attributes used with \_\_\_\_\_ for a description, but also refer to the predefined attribute pragmas listed below:







Programming Reference

**Example: POU's, actions** *Attribute 'vision' for function fun1:*

```
{attribute 'vision'}
FUNCTION fun1 : INT
VAR_INPUT
  i : INT;
END_VAR
VAR
  END_VAR
```

**Example: variables** *Attribute 'docount' for variable ivar:*

```
PROGRAM Plc_Main
VAR
  ivar:INT; {attribute 'docount'};
  bvar:BOOL;
END_VAR
```

**Example: types** *Attribute 'atype' for data type DUT\_1:*

```
{attribute 'atype'}
TYPE DUT_1 :
STRUCT
  a:INT;
  b:BOOL;
END_STRUCT
END_TYPE
```

[To use conditional pragmas, page 546.](#)

**Attribute 'call\_after\_init'**

This pragma can be used to define a method to be implicitly called after the initialization of a function block instance.

For this purpose, the attribute has to be added to the function block as well as to the method (due to reasons of performance).

and after the variable values of an initialization expression became valid in the instance declaration. This functionality is supported from the compiler version >= 3.4.1.0.

*Syntax:*

```
{attribute 'call_after_init'}
```

**Example**

*With the following function block definition:*

```
{attribute 'call_after_init'}
FUNCTION_BLOCK FB
... <functionblock definition>
```

*and the method definition:*

```
{attribute 'call_after_init'}
METHOD FB_AfterInit
... <method definition>
```

*... a declaration such as:*

```
inst : FB := (in1 := 99);
```

*... is implemented in the following code processing:*

```
inst.FB_Init();
inst.in1 := 99;
inst.FB_AfterInit();
```

Thus, the FB\_AfterInit can react on the user-defined initialization.



### Attribute 'displaymode'

The display mode of an individual variable can be defined using this pragma.

This specification overwrites the global setting for the display of all monitoring variables made using the commands in the display mode submenu (located in the debug menu by default).

The pragma has to be located in the line above the line which contains the variable declaration.

#### Syntax:

---

```
{attribute 'displaymode':='<displaymode>'}
```

---

The following definitions are possible:

#### for a display in binary format:

---

```
{attribute 'displaymode':'bin'}  
{attribute 'displaymode':'binary'}
```

---

#### for a display in decimal format:

---

```
{attribute 'displaymode':'dec'}  
{attribute 'displaymode':'decimal'}
```

---

#### for a display in hexadecimal format:

---

```
{attribute 'displaymode':'hex'}  
{attribute 'displaymode':'hexadecimal'}
```

---

#### Example

---

```
VAR  
  {attribute 'displaymode':'hex'}  
  dwVar1: DWORD;  
END_VAR
```

---

### Attribute 'enable\_dynamic\_creation'

### In preparation ###

### Attribute 'external\_name'

The pragma specifies the name of an externally implemented function or function block in the runtime environment.

It can only be used for functions and function blocks.

#### Syntax:

---

```
{attribute 'external_name':='<implementation_name>'}
```

---

#### Example

---

```
{attribute 'external_name':'myFunctionBlockImplementationName'}  
FUNCTION_BLOCK MyFunctionBlock  
...
```

---

### Attribute 'expandfully'

The components of an array used as input variable for referenced visualizations in the of the visualization can become visible using this pragma.

#### Syntax:

---

```
{attribute 'expandfully'}
```

---

#### Example:

Programming Reference

Visualization **visu** is to be inserted into a frame in the visualization **visu\_main**. **arr** is defined as input variable in the "visu" interface editor and is thus available for assignments in the property dialog of the frame in **visu\_main**.

To provide the individual components of the array in this "Property" dialog, the 'ExpandFully' attribute has to be added to the interface editor of **visu** directly in front of **arr**.

*Declaration in the interface editor of "visu":*

```
VAR_INPUT
{attribute 'expandfully'}
arr : ARRAY[0..5] OF INT;
END_VAR
```

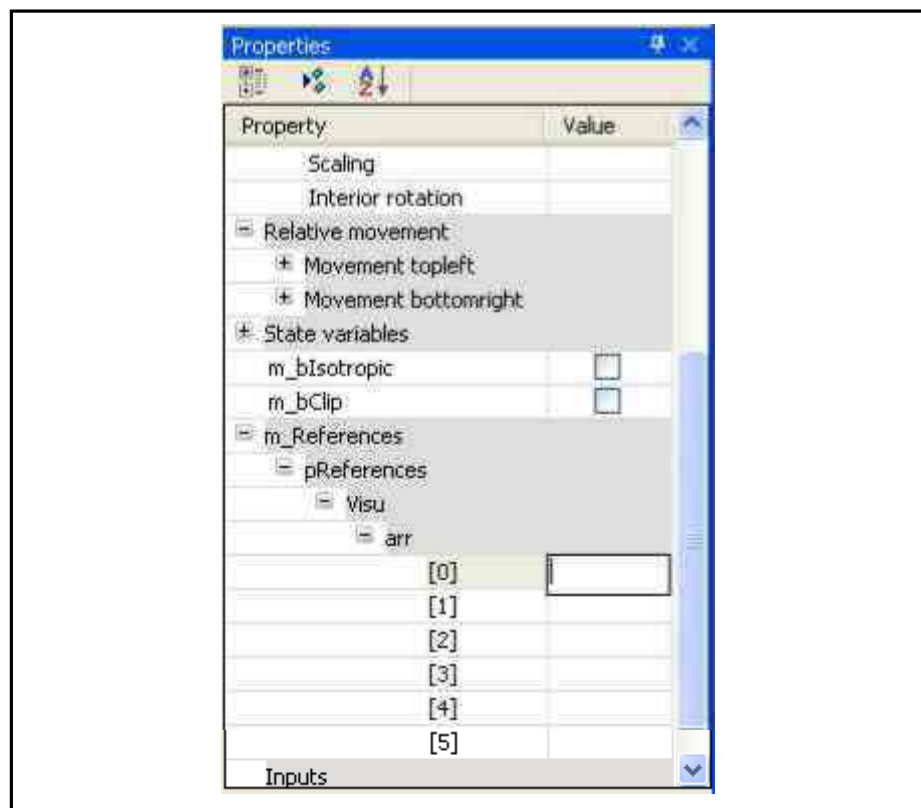


Fig.5-5: "Properties" dialog for the frame in "visu\_main"

**Attribute 'global\_init\_slot'**

This pragma can only be used for signatures.

The sequence for the initialization of variables from global variable lists is not specified from the very beginning. Sometimes, however, specifying such a sequence is necessary for example if the variables of a list of variables depend on another list.

In this case, the pragma for specifying the sequence during the global initialization can be used.

*Syntax:*

```
{attribute 'global_init_slot' := '<value>'}
```

The placeholder <value> has to be replaced by an integer value determining the significance in the initialization sequence.

The default value is 5000.

Programming Reference

A lower value causes an earlier initialization. If several signatures have the same significance for the attribute 'global\_init\_slot', the sequence of their initialization remains unspecified.

Example: The example project contains two global variable lists, GVL\_1 and GVL\_2:



Fig.5-6: Two global variable lists

The global variable "A" is part of the global variable list GVL\_1:

```
{attribute 'global_init_slot' := '300'}
VAR_GLOBAL
  A: INT:= 1000;
END_VAR
```

The initialization values of the variables "B" and "C" from GVL\_2 depend on the variable "A".

Initialization values of the variables "B" and "C":

```
{attribute 'global_init_slot' := '350'}
VAR_GLOBAL
  B : INT:=A+1;
  C : INT:=A-1;
END_VAR
```

If the attribute 'global\_init\_slot' of the global variable list GVL\_1 is set to 300 - that is the lowest initialization value in the example - it is ensured that the expression "A+1" is well defined when "B" is initialized.

**Attribute 'hide'**

Use the pragma to prevent that variables or even entire signatures become visible in the "List component" functionality, in the input assistance or in the declaration part in online mode.

Only the variable directly following the pragma becomes invisible.

Syntax:

```
{attribute 'hide'}
```

To hide all local variables of a signature, use the attribute

**Example:**

Function block myPOU with attribute "hide":

```
FUNCTION_BLOCK myPOU
VAR_INPUT
  a:INT;
  {attribute 'hide'}
  a_invisible: BOOL;
  a_visible: BOOL;
END_VAR
VAR_OUTPUT
  b:INT;
END_VAR
VAR
END_VAR
END_VAR
```

In the main program, two instances of the function block myPOU are defined:



## Programming Reference

### *Two instances of the function block myPOU in the program Plc\_Main:*

```
PROGRAM Plc_Main
VAR
    POU1, POU2: myPOU;
END_VAR
```

While the input value for POU1 is implemented for example, the "List component" function that opens when jogging "POU1" displays (in the implementation part of PLC\_PRG) the variables "a", "a\_visible" and "b", but not the hidden variable "a\_invisible".

#### **Attribute 'hide\_all\_locals'**

Use this pragma to hide local variables of a signature in the display of the "List component" functionality and the input assistant of the online monitoring in the declaration part.

The effect of this attribute is identical to that of the usage of the attribute on each local variable.

#### *Syntax:*

```
{attribute 'hide_all_locals'}
```

#### **Example:**

The function block myPOU is implemented using the attribute:

#### *Function block myPOU with attribute:*

```
{attribute 'hide_all_locals'}
FUNCTION_BLOCK myPOU
VAR_INPUT
    a:INT;
END_VAR
VAR_OUTPUT
    b:BOOL;
END_VAR
VAR
    c,d:INT;
END_VAR
```

In the main program, two instances of the function block myPOU are defined:

#### *Instances of the function block*

```
PROGRAM Plc_Main
VAR
    POU1, POU2: myPOU;
END_VAR
```

While an input value for POU1 is implemented, the input assistance (Intelligence), which opens when typing "POU1" (in the statements of Plc\_Main), displays the variables "a" and "b", but not the hidden local variables "c" or "d".

#### **Attribute 'initialize\_on\_call'**

This pragma can only be used for input variables.

An input variable of a function block that has this attribute is initialized every time the function block is called. If an input expects a pointer and if it was removed during an online change, this input is set to ZERO.

#### *Syntax:*

```
{attribute 'initialize_on_call'}
```

#### **Attribute 'init\_namespace'**

A STRING or WSTRING variable provided with this pragma is initialized with the current namespace.





To use this pragma, the additional attribute used for the STRING variable.

has to be

*Syntax:*

---

```
{attribute 'init_namespace'}
```

---

*Example*

---

```
PROGRAM PLC_PRG
VAR
    {attribute 'init_namespace'}
    {attribute 'noinit'}
    newString: STRING;
END_VAR
```

---

The variable "newString" is initialized with the current namespace, e.g. "PLC1.app1.PLC\_PRG.newString".

**Attribute 'init\_on\_onlchange'**

If the pragma is applied to a variable, it is initialized during each

*Syntax:*

---

```
{attribute 'init_on_onlchange'}
```

---

**Attribute 'instance-path'**

The pragma can be applied to a local string variable.

The string variable is initialized with the Project Explorer path of the POU to which it belongs. This can be useful for error messages.

The application of this pragma requires the application of the attribute to the associated POU and the application of the additional attribute to the string variable itself.

*Syntax:*

---

```
{attribute 'instance-path'}
```

---

Example: Function block "POU" with all necessary attributes

*Declaration of the function block POU:*

---

```
{attribute 'reflection'}
FUNCTION_BLOCK POU
VAR
    {attribute 'instance-path'}
    {attribute 'noinit'}
    str: STRING;
END_VAR
```

---

In the main program, one instance, "myPOU", of the function block POU is defined:

*Instance creation and usage:*

---

```
PROGRAM Plc_Main
VAR
    myPOU:POU;
    myString: STRING;
END_VAR
myPOU();
myString:=myPOU.str;
```

---

The initialization of the instance myPOU contains the string variable "str" which contains the path of the instance myPOU.

In the example "DCC\_Control.Application.Plc\_Main.myPOU" this path is assigned to the variable "myString" in the main program.

Programming Reference

**Attribute 'linkalways'**

Use this pragma to highlight the belonging object at the compiler. Thus, it is always included in the compiler information which means that it is always compiled and loaded to the PLC. The pragma is only effective for POU's and GVL's located below an application or in libraries below an application.

The compiler option `LINKALWAYS` performs the same.

*Syntax*

```
{attribute 'linkalways'}
```

`LINKALWAYS`, the highlighted POU is used as basis for the selectable variables of the symbol configuration.

**Example** The attribute 'linkalways' is used to implement the global variable list "GVLMoreSymbols":

*Global variable list "GVLMoreSymbols"*

```
{attribute 'linkalways'}
VAR_GLOBAL
g_iVar1: INT;
g_iVar2: INT;
END_VAR
```

This code provides the variables of "GVLMoreSymbols" as selectable symbols.

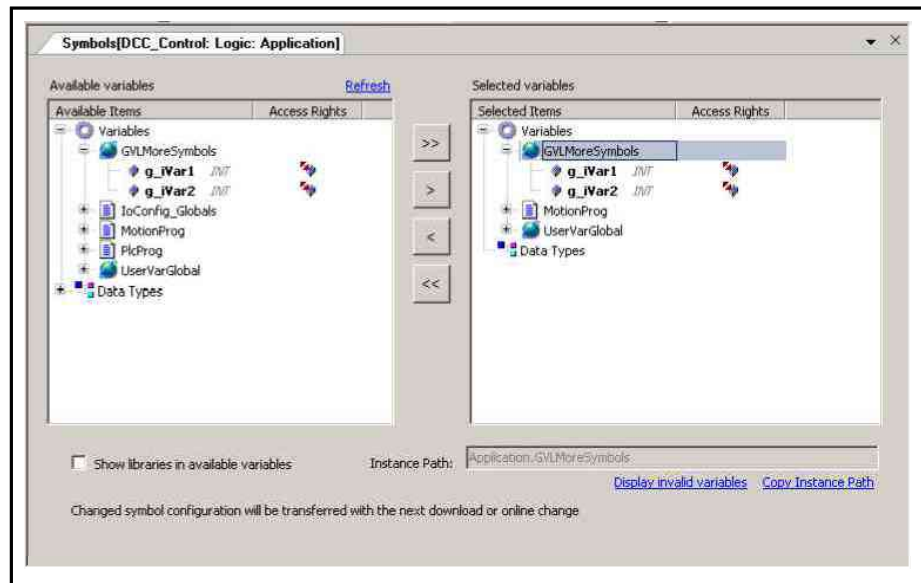


Fig. 5-7: Editor of the symbol configuration

**Attribute 'monitoring'**

In online mode, a `PROPERTY` can be monitored either using the `MONITORING` or a `MONITORING`.

Monitoring can be enabled by inserting the 'monitoring' attribute pragma into the line above the definition of the property. Subsequently, name, type and value of the variables are displayed in the online view of the function block using the property or in a `PROPERTY`. Values to force the property variables can also be prepared there.

There are two different ways to display the current value of the "properties" variables.

Programming Reference

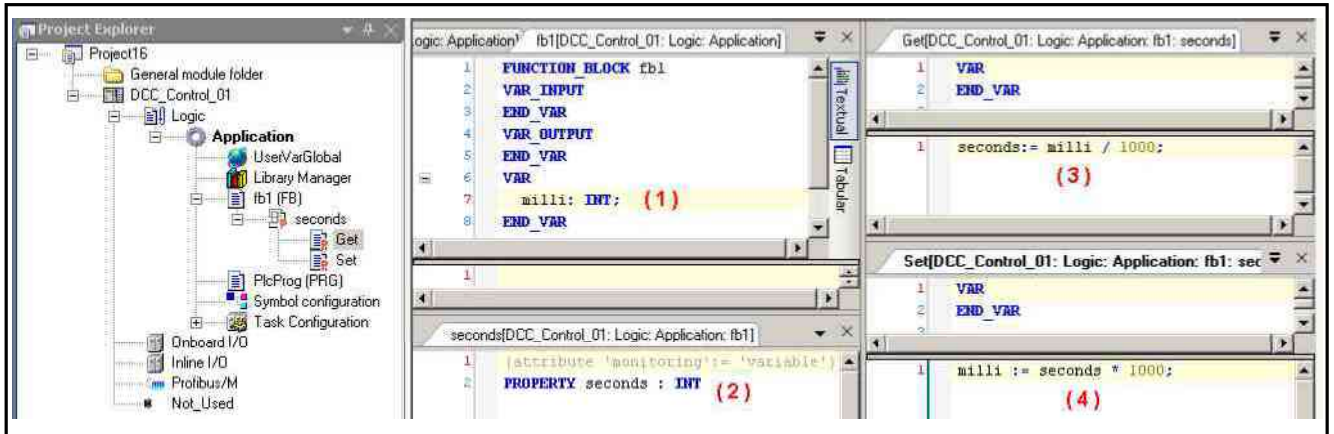
Decide for each case which attribute pragma is suitable to display the desired value. It depends on whether further operations with the variables are implemented in the property.

1. Pragma **{attribute 'monitoring':='variable'}**

An implicit variable is created for the property that receives the current "properties" value whenever the application calls either the "Set" or "Get" method. The value of these implicit variable is shown in the monitoring.

*Syntax:*

```
{attribute 'monitoring':='variable'}
```



- (1) Function block "fb1" with the local variable 'milli'
- (2) Property 'seconds' with attribute pragma
- (3) "Get" of the 'seconds' property
- (4) "Set" of the 'seconds' property

Fig. 5-8: Example of the 'seconds' property prepared for monitoring

The figure below shows the program with the test variable 'testvar' and the declaration of the function block instance in the upper part.

The implementation includes in line 1: "Set" method and in line 2: "Get" method.

The figure below shows the monitoring including the display of the 'seconds' property.

Programming Reference

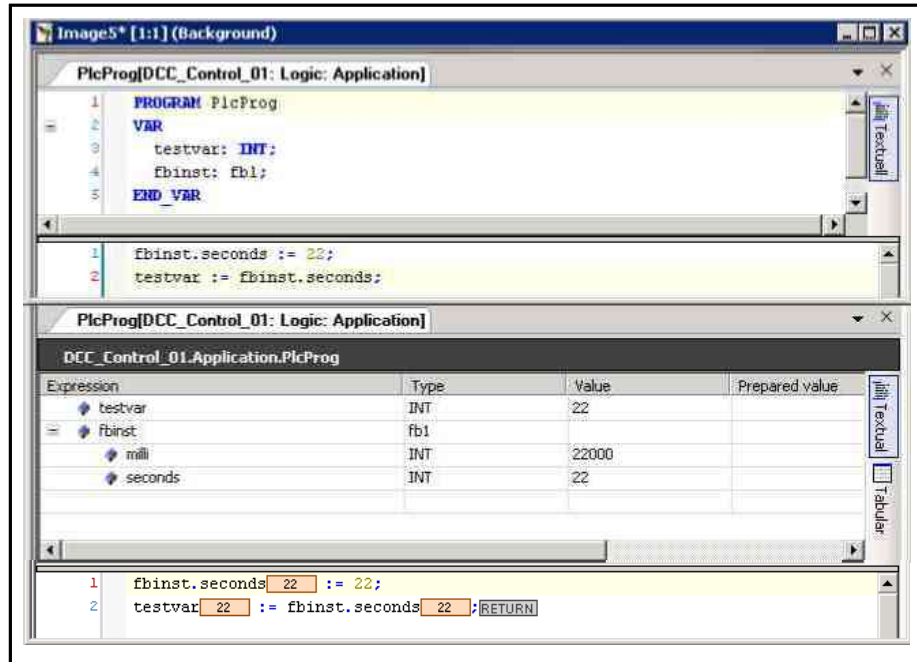


Fig.5-9: Example of a monitoring view with the 'seconds' property

2. Pragma **{attribute 'monitoring':='call'}**

This attribute can only be used for properties that return only simple data types or pointers, but no structured types:

The monitored value is obtained by calling the property directly, that means that the monitoring service of the runtime system calls the "Get" method.

If operations are implemented on the variables in the "property", the value can still change!

*Syntax:*

```
{attribute 'monitoring':='call'}
```

**Attribute 'no\_check'**

The pragma is added to a POU to impede each call of a check function ( ).

Since the check functions can affect the processing velocity of the program, it is reasonable to apply the attribute to function block that have already been checked or are often called.

*Syntax:*

```
{attribute 'no_check'}
```

**Attribute 'no\_copy'**

In general, an requires a reallocation of instance, e.g. of a POU.

In this case, the value of the variables in the instance is copied.

However, if the pragma is applied to a variable, a copy of the variable value is not made. Instead, the variable is re-initialized during an online change.

This can be useful for a local pointer variable that points to a variable that was just moved due to an online change (which thus also changed the address).

*Syntax:*

Programming Reference

---

```
{attribute 'no_copy'}
```

---

**Attribute 'no-exit'**

, the call of this method can be suppressed for a special instance of the function block by applying the pragma to the instance.

*Syntax:*

---

```
{attribute 'symbol' := 'no-exit'}
```

---

*Example:*

The exit method "FB\_Exit" is added to the function block called "POU":

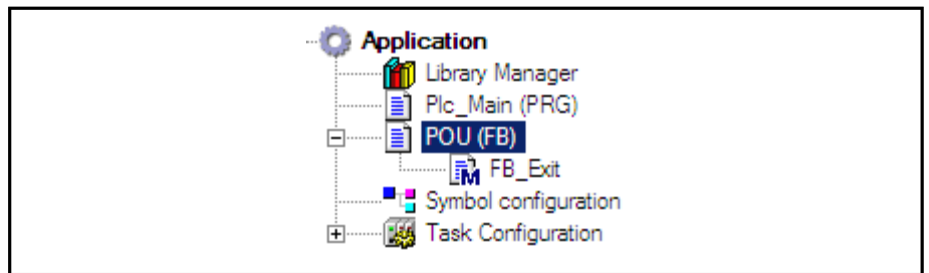


Fig.5-10:

In the main program PLC\_PRG, two instances of the function block "POU" are created:

*Declaration of the program PLC\_PRG:*

---

```
PROGRAM PLC_PRG
VAR
    POU1 : POU;
    {attribute 'symbol' := 'no-exit'}
    POU2 : POU;
END_VAR
```

---

The method named "FB\_exit" is a special method for a function block. It has to be declared explicitly. There is no implicit declaration.

The "exit" method - if available - is called for all declared instances of the function block before a new download or during online changes for all new or deleted instances.

Interface of the FB\_exit method: There is only one obligatory parameter:

*Interface of the FB\_exit method:*

---

```
METHOD fb_exit : BOOL
VAR_INPUT
    bInCopyCode : BOOL; // if TRUE, the exit method is called
                    // to leave the instance, which is copied
                    // afterwards (Online Change).
END_VAR
```

---

If the variable "bInCopyCode" is assigned to the value TRUE within POU1, the "exit" method FB\_Exit is called. In contrast, the value of the variable "bInCopyCode" in POU2 has no effect.

**Attribute 'no\_init'**

Variables that have the pragma are not initialized implicitly.

The pragma refers only to the variables declared in direct succession.



Programming Reference

*Alternatives in the syntax:*

```
{attribute 'no_init'}
{attribute 'no-init'}
{attribute 'noinit'}
```

*Example*

```
PROGRAM PLC_PRG
VAR
    A : INT;
    {attribute 'no_init'}
    B : INT;
END_VAR
```

When the respective application is reset, the integer variable "A" is initialized with 0 again while the variable "B" keeps its current variable.

**Attribute 'no\_virtual\_actions'**

This attribute concerns function blocks derived from a function block implemented in the SFC and that use the SFC procedure of this basic class.

The actions called from there show the same virtual behavior as the methods. That means that the implementations of the actions into the basic class can be replaced by the derived class with individual, specific implementations.

However, if the basic class gets the pragma {attribute 'no\_virtual\_actions'}, its actions are protected against overload.

*Syntax:*

```
{attribute 'no_virtual_actions'}
```

**Example:**

The following example shows the function block POU\_SFC that creates the basic class for the derived function block POU\_child.

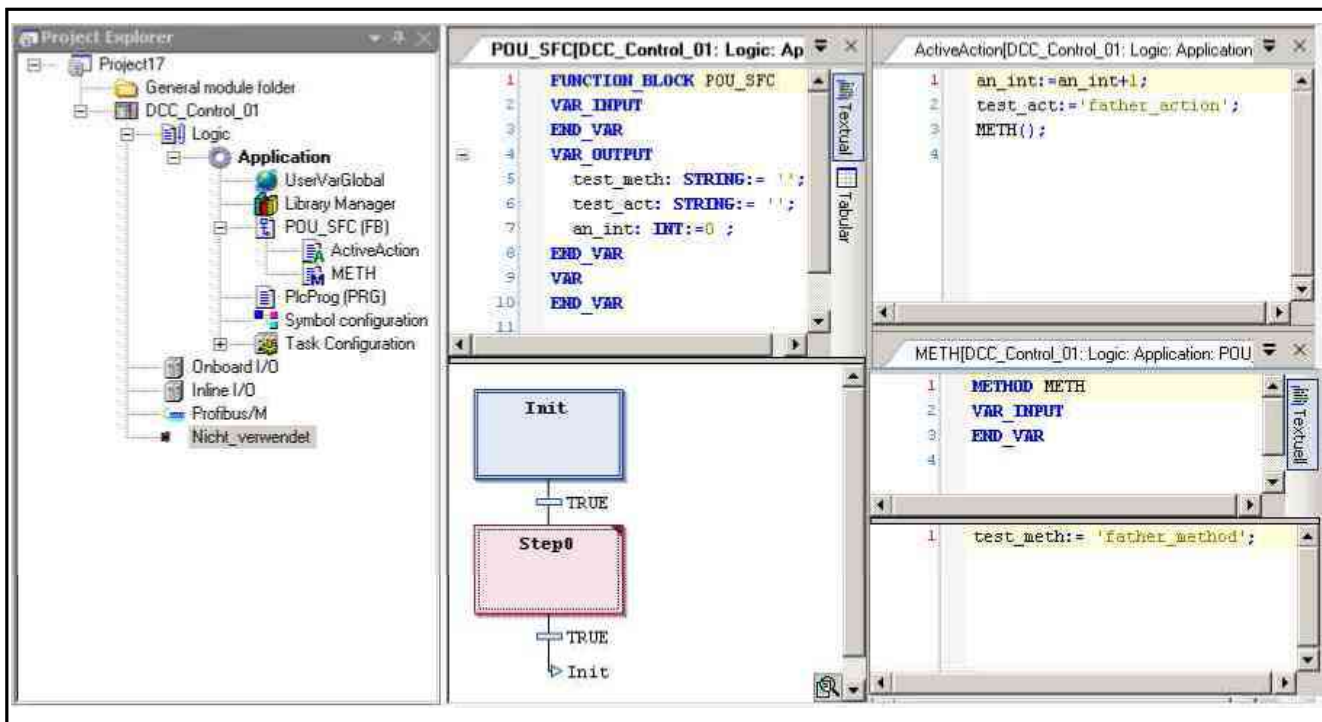


Fig.5-11: Function block POU\_SFC with the action "ActiveAction" and the method "METH"

Programming Reference

The exemplary implementation of this sequence is limited to the initial step followed by only one step with the linked step action "ActiveAction" which assigns the output variables and calls the "METH" method.

The "METH" method assigns the string 'father\_method' to the "test\_meth" variable in the basic class.

The function block POU\_child derives from the function block POU\_SFC (basic class).

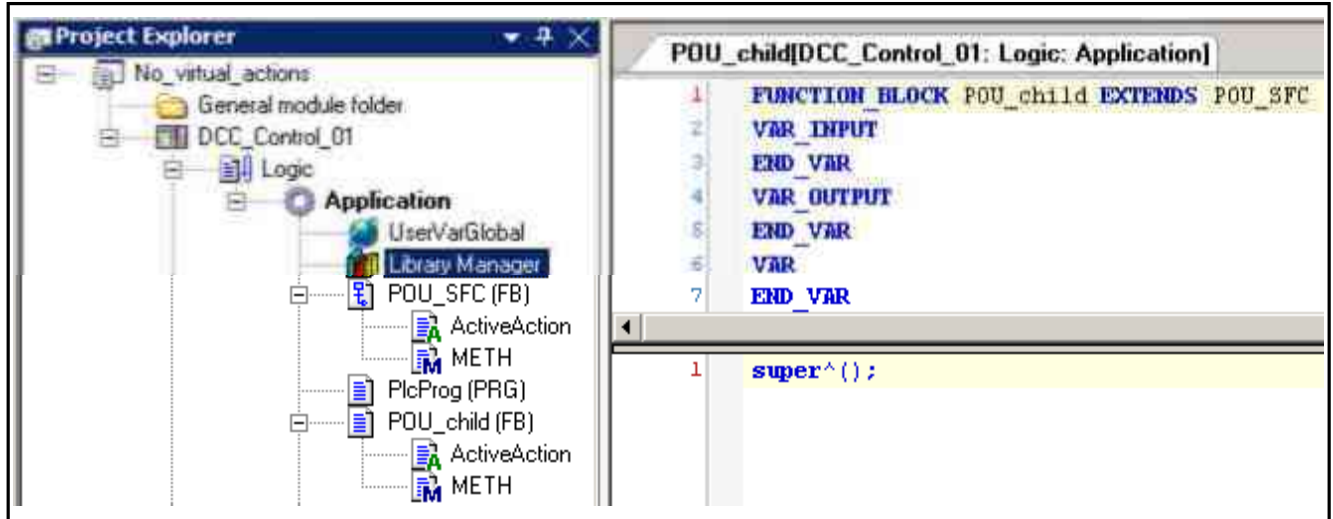


Fig.5-12: Derived function block POU\_child

POU\_Child uses ST as implementation code and is provided with the "Active-Action" action and the "METH" method.

For the derived class POU\_child, the step action is replaced by a special implementation of "ActiveAction" only differing from the original by assigning the "child\_action" string instead of "father\_action" to the "test\_act" variable.

The METH method, assigning the "father\_method" string to the "test\_meth" variable in the basic class, is overwritten in such as way that "test\_meth" gets the "child\_method" value.

The main program "PlcProg" calls an instance of the function block "POU\_child" called "Child". As expected, the value of the strings reflect the call of action and method of the derived class:

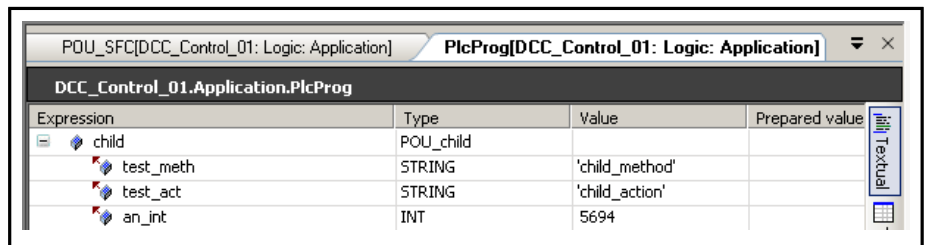


Fig.5-13: Main program "PlcProg"

The "no\_virtual\_actions" attribute precedes the basis  
{attribute 'no\_virtual\_actions' }

FUNCTION\_BLOCK POU\_SFC...

a different behavior can thus be observed: the implementation of the derived class is used for the "METH" method. Calling the step action now results in calling the "ActiveAction" of the basic class.

Thus, "test\_act" is now provided with the value "father\_action":



Programming Reference

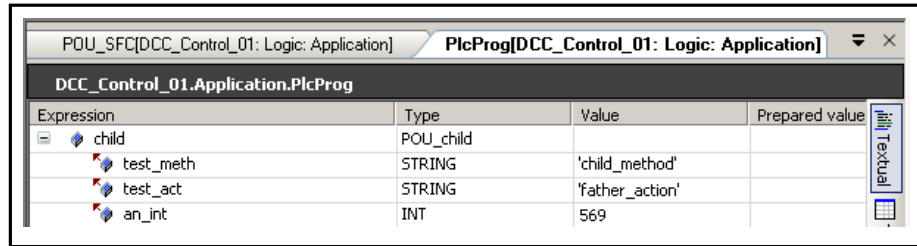


Fig.5-14: Main program "PlcProg" online, using the attribute "no\_virtual\_actions" for FUNCTION\_BLOCK POU\_SFC

**Attribute 'obsolete'**

An "Obsolete pragma" can be added to a data type definition in order to output a defined warning during compilation if the data type (structure, function block, etc.) is used in the project.

This way, it can be noted that a data type is no longer valid, since an interface has changed for example and that this should be updated in the project.

that is used locally, this warning is defined "centrally" for all instances of a data type.

This pragma always affects the current line or, if it is located in a separate line, the following line.

*Syntax:*

```
{attribute 'obsolete' := 'user defined text'}
```

The attribute is added to the declaration of the function block "fb1":

*Example*

```
{attribute 'obsolete' := 'datatype fb1 not valid!'}
FUNCTION_BLOCK fb1
VAR_INPUT
  i : INT;
END_VAR
```

If "fb1" is used as (data) type, e.g. in "fbinst: fb1;", this warning is output when the project is compiled: "datatype fb1 not valid"

**Attribute 'pack\_mode'**

The pragma specifies how a data structure is packed during the allocation. The attribute has to be inserted above the data structure and affects the zipping of the entire structure.

*Syntax:*

```
{attribute 'pack_mode' := '<value>'}
```

The placeholder <value>, enclosed in simple apostrophes, has to be replaced by one of the following values:

Value	Associated packing mode
0	Aligned, i.e. there are no memory gaps
1	1-byte aligned (same as aligned)
2	2-byte aligned, i.e. the maximum size of a memory gap is 1 byte
4	4-byte aligned, i.e. the maximum size of a memory gap is 3 bytes
8	8-byte aligned, i.e. the maximum size of a memory gap is 7 bytes

*Example*

```
{attribute 'pack_mode' := '1'}
TYPE myStruct:
STRUCT
  Enable: BOOL;
  Counter: INT;
  MaxSize: BOOL;
  MaxSizeReached: BOOL;
END_STRUCT
END_TYPE
```

The memory space for a variable of the data type myStruct is assigned "1-byte aligned":

If the memory address of its component is "Enable", e.g. 0x0100, the component "Counter" follows at the address 0x0101, MaxSize at the address 0x0103 and MaxSizeReached at the address 0x0104.

If 'pack\_mode'=2, "Counter" is at 0x0102, "MaxSize" is at 0x0104 and "MaxSizeReached" is at 0x0106.



The attribute can also be applied to POU's.

Due to possibly internal pointers, deal carefully with the application of "Attribute Pack\_mode".

**Attribute 'parameterstringof'**

The attribute of the pragma can be used to provide the instance name of a variable to a visualization function block.

*Syntax:*

```
{attribute 'parameterstringof' := '<variable>'}
```

In the main program, the instance myDUT of the user-defined structure DUT is created.

*Example*

```
PROGRAM PLC_PRG
VAR
  myDUT: DUT;
END_VAR
```

This instance is the input of a visualization block "Vis" (in the input parameter "instance") which is referenced by a frame of another visualization "MainVisu":

Elementname	GenElemInst_2
Clipping	<input type="checkbox"/>
Show frame	<input type="checkbox"/>
Scaletype	ANISOTROPIC
References	_35.CoDeSys.VisualElem.StructuredTypeNode
References	PLCWinNT.Application.Vis
instance	PLC_PRG.myDut
Position	
v	1n4

Fig.5-15: Section from the element properties of a visualization element of MainVisu.

In the interface editor belonging to "Vis", there is the input/output variable "instance" and also another input variable called "instanceStr":

Programming Reference

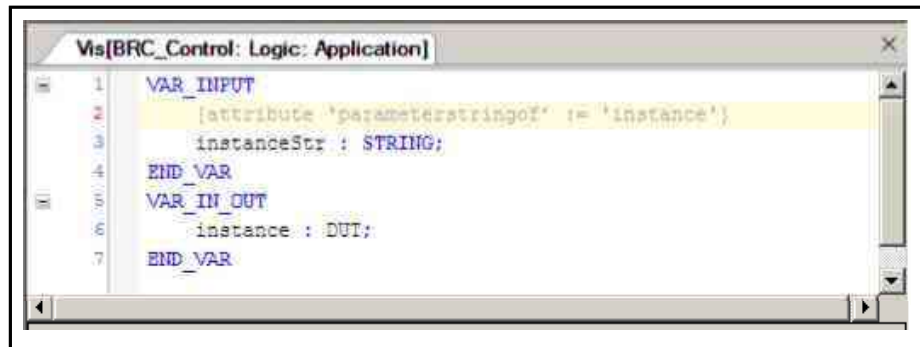


Fig.5-16: Interface editor

Although "instanceStr" is an input variable, it is not executed as an input in the reference (compare with the figure above!). This is caused since the variable "instanceStr" has the 'parameterstringof' attribute and is therefore automatically initialized with the name of the variables specified with the attribute. In the example, "instance" is the associated variable, so the string variable "instanceStr" is set to "PLC\_PRG.myDUT" and can only be used in the visualization "Vis" as text variable for a placeholder %s for example.

**Attribute 'qualified\_only'**

If the pragma precedes a global variable list, the "GVL" variables can only be addressed by specifying the global variable name, e.g.: "gvl.g\_var".

This also applies to variables of the type enumerations.

The attribute "Qualified\_Only" can be used to prevent confusion with local variables.

*Syntax:*

---

```
{attribute 'qualified_only'}
```

---

The following global variable list "GVL" has the attribute 'qualified\_only':

*Example*

---

```
{attribute 'qualified_only'}
VAR_GLOBAL
  iVar:INT;
END_VAR
```

---

Within a POU, e.g. within the program PLC\_Main, the global variable "iVar" can only be addressed by using the prefix GVL.

*Example:*

```
GVL.iVar:=5;
```

However, the incomplete call of the variables generates an error:

```
iVar:=5;
```

**Attribute 'reflection'**

Signatures are added to the pragma.

For optimization purposes, this attribute has to be specified to function blocks containing the

*Syntax:*

---

```
{attribute 'reflection'}
```

---

Function block "POU" with all necessary attributes:



### Example

```
{attribute 'reflection'}  
FUNCTION_BLOCK POU  
VAR  
    {attribute 'instance-path'}  
    {attribute 'noinit'}  
    str: STRING;  
END_VAR
```

In the main program, one instance, "myPOU", of the function block POU is defined:

### Application in the program:

```
PROGRAM Plc_Main  
VAR  
    myPOU:POU;  
    myString: STRING;  
END_VAR  
myPOU();  
myString:=myPOU.str;
```

The initialization of the instance myPOU contains the string variable "str" which contains the path of the instance myPOU.

In the example "DCC\_Control.Application.Plc\_Main.myPOU" this path is assigned to the variable "myString" in the main program.

### Attribute 'relative\_offset'

The attribute is used in structure definitions to shift structure elements in the memory space allocated by the structure. It is thus for example possible to allocate two structure entries on one memory space.

### Syntax:

```
{attribute 'relative_offset' := '<offset>'}
```

Counting the relative offset always starts at the starting address of the structure. The attribute becomes effective for all the following structure entries.

### Application in the structure definition:

```
TYPE STRUCT_TEST :  
STRUCT  
    wTest: WORD;  
  
    {attribute 'relative_offset' := '0'}  
    bit_0 : BIT; // Bit 0  
    bit_1 : BIT; // Bit 1  
    bit_2 : BIT; // Bit 2  
    bit_3 : BIT; // Bit 3  
    bit_4 : BIT; // Bit 4  
    bit_5 : BIT; // Bit 5  
  
END_STRUCT  
END_TYPE
```

### Attribute 'symbol'

The pragma defines which variables are included in the i.e. which variables should be exported to the symbol list as symbols.

These variables are provided for the external access as XML file in the project directory and for users as a hidden file on the target system; e.g. for access by an OPC server.

The variables with the attribute are loaded to the control, even if they are not explicitly configured or visible in the editor of the symbol configuration.

## Programming Reference



Note that the symbol configuration has to be created as an object below the respective application in the Project Explorer.



Note that only symbols from programs or global variable lists can be accessed. To access a symbol, the symbol name has to be entered completely.

*Syntax:*

```
{attribute 'symbol' := 'none' | 'read' | 'write' | 'readwrite'}
```

The pragma statement can be added to individual variables or can be assigned collectively to all variables declared in a program.

- To affect just a single variable, the pragma has to be placed in the line in front of the variable declaration.
- To affect all variables in the declaration part of a program, the pragma has to be placed in the first line of the declaration editor. However, even in this case, statements for individual variables can be positioned explicitly in the respective lines.

The parameter determines the possible **access** to a symbol. Possible specifications: 'none', 'read', 'write' or 'readwrite'. If no parameter is specified, the default value 'readwrite' applies.

**Example:**

The variables "A" and "B" are exported with read-only and write access using the following configuration. Variable "D" is exported with read-only access.

*Examples:*

```
{attribute 'symbol' := 'readwrite'}  
PROGRAM PLC_PRG  
VAR  
  A : INT;  
  B : INT;  
  {attribute 'symbol' := 'none'}  
  C : INT;  
  {attribute 'symbol' := 'read'}  
  D : INT;  
END_VAR
```

**Conditional Pragmas**

The extension of the programming language "ST", supports a variety of conditional that affect the code generation in the precompiling or compiling process (compilation of the project).

*The compilation of a implementation code becomes conditional to:*

- a specific data type or variable declared
- a data type of a variable with a specific attribute
- a variable with a specific data type
- a specific function block or task existing or part of the call tree
- etc.



If is not possible for a POU or a GVL created in the POU window to use a "{define...}" declared in an application.

"defines" in applications become only effective for interfaces located below the application.



Programming Reference

{define <b>identifier</b> string}	During the precompilation, all subsequent instances of the identifier <b>identifier</b> are replaced by the specified character string <b>string</b> if it is not empty (which is permitted). The identifier remains defined and valid until the end of its application area or until it is reset by an {undefine} statement. Required for a conditional compilation; see
{undefine identifier} .	The preprocessor definition of the identifier <b>identifier</b> (by {define}, see above) is reset. The identifier is now undefined again. If the currently specified identifier is not defined at all, the pragma is ignored.
{IF expr} ... {ELSIF expr} ... {ELSE} ... {END_IF}	These Pragmas are for the <b>conditional</b> compilation. The specified <b>expr</b> expressions have to be constant at the time of compilation. They are evaluated in the sequence in which they appear until one of the expressions displays a value not equal to "zero". The text linked with the statement is prepared and then compiled. The other lines are ignored. The sequence of the sections is specified. However, the <b>elsif</b> and <b>else</b> sections are optional and <b>elsif</b> sections can occur unlimitedly.  Within the <b>expr</b> constant, several conditional compilation can be used, see below.

Fig.5-17: Possible effects of the compilation

**Conditional compilation operators**

One or more operators can be used in the constant expression **expr** of a conditional compilation pragma **{IF}** or **{ELSIF}**, see above. However, they may not be defined {define} or undefined {undefine}.

Note that like the definition, these expressions can also be entered as "compiler definitions" using a {define} in of an object.

The following operators are supported:

Operator	Effect and example
defined ( <b>identifier</b> )	This operator causes that the expression gets the value TRUE if the identifier "identifier" was defined using a statement and was not undefined afterwards with an statement. Otherwise, FALSE is returned.
defined (variable: <b>variable</b> )	This operator causes that the expression gets the value TRUE if the variable "variable" is declared in the current validity range. Otherwise, FALSE is returned.
defined (type: <b>identifier</b> )	This operator causes that the expression gets the value TRUE if a data type with an identifier "identifier" is declared. Otherwise, FALSE is returned.
defined (pou: <b>pou-name</b> )	This operator causes that the expression gets the value TRUE if a function block or an action with the name "pou-name" is present. Otherwise, FALSE is returned.
not yet implemented: defined (task: <b>identifier</b> )	This operator causes that the expression gets the value TRUE if a task with the name "identifier" is defined. Otherwise, FALSE is returned.
not yet implemented: defined (resource: <b>identifier</b> )	This operator causes that the expression gets the value TRUE if a resource object with the name "identifier" is present for the application. Otherwise, FALSE is returned.





Programming Reference

Operator	Effect and example
hasattribute (pou: pou-name, attribute)	This operator causes that the expression gets the value TRUE if the attribute "attribute" is specified in the first line of the declaration part of the function block "pou-name". Otherwise, FALSE is returned. <a href="#">hasattribute (pou: pou-name, 'attribute'), page 550</a>
hasattribute (variable: variable, attribute)	This operator causes that the expression gets the value TRUE if the attribute "attribute" is assigned to the variable "variable" using the {attribute} statement in the line in front of the variable declaration. Otherwise, FALSE is returned.
hastype (variable: variable, type-spec)	This operator causes that the expression gets the value TRUE if the variable "variable" is a data type. Otherwise, FALSE is returned.
hasvalue (define-ident, char-string)	This operator causes that the expression gets the value TRUE if a variable is defined with an identifier "define-ident" and has the value "char-string". Otherwise, FALSE is returned.
NOT operator	The expression receives the value TRUE if the inverse value of the operator "operator" returns TRUE. "operator" can be one of the operators described in this table.
AND operator	The expression returns the value TRUE if the specified "operator" operators both return TRUE. "Operator" can be one of the operators described in this table.
OR operator	The expression returns TRUE if both of the specified operators "operator" return TRUE. "Operator" can be one of the operators described in this table.
(operator)	Operator parentheses

**defined (identifier)**

This operator causes that the expression gets the value TRUE if the identifier "identifier" was defined using a {define} statement and was not undefined afterwards with an {undefine} statement. Otherwise, FALSE is returned.

Prerequisite: There are two applications, "App1" and "App2". The variable "hallo" is defined by a {define} statement in "App1", but not in "App2".

*Example*

```
{IF defined (pdef1)}
    // this code is processed in App1
    {info 'pdef1 defined'}
    hugo := hugo + SINT#1;
{ELSE}
    // the following code is only processed in App2
    {info 'pdef1 not defined'}
    hugo := hugo - SINT#1;
```

There is also an example of a included:

Only the information "pdef1 defined" is displayed in the message window if the application is compiled, since "pdef1" is really defined. The message "pdef1 not defined" is output if "pdef1" is not defined.

**defined (variable: variable)**

This operator causes that the expression gets the value TRUE if the variable **variable** is declared in the current validity range. Otherwise, FALSE is returned.



Programming Reference

Prerequisite: There are two applications, App1 and App2. Variable "g\_bTest" is declared in "App1", but not in "App2.Precondition"

*Example*

```
{IF defined (variable:g_bTest)}
// the following code is processed in App2 only
g_bTest := x > 300;
{END_IF}
```

**defined (type: identifier)**

This operator causes that the expression gets the value TRUE if a data type with an identifier **identifier** is declared. Otherwise, FALSE is returned.

Prerequisite: There are two applications, "App1" and "App2". The data type "DUT" is declared in "App1", but not in "App2".

*Example*

```
{IF defined (type:DUT)}
// the following code is processed in App1 only
bDutDefined := TRUE;
{END_IF}
```

**defined (pou: pou-name)**

This operator causes that the expression gets the value TRUE if a function block or an action with the name "pou-name" is present. Otherwise, FALSE is returned.

Prerequisite: There are two applications, "App1" and "App2". The block "CheckBounds" is present in "App1", but not in "App2".

*Example*

```
{IF defined (pou:CheckBounds)}
// the following code is processed in App1 only
arrTest[CheckBounds(0,i,10)] := arrTest[CheckBounds(0,i,10)]+1;
{ELSE}
// the following code is processed in App2 only
arrTest[i] := arrTest[i]+1;
{END_IF}
```

**not yet implemented: defined (task: identifier)**

This operator causes that the expression gets the value TRUE if a task with the name "identifier" is defined. Otherwise, FALSE is returned.

Prerequisite: There are two applications, "App1" and "App2". The task "PLC\_PRG\_Task" is defined in "App1", but not in "App2".

*Example*

```
{IF defined (task:PLC_PRG_Task)}
// the following code is processed in App1 only
erg := plc_prg.x;
{ELSE}
// the following code is processed in App2 only
erg := prog.x;
{END_IF}
```

**not yet implemented: defined (resource: identifier)**

This operator causes that the expression gets the value TRUE if a resource object with the name "identifier" is present for the application. Otherwise, FALSE is returned.

Prerequisite: There are two applications, "App1" and "App2". a resource object "glob\_var1" (Global Variable List) is present for "App1", but not for "App2".

*Example*

```
{IF defined (resource:glob_var1)}
// the following code is processed in App1 only
gvar_x := gvar_x + ivar;
{ELSE}
// the following code is processed in App2 only
x := x + ivar;
{END_IF}
```



Programming Reference

**hasattribute (pou: pou-name, 'attribute')**

This operator causes that the expression gets the value TRUE if the attribute "attribute" is specified in the first line of the declaration of the function block pou-name. Otherwise, FALSE is returned.

Prerequisite: There are two applications, "App1" and "App2". A function "fun1" is defined in "App1" and "App2", but in "App1" the attribute 'vision' is also assigned to it:

*Definition of fun1 in App1:*

```
{attribute 'vision'}
FUNCTION fun1 : INT
VAR_INPUT
  i : INT;
END_VAR
VAR
END_VAR
```

*Definition of fun1 in App2:*

```
FUNCTION fun1 : INT
VAR_INPUT
  i : INT;
END_VAR
VAR
END_VAR
```

*Pragma statement:*

```
{IF hasattribute (pou: fun1, 'vision')}
  // the following code is processed in App1 only
  ergvar := fun1(ivar);
{END_IF}
```

**hasattribute (variable: variable, 'attribute')**

This operator causes that the expression gets the value TRUE if the attribute **attribute** is assigned to the variable **variable** using the {attribute} statement in the line in front of the variable declaration. Otherwise, FALSE is returned.

Prerequisite: There are two applications, "App1" and "App2". Variable "g\_globalInt" is used in "App1" and "App2", but in "App1" the attribute 'docount' is also assigned to it.

*Declaration of g\_globalInt in App1:*

```
VAR_GLOBAL
{attribute 'docount'}
  g_globalInt : INT;
  g_multiType : STRING;
END_VAR
```

*Declaration of g\_globalInt in App2:*

```
VAR_GLOBAL
  g_globalInt : INT;
  g_multiType : STRING;
END_VAR
```

*Pragma statement:*

```
{IF hasattribute (variable: g_globalInt, 'docount')}
(* the following code line is executed in App1 only,
  because there the variable g_globalInt is defined
  with the attribute 'docount'*)
  g_globalInt := g_globalInt + 1;
{END_IF}
```

**hastype (variable: variable, type-spec)**

This operator causes that the expression gets the value TRUE if the variable **variable** is of the data type **type-spec**. Otherwise, FALSE is returned.



**Available data types of "type-spec"**

ANY	LINT	WSTRING
ANY_DERIVED	DINT	STRING
ANY_ELEMENTARY	INT	
ANY_MAGNITUDE	SINT	TIME
ANY_BIT	ULINT	DATE_AND_TIME
ANY_STRING	UDINT	DATE
ANY_DATE	UINT	TIME_OF_DAY
ANY_NUM	USINT	
ANY_REAL	LWORD	
ANY_INT	DWORD	
	WORD	
LREAL	BYTE	
REAL	BOOL	

Prerequisite: There are two applications, "App1" and "App2". The variable "g\_multitype" is declared in "App1" with data type LREAL, but in "App2" with the data type STRING:

*Example*

```
{IF (hastype (variable: g_multitype, LREAL))}
// the following code line is executed in App1 only
g_multitype := (0.9 + g_multitype) * 1.1;
{ELSIF (hastype (variable: g_multitype, STRING))}
// the following code line is executed in App2 only
g_multitype := 'this is a multitalent';
{END_IF}
```

**hasvalue (define-ident, char-string)**

This operator causes that the expression gets the value TRUE if a variable is defined with an identifier **define-ident** and has the value **char-string**. Otherwise, FALSE is returned.

Prerequisite: There are two applications, "App1" and "App2". The variable "test" is used in the applications "App1" and "App2"; in "App1" it has the value "1", in "App2" the value "2":

*Example*

```
{IF hasvalue(test, '1')}
(* the following code is executed in App1 because
there the value of the test variable is 1 *)
x := x + 1;
{ELSIF hasvalue(test, '2')}
(* the following code is executed in App1 because
there the variable test is 2 *)
x := x + 2;
{END_IF}
```

**NOT operator**

The expression gets the value TRUE if the reciprocal of the **operator** operator returns TRUE. The **operator** can be one of the operators described in this table.

Prerequisite: There are two applications, "App1" and "App2". The POU "PLC\_PRG1" is present in "App1" and "App2", but the POU "CheckBounds" is only in "App1":



Programming Reference

*Example*

```
{IF defined(pou: PLC_PRG1) AND NOT (defined(pou: CheckBounds))}
  // the following code line is executed in App2 only
  bANDNotTest := TRUE;
{END_IF}
```

**AND operator**

The expression returns the value TRUE if the specified **operator** operators both return TRUE. **Operator** can be one of the operators described in this table.

Prerequisite: There are two applications, "App1" and "App2". The POU "PLC\_PRG1" is present in "App1" and "App2", but the POU "CheckBounds" is only in "App1":

*Example*

```
{IF defined(pou: PLC_PRG1) AND (defined(pou: CheckBounds))}
  (* the following code line is executed in App1 only,
  because there only PLC_PRG1 and CheckBounds are defined.*)
  bORTest := TRUE;
{END_IF}
```

**OR operator**

The expression returns TRUE if both of the specified operators **operator** return TRUE. **Operator** can be one of the operators described in this table.

Prerequisite: There are two applications, "App1" and "App2". The POU "PLC\_PRG1" is present in "App1" and "App2", but the POU "CheckBounds" is only in "App1":

*Example*

```
{IF defined(pou: PLC_PRG1) OR (defined(pou: CheckBounds))}
  (* the following code line is execute in App1 and App 2,
  because at least one of the POEs contains PLC_PRG1
  and CheckBounds *)
  bORTest := TRUE;
{END_IF}
```

(operator) Operator parentheses

## 5.2 Data Types

### 5.2.1 Data Types, General Information

A data type is assigned to each identifier. A data type specifies the memory reserved and the values corresponding to the memory contents.

When programming in IndraLogic, memory can be reserved for instances of

- $\text{ARRAY[...]}$  and  $\text{ARRAY[...]} \text{ OF } \text{POU\_NAME}$
- $\text{ARRAY[...]} \text{ OF } \text{FUNCTION\_BLOCK\_NAME}$
- of function blocks

The following basic data types are supported:

	1 bit	8 bits	16 bits	32 bits	64 bits
Boolean variable	*				
Bit sequence					
Signed integers					
Unsigned integers					
Floating point number					



	1 bit	8 bits	16 bits	32 bits	64 bits
Time					*
Character string		**	**		
Reference				*	
Pointer				*	

\* Extension of the EN 61131-3 standard  
 \*\* Memory requirement of a character of the character string  
 Fig.5-18: Basic Data Types

## 5.2.2 Basic Data Types

### Basic Data Types, General Information

IndraLogic supports all data types described in the IEC 61131-3 standard.

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

Data types can also be defined themselves; see

### BIT

Variables of the data type BIT can assume the truth values TRUE (1) and FALSE (0).

One bit is reserved as memory space in the structures. If the data type is used outside the structures, one byte is reserved as memory space.

The data type BIT is not part of the standard IEC 61131-3.

Also refer to

- (operands).

### BOOL

Variables of the data type BOOL can accept the truth values TRUE (1) and FALSE (0).

An eight bit memory space is reserved.





Programming Reference

Also refer to

- (operands).

**Bit Sequence**

Each data type has a different data width (number of bits).

List of available bit sequence data types (with range limits):

Data type	Lower limit	Upper limit	Memory space
BYTE	16#00	16#FF	8 bits
WORD	16#0000	16#FFFF	16 bits
DWORD	16#0000 0000	16#FFFF FFFF	32 bits
LWORD	16#0000 0000 0000 0000	16#FFFF FFFF FFFF FFFF	64 bits

Also refer to

- (operands)
- (operands)

**Integer Data Types**

Each integer data type covers a specific number range.

List of available integer data types (with range limits):

Data type	Lower limit	Upper limit	Memory space
SINT	-128	127	8 bits
USINT	0	255	8 bits
INT	-32768	32767	16 bits
UINT	0	65535	16 bits
DINT	-2147483648	2147483647	32 bits
UDINT	0	4294967295	32 bits
LINT	-2 <sup>63</sup>	2 <sup>63</sup> -1	64 bits
ULINT	0	2 <sup>64</sup> -1	64 bits



When data types are converted from large to small, information can be lost.

Also refer to

- (operands)
- (operands)

**Floating Point Numbers**

REAL and LREAL are so-called floating point data types. The data types REAL and LREAL are used with rational numbers. The reserved memory space is 32 bits for REAL and 64 bits for LREAL.

Values allowed for REAL:

±(1.175494351e-38 to 3.402823466e+38)

Values allowed for LREAL:

±(2.2250738585072014e-308 to 1.7976931348623158e+308)



If a 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.

*Also refer to*

- (operands)

## Text Variables

A variable of the data type STRING can accept any character string. The size specification for the memory reservation for the declaration refers to characters (1 byte) and can be made in parentheses or square brackets. If no size is specified, 80 characters are defined as default.

In principle, the string length is not limited, but the string functions can only process lengths between 1 - 255!

When a variable is initialized with a string that is too long for the variable data type, the string is cut at the end accordingly.

String declaration with 35 characters: \_\_\_\_\_

*Example:*

```
str:STRING(35):= 'This is a String';
```

A variable of the data type WSTRING can accept any character string. The size specification for the memory reservation for the declaration refers to characters (2 byte) and can be made in parentheses or square brackets.

Compared to the STRING (ASCII), the WSTRING is interpreted in Unicode format.

*Example:*

```
wstr: WSTRING:= "This is a WString";
```

When a variable is initialized with a WString that is too long for the variable data type, the WString is cut at the end accordingly.

*Also refer to*

- (operands)
- (operands).

## Time Data Types

The data types "TIME", "TIME\_OF\_DAY" (abbreviated "TOD"), "DATE" and "DATE\_AND\_TIME" (abbreviated "DT") have a size of 32 bits.

TIME has a size of 32 bits and a resolution in milliseconds.

For TOD, the time is given in milliseconds and calculations begin for TOD at 00:00 o'clock.

For DATE and DT, the time is given in seconds and the calculations begin on January 1st 1970 at 00:00.



Programming Reference



The following reference declarations cannot be made:  
REFERENCE TO REFERENCE  
or  
ARRAY OF REFERENCE  
or  
POINTER TO REFERENCE.



References are initialized (with 0) with the compiler version ≥ V3.3.0.0.

**Check for valid references**

The operator "`__ISVALIDREF`" can be used to check if a reference 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.

**Example:**

*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
```

**Pointer (POINTER)**

In an extension of the IEC 61131-3 standard, the usage of pointers is supported:

Pointers save the addresses of variables, programs, function blocks, methods and functions while an application program is running.

A pointer can point to each of the named objects and to every and even to



Implicit used!

can be

*Syntax of a pointer declaration:*

```
<Name>: POINTER TO <data_type | Functionblock | Program | Method | Function>;
```

Dereferencing a pointer means obtaining the value that is currently at the address to which the pointer is pointing. A pointer can be dereferenced by adding the content operator "`^`" to the pointer indicator; see "`pt^`" in the example below.

## Programming Reference

The address operator "ADR" can be used to assign the address of a variable to a pointer.

*Example:*

---

```
VAR pt:POINTER TO INT; // Declaration of pointer pt
    var_int1:INT := 5; // Declaration of variables var_int1 and var_int2
    var_int2:INT; END_VAR

pt := ADR(var_int1); // Address of var_int1 is assigned pointer pt
var_int2:= pt^;
(* Value 5 of var_int1 is assigned to variable
   var_int2 by dereferencing pointer pt *)
```

---

### Function pointer

In contrast to IndraLogic 1.x, function pointers that replace the INDEXOF operator are now supported as well.

These pointers can be forwarded to external libraries, but a **function pointer cannot be called in an application in the programming system.**

The function of the runtime system to register callback functions (system library function) expects the function pointer. Depending on the callback registered, the respective function is then implicitly called by the runtime system (for example at stop).

To enable a system call (runtime system), the corresponding `CALLBACK` has to be set for the function object.

The `CALLBACK` can be used for functions, programs, function blocks and methods.

Since functions can change after an online change, the address of a pointer that points **to the function is output instead of the address of the function.** This address is valid as long as the function exists on the target system.

### Index access to pointers

An extension of the IEC 61131-3 standard allows the index access "[ ]" to variables of the type POINTER,

- `pint[i]` returns the basic data type.
- Index access to pointers is carried out arithmetically:

If the index access is used with a variable of type pointer, the offset is calculated using

$$\text{pint}[i] = (\text{pint} + i * \text{SIZEOF}(\text{base type}))^{\wedge}$$

The index access also causes an implicit dereferencing of the pointer. The resulting data type is the basic data type of the pointer.

Note: `pint[7] ≠ (pint + 7)^{\wedge}`!

- If the index access is used with a STRING type variable, the sign at offset index-expr. is obtained.

The result is of type BYTE.

`str[i]` returns the *i*th character of the string as SINT (ASCII).

- If the index access is used with a WSTRING type variable, the sign at offset index-expr. is obtained.

The result is of type WORD.

`wstr[i]` returns the *i*th character of the string as INT (Unicode).



**references**, which, in contrast to pointers directly affect a value, can also be used.

---

**Monitoring function for pointers**

The implicit monitoring function "CheckPointer" can be used to monitor the memory access by pointers at runtime.

It has to be integrated into the application as an object using the command

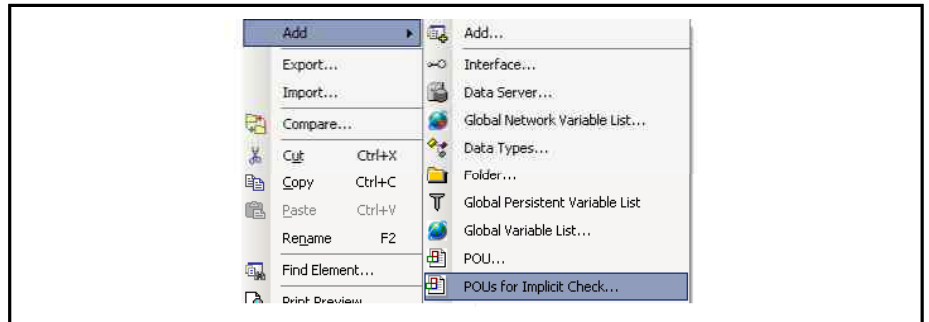


Fig.5-19: Add -> POU's for Implicit Check

After selecting the respective checkbox, select a program language and confirm it with "Open". The CheckPointer function opens in the respective editor.



Fig.5-20: Selecting CheckPointer



The declaration of the functions is specified for all languages and may not be modified! Only local variables may be added. In contrast to other monitoring functions, there is no suggestion for the implementation of the CheckPointer. Users have to carry out the implementation!

The CheckPointer function checks whether the transferred pointer points to a valid memory address and if the position of referenced memory area corresponds to the type of variables to which the pointer points. If both conditions are met, the pointer itself is returned. Otherwise, CheckPointer should perform an appropriate error handling.



No implicit call of the monitoring function is carried out for the THIS pointer.

*Template:*

```
(* Automatically generated code: DO NOT EDIT *)
FUNCTION CheckPointer : POINTER TO BYTE
VAR_INPUT
    ptToTest : POINTER TO BYTE;
    iSize : DINT;
```



Programming Reference

```
iGran : DINT;
bWrite: BOOL;
END_VAR

(* No standard implementation. Please enter your code here. *)
CheckPointer := ptToTest;
```

At the call, the functions of the following input parameters are transferred.

- ptToTest: Target address of the pointer
- iSize: Size of the referenced variable; the data type of iSize has to be compatible with INT and cover the range of the variables.
- iGran: Granularity of the referenced size, i.e. the largest unstructured data type in the referenced variables. The data type of iGran has to be compatible with INT.
- bWrite: Type of access ( TRUE=write access FALSE=read access. The data type of bWrite has to be BOOL

If the check is positive, the unmodified input pointer is returned (ptToTest).

### 5.2.3 User-defined Data Types

#### User-Defined Data Types, General Information

Except for the standard data types, users can define their own data types in a project.

These definitions can be made by creating in the Project Explorer or in the declaration of a function block.

Note to be as consistent as possible at the for objects.

*The following user-defined data types can be created:*

- 
- 
- 
- 
- 

#### Arrays (ARRAY)

One-, two- and three-dimensional arrays of elementary data types are supported. Arrays can be defined in the declaration of a function block and in the global variable lists. Note that there are

*Syntax:*

```
<Array_Name>:ARRAY [<ll1>..ul1>,<ll2>..ul2>,<ll3>..ul3>] OF <elem.Type>
```

ll1, ll2, ll3 identify the lower limit of an array dimension,

ul1, ul2 and ul3 identify the upper limit. These limit values have to be integers.

*Example:*

```
Card_game: ARRAY [1..13, 1..4] OF INT;
```

Initialization of arrays



In contrast to IndraLogic 1.x, "square brackets" have to be placed around the initialization value(s)!

**Complete initialization of an array:**

*Example:*

```
arr1 : ARRAY [1..5] OF INT := [1,2,3,4,5];
(* short for 1,7,7,7*)
arr2 : ARRAY [1..2,3..4] OF INT := [1,3(7)];
(* short for 0,0,4,4,4,4,2,3*)
arr3 : ARRAY [1..2,2..3,3..4] OF INT := [2(0),4(4),2,3];
```

**Initialization of an array of a structure:**

*Example:*

```
TYPE STRUCT1: // Structure definition
STRUCT
    p1:int;
    p2:int;
    p3:dword;
END_STRUCT
END_TYPE
// Array initialization in the application declaration
ARRAY[1..3] OF STRUCT1:= [(p1:=1, p2:=10,p3:=4723),
                          (p1:=2, p2:=0, p3:=299),
                          (p1:=14,p2:=5, p3:=112)];
```

**Partial initialization of an array:**

*Example:*

```
arr4 : ARRAY [1..10] OF INT := [1,2];
```

Elements without any assigned value as initialization value are initialized with the default value of the basic data type. In the example above, the elements arr4[3] to arr4[10] are initialized with 0.

**Access to array components**

In a two-dimensional array, the components are accessed as follows:

<Array-Name> [ Index1 , Index2 ] >

*Example:*

```
Card_game [9,2]
```

**Functions to check the array limits:**

To guarantee a correct access to the array elements, the "CheckBounds" function has to be available to the application.

It is integrated into the application as an object using the command

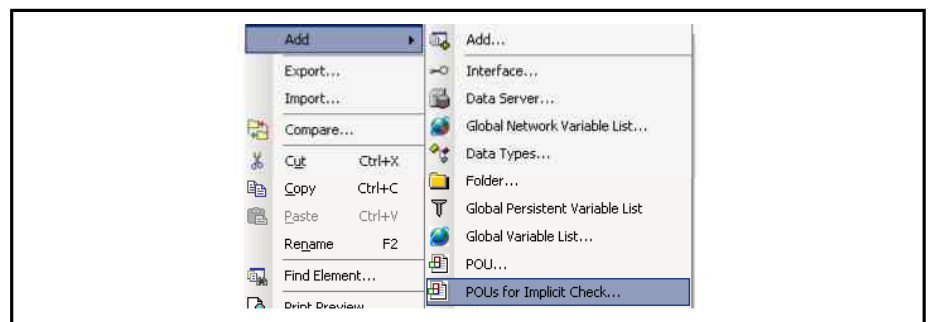


Fig.5-21: Add -> POU's for Implicit Check

After selecting "CheckBounds", select a program language and confirm it with "Open". The "CheckBounds" function opens in the respective editor.

Programming Reference



Fig.5-22: Selecting CheckBounds

The declaration part of the functions is specified for all languages and may not be modified! Only local variables may be added. A suggestion to implement the function is available in the programming language ST and may be modified as desired.

The task of the CheckBounds function is to monitor array limits and their exceedances. For example, if array limits are exceeded, an error flag can be set or the array indices can be changed. The function is called implicitly as soon as values are assigned to a variable of the type array.



To maintain the monitoring functionality, the declaration part of the function may not be modified!

**Using the CheckBounds function. The function is programmed in ST as follows by default:**

*Example to use the CheckBounds function*

```
// Declaration:
FUNCTION CheckBounds : INT
VAR_INPUT
    index, lower, upper: INT;
END_VAR

// Program:
(* Automatically generated code:
This is a proposal for implementation. *)
IF index < lower THEN
    CheckBounds := lower;
ELSIF index > upper THEN
    CheckBounds := upper;
ELSE
    CheckBounds := index;
END_IF
```

When calling, the function obtains the following input parameters:

- index: Index of the array element
- lower: Lower limit of the array dimension
- upper: Upper limit of the array dimension

The return value is the index of the array element as long as it is located in the valid range. Otherwise, depending on how the limit range has been violated, the upper or lower limit is returned.

Programming Reference

In the program below, the defined upper limit of the array "a" has been exceeded:

*Example:*

```
PROGRAM PLC_PRG
VAR
    a: ARRAY[0..7] OF BOOL;
    b: INT:=10;
END_VAR

a[b]:=TRUE;
```

In this case, the implicit call of the function CheckBounds carried out during the assignment causes the index 10 to be changed to the upper limit of the array range "a".

Thus, the value TRUE is assigned to the element [7]. This corrects attempted array accesses outside the valid array range.

**Structures (STRUCT)**

Structures are created in the project as "DUT" ( ) objects using the command "Add".

Alternatively, add a DUT from the application library.

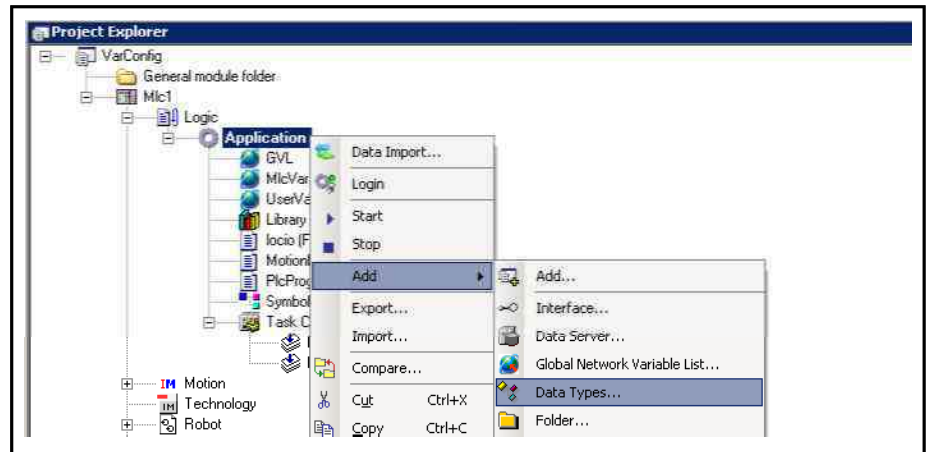


Fig.5-23: Adding a DUT

Structures start with the keywords **TYPE** and **STRUCT** and end with **END\_STRUCT** and **END\_TYPE**.



In contrast to IndraLogic 1.x, a colon ":" has to be placed after **TYPE** in the structure declaration.

The syntax for structure declaration is as follows:

*SYNTAX:*

```
TYPE <structure_name>:
STRUCT
    <variable_declaration 1>
    ..
    <variable_declaration n>
END_STRUCT
END_TYPE
```

<structure\_name> is a type that is identified in the entire project and that can be used as a standard data type.

Nested structures are permitted.



Programming Reference

The only limitation is that addresses may not be assigned to variables (AT declaration is not allowed!).

**A structure definition called polygon line:**

*Example: Structure with arrays as elements*

---

```

TYPE Polygonline:
  STRUCT
    Start:ARRAY [1..2] OF INT;
    Point1:ARRAY [1..2] OF INT;
    Point2:ARRAY [1..2] OF INT;
    Point3:ARRAY [1..2] OF INT;
    Point4:ARRAY [1..2] OF INT;
    End:ARRAY [1..2] OF INT;
  END_STRUCT
END_TYPE

```

---

**Initializing structures:**

*Example:*

---

```

Poly_1:polygonline:=(Start:=[3,3], Point1 =[5,2], Point2:=[7,3],
  Point3:=[8,5], Point4:=[5,7], End := [3,5]);

```

---

Initializations with variables are not permitted.

An example for initializing an array of a structure is located under

**Access to structure components**

Structure components are accessed according to this syntax:

```
<StructureName>.<ComponentName>
```

For the example of the "polygon line" structure above, also use

```
Poly_1.Start
```

to access the "Start" component.

**Union (UNION)**

In an extension of the IEC 61131-3 standard it is possible to declare "unions" in user-defined data types.

In a union, all components have the same offset, i.e. they occupy the same memory location.

Thus, in the following example declaration of a union an assignment to name1.a would also apply to name1.b.

*Example of a declaration in the data type editor (DUT):*

---

```

TYPE name1: UNION
  a : LREAL;
  b : LINT;
END_UNION
END_TYPE

```

---

*Example of the initialization and use of UNION 'name1':*

---

```

PROGRAM Plc_Main
VAR
  UnionInit:name1;
END_VAR

UnionInit.a:=10; // LREAL, 10.0

```

---

**Enumerations**

An enumeration is a consisting of a sequence of string constants.



## Programming Reference

Enumeration values are recognized globally in the project even if they are declared within a function block.

An enumeration is created in the project as "DUT" ( ) objects using the command "Add".



In contrast to IndraLogic 1.x, a local enumeration declaration can no longer be made - except in TYPE.

### Syntax:

```
TYPE
<name>:(<Enum_0>,<Enum_1>, ...,<Enum_n>)| <base_data_type>;
END_TYPE
```

A variable of type <identifier> can accept one of the enumeration values <Enum\_..> and is initialized with the first of these values.

The values are compatible with integers, i.e. operations can be carried out with them as with INTEGER variables.

A number x can be assigned to each enumeration value.

If this assignment is not made explicitly in the declaration, the first component gets "0" and the next components "1, 2", etc.

If the assignment of number values is made in the declaration, ensure that the sequence of numbers increases in the enumeration.

### Example:

```
TYPE TRAFFIC_SIGNAL: (red, yellow, green:=10);
// The values for the colors are red=0, yellow=1, green=10
END_TYPE
TRAFFIC_SIGNAL1 : TRAFFIC_SIGNAL;
TRAFFIC_SIGNAL1:=0; // The value of the traffic signal is red
FOR i:= red TO green DO
i := i + 1;
END_FOR;
```

### Extensions of the EN 61131-3 standard:

1. The type name of an enumeration can be used (as ) to provide unique access to an enumeration constant. This allows to use the same constant in different enumerations.

### Definition of two enumerations:

```
TYPE COLORS_1: (red, blue);
END_TYPE
TYPE COLORS_2: (green, blue, yellow);
END_TYPE
```

### Enumeration value "blue" in a function block:

```
// Declaration:
colorvar1 : COLORS_1;
colorvar2 : COLORS_2;

// Implementation:

(* possible: *)
colorvar1 := colors_1.blue;
colorvar2 := colors_2.blue;

(* not possible: *)
colorvar1 := blue;
colorvar2 := blue;
```

2. The basic data type of the enumeration - preset as INT - can be defined by another data type.





## Programming Reference

*Basic data type for the enumeration BigEnum is supposed to be DINT:*

```
TYPE BigEnum : (yellow, blue, green:=16#8000) DINT;  
END_TYPE
```

## Subrange Types

with a value range that only includes a subset of a basic type. Note that it can . The declaration can be made in a , but a variable can also be declared directly with a subrange type.

*Syntax for the declaration as Data Unit Type (DUT):*

```
TYPE <Name>:<Inttype> (<ll>..
```

<Name>	Has to be a valid IEC identifier
<Inttype>	One of these data types is possible: SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, (BYTE, WORD, DWORD, LWORD).
<ll>	Constant that has to be compatible with the basic data type and that determines the lower limit of the range type. The lower limit itself belongs to this range.
<ul>	Constant that has to compatible with the basic data type and that determines the upper limit of the range type. The upper limit itself belongs to this range.

### Examples:

*Subranges in type definition:*

```
TYPE  
  SubInt : INT (-4095..4095);  
END_TYPE
```

*Direct declaration of a variable with a subrange type:*

```
VAR  
  i : INT (-4095..4095);  
  ui : UINT (0..10000);  
END_VAR
```

If a constant is assigned to a subrange type (in the declaration or in the statement) and this constant is not within the range (e.g. 1=5000), an error message is output.

### Functions for range monitoring

To monitor the range limits of a subrange type at runtime, the function "CheckRangeSigned" or "CheckRangeUnsigned" has be integrated into the application.

They are integrated into the application as an object using the command .

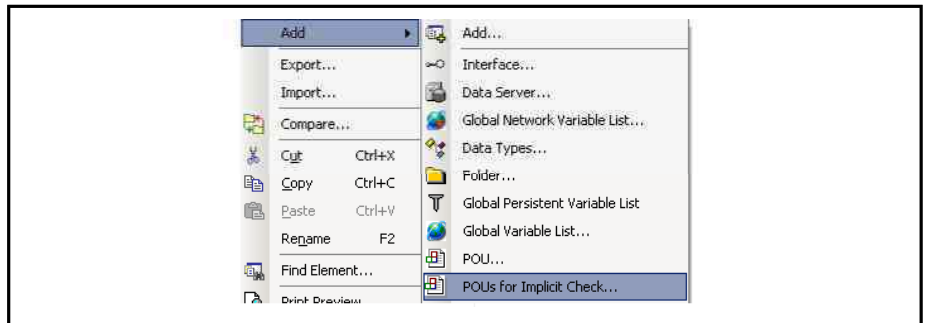


Fig.5-24: Add -> POU's for Implicit Check

After selecting the respective checkbox, select a program language and confirm it with "Open". The "CheckRangeSigned" or "CheckRangeUnsigned" function opens in the respective editor.



Fig.5-25: Selecting "CheckRangeSigned" or "CheckRangeUnsigned"

The declaration of the functions is specified for all languages and may not be modified! Only local variables may be added. A suggestion for programming the functions is available in ST. It may be freely modified.

The task of the functions "CheckRangeSigned" or "CheckRangeUnsigned" is to monitor range limits and instances and their exceedances.

If range limits are exceeded, an error flag can be set or a value can be changed for example.

The function is called implicitly when a value is assigned to a variable of the type subrange.



To maintain the monitoring functionality, the declaration part of the function may not be modified!

If a value is assigned to a variable of a signed subrange type, this causes an automatic call of the "CheckRangeSigned" function. The function, which limits the assignment value to the subrange specified at the variable declaration, is implemented in ST by default as follows:

*Example:*

```
// Automatically generated code : DO NOT EDIT
FUNCTION CheckRangeSigned : DINT
VAR_INPUT
```

## Programming Reference

```

value, lower, upper: DINT; END_VAR

// Automatically generated code :
// It deals with an implementation suggestion.
IF (value < lower) THEN
    CheckRangeSigned := lower;
ELSIF(value > upper) THEN
    CheckRangeSigned := upper;
ELSE
    CheckRangeSigned := value;
END_IF

```

At the call, the functions of the following input parameters are transferred.

- Value: Value of the variable to be assigned by the subrange type
- lower: Lower range limit
- upper: Upper range limit

The return value is the assigned value itself as long as it is within the valid range.

Otherwise, the upper or lower limit is returned depending on the violation of the subrange.

The

*assignment without limit monitoring*

```
i:=10*y
```

is now replaced explicitly by

*the assignment with limit monitoring*

```
i := CheckRangeSigned(10*y, -4095, 4095);
```

For example, if "y" has the value "1000", the value "10\*1000=10000" is not assigned to the variable "i" as intended in the code, but the value of the upper range limit instead which is "4095".

The same applies for the "CheckRangeUnsigned" function.



If neither the "CheckRangeSigned" nor the "CheckRangeUnsigned" function is available, the subrange is not checked for the respective variables at runtime! In this case, any value between -32768 and 32767 can be assigned to a variable of a subrange type of the data type INT!



When the functions "CheckRangeSigned" and "CheckRangeUnsigned" are integrated, endless loops can result.

This is the case for example if the counter variable of a FOR loop is a subrange type and the counting range of the loop leaves the defined subrange!

*Example:*

```

VAR
    ui : UINT (0..10000);
END_VAR

FOR ui:=0 TO 10000 DO
    ...
END_FOR

```

The FOR loop is never exited, since the monitoring function "CheckRangeSigned" prevents "ui" from being set to a value greater than 1000.



## 5.3 IEC Operators and Standard-Extending Functions

### 5.3.1 IEC Operators and Standard-Extending Functions, General Information

IndraLogic 2G supports all IEC operators. In contrast to the default functions, these operators are implicitly known across the entire project.

In addition to the IEC operators, the following operators, which are not described by the IEC standard, are also supported: ANDN, ORN, XORN, INDEXOF and SIZEOF (see arithmetic operators), ADR and BITADR and content operators (see address operators), some "namespace operators".

Operators are used in a function block like functions.



For operations with floating point data types, the result of calculation depends on the target system hardware used!

- For REAL variables, the "double precision" calculation is used for the controls L45/L65.
- Due to the processor, this approach is not possible for L25 controls. Therefore, it is calculated with the "single precision" method.

#### *Categories of operators:*

- Assignment operators,
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

### 5.3.2 Arithmetic Operators Arithmetic Operators, Overview

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

- 
- 
- 
- 
- 
- 
- 
- 

The following operators extend the standard:

- 
-

Programming Reference

**ADD**

**IEC operator:** Addition of variables.

Types allowed: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL and LREAL.

Two TIME variables can be added; the result can also be from a different time data type,

(e.g.  $t\#45s + t\#50s = t\#1m35s$ ).

<b>LD</b>	7		
<b>ADD</b>	2		
<b>ADD</b>	4		
<b>ADD</b>	7		
<b>ST</b>	iVar		

Fig.5-26: Operator ADD in IL

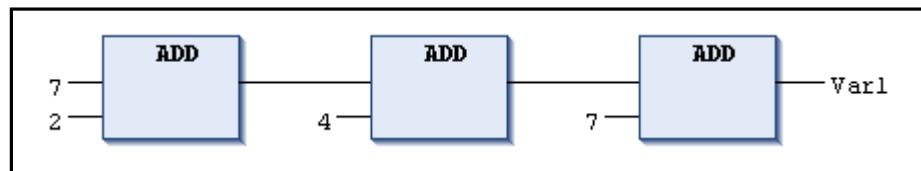


Fig.5-27: Operator ADD in FBD

Operator ADD in ST

```
var1 := 7+2+4+7;
```

**MUL**

**IEC operator:** Multiplication of variables.

Types allowed: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL and LREAL.

<b>LD</b>	7		
<b>MUL</b>	2		,
	4		,
	7		
<b>ST</b>	Var1		

Fig.5-28: Operator MUL in IL

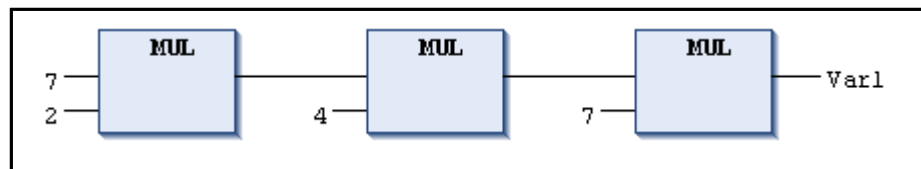


Fig.5-29: Operator MUL in FBD

Operator MUL in ST

```
var1 := 7*2*4*7;
```

**SUB**

**IEC operator:** Subtraction of one variable from another.

Programming Reference

Types allowed: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL and LREAL.

A TIME variable can also be subtracted from a variable of a different TIME type. The result is located in a variable of a third TIME type. However, note that negative TIME values are undefined.

<b>LD</b>		7		
<b>SUB</b>		2		
<b>ST</b>		Var1		

Fig.5-30: Operator SUB in IL

**Result:** Content of Var1 is 5

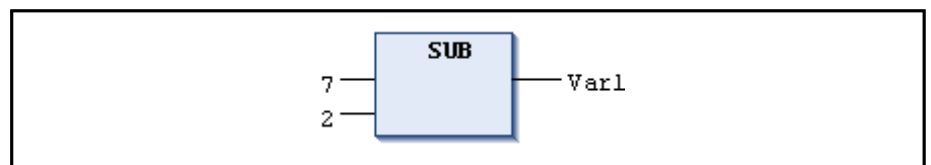


Fig.5-31: Operator SUB in FBD

Operator SUB in ST

```
var1 := 7-2;
```

**DIV**

**IEC operator:** Division of one variable by another.

Types allowed: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL and LREAL.

<b>LD</b>		8		
<b>DIV</b>		2		
<b>ST</b>		Var1		

Fig.5-32: Operator DIV in IL

**Result:** Content of Var1 is 4

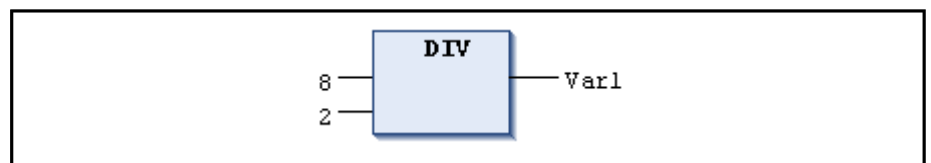


Fig.5-33: Operator DIV in FBD

Operator DIV in ST

```
var1 := 8/2;
```



The behavior when dividing by zero can depend on the target system!

The behavior at a division can be preset by the user.

**Functions for check**

The functions **CheckDivInt**, **CheckDivLint**, **CheckDivReal** and **CheckDivLReal** can be used to monitor the value of a divisor in order to prevent division by 0. After they are integrated into the application, its call automatically precedes each division that occurs in the corresponding code.



Programming Reference

They have to be integrated into the application as an object using the command .

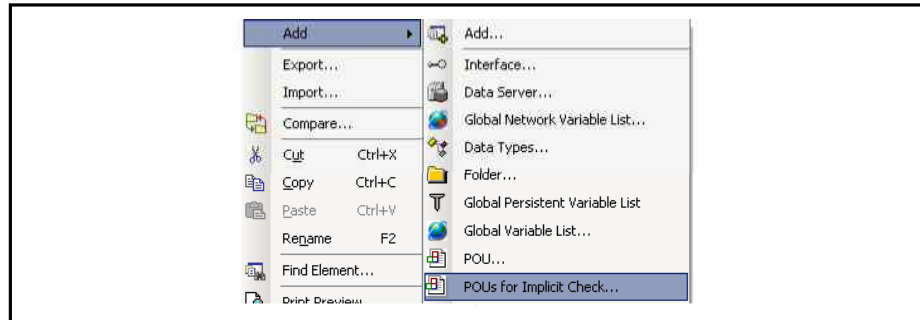


Fig.5-34: Add -> POU's for Implicit Check



Fig.5-35: Selection "CheckDivReal"

The declaration part of the functions is fixedly determined and may not be changed. Only local variables may be added. A suggestion to implement the functions is available in ST.

See the following example to implement the function CheckDivReal:

*Default implementation of the function CheckDivReal in ST:*

```
// Implicitly generated code : DO NOT EDI
FUNCTION CheckDivReal : REAL
VAR_INPUT
  divisor:REAL;
END_VAR
// Implicitly generated code :
// only an suggestion for implementation
IF divisor = 0 THEN
  CheckDivReal:=1;
ELSE
  CheckDivReal:=divisor;
END_IF;
```

The operator DIV uses the output of the function CheckDivReal as a divisor.

In the example program below, division by 0 is prevented, since the function "CheckDiv Real" changes the value of the divisor "d" - implicitly initiated with "0" - to "1" before the division is executed. The result of the division is thus 799.

*Example in ST:*

```
PROGRAM PLC_PRG
VAR
  erg:REAL;
  v1:REAL:=799;
  d:REAL;
END_VAR

erg:= v1 / d;
```

**MOD**

**IEC operator:** Modulo division of one variable by another.

Types allowed: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT. This function returns the integer remainder of the division as a result.

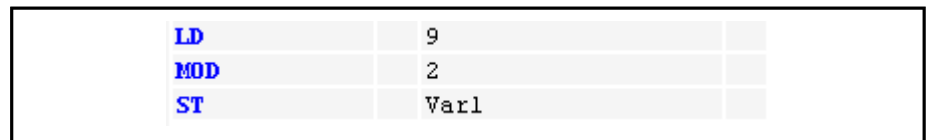


Fig.5-36: Operator MOD in IL

Result: Content of Var1 is 1

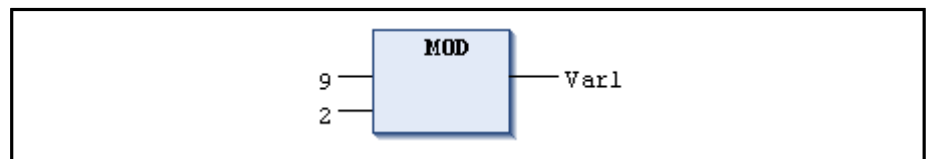


Fig.5-37: Operator MOD in FBD

*Operator MOD in ST:*

```
var1 := 9 MOD 2;
```

**MOVE**

**IEC operator:** Assignment of a variable to another variable of a corresponding type.

Since MOVE is available as function block in the CFC, FBD and LD editors, the EN/ENO functionality can also be used for a variable assignment.

**Example in CFC in connection with the EN/ENO function:**

Only if en\_i is TRUE, the value of variable var2 is assigned to variable var1.

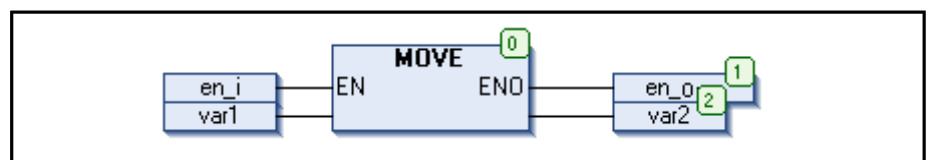


Fig.5-38: Operator MOVE in CFC

Without this EN/ENO functionality, using the MOVE operator is not reasonable, since the code piece becomes a simple assignment.

Programming Reference

*Example:*

Example: Operator MOVE in IL (content of var2 is the content of var1):

<b>LD</b>	var1		
<b>MOVE</b>			
<b>ST</b>	var2		

Fig.5-39: Operator MOVE in IL

This corresponds to:

<b>LD</b>	var1		
<b>ST</b>	var2		

Fig.5-40: Operators LD and ST in IL

*Operator MOVE in ST:*

```
ivar2 := MOVE(ivar1);
// you get the same result with:
ivar2 := ivar1;
```

**SIZEOF**

This arithmetic operator is not specified by the IEC 61131-3 standard. It can be used to specify the number of bytes needed by the specified variable x.

The SIZEOF operator always returns an unsigned value. The type of the return variable adjusts to the size of the variable x.

Return value of SIZEOF(x)	Data type of the constants implicitly used for the size found
$0 \leq \text{size of } x < 256$	USINT
$256 \leq \text{size of } x < 65536$	UINT
$65536 \leq \text{size of } x < 4294967296$	UDINT
$4294967296 \leq \text{size of } x$	ULINT

Fig.5-41:

*Declaration:*

```
arr1:ARRAY[0..4] OF INT;
Var1:INT;
```

*Operator SIZEOF in ST:*

```
var1 := SIZEOF(arr1); (* i.e.: var1:=USINT#10; *)
```

<b>LD</b>	arr1		
<b>SIZEOF</b>			
<b>ST</b>	Var1		

Fig.5-42: Operator Sizeof in IL

Result is 10

This arithmetic operator is not specified by the IEC 61131-3 standard.

**INDEXOF**

Programming Reference

The internal index of a function block can be determined with this function.

*Example: Operator INDEXOF in ST:*

```
var1 := INDEXOF(POU2);
```

### 5.3.3 Bit String Operators

#### Bit String Operators, Overview

The following bit string operators described by the IEC 61131-3 standard are supported:

- 
- 
- 
- 

Not yet available: The following bit string operators are supported in an extension of the standard:

- 
- 
- 

#### AND

Bit string operators compare the bits from two or more operands.

**IEC bit string operator:** AND bit by bit from bit operands. If the input bits are 1, the output bit is 1. Otherwise, it is 0.

Types allowed: BOOL, BYTE, WORD, DWORD, LWORD.

*Declaration:*

```
VAR
  Var1:BYTE;
END_VAR
```

<b>LD</b>	2#1001_0011	
<b>AND</b>	2#1000_1010	
<b>ST</b>	var1	

Fig.5-43: Operator AND in IL

Result: Content of Var1 is 2#1000\_0010

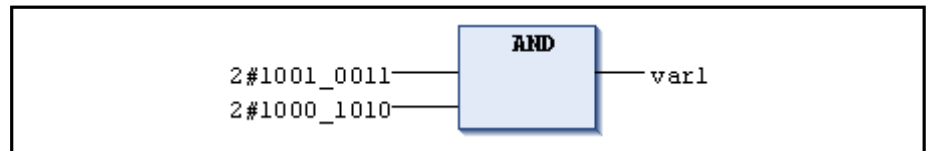


Fig.5-44: Operator AND in FBD

*Operator AND in ST:*

```
var1 := 2#1001_0011 AND 2#1000_1010
```

#### ANDN

### In preparation ###

Programming Reference

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

**OR**

**IEC bit string operator:** OR bit by bit from bit operands. If at least one of the input bits is 1, the output bit is 1. Otherwise, it is 0.

Types allowed: BOOL, BYTE, WORD, DWORD, LWORD.

*Declaration:*

```
VAR
  Var1:BYTE;
END_VAR
```

<b>LD</b>	16#FF		
<b>OR</b>	16#a2		
<b>ST</b>	var1		

Fig.5-45: Operator OR in IL

Result: Content of var1 is 2#1001\_1011

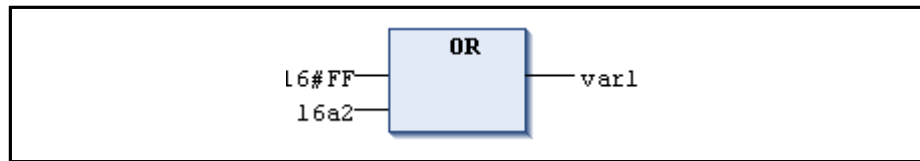


Fig.5-46: Operator OR in FBD

*Operator OR in ST:*

```
Var1 := 2#1001_0011 OR 2#1000_1010
```

**ORN**

### In preparation ###

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

**XOR**

**IEC bit string operator:** XOR bit by bit from bit operands. If only one of the two bit inputs is 1, the result is 1. If both inputs are 0 or 1, the result is 0.

Types allowed: BOOL, BYTE, WORD, DWORD, LWORD.



Note the behavior of the XOR function block in extended form, that is with more than two inputs: The inputs are checked in pairs and, in turn, the respective results are then compared with each other (meets the standard, but not necessarily expectations).

*Declaration:*

```
VAR
  Var1:BYTE;
END_VAR
```

<b>LD</b>	2#1001_0011		
<b>XOR</b>	2#1000_1010		
<b>ST</b>	var1		

Fig.5-47: Operator XOR in IL

Result: var1 has the content 2#0001\_1001

Programming Reference

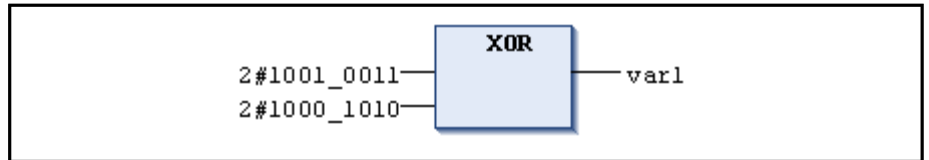


Fig.5-48: Operator XOR in FBD

Operator XOR in ST:

```
Var1 := 2#1001_0011 XOR 2#1000_1010
```

**XORN**

### In preparation ###

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

**NOT**

**IEC bit string operator:** NOT bit by bit of a bit operand. The output bit is 1 if the corresponding input bit is 0 and vice versa.

Types allowed: BOOL, BYTE, WORD, DWORD, LWORD.

Declaration:

```
VAR
  Var1:BYTE;
END_VAR
```

<b>LD</b>	2#1001_0011	
<b>NOT</b>		
<b>ST</b>	var1	

Fig.5-49: Operator NOT in IL

Result: Content of var1 is 2#0110\_1100

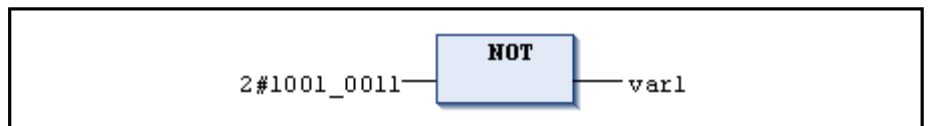


Fig.5-50: Operator NOT in FBD

Operator NOT in ST:

```
Var1 := NOT 2#1001_0011
```

**5.3.4 Bit Shift Operators**

**Bit Shift Operators, Overview**

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

- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_

**SHL**

**IEC operator:** Shifting an operand to the left bit by bit.



Programming Reference

**SYNTAX:**

---

erg:= SHL (in, n)

---

**in:** Operand to be shifted to the left.

**n:** Number of bits by which **in** is shifted to the left.

If **n** exceeds the data type width, BYTE, WORD, DWORD and LWORD operands are filled with zeroes, while fixed sign type operands, e.g. INT, are shifted arithmetically, i.e. they are filled with the value of the top bit.



Note that the number of bits considered for the arithmetic operations is specified by the data type of the input variable "in". If this is a constant, the smallest possible data type is considered. The data type of the output variable does not affect the arithmetic operation.

In the following example in a hexadecimal display, note that the different results for erg\_byte and erg\_word depend on the data type of the input variable (BYTE or WORD), although the values of the input variables in\_byte and in\_word are equal.

*Operator SHL in ST:*

```
PROGRAM shl_st
VAR
  in_byte : BYTE:=16#45;
  in_word : WORD:=16#45;
  erg_byte : BYTE;
  erg_word : WORD;
  n: BYTE :=2;
END_VAR
erg_byte:=SHL(in_byte,n); (* 16#14*)
erg_word:=SHL(in_word,n); (* 16#01141*)
```

<b>LD</b>	in_byte	
<b>SHL</b>	2	
<b>ST</b>	erg_byte	

Fig.5-51: Operator SHL in IL

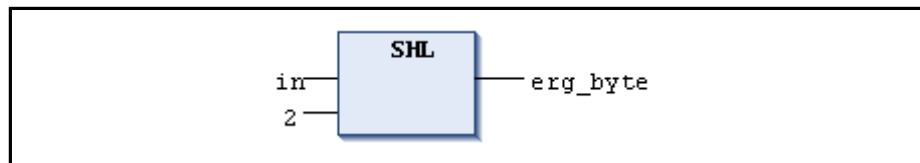


Fig.5-52: Operator SHL in FBD

**SHR**

**IEC operator:** Shifting an operand to the right bit by bit.

**SYNTAX:**

---

erg:= SHR (in, n)

---

**in:** Operand to be shifted to the right.

**n:** Number of bits by which **in** is shifted to the right.

If **n** exceeds the data type width, BYTE, WORD, DWORD and LWORD operands are filled with zeroes, while fixed sign type operands, e.g. INT, are shifted arithmetically, i.e. they are filled with the value of the top bit.



Note that the number of bits considered for the arithmetic operations is specified by the data type of the input variable "in". If this is a constant, the smallest possible data type is considered. The data type of the output variable does not affect the arithmetic operation.

In the following example in a hexadecimal display, note that the different results for erg\_byte and erg\_word depend on the data type of the input variable (BYTE or WORD), although the values of the input variables in\_byte and in\_word are equal.

*Operator SHR in ST:*

```
PROGRAM shr_st
VAR
  in_byte : BYTE:=16#45;
  in_word : WORD:=16#45;
  erg_byte : BYTE;
  erg_word : WORD;
  n: BYTE :=2;
END_VAR
erg_byte:=SHR(in_byte,n); (* 16#11 *)
erg_word:=SHR(in_word,n); (* 16#0011 *)
```

LD	in_byte	
SHR	2	
ST	erg_byte	

Fig.5-53: Operator SHR in IL

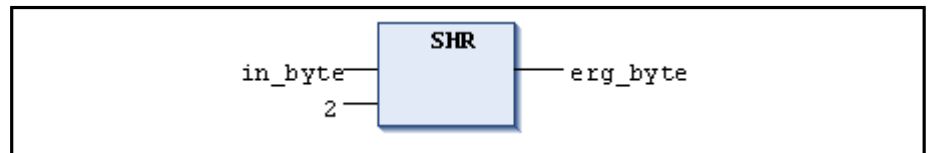


Fig.5-54: Operator SHR in FBD

**ROL**

**IEC operator:** Rotation of an operand bit by bit to the left.

**SYNTAX:**

```
erg:= ROL (in, n)
```

Data types allowed: BYTE, WORD, DWORD and LWORD.

**in** is shifted **n** times 1 bit to the left and at the same time, the bit with the outermost position at the left is inserted again from the right.



Note that the number of bits considered for the arithmetic operations is specified by the data type of the input variable "in". If this is a constant, the smallest possible data type is considered. The data type of the output variable does not affect the arithmetic operation.

In the following example in a hexadecimal display, note that the different results for erg\_byte and erg\_word depend on the data type of the input variable (BYTE or WORD), although the values of the input variables in\_byte and in\_word are equal.

*Operator ROL in ST:*

```
PROGRAM rol_st
VAR
  in_byte : BYTE:=16#45;
```

Programming Reference

```

in_word : WORD:=16#45;
erg_byte : BYTE;
erg_word : WORD;
n: BYTE :=2;
END_VAR
erg_byte:=ROL(in_byte,n); (* 16#15 *)
erg_word:=ROL(in_word,n); (* 16#0114 *)

```

<b>LD</b>	in_byte	
<b>ROL</b>	n	
<b>ST</b>	erg_byte	

Fig.5-55: Operator ROL in IL

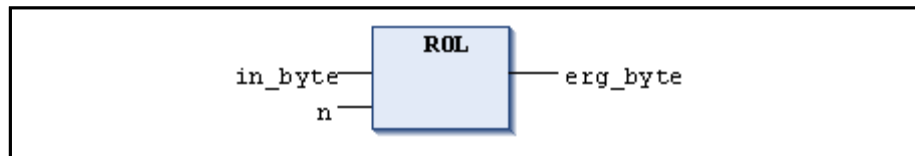


Fig.5-56: Operator ROL in FBD

**ROR**

**IEC operator:** Rotation of an operand bit by bit to the right.

**SYNTAX:**

```
erg = ROR (in, n)
```

Data types allowed: BYTE, WORD, DWORD and LWORD.

**in** is shifted **n** times 1 bit to the right and at the same time, the bit with the outermost position at the right is inserted again from the left.



Note that the number of bits considered for the arithmetic operations is specified by the data type of the input variable "in". If this is a constant, the smallest possible data type is considered. The data type of the output variable does not affect the arithmetic operation.

In the following example in a hexadecimal display, note that the different results for erg\_byte and erg\_word depend on the data type of the input variable (BYTE or WORD), although the values of the input variables in\_byte and in\_word are equal.

**Operator ROR in ST:**

```

PROGRAM ror_st
VAR
in_byte : BYTE:=16#45;
in_word : WORD:=16#45;
erg_byte : BYTE;
erg_word : WORD;
n: BYTE :=2;
END_VAR
erg_byte:=ROR(in_byte,n); (* 16#51*)
erg_word:=ROR(in_word;n); (* 16#4011*)

```

<b>LD</b>	in_byte	
<b>ROR</b>	n	
<b>ST</b>	erg_byte	

Fig.5-57: Operator ROR in IL

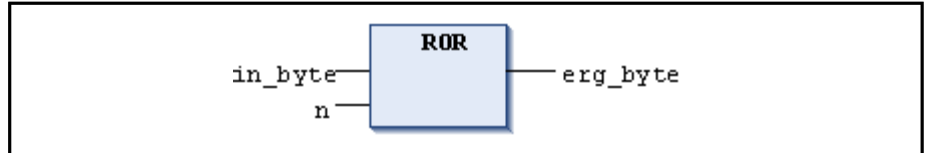


Fig.5-58: Operator ROR in FBD

## 5.3.5 Selection Operators

### Selection Operators, Overview

All selection operators can also be used for constants and variables.  
To ensure clarity in the examples, the following includes only constants:

- 
- 
- 
- 
- 

### SEL

**IEC selection operator:** Binary selection. **G** specifies whether **IN0** or **IN1** is assigned to the variable **OUT**.

**SYNTAX:**

---

```
OUT := SEL(G, IN0, IN1) // bedeutet:
                        // OUT := IN0; if G=FALSE
                        // OUT := IN1; if G=TRUE.
```

---

Data types allowed:

IN0, IN1, OUT: Can be any type.

G: BOOL.



An expression preceding IN0 is not calculated if G is TRUE!

---

*Operator SEL in IL:*

---

```
LD TRUE
SEL 3,4 // IN0= 3, IN1= 4
ST Var1 // 4

LD FALSE
SEL 3,4 // IN0= 3, IN1= 4
ST Var1 // 3
```

---

*Operator SEL in ST:*

---

```
Var1:=SEL(TRUE,3,4); // 4
```

---

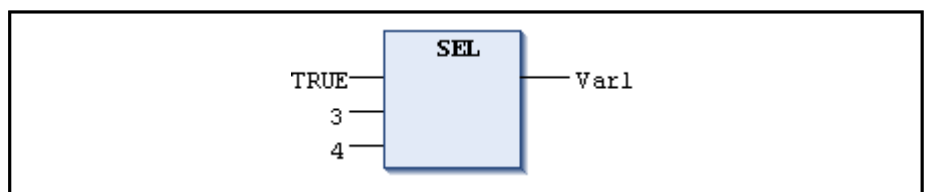


Fig.5-59: Operator SEL in FBD

Programming Reference

**MAX**

**IEC selection operator:** Maximum function. Returns the greater of the two values.

**SYNTAX:**

---

```
OUT := MAX(IN0, IN1)
```

---

IN0, IN1 and OUT can be any type.

<b>LD</b>	90		
<b>MAX</b>	30		
<b>MAX</b>	40		
<b>MAX</b>	77		
<b>ST</b>	Var1		

Fig.5-60: Operator MAX in IL

Result is 90

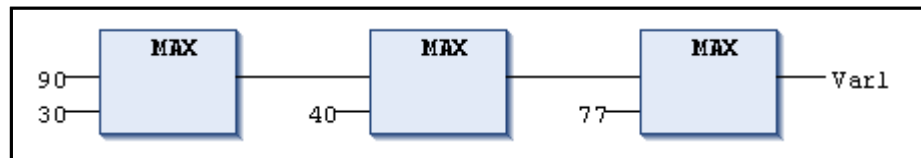


Fig.5-61: Operator MAX in FBD

**Operator MAX in ST**

---

```
Var1:=MAX(30,40); // 40
Var1:=MAX(40,MAX(90,30)); // 90
```

---

**MIN**

**IEC selection operator:** Minimum function. Returns the smaller of the two values.

**SYNTAX:**

---

```
OUT := MIN(IN0, IN1)
```

---

IN0, IN1 and OUT can be any type.

<b>LD</b>	90		
<b>MIN</b>	30		
<b>MIN</b>	40		
<b>MIN</b>	77		
<b>ST</b>	Var1		

Fig.5-62: Operator MIN in IL

Result is 30

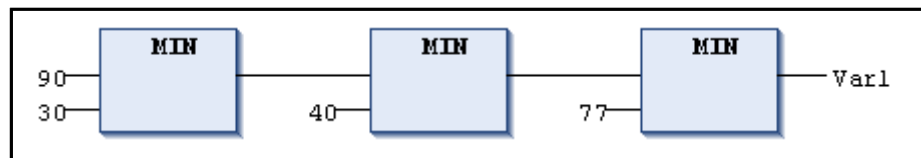


Fig.5-63: Operator MIN in FBD

**Operator MIN in ST**

---

```
Var1:=MIN(90,30); // 30;
Var1:=MIN(MIN(90,30),40); // 30;
```

---

## LIMIT

Programming Reference

IEC selection operator: Limitation

**SYNTAX:**

```
OUT := LIMIT(Min, IN, Max) // means:
OUT := MIN (MAX (IN, Min), Max)
```

"Max" is the upper limit value, "Min" the lower limit value for the result. If the value IN exceeds the upper limit Max, then LIMIT returns Max. If IN is lower than Min, then the result is Min.

IN and OUT can be any type.

<b>LD</b>	90		
<b>LIMIT</b>	30	,	
	80		
<b>ST</b>	Var1		

Fig.5-64: Operator LIMIT in IL

Result is 80

Operator LIMIT in ST

```
Var1:=LIMIT(30,90,80); // 80
```

## MUX

IEC selection operator: Operator: Multiplexer

**SYNTAX:**

```
OUT := MUX(K, IN0, ..., INn) // means:
OUT := INk.
```

IN0, ...,INn and OUT can be any type.

K has to be of types BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT or ULINT.

MUX selects the kth value from a set of values.

The first value is K=0.

If K is greater than the number of the other inputs (n), the final value is forwarded (INn).



To improve performance at runtime, only the expression preceding  $IN_k$  is calculated!

In contrast, all branches are calculated in the simulation.

<b>LD</b>	0		
<b>MUX</b>	30	,	
	40	,	
	50	,	
	60	,	
	70	,	
	80		
<b>ST</b>	Var1		

Fig.5-65: Operator MUX in IL

Result is 30



Programming Reference

*Operator MUX in ST*

```
Var1 := MUX(0, 30, 40, 50, 60, 70, 80); // 30;
```

## 5.3.6 Relational Operators

### Relational Operators, Overview

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

- 
- 
- 
- 
- 
- 
- 

These are Boolean operators each comparing two inputs (first and second operand).

GT

**IEC relational operator:** greater than.

A Boolean operator with a result of TRUE if the first operand is greater than the second.

*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.

<b>LD</b>	20		
<b>GT</b>	30		
<b>ST</b>	Var1		

Fig.5-66: Operator GT in IL

Result is FALSE

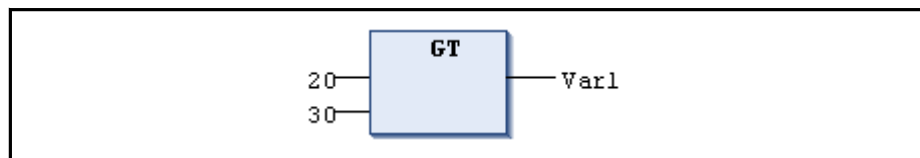


Fig.5-67: Operator GT in FBD

*Operator GT in ST*

```
VAR1 := 20 > 30 > 40 > 50 > 60 > 70; // FALSE
```

LT

**IEC relational operator:** Less than.

A Boolean operator with the result TRUE if the first operand is less than the second.

The operands can be any of the following types: Programming Reference

- 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.

<b>LD</b>	20		
<b>LT</b>	30		
<b>ST</b>	Var1		

Fig.5-68: Operator LT in IL

Result is TRUE

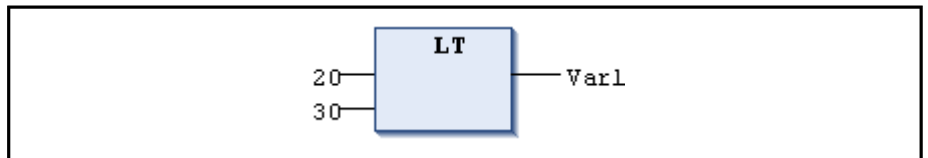


Fig.5-69: Operator LT in FBD

Operator LT in ST

```
VAR1 := 20 < 30; // TRUE
```

## LE

**IEC relational operator:** Less than or equal to.

A Boolean operator with the result TRUE if the first operand is less 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.

<b>LD</b>	20		
<b>LE</b>	30		
<b>ST</b>	Var1		

Fig.5-70: Operator LE in IL

Result is TRUE

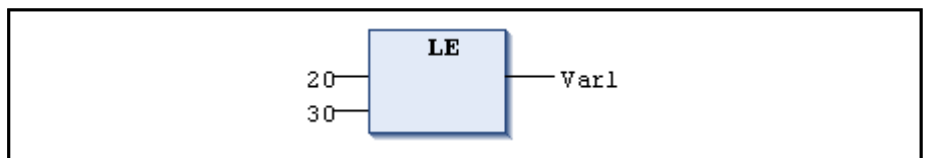


Fig.5-71: Operator LE in FBD

Operator LE in ST

```
VAR1 := 20 <= 30; // TRUE
```