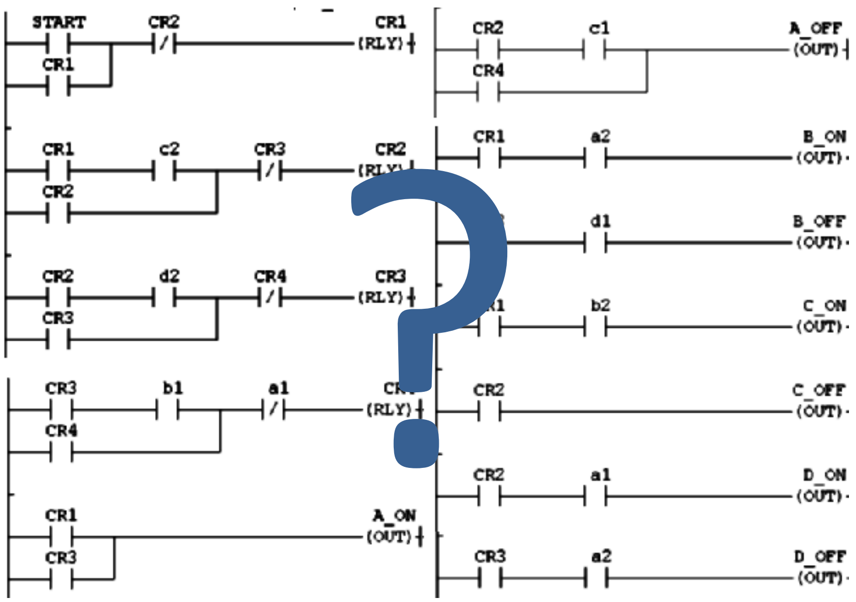
# Software model of the standard IEC 61131-3

Industrial control

KOVÁCS Gábor
gkovacs@iit.bme.hu

# The SMORES principle

- A good software is
  - **S**calable
  - **MO**dular
  - **R**eusable
  - **E**xtensible
  - **S**imple



# Origin of the standard IEC 61131

- Manufacturers develop their products individually
- Common practices and principles of development have evolved
- Use of these principles in development tools of different implementers (e.g. Rockwell, Siemens etc.) is everything but uniform
- 1979: official demand for a uniform standard
- 1985: publication of the first edition of the standard IEC 61131

# The standard IEC 61131

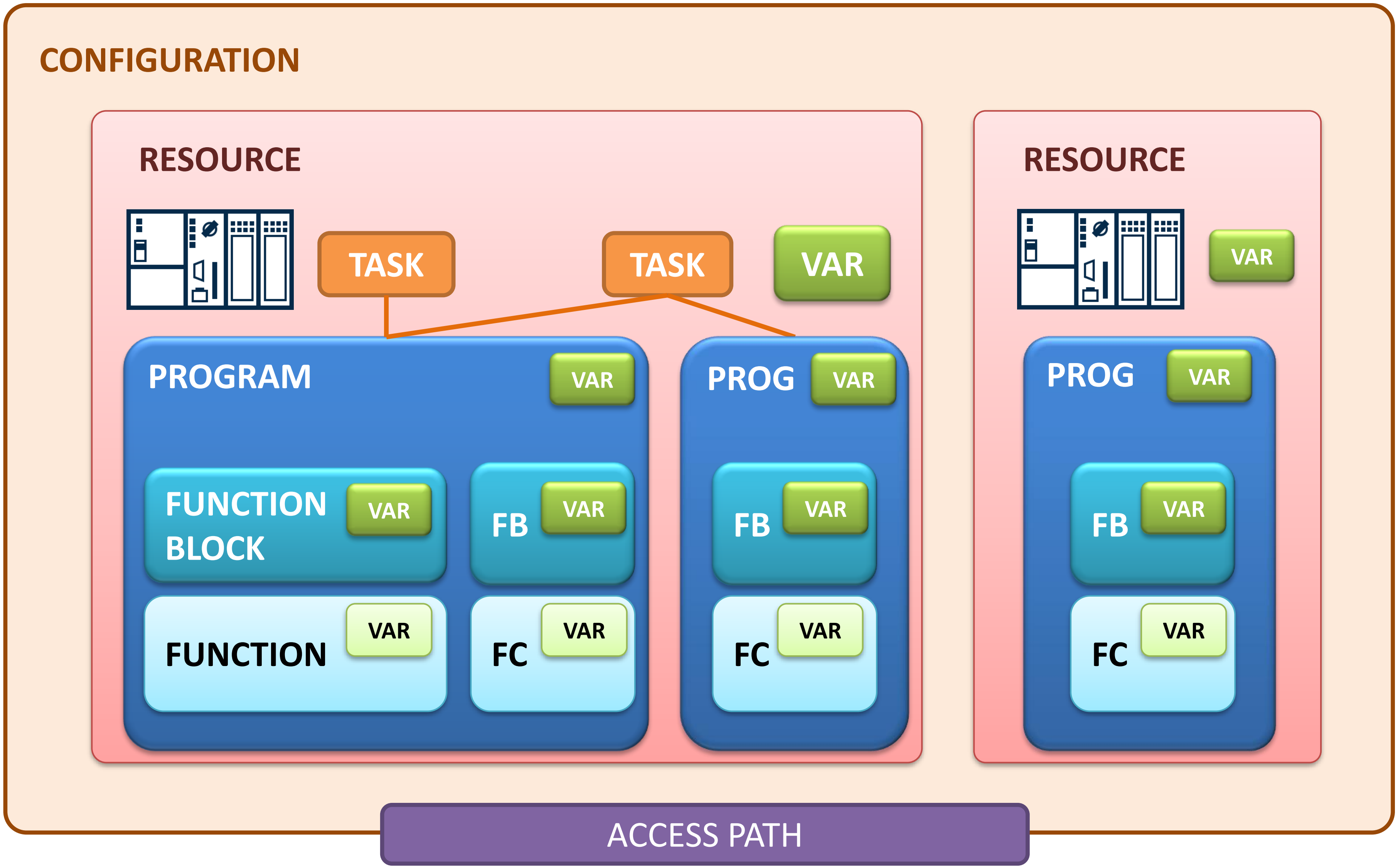
- IEC 61131 – Programmable controllers
  *(Automates Programmables)*

- Parts of the standard
  1. General information (ed. 2, 2003)
  2. Equipment requirements and tests (ed. 3, 2017)
  3. **Programming languages (ed. 3, 2013)**
  4. User guidelines (ed. 2, 2004)
  5. Communications (2000)
  6. Functional safety (2012)
  7. Fuzzy logic programming (2000)
  8. Guideline for the application and implementation of programming languages (ed. 3, 2017)
  9. Digital sensor interface (2013)
  10. PLCopen XML (2019)

# Application of the standard IEC 61131-3

- Initially major manufacturers used to treat the standard only as a recommendation
- State-of-the-art development environments have become mostly IEC 61131-3 compliant
- Principles are met and most features are available, however environments may
  - use different terms for the same features
  - not implement all features
  - have features not fully compliant with the standard

# Overview


CONFIGURATION
RESOURCE
TASK
TASK
VAR
PROGRAM
VAR
FUNCTION BLOCK
VAR
FB
VAR
FUNCTION
VAR
FC
VAR
PROG
VAR
FB
VAR
FC
VAR
RESOURCE
VAR
PROG
VAR
FB
VAR
FC
VAR
ACCESS PATH

# Comments

- Comments might be used in any textual part (declaration or body) of POUs
- Use of comments in graphical programs is implementer specific
- Single line comments are preceded by double slashes: `//`
- Multi-line comments are enclosed by `(*` and `*)` or by `/*` and `*/`
- Best practices:
  - start each line of the comment with `//` (if supported by the development environment)
  - do not embed multi-line comments
  - do not leave any commented code in the production version

# Identifiers

- Elements of the application (variables, types, POUs) can be referred to by their identifiers
- Indentifiers are strings of letters, numbers and underscores (e.g. `Level_Switch_1`)
- Character set of identifiers depends on the development environment
- Identifiers shall begin with a letter and shall not contain double underscores (`__`)
- Identifiers are <u>not</u> case sensitive
- Best practices:
  - UpperCamelCase
  - use characters of the English alphabet only

# Data types

- Elementary data types
  - bit string
  - numeric
  - date, time and duration
  - character and string
- Derived data types
  - directly derived data types
  - enumerated types
  - subrange
  - array
  - structure
- Generic data types (ANY)

# Bit string data types

| Data type | Description | Bits | Range | Initial value | Prefix |
|---|---|---|---|---|---|
| BOOL | Boolean | 1 | [0,1] | 0 | x |
| BYTE | Bit string | 8 | [0,…,16#FF] | 0 | b/by |
| WORD | Bit string | 16 | [0,…,16#FFFF] | 0 | w |
| DWORD | Bit string | 32 | [0,…,16#FFFF_FFFF] | 0 | dw |
| LWORD | Bit string | 64 | [0,…,16#FFFF_FFFF_FFFF_FFFF] | 0 | lw |

Bit string literals

- decimal: 143
- binary: 2#0000_1011
- hexadecimal: 16#0F
- underscore (_) can be used us unprocessed separator

Boolean literal

- TRUE / FALSE
- 1 / 0

# Integer data types

| Data type | Description | Bits | Range | Initial value | Prefix |
|---|---|---|---|---|---|
| SINT | Short integer | 8 | [-128,…,+127] | 0 | si |
| INT | Integer | 16 | [-32768,…,32767] | 0 | i |
| DINT | Double integer | 32 | $[-2^{31},\ldots,+2^{31}-1]$ | 0 | di |
| LINT | Long integer | 64 | $[-2^{63},\ldots,+2^{63}-1]$ | 0 | li |
| USINT | Unsigned short integer | 8 | [0,…,255] | 0 | usi |
| UINT | Unsigned integer | 16 | [0,…,65535] | 0 | ui |
| UDINT | Unsigned double integer | 32 | $[0,\ldots,+2^{32}-1]$ | 0 | udi |
| ULINT | Unsigned long integer | 64 | $[0,\ldots,+2^{64}-1]$ | 0 | uli |

Integer literals

- signed: decimal only (e.g. `-3`)
- unsigned: decimal, binary, hexadecimal
- underscore (`_`) can be used us unprocessed separator, e.g. `1_712`

# Floating point data types

| Data type | Description | Bits | Range | Initial value | Prefix |
|---|---|---|---|---|---|
| `REAL` | Single prec. floating point | 32 | According to IEEE 754 | 0.0 | `r` |
| `LREAL` | Double prec. floating point | 64 | | | `lr` |

Floating point literals

- decimal format
  - `-12.0`
  - `0.21`
- exponential format
  - `1.34E-3` : $1.34 \cdot 10^{-3}$
  - `-1.0E3` : $-1000$

# Duration

| Data type | Description | Initial value | Prefix |
|---|---|---|---|
| `TIME` (`T`) | Duration (range is implementer specific)<br>Common units: day, hour, min, sec, ms | `T#0s` | `tim` |
| `LTIME` (`LT`) | Duration (64 bits, ns resolution) | `LT#0s` | `ltim` |

Note: resolution of the representation and accuracy of timing are not required to be the same

# Duration literals

- Format:

  `<T|TIME>#<duration>`

  `<LT|LTIME>#<duration>`

- `duration`: combination of values and time units in descending order
- Values are integers except for the least significant one (might be real)
- Duration might be negative
- Overflow in the value corresponding to the most significant unit is allowed
- Underscore separators might be used

| Symbol | Time unit |
|---|---|
| `d` | Day |
| `h` | Hour |
| `m` | Minute |
| `s` | Second |
| `ms` | ms |
| `us` | µs |
| `ns` | ns |

Although the standard allows the use of µs and ns units, these are not commonly supported by the development environments.

# Example – Duration literals

- `t#2d_3h_4m` – 2 days 3 hours 4 minutes
- `t#3h12.3s` – 3 hours 12.3 seconds
- „Overflow" is permitted:

  `t#25h = t#1d1h` – 1 day 1 hour

# Date and time

| Data type | Description | Initial value | Prefix |
|---|---|---|---|
| DATE (D) | Date: YYYY-MM-DD | Implementer specific (e.g. d#0001-01-01) | dt |
| LDATE (LD) | Date: YYYY-MM-DD | ld#1970-01-01 | ldt |
| TIME_OF_DAY (TOD) | Time: hh:mm:ss.s | Implementer specific (commonly tod#00:00:00) | tod |
| LTIME_OF_DAY (LTOD) | Time: hh:mm:ss.s | ltod#00:00:00 | ltod |
| DATE_AND_TIME (DT) | Date and time: YYYY-MM-DD-hh:mm:ss.s | Implementer specific (e.g. dt#0001-01-01-00:00:00) | dt |
| LDATE_AND_TIME (LDT) | Date and time: YYYY-MM-DD-hh:mm:ss.s | dt#1970-01-01-00:00:00 | ldt |

Representation of long (L) types is a 64 bit unsigned integer, which stores the nanoseconds elapsed since the date/time corresponding to the initial value. Resolution of representation is not required to be the same as resolution of the real-time clock.

# Date and time literals

- Date

```
<DATE|D|LD>#<YYYY>-<MM>-<DD>
D#1986-02-11
```

- Time of day

```
<TIME_OF_DAY|TOD|LTOD>#<hh>:<mm>:<ss.s>
TOD#21:12:3.2
```

- Date and Time

```
<DATE_AND_TIME|DT|LDT>#<YYYY>-<MM>-<DD>-
<hh>:<mm>:<ss.s>
DT#1986-02-11-19:07:21.6
```

# Characters and strings

| Data type | Description | Initial value | Prefix |
|---|---|---|---|
| `CHAR` | Single-byte encoded character | ‘’ (empty) | `c` |
| `WCHAR` | Double-byte encoded character | ”” (empty) | `wc` |
| `STRING` | Variable-length character string (single-byte encoded) | ‘’ (empty) | `str` |
| `WSTRING` | Variable-length character string (double-byte encoded) | ”” (empty) | `wstr` |

Maximal length of string variables might be limited during declaration, e.g. `STRING[4]`

# Character and string literals

- Single-byte encoded characters/strings
  - `’a’`,`’Robin Hood’`
  - Special characters: `$` prefix (e.g. `$’`, `$R`, `$$`)
  - Character codes: `$<hex>`, where `<hex>` is the 8-bit code of the character, e.g. `$4F`

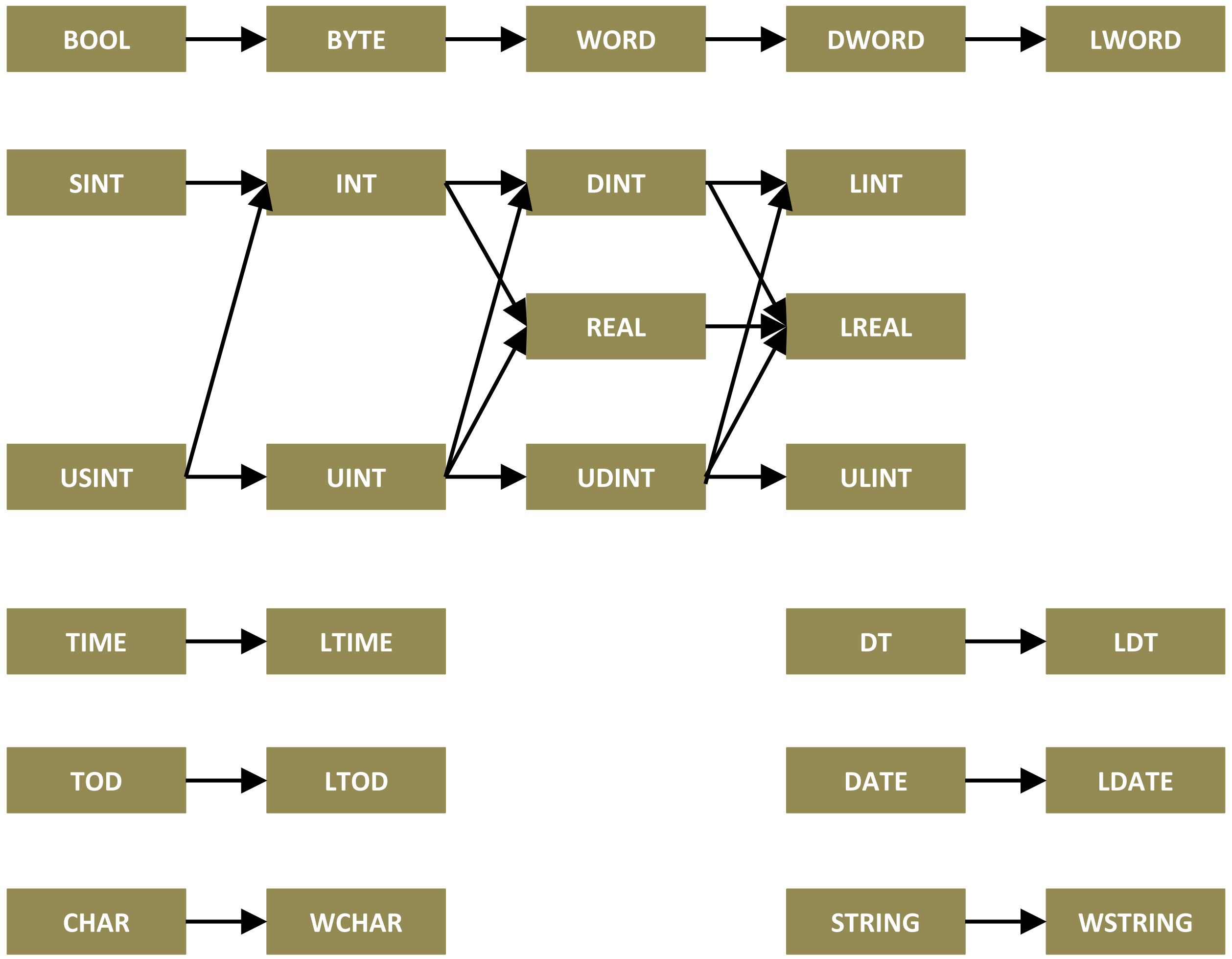
- Double byte encoded characters/strings
  - `”á”`,`”Rózsa Sándor”`
  - Special characters: `$` prefix (e.g. `$”`, `$R`, `$$`)
  - Character codes: `$<hex>`, where `<hex>` is the 16-bit code of the character, e.g. `$007A`

# Type conversion

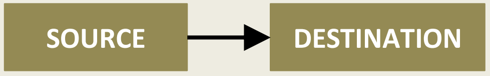
- Implicit type conversion
  - keeps value and accuracy
  - can be used in function or function block calls or in assignments
  - implicit conversion is available for defined source and destination data types only

- Explicit type conversion
  - supported by various type conversion functions

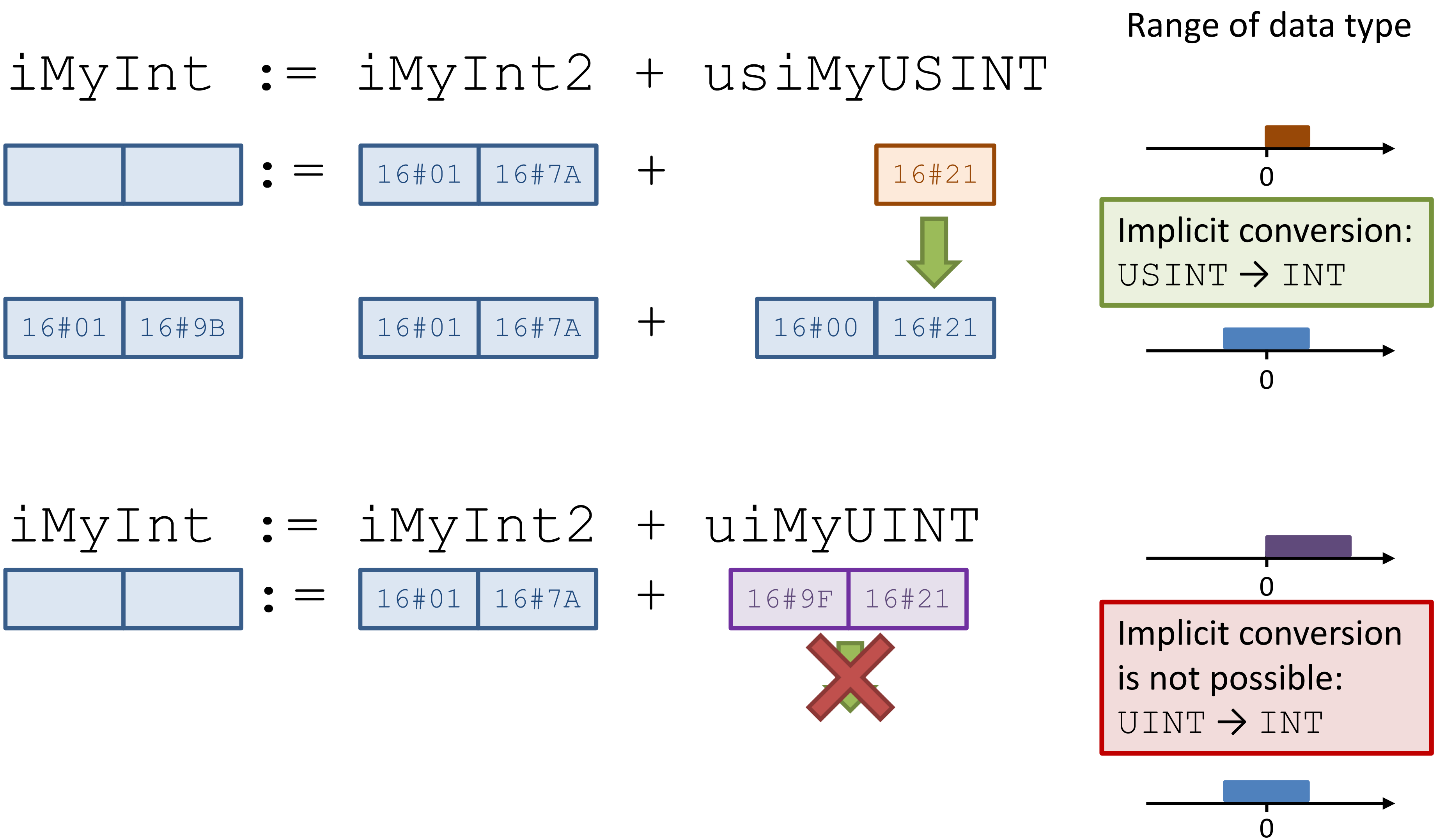
# Implicit type conversion




Implicit type conversion is supported from source data type to destination data type



# Example - Implicit type conversion



# Typed literals

- If not specified explicitly, types of numeric and string literals are assigned automatically

- Type of a literal can be specified explicitly by a prefix containing the data type followed by `#`
  - `INT#5`
  - `BOOL#0`
  - `WSTRING#'HELLO WORLD!'`

# Derived data types

- Data types defined by the user
  - directly derived data type
  - enumerated data type / named values
  - subrange
  - array
  - structured data type

# Explicit type definition

- Type definition delimited by `TYPE..END_TYPE` keywords

```
TYPE
      <TYPE_ID>: <TYPEDEF> := <INIT_VAL>;
END_TYPE
```

  - `<TYPE_ID>` : identifier of the new data type
  - `<TYPEDEF>` : type definition
  - `<INIT_VAL>`: initial value of the new data type

- Variables of the defined type can be declared using the type identifier:

```
VAR
      <VAR_ID> : <TYPE_ID>;
END_VAR
```

# Implicit type definition

- Type of a single variable can be defined implicitly during its declaration:

```
VAR
      <VAR_ID>: <TYPEDEF> := <INIT_VAL>;
END_VAR
```

Implicit type definition applies to the given variable only, can not be reused

# Directly derived data type

- Redefinition of an elementary data type
  - different type identifier
  - optional assignment of a different initial value (must be compatible with the base data type)

```
TYPE
   QINT      : LINT;
   TrueBool : BOOL := TRUE;
END_TYPE
VAR
   xMyTB: TrueBool;
END_VAR
```

The variable `MyTB` might be used anywhere a `BOOL` typed variable can be, but its initial value is `TRUE`.

# Enumerated data type

- Implicit assignment of numerical values to text labels
- Default initial value is the first label
- Labels are replaced by corresponding numerical values during compilation
- The standard does not allow arithmetic operations on enumerated data types (some environments do support arithmetic operations)
- *Its base data type is generally integer (implementer specific)*
- *Numerical values are generally of a continuous range with 0 assigned to the first label in the list (feature not defined in the standard)*

# Example – Enumerated data type

```
TYPE
     TRAFFIC_LIGHT: (RED, AMBER, GREEN);
END_TYPE
VAR
     eMyLight : TRAFFIC_LIGHT;
     eMyMachine: (Idle,Running,NA):=NA;
END_VAR
…
eMyLight := RED;
IF (eMyMachine=Idle)…
…
```

Explicit type definition, initial value is `RED`.

Implicit type definition, initial value is `NA`.

Labels can be used in assignments and conditional expressions.

# Data type with named values

- Explicit assignment of numerical values to text labels
- Data type can be specified
- Any arbitrary value can be labeled, values of the data type might be constant expressions of previously defined labels
- Might store values not associated to labels
- Might be used in arithmetic operations

# Example – Data type with named values

```
TYPE
   HEXCOLORS: DWORD(
                  RED:=16#FF0000,
                  GREEN:=16#00FF00,
                  BLUE:=16#0000FF,
                  WHITE:=RED OR GREEN OR BLUE
                ) := BLUE;
END_TYPE
VAR
   eLineColor: HEXCOLORS:=GREEN;
END_VAR
…
eLineColor:=16#8F0231;
…
```

The label `RED` is associated to the RGB color code `0xFF0000`

32 bit representation, the type can be used anywhere where `DWORD` can be

Constant expressions based on previously defined labels can also be used (`WHITE:=16#FFFFFF`)

A variable might be assigned values not listed in its type definition

# Typed labels

- The same label might be used by different types (enumerated and data type with named values)
- In that case only typed label literals can be used in assignments or expressions: `TYPE#VALUE`
- This feature is not supported by all development environments

- **Best practice: use a label in one single type definition only**

# Example – Typed labels

```
TYPE
    TRAFFIC_LIGHT: (RED, AMBER, GREEN);
    PAINT: WORD(
                RED:=16#F000,
                GREEN:=16#00F0,
                BLUE:=16#000F);
END_TYPE
VAR
    eMyPaint: PAINT;
    eMyLight: TRAFFIC_LIGHT;
END_VAR
…
eMyPaint:=RED;
eMyPaint:=PAINT#RED;
eMyLight:=AMBER;
```

Labels `RED` and `GREEN` are used in two type definitions, their values are ambiguouses

Incorrect: `RED` is ambiguous

Correct: typed label defines that label of the `PAINT` type is used hence value is unambiguous

Correct: label `AMBER` is used in one single type only

# Subrange data type

- Range of elements is limited
  `(Lower_Limit..Higher_Limit)`

- Can be defined based on any numeric type

- Assignment of a value outside the range is an error *(in practice the behavior is implementer-specific -  generally truncation to the limit but might raise a tun-time exception)*

- Initial value: lower limit (not necessarily 0)

# Example – Subrange data type

```
TYPE
    ADC_12_BIT:  UINT(0..4095);
    TEMPERATURE: SINT(-40..70):=0;
END_TYPE
VAR
    sbAdValue: ADC_12_BIT;
END_VAR
…
sbAdValue:=4981;
```

Unsigned integer type with limited range

Initial value is seto to `-40` instead of `0`

Prefix of subrange-typed variables: `sb`

Error: value is outside the range defined

# Array data type

- Collection of elements of the same data type:
  - elementary data type
  - derived data type
  - function block instance
- Multi-dimension arrays are supported
- Subscript range(s) is defined by literals or constant expressions of type `ANY_INT` (limits might be negatives also)
- Initial value of each element can be specified
- Elements are referenced by subscript(s) in brackets: `[i]`

# Example – Array data type

```
TYPE
    MATRIX: ARRAY [1..16,1..16] OF REAL;
END_TYPE
VAR
    aMat  : MATRIX;
    aVec  : ARRAY [-4..9] OF INT:=[1,2,3,11(0)];
END_VAR

aVec[2]:=4;
aMat[1,4]:=4.231;
```

Two-dimensional array of floating point numbers

Prefix `a` is used for array variables

Implicit type definition is applicable for arrays

Use of arbitrary subscript range

Initial values of elements:
`b[-4]=1`
`b[-3]=2`
`b[-2]=3`
`b[-1]=0`
…
`b[10]=0`

Referencing elements by subscript(s) in brackets

# Structured data type

- Collection of sub-elements of the specified types which can be accessed by their specified names
- Sub-elements might be of elementary or derived data types
- Each element can be assigned an individual initial value
- Elements can be accessed as `<VariableID>.<FieldID>`

# Example – Structured data type

```
TYPE
   MEASUREMENT: STRUCT
      dtTimeStamp: DATE_AND_TIME;
      iData: INT:=-1;
   END_STRUCT
   MEASUREMENT_LOG: STRUCT
      strInstrumentID: STRING;
      aLog: ARRAY[1..100] OF Measurement;
   END_STRUCT
END_TYPE
VAR
   stMyLog: MEASUREMENT_LOG:=(strInstrumentID:='ACME12');
END_VAR
…
stMyLog.aLog[12].iData:=981;
```

Structure for recording date and result of a measurement

Initial values might be assigned to elements of the structure

Elements might be of any type, even arrays of structures

An instance-specific initial value for each element might be assigned during declaration of a structured data type variable

Referencing elements of a structure

# Generic data types

**ANY**

**ANY_ELEMENTARY**

**ANY_ DERIVED**

**ANY_MAGNITUDE**

**ANY_NUM**

**ANY_REAL**

REAL, LREAL

**ANY INT**

SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT

**ANY_ DURATION**

TIME, LTIME

**ANY_BIT**

BOOL, BYTE, WORD, DWORD, LWORD

**ANY_ CHARS**

**ANY_ CHAR**

CHAR, WCHAR

**ANY_ STRING**

STRING, WSTRING

**ANY_DATE**

DATE, DT, TOD

Generic data types are used in declarations of standard functions and function blocks only.

# References

- A reference is a pointer to a previously declared variable or function block instance (its value might be `NULL`)
- Declaration is strictly connected to a data type by the keyword `REF_TO <DataType>`
  - elementary data type
  - derived data type
  - function block type
- It is required to define an initial value in the declaration
- Possible values to assign:
  - address of an existing variable or function block instance: `REF(<Variable>)`
  - `NULL`
- Dereferencing: `^` operator

# Example - Reference

```
TYPE
   PERSON: STRUCT
      strName: STRING;
      uiAge: UINT;
   END_STRUCT
END_TYPE
VAR
   iNum: INT;
   stHomer: PERSON;
   refUInt: REF_TO UINT := NULL;
   refPerson: REF_TO PERSON := REF(stHomer);
END_VAR

refUInt := REF(stHomer.uiAge);
refPerson^.Name := 'Homer Simpson'
```

Reference pointing to a `UINT` typed variable with `NULL` initial value

Reference pointing to a derived data type with the reference of a variable as initial value

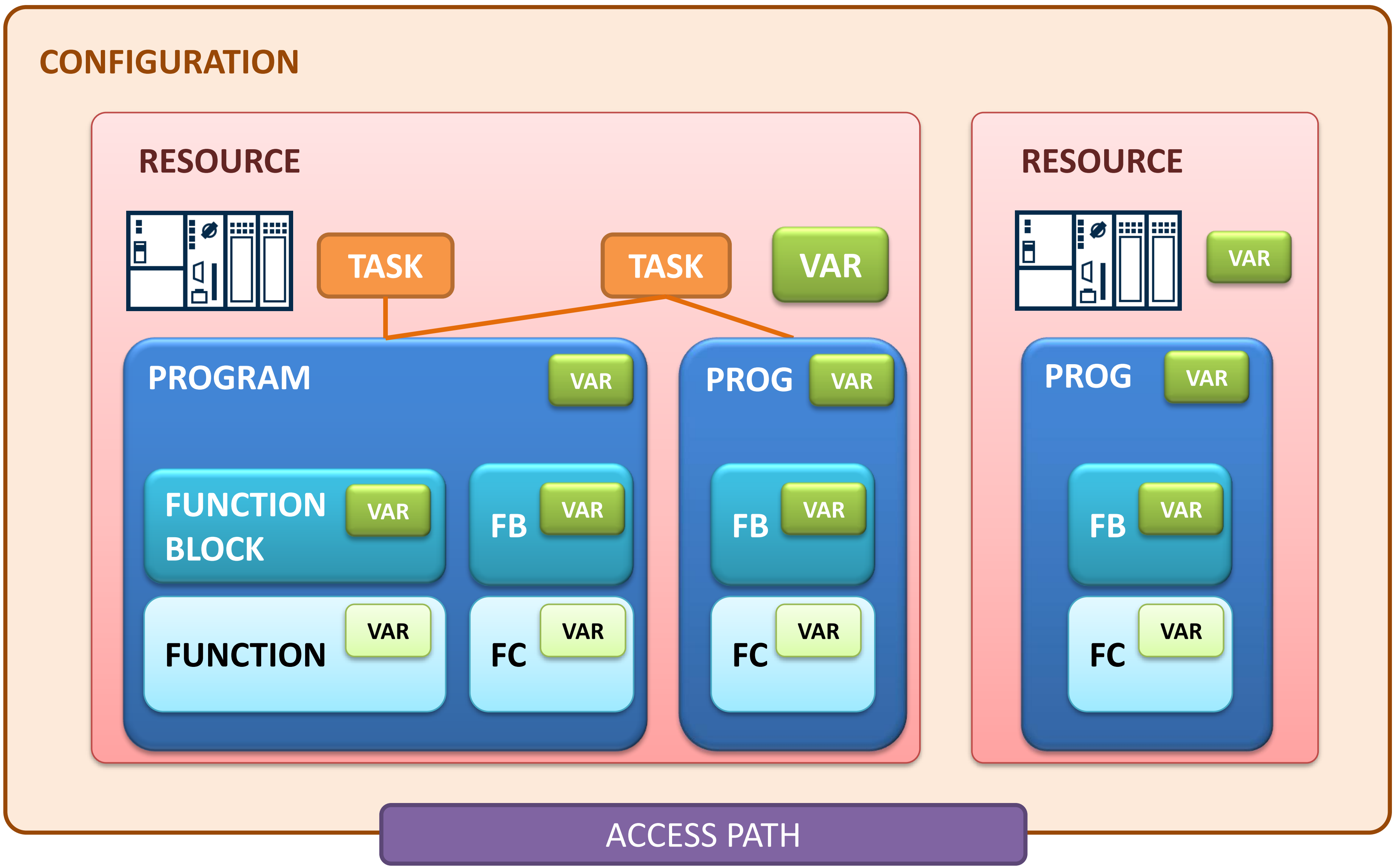
`refUInt` points to the field `uiAge` of the structure `stHomer`

Dereferencing

# Use of references

- References are not available in most of development environments, many times their use does not comply the standard (e.g. CODESYS)
- High-abstraction features of the standard make direct use of references in most cases
- If the use of references is required, use them with extra care!
- Don’t forget: a reference is not a pointer
- Never use pointer arithmetics

# Overview


CONFIGURATION
RESOURCE
TASK
TASK
VAR
PROGRAM
VAR
FUNCTION BLOCK
VAR
FB
VAR
FUNCTION
VAR
FC
VAR
PROG
VAR
FB
VAR
FC
VAR
RESOURCE
VAR
PROG
VAR
FB
VAR
FC
VAR
ACCESS PATH

# Program organization units

**POU type and identifier**

**Declaration part: variables**

**POU body: code**

Variables used by the POU

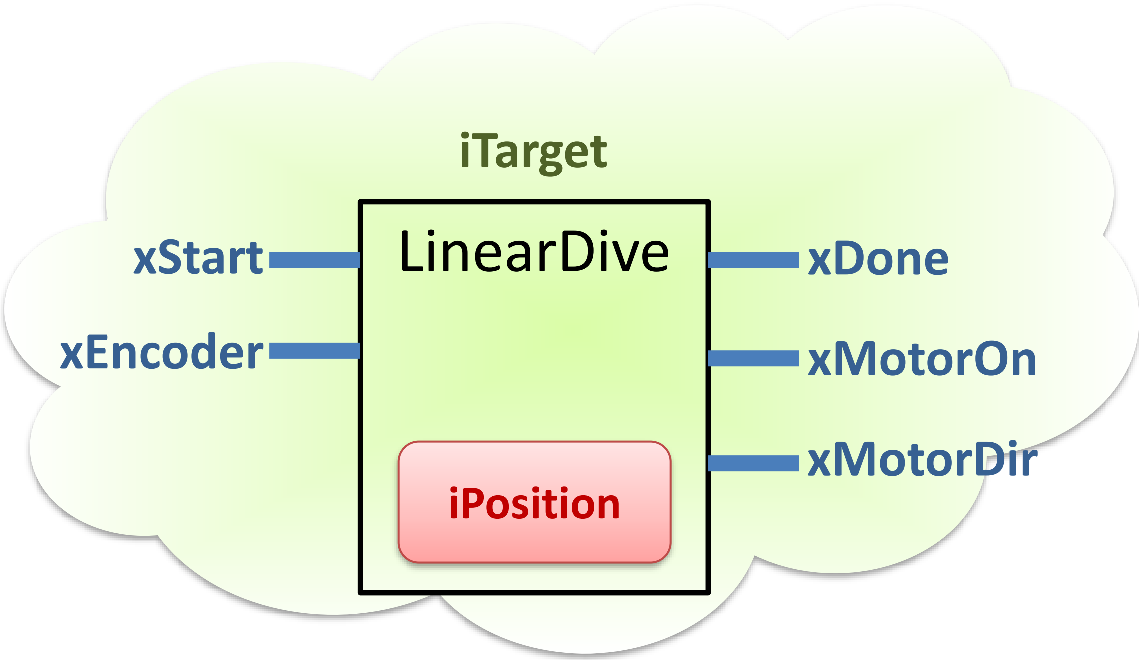
- Local variables
- Interface variables
- Global variables

- Ladder Diagram (LD)
- Function Block Diagram (FBD)
- Structured Text (ST)
- Sequential Function Chart (SFC)
- *Instruction List (IL)*

# Variables

- Variables can be declared in blocks according to their types delimited by the keywords `VARxxx` and `END_VAR`

- Types of variables
  - Local
  - Interface
  - Global



# Local (internal) variables

- `VAR` – local variable
  - internal to the POU: can be accessed only inside the POU
  - persistent (static): keeps its value between calls of the POU (except for functions)

- `VAR_TEMP` – temporary variable
  - internal to the POU: can be accessed only inside the POU
  - not persistent: variable is re-initialized to its initial value at each call (equivalent to `VAR` for functions)

# Interface variables

- `VAR_INPUT` – input variable
  - supplied by an external entity
  - can not be modified within the POU
- `VAR_OUTPUT` – output variable
  - supplied by the POU to an external entity
  - can be modified within the POU
- `VAR_IN_OUT` – input-output variable
  - supplied by an external entity
  - can be modified within the POU and supplied to external entities

# Parameter passing

- `VAR_IN` and `VAR_OUT`: **pass by value**
  - the value of the parameter is passed (e.g. `4`, „`text`”)
  - only the POU supplying the value can modify it

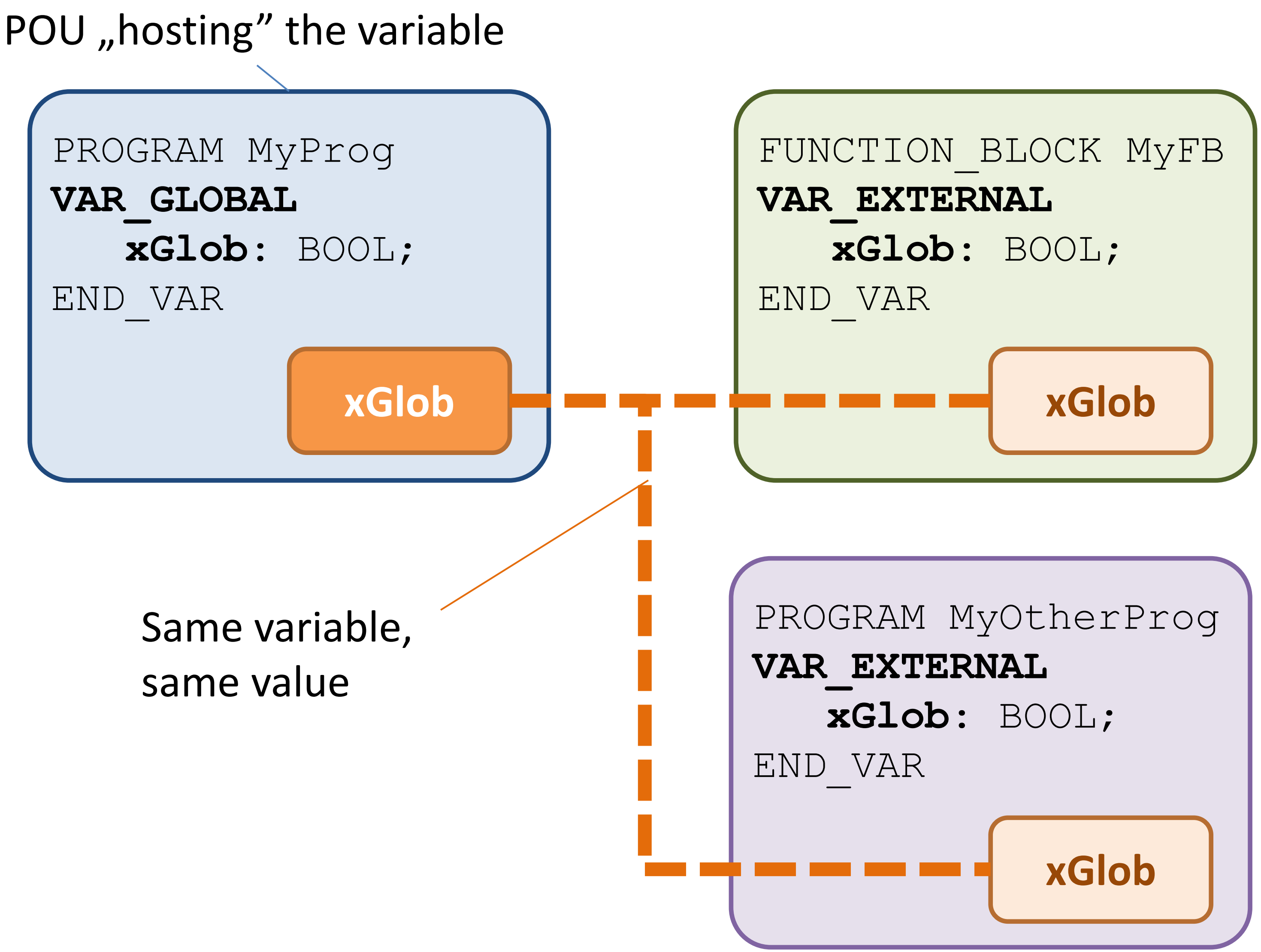
- `VAR_IN_OUT`: **pass by reference**
  - a reference of the memory address where the value of the parameter is stored (pointer) is passed
  - both the supplier and supplied POU can modify the value of the parameter

# Global variables

- Global variables can be accessed from multiple POUs
- Global variables are referenced by their identifiers

- Declaration as
  - `VAR_GLOBAL`: inside one single POU (program), resource or configuration, which „hosts" the variable
  - `VAR_EXTERNAL`: inside other POUs, in which the global variable is used (program, function block, function)

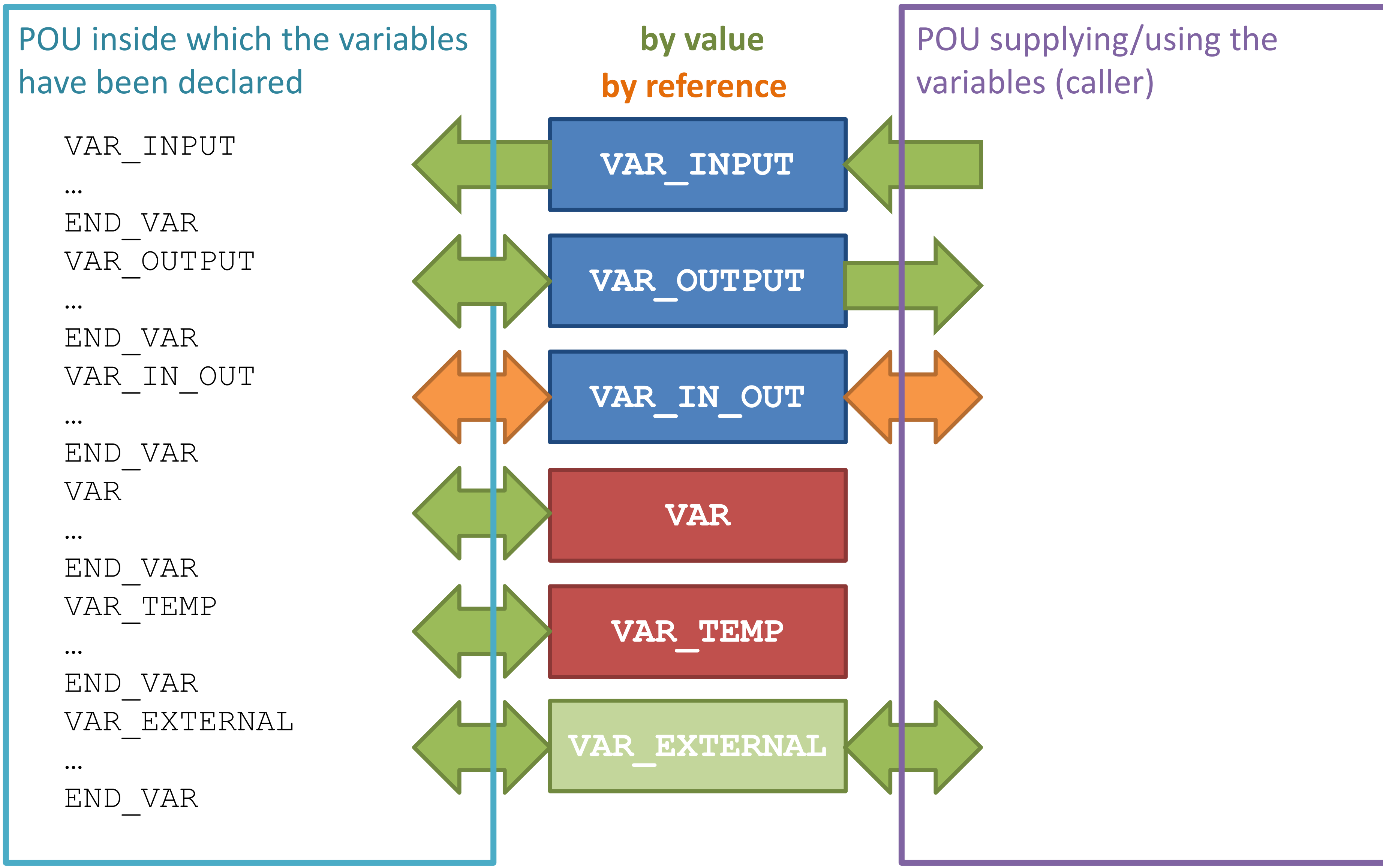
# Global variables



# Configuration variables

- `VAR_CONFIG`
  - can be declared in configurations
  - allows instance-specific assignment of physical addresses and initial values to variables of POUs
  - currently not supported by most development environment
- `VAR_ACCESS`
  - access path declared in a configuration
  - allows named access of variables of POUs from outside the configuration
  - not used in practice

# Access of variables



# Variable declaration

```
VAR
   <ID> [AT <LOCATION>]: <TYPE> [:=INITIAL_VALUE];
END_VAR
```

- `<ID>` : identifier of variable or variables (comma separated list)
- `<TYPE>` : data type
- `<INITIAL_VALUE>` : initial value (optional)
- `<LOCATION>` : physical location (optional)

# Assignment of initial value in declaration

- Instance-specific initial value of a variable might be a literal or a constant expression compatible with its data type
- Initial values can be assigned to a variable of any type, including array and structure data types
- If no instance-specific initial value is given, initial value of the variable equals the initial value of its data type

```
VAR
      iMyIntNum    : INT := 13;
      xBePositive  : BOOL := TRUE;
      xBeNegative  : BOOL;
END_VAR
```

As no initial value is specified, initial value of the variable `xBeNegative` is the initial value of its data type, i.e. `FALSE`

# Initial values of structured type variables

- Sub-elements of a structured data type variable can be initialized individually
- If no initial value is specified, the sub-element is initialized to the value given in the type definition
- If no initial value is given in the type definition, the sub-element is initialized to the initial value of its data type

```
TYPE
 PERSON: STRUCT
  strName:  STRING;
  uiAge:    UINT:=21;
  xIsVip:   BOOL;
 END_STRUCT
END_TYPE
```

```
VAR
 stJane: PERSON:=
         (strName:='Jane Doe');
END_VAR
```

Initial values of elements of `stJane`:

```
stJane.strName  = 'Jane Doe'
stJane.uiAge    = 21
stJane.xIsVip   = FALSE
```

# Qualifiers

- Qualifiers apply for each variable declared in a block (e.g. `VAR`, `VAR_INPUT`)
- **`RETAIN`**: retentive variables, their value is kept even in case of power outage (variables stored in battery-powered RAM)
- **`NON-RETAIN`**: non-retentive variables
- **`CONSTANT`**: constant, its value can not be modified

A variable declared without `RETAIN` / `NOT-RETAIN` qualifier might be retentive or non-retentive depending on the given implementation!

# Example – Qualifiers

```
VAR_OUTPUT RETAIN
    xRetOut1 : BOOL;
    xRetOut2 : BOOL;
END_VAR
VAR CONSTANT
    CONST_NUM : INT := 16#C4;
END_VAR
VAR_EXTERNAL CONSTANT
    xGlob : BOOL;
END_VAR
```

The **RETAIN** qualifier applies for both variables inside the block

Value of `CONST_NUM` can not be modified

Value of the global variable `xGlob` can not be modified <u>inside this given POU</u> (it might be modified by other POUs where `xGlob` is declared without **CONSTANT** qualifier)

# Initialization of variables

| Condition | Warm start | | Cold start |
|---|---|---|---|
| | Retentive | Non-retentive | |
| Declaration of the variable includes an initial value | Value stored in battery-powered RAM A B | Instance-specific initial value C | Instance-specific initial value A C |
| Derived data type, type definition includes the assignment of an initial value | | Initial value in type definition D | Initial value in type definition D |
| Elementary data type or derived data type without assignment of initial value in type definition | | Initial value of the base data type E F | Initial value of the base data type B E F |

```
TYPE
   D_TYPE: INT:=9;
   E_TYPE: INT;
END_TYPE

VAR RETAIN
   A: INT:=4;
   B: INT;
END_VAR
```

```
VAR NON_RETAIN
   C: INT:=6;
   D: D_TYPE;
   E: E_TYPE;
   F: INT;
END_VAR
```

If no qualifier `RETAIN` / `NON_RETAIN` is given in the declaration, initial values of variables in case of warm start is implementer specific.

# Directly represented variables

- Variables are assigned memory locations automatically by the compiler by default
- With the `AT` keyword, variables can be assigned to
  - specific memory locaitons
  - physical inputs
  - physical outputs
- These directly represented variables can be referenced by
  - their symbolic identifier
  - if missing, by the direct address of the physical object (strongly discouraged)

# Addressing directly represented variables

Address format: `%LocationPrefixSizePrefixN`

| Location | | Size | |
|---|---|---|---|
| **Prefix** | **Description** | **Prefix** | **Description** |
| `I` | Input | `X` or missing | Bit |
| `Q` | Output | `B` | Byte size (8 bit) |
| `M` | Memory | `W` | Word size (16 bit) |
| | | `D` | Double word size (32 bit) |
| | | `L` | Long word size (64 bit) |

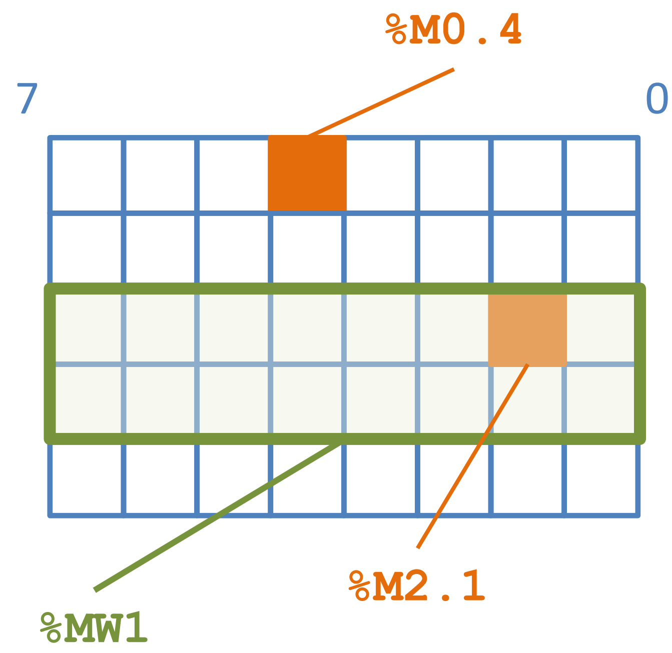
**N**: Unsigned integer or (in case of hierarchic location) list of integers delimited by . (dot) characters, e.g. `%I3.2`

# Examples – Directly represented variables

| Address (variable) | Description | Default data type |
|---|---|---|
| `%QX12` or `%Q12` | Output #12 of a compact PLC with one single digital output module | `BOOL` |
| `%IX0.2` or `%I0.2` | Digital output #2 of the output module #0 (hierarchical address) | `BOOL` |
| `%IB2.0` | First 8-bit channel of the input module #2 | `BYTE` |
| `%MX2.4` or `%M2.4` | Bit #4 of byte #2 in the memory (fixed memory location) | `BOOL` |
| `%MW23` | Memory word #23 (16 bits, fixed memory location) | `WORD` |

# Addressing

- Access of memory objects, inputs are outputs are uniform
- There exist no dedicated bit-, byte- or word-based memory areas
- Locations can be accessed various ways: danger of unintended overwriting of values
- Best practices
  - do not use absolute addressing for the memory
  - take extra care of overlapping areas



# Declaration of directly represented variables

```
VAR
      AT %Q0.1 : BOOL;
      AT %MW12 : SINT:=16;
      xSwitch1 AT %I0.2 : BOOL;
      xMotor3 AT %Q1.3 : BOOL;
END_VAR
…
%Q0.1:=TRUE;
xMotor3:=1;
```

Directly represented variable without symbolic identifier – **strongly discouraged!**

Unique initial values can be specified also for directly represented variables

Directly represented variable, can be referenced by its symbolic identifier only (not by its address).

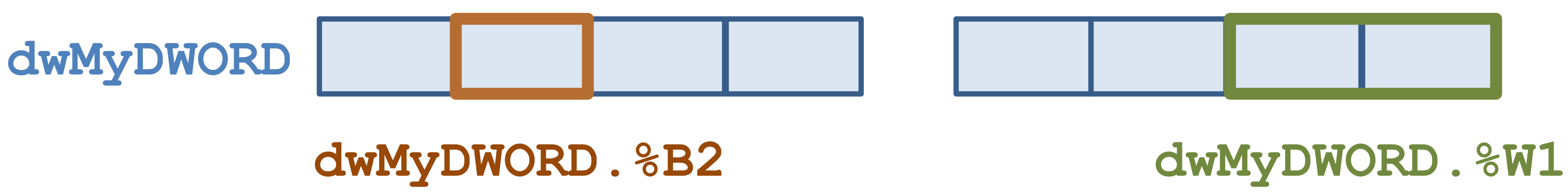
Only addresses declared in the declaration part (without symbolic identifier) can be used

# Use of directly represented variables

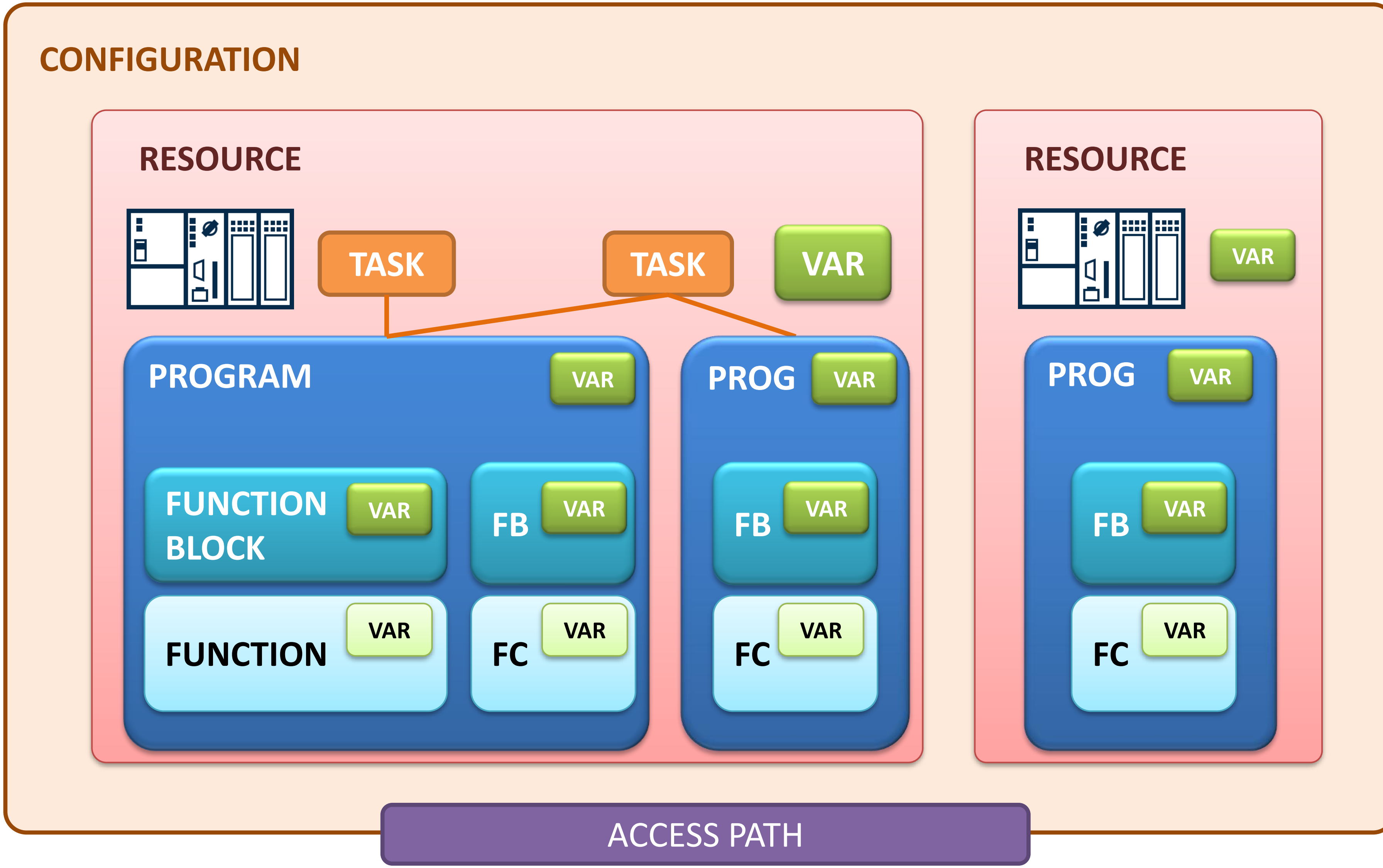
- The standard allows the use of directly represented variables in every program organization unit, although it is not recommended
- **Best practices:**
  - use configuration variables (if supported by the environment)
  - connect physical inputs to input variables of a program
  - connect physical outputs to output variables of a program
  - do not use variables associated to fixed memory locations
  - do not use directly represented variables in functions or function blocks

# Partial access of bit string variables

- Standard allows partial access to bits, bytes, words and double words of variables of type `ANY_BIT` in a way similar to physical addressing: `bMyByte:=dwMyDWord.%B3`
- This feature is not supported by most of the environments
- However, single bits of a variable can be commonly accessed using the . (dot) notation, e.g. `bMyByte.1`



# Overview


CONFIGURATION
RESOURCE
TASK
TASK
VAR
PROGRAM
VAR
FUNCTION BLOCK
VAR
FB
VAR
FUNCTION
VAR
FC
VAR
PROG
VAR
FB
VAR
FC
VAR
RESOURCE
VAR
PROG
VAR
FB
VAR
FC
VAR
ACCESS PATH

# Program Organization Units (POUs)

- Program Organization units allow the development of modular and well-structured applications
- Program organization units implement a well-defined part of the application
- Program organization units can be executed multiple times

# Function

- **A function, when called with the same parameters, always provides the same outputs**
- A function does not store its state, i.e. inputs, internal variables and outputs/result
- A function is callable by all POUs inside a project
- Function execution delivers typically a temporal result (*return value*)
- Result is optional according to the standard, but most environments require functions to return a result
- Result can be set inside the function by assigning value to a variable with the same identifier as the function (declared implicitly)
- Input, output and in-out variables can be used
- Functions can call only other functions

# Function declaration

`FUNCTION` keyword
*beginning of function declaration*

```
FUNCTION MyFC : BOOL
VAR_INPUT
    iIn: INT;
    xIn: BOOL;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
VAR_EXTERNAL
    xG: BOOL;
END_VAR
(* BODY *)
END_FUNCTION
```

Function identifier

Data type of result (return value)

Input variables
*optional, but at least one input variable is used in practice*

Output variables
*optional, allows passing further values to the caller*

Input - output variables
*optional, passed by reference*

Local variables
*optional, values **not stored** between calls*

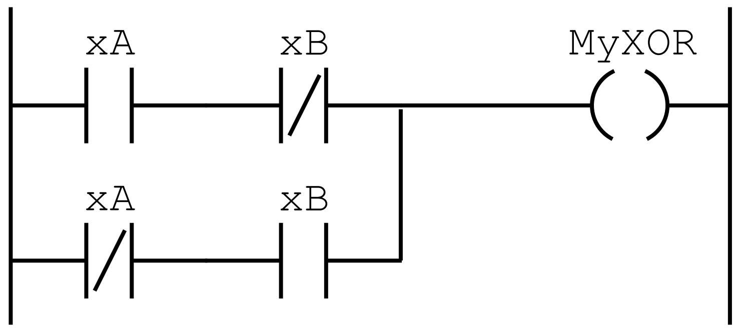
Temporary variables
*optional, as values of local variables are not stored between calls, temporary variables are not used in practice*

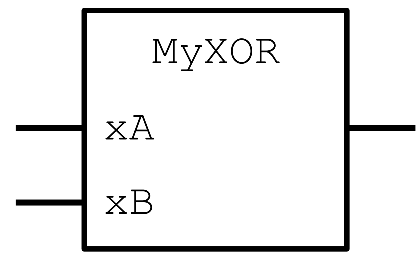
Global variables
*optional, only variables declared in* `VAR_GLOBAL` *blocks of other POUs might be used*

`END_FUNCTION`
*Keywords closing declaration*

# Example - Function

```
FUNCTION MyXOR: BOOL
VAR_INPUT
   xA,xB : BOOL;
END_VAR
```




```
END_FUNCTION
```

# Function call

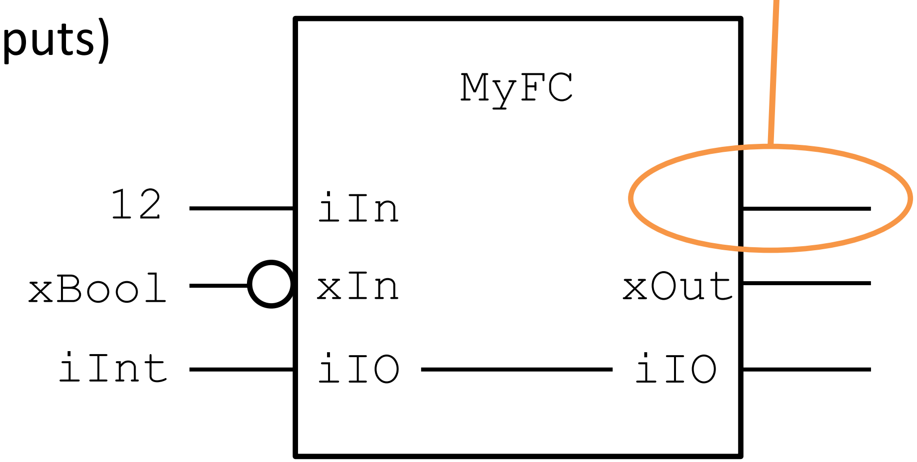
- Assignment of values to input parameters - *optional*
  - literals, variables
  - implicit type conversion might be used
- Assignment of values to in-out parameters - ***obligatory***
  - variables (literal can not be assigned!)
  - type conversion is not applicable
- Storing or using the result (return value) - *optional*
- Storing output variables - *optional*

# Graphical function call

- Function call is inserted to the diagram as a box
- Input and output connections are optional
- Result of the function appears as the topmost connection at the right side
- Inputs (`VAR_INPUT`)
    - left side
    - negation: O symbol (for Boolean inputs)
- Outputs (`VAR_OUTPUT`)
    - right side
    - negation: O symbol (for Boolean outputs)
- In-out parameters (`VAR_IN_OUT`)
    - appear at both sides
    - two ports are connected inside the box

```
FUNCTION MyFC: BOOL
VAR_INPUT
    iIn: INT;
    xIn: BOOL;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
```

Result
(return value)



# Textual function call

- A function call is an expression which evaluates to the result (return value) of the function
- Value of the expression can be assigned to a variable or can be used in an other expression, e.g.

```
xMyBool := AND(xMyBool2,XOR(xMyBool3,xMyBool4))
```

# Formal function call

- The parameter list contains assignments of formal parameter names and values
  - Input parameters: `VarIn := Param`
  - Otput parameters: `VarOut => Param`
  - In-out parameters: `VarInOut := Param`
- Parameters can be omitted, in that case initial value given in the declaration of the corresponding variable of the function is used
- Parameters can be assigned in any arbitrary order

```
xB:=MyFC(iIn:=12,xIn:=xA,iIO:=iA,iOut=>iC)
```

```
xB:=MyFC(iIO:=iA,xIn:=xA)
```

`xA, xB, iA, iB, iC`: variables declared in the caller POU

```
FUNCTION MyFC: BOOL
VAR_INPUT
   iIn: INT;
   xIn: BOOL;
END_VAR
VAR_OUTPUT
   iOut: INT;
END_VAR
VAR_IN_OUT
   iIO: INT;
END_VAR
```

# Informal function call

- Parameter list contains only values assigned to input and in-out variables
- Values need to be given in the same order as the parameters defined in the declaration of the function
- Input and in-out parameters can not be omitted
- Use of `EN`/`ENO` is not permitted
- Values of output parameters can not be accessed

```
xB:=MyFC(iA,TRUE,iB)
```

```
FUNCTION MyFC: BOOL
VAR_INPUT
    iIn: INT;
    xIn: BOOL;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
```

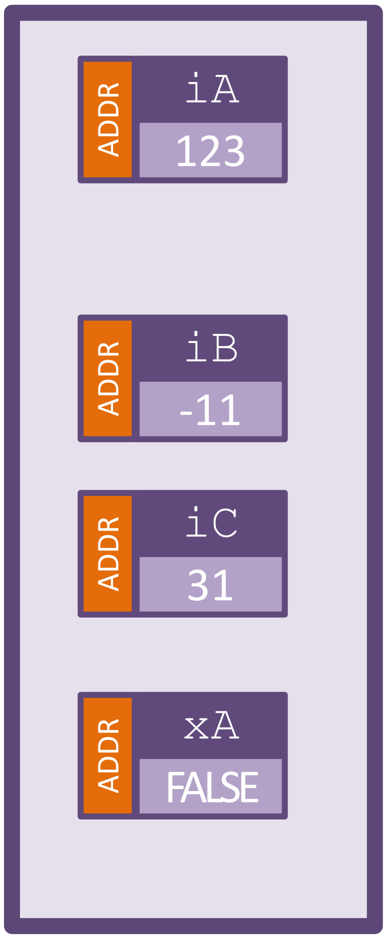
`xA, xB, iA, iB, iC`: variables declared in the caller POU

# Input-output (`IN_OUT`) parameters

- Pass by reference – **only variables with a memory address can be assigned** to them
  - literal, expression or return value of a function can not be assigned
  - assignment can not be omitted during the call
- The called POU can write the assigned variable –**only variables which can be written by the caller can be passed**
  - local variable of the caller POU
  - output variables declared in the caller POU
  - input-output variable declared in the caller POU
  - global variable declared as EXTERNAL in the caller POU
- **Assignment of input-output (`IN_OUT`) parameters can not be omitted during the call**

# Function call

```
xA:=myFC(iIn:=iA,iIO:=iB,iOut=>iC);
```



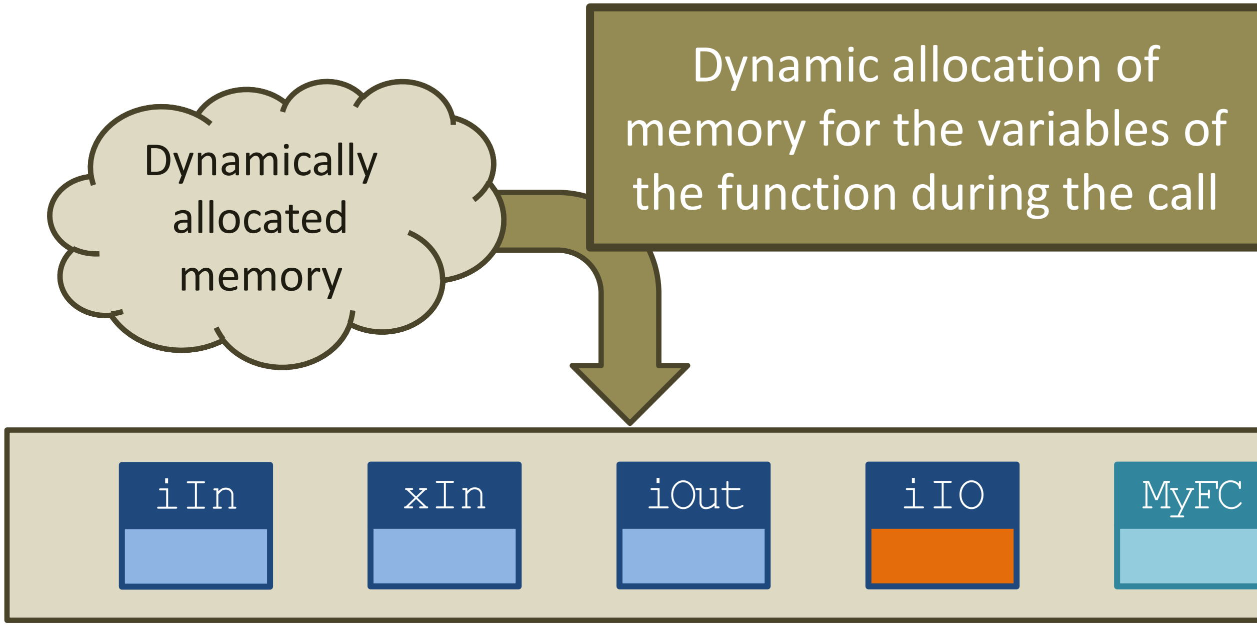

Memory of the caller POU

```
FUNCTION MyFC: BOOL
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT:=123;
    xL: BOOL;
END_VAR
```

Declaration part of the called funtion

# 1. Memory allocation

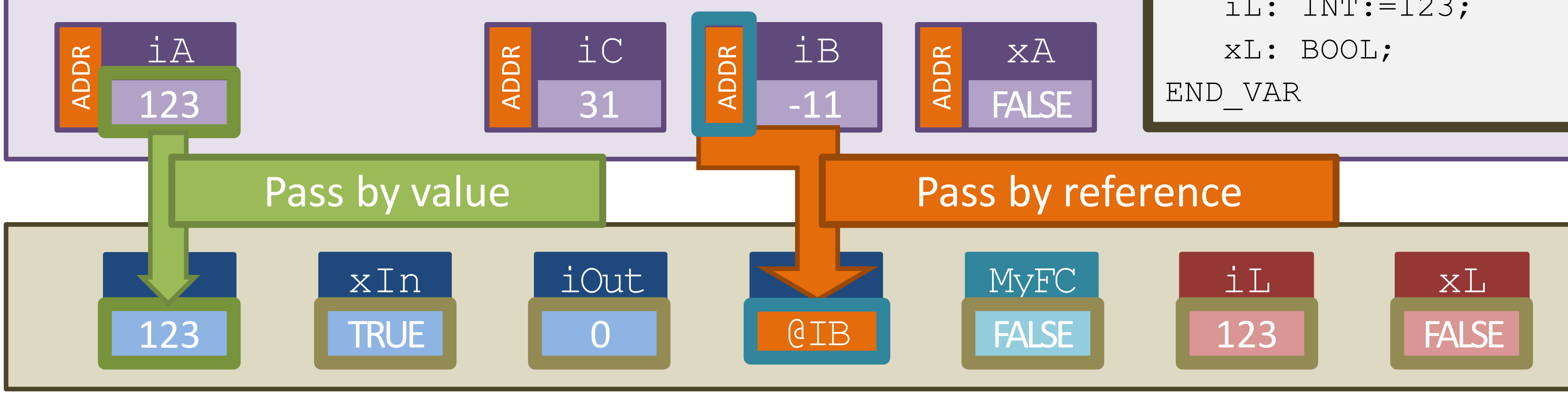
```
FUNCTION MyFC: BOOL
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT:=123;
    xL: BOOL;
END_VAR
```



# 2. Parameter passing

```
xA:=myFC(iIn:=iA,iIO:=iB,
         iOut=>iC);
```

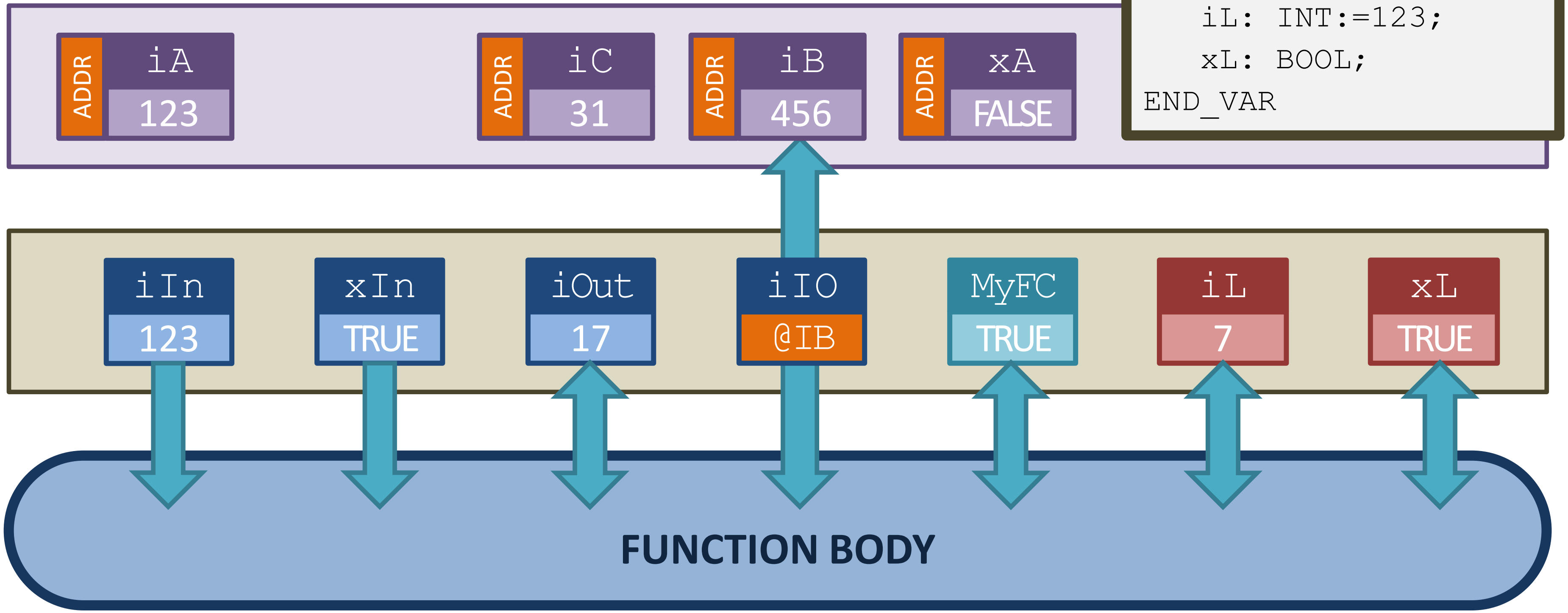
```
FUNCTION MyFC: BOOL
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT:=123;
    xL: BOOL;
END_VAR
```




Local and output variables, input variables without parameters assigned during the call, as well as the result are initialized to their initial values

# 3. Execution of the body

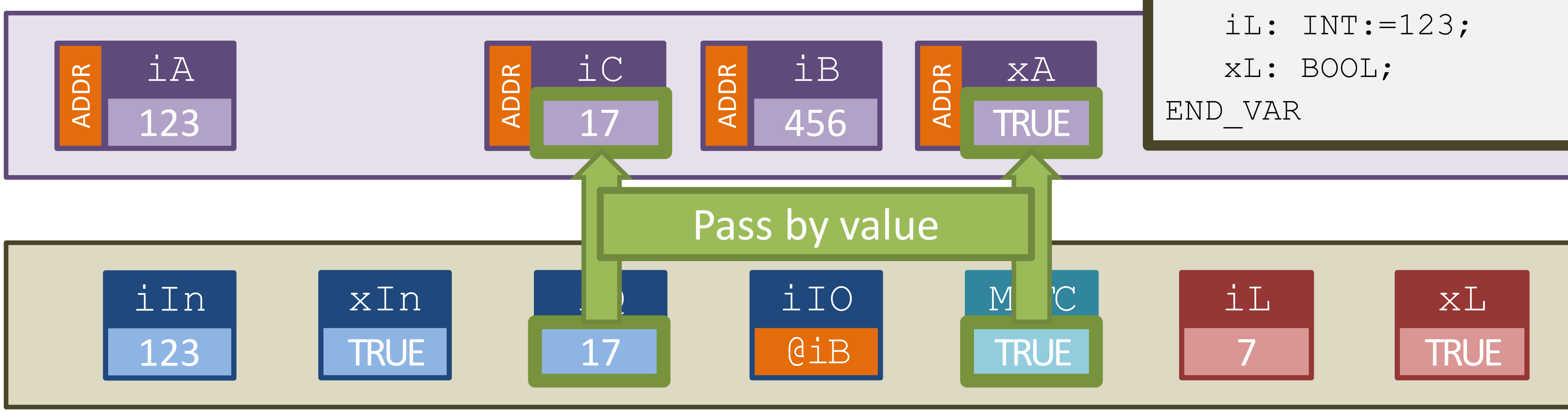
```
FUNCTION MyFC: BOOL
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iInOut: INT;
END_VAR
VAR
    iL: INT:=123;
    xL: BOOL;
END_VAR
```



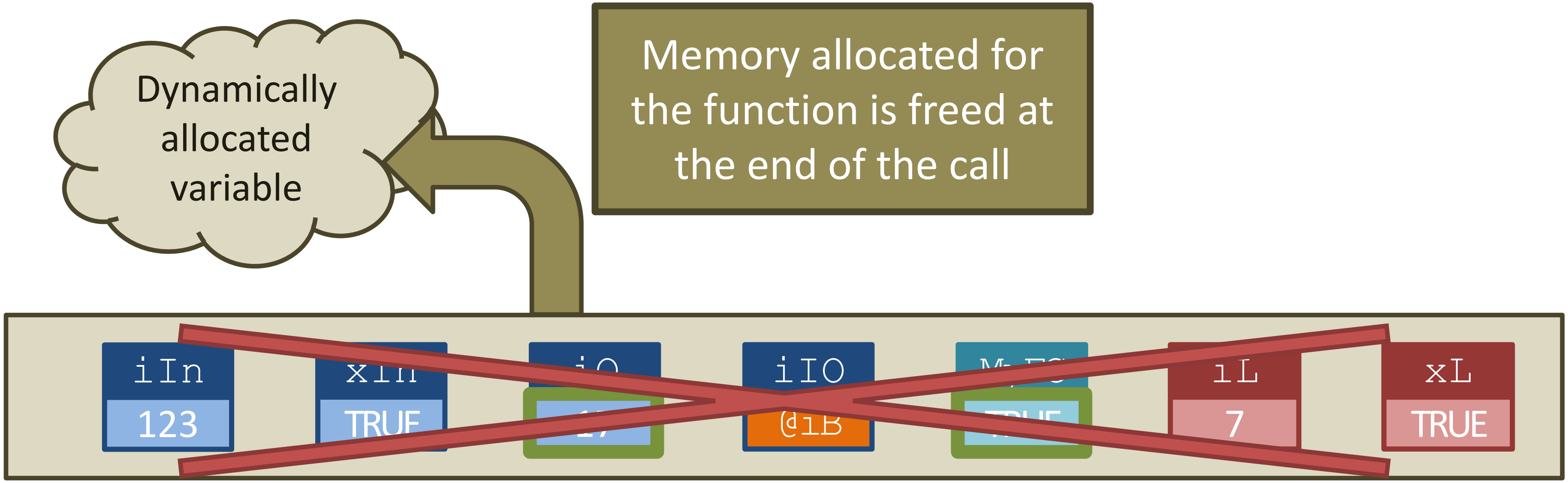
# 4. Result and outputs

```
xA:=myFC(iIn:=iA,iIO:=iB,
         iOut=>iC);
```

```
FUNCTION MyFC: BOOL
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT:=123;
    xL: BOOL;
END_VAR
```



# 5. Memory de-allocation



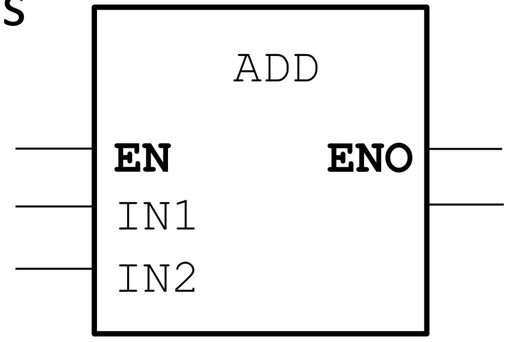
# Access to the variables of a function

As memory in which variables of a function are stored is allocated dynamically, variables can only be accessed during the call

```
FUNCTION MyFC: BOOL
VAR_INPUT
   iIn: INT;
   xIn: BOOL;
END_VAR
VAR_OUTPUT
   iOut: INT;
END_VAR
VAR_IN_OUT
   iIO: INT;
END_VAR
```

```
MyFC.xIn:=TRUE ✗

iA:=MyFC.iOut ✗

MyFC(xIn:=TRUE,iOut=>iA) ✓
```

# Execution control

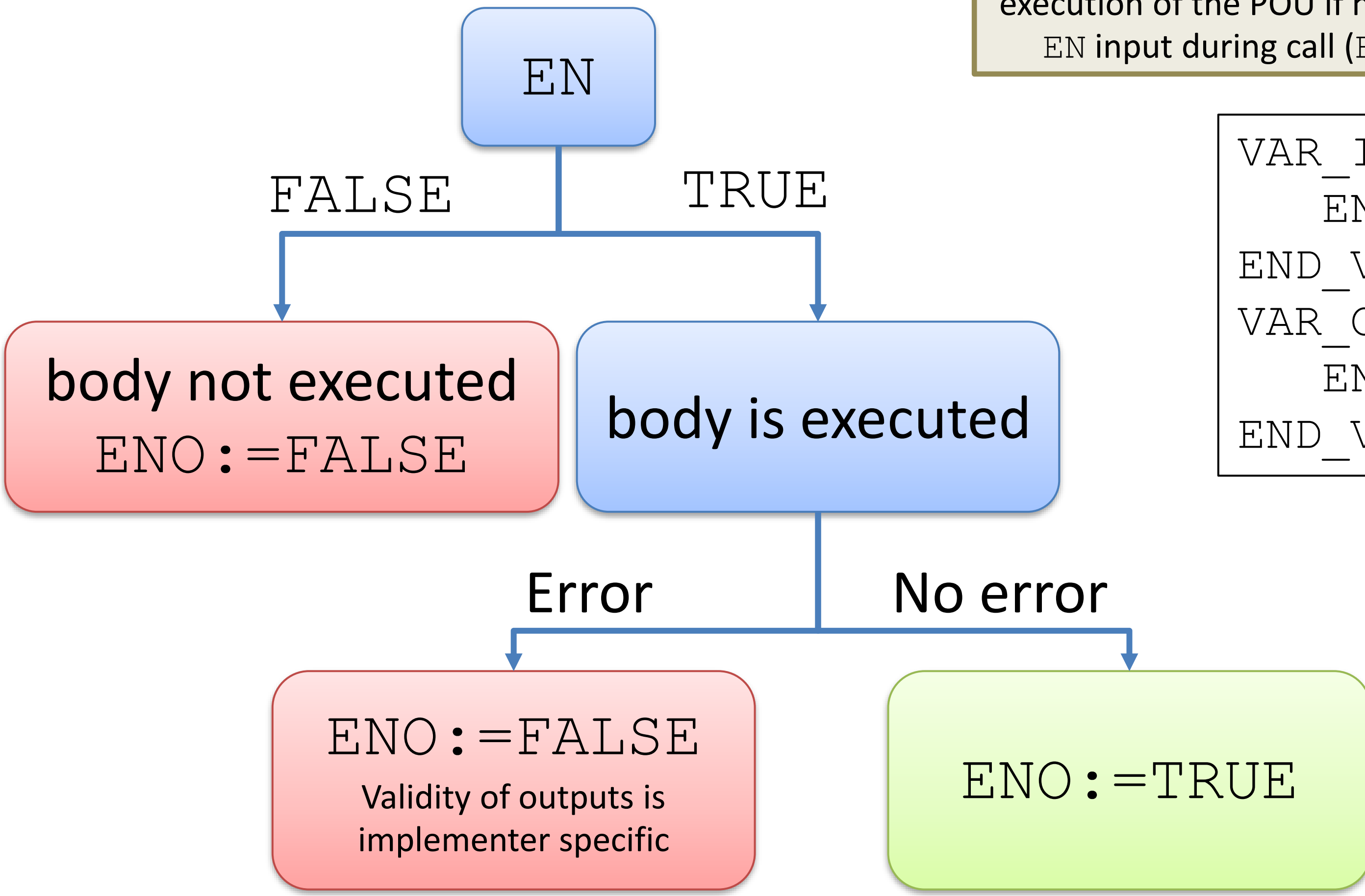



- Execution control Boolean input/output variables
  - `EN`: **En**able input
  - `ENO`: **En**able **O**ut output
- Use of EN/ENO is optional
- Standard functions support `EN/ENO`
- In graphical form execution control inputs and outputs are the topmost ones (`ENO` precedes the result of the function)
- Execution control variables can be accessed during the call only
- Value of `ENO` can be set inside the function only

# Execution control: standard operation

Using `TRUE` as initial value of `EN` allows the execution of the POU if no value is assigned to the `EN` input during call (`EN` is left unconnected)

```
VAR_INPUT
   EN: BOOL := TRUE;
END_VAR
VAR_OUTPUT
   ENO: BOOL;
END_VAR
```




*Note: it is allowed for the body of user-defined functions or function blocks to be executed with* `EN=FALSE`*, optionally executing a different algorithm*

# Overloaded and extensible functions

- **Overloaded** functions
  - might be called with various parameter types
  - inputs have generic data types, e.g. `ANY_INT`, `ANY_MAGNITUDE`
- **Extensible** functions
  - might be called with any arbitrary number of inputs, e.g. `ADD(In1,In2…)`
- Only standard or implementer-supplied functions can be overloaded or extensible

# Standard functions

- Shall be implemented in all IEC 61131-3 compliant development environments
- Commonly used functions for a wide variety of operations
- Might be overloaded and/or extensible

# Numeric type conversion

- Type conversion functions
  - typed: `*_TO_**`, e.g. `REAL_TO_INT`
  - overloaded: `TO_**`, e.g. `TO_INT`
- Steps of the conversion process
  1. extension of the source value to the largest compatible data type (e.g. from `INT` to `LINT`)
  2. conversion to the largest data type compatible with the result data type
  3. conversion to the actual result data type
- In case of out-of-range values, behavior is implementer specific (error/truncation)
- Real→ integer conversion is carried out by rounding to nearest integer or to nearest even integer in case of ambiguity (IEC 60599)

# Example - Numeric type conversion

```
iMyInt:=REAL_TO_INT(REAL#712.5)
```

`REAL#712.5`

1. Extension of the source value to the largest compatible data type (`LREAL`)

`LREAL#712.5`

2. Conversion to the largest data type compatible with the result data type (`LINT`)

As values 712 and 713 are at the same distance, rounding to the next even integer

`LINT#712`

3. Conversion to the actual result data type (`INT`)

`INT#712`

# Numeric type conversion

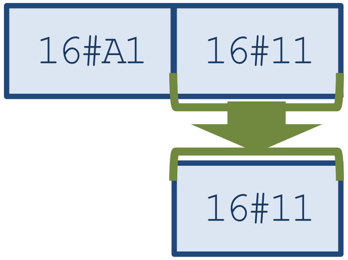
- Truncation
  - deprecated: `TRUNC`
  - typed: `*_TRUNC_**`, pl. `REAL_TRUNC_INT`
  - overloaded: `TRUNC_**`, pl. `TRUNC_INT`
- Truncation is defined from real to integer data types only: `ANY_REAL_TRUNC_ANY_INT`
- Truncation is carried out towards zero
- In case of out-of-range values, behavior is implementer specific (error/truncation)

# Bit string conversion functions

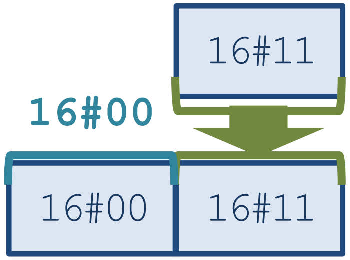
- Data is copied bitwise
- If the destination data type is smaller than the source, only the rightmost bytes are stored
- If the source data type is smaller than the destination, leftmost bytes are filled with zeros
- Typed function for conversion between bit strings and bit strings, characters, integer types
  - `DWORD_TO_BYTE`
  - `BYTE_TO_REAL`
  - etc.

# Example - Bit string conversion functions

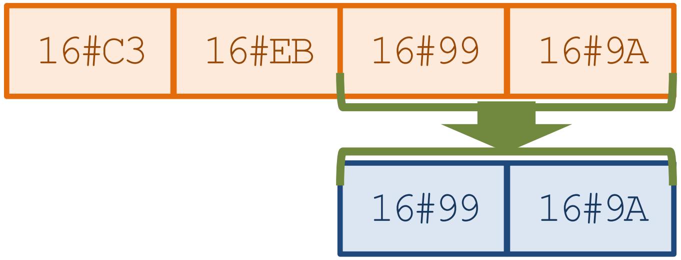
```
byMyByte:=
WORD_TO_BYTE(wMyWord)
```




```
wMyWord:=
BYTE_TO_WORD(byMyByte)
```




```
wMyWord:=
REAL_TO_WORD(-471.2)
```



# Other conversion functions

- Conversion of duration, date and time data types

  (e.g. `TOD_TO_LTOD`)

- Conversion of character and string data types

  (e.g. `CHAR_TO_STRING`)

# Numerical and arithmetic functions

- Numerical functions:
  - `ABS`
  - `SQRT`
  - `LN, LOG, EXP`
  - `SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2`

- Arithmetic functions:
  - `ADD (+), MUL(*)`: extensible
  - `SUB (-), DIV (/), MOD`
  - `EXPT` (power, `**`): $\mathrm{OUT}:=\mathrm{IN1}^{\mathrm{IN2}}$
  - `MOVE` (data move, `:=`): `OUT:=IN`

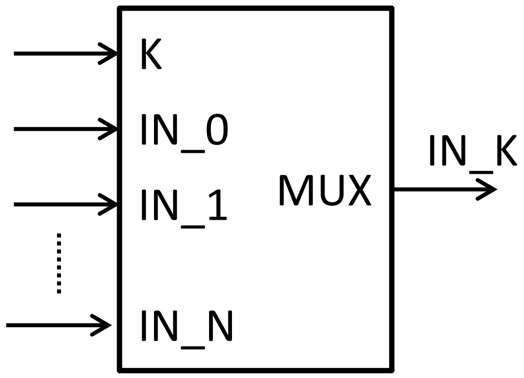
# Bit string and Boolean functions

- Bit shift functions
  - `SHL, SHR` – shift by N bits, fill with zeros
  - `ROR, ROL` – rotation by N bits

- Bitwise Boolean function
  - `AND (&), OR (>=1), XOR (=2k+1)` extensibles
  - `NOT`

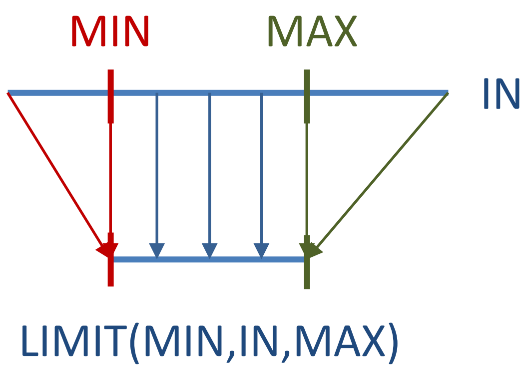
# Selection functions

- `MUX`
  - `A:=MUX(1,B,C,D)` results in `A=C`
  - extensible, overloaded
- `SEL`
  - binary selection between two inputs (ternary operator)




- `MIN, MAX`
  - extensible, overloaded
- `LIMIT`
  - `Limit(Min,In,Max)`
  - `Limit:=`

    `MIN(MAX(In,MinVal),MaxVal)`
  - overloaded



# Comparison function

- `GT (>), GE (>=)`
- `EQ (=), NE (<>)`
- `LE (<=) , LT (<)`

- Comparison functions are extensible:

```
GT(IN1, IN2, IN3) = (IN1>IN2)&(IN2>IN3)
```

# String functions

- `LEN`: length
- `LEFT, RIGHT, MID`: substring selection
- `CONCAT`: concatenation (extensible)
- `INSERT, DELETE, REPLACE, FIND`: insertion, deletion, replacement or search of substring

# User-defined functions

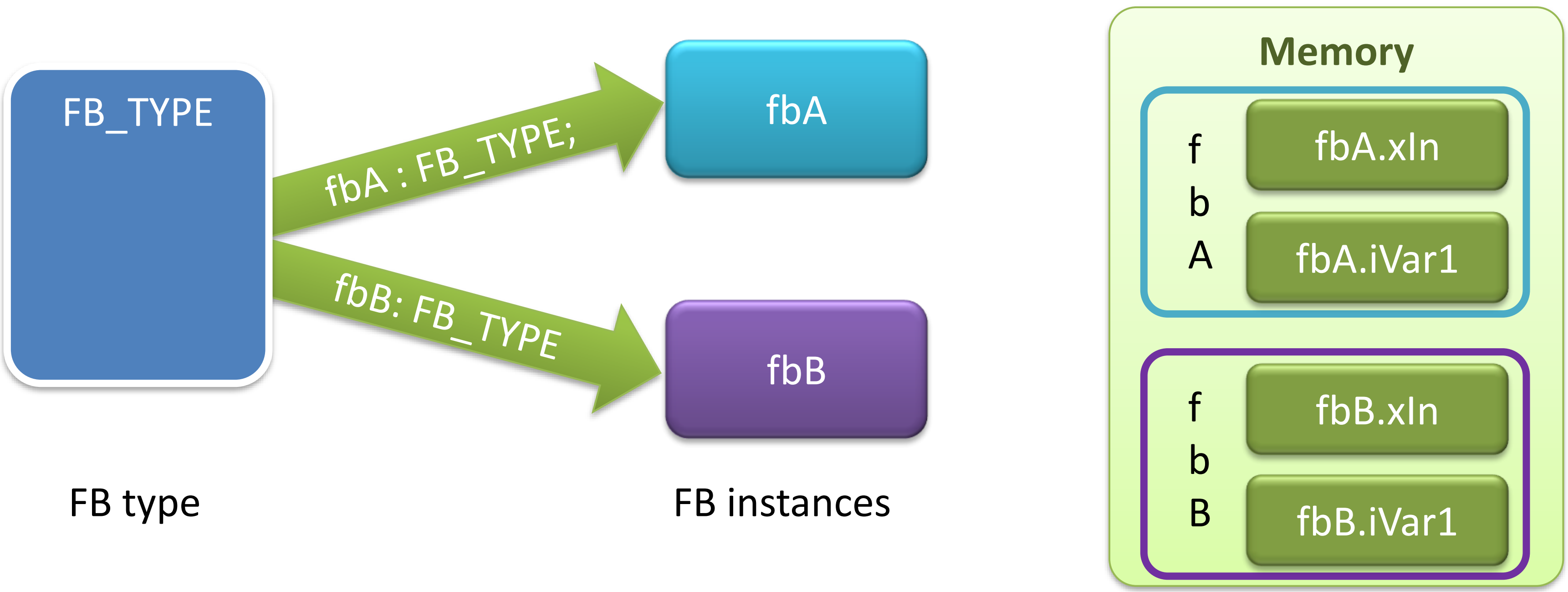
- User might define any arbitrary function
- User-defined functions are available (can be called) in the whole project
- User-defined functions might call any standard or other user-defined function

# Function block

„A function block is an independent entity encapsulating a data structure and an algorithm working on it.”

# Function block instances

- Function block type ≈ Class
- Function block instance ≈ Object
  - function blocks need to be instantiated, only instances might be called
  - variables are associated to instances not types
  - instances do not influence directly the operation of other instances



# Instance: encapsulation

- Instantiation: memory declaration for variables of the instance
  - `VAR_INPUT`
  - `VAR_OUTPUT`
  - `VAR`
- Static memory tightly attached to <u>the instance</u>: values are stored between calls
- `VAR_TEMP`: dynamically allocated memory, values not stored between calls
- `VAR_IN_OUT:` implementation specific, allocation might be static or dinamic, but the parameter needs to be assigned during each call
- Instantiation of a function block is formally the same as declaration of a variable: `MyInstance: MyFBType;`

# Instance: structure

- Variables of the FB instance are represented as a structure
- Interface variables of the FB instance can be accessed from the calling POU:
  - inputs might be written (not just during the call)[1]
  - outputs might be read (not just during the call)
  - input-output variables might be set (during the call only)
  - local variables can not be accessed
- Referencing the variables of the FB instance:

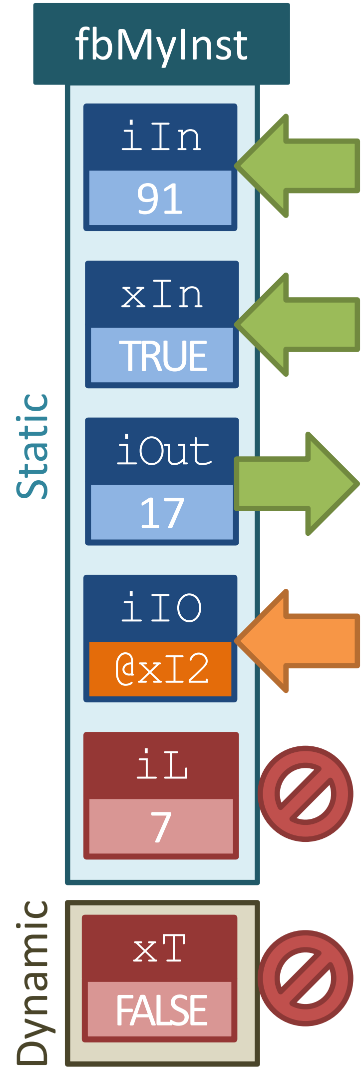
  <FB_name>.<Var_name>

**Setting variables of the instance is not a call, code in the body of the FB is not executed!**

[1] However not allowed by the standard, many development environments allow read of input variables.

# Accessing elements of the structure

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
   iIn: INT;
   xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
   iOut: INT;
END_VAR
VAR_IN_OUT
   iIO: INT;
END_VAR
VAR
   iL: INT;
END_VAR
VAR_TEMP
   xT: BOOL;
END_VAR
```




```
PROGRAM MyPRG
VAR
   fbMyInst: FB_TYPE;
   iA,iB,iC: INT;
   xA: BOOL;
END_VAR

fbMyInst.iIn:=4;
xA:=fbMyInst.xIn;      ?
iA:=fbMyInst.iOout;
fbMyInst.iOut:=iB;
iA:=fbMyInst.iIO;
fbMyInst.iIO:=iA;
fbMyInst(iIO:=iB);
MyInst.iL:=iA;
xA:=MyInst.xT;
```

**Supported by most environments**

**Access during the call only**

# Instance: structure

- Variables of the instance might be given an instance-specific initial value
- If not initialized, values are assigned initial values given in the declaration of the FB instance or the initial value according to their data type

```
VAR
  MyInst: FB_TYPE:=(iIn:=4,
                    iL:=12);
END_VAR
```

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
   iIn: INT;
   xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
   iOut: INT;
END_VAR
VAR_IN_OUT
   iIO: INT;
END_VAR
VAR
   iL: INT;
END_VAR
VAR_TEMP
   xT: BOOL;
END_VAR
```

# Function block type declaration

`FUNCTION_BLOCK` keywords
*denotes start of function block type declaration*

Identifier of the FB type

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
    iIn: INT;
    xIn: BOOL;
END_VAR
VAR_OUTPUT
    xOut: BOOL;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
VAR_EXTERNAL
    xG: BOOL;
END_VAR
 (* BODY *)
END_FUNCTION_BLOCK
```

Input variables
*optional, but at least one input variable is used in practice*

Output variables
*optional, allows passing values to the caller*

Input - output variables
*optional, passed by reference*

Local variables
*optional, values **stored** between calls*

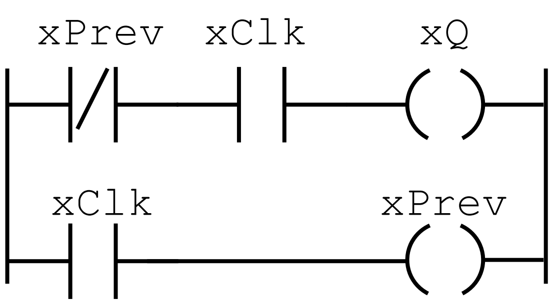
Temporary variables
*optional, values **not stored** between calls*

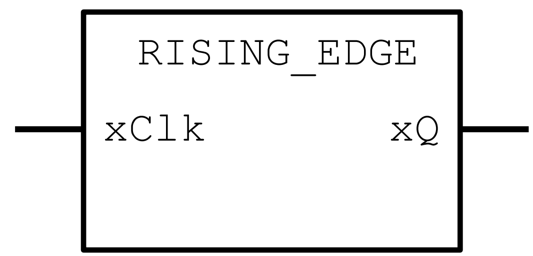
Global variables
*optional, only variables declared in* `VAR_GLOBAL` *blocks of other POUs might be used*

`END_FUNCTION_BLOCK`
*keyword for closing declaration*

# Example: edge-sensing function block

```
FUNCTION_BLOCK RISING_EDGE
VAR_INPUT
   xClk: BOOL;
END_VAR
VAR_OUTPUT
   xQ: BOOL;
END_VAR
VAR
   xPrev: BOOL;
END_VAR
```

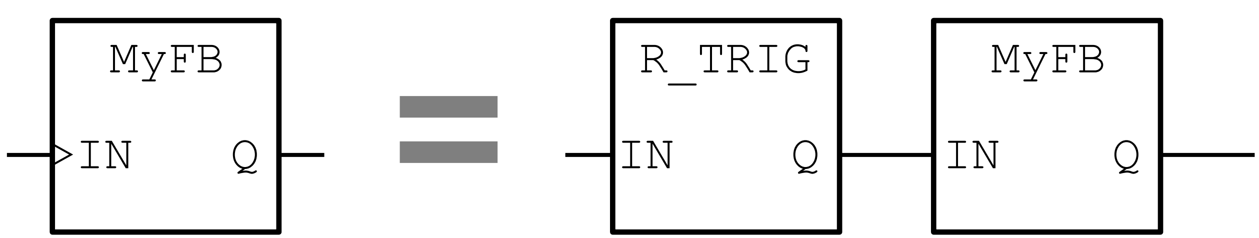



```
END_FUNCTION_BLOCK
```

# Edge-sensing input variables

- Input variables (`VAR_INPUT`) of programs and function blocks might be declared with edge sensing qualifiers
- Might be specified for each variable:
  - `R_EDGE` – positive edge sensing input (> notation)
  - `F_EDGE` – negative edge sensing input (< notation)
- Behavior is the same as connecting an edge-sensing function block to the input
- Most development environments support use of edge-sensing inputs for standard function blocks only



# Edge-sensing input variables

Called FB

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
   xIn1: BOOL R_EDGE;
   xIn2: BOOL F_EDGE;
END_VAR
…
```

```
PROGRAM MyProg
VAR
   fbMyFB:  FB_TYPE;
   xSignal: BOOL;
END_VAR

fbMyFB(xIn1:=xSignal,
       xIn2:=xSignal);
…
```

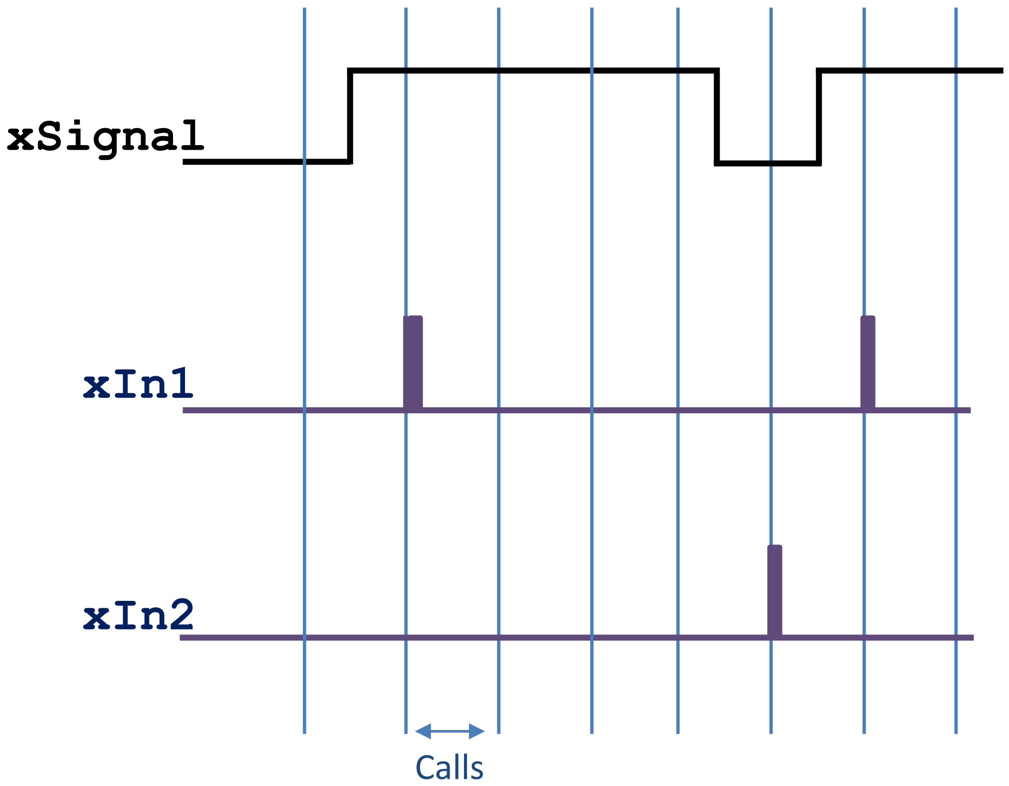



Figure shows values of variables xIn1 and xIn2 accessed from the called POU

# Visibility of local variables

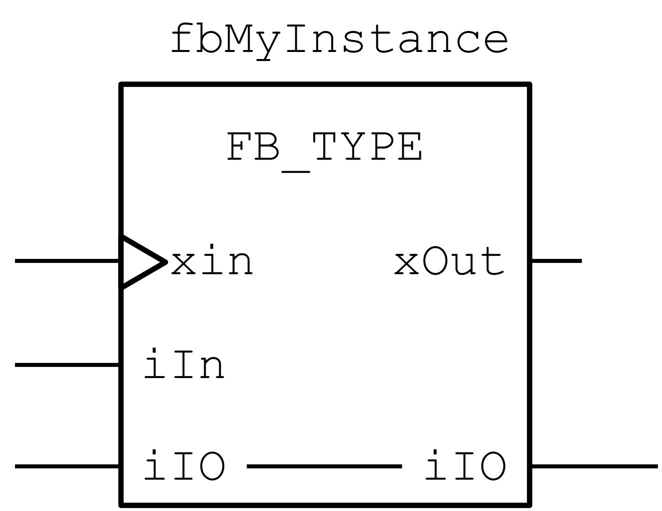
- According to Edition 2013 of the standard, visibility of local variables might be set
  - `PUBLIC` – local variables of the function block instance might be accessed externally
  - `PRIVATE` – local variables of the function block instance might be accessed externally
- Default behavior (without qualifier): local variables can not be accessed externally
- Most development environments do not support the use of visibility qualifiers

Best practice: do not use visibility qualifiers and treat all local variables as **`PRIVATE`** ones.

# Graphical function block call

- Instance identifier above the block, FB type inside the block
- Inputs (`VAR_INPUT`)
  - at the left side
  - negation: O symbol
  - positive edge sensing: > symbol
  - negative edge sensing: < symbol
- Outputs (`VAR_OUTPUT`)
  - at the right side
- Input-output (`VAR_IN_OUT`)
  - shown at both sides
  - labels connected internally
- `EN`/`ENO` might be used

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
   xIn: BOOL R_EDGE;
   iIn: INT;
END_VAR
VAR_IN_OUT
   iIO: INT;
END_VAR
VAR_OUTPUT
   xOut: BOOL;
END_VAR
```



# Textual function block call

- Only formal call is allowed: parameter list contains association of values to formal parameters
  - Input variables: `VarIn := Param`
  - Output variables: `VarOut => Param`
  - Input-output variables: `VarInOut := Param`
- Parameters might be omitted, their order might be any arbitrary

```
VAR
   fbMyInstance: FB_TYPE;
END_VAR

fbMyInstance(xOut=>xA,
             xIn:=TRUE,
             iIO:=iA);
```

`xA`, `iA`: variables declared in the caller POU

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
   xIn: BOOL R_EDGE;
   iIn: INT;
END_VAR
VAR_IN_OUT
   iIO: INT;
END_VAR
VAR_OUTPUT
   xOut: BOOL;
END_VAR
```

# Steps of function block call

## 0. Declaration of the instance

```
PROGRAM MyPRG
VAR
    fbInst: FB_TYPE;
    iA,iB,iC: INT;
    xA: BOOL;
END_VAR
```

Declaration part of the caller POU

Allocation of memory for static variables of the **function block instance** in compile-time

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
```


Variables are initialized for their instance-specific initial values (if specified) or the initial values of their data type.

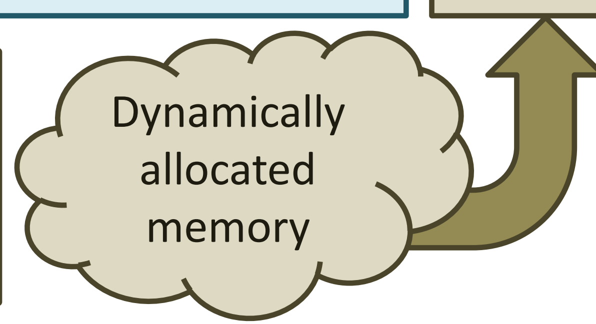
# 1. Memory allocation

```
fbInst(iIn:=iA,iIO:=iB,
        iOut=>iC);
```

The call illustrated here is not the first, hence values of static variables might differ from their initial values.

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
```

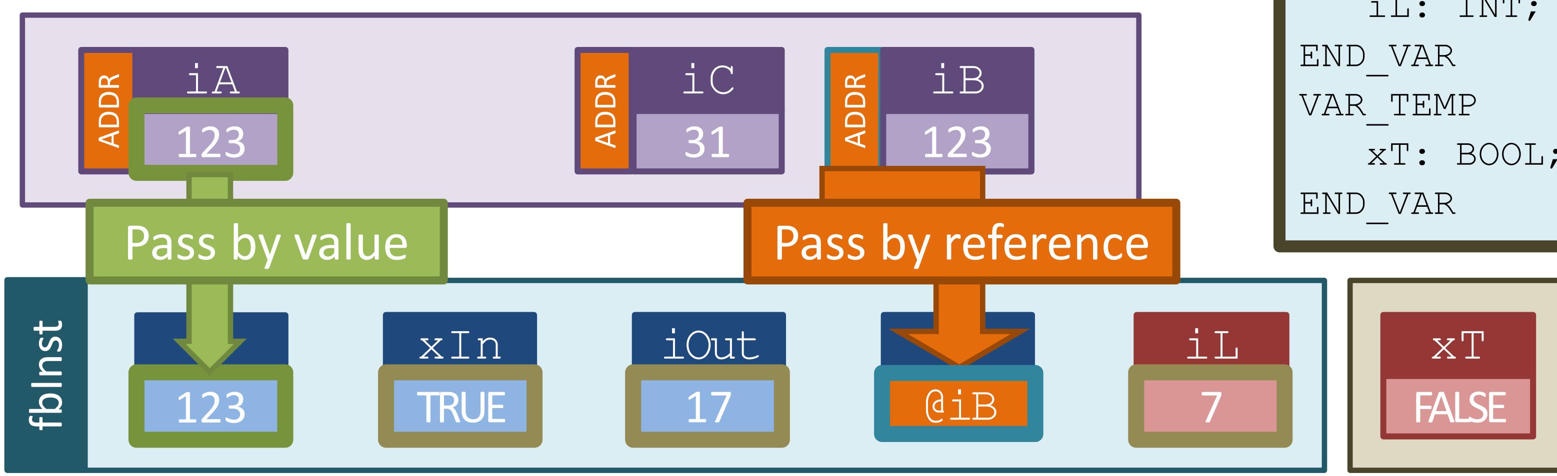

Memory for temporary variables is allocated dynamically during the call, their values are initialized at each call.



# 2. Parameter passing

```
fbInst(iIn:=iA,iIO:=iB,
        iOut=>iC);
```

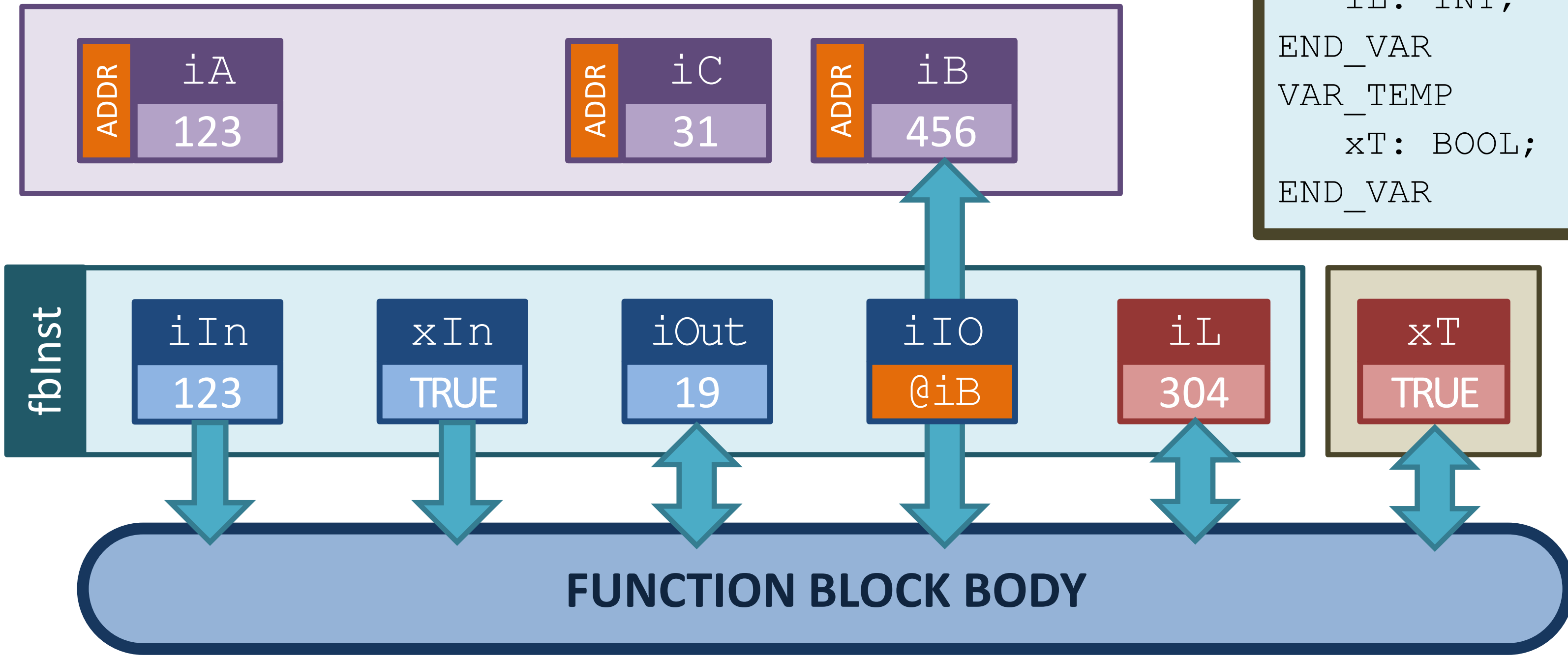
```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
```




Input and input-output variables not assigned during the call, static local variables and output variables retain their value from the last call (they are set to their initial values during the first call).

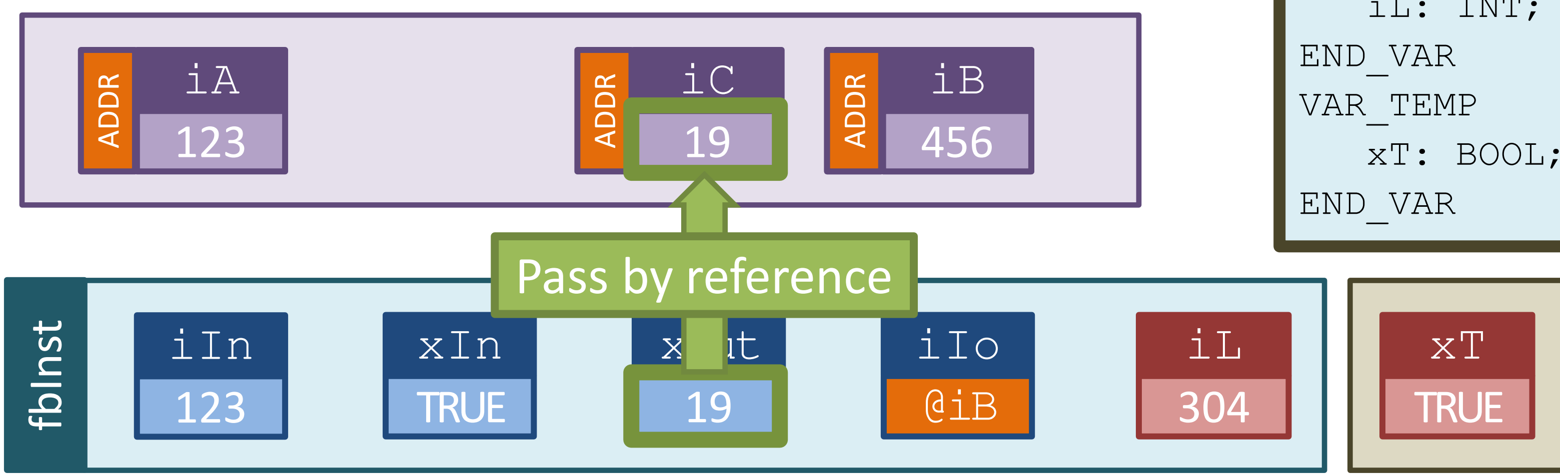
# 3. Execution of the body

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
```



# 4. Passing output values

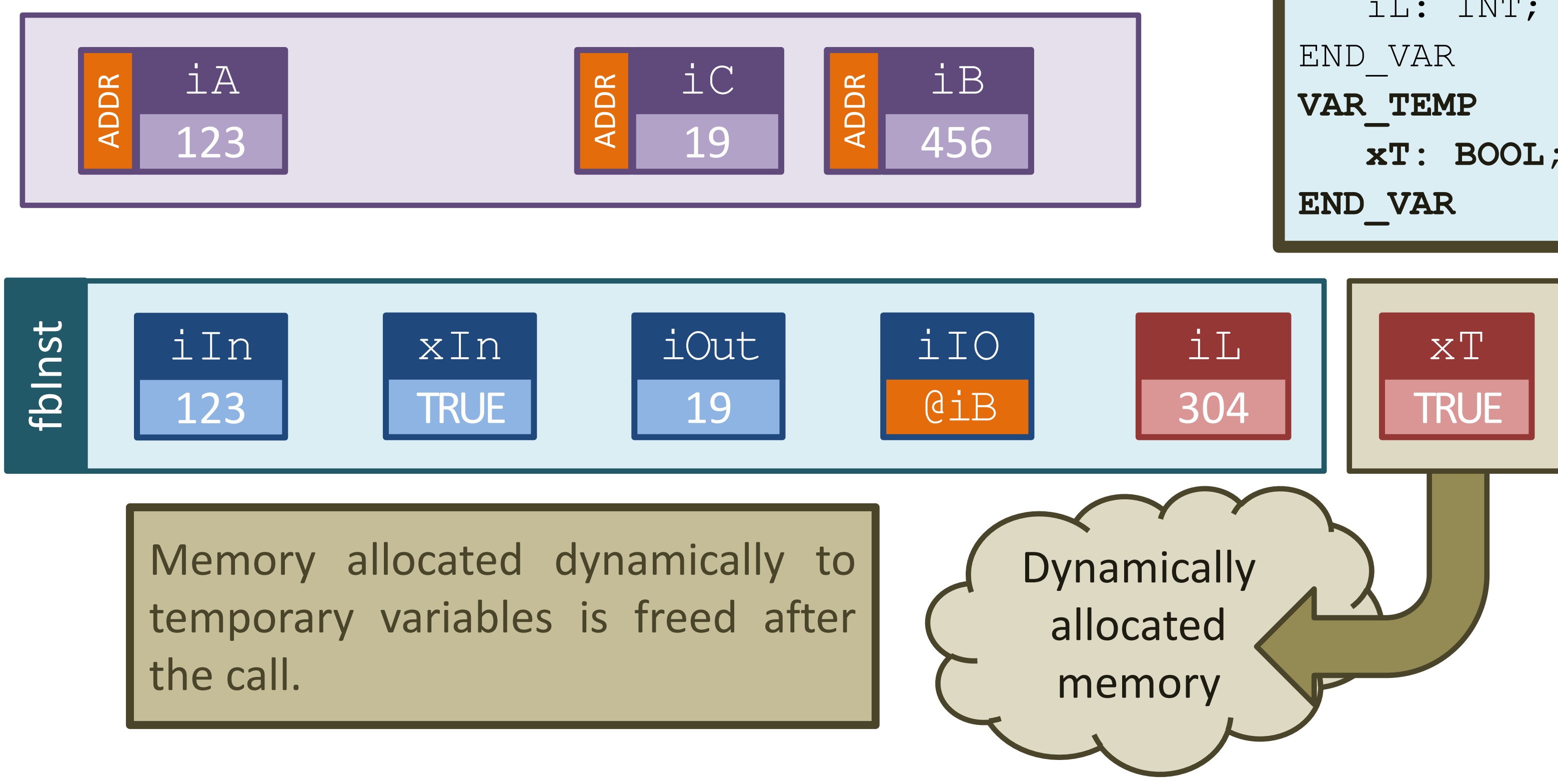
```
fbInst(iIn:=iA,iIO:=iB,
       iOut=>iC);
```




```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
```

# 5. Memory deallocation

```
FUNCTION_BLOCK FB_TYPE
VAR_INPUT
    iIn: INT;
    xIn: BOOL:=TRUE;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
```




Memory allocated dynamically to temporary variables is freed after the call.

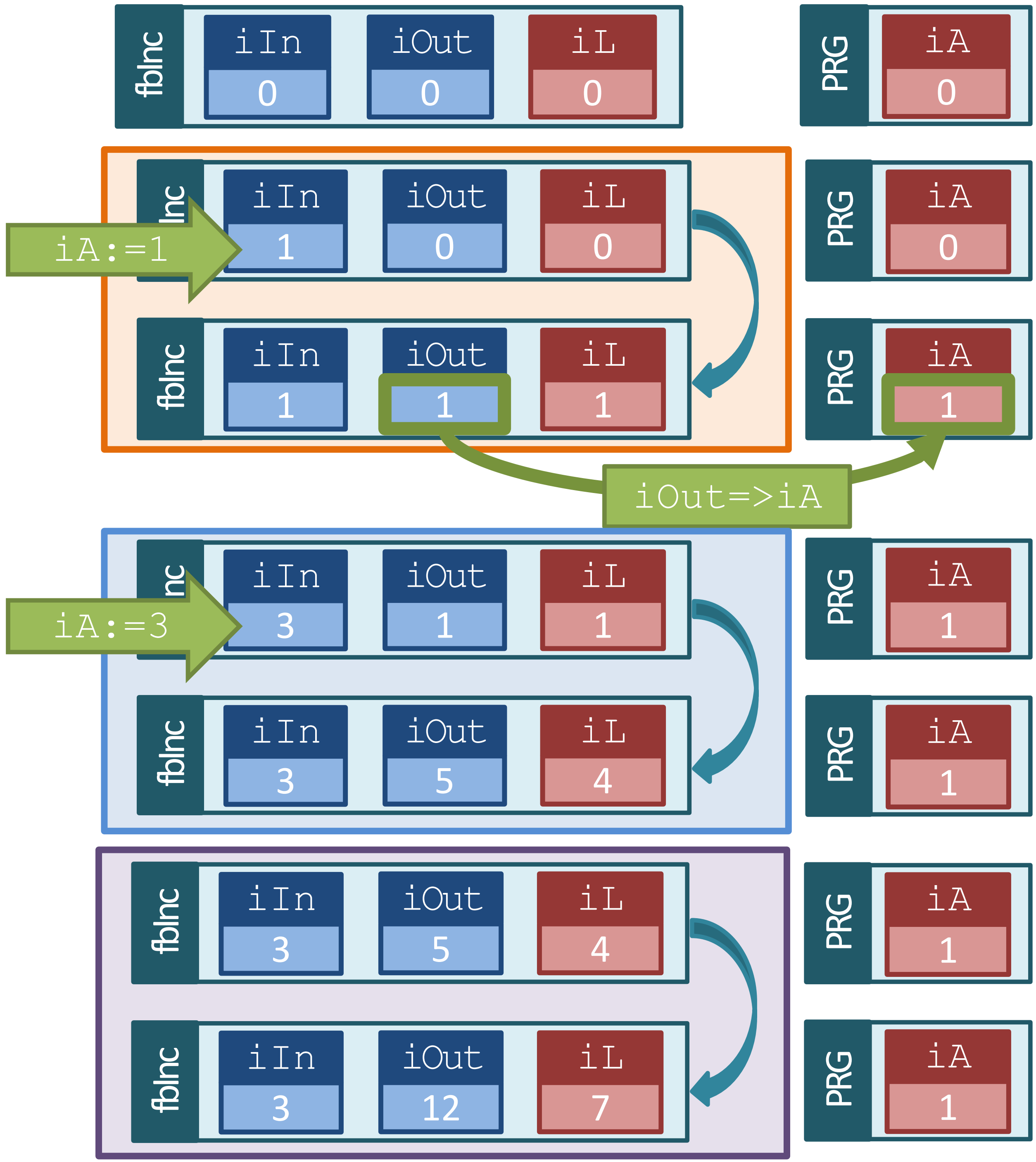
# Missing assignments

- Input variables (`VAR_INPUT`) not assigned during the call retain their values from the preceding call or direct assignment (they are set to their initial values in case of the first call)

- Input-output variables (`VAR_IN_OUT`) need to be assigned during every call!

```
PROGRAM PRG
VAR
    fbInc: INC;
    iA: Int;
END_VAR
MyInc(iIn:=1,iOUT=>iA);
MyInc(iIn:=3);
MyInc();
```

```
FUNCTION_BLOCK INC
VAR_INPUT
    iIn: INT;
END_VAR
VAR_OUTPUT
    iOut: INT;
END_VAR
VAR
    iL: INT;
END_VAR

iL:=iL+iIn;
iOut:=iL+iOut;

END_FUNCTION_BLOCK
```



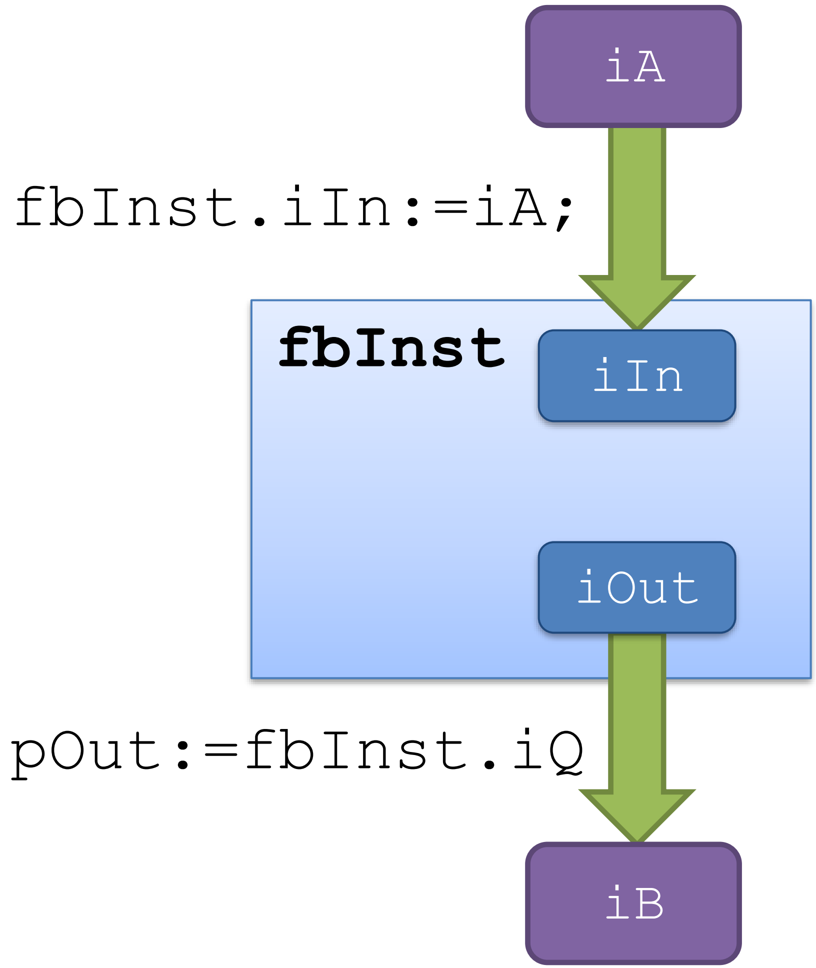
# Function block call

**A function block declared is not executed during every PLC cycle, only if it is called!**

- Access of inputs and outputs of a function block instance might be independent of the call
- When an input of the function block instance is changed, outputs are changed only during the next call

# FB „call” – incorrect

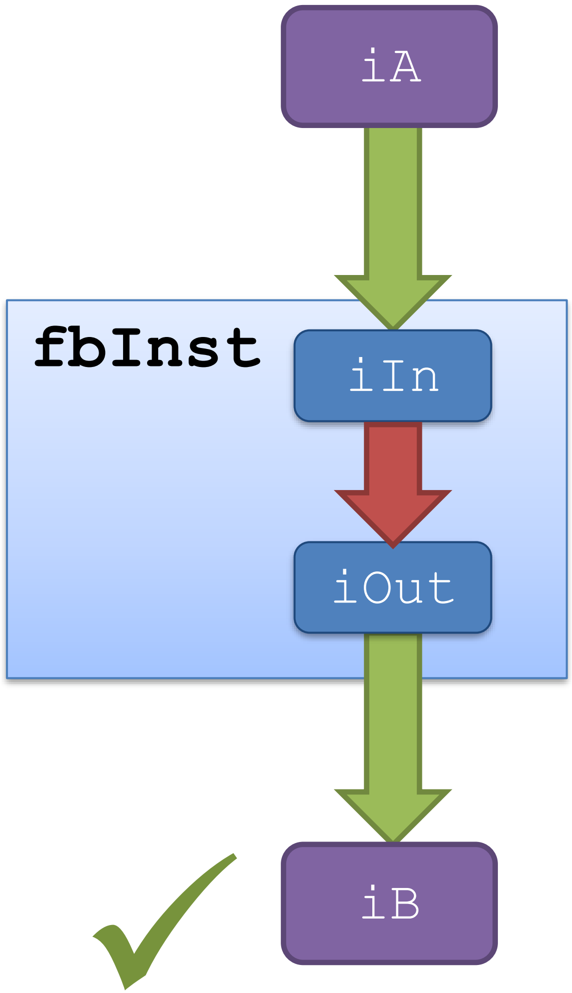
```
PROGRAM MyProg
VAR
    iA:      INT;
    iB:      INT;
    fbInst: FB_TYPE;
END_VAR

fbInst.iIn:=iA;
iB:=fbInst.iOut;
```




**As the function block is not called, its body is not executed, outputs will not be changed.**

# FB call – recommended

```
PROGRAM MyProg
VAR
    iA:      INT;
    iB:      INT;
    fbInst: FB_TYPE;
END_VAR

fbInst( iIn:=iA,
        iOut=>iB);
```

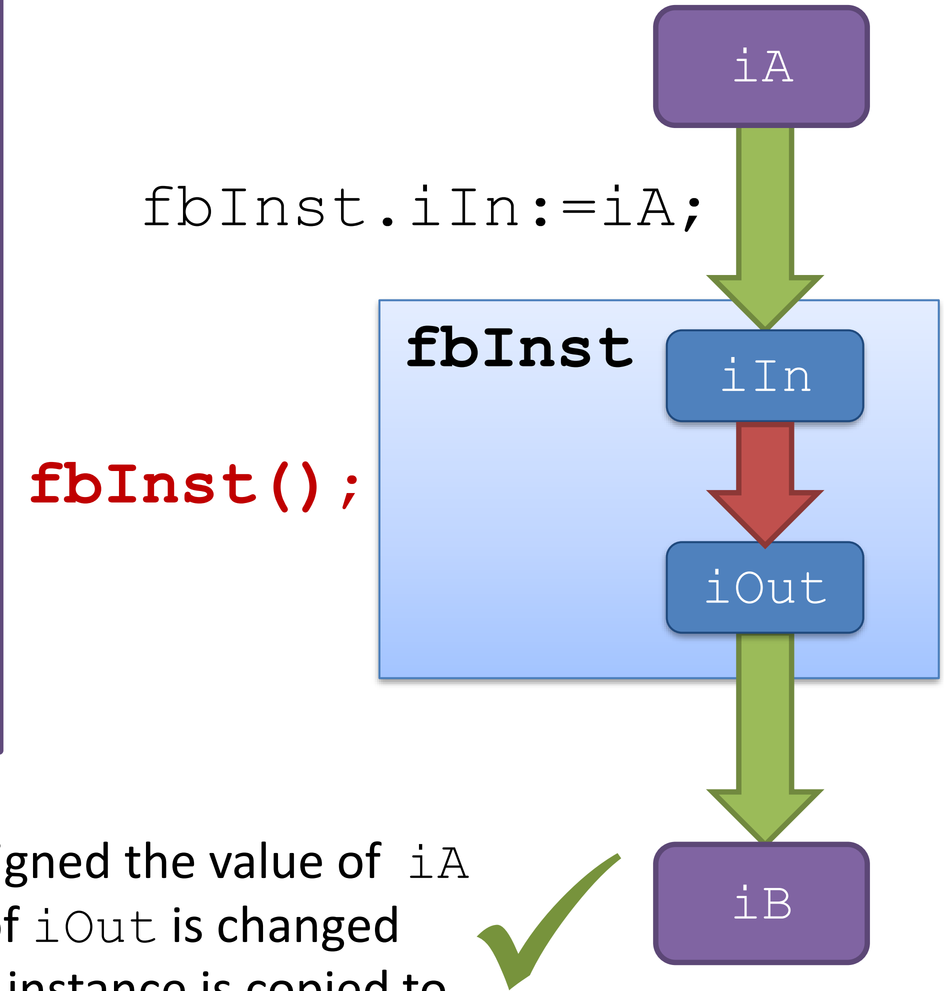



During the call (one single line):

1. variable `iIn` of the instance is assigned value of `iA`
2. body is executed, value of `iOut` is changed
3. new value of the variable `iOut` of the instance is copied to the variable `iB`

# FB call – correct

```
PROGRAM MyProg
VAR
    iA:      INT;
    iB:      INT;
    fbInst:  FB_TYPE;
END_VAR

fbInst.iIn:=iA;
fbInst();
iB:=fbInst.iOut;
```




During the three instructions

1. variable `iIn` of the instance is assigned the value of `iA`
2. body of the FB is executed, value of `iOut` is changed
3. new value of variable `iOut` of the instance is copied to the variable `iB`

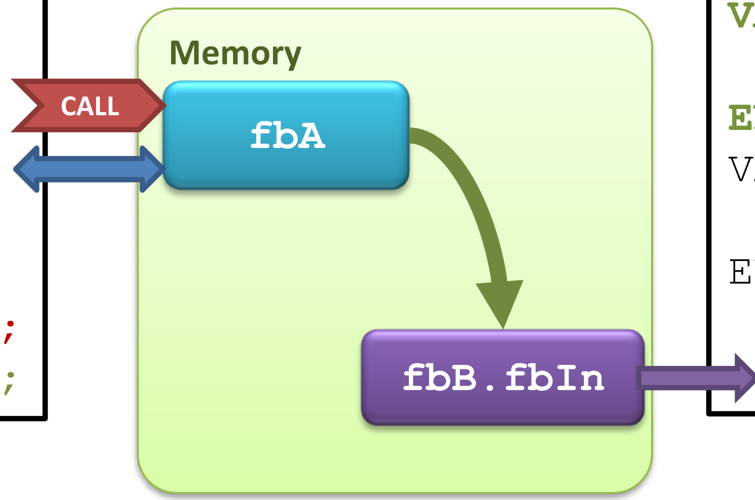
# Using function block instances as parameters

- One might pass a function block instance to a function block or program
  - as a data structure
  - as a callable „object”

- In which cases it might be useful?
  - passing a data set (structure)
  - passing a commonly used module (e.g. logging)

# Passing an FB-instance as input parameter

```
PROGRAM MyProg
VAR
    fbA: FB_A;
    fbB: FB_B;
END_VAR

fbA.xA:=TRUE;
fbA(xA:=FALSE);
fbB(fbIn:=fbA);
```



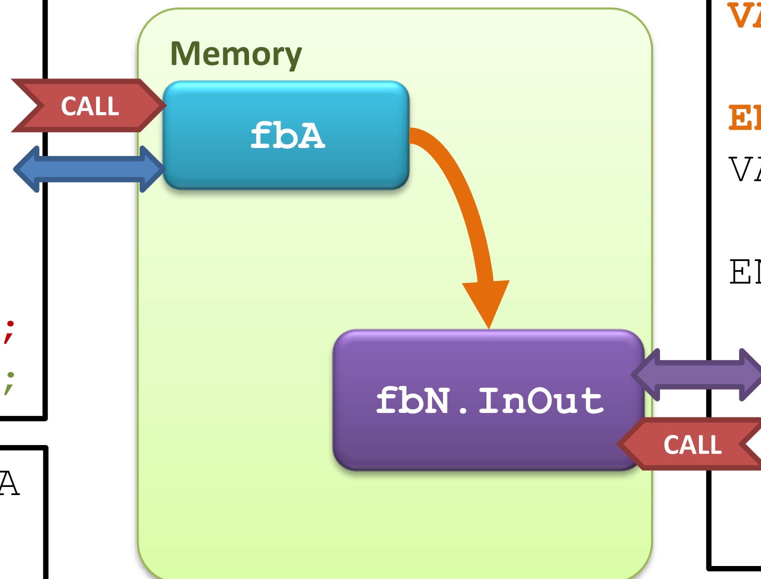

```
FUNCTION_BLOCK FB_B
VAR_INPUT
    fbIn: FB_A;
END_VAR
VAR
    xL,xM: BOOL;
END_VAR

  xL:=fbIn.xQ;
```

```
FUNCTION_BLOCK FB_A
VAR_INPUT
    xA: BOOL;
END_VAR
VAR_OUTPUT
    xQ: BOOL;
END_VAR
(* code *)
```

- Passing variables by value – copy of the data structure of `fbA`
- The called POU (`fbB`) can read output variables of the instance passed (`fbA`)
- POU called can not call the function block instance passed

# Passing an FB-instance as input-output parameter

```
PROGRAM MyProg
VAR
    fbA: FB_A;
    fbB: FB_B;
END_VAR

fbA.xA:=TRUE;
fbA(xA:=FALSE);
fbB(fbIn:=fbA);
```



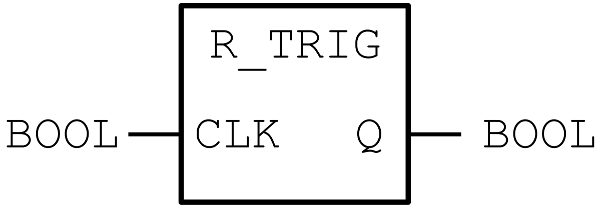

```
FUNCTION_BLOCK FB_B
VAR_IN_OUT
    fbInOut: FB_A;
END_VAR
VAR
    xL,xM: BOOL;
END_VAR

xL:=fbIn.xQ;
fbInOut.xA:=xM;
fbInOut(xA:=xL,
        xQ=>xM);
```

```
FUNCTION_BLOCK TYPE_A
VAR_INPUT
    xA: BOOL;
END_VAR
VAR_OUTPUT
    xQ: BOOL;
END_VAR
(* code *)
```

- Passing the instance (`fbA`) as reference
- Called POU (f`bB`) can read and write interface variables of the passed instance (`fbA`)
- The POU called (`fbB`) might call the instance passed (`fbA`)

# Standard function blocks

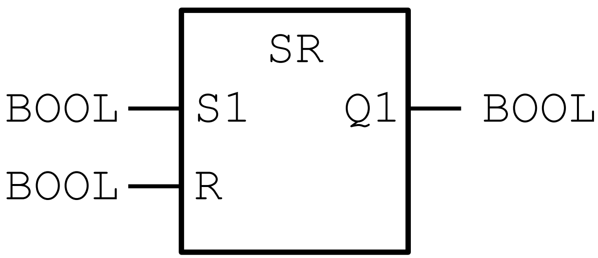
- Bistable elements
  - `SR` – Overriding Set
  - `RS` – Overriding Reset
- Edge-sensing function blocks
  - `R_TRIG`: positive edge sensing
  - `F_TRIG`: negative edge sensing




- Counters
  - `CTU, CTD, CTUD`
- Timers
  - `TON, TOF, TP`

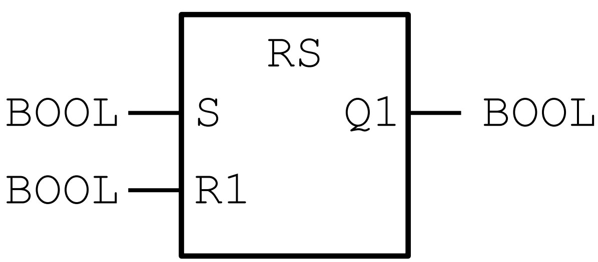
# Bistable elements

- SR: dominating Set




```
IF (S1=TRUE)
  THEN Q1:=TRUE;
  ELSIF (R=TRUE)
    THEN Q1:=FALSE;
END_IF
```

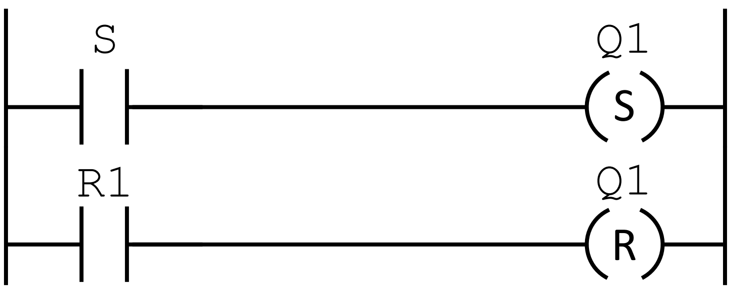
- RS: dominating Reset




```
IF (R1=TRUE)
  THEN Q1:=FALSE;
  ELSIF (S=TRUE)
    THEN Q1:=TRUE;
END_IF
```

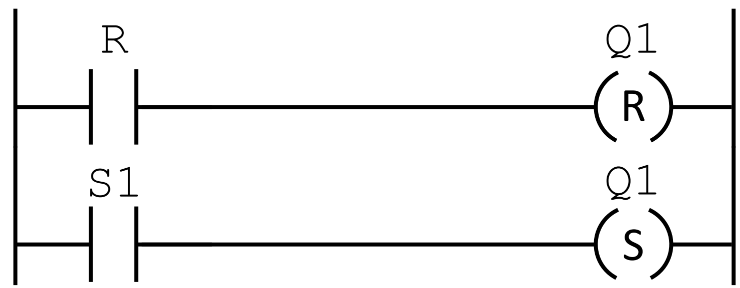
# Implementation of bistable elements - example

```
FUNCTION_BLOCK RS
VAR_INPUT
   R1,S : BOOL;
END_VAR
VAR_OUTPUT
   Q1   : BOOL;
END_VAR
```




```
END_FUNCTION_BLOCK
```

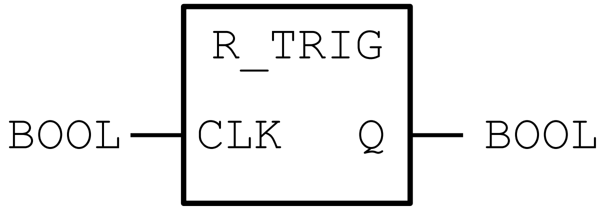
```
FUNCTION_BLOCK SR
VAR_INPUT
   S1,R : BOOL;
END_VAR
VAR_OUTPUT
   Q1   : BOOL;
END_VAR
```



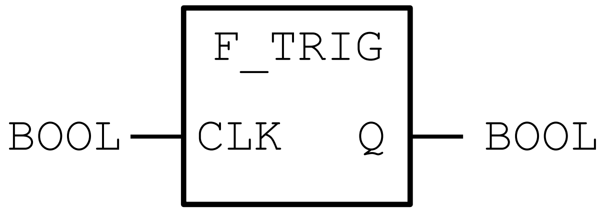

```
END_FUNCTION_BLOCK
```

# Edge-sensing function blocks

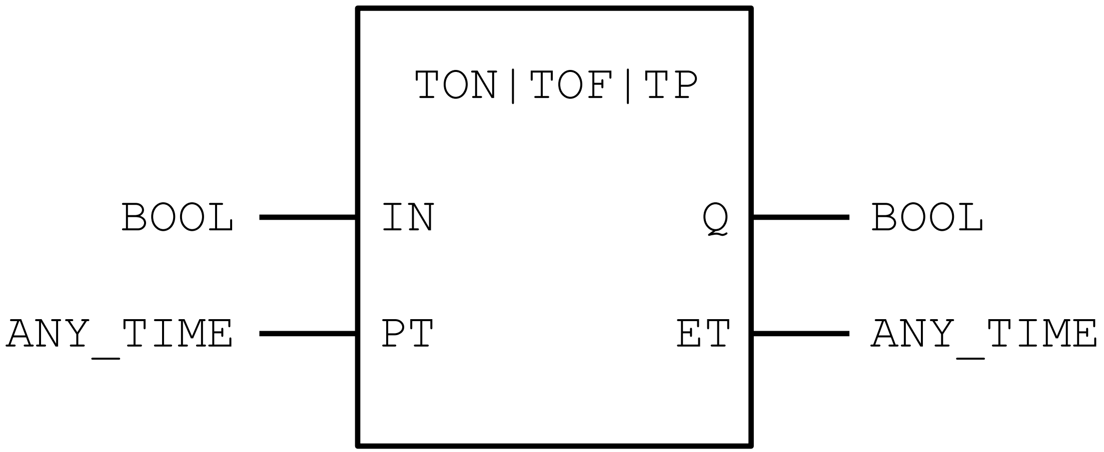
- Positive edge-sensing: `R_TRIG`




- Negative edge-sensing: `F_TRIG`

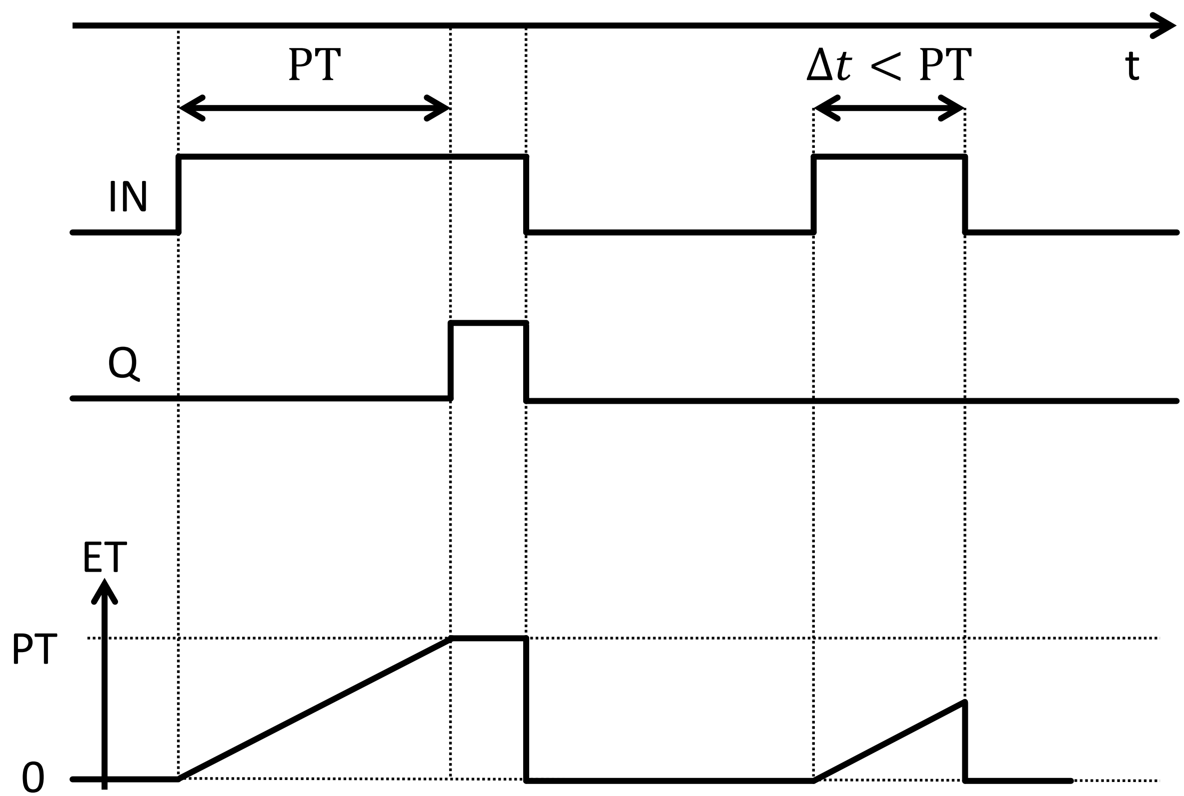


# Timers

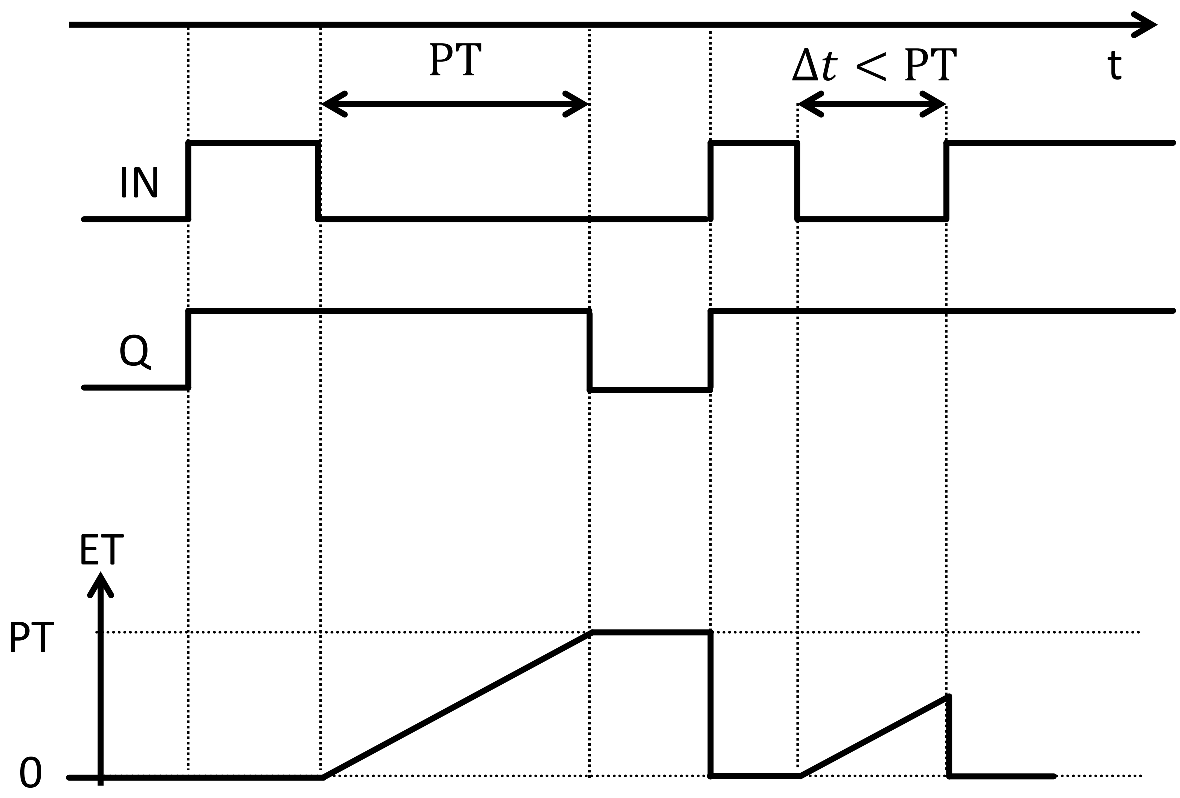



`IN`: Timer input
`PT`: Preset time
`Q`: Timer output
`ET`: Elapsed time

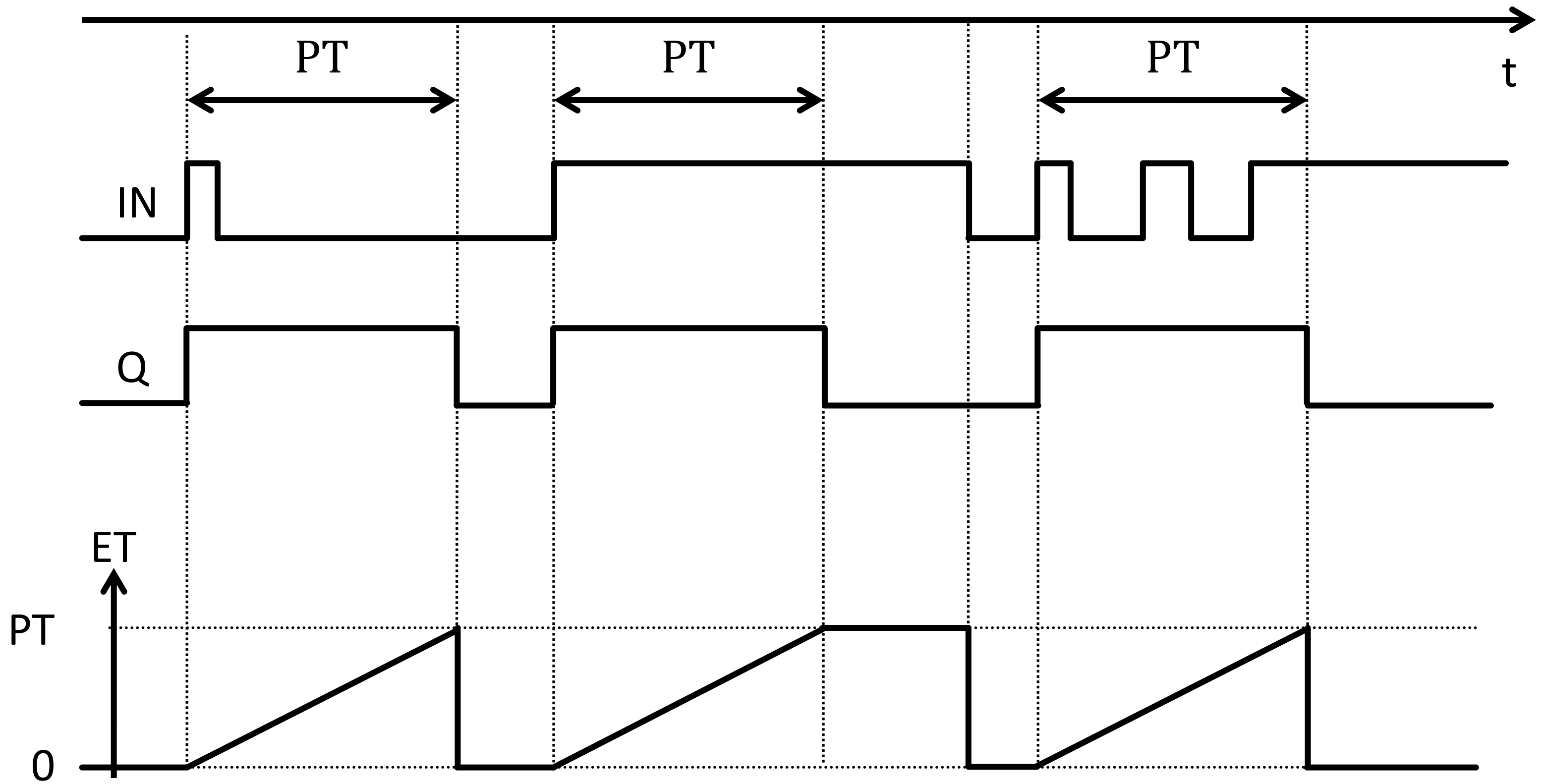
# On-delay timer (TON)



# Off-delay timer (TOF)



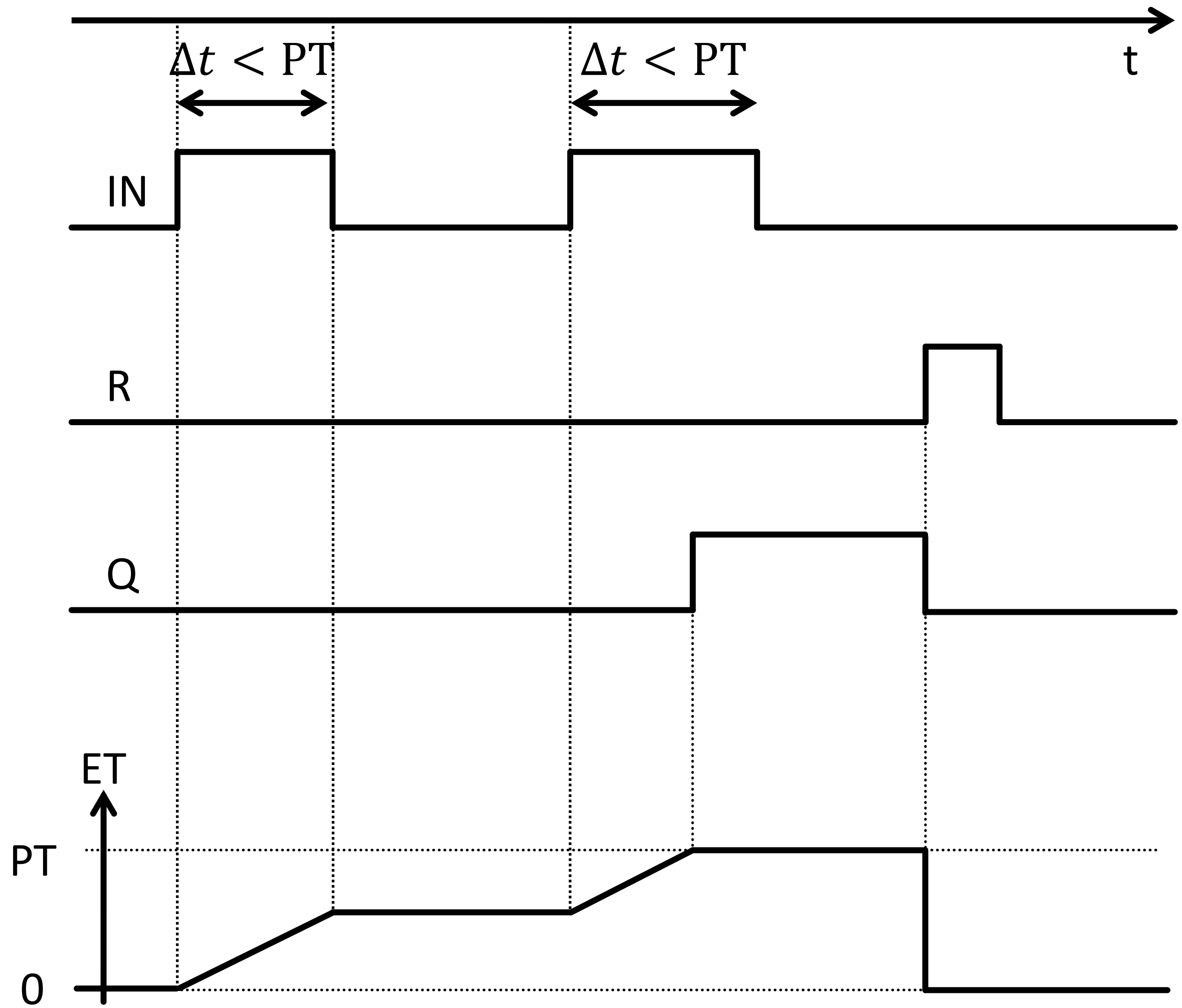
# Pulse timer (TP)



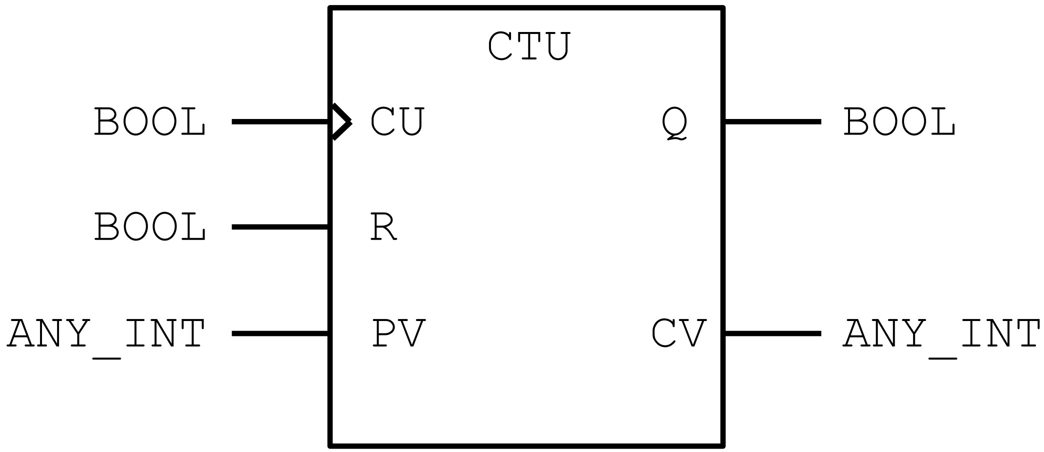
# Retentive timers

- Internal counters of retentive TON/TOF timers are not reset by falling/rising edge of the input
- Retentive timers measure the time passed with value of input 1/0 in a cumulative way
- Reset input is present for resetting the timer value
- Retentive timers are non-standard function blocks available in many development environments

# Retentive TON-timer



# Up-counter (CTU)




CU: **Edge-sensing** counting input

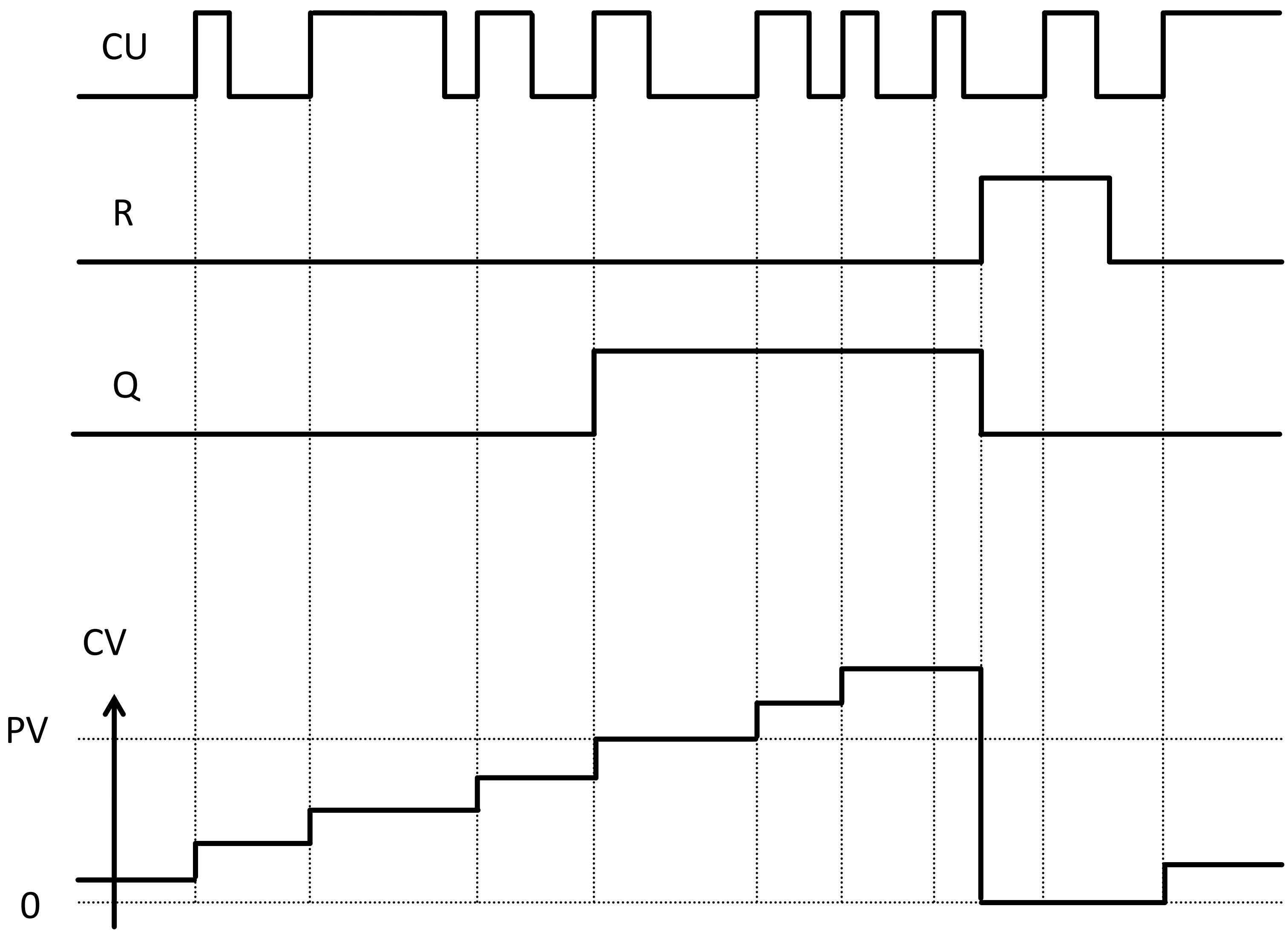
R: Reset – $\mathrm{CV} \coloneqq 0$

PV: Preset value

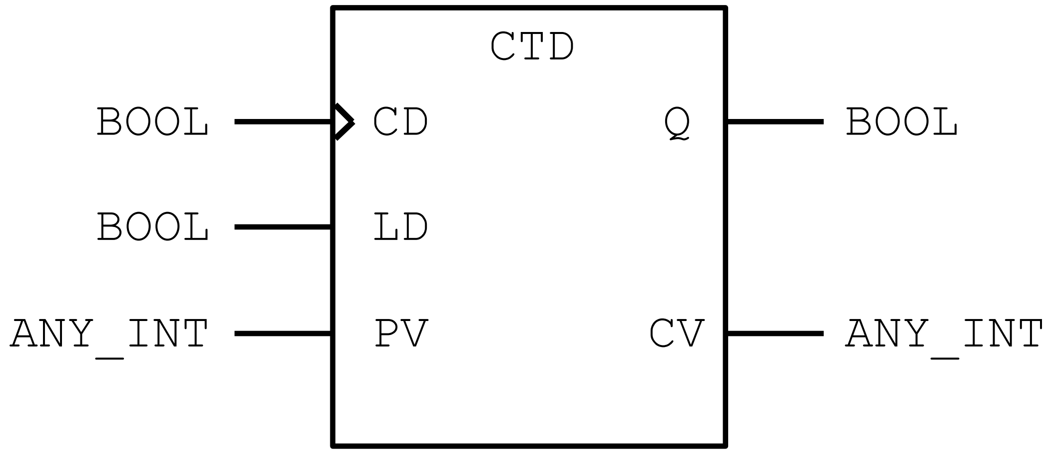
Q: Status output: has the counter value reached the preset value? $\mathrm{Q} = (\mathrm{CV} \geq \mathrm{PV})$

CV: Counter value

# Up-counter (CTU)

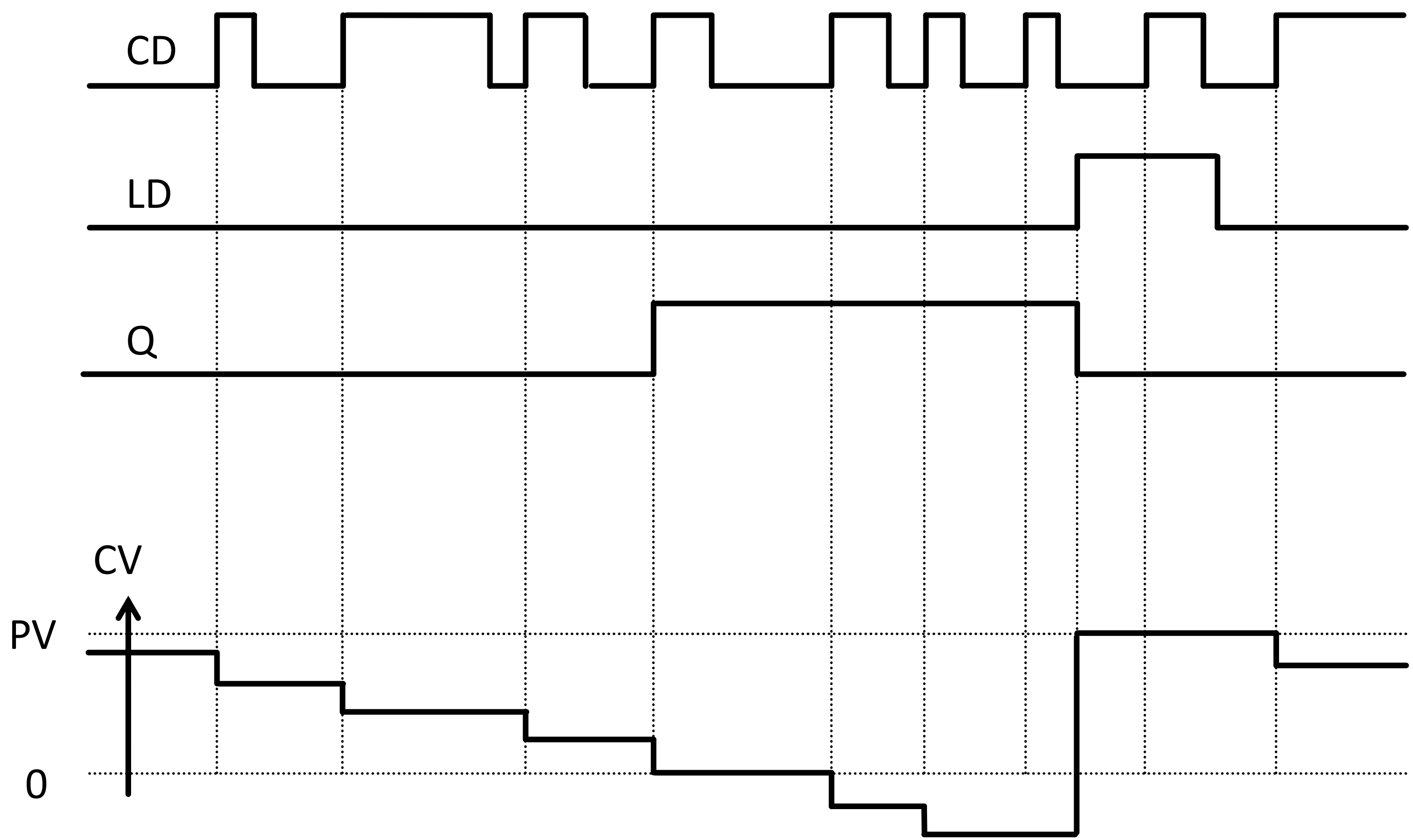


# Down-counter (CTD)



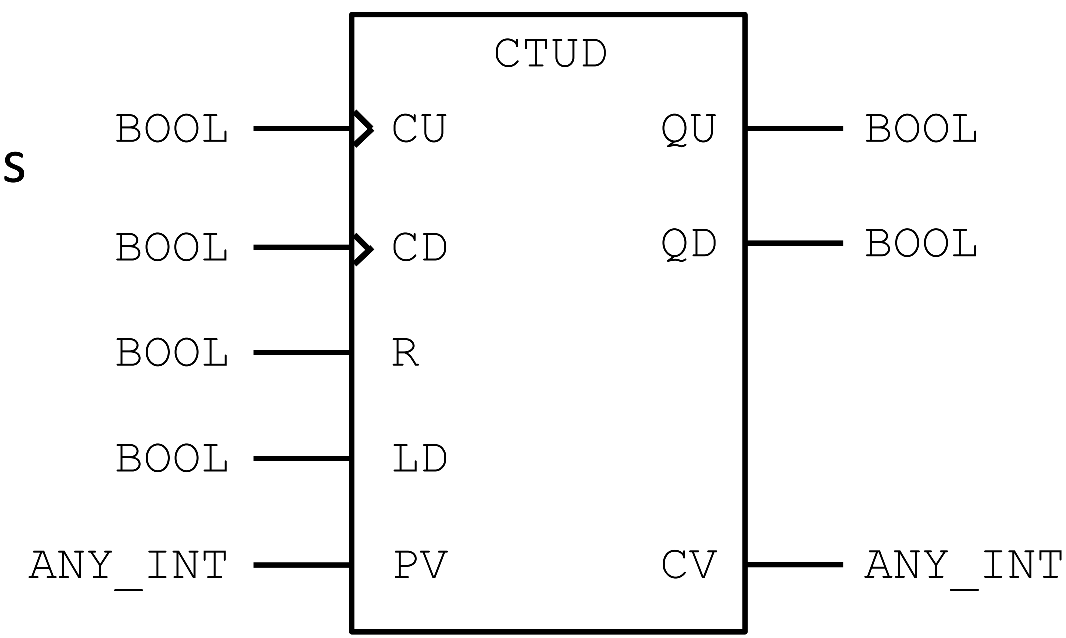

`CD`: **Edge-sensing** counting input
`LD`: Load initial value to the counter value - $\mathrm{CV} \coloneqq \mathrm{PV}$
`PV`: Preset value
`Q`: Status output – has the counter value reached value zero?
$\mathrm{Q} = (\mathrm{CV} \leq 0)$
`CV`: Counter value

# Down-counter

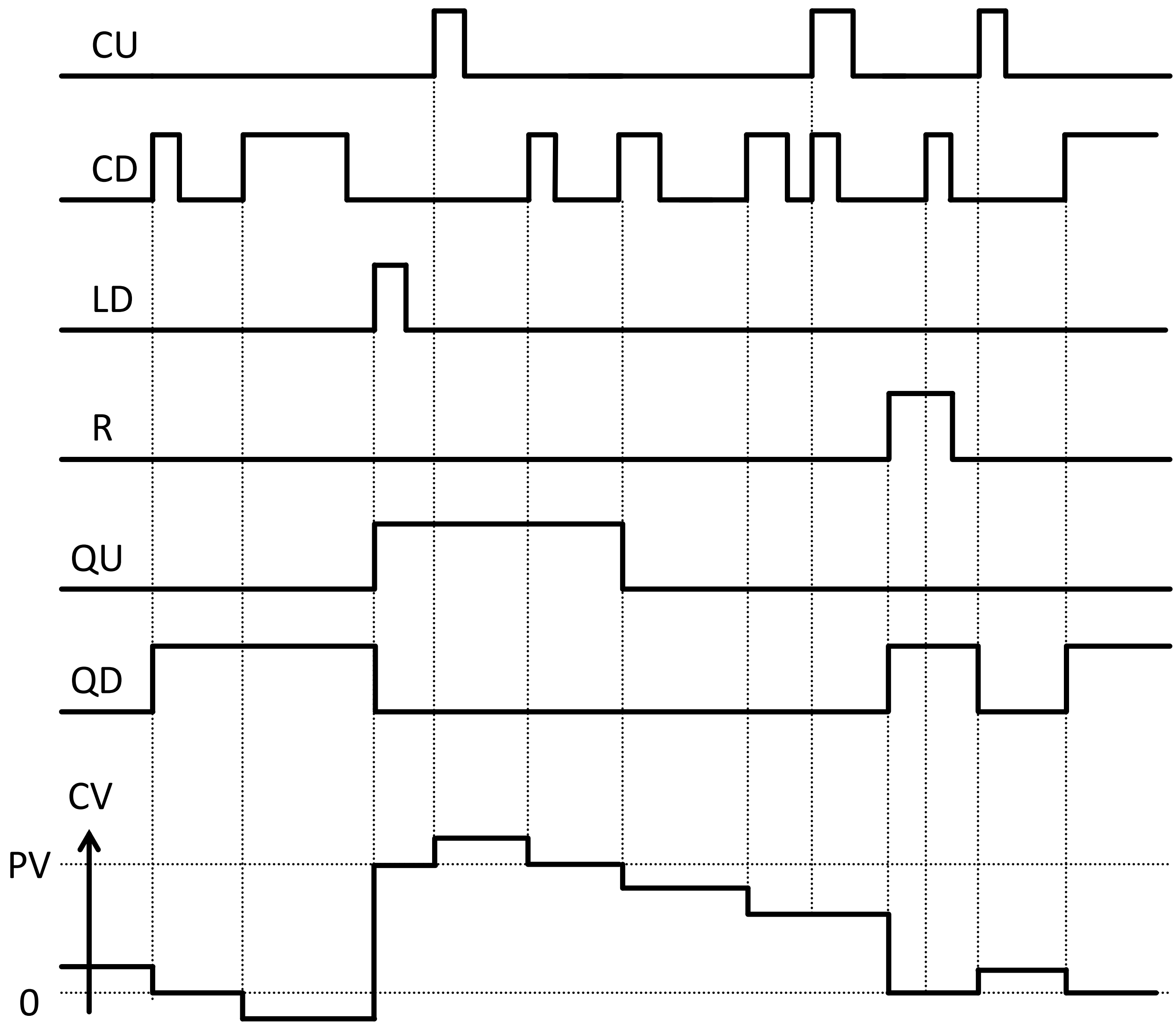


# Up-down counter (CTUD)

`CU`: **Edge-sensing** count up input
`CD`: **Edge-sensing** count down input
`R`: Reset of the counter (higher priority) - $\mathrm{CV} \coloneqq 0$
`LD`: Load of initial value - $\mathrm{CV} \coloneqq \mathrm{PV}$
`PV`: Preset value
`QU`: Up-counting status
$\mathrm{QU} = (\mathrm{CV} \geq \mathrm{PV})$
`QD`: Down-counting status
$\mathrm{QD} = (\mathrm{CV} \leq 0)$
`CV`: Counter value



# Up-down counter (CTUD)




R is dominating over LD

# Implemenation of an up-down counter

`PVmin` and `PVmax` are implementer-specific minimal and maximal values of the counter value

```
FUNCTION_BLOCK CTUD
VAR_INPUT
   CU: BOOL R_EDGE;
   CD: BOOL R_EDGE;
   R:  BOOL;
   F:  BOOL;
   PV: INT;
END_VAR
VAR_OUTPUT
   QU: BOOL;
   QD: BOOL;
   CV: INT;
END_VAR
```

```
IF R
THEN CV:=0;
ELSIF LD
   THEN CV:=PV;
   ELSE
      IF NOT(CU AND CD)
      THEN
         IF CU AND (CV < PVmax)
         THEN CV:=CV+1
         ELSIF CD AND (CV > PVmin)
            THEN CV:=CV-1;
         END_IF;
      END_IF;
   END_IF;
END_IF;
QU:=(CV>=PV);
QD:=(CV<=0);
```

*Code can be understood without exact knowledge of the syntax of the ST language.*

# User-defined function blocks

- User might define or instantiate any arbitrary function block type
- User-defined function blocks might call any standard or user-defined function
- User-defined function blocks might instantiate and call any standard or other used-defined function block

# Program

- „Main routine”
- Additionally to the features of function blocks programs might host global variables (`VAR_GLOBAL`)
- Programs can not be called by other POUs
- Programs are scheduled by tasks
  - tasks instantiate programs
  - multiple instances of a program might be scheduled

# Interface variables of programs

- Programs might have interface variables
- Input and output variables of programs
  - might be assigned in the declaration of a task
  - might be assigned to physical inputs and output in the declaration part (recommended in case of simple applications)

# Program declaration

`PROGRAM` **keywords**
*denotes start of program declaration*

Identifier of the program

```
PROGRAM MyPRG
VAR_INPUT
    iIn: INT;
END_VAR
VAR_OUTPUT
    xOut: BOOL;
END_VAR
VAR_IN_OUT
    iIO: INT;
END_VAR
VAR
    iL: INT;
END_VAR
VAR_TEMP
    xT: BOOL;
END_VAR
VAR_GLOBAL
    iG: INT;
END_VAR
VAR_EXTERNAL
    xG: BOOL;
END_VAR
(* BODY *)
END_PROGRAM
```

Input variables
*optional, but at least one input variable is used in practice*

Output variables
*optional, allows passing values to the caller*

Input - output variables
*optional, passed by reference*

Local variables
*optional, values **stored** between calls*

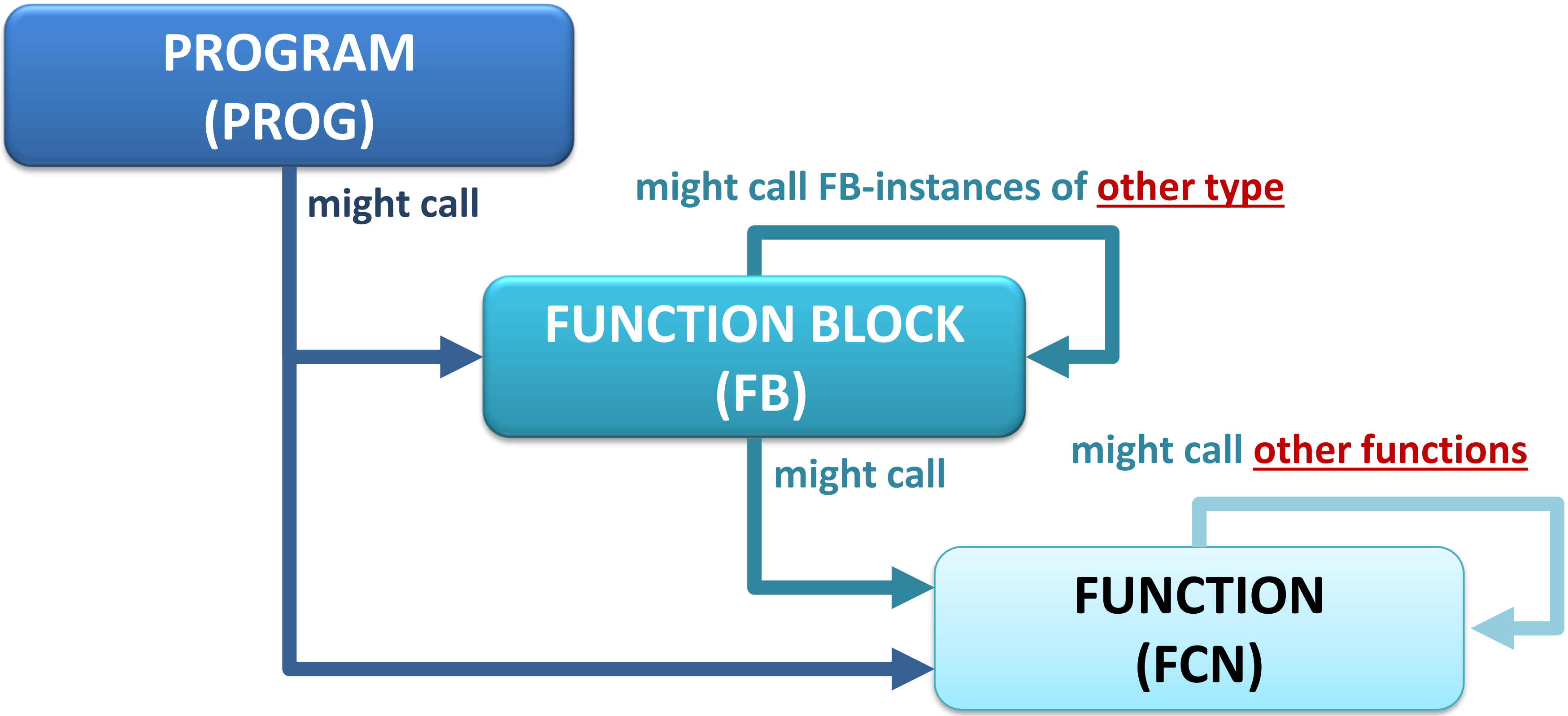
Temporary variables
*optional, values **not stored** between calls*

Global variables hosted by the program
*optional*

Global variables hosted by other POUs
*optional*

`END_PROGRAM` *keyword for closing declaration*

# POU calls




- Textual call – in textual languages (IL, ST)
- Graphical call – in graphical languages (LD, FBD)

# Recursion is forbidden

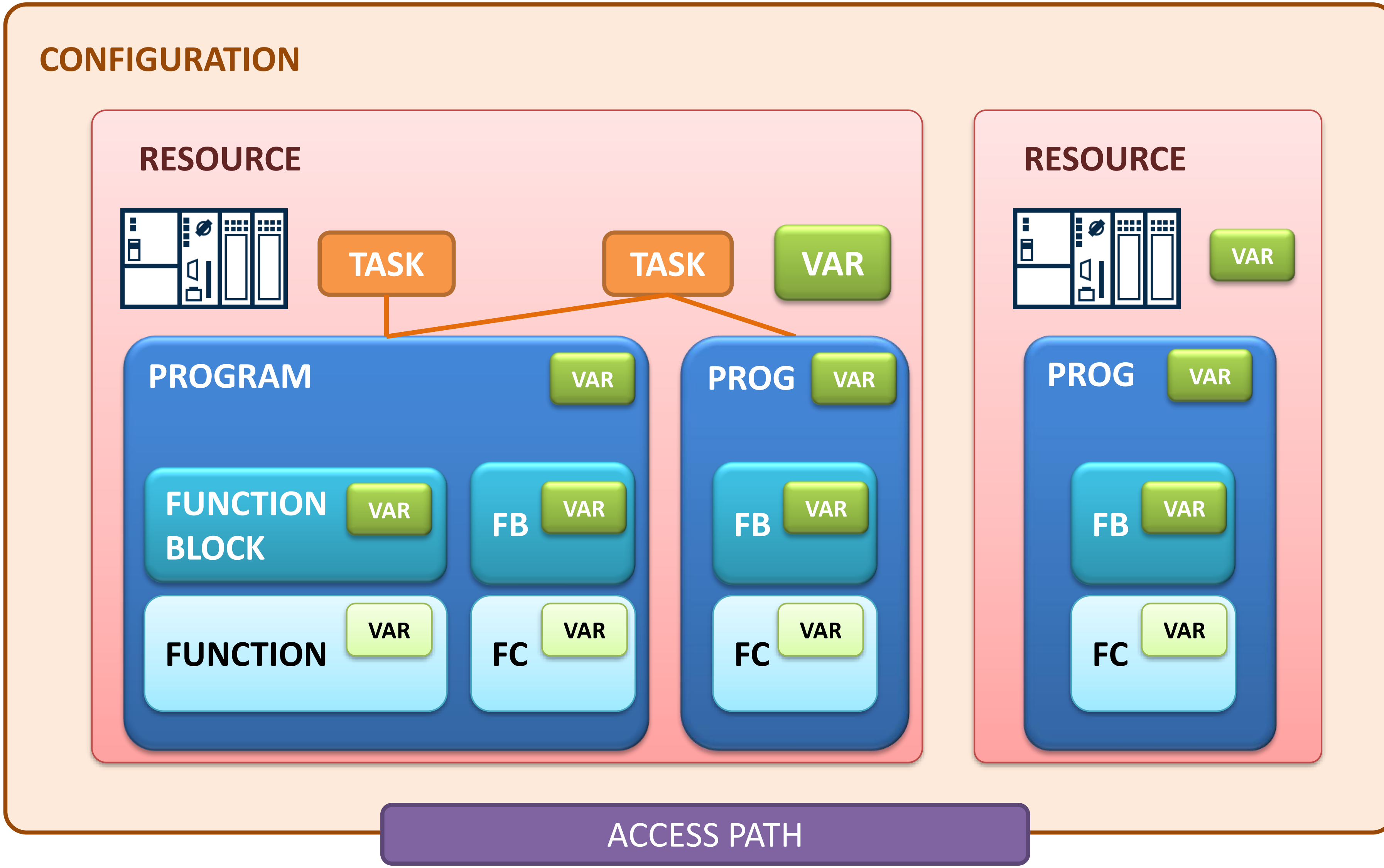
- Allowed by the standard but forbidden in most environments
- A POU shall not call itself neither in a direct nor in an indirect way
- Calling an other instance of a POU of the same type is considered as recursion
- Best practice: avoid recursion

```
FUNCTION_BLOCK FB1
VAR
   fbInst: FB1;
END_VAR
…
fbInst(…)
…
```

```
FUNCTION FC1
…
FC2(…)
…
```

```
FUNCTION FC2
…
FC1(…)
…
```

# Overview



CONFIGURATION
RESOURCE
TASK
TASK
VAR
PROGRAM
VAR
FUNCTION BLOCK
VAR
FB
VAR
FUNCTION
VAR
FC
VAR
PROG
VAR
FB
VAR
FC
VAR
RESOURCE
VAR
PROG
VAR
FB
VAR
FC
VAR
ACCESS PATH

# Tasks

- Programs and FB instances are associated to resources (CPUs) by the tasks
- Execution of programs and FB instances is governed by tasks
    - instances of a program or FB-type might be associated to different tasks
    - a task might be associated with multiple programs or FB instances
- By assigning to a task, a program is transformed to a run-time object
- Each time a task is scheduled, it executes associated programs and function block instances <u>once</u>

# Scheduling types

- Periodic
  - runs periodically (e.g. every 10ms)
- Event driven (*interrupt*)
  - executed <u>once</u> upon the rising edge of a Boolean variable
- Program without explicit task assignment *(freewheeling)*
  - cyclic execution with lowest priority
  - executed when CPU is not used by other tasks

# Task priority

- Multiple priority levels

  (number of levels is implementer specific)

- 0: highest priority

- The same priority might be assigned to multiple tasks

# Task declaration

`TASK <ID>(SINGLE,INTERVAL,PRIORITY)`

- `ID` (`STRING`) – task identifier
- `SINGLE (BOOL)`
  - Boolean input, task scheduled at rising edge
  - only for event driven tasks
- `INTERVAL (TIME)`
  - period time
  - only for periodic tasks
- `PRIORITY (INT) - priority level`

# Task declaration – example

```
TASK PeriodicTask(INTERVAL:=T#100ms,PRIORITY:=9);
```

Periodic task, scheduled each 100ms. Priority level 9 (low priority).

```
TASK EventTask(SINGLE:=%I0.4,PRIORITY:=1);
```

Event driven task, executed upon each rising edge of physical input `%I0.4`. Priority level 1 (high priority).

# Association of programs and tasks

```
PROGRAM <ProgInstance>
WITH <TaskID>: ProgType();
```

- `ProgInstance`: identifier of program instance scheduled by the task
- `TaskID`: task identifier
- `ProgType`: identifier of the program type (`PROGRAM <ProgType>` in declaration)
- Association of input, output and input-output variables to interface variables of the program are given in parentheses

```
PROGRAM ProgType1
VAR_INPUT
   xA1: BOOL;
END_VAR
VAR_OUTPUT
   xY1: BOOL;
END_VAR
(* … *)
END_PROGRAM
```

```
PROGRAM ProgType2
VAR_INPUT
   xA2: BOOL AT %I1.2;
END_VAR
VAR_OUTPUT
   xY2: BOOL AT %Q0.3;
END_VAR
(* … *)
END_PROGRAM
```

```
TASK PeriodicTask(INTERVAL:=T#100ms,PRIORITY:=9);
TASK EventTask(SINGLE:=%I0.4,PRIORITY:=1);

PROGRAM PProg WITH PeriodicTask:ProgType1(xA1:=%I0.0,xY1=>%Q0.2);
```

This instance of `ProgType1` is executed periodically.

```
PROGRAM EProg WITH EventTask:ProgType1(xA1:=TRUE,xY1=>%Q0.1);
```

Other instance of `ProgType1` is executed upon rising edge of input `%I0.4`

```
PROGRAM CProg: ProgType2;
```

Instance of `ProgType2` runs cyclically when the CPU is free (no explicit task association)

# Execution of FB instances

- Function block instances are declared by programs or other FB instances

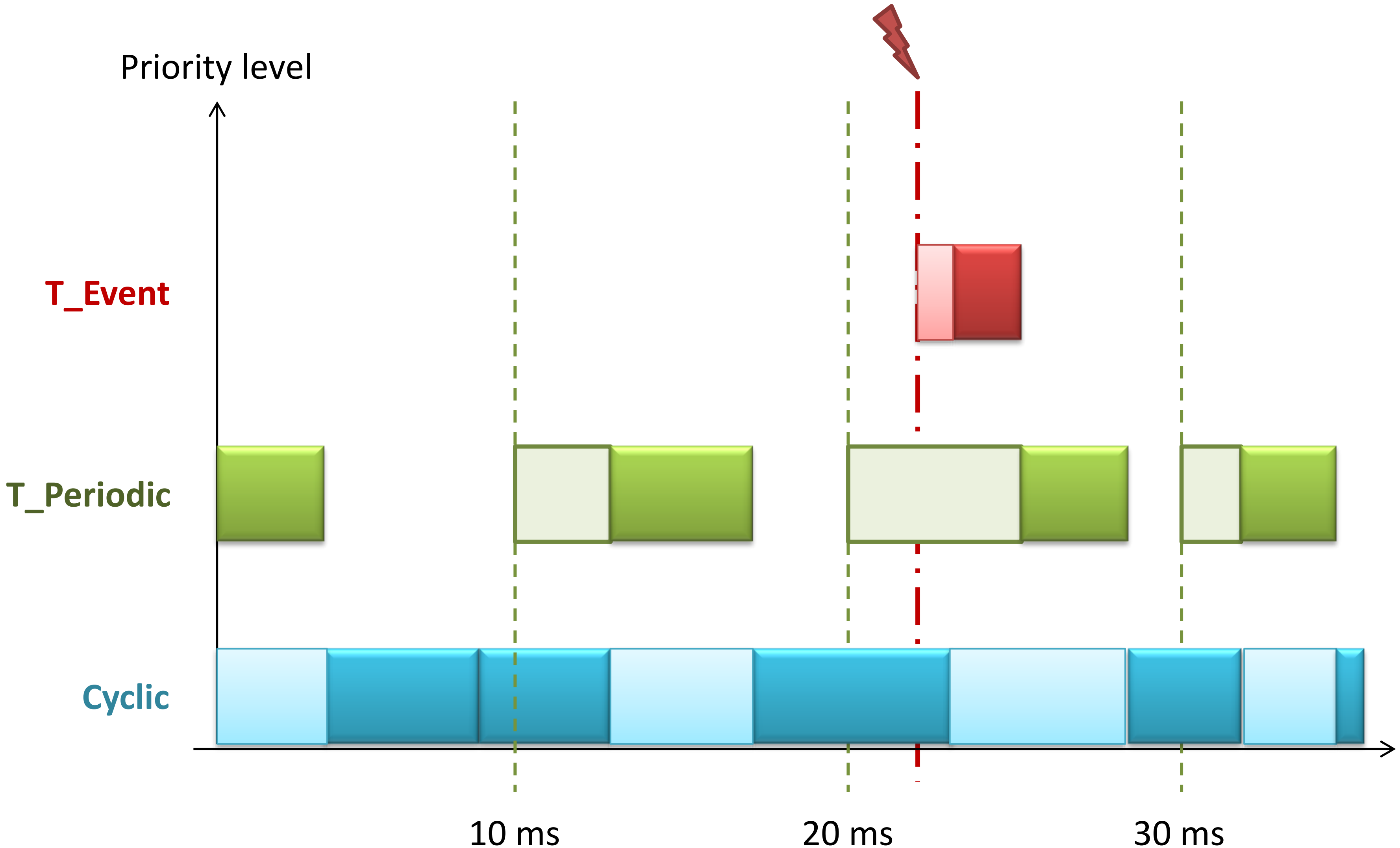
- Execution of a function block instance
  - execution of an FB instance is governed by a task if it is associated explicitly to the FB instance
  - execution of an FB instance not associated to a task is governed by the execution of the program in which the instance is declared

**Best practice: do not associate FB instances to tasks; scheduling of FB execution should be left to tasks scheduling programs in which FB instances are instantiated**

# Non-preemptive scheduling

- Tasks can not interrupt execution of other tasks
- Tasks ready to run are queued
- If all POUs associated to a task have finished their execution, the highest priority task in the queue is scheduled
- Tasks with the same priority are scheduled according to their duration waiting to avoid starvation

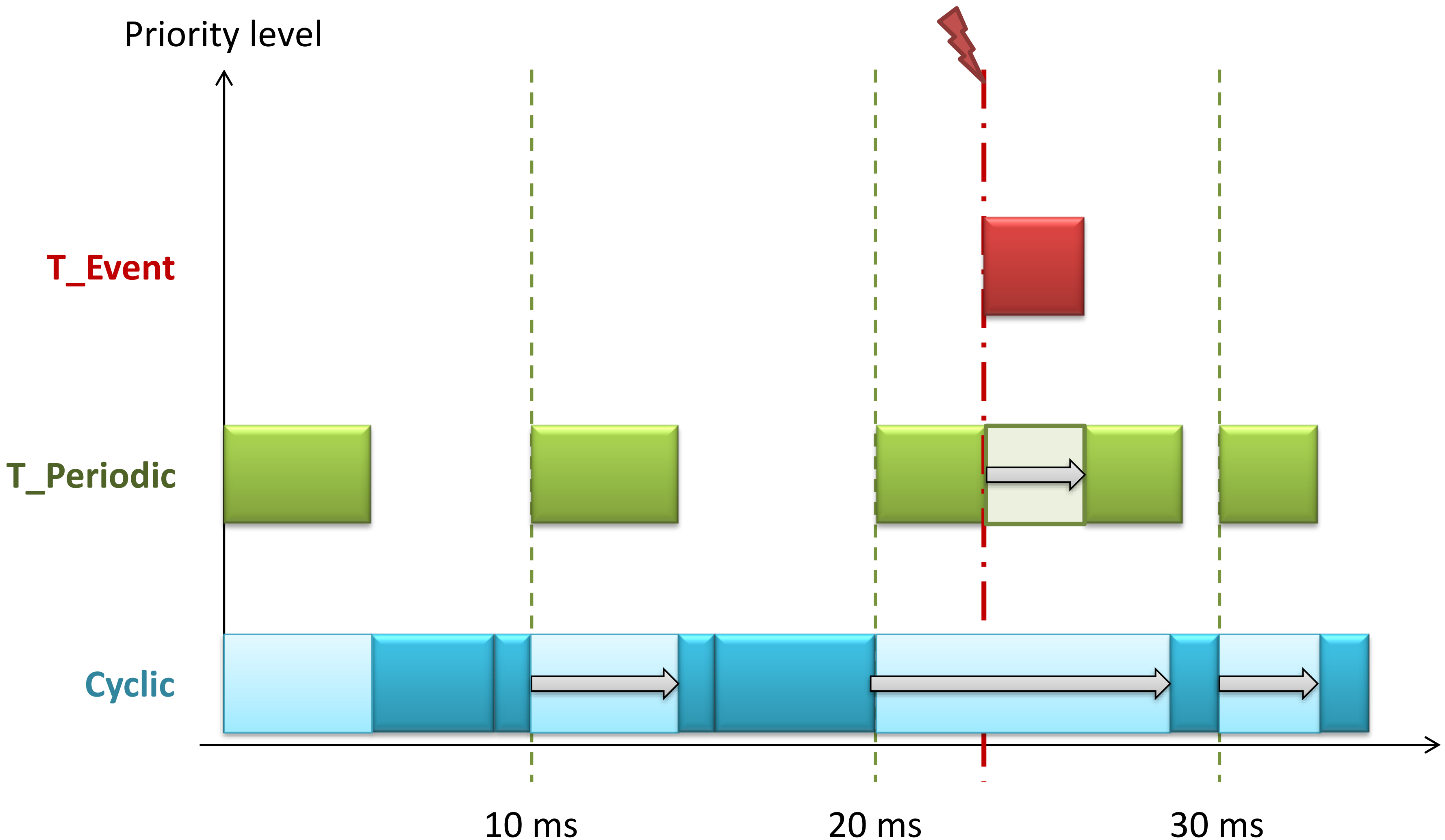
# Non-preemptive scheduling



# Preemptive scheduling

- If a higher priority task is scheduled, it might interrupt execution of POUs assigned to a lower priority task
- Execution of the POU interrupted is suspended and continued only if there are no tasks with higher priority in the queue

# Preemptive scheduling



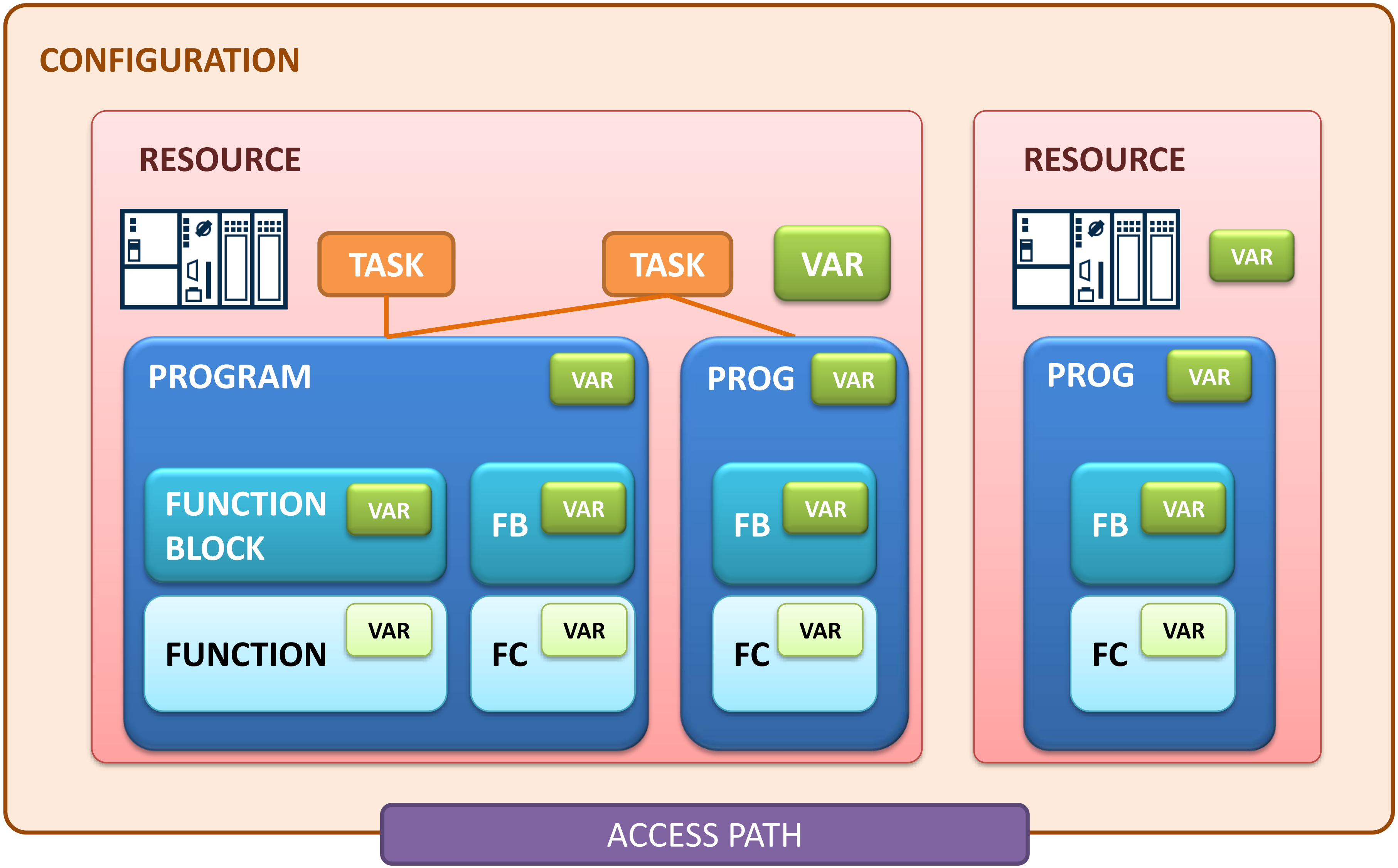
# Scheduling caveats

- Tasks with higher priority might prevent execution of ones with lower priority
- Periodic tasks might miss samples
- Input, output and global variables might be modified by POUs scheduled by other tasks in case of preemptive scheduling
- Careful design of scheduling is recommended

# Rules of thumb

- Do not use too many tasks
- A variable shall be written in one single task
- A program shall read a variable written in a POU scheduled by another task only once per execution
- Tasks shall schedule programs only, not FB instances
- Details of scheduling is implementer specific – always read the manual

# Overview


CONFIGURATION
RESOURCE
TASK
TASK
VAR
PROGRAM
VAR
FUNCTION BLOCK
VAR
FB
VAR
FUNCTION
VAR
FC
VAR
PROG
VAR
FB
VAR
FC
VAR
RESOURCE
VAR
PROG
VAR
FB
VAR
FC
VAR
ACCESS PATH

# Resources

- A resource corresponds to a CPU
- Elements associated to a resource:
  - global variables available for all POUs running on the given resource (if declared in a `VAR_EXTERNAL` block)
  - tasks and POUs scheduled by them
- Directly represented variables of programs can be assigned to physical addresses  inside a resource

# Resource - example

Resource identifier

```
RESOURCE MyResource ON CPU001
```

CPU identifier

```
VAR_GLOBAL
      xG : BOOL;
END_VAR
```

Variable accessible for each POU of the resource having an `xG: BOOL;` declaration inside a `VAR_EXTERNAL` block

```
TASK T_Periodic(INTERVAL:=t#10ms,PRIORITY:=10);
```

Task associated to the resource

```
PROGRAM PeriodicProg WITH T_Periodic:
      ProgType1(Start:=%I0.1,Fail=>xG);

END_RESOURCE
```

Program running on the resource, scheduled by the task `T_Periodic`

*Note: notation is defined by the standard, most development environments use a graphical form (project tree) for association of elements to CPUs.*

# Configuration

- Configurations might join multiple resources
- Configurations provide a way to associate instance-specific locations and initial values to variables of POUs
- Not supported by state-of-the-art development environments

Wildcard (`*`) notation suggests that the given variable is associated to a specific physical address in the configuration

```
PROGRAM MyProg
VAR_INPUT
   xA AT %I* : BOOL;
END_VAR
(* … *)
END_PROGRAM
```

```
CONFIGURATION MyConfig
   RESOURCE MyResource ON CPU001
      PROGRAM P1: MyProg;
   END_RESOURCE
   MyResource.P1.xA AT %IX0.2: BOOL:=TRUE;
END_CONFIGURATION
```

Physical address and initial value of variable `xA` of the program `P1` running on the resource `MyResource` is given in the configuration

# Access Path

- *Access Path* variables declared inside a configuration provide a way for information exchange between configurations
- Other configurations might access value of variables of POUs running inside a configuration by referring to a simple label
- Not supported (in this form) in practice

```
CONFIGURATION MyConfig
   RESOURCE MyResource ON CPU001
      PROGRAM P1: MyProg;
   END_RESOURCE
   VAR_ACCESS
      Alpha: MyResource.P1.xA : BOOL READ_WRITE;
      Gamma: MyResource.P1.uiQ : UINT READ_ONLY;
   END_VAR
END_CONFIGURATION
```

Variable `xA` of the program `P1` running on resource `MyResource` might be read and written by other configurations using the identifier `Alpha`

Variable `uiQ` of the program `P1` running on resource `MyResource` can be read only using the label `Gamma`