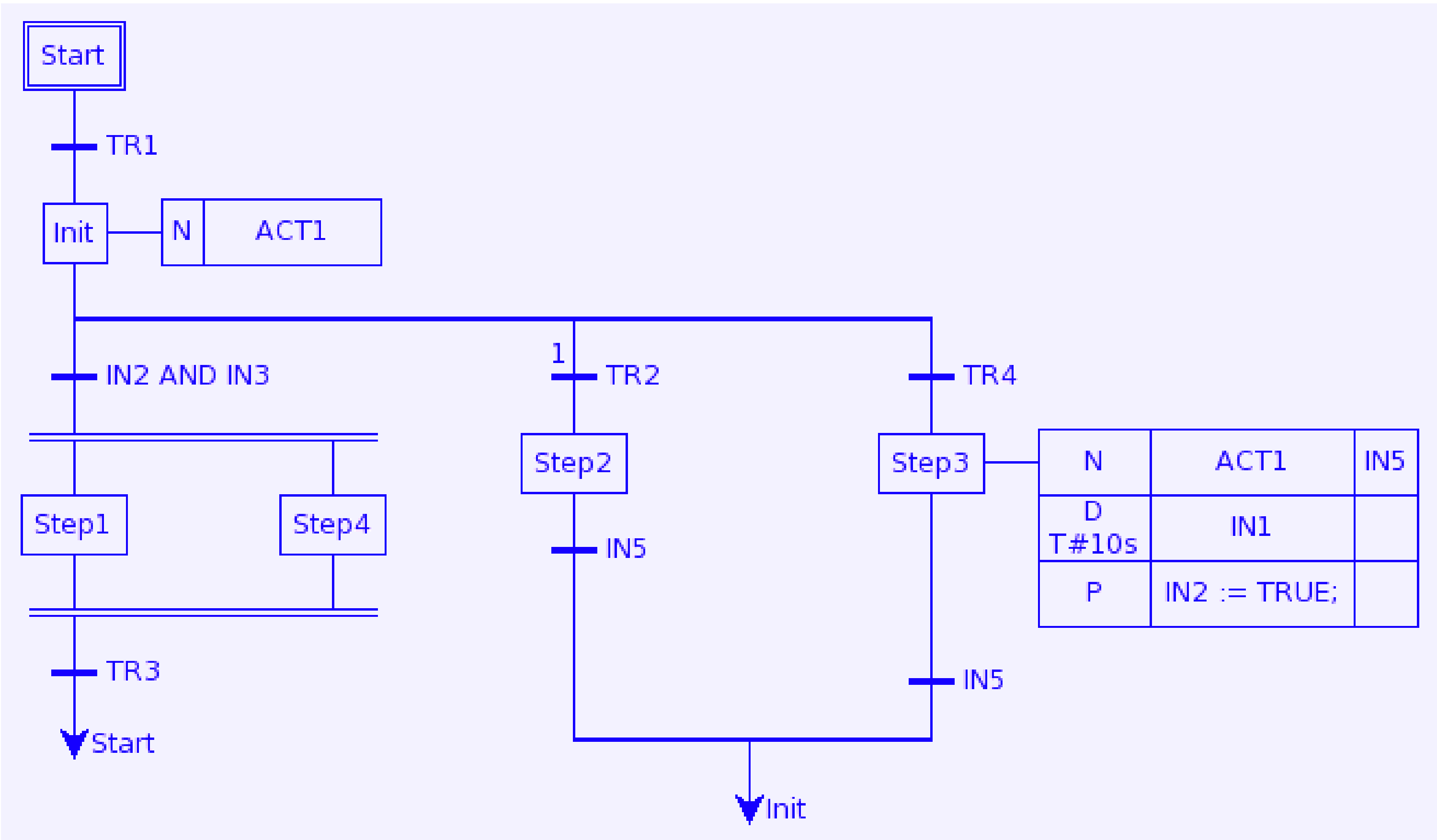
Overview of IEC 61131-3



Mário de Sousa

msousa@fe.up.pt

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Overview of IEC 61131-3

- **Introduction to IEC 61131 and Related Standards**
- **IEC 61131-3 Standard Architecture**
- **Programming Languages**
- **IEC 61131-3 Execution Model**
- **Standard Functions and Function Blocks**
- **Examples**

Introduction to IEC 61131

- IEC 61131 standardizes many aspects of PLCs
- IEC 61131 is composed by 8 parts:
  - 61131-1 General information
  - 61131-2 Equipment requirements and tests
  - 61131-3: Programming languages
  - 61131-4: User guidelines
  - 61131-5: Messaging service specification
  - 61131-6: Communications via fieldbus
  - 61131-7: Fuzzy control programming
  - 61131-8: Guidelines for the application and implementation of programming languages (Non Normative)

Introduction to IEC 61131

- IEC 61131-3
  - First version approved in 1992
  - Second version approved in 2003
    - Introduces some changes, but these will mostly go unnoticed by most programmers
  - Third version approved in 2013
    - Introduces many innovations, including full support for Object Oriented programming (classes, inheritance, interfaces, ...).

Many Manufacturers are currently supporting some variation of the second version.

These slides focus on version 2

Introduction to IEC 61131

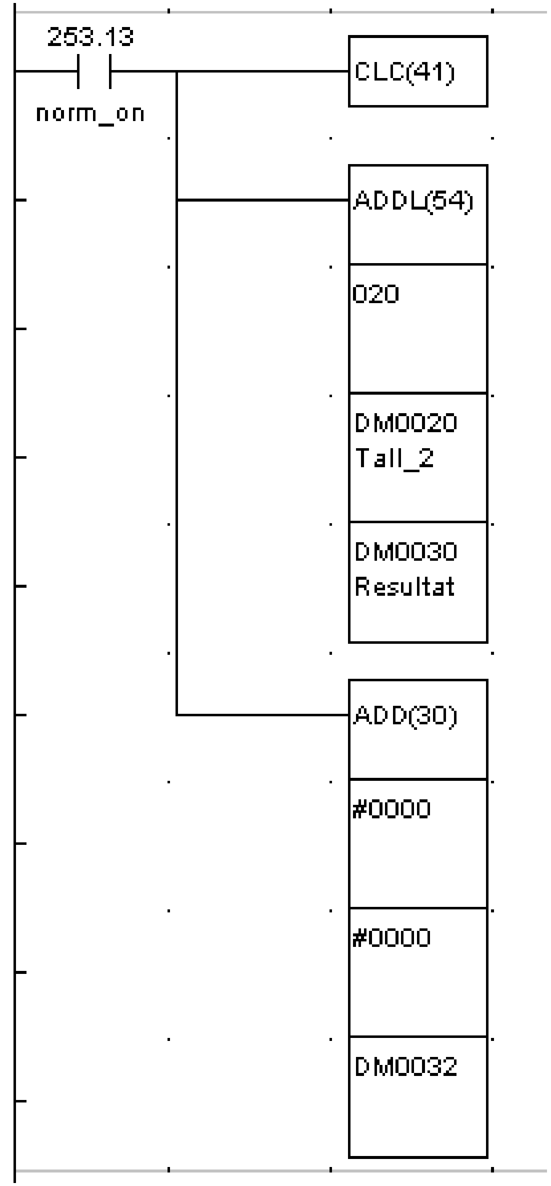
- Advantages of using IEC 61131-3
  - Usually provides more features than previous proprietary Programming Languages
  - Reduces training time when changing to new PLC vendor
  - Eases exchanging and porting programs between PLCs of distinct vendors

- Potential issues in using IEC 61131-3
  - Most vendors support a rather reduced subset of IEC 61131-3 concepts
  - The IEC 61131-3 standard leaves many details unspecified, resulting in differences between IEC 61131-3 implementations
    (may lead to strange bugs...)

Introduction to IEC 61131

Before the standard...

- All had LD as one of their possible programming languages
- Use of symbols and programming possibilities varied
  - Different ”dialects”
- Difficult to structure programs
  - Limited amount of sub routines
  - Lacking possibility of user defined functions
- Weak with regard to arithmetic operations
  - Blocks for everything (ADD, SUB, etc.)
- Difficult to re-use code
- Limited possibilities to control the execution order



Introduction to IEC 61131

The standard and the improvements

- Based on existing systems and languages from the biggest PLC manufacturers
  - Summing of the different "dialects”
- The standard covers more than just languages:
  - Addressing
  - Execution rules
  - Data types
  - Use of symbols (identifiers, keyword, etc.)
  - Connection between the languages

Introduction to IEC 61131

The standard and the improvements

- Allows development of structured code
  - => better quality, fewer software faults
  - => Encapsulation of program units
  - => Possible hierarchical structure

- Existence of typed variables, and strong type
  - => better quality, fewer software faults

- Support for complex data structures
- Provides several languages to choose from
- Allows mixing of languages in same program
  - => We can choose the best language for the job at hand

Introduction to IEC 61131

Implementation of the standard

- Not an absolute standard
  - It defines guidelines summed up in 62 tables
  - The manufacturer decide in which extent they wish to follow the standard, and must make documentation with reference to the table items
- 3 levels of certification
  - Basis: Fundamental elements in the languages
  - Portability: deals with compatibility
  - Full level
- The certification is issued by PLCopen

Introduction to IEC 61131

- PLCopen (www.plcopen.org)

  "is a vendor- and product-independent worldwide association"
- Has approved several accompanying 'standards':
  - TC 2 – Motion Control
    - Part 1 - Function Blocks for Motion Control
    - Part 2 - Extensions
    - Part 3 - User Guidelines and Examples
    - Part 4 - Coordinated Motion
    - Part 5 - Homing Procedures
    - Part 6 - Extensions for Fluid Power
  - TC 4 – Communication
    - "OPC UA Information Model for IEC 61131-3", version 1.00
  - TC 5 – Safety
    - Safety Specification Part 1 - Concepts and Function Blocks
    - Safety Specification Part 2 - User Guidelines
  - TC 6 – XML
    - PLCopen XML schema

Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

IEC 61131-3 Standard Architecture

- What exactly is defined in the standard?

Data Types	**Programming Languages**	**POUs (Program Organization Units)**
Bool Int Real Time Date Time_of_Day Date_and_Time String Byte Word ...	LD FBD IL ST SFC	Function Function Block Program Configuration

Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- **IEC 61131-3 Standard Architecture**
- Programming Languages
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

Programming Languages

LD

FBD

IL

ST

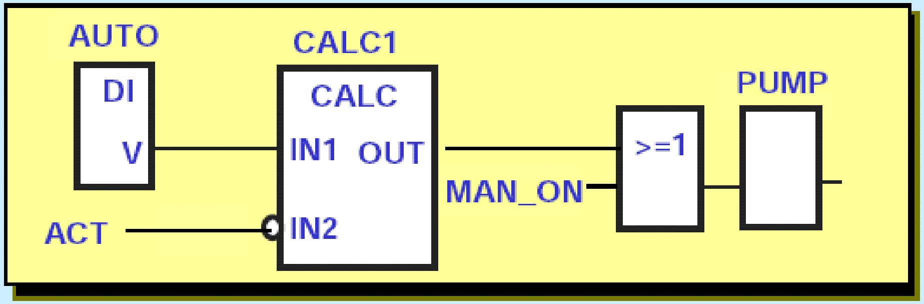
SFC

IEC 61131-3 Standard Architecture

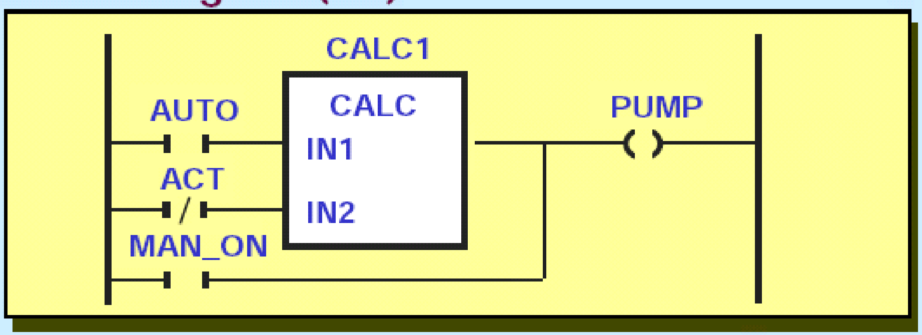
- Programming Languages

Graphical Languages

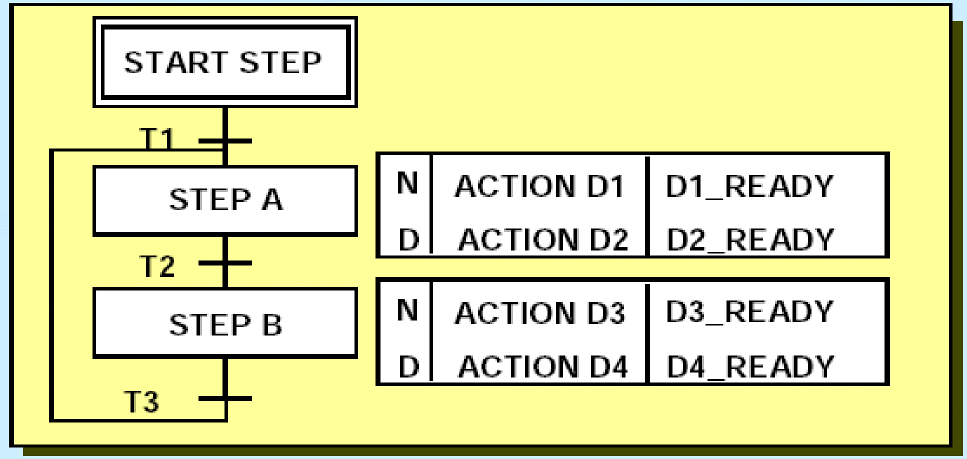
Function Block Diagram (FBD)



Ladder Diagram (LD)



Sequential Flow Chart (SFC)



Textual Languages

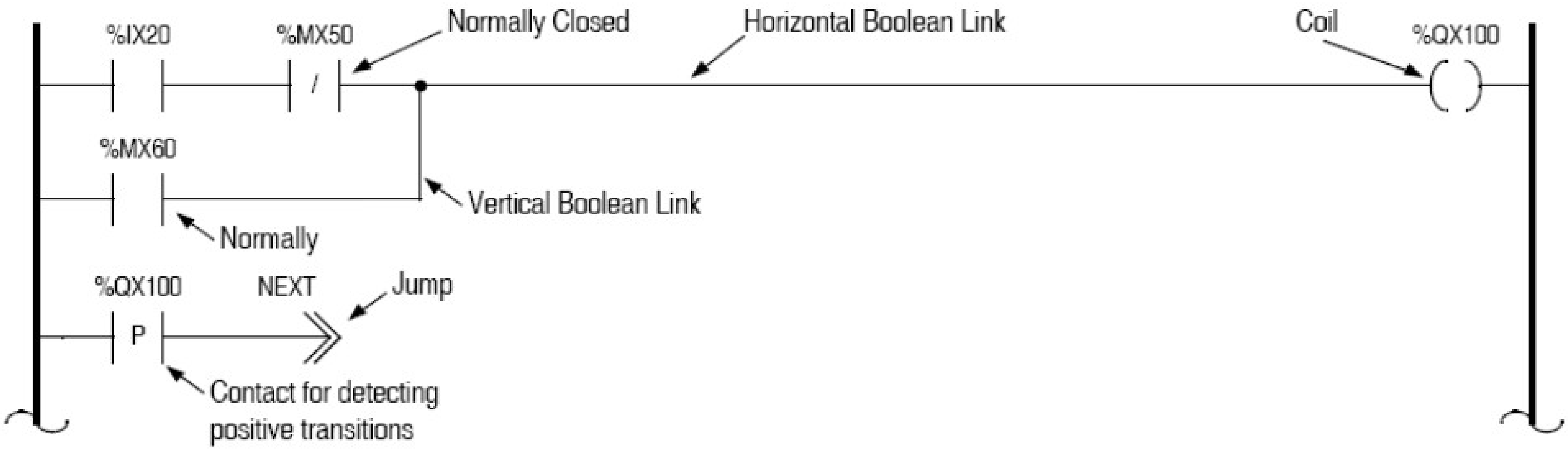
Instruction List (IL)

```
A:  LD    %IX1  (* PUSH BUTTON *)
    ANDN  %MX5  (* NOT INHIBITED *)
    ST    %QX2  (* FAN ON *)
```

Structured Text (ST)

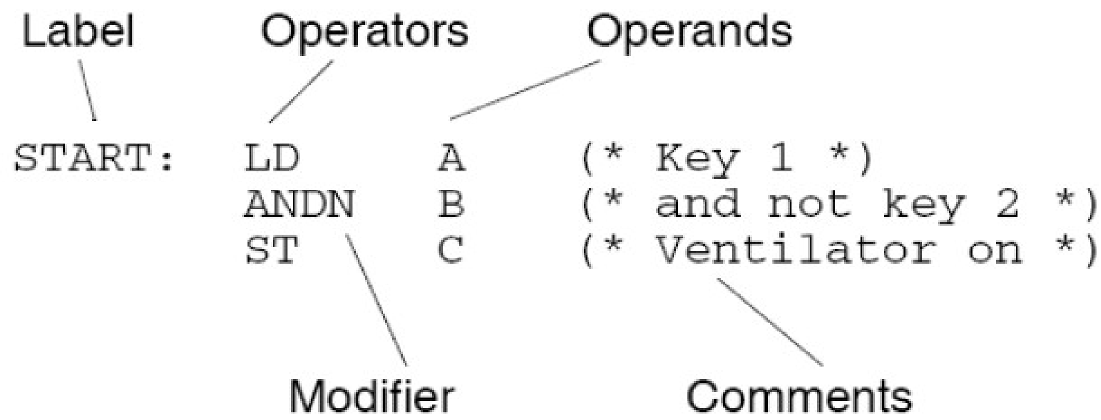
```
VAR CONSTANT X : REAL := 53.8 ;
Z : REAL; END_VAR
VAR aFB, bFB :  FB_type; END_VAR

bFB(A:=1, B:='OK');
Z := X - INT_TO_REAL (bFB.OUT1);
IF Z>57.0 THEN aFB(A:=0, B:="ERR");
ELSE  aFB(A:=1, B:="Z is OK");
END_IF
```

LD – Ladder Diagram



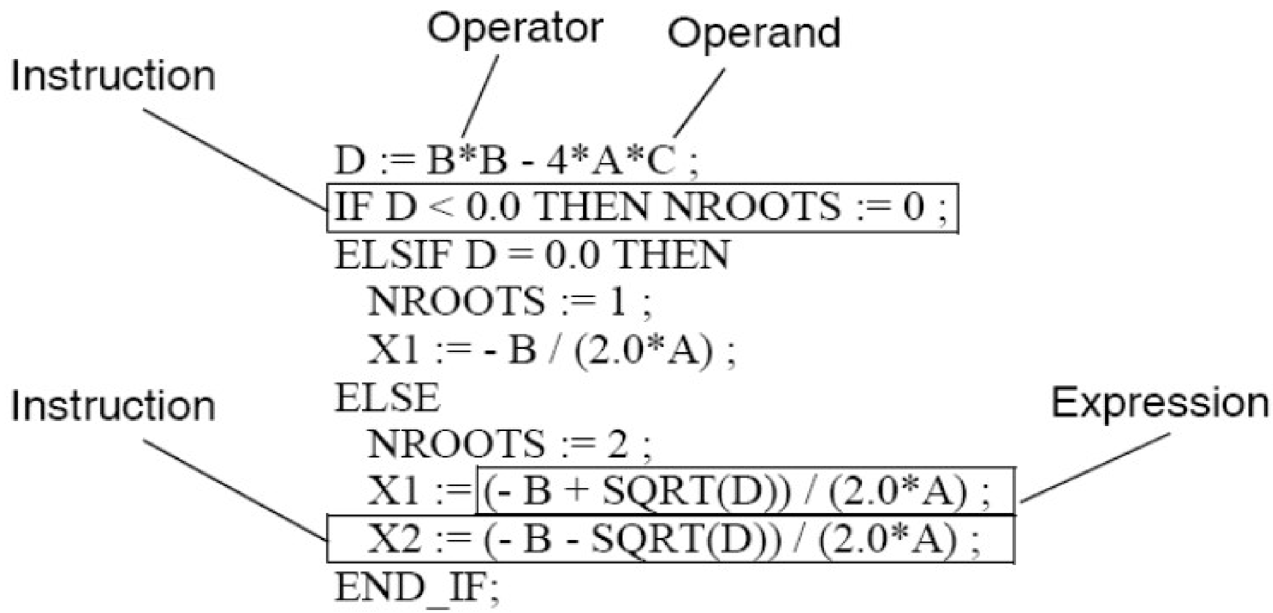
- Graphical language, low level, similar to designing a relay based electric circuit.
- Included in standard because:
  - Simple language that can be used by electricians with no formal training in programming
  - Extensively used in industry
- Difficulty in programming complex control sequences (e.g.: FOR, WHILE, REPEAT, ...)
- Usually very efficient (fast execution and low memory usage)
- Very difficult to analyse (read and interpret) complex/long programs.

IL – Instruction List



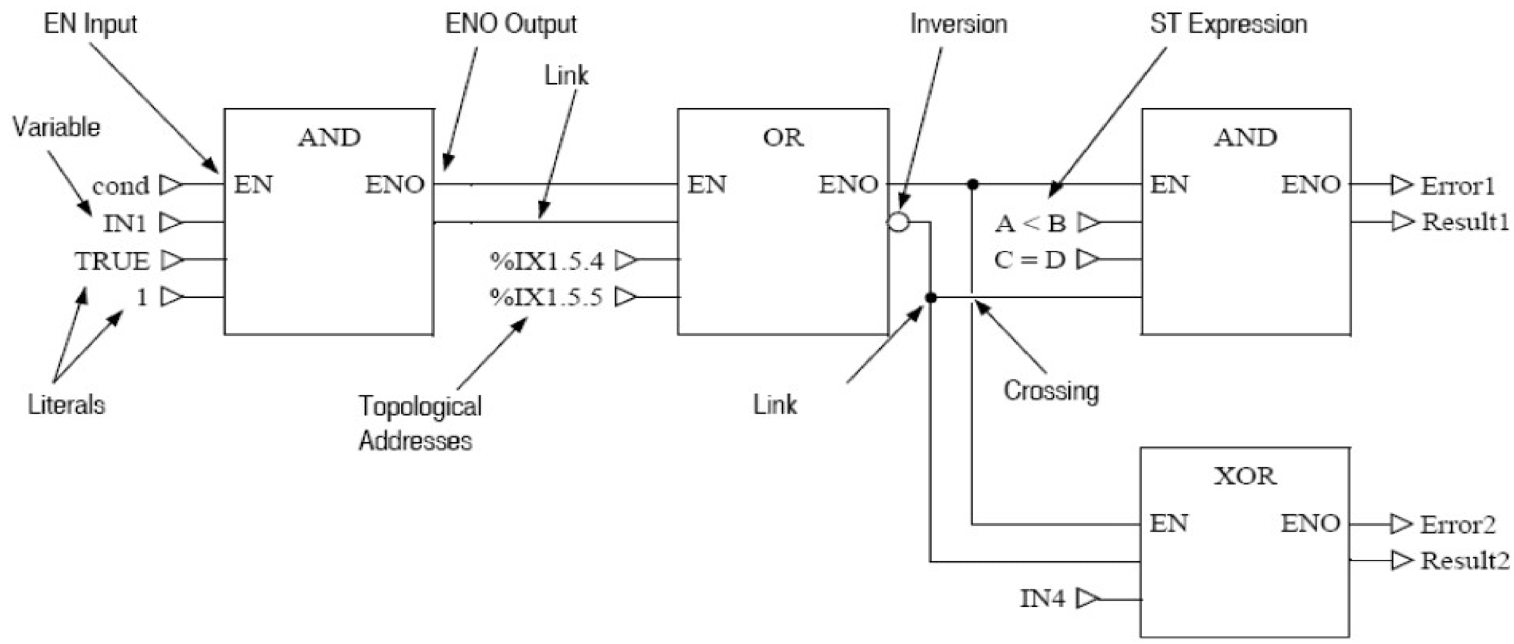
- Low level textual programming language, similar to Assembly programming of a micro-controller.
- Included in the standard mostly for historical reasons
- Difficulty in programming complex control sequences (e.g.: FOR, WHILE, REPEAT, ...)
- Usually very efficient (fast execution and low memory usage)
- Very difficult to analyse (read and interpret) complex/long programs.

ST – Structured Text



- High level programming language, with syntax similar to Pascal
- Supports complex control flow instructions (FOR, WHILE, REPEAT, IT, CASE, etc...)
- Easy to build complex mathemtical/boolean expressions
- Execution speed and memory use typically slower than LD or IL, but faster than SFC

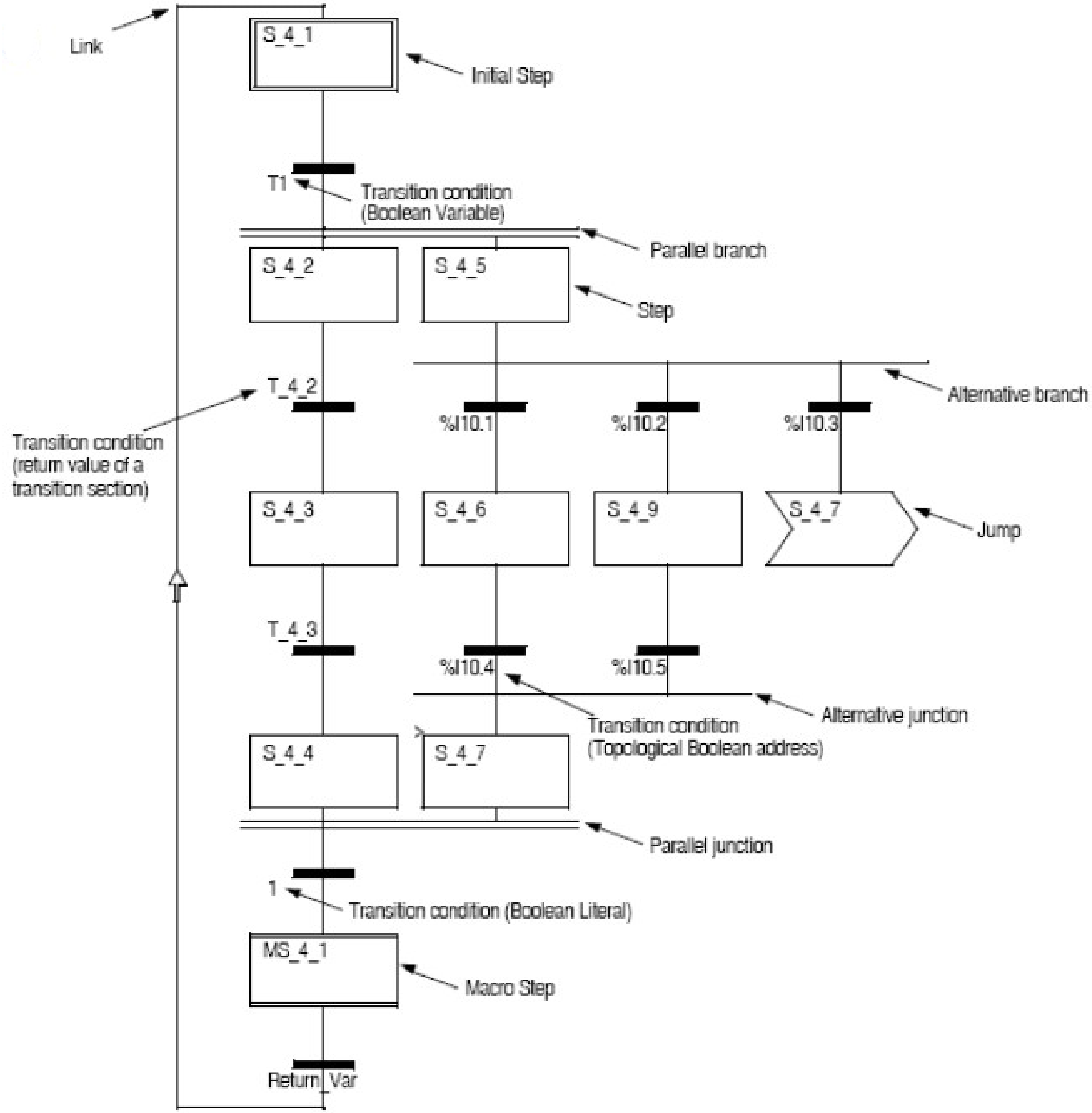
FBD – Function Block Diagram



- High level graphical programming language, based on the signal flows.
  - Incorporates concepts based on object oriented programming
  - Each 'block' represents a block of code, or a block of code and associated data
- Very often used in the process industry
- For simple programs, these become very easy to interpret and understand
- May bring issues when the correct execution of the program depends on a specific sequence of execution of each of the 'blocks'

SFC – Sequential Function Chart

- High level graphical programming language
- Based on the GRAFCET standard
  - Fixes some (but not all) of the ambiguities of GRAFCET
- Used to describe operations to be executed sequenctially, in parallel, and / or concurrently.
- May be used as a way of structuring code
- Typically results in slower execution times, and more memory use.
- Ideal for implementing state machines for discrete control systems.



Overview of IEC 61131-3

- **Introduction to IEC 61131 and Related Standards**
- **IEC 61131-3 Standard Architecture**
- **Programming Languages**
- **IEC 61131-3 Execution Model**
- **Standard Functions and Function Blocks**
- **Examples**

Data Types

Bool
Int
Real
Time
Date
Time_of_Day
Date_and_Time
String
Byte
Word
...

IEC 61131-3 Standard Architecture

- Data Types - Integers

Type	Coment	Default Value	Size (Bits)	Limits
SINT	Short integer	0	8	-128 a 127
INT	Integer	0	16	-32768 a 32767
DINT	Double Integer	0	32	-2 147 483 648 a 2 147 483 647
LINT	Long Integer	0	64	...
USINT	Unsigned Short integer	0	8	0 a 255
UINT	Unsigned Integer	0	16	0 a 65535
UDINT	Unsigned Double Integer	0	32	0 a 4 294 967 295
ULINT	Unsigned Long Integer	0	64	...

IEC 61131-3 Standard Architecture

- Data Types - Reals

Type	Coment	Default Value	Nº Bits	Limits
REAL	Real	0	32	
LREAL	Long Real	0	64	

IEC 61131-3 Standard Architecture

- Data Types – Bit Oriented

Type	Coment	Default Value	Nº Bits	Limits
BOOL	Boolean Logic Value	FALSE	1	FALSE a TRUE
BYTE	Sequence of 8 bits	0	8	0 a 255
WORD	Sequence of 16 bits	0	16	0 a 65535
DWORD	Sequence of 32 bits	0	32	0 a 4 294 967 295
LWORD	Sequence of 64 bits	0	64	0 a ...

Arithmetic operations with these data types is NOT allowed/supported!

Only bit oriented boolean operations are allowed/supported!

IEC 61131-3 Standard Architecture

- Data Types - Strings

Type	Coment	Default Value	Bits Per Char.	Constants of this Data Type
STRING	Sequence of characters. Variable length.	''	8	'Hello World!'
WSTRING	Sequence of characters. Variable length.	“”	16	“Hello World!”

IEC 61131-3 Standard Architecture

- Data Types – Date and Time

Type	Coment	Default Value	Constants of this Data Type
TIME	Time Interval	T#0s	T#1.56d T#2d_5h_23m_5s_4.6ms
DATE	A Date	D#0001-01-01	D#1968-12-11
TOD	Time of Day	TOD#00:00:00	TOD#14:32:34.5
Time_of_Day	Time of Day	TOD#00:00:00	
DT	Date and Time of Day	DT#0001-01-01-00:00:00	DT#1968-12-11-14:32:34.5
Date_and_Time	Date and Time of Day	DT#0001-01-01-00:00:00	

IEC 61131-3 Standard Architecture

- Data Type Hierarchy

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

[Taken from IEC61131-3]

IEC 61131-3 Standard Architecture

- Derived Data Types
  - New user defined data types are supported...
  - Based on an elementary data type, but with new default value:

    ```
    TYPE FREQ: REAL := 50.0; END_TYPE
    ```
  - By enumeration of possible values:

    ```
    TYPE ANALOG_SIGNAL_T: (SINGLE_ENDED,DIFFERENTIAL); END_TYPE
    ```
  - By defining a sub-range of values:

    ```
    TYPE ANALOG_DATA: INT (-4095..4095); END_TYPE
    ```
    - A variable of a sub-range data type may be used anywhere a variable of the base type would be considered correct.
  - By defining an array of variables:

    ```
    TYPE ANALOG_16_INPUT_DATA: ARRAY [1..16] OF ANALOG_DATA; END_TYPE
    ```
  - ...

Note: Examples copied off standard

IEC 61131-3 Standard Architecture

- Derived Data Types (continued)
  - ...
  - By defining a data structure:

```
TYPE
  ANALOG_CHANNEL_CONFIGURATION: STRUCT
    RANGE : ANALOG_SIGNAL_RANGE;
    MIN_SCALE : ANALOG_DATA;
    MAX_SCALE : ANALOG_DATA;
  END_STRUCT ;

  ANALOG_16_INPUT_CONFIGURATION: STRUCT
    SIGNAL_TYPE : ANALOG_SIGNAL_TYPE;
    FILTER_PARAMETER : SINT (0..99);
    CHANNEL : ARRAY [1..16] OF ANALOG_CHANNEL_CONFIGURATION;
  END_STRUCT;
END_TYPE
```

Note: Examples copied off standard

Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

POUs (Program Organization Units)

Function

Function Block

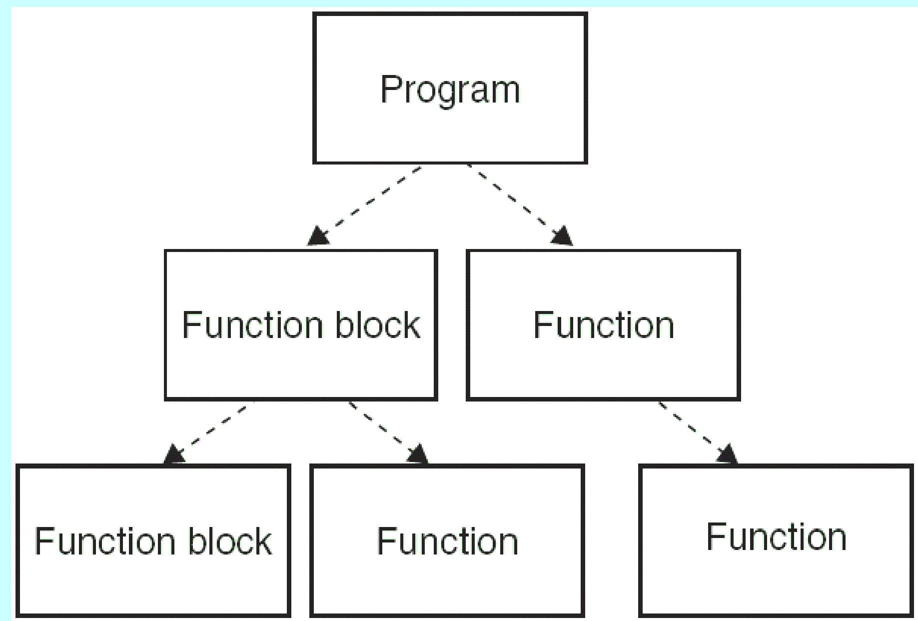
Program

Configuration

IEC 61131-3 Standard Architecture

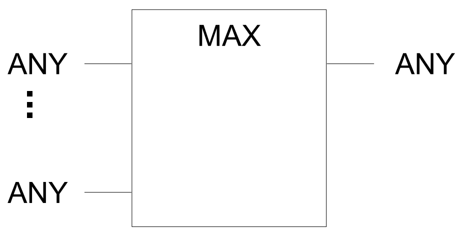
- POU - Program Organization Units
  - The code may organized in blocks, allowing for code re-use, structured code, etc...
  - Supported POUs:
    - PROGRAM
    - FUNCTION
    - FUNCTION BLOCK

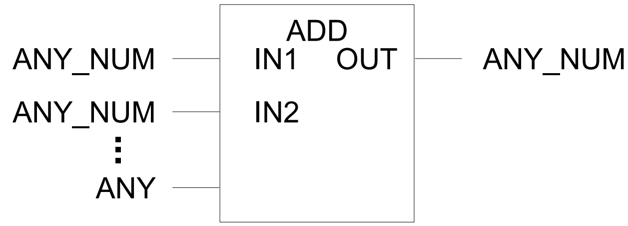
Relationship between POUs



IEC 61131-3 Standard Architecture

- FUNCTION
  - Block of code with input and output variables
  - Process input data → produce output data
  - Output data only depends on input data (idem-potent)

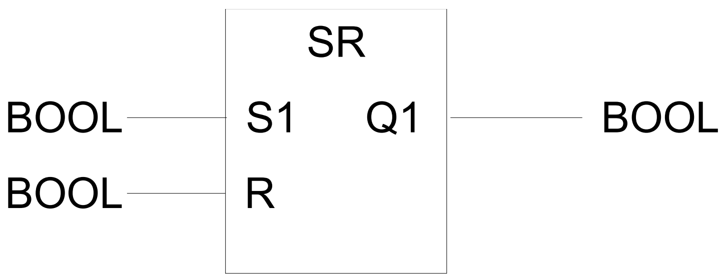




U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

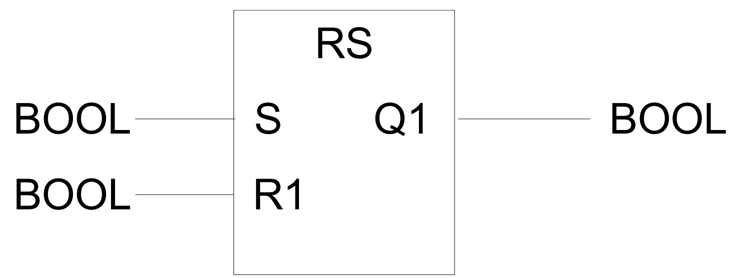
IEC 61131-3 Standard Architecture

- FUNCTION_BLOCK
  - Block of code with input, output, and internal variables
  - Internal variables store state between consecutive invocations
  - Process input + internal data → produce output + internal data
  - Calling with the same arguments can give different results
  - Instances of function blocks has to be explicitly declared, and there can be several instances of the same FB in a program (e.g. several Timers).



Set dominant RS Flip-Flop

Q1 := S1 OR (R AND NOT Q1)



Reset dominant RS Flip-Flop

Q1 := NOT R1 AND (S OR Q1)

Mário de Sousa (msousa@fe.up.pt), 2011

IEC 61131-3 Standard Architecture
Programs

- A Program is the highest level POU, somewhat similar to the main() in C programs.
- A Program may not be called by any other code, nor call itself recursively!
- May be written in any of the programming languages: LD, IL, FBD, ST, SFC

```
PROGRAM Flicker
  VAR
    light : BOOL;
  END_VAR

  (* Program Body in ST *)
  light := NOT (light);

END_PROGRAM
```

IEC 61131-3 Execution Model
The CONFIGURATION POU

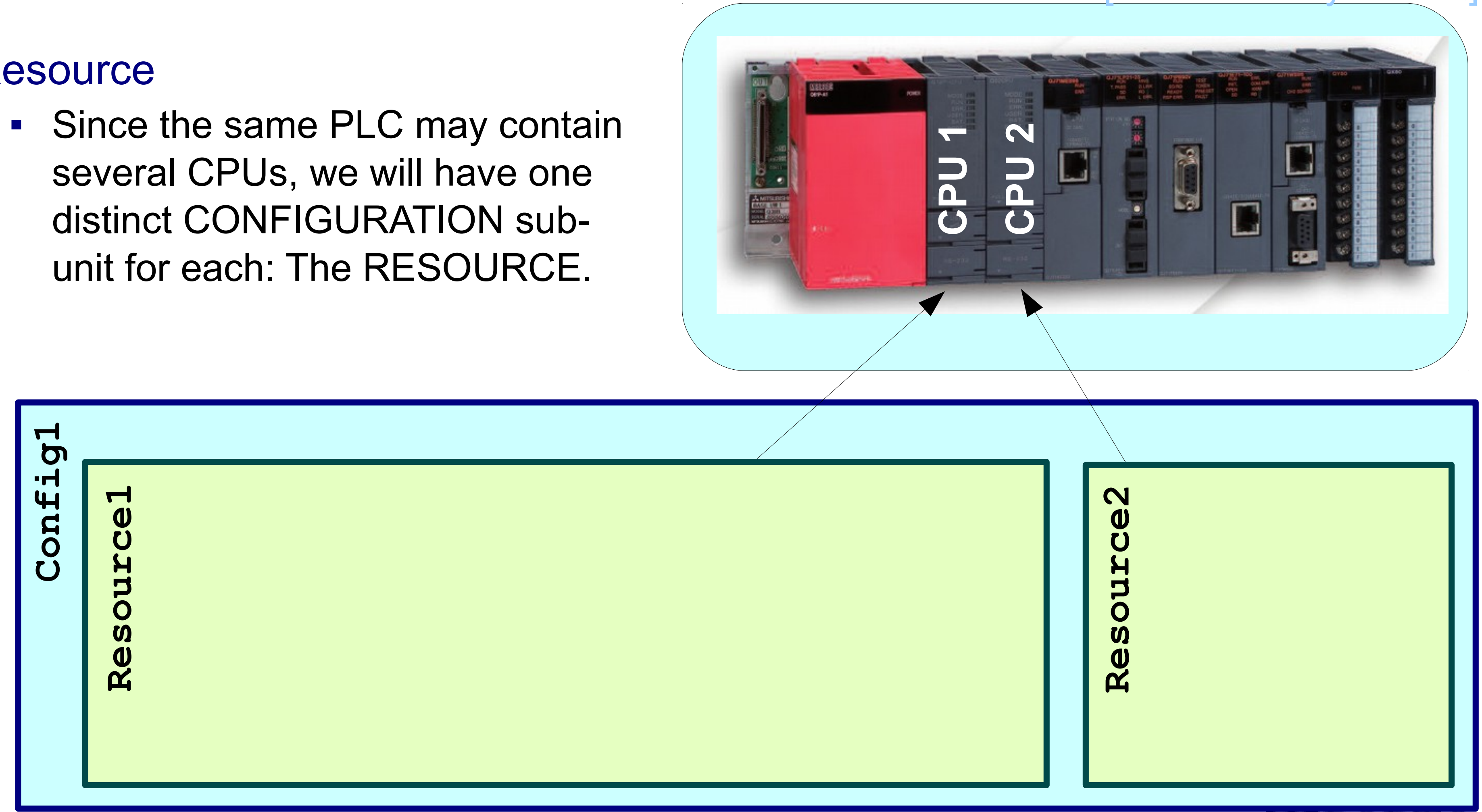
- Configuration
  - A configuration brings together all the information necessary to define the exact desired state of the software to download to the PLC.
  - Includes:
    - Variable definitions
    - Program Instances
    - Scheduling of program execution
  - Does NOT include
    - Specific hardware configuration!

[Mitsubishi-System Q]

IEC 61131-3 Execution Model
The CONFIGURATION POU

[Mitsubishi-System Q]

- Resource
  - Since the same PLC may contain several CPUs, we will have one distinct CONFIGURATION sub-unit for each: The RESOURCE.

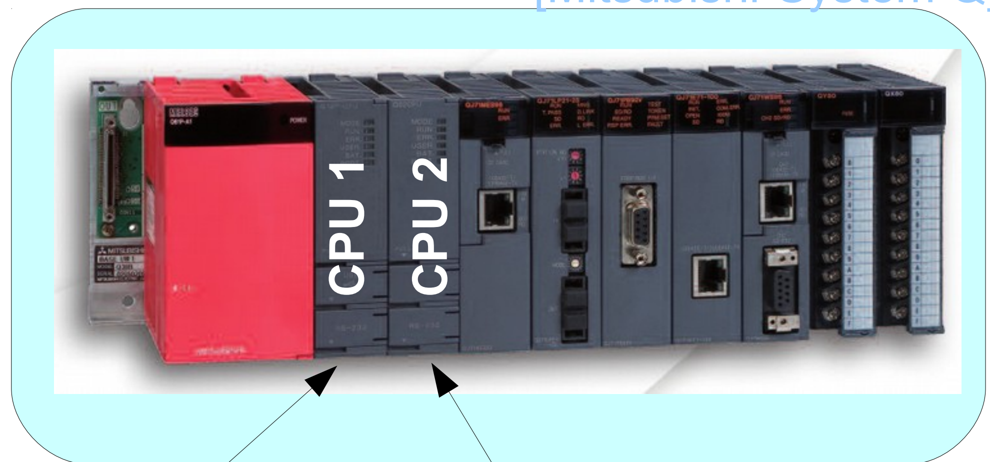


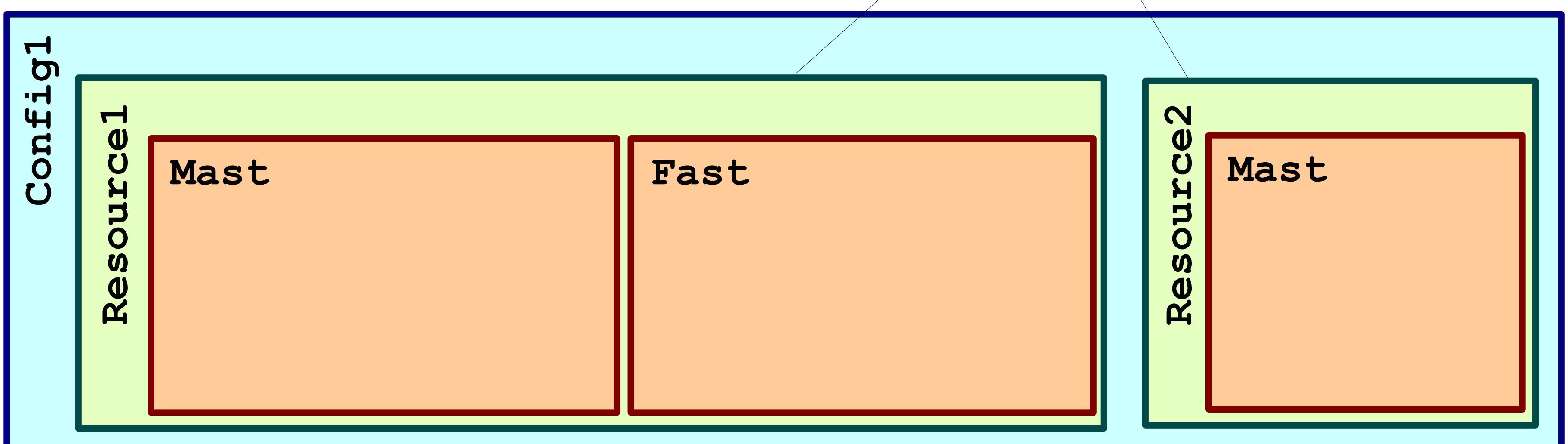
U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

IEC 61131-3 Execution Model
The CONFIGURATION POU

- Tasks
  - Each Resource may have one or more TASKS, which are the entities that are going to be directly executed by the PLC's Operating System.

[Mitsubishi-System Q]

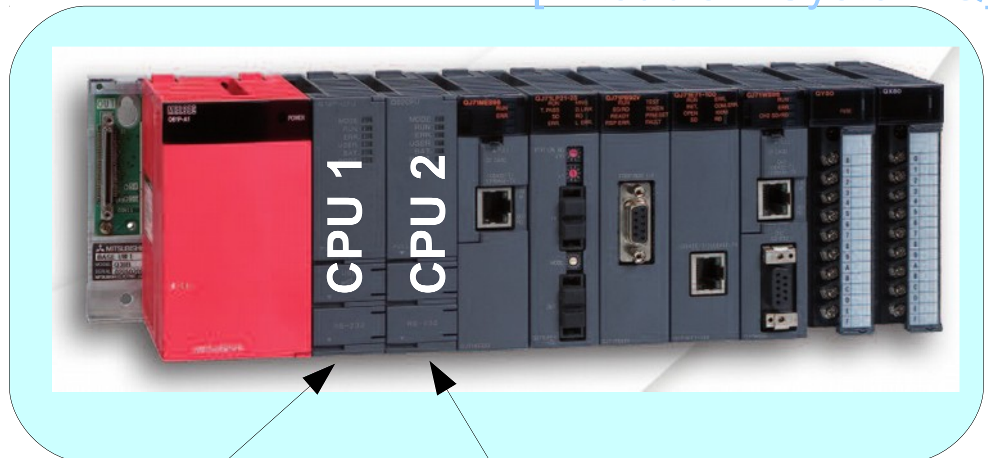


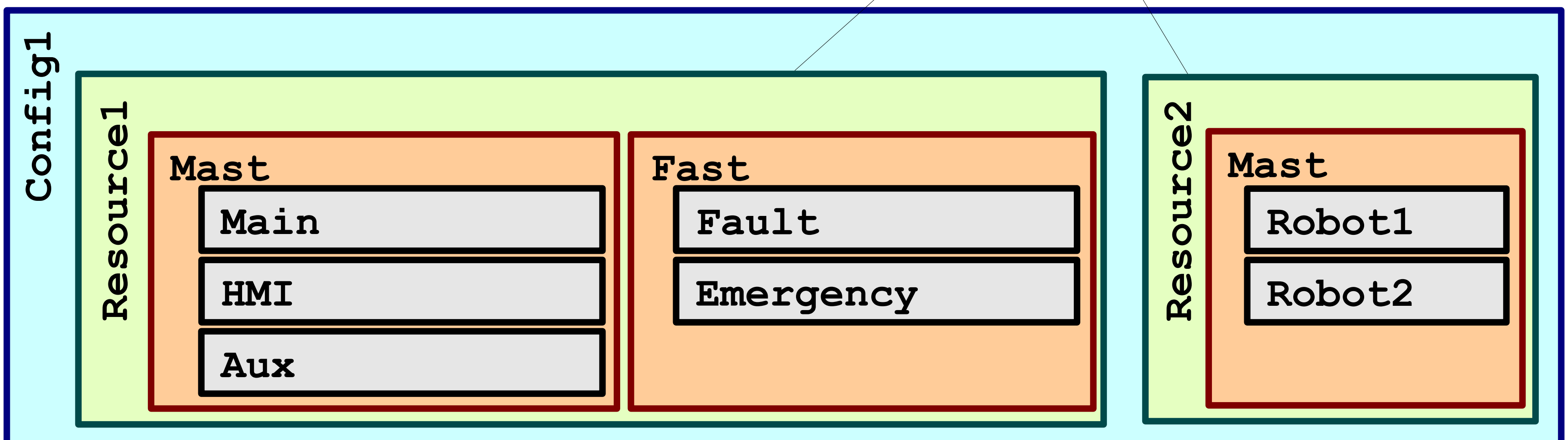


IEC 61131-3 Execution Model
The CONFIGURATION POU

- Programs
  - We then configure which PROGRAM instances will be executed by each task. (PROGRAMS are like FBs, we must create instances...)
  - The programs will in turn call Functions and Function Blocks.

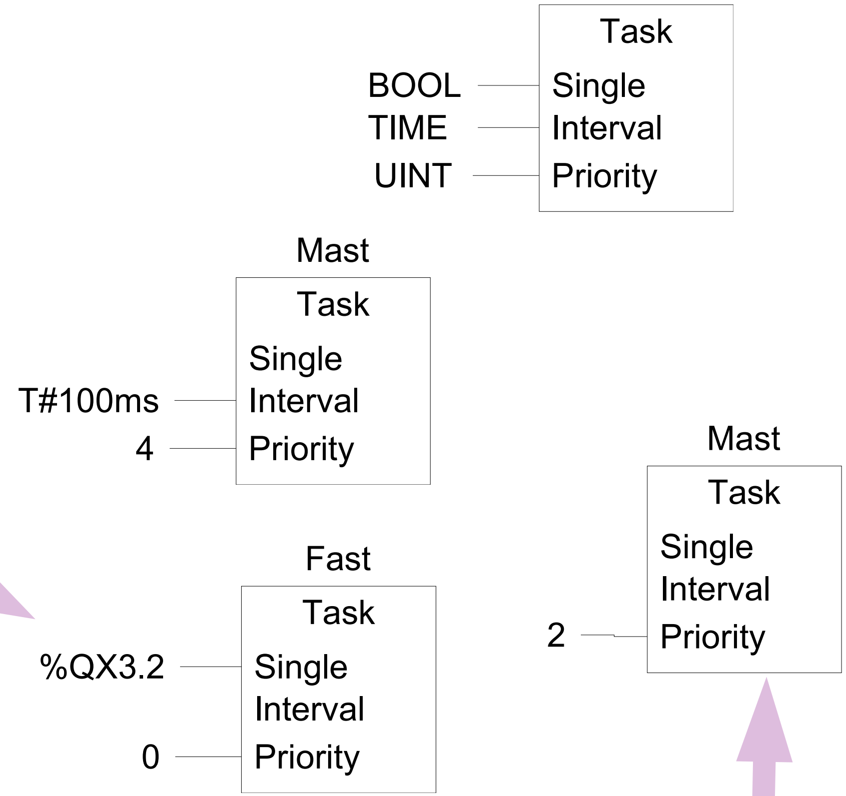
[Mitsubishi-System Q]





IEC 61131-3 Execution Model
Scheduling of Tasks

- Task configuration
  - A task's activation for execution may be configured to be triggered:
    - Periodically (time triggered)
    - By the occurrence of an event – i.e. rising edge of boolean value (event triggered)
  - Default (unnamed task): Cyclic. (i.e. continuous) execution
    - This task executes all Programs instances that have not been associated to a task
  - Since the CPU can only execute one task at a time, a Priority parameter **<u>must</u>** be provided to decide which task to execute (0 is highest priority).

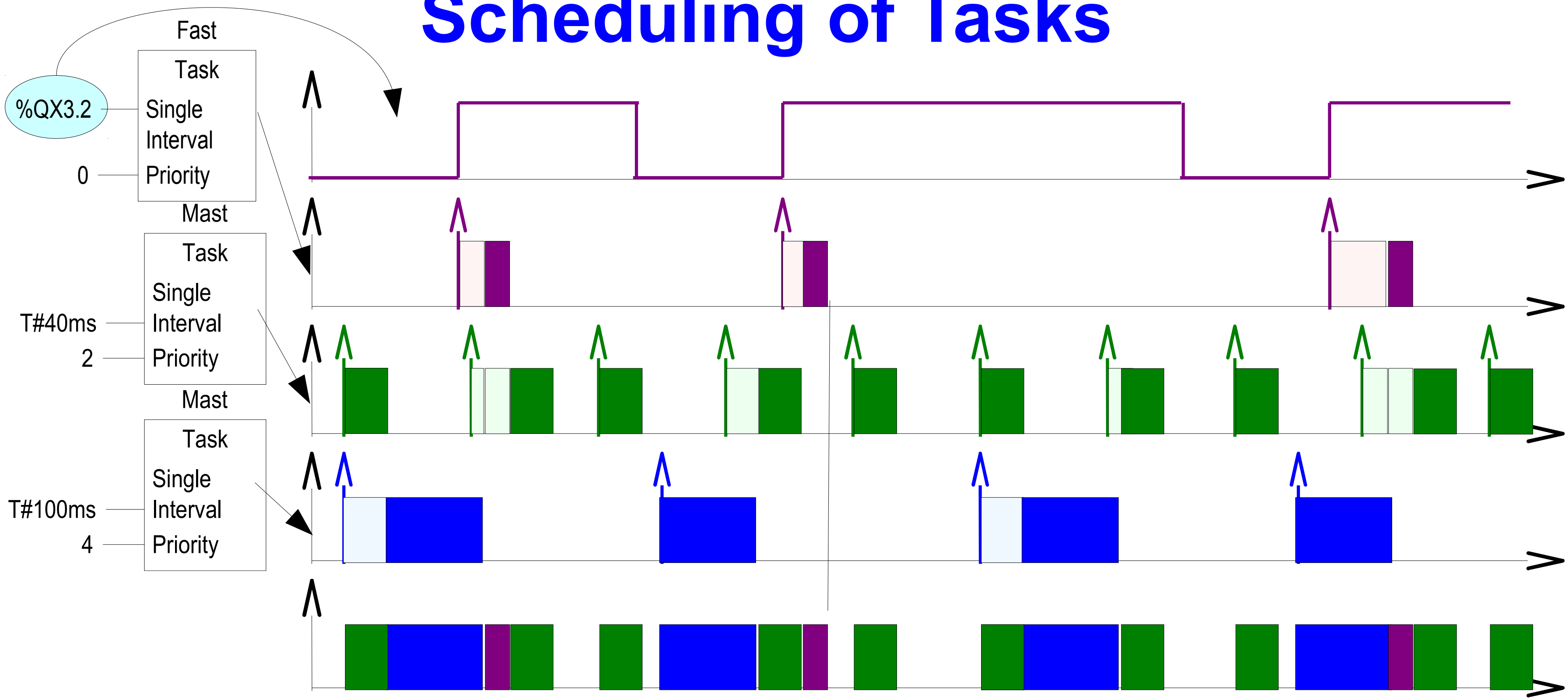


This task will never execute!

The standard allows for both pre-emptive and non pre-emptive scheduling (implementation dependent)

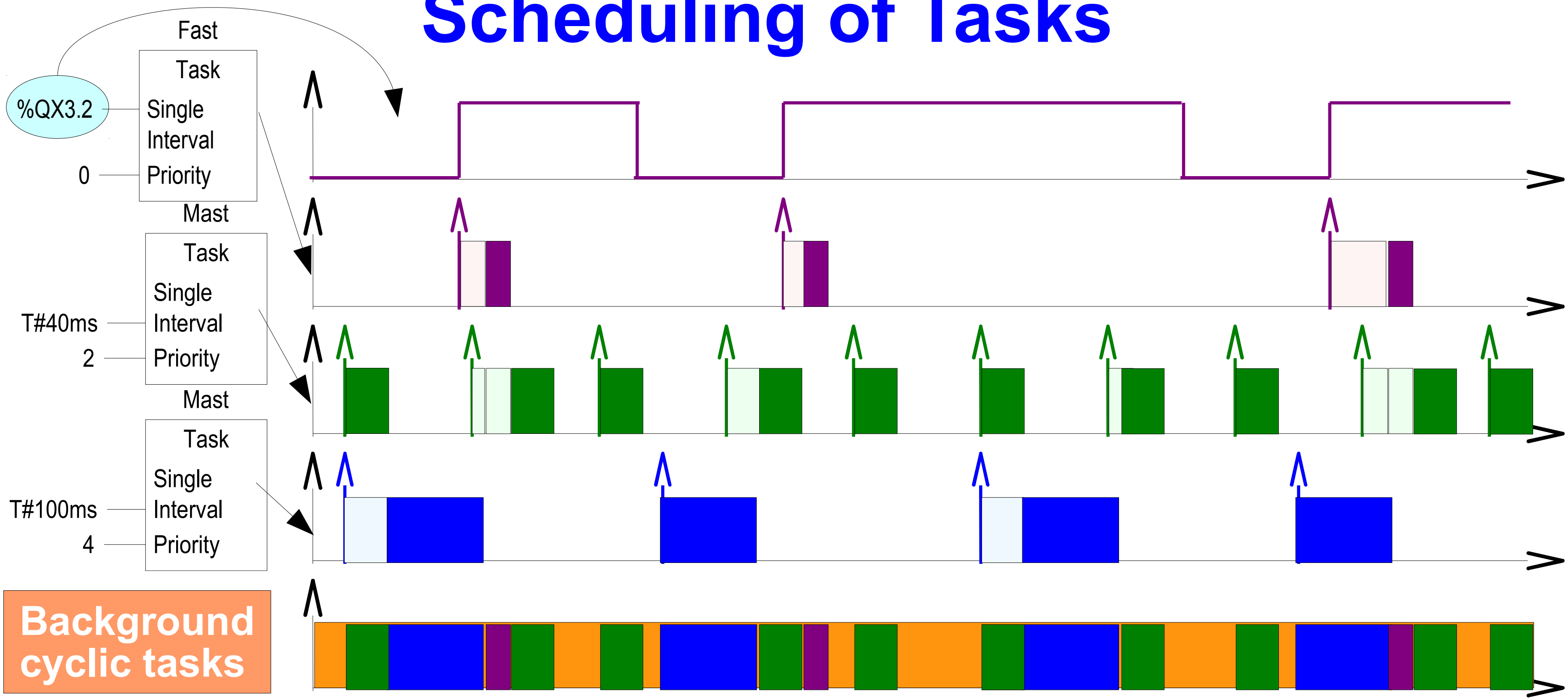
IEC 61131-3 Execution Model

Scheduling of Tasks



- Non Pre-emptive scheduling
  - Once a task starts executing, it will run until the end.
  - Once a task finishes, the higher priority task of all currently waiting tasks will run
  - Provides lower schedulability
  - Start of High priority tasks may get delayed for long time.

IEC 61131-3 Execution Model
Scheduling of Tasks

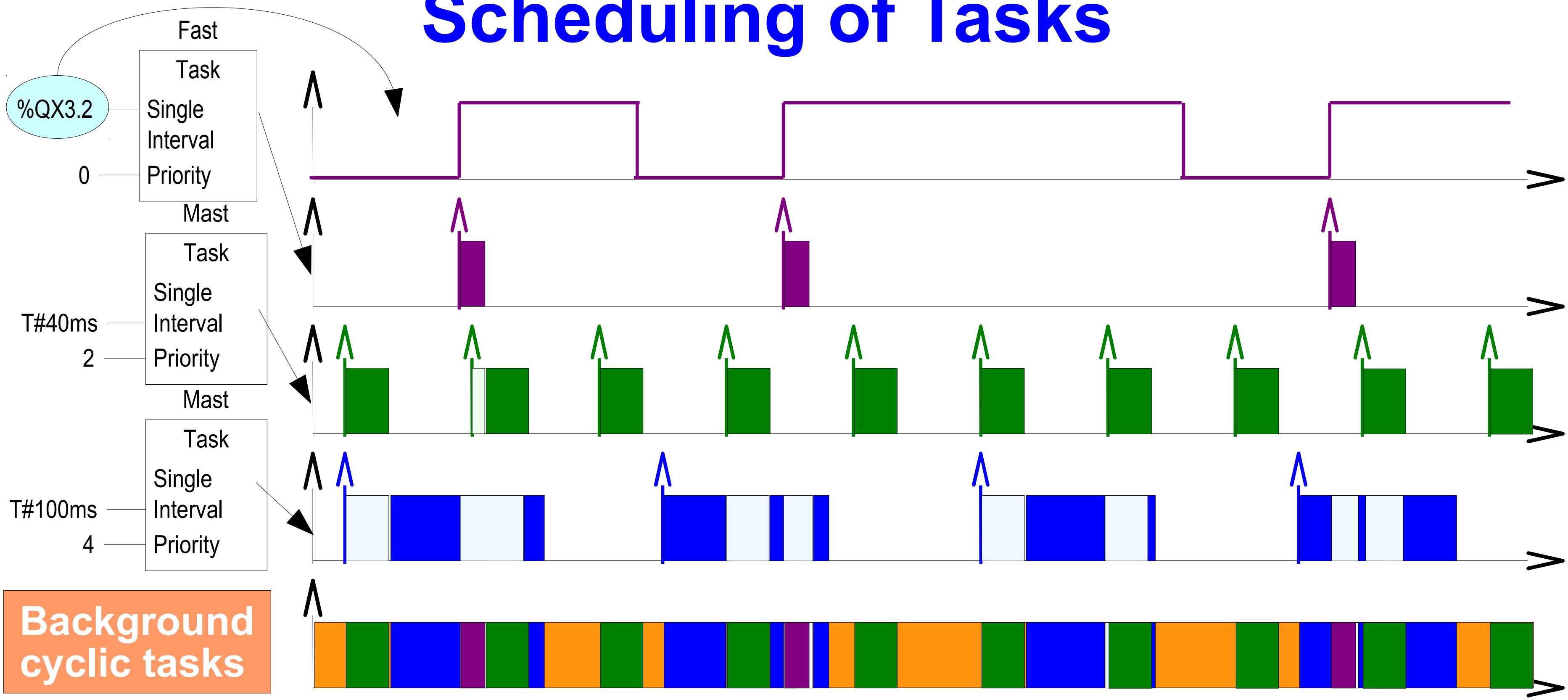


- Non Pre-emptive scheduling
  - Even in this case, background tasks are usually pre-empted!

If this were not the case, high priority tasks would get delayed even further!

Remember: background task is usually very long!

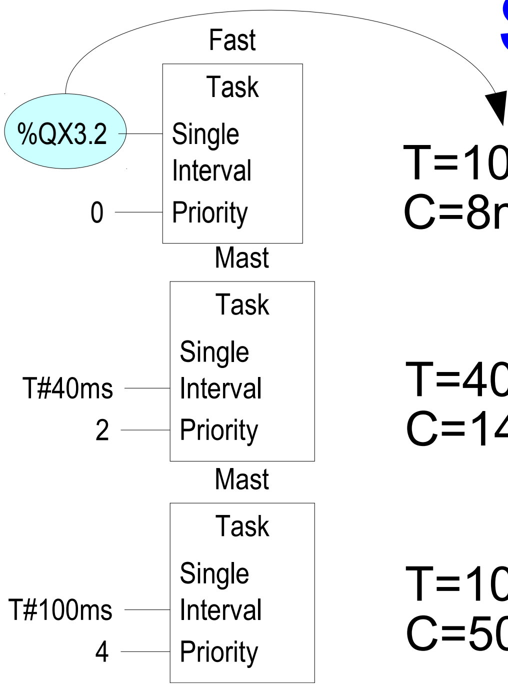
IEC 61131-3 Execution Model
Scheduling of Tasks



- Pre-emptive scheduling
  - Higher priority tasks interrupt lower priority tasks
  - Lower priority tasks that were executing will continue at a later time
  - May lead to many task switching
  - Provides higher schedulability

Mário de Sousa (msousa@fe.up.pt), 2011

IEC 61131-3 Execution Model
Scheduling of Tasks



T=100ms
C=8ms → U =8/100 =8%

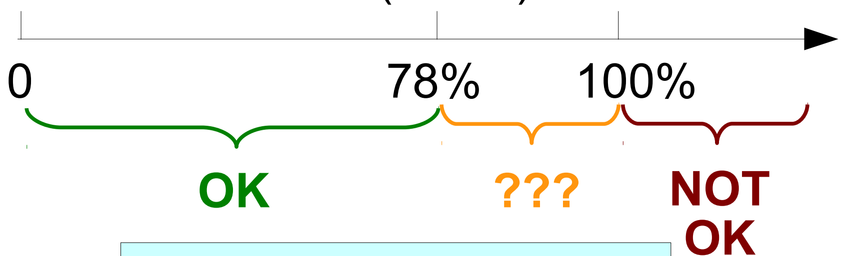
T=40ms
C=14ms → U =14/40 =35%

T=100ms
C=50ms → U =50/100 =50%

Liu & Leyland schedulability criteria (1973)

n → nº of tasks

$n(2^{1/n}-1)$



U_{total} = 50+35+8 = 93%

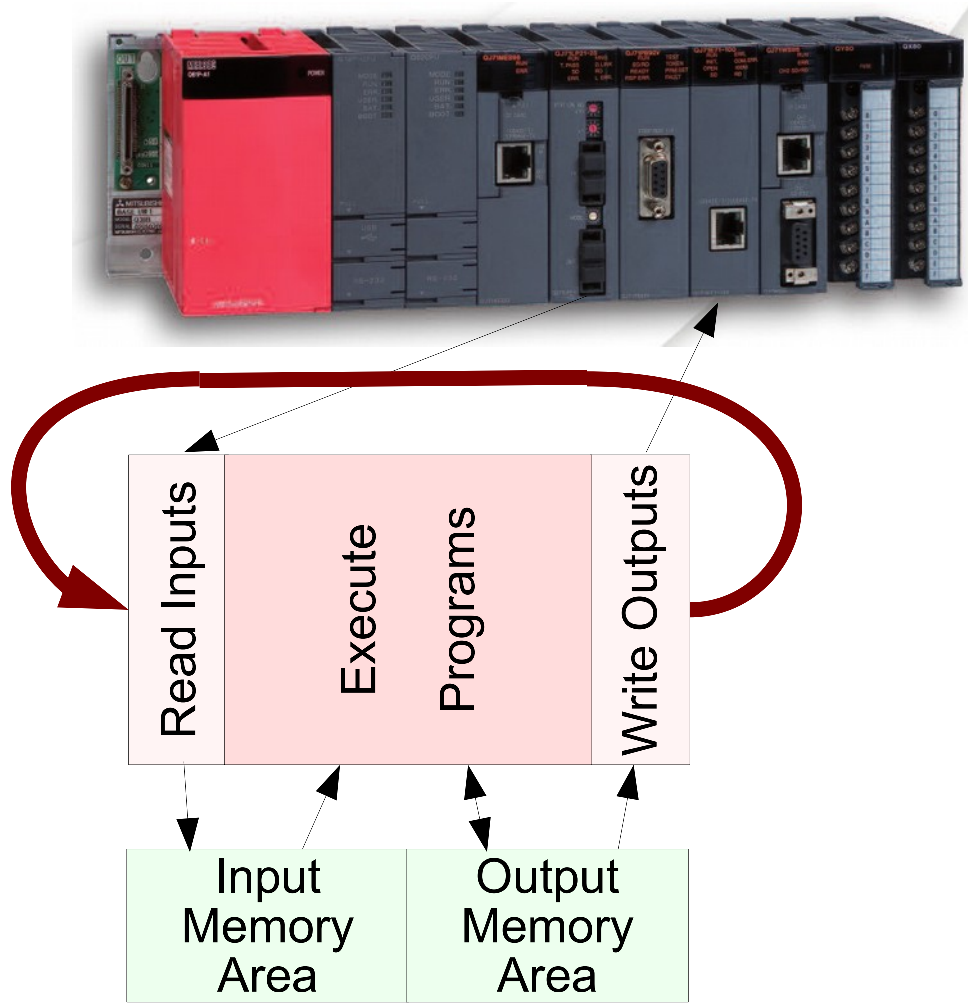
Schedulable???

- Pre-emptive Rate Monotonic Fixed Priority Scheduling
  - Tasks are given priorities in the same order as their periods (shorter periods → higher priorities)
  - We want all tasks to finish execution before being activated again (Deadline = Period)
  - For event triggered tasks we must consider the minimum inter-arrival time.

IEC 61131-3 Execution Model
Scheduling of Tasks

- A traditional PLC's execution model consider three main phases in the scan cycle
  - Read physical Inputs
  - Execute Program
  - Update Outputs

- IEC 61131-3 compliant PLCs may be running many tasks. When should they read the physical inputs, and write the outputs?



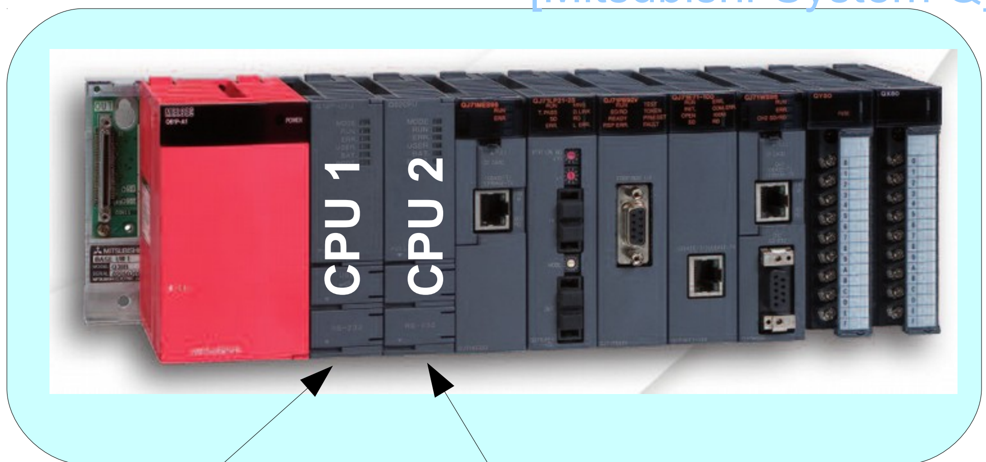
The standard doesn't say...

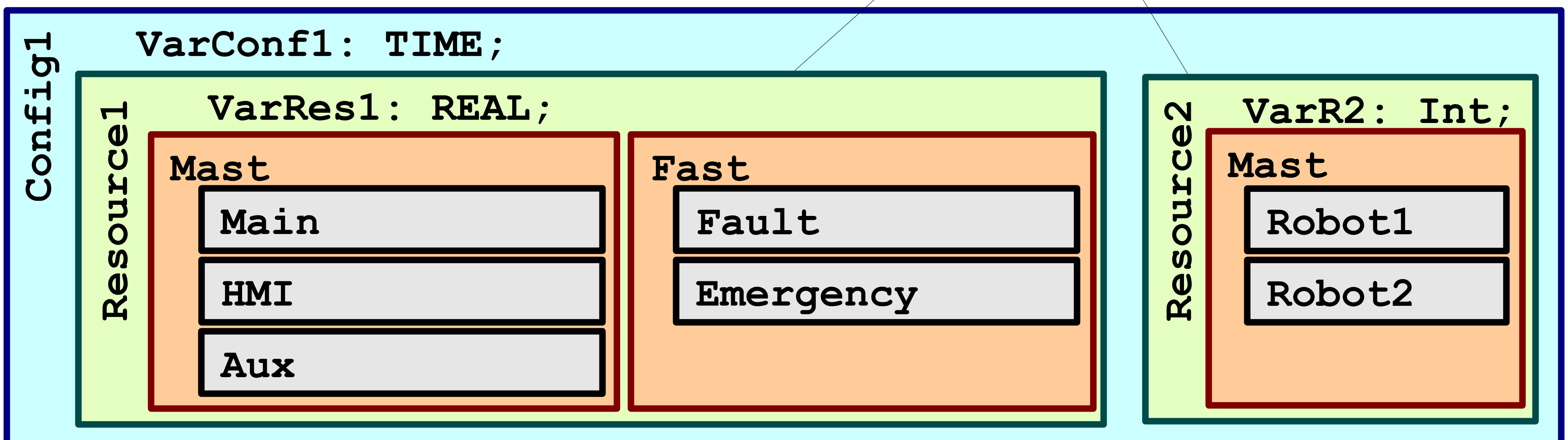
Anything is possible!

IEC 61131-3 Execution Model
Communication Model

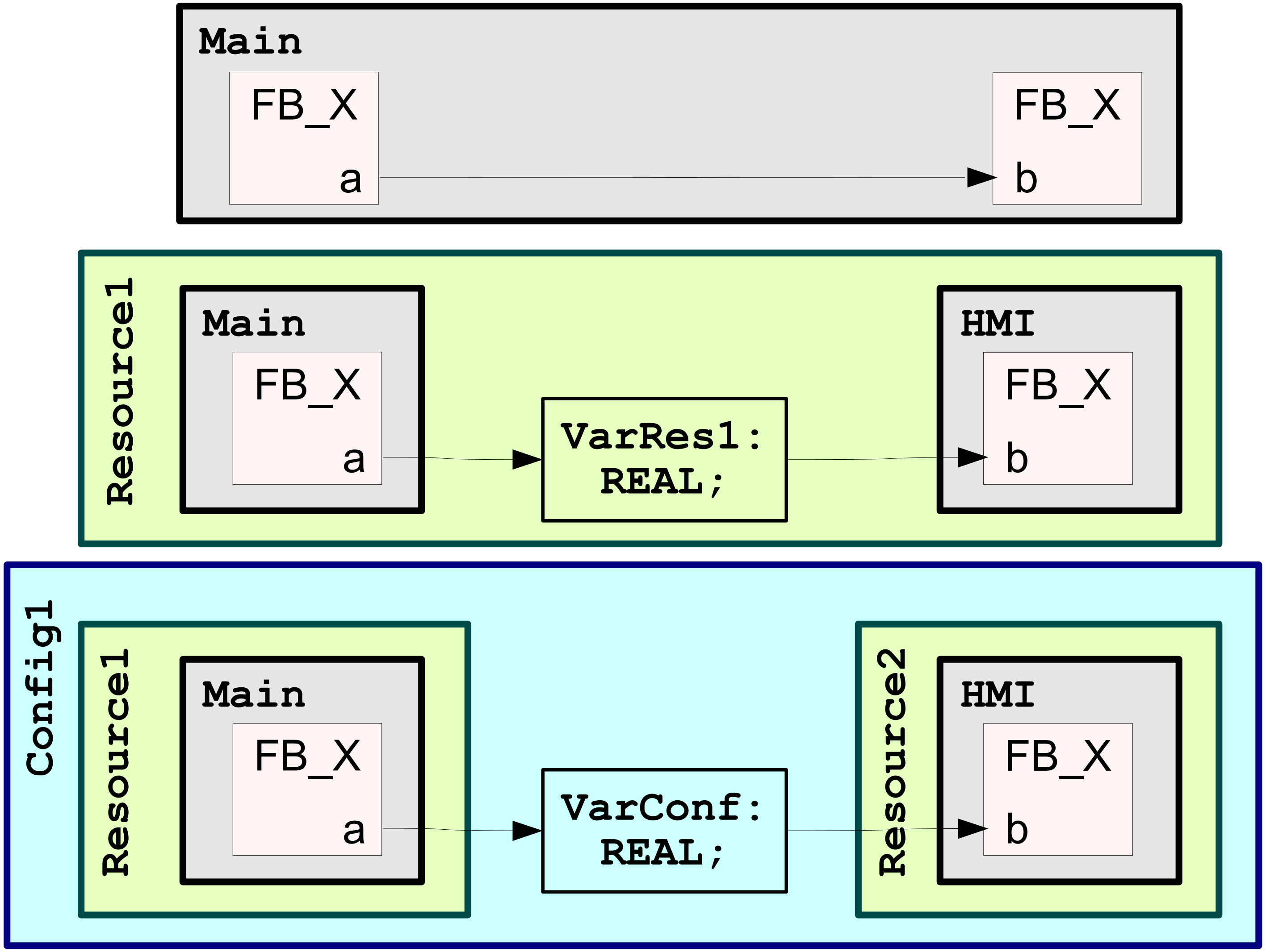
- Communication Model
  - Often, the programs need to exchange data so as to co-operate effectively.
  - This may be done using shared global variables.
    - In the configuration
    - In each resource

[Mitsubishi-System Q]





IEC 61131-3 Execution Model
Communication Model



Data Flow within a PROGRAM

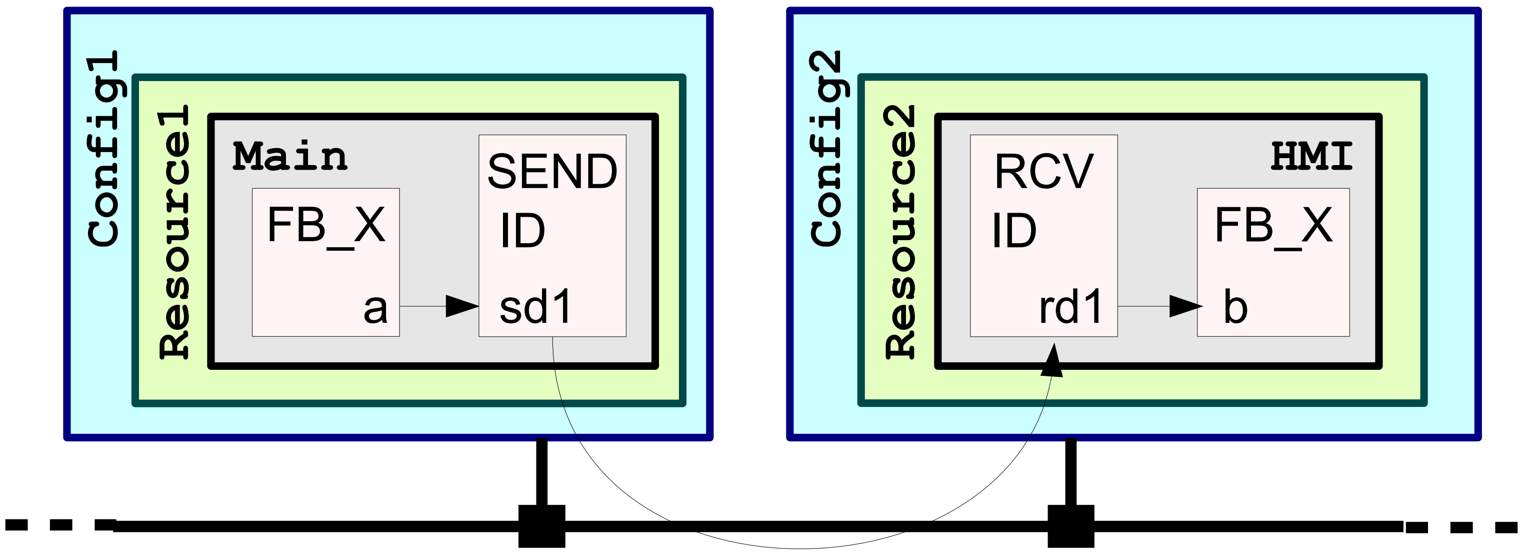
Data Flow between two PROGRAMs in the same RESOURCE

Data Flow between two PROGRAMs in the same CONFIGURATION

Mário de Sousa (msousa@fe.up.pt), 2011

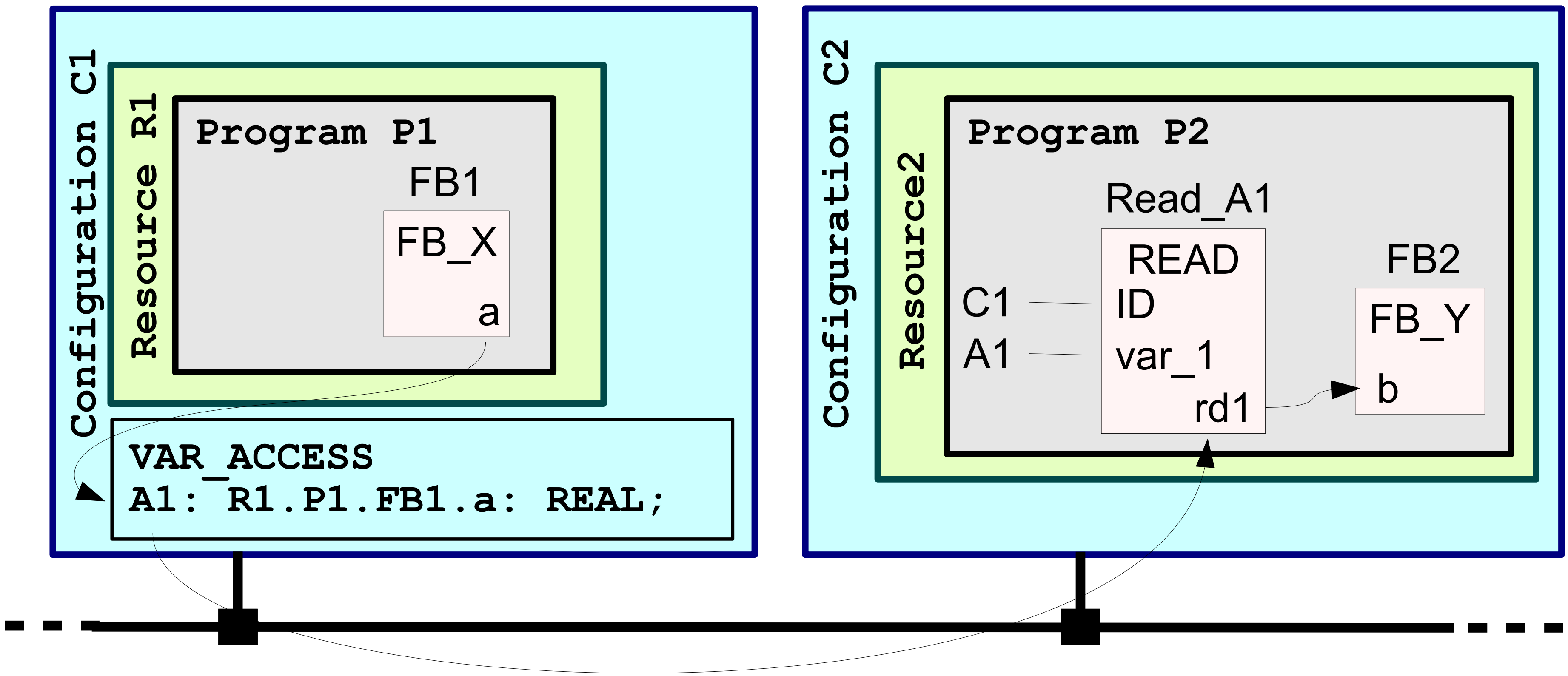
U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

IEC 61131-3 Execution Model
Communication Model



Data Flow between
two PROGRAMs using
IEC61131-5 comm. FBs

IEC 61131-3 Execution Model
Communication Model



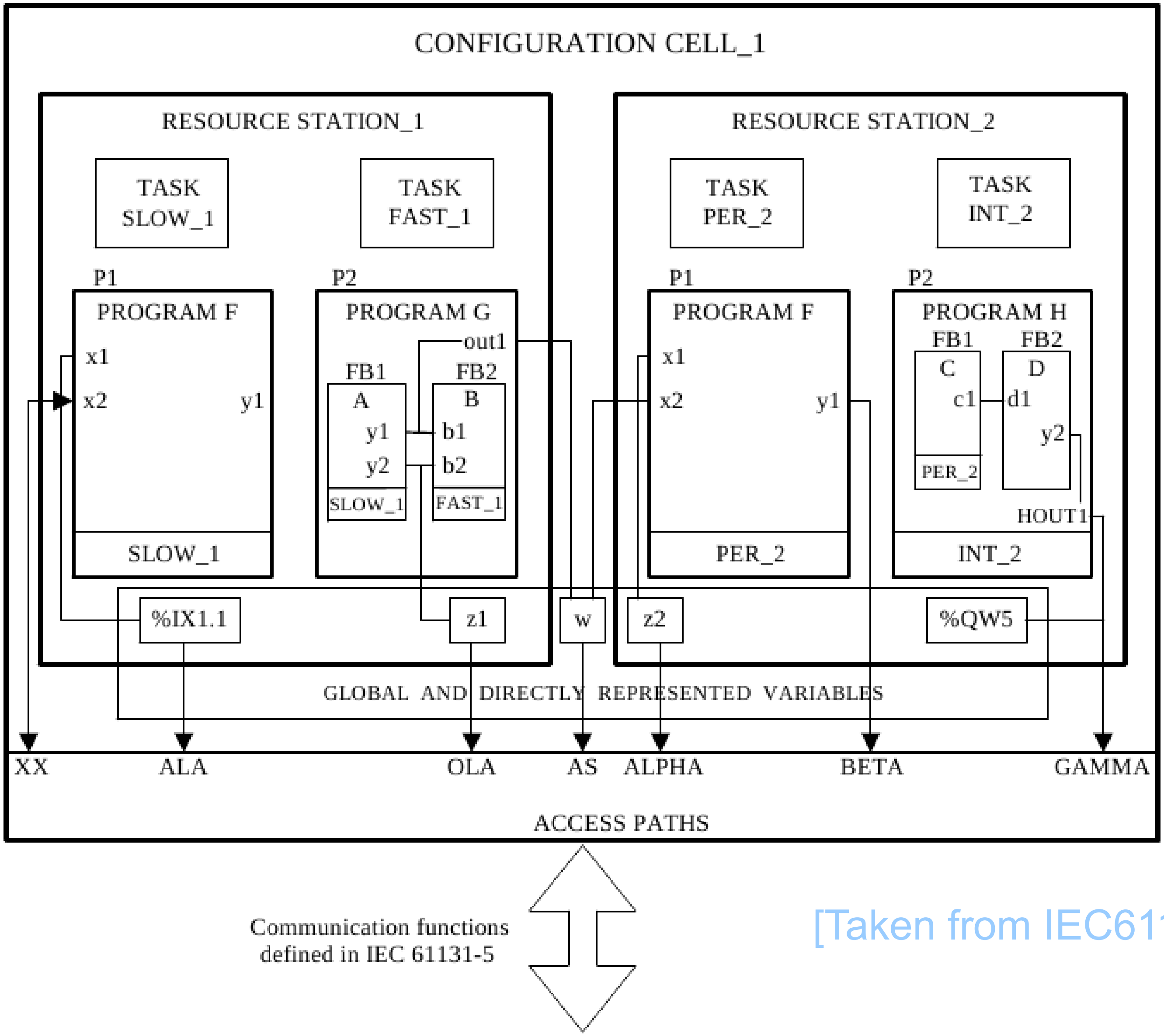
Data Flow between two PROGRAMs using IEC61131-5 comm. FBs and ACCESS paths

All direct variables may also be read using the READ FB!

Mário de Sousa (msousa@fe.up.pt), 2011

IEC 61131-3 Execution Model

- Configuration Example



[Taken from IEC61131-3]

IEC 61131-3 Execution Model

- Configuration Example

```
CONFIGURATION CELL_1
  VAR_GLOBAL  w: UINT;  END_VAR
  RESOURCE STATION_1 ON PROCESSOR_TYPE_1
    VAR_GLOBAL  z1: BYTE;  END_VAR
    TASK SLOW_1(INTERVAL := t#20ms, PRIORITY := 2) ;
    TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1) ;
    PROGRAM P1 WITH SLOW_1 :
                 F(x1 := %IX1.1) ;
    PROGRAM P2 : G(OUT1 => w,
                   FB1 WITH SLOW_1,
                   FB2 WITH FAST_1)  ;
  END_RESOURCE
  RESOURCE STATION_2 ON PROCESSOR_TYPE_2
    VAR_GLOBAL  z2      : BOOL ;
                AT %QW5 : INT  ;
    END_VAR
    TASK PER_2(INTERVAL := t#50ms, PRIORITY := 2) ;
    TASK INT_2(SINGLE := z2,       PRIORITY := 1) ;
    PROGRAM P1 WITH PER_2 :
           F(x1 := z2, x2 := w)  ;
    PROGRAM P4 WITH INT_2 :
           H(HOUT1 => %QW5,
             FB1 WITH  PER_2);
  END_RESOURCE
```

IEC 61131-3 Execution Model

- Configuration Example

```
VAR_ACCESS
   ABLE     : STATION_1.%IX1.1      : BOOL READ_ONLY  ;
   BAKER    : STATION_1.P1.x2       : UINT READ_WRITE ;
   CHARLIE  : STATION_1.z1          : BYTE            ;
   DOG      : w                     : UINT READ_ONLY  ;
   ALPHA    : STATION_2.P1.y1       : BYTE READ_ONLY  ;
   BETA     : STATION_2.P4.HOUT1    : INT READ_ONLY   ;
   GAMMA    : STATION_2.z2          : BOOL READ_WRITE ;
   S1_COUNT: STATION_1.P1.COUNT    : INT;
   THETA    : STATION_2.P4.FB2.d1   : BOOL READ_WRITE;
   ZETA     : STATION_2.P4.FB1.c1   : BOOL READ_ONLY;
   OMEGA    : STATION_2.P4.FB1.C3   : INT READ_WRITE;
END_VAR
VAR_CONFIG
   STATION_1.P1.COUNT : INT := 1;
   STATION_2.P1.COUNT : INT := 100;
   STATION_1.P1.TIME1 : TON := (PT := T#2.5s);
   STATION_2.P1.TIME1 : TON := (PT := T#4.5s);
   STATION_2.P4.FB1.C2 AT %QB25 : BYTE;
END_VAR
END_CONFIGURATION
```

IEC 61131-3 Standard Architecture
Functions

- A Function may be used to group code that we wish to call from different locations of the program.
- It is practically equivalent to functions in C and Pascal, (but not identical !)
- May be coded in: LD, IL, FBD, ST

```
FUNCTION Fact : LINT
  VAR_INPUT
    n : LINT;
  END_VAR
  (* Program Body in ST *)
  Fact := 1;
  WHILE (n > 1) DO
    Fact := Fact * n;
    n := n - 1;
  END_WHILE
END_FUNCTION
```

```
FUNCTION Comb : LINT
  VAR_INPUT
    n, m : LINT;
  END_VAR
  (* Program Body in ST *)
  Comb :=
        fact(n)/(fact(m)*fact(n-m));
END_FUNCTION
```

IEC 61131-3 Standard Architecture
Functions

- May have parameters
  - INPUT,
  - OUTPUT, (IEC 61131-3 v2)
  - IN_OUT, (IEC 61131-3 v2)
- The function itself returns one parameter (the name of the function works as a variable).
- Recursion is NOT allowed!

WRONG!!

```
FUNCTION Fact : LINT

  VAR_INPUT
    n : LINT;
  END_VAR

  (* Program Body in ST *)
  IF (n < 2)
    Fact := 1;
  ELSE
    Fact := n*Fact(n-1);
  END_IF

END_FUNCTION
```

IEC 61131-3 Standard Architecture
Functions

- Functions may be called using any programming language
  (except SFC, which is not really a programming language):
  - LD
  - IL
  - ST
  - FBD

```
PROGRAM Foo

  VAR
    c : LINT;
  END_VAR

  (* Program Body in ST *)
    c := fact (20);
    c := comb(10, 4);

END_PROGRAM
```

IEC 61131-3 Standard Architecture

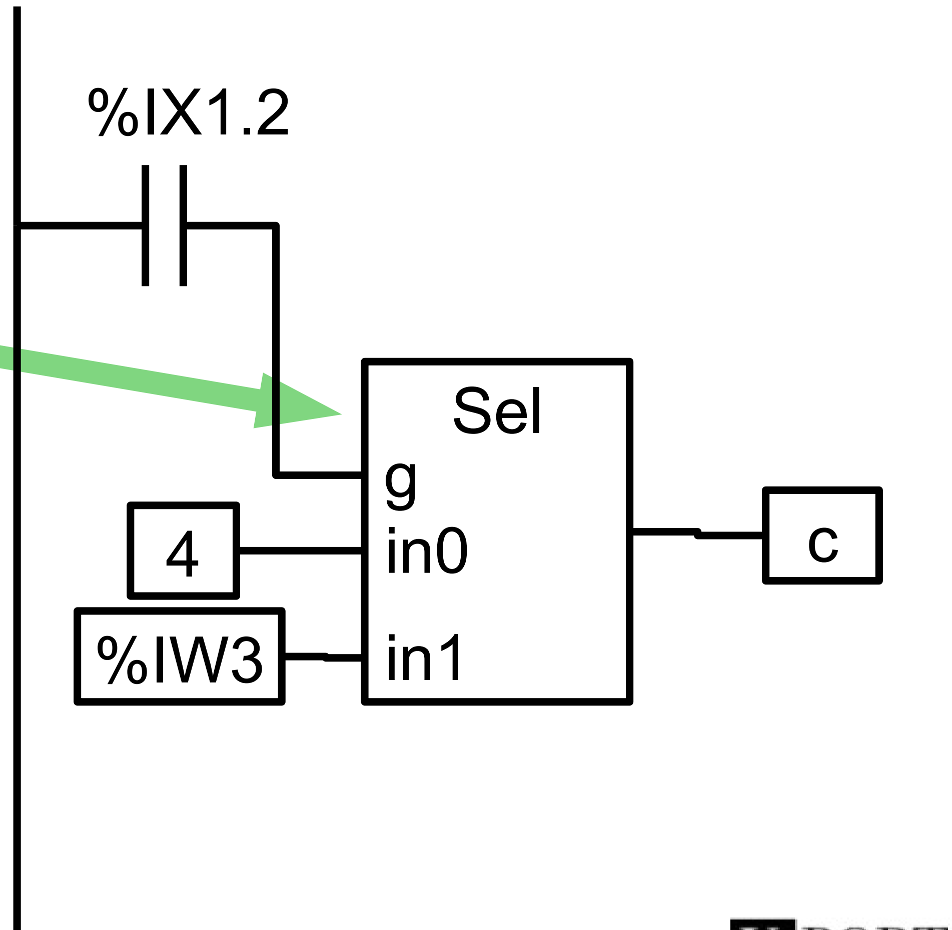
Functions

- Functions may be called using any programming language
  (except SFC, which is not really a programming language):
  - LD
  - IL
  - ST
  - FBD

```
PROGRAM Foo

  VAR
    c : LINT;
  END_VAR

  (* Program Body in IL *)
  LD 20
  Fact
  ST c
  LD 10
  Comb 4
  ST c

END_PROGRAM
```

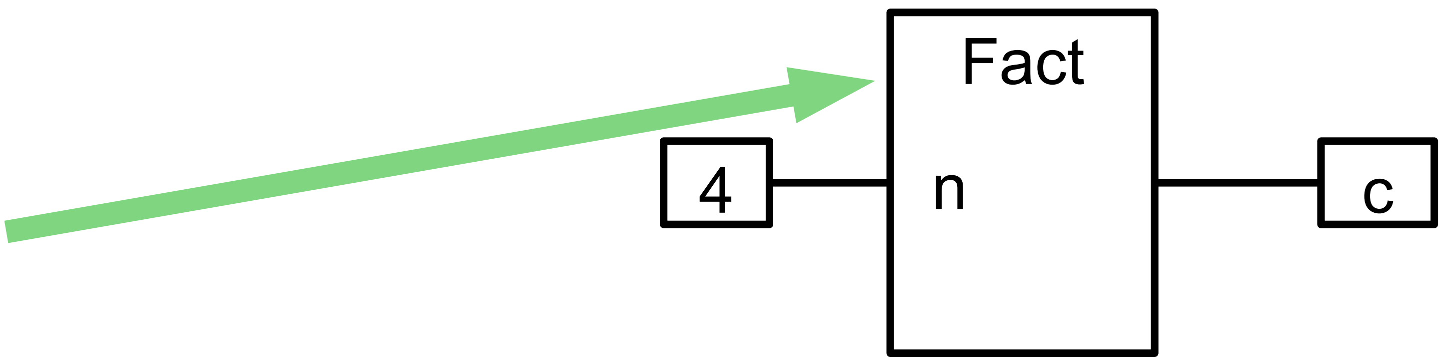
IEC 61131-3 Standard Architecture
Functions

- Functions may be called using any programming language
  (except SFC, which is not really a programming language):
  - LD
  - IL
  - ST
  - FBD



IEC 61131-3 Standard Architecture
Functions

- Functions may be called using any programming language
  (except SFC, which is not really a programming language):
  - LD
  - IL
  - ST
  - FBD



IEC 61131-3 Standard Architecture
Function Blocks

- A Function Block may be used to group code, and store state/data, that we wish to call and use from different locations of the program.
- Is different from a function as it may store persistent internal state/variables
(similar to an object in C++)
- May be written in any of the programming languages:
LD, IL, FBD, ST, SFC

```
FUNCTION_BLOCK Count_Up
  VAR_INPUT
    clk : BOOL;
  END_VAR
  VAR_OUTPUT
    count : INT;
  END_VAR
  VAR
    old_clk : BOOL := FALSE;
  END_VAR

  (* Program Body in ST *)
  IF (clk AND NOT old_clk) THEN
    count := count + 1;
  END_IF
END_FUNCTION_BLOCK
```

IEC 61131-3 Standard Architecture
Function Blocks

- We must distinguish between:
  - A Function Block type
  - A Function Block instance

Unfortunately the standard always refers to these two entities with the same name: Function Block

- Is different from a function as it may store persistent internal state/variables

A FB may not be called directly!

```
PROGRAM Flicker
  VAR
    light : BOOL;
    counter : count_up;
  END_VAR

  (*Program Body in ST*)
  light := NOT (light);
  counter (light);
  count_up (light);

END_PROGRAM
```

CORRECT!

WRONG!!

Mário de Sousa (msousa@fe.up.pt), 2011

IEC 61131-3 Standard Architecture
Function Blocks

- Function Blocks may have
  - INPUT
  - OUTPUT
  - IN_OUT

  parameters.

- Function Blocks may have internal variables:
  - Persistent
  - Temporary

NOTES
• Output variables are also persistent.
• Input variables are also persistent

```
FUNCTION_BLOCK Foo
  VAR_INPUT
    var1 : BOOL;
  END_VAR
  VAR_OUTPUT
    var2 : BOOL;
  END_VAR
  VAR_IN_OUT
    var3 : BOOL;
  END_VAR
  VAR
    var3 : BOOL;
  END_VAR
  VAR_TEMP
    var3 : BOOL;
  END_VAR
```

IEC 61131-3 Standard Architecture
Function Blocks

- Function Blocks may be instantiated inside
  - Programs
  - Function Blocks

  - But NOT Functions !!

FUNCTIONS must be idem-potent !

CORRECT!

WRONG!!

```
FUNCTION CRC: SINT

  VAR_INPUT
    n0,n1,n2,n3: SINT;
    count1: count_up;
  END_VAR
  VAR
    temp: SINT;
    count2: count_up;
  END_VAR

  (* Program Body in ST *)
  ...
END_PROGRAM
```

IEC 61131-3 Standard Architecture
Function Blocks

- Function Block may be called using any programming language (except SFC, which is not really a programming language):
  - LD
  - IL
  - ST
  - FBD

```
PROGRAM Flicker
  VAR
    light : BOOL;
    counter : count_up;
  END_VAR

  (* Program Body in ST *)
  light := NOT (light);
  counter (light);
  IF (counter.count > 30)
    THEN ...
  END_IF
END_PROGRAM
```

The FB code is only executed when the FB instance is invoked!

Direct access to input / output FB variables is allowed.
Direct access to internal variables is NOT allowed.
This does not imply execution of the FB code!

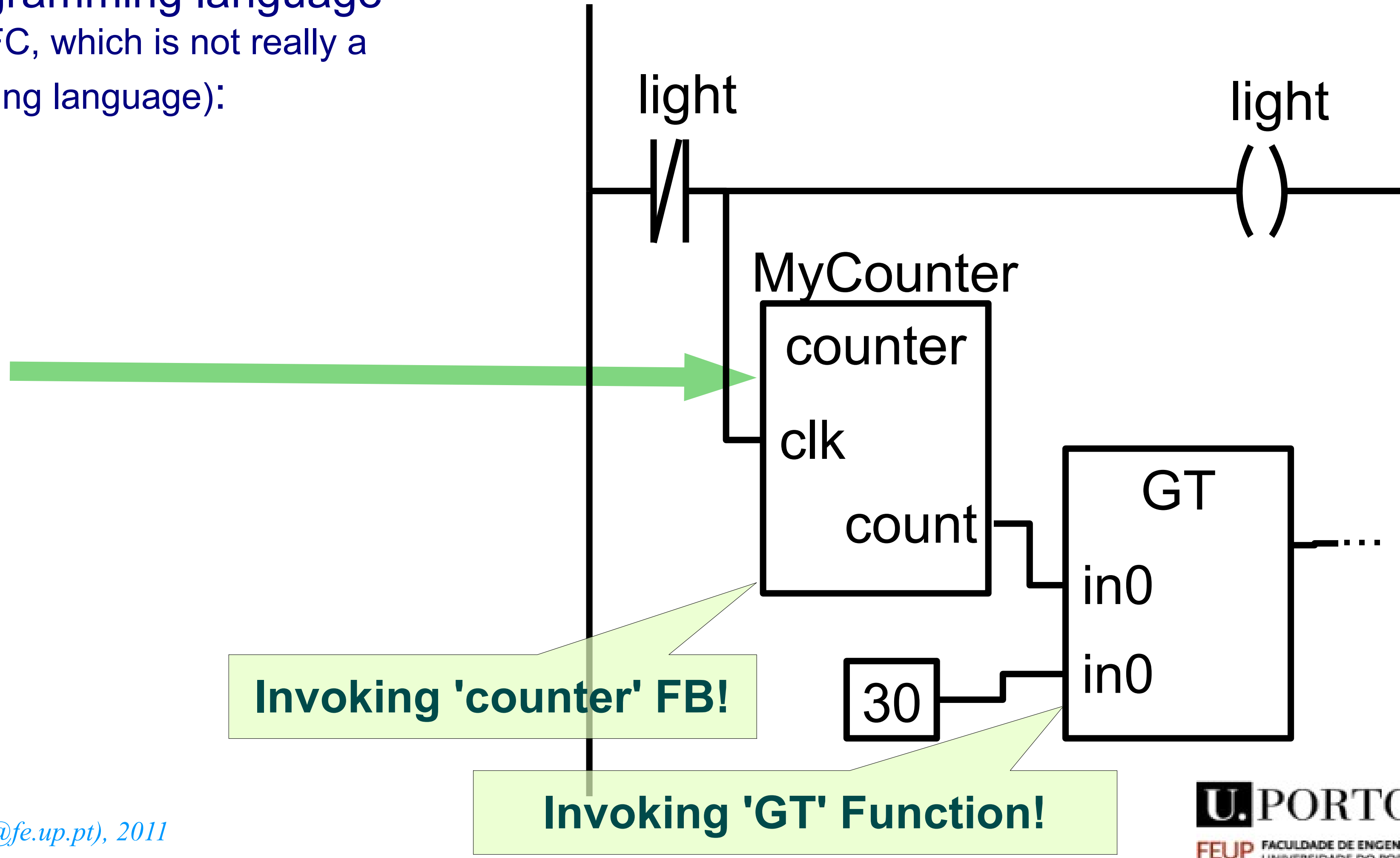
IEC 61131-3 Standard Architecture
Function Blocks

- Function Block may be called using any programming language (except SFC, which is not really a programming language):
  - LD
  - IL
  - ST
  - FBD

```
PROGRAM Flicker
  VAR
    light : BOOL;
    counter : count_up;
  END_VAR

  (* Program Body in IL *)
  LDN light
  ST light
  CALL counter (light)
  LD counter.count
  GT 30
  ...
END_PROGRAM
```

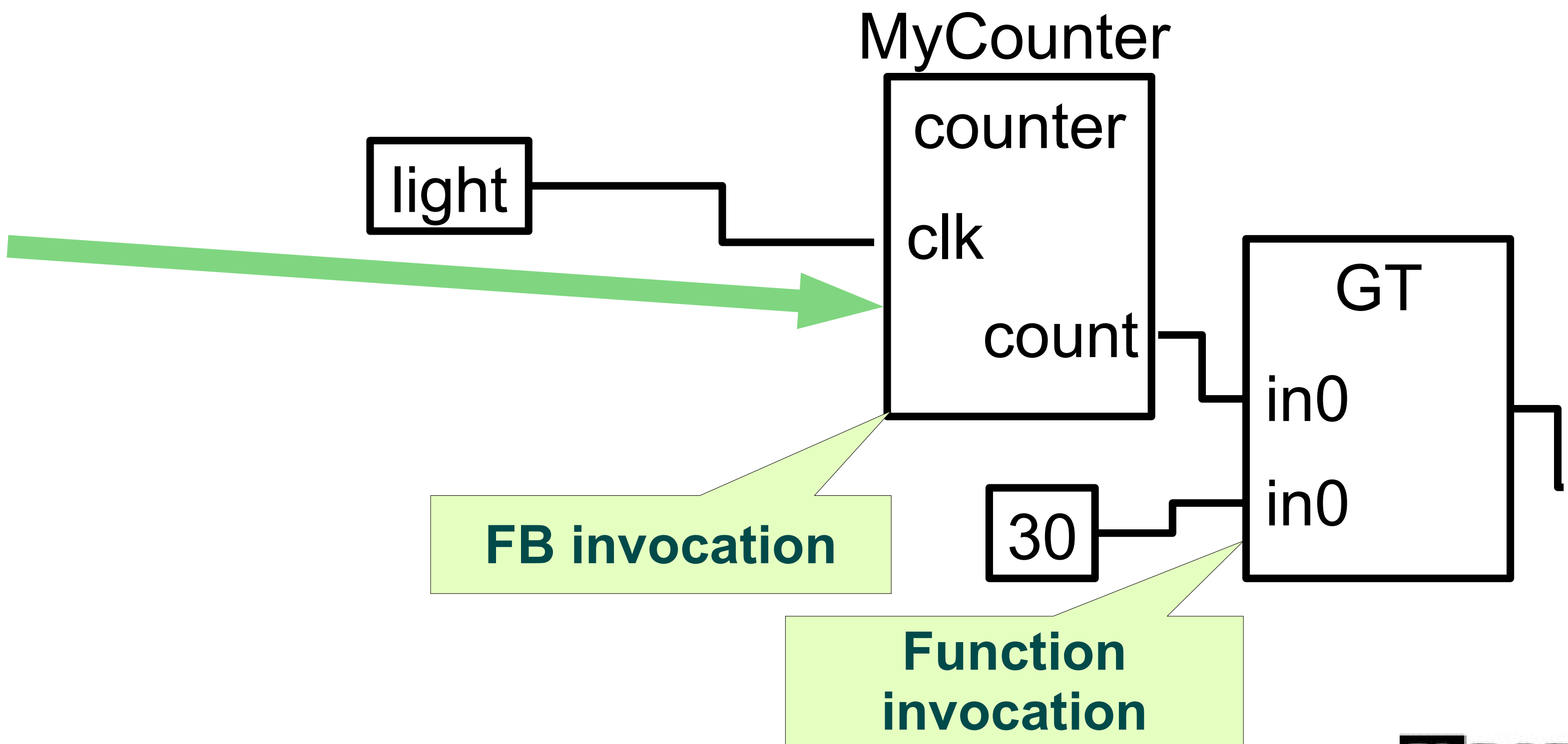
IEC 61131-3 Standard Architecture
Function Blocks

- Function Block may be called using any programming language (except SFC, which is not really a programming language):
  - LD
  - IL
  - ST
  - FBD



IEC 61131-3 Standard Architecture
Function Blocks

- Function Block may be called using any programming language
  (except SFC, which is not really a programming language):
  - LD
  - IL
  - ST
  - FBD



IEC 61131-3 Standard Architecture
Function Blocks

Calling a Function or a FB without providing explicit parameters to all Input variables is allowed!

In this case, the default values for the input variables is used

```
FUNCTION_BLOCK PID_t
  VAR_INPUT Error: Real; END_VAR
  VAR_INPUT P, I, D: Real; END_VAR
  VAR_OUTPUT out: Real; END_VAR
  ...
END_FUNCTION_BLOCK

PROGRAM Oven
  VAR PID: PID_t; END_VAR
  VAR temp_error: real; END_VAR
  VAR pwm_out: real; END_VAR
  ...
  PID(temp_error);
  pwm_out := PID.out;
  ...
END_PROGRAM
```

But, remember...

NOTES
- Output variables are also persistent.
- Input variables are also persistent

If another value was passed to the Input variable of a FB in a previous invocation, that value is used instead!

Overview of IEC 61131-3

- **Introduction to IEC 61131 and Related Standards**
- **IEC 61131-3 Standard Architecture**
- **Programming Languages**
- **IEC 61131-3 Execution Model**
- **Standard Functions and Function Blocks**
- **Examples**

Variables

IEC 61131-3 Standard Architecture

- Variables
  - Variables may be declared in several locations, typically inside POUs

  - A variable's visibility scope is limited to the POU in which it is declared

  - A variable's identifier (i.e. its 'name')
    - May contain any number of letters, numbers, or underscores ('_')
    - Identifiers are case insensitive (e.g. 'RunMode' <=> ' RUNMODE' <=> 'runmode')
    - Must start with a letter
    - May not contain two or more consecutive underscores
      (e.g. valid: 'Mode_', 'Run_Mode', 'Run_Mode_', 'Valve1', 'Valve_1', 'T1_Valve')
      (e.g. invalid: 'Mode__', '_Run_Mode', 'Run__Mode', '1Valve', '1_Valve', '#1_Valve')

IEC 61131-3 Standard Architecture

Direct Addressing

- Percentage sign ‘%’
- Area-prefix
  - I input
  - Q output
  - M marker
- Size
  - X single bit
  - None single bit
  - B byte (8 Bits)
  - W word (16 Bits)
  - D double word (32 Bits)

- Examples

%IW215

%QX1.1

%MD48

IEC 61131-3 Standard Architecture

- Variables
  - Direct Variables – variables that directly reference a memory area
  - Direct Variables Memory Areas
    - %I - input memory (typically where state of physical inputs has been mapped)
    - %Q - output memory (typically memory that will be copied to physical outputs)
    - %M - internal memory
  - Direct Variables Data Types
    - %M - BOOL, 1 bit variable
    - %MX - identical to %M
    - %MB - BYTE, 8 bits
    - %MW - WORD, 16 bits
    - %MD - DWORD, 32 bits
    - %ML - LWORD, 64 bits
  - Direct Variable Location
    - Specified by a sequence of unsigned integers, separated by periods.
    - Interpreted as a hierarchical physical or logical address
    - Mapping is implementation defined.
      e.g.: %MW3.42.21.3

IEC 61131-3 Standard Architecture

Variable Declaration

```
VAR
    Level              : UINT;
    Level_Setpoint     : USINT   := 120;
    Manip_value        : REAL;
    DI_1, DI_2, DO_1   : BOOL;
END_VAR
```

For *global* variables, use the keywords VAR_GLOBAL and END_VAR

IEC 61131-3 Standard Architecture

Variable Declaration

```
VAR
    Dig_in      AT %IX0.0   : BOOL      := TRUE;
    Dig_out     AT %QX0.5   : BOOL;
    Manip_val               : REAL      := 48.5;
    Temp_ref                : INT       := 70;
    Label                   : STRING    := 'Degrees';
    Light                   : BOOL
    time1                   : TOD;
    time2                   : TIME      := T#70m_30s;
    date1                   : DATE      := DATE#2007-06-18;
END_VAR
```

IEC 61131-3 Standard Architecture

Variable Declaration

- If you want to declare a constant, that is a value that's not going to be changed by the program, you can use

```
VAR CONSTANT
        Pi  : REAL    := 3.14;
END_VAR
```

- To declare a variable which value shall be stored in case of a power failure, you use

```
VAR RETAIN
        Blaha     : INT;
END_VAR
```

IEC 61131-3 Standard Architecture

- Variables
  - Initialization of Direct Memory Areas
    - The initial value of a direct memory area my be explicitly defined

      e.g.: `VAR AT %IW203: INT := 465; END_VAR`
  - Location of Symbolic Variables
    - The memory where where variables with a symbolic name are located is automatically determined by the IEC 61131-3 compiler/execution environment
    - However, an explicit memory location may be specified for these variables too

      e.g.: `VAR Temp1 AT %IW10.6: INT; END_VAR`
  - CONSTANT modifier
    - Variables declared as constant may only be read.
  - RETAIN modifier
    - Variables declared with RETAIN will retain their state for the next warm restart (as defined in IEC 61131-1).

IEC 61131-3 Standard Architecture

- Variables
  - Variables may have distinct semantics: depends on how they are declared

	FUNCTION	FB	PROGRAM	CONFIGURATION	RESOURCE
VAR	X	X	X		
VAR_TEMP		X	X		
VAR_INPUT	X	X	X		
VAR_OUTPUT	X	X	X		
VAR_IN_OUT	X	X	X		
VAR_EXTERNAL		X	X		
VAR_GLOBAL				X	X
VAR_ACCESS				X	
VAR_CONFIG				X	

VAR_CONFIG does not declare new variables.
It is only used to configure previously declared variables!

IEC 61131-3 Standard Architecture

- Variables
  - Modifiers also change variable semantics

	RETAIN NON_RETAIN	AT	CONSTANT	R_EDGE F_EDGE	READ_ONLY WRITE_ONLY
VAR	X	X	X		
VAR_TEMP					
VAR_INPUT	X			X	
VAR_OUTPUT	X				
VAR_IN_OUT					
VAR_EXTERNAL			X		
VAR_GLOBAL	X	X	X		
VAR_ACCESS					X
VAR_CONFIG		X			

VAR_INPUT may store state between FB invocations
=> RETAIN also makes sense in this case!

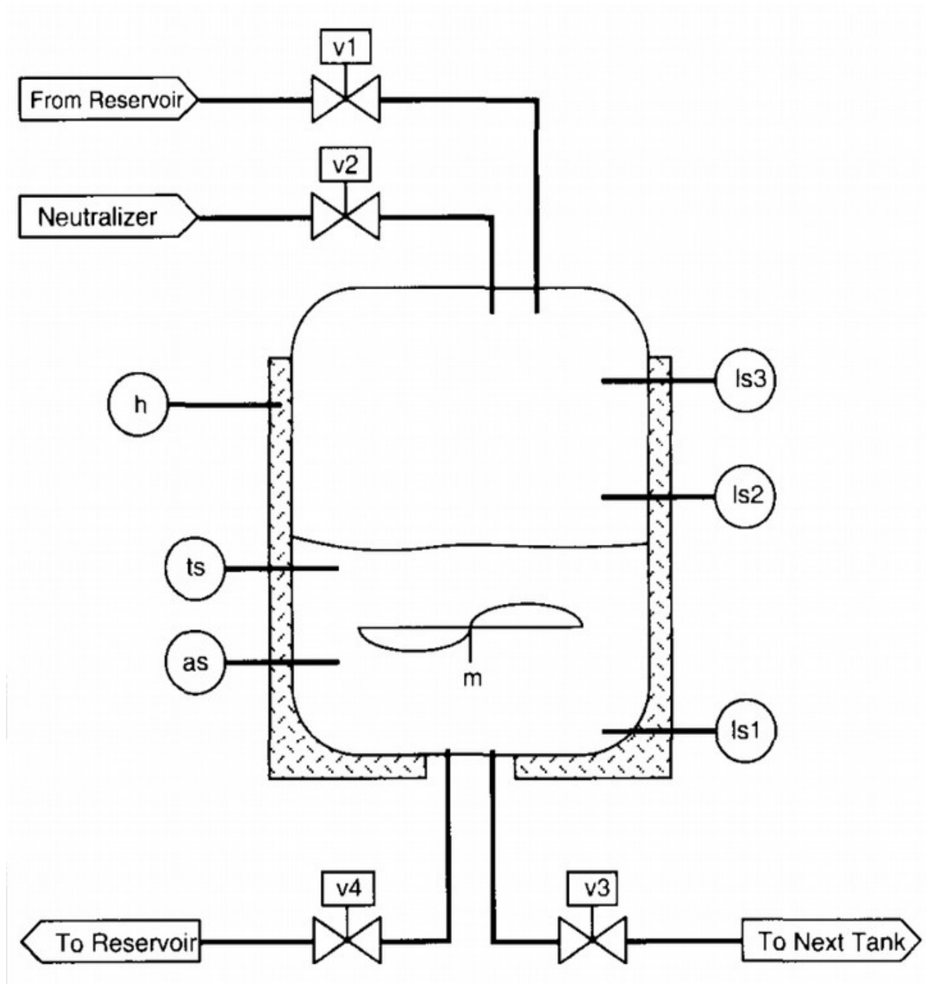
Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- **Programming Languages**
  - **LD – Ladder Diagram**
  - FBD – Function Block Diagram
  - ST – Structured Text
  - SFC – Sequential Function Chart
  - FBD – Function Block Diagram
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

LD - example

Requirements:

- start with every actuator OFF and empty tank
- "start" button is pressed → open valve "v1" until the solution reaches "ls2" level. "v1" is then turned OFF;
- After solution reaches "ls2" level, "m" is activated. It is only deactivated after the solution goes below "ls1" level;
- Whenever the temperature of the solution is below a pre-defined value (indicated by "ts" being OFF), the heater "h" is turned ON;
- Whenever the pH of the solution is unbalanced (indicated by "as" being OFF), "v2" is opened;
- "v2" makes the solution level rise. If the solution reaches "ls3" level, "v2" is closed and "v4" is opened. Valve "v4" reduces the level of solution in the tank. When solution goes below "ls2" level, turn "v4" OFF and turn "v2" ON;

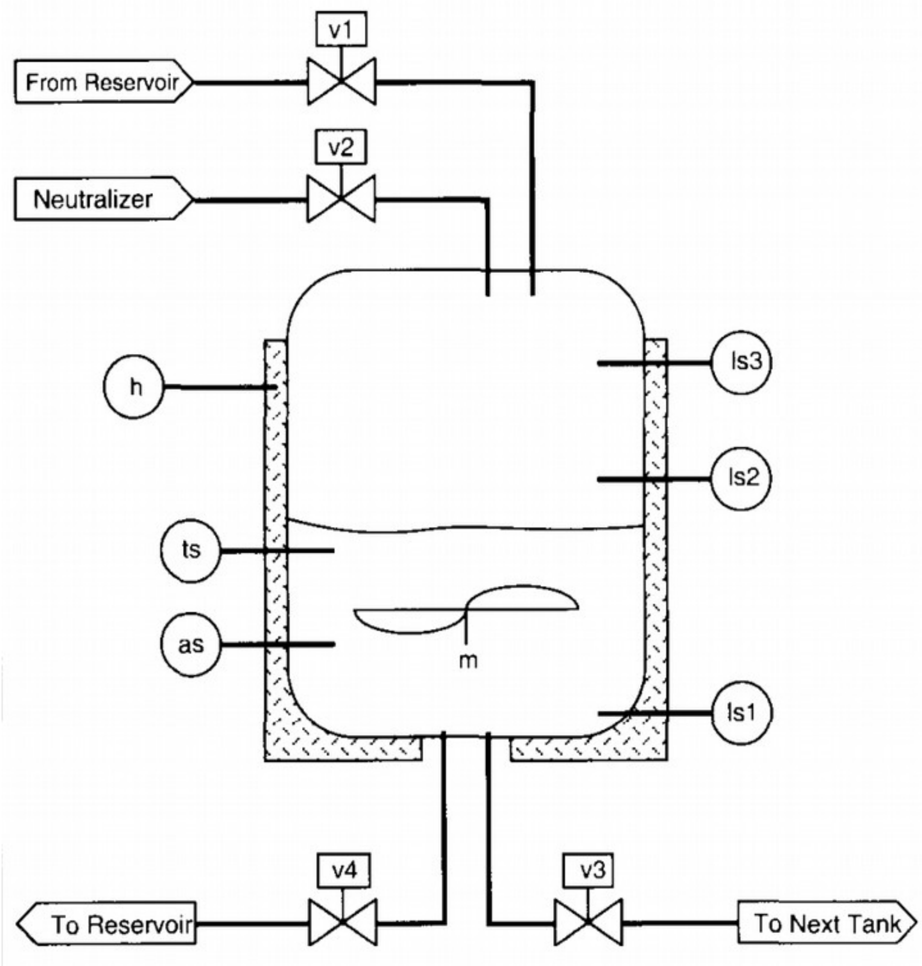


From: A. Falcione and B.H. Krogh, Design recovery for relay ladder logic,Control Systems IEEE, vol. 13, pp. 90-98, 1993, ISSN 1066-033X.

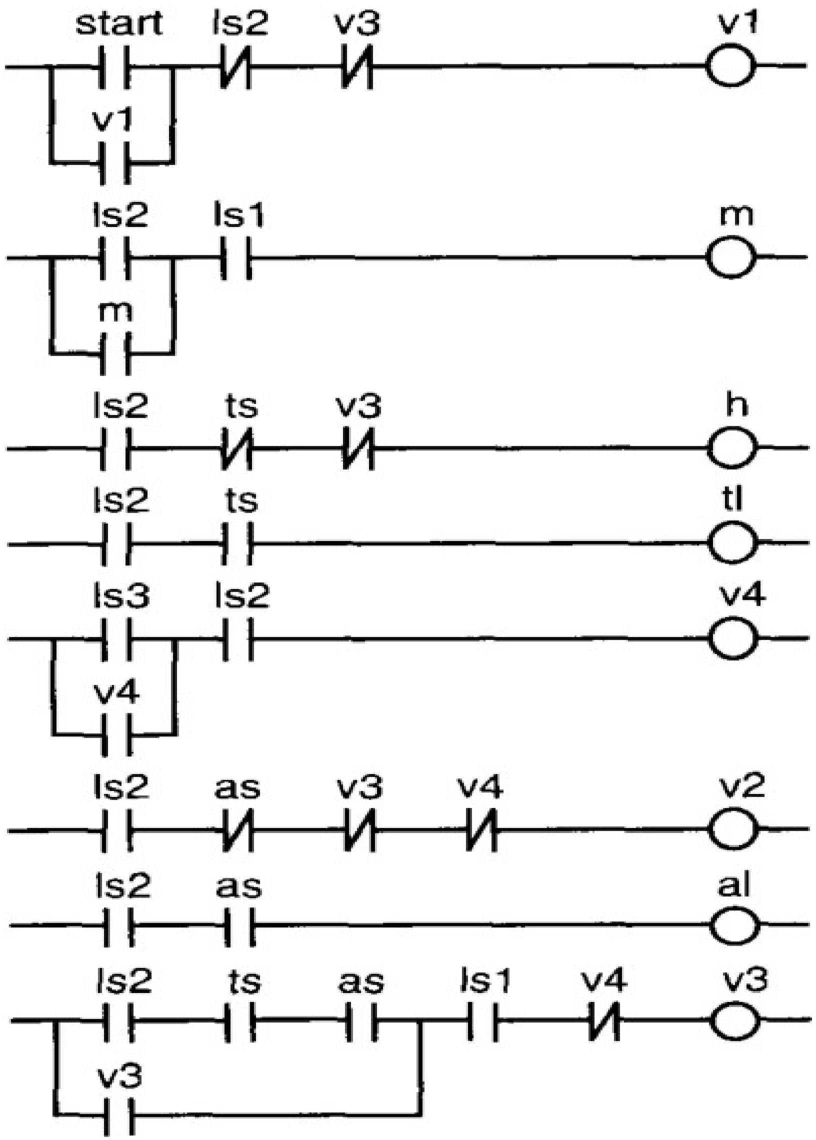
LD - example

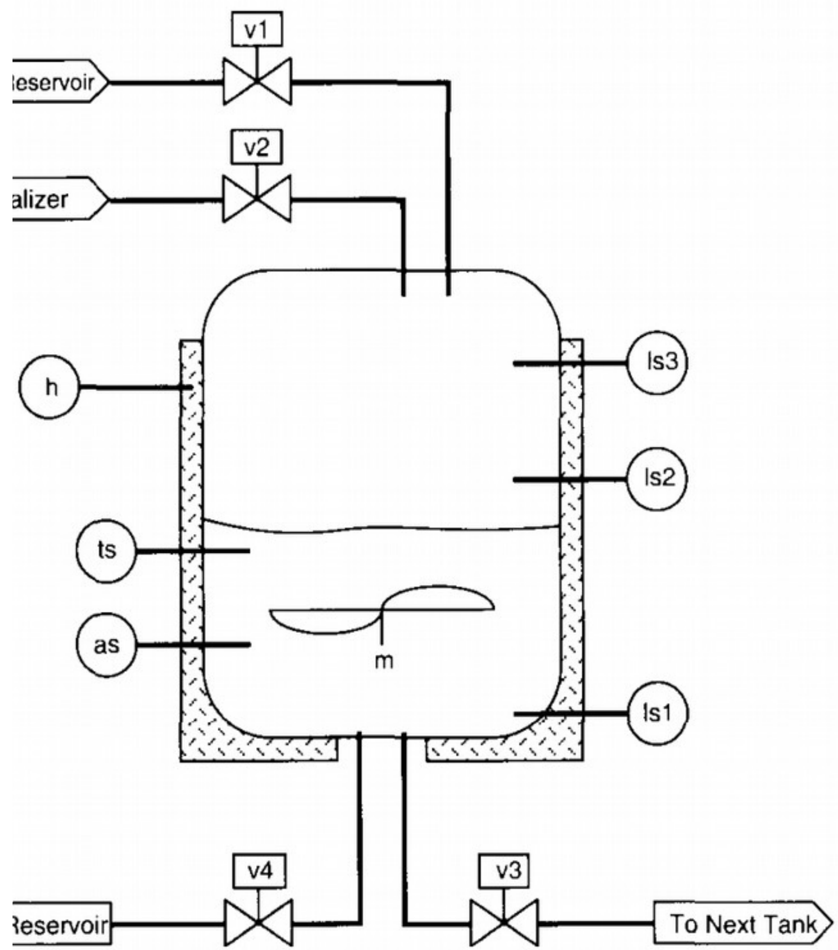
Requirements:
(continuation)

- If the temperature and pH are optimal, "v3" is opened and the level of solution is reduced. When it goes below "ls1" the tank is empty - return to the starting point and wait for the next "start" activation;
- Whenever "ts" is ON, a light "tl" becomes ON;
- Whenever "as" is ON, a light "al" becomes ON.

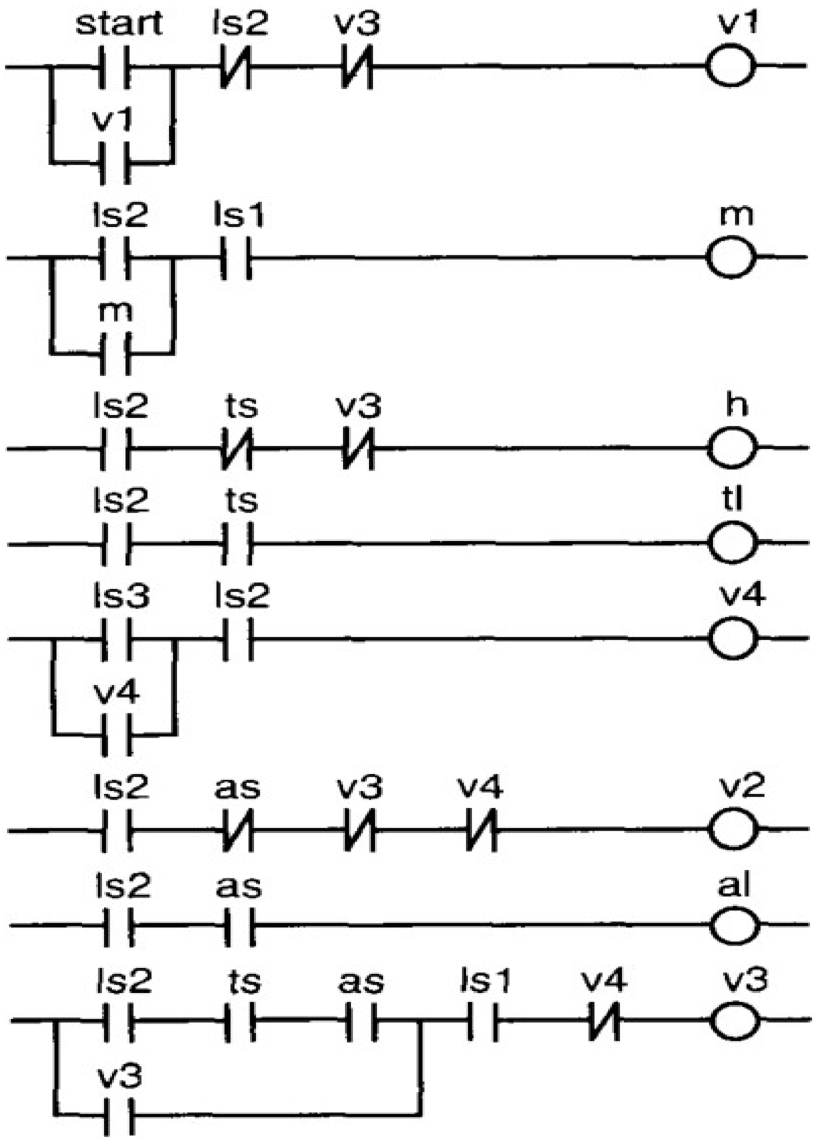


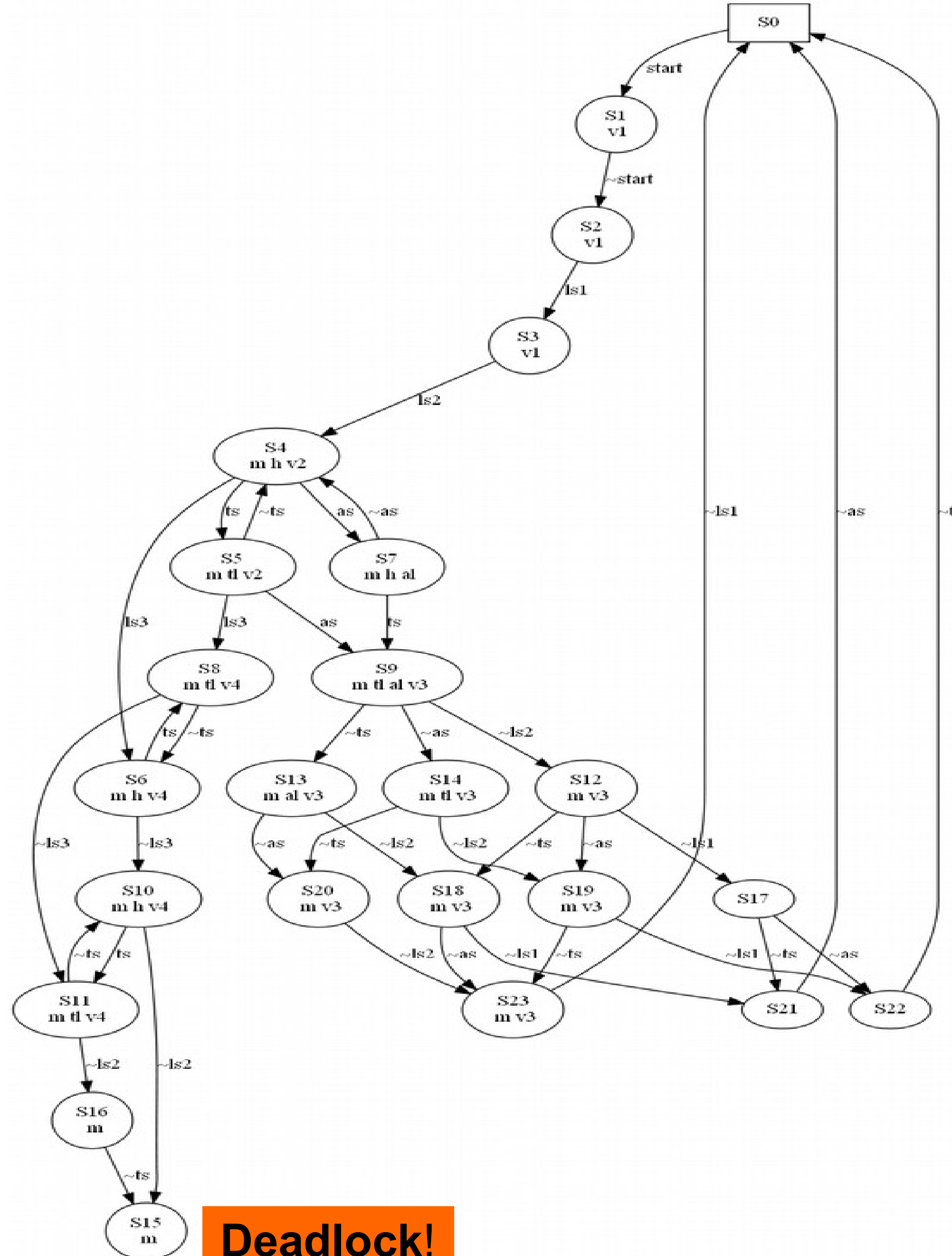
LD - example


start
ls2
v3
v1
v1
ls2
ls1
m
m
ls2
ts
v3
h
ls2
ts
tl
ls3
ls2
v4
v4
ls2
as
v3
v4
v2
ls2
as
al
ls2
ts
as
ls1
v4
v3
v3


v1
v2
alizer
h
ls3
ls2
ts
as
m
ls1
v4
v3
To Next Tank

LD - example





Deadlock!

IEC 61131-3 Standard Architecture

- LD

Table 61 - Contacts [a]

Static contacts		
No.	**Symbol**	**Description**
1 2	*** --| |-- or *** --! !--	**Normally open contact** The state of the left link is copied to the right link if the state of the associated Boolean variable (indicated by "***") is ON. Otherwise, the state of the right link is OFF.
3 4	*** --|/|-- or *** --!/!--	**Normally closed contact** The state of the left link is copied to the right link if the state of the associated Boolean variable is OFF. Otherwise, the state of the right link is OFF.
Transition-sensing contacts		
5 6	*** --|P|-- or *** --!P!--	**Positive transition-sensing contact** The state of the right link is ON from one evaluation of this element to the next when a transition of the associated variable from OFF to ON is sensed at the same time that the state of the left link is ON. The state of the right link shall be OFF at all other times.
7 8	*** --|N|-- or *** --!N!--	**Negative transition-sensing contact** The state of the right link is ON from one evaluation of this element to the next when a transition of the associated variable from ON to OFF is sensed at the same time that the state of the left link is ON. The state of the right link shall be OFF at all other times.

[Taken from IEC61131-3]

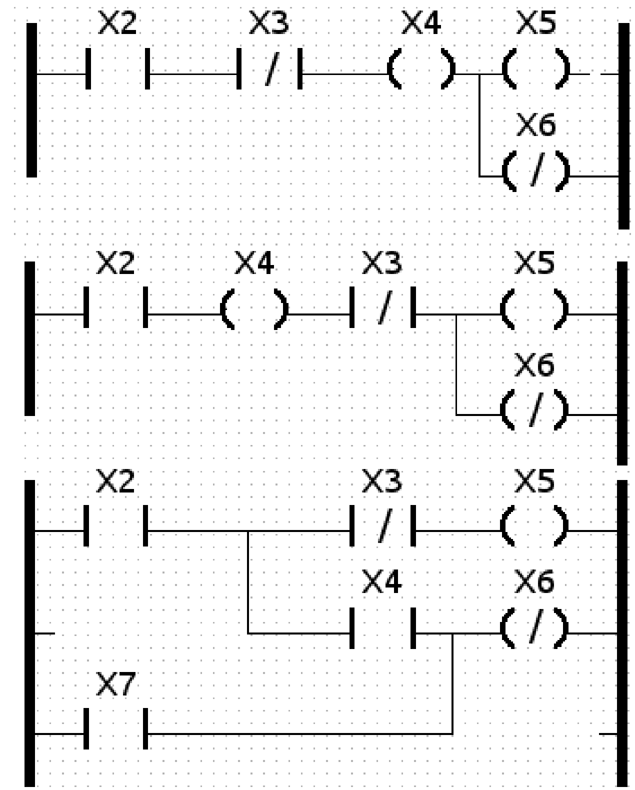
IEC 61131-3 Standard Architecture

- LD

Table 62 - Coils		
No.	**Symbol**	**Description**
		Momentary coils
1	`***` `--( )--`	**Coil** The state of the left link is copied to the associated Boolean variable and to the right link.
2	`***` `--(/)--`	**Negated coil** The state of the left link is copied to the right link. The inverse of the state of the left link is copied to the associated Boolean variable, that is, if the state of the left link is `OFF`, then the state of the associated variable is `ON`, and vice versa.
		Latched Coils
3	`***` `--(S)--`	**SET (latch) coil** The associated Boolean variable is set to the `ON` state when the left link is in the `ON` state, and remains set until reset by a `RESET` coil.
4	`***` `--(R)--`	**RESET (unlatch) coil** The associated Boolean variable is reset to the `OFF` state when the left link is in the `ON` state, and remains reset until set by a `SET` coil.
		Transition-sensing coils
8	`***` `--(P)--`	**Positive transition-sensing coil** The state of the associated Boolean variable is `ON` from one evaluation of this element to the next when a transition of the left link from `OFF` to `ON` is sensed. The state of the left link is always copied to the right link.
9	`***` `--(N)--`	**Negative transition-sensing coil** The state of the associated Boolean variable is `ON` from one evaluation of this element to the next when a transition of the left link from `ON` to `OFF` is sensed. The state of the left link is always copied to the right link.

[Taken from IEC61131-3]

IEC 61131-3 Standard Architecture



- LD – Ladder Diagram
  - Multiple coils in series are allowed
    - The output value is the same as the input value
  - Coils to the left of a Contact are allowed.
    - The output value is the same as the input value
  - Power flows from left to right
  - Conditional & Unconditional Jumps are allowed
    - Makes it possible to implement complex code flow

```
|   %IX20   %MX50
+---| |-----| |--->>NEXT
|
|

NEXT:
|    %IX25       %QX100 |
+----| |----+----( )---+
|    %MX60  |           |
```

Overview of IEC 61131-3

- **Introduction to IEC 61131 and Related Standards**
- **IEC 61131-3 Standard Architecture**
- **Programming Languages**
- **IEC 61131-3 Execution Model**
- **Standard Functions and Function Blocks**
- **Examples**

POUs (Program Organization Units)

Function

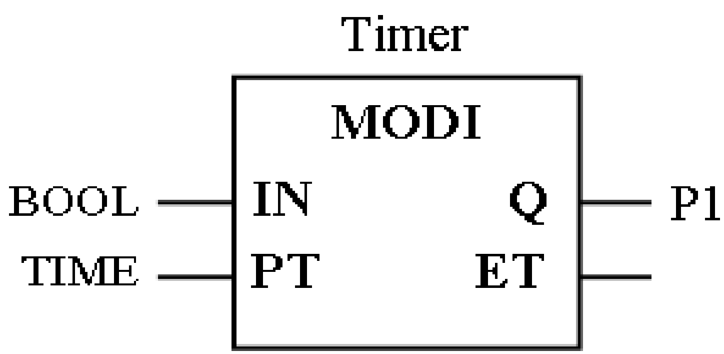
Function Block

Program

Configuration

Timers

- Are used to alter the state of an Boolean variable some user-defined time after a condition has been met.
- Symbol in LD/FBD:



- 3 different modes (types)
  - TON, TOF and TP

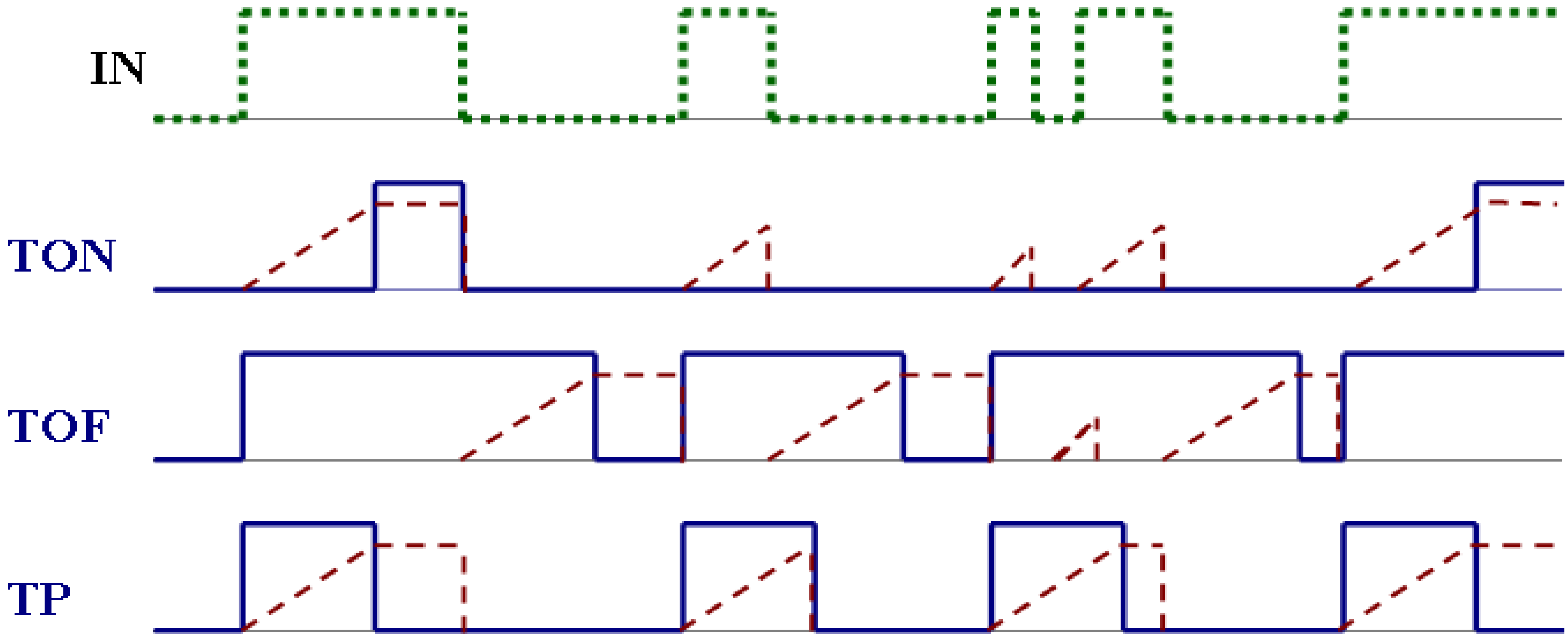
Timers

- In-variable IN:
  - when this variable changes state, the Timer will be "trigged/activated"
- In-variable PT (Predefined Time):
  - Associate it with a variable of data type TIME or assigned a time-value direct, i.e. t#2m30s
- Out-variable ET (Elapsed time):
  - A variable of data type TIME containing the value of the elapsed time since the Timer was activated
  - When the value of ET equals PT, the Timers output, will change state
- Timer output Q:
  - The state of this depends on IN, ET vs. PT and the timer-mode.

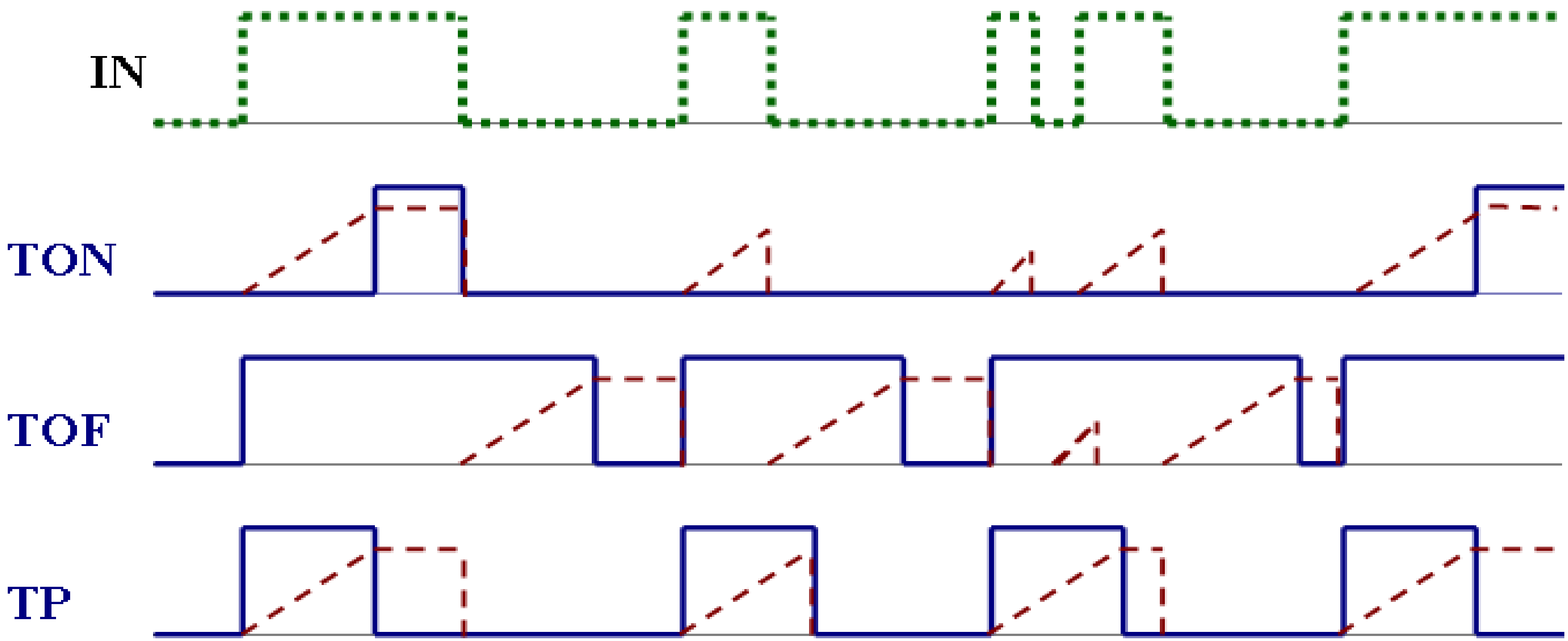
Timers – different modes

- TON (On-delay):
  - The output Q becomes TRUE a defined time (PT) after the condition on the control-input IN becomes True.
  - The input IN must be TRUE longer than the time defined for variable PT for the output Q to become TRUE.
- TOF (Off-Delay):
  - Q is set TRUE immediately when the IN-condition is fulfilled (becomes TRUE).
  - After the input IN changes state to FALSE again, the output Q becomes FALSE a time PT later, as long as the input IN doesn't changes state to TRUE in the meantime.
- TP (Pulse):
  - Q becomes TRUE as IN becomes TRUE, and stays TRUE in a user-defines time PT, no matter the state of IN.

Timers – different modes



Timers – different modes



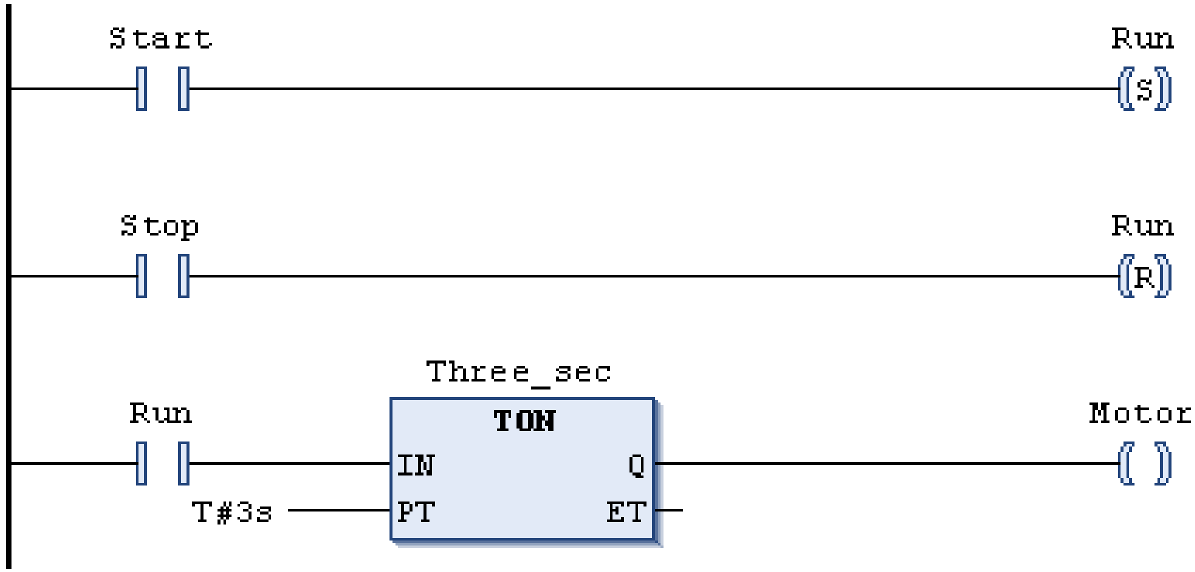
Exercise:
Write a program where a Motor starts 3 seconds after a Start-button is pressed.

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Timer – Example in LD

Exercise:
Write a program where a Motor starts 3 seconds after a Start-button is pressed.

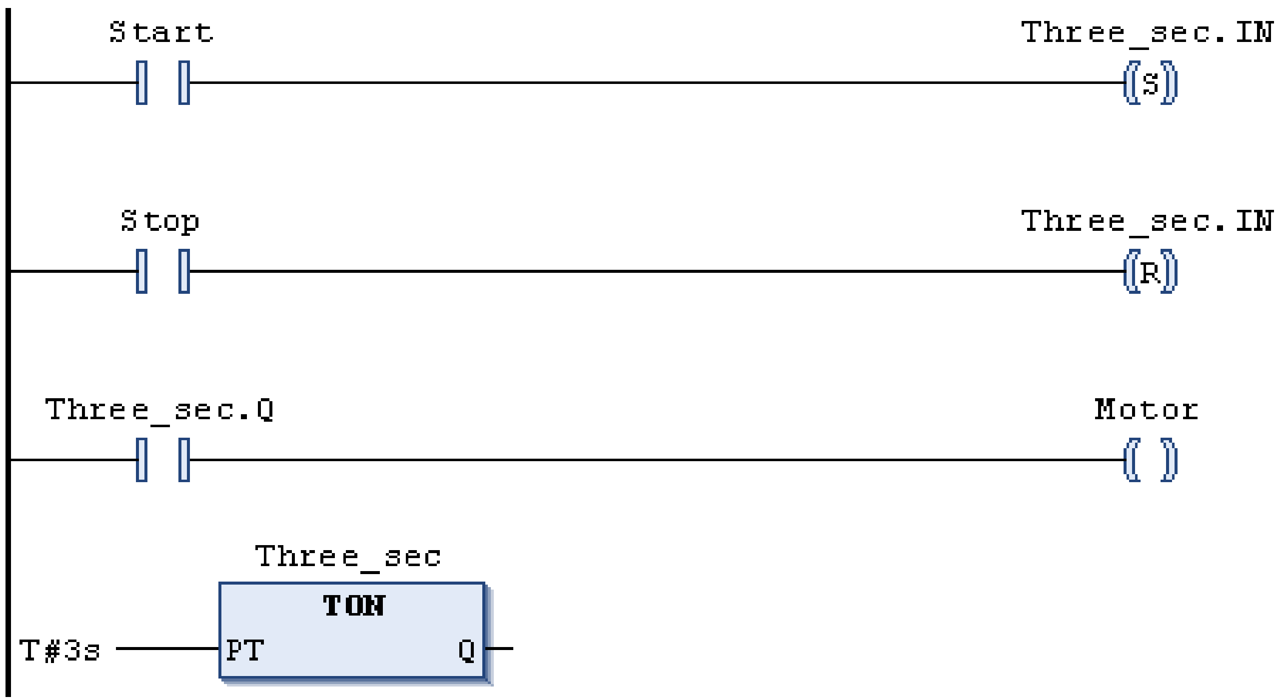
```
PROGRAM Timer1
VAR
     Start, Stop, Run       : BOOL;
     Three_sec              : TON;
     Motor    AT %Q4.2      : BOOL;
END_VAR
```



Timer – Example (Object referencing)

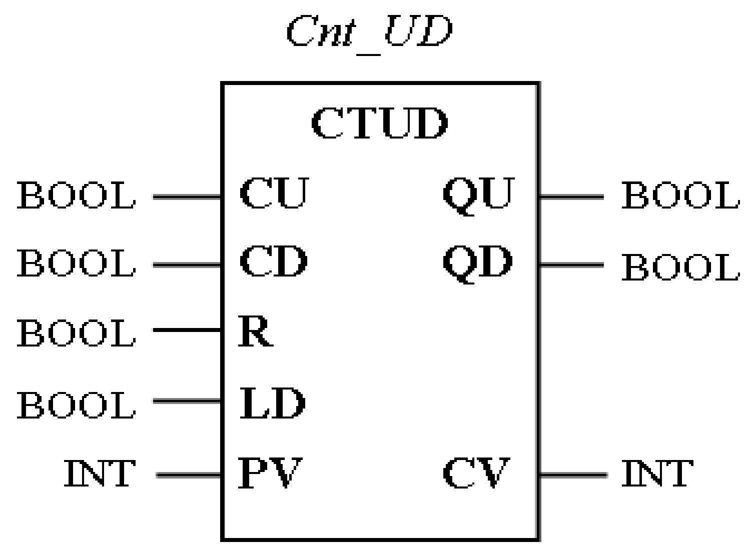
Exercise:
Write a program where a Motor starts 3 seconds after a Start-button is pressed.

```
PROGRAM Timer2
VAR
    Start     AT %IX2.4   : BOOL;
    Stop      AT %IX2.5   : BOOL;
    Motor     AT %QX4.2   : BOOL;
    Three_sec             : TON;
END_VAR
```



Counters

- Are being used to count pulses, like for instance each time a boolean signal changes state for False to True.
- 3 different types: CTU, CTD and CTUD

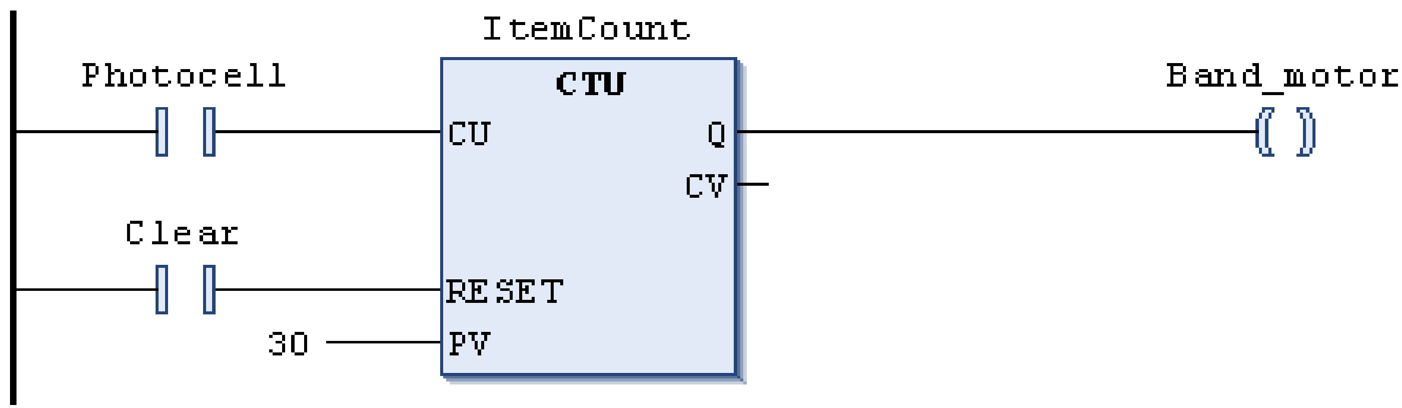


- In and Out:
  - **PV** Preset value
  - **CU** Count Up
  - **CD** Count Down
  - **R** Reset => CV = 0
  - **CV** Current Value
  - **LD** Load => CV = PV (CTD/CTUD)
  - **Q/QU** = True when CV >= PV (CTU/CTUD)
  - **Q/QD** = True when CV = 0 (CTD/CTUD)

Counter – Example in LD

Exercise:
Write a program where %QX2.0 is activated once 30 work pieces pass photocell.

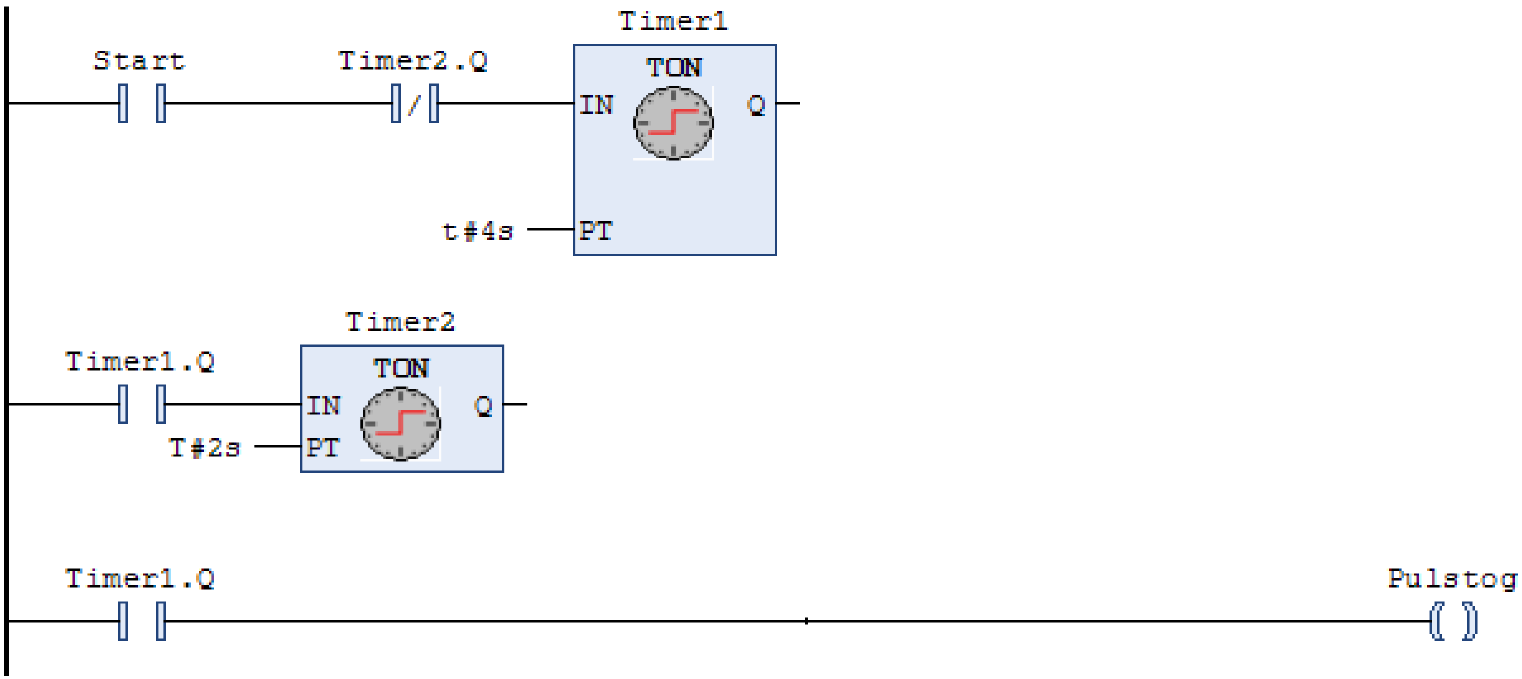
```
PROGRAM Count
VAR
    Photocell  AT %IX1.0 : BOOL;
    Clear      AT %IX1.1 : BOOL;
    Band_Motor AT %QX2.0 : BOOL;
    ItemCount            : CTU;
END_VAR
```



Example: Blink

Exercise:
Write a program that toggles a signal (Pulsetog) on (2s) and off (4s) continuously.

```
PROGRAM BLINK
VAR
    Start   : BOOL;
    Pulstog : BOOL;
    Timer1, Timer2 : TON;
END_VAR
```



Example: Blink v.2

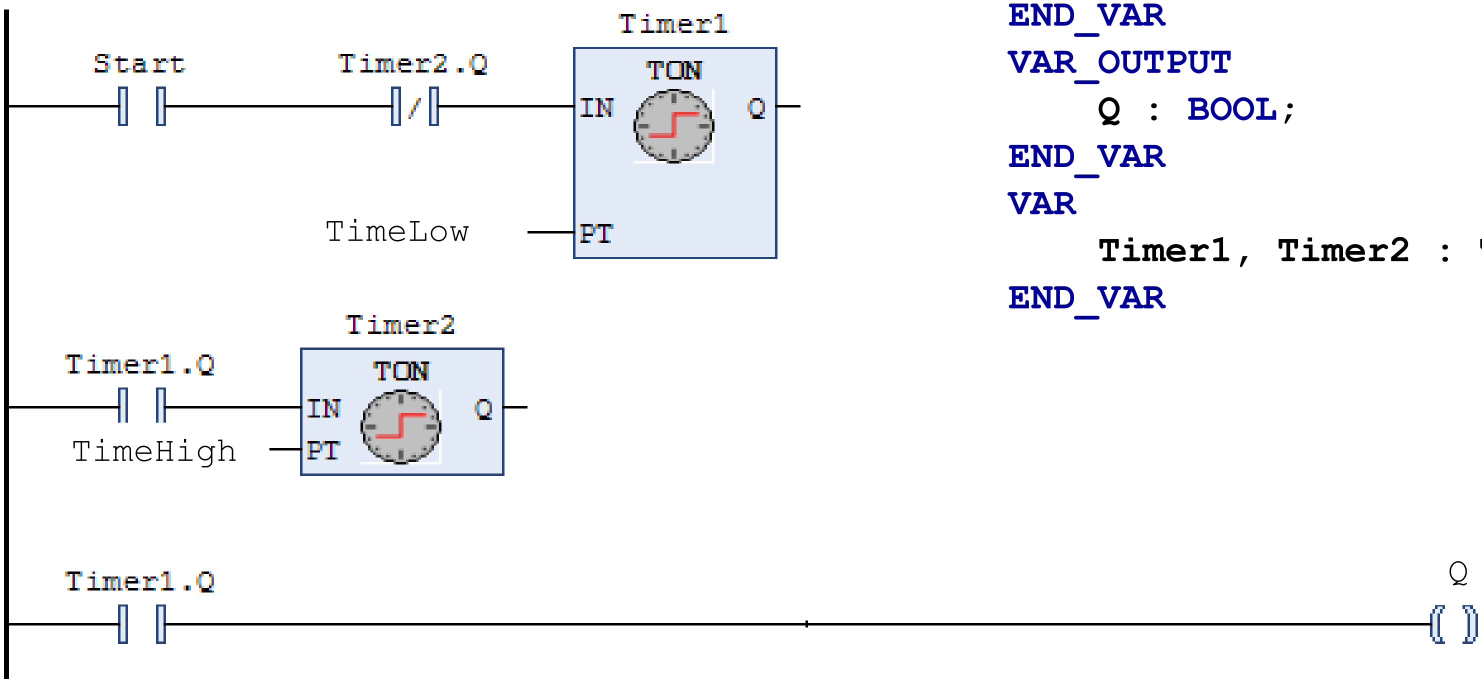
Exercise:
Write a program that toggles a signal (Puls) on (2s) and off (4s) continuously.

```
PROGRAM BLINK_v2
VAR
    Blinker : BLINK;
    Start   : BOOL;
    Puls    : BOOL;
END_VAR
```

Here an existing function block, BLINK, is used instead.
*The BLINK FB is **not** part of the IEC 61131-3 standard library*
(but is present in Codesys). We can create the BLINK FB ourselves!

Example: User defined Function Block

Exercise:
Write a FUNCTION_BLOCK that toggles an output (Q) on and off continuously.
Q stays on for TimeHigh seconds, and off for TimeLow seconds.




```
FUNCTION_BLOCK BLINK
VAR_INPUT
    Start : BOOL;
    TimeLow, TimeHigh  : Time;
END_VAR
VAR_OUTPUT
    Q : BOOL;
END_VAR
VAR
    Timer1, Timer2 : TON;
END_VAR
```

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

User Defined Function Blocks

- A Function Block is a POU that is specially designed to be called from other POUs (Programs and FBs). Unlike a Function, a FB has memory so that each call, with the same arguments, may cause a different result.

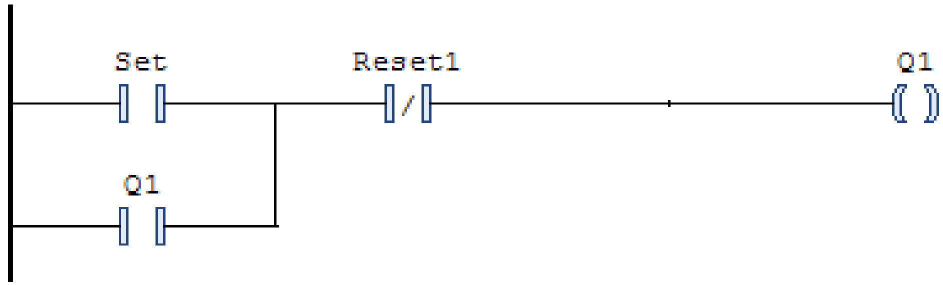
- When declaring we use the keyword **FUNCTION_BLOCK**.

- A FB will normally have one or more in-signals ("arguments") and one or out-signals. So, when declaring the variables, we must use the keywords VAR_INPUT and VAR_OUTPUT (in addition to VAR), in the declaration field. (A Timer for instance has the inputs IN, and PT, and the outputs Q and ET).
  - When calling the FB in a graphical language program, the block will automatically appear as a box with the number of inputs and outputs that you specified in the declaration of the FB.

Example: Your own 'RS-Toggle'

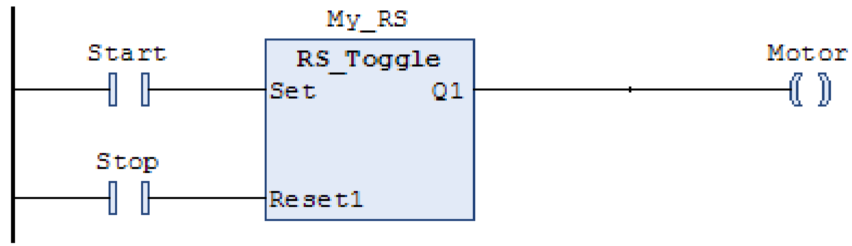
Example:
Make our own bi-stable FB of type RS-Toggle.
This FB has 2 inputs, Set and Reset1 and 1 output, Q1

```
FUNCTION_BLOCK RS_Toggle
VAR_INPUT
     Set      :BOOL;
     Reset1   :BOOL;
END_VAR
VAR_OUTPUT
     Q1       :BOOL;
END_VAR
VAR
END_VAR
```



Calling the FB:

```
PROGRAM Calls
VAR
     Start, Stop, Motor   :BOOL;
     My_RS    :RS_Toggle;
END_VAR
```

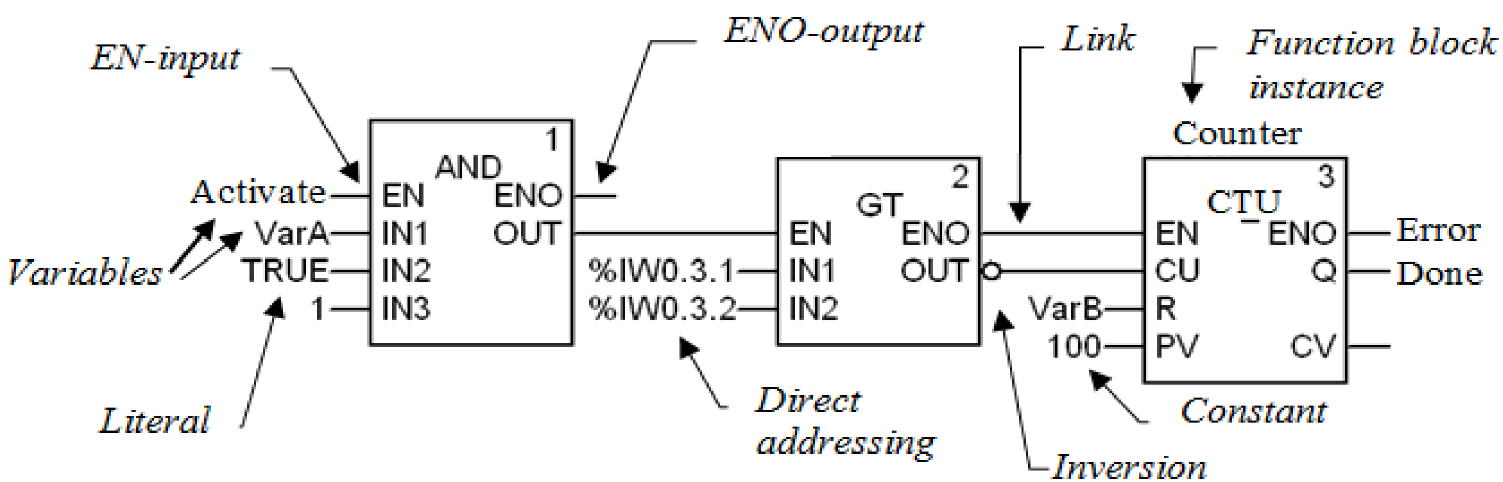


Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
  - LD – Ladder Diagram
  - FBD – Function Block Diagram
  - IL – Instruction List
  - ST – Structured Text
  - SFC – Sequential Function Chart
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

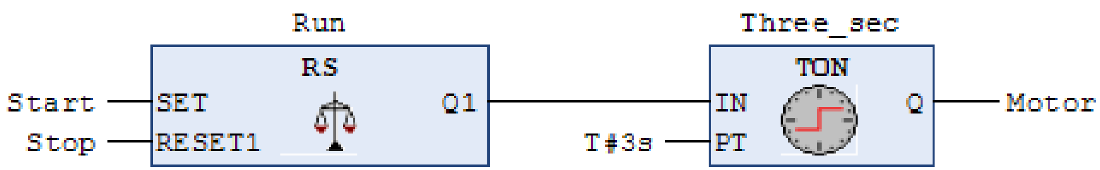
FBD – Function Block Diagram

- FBD language is graphic and follows the same guidelines that are specified for LD with respect to graphics and structure.
  - The structure of function block diagrams is consistent with the structure of Ladder diagrams with respect to graphic symbols, signal flow, order of execution and structuring of the code.
  - Contacts are not used in FBD, and neither are rails. Instead, variables are used directly as input arguments to the functions and function blocks. You can also use literals on input connections.



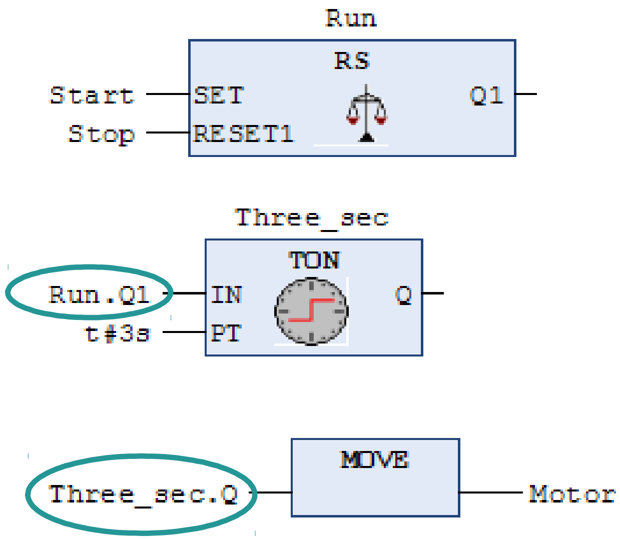
Timer – Example in FBD

```
PROGRAM TimEx1
VAR
    Start     AT %IX2.4    :BOOL;
    Stop      AT %IX2.5    :BOOL;
    Motor     AT %QX5.1    :BOOL;
    (*Declaring an instance of a RS and one TON-timer: *)
    Run            :RS;
    Three_sec      :TON;
END_VAR
```



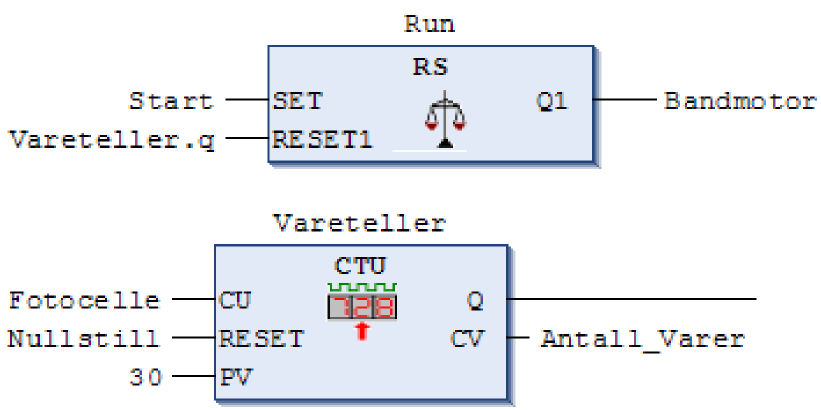
Timer – Example– Object referencing

```
PROGRAM Tim_objref
VAR
    Start     AT %IX2.4    :BOOL;
    Stop      AT %IX2.5    :BOOL;
    Motor     AT %QX5.1    :BOOL;
    Run                :RS;
    Three_sec          :TON;
END_VAR
```



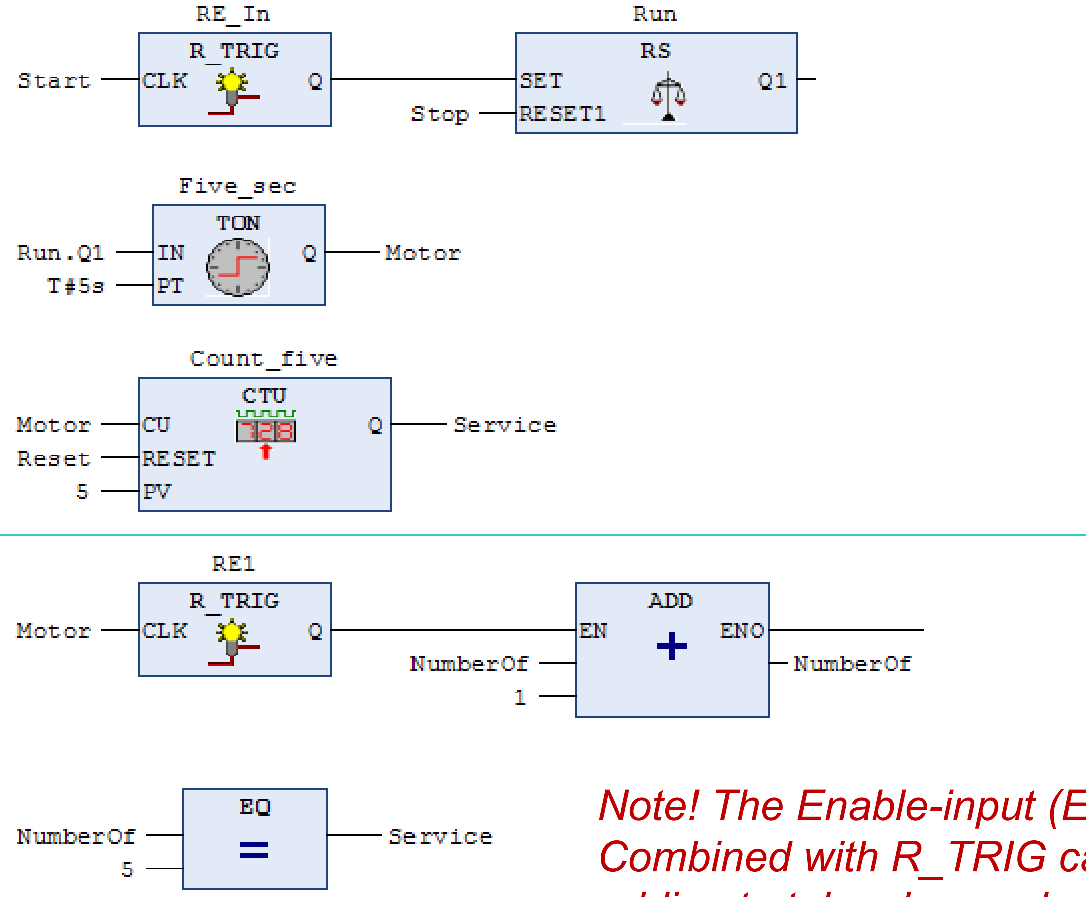
Counter– Example in FBD

```
PROGRAM Vareteller_fbd
VAR
    Fotocelle, Nullstill, Bandmotor :BOOL;
    Start    :BOOL;
    Antall_Varer     :USINT;
    Vareteller  :CTU; (*Deklarer en forekomst av en teller *)
    Run         :RS;  (*Deklarer en forekomst av en RS-vippe *)
END_VAR
```



Counting: 2 alternative methods

```
PROGRAM A_Counting
VAR
    RE_In      :R_TRIG;
    RE1        :R_TRIG;
    Run        :RS;
    Start      : BOOL;
    Stop       :BOOL;
    Motor      :BOOL;
    Five_sec      :TON;
    Count_five    :CTU;
    Reset      :BOOL;
    Service :BOOL;
    NumberOf:USINT;
END_VAR
```

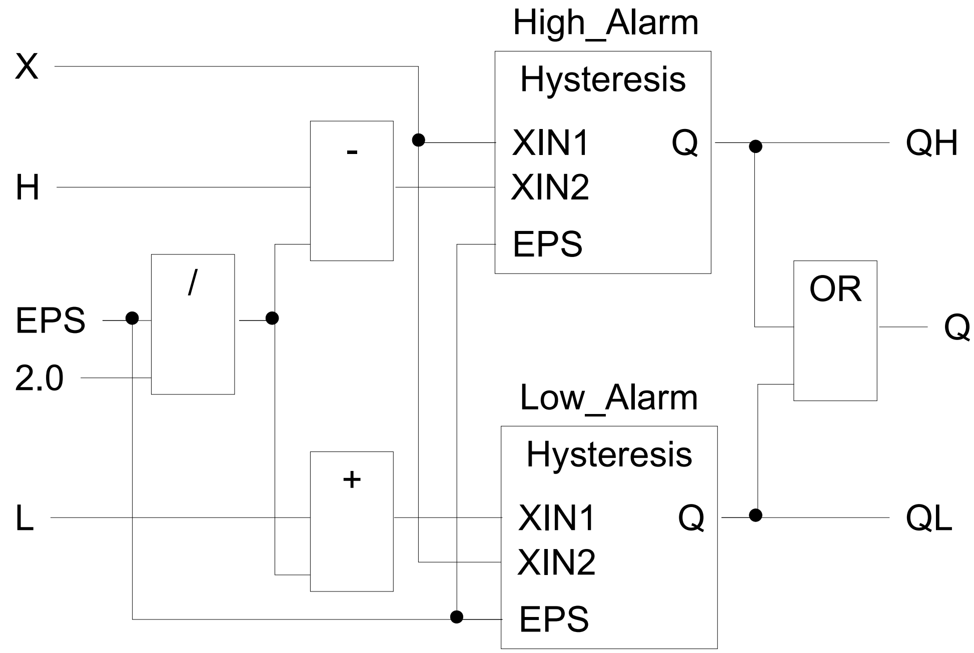


Note! The Enable-input (EN) Combined with R_TRIG causes the adding to take place only once each time the Motor starts.

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

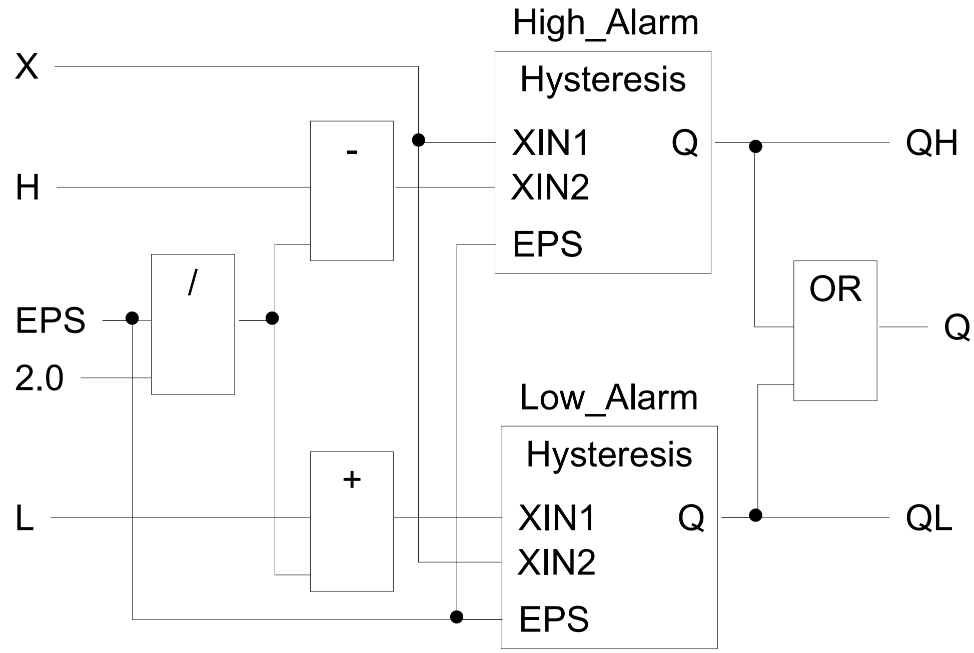
IEC 61131-3 Programming Languages
FBD – Function Block diagram

- An FBD graphically represents the signal/data flow through blocks.
- In an FBD, blocks may be either Functions and/or FB instances
- Signals and data flows may be of any data type!
- The body of a POU may consist of more than one network!



IEC 61131-3 Programming Languages
FBD – Function Block diagram

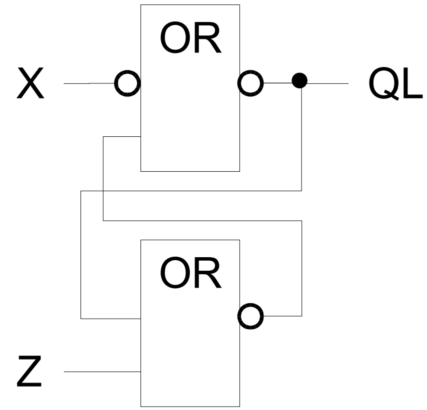
- FBD connection rules
  - Signals may diverge from one output to many inputs
  - Signals may not converge to a single input (i.e. wired OR, as used in LD, is not allowed in FBD)
  - Connected inputs and outputs must be of same data type.
  - IN_OUT parameters must be connected to a variable, and appear on the left of a FB.
  - Other Outputs and Inputs of blocks may be left un-connected

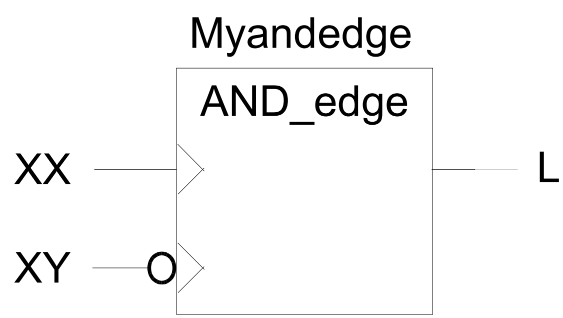


Mário de Sousa (msousa@fe.up.pt), 2011

IEC 61131-3 Programming Languages
FBD – Function Block diagram

- FBD symbols
  - Function Block Instances have their instance name displayed over the Block.
  - Inputs may be negated, without having to use an explicit NOT (circle at input or output)
  - Inputs with implicit edge detection (R_TRIG, F_TRIG) are shown with > and < respectively, or as defined in IEC617-12
  - Long lines may be replaced by connectors... (does not imply storage of signal state from one evaluation to the next of the FBD!)






```
Myandedge
+---------+
|AND_edge|
>         |
|         |
<         |
|         |
+---------+
```

>singal1>

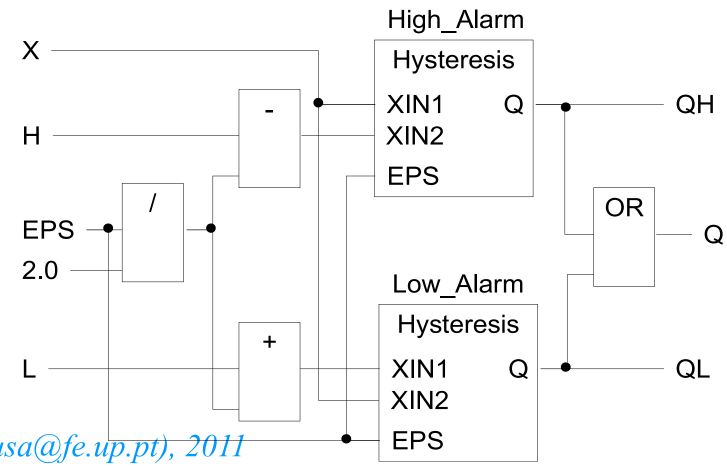
>singal1>

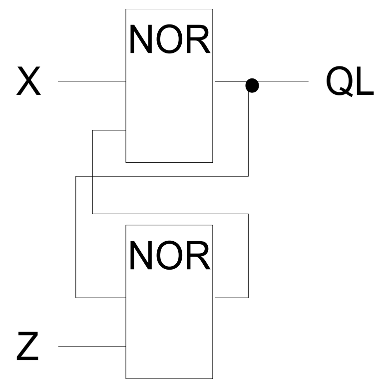
Mário de Sousa (msousa@fe.up.pt), 2011

IEC 61131-3 Programming Languages
FBD – Function Block diagram

- FBD execution
  - Signals flow from right side of blocks (outputs) to the left side of blocks (input).
  - All input values are updated/evaluated before any block is evaluated
  - Once a block starts being evaluated, all of its outputs must be updated before its evaluation is considered complete (i.e. blocks are evaluated one at a time).
  - The evaluation of a FBD is not complete until all outputs of all its blocks have been evaluated (i.e. all blocks in a network are always executed exactly once).

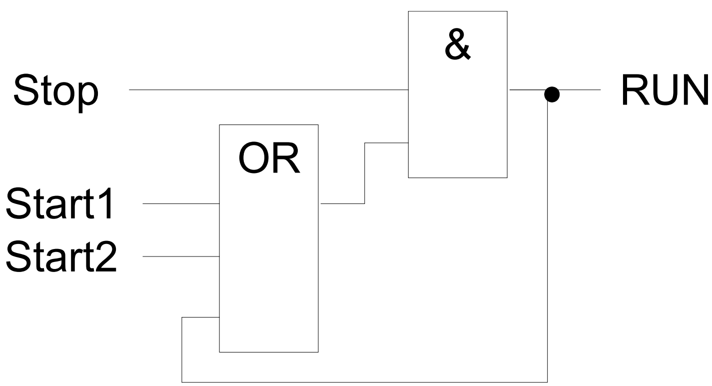
Following these rules, what would be the evaluation order of the following FBDs ?

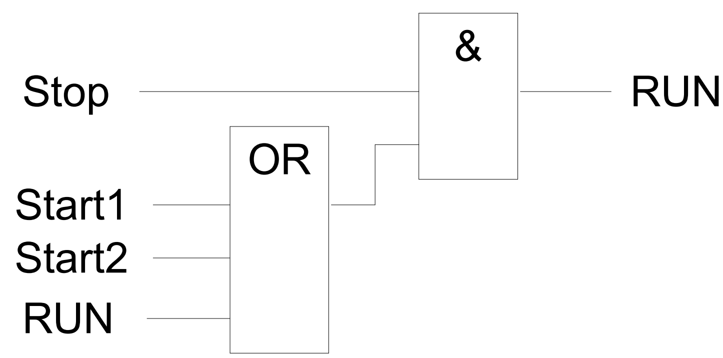




IEC 61131-3 Programming Languages
FBD – Function Block diagram

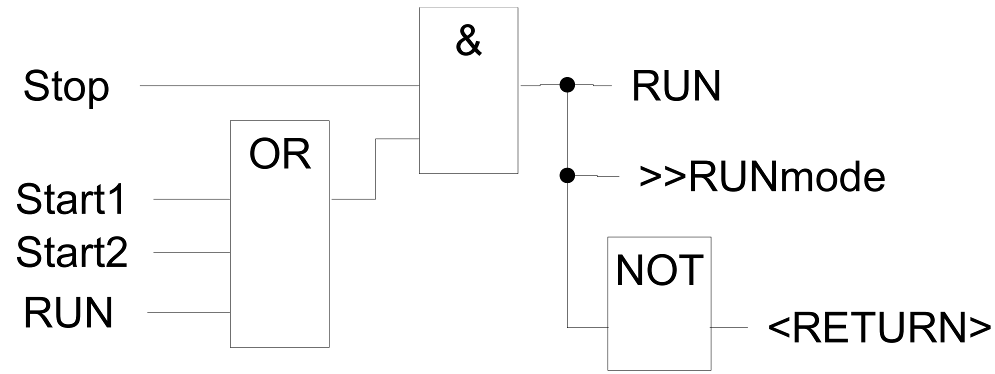
- FBD execution
  - Evaluation rules are not completely deterministic
  - It is possible to define other evaluation orders in an implementation dependent manner. (Usual default is from top to bottom, left to right – however this is not mandated by the standard).
  - Feedback loops may also be interrupted using variables...

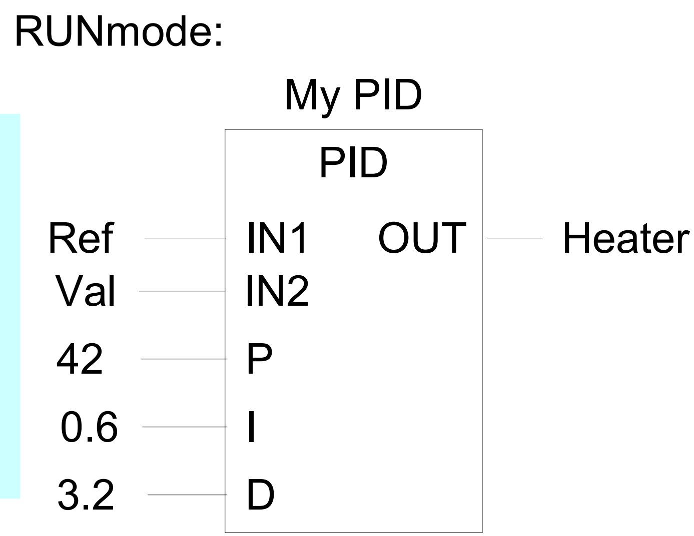




IEC 61131-3 Programming Languages
FBD – Function Block diagram

- FBD execution control
  - Explicit (conditional) jumps to labelled networks are supported (symbol: >>labelA)
  - Explicit (conditional) Return from POU are also supported. (symbol: <RETURN>





NOTE
All evaluation rules are still valid, including...

"The evaluation of a FBD is not complete until all outputs of all its blocks have been evaluated (i.e. all blocks in a network are always executed exactly once)."

Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

POUs (Program Organization Units)

Function

Function Block

Program

Configuration

Functions

- A function is a POU that can have one or more inputs and one or more outputs
  (IEC 61131-3 version 1 (1993) only supported a single output)

- They don't have internal memory and will therefore return the same value every times they are called with the same input arguments.

- The name of the functions acts as an variable, so that the output-value is returned in the functions name =>
  - This implies that a function must be declared with a data type

- Functions are particularly useful to perform mathematical calculations on also for string handling

Standard Functions: Arithmetic Oper.

Function name	Operator	ST expression
ADD	+	Addition: Out := IN1+IN2+…+INx;
SUB	-	Subtraction: Out := IN1 – IN2;
MUL	*	Multiplication: Out := IN1*IN2*…*INx;
DIV	/	Division: Out := IN1 / IN2;
MOD		Modulation: Out := IN1 MOD IN2;
EXPT	**	Eksponential: Out := IN1**IN2; (= $IN1^{IN2}$)
MOVE	:=	Assignment: Out := IN;

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Standard Functions: Comparisons

Name	Operator	Description (ST)
EQ	=	Out := (IN1=IN2) & (IN3=IN4) …
GT	>	Out := (IN1>IN2) & (IN2>IN3) & … & (INn-1>INx)
GE	>=	Out := (IN1>=IN2) & (IN2>=IN3) & … & (INn-1>=INx)
LT	<	Out := (IN1<IN2) & (IN2<IN3) & … & (INn-1<INx)
LE	<=	Out := (IN1<=IN2) & (IN2<=IN3) & … & (INn-1<=INx)
NE	<>	Out := IN1<>IN2

Standard Functions: many more...

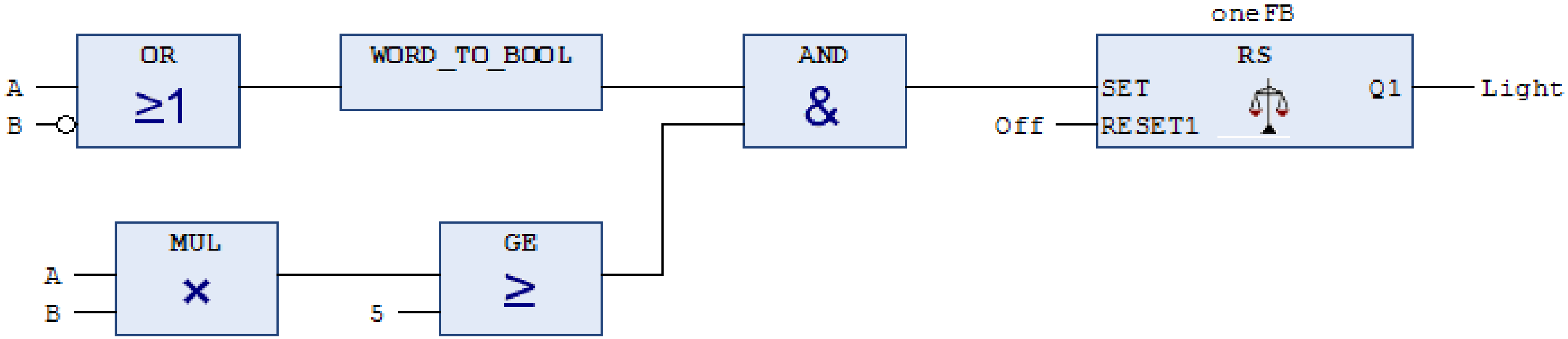
- Some more standard functions:
  - Boolean: AND, OR, …
  - Mathematical oper: ADD, SUB, SQRT, SIN, COS, ...
  - Comparison: GT, LT, MIN, MAX, …
  - String: LENGTH, …
  - Conversion: WORD_TO_INT, TIME_TO_LONG, ...

- Some of these functions may have extendable number of inputs!

- In Structured Text (ST), many of these functions are implemented as operators:
  - +, -, *, /, :=, <, >, <=, >=, <>

Standard functions in FBD

```
PROGRAM F_FunInFBD
VAR
    On, Off, Light   :BOOL;
    A, B       : WORD;
    oneFB      : RS;
END_VAR
```



Hmmm... There is something not quite right here!
It is not legal to multiply (MUL) two variables of type WORD.

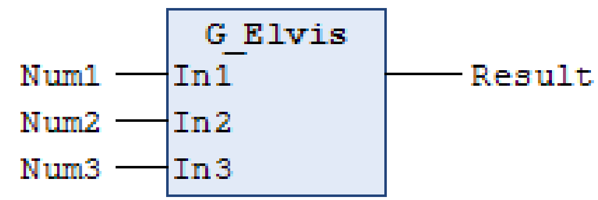
User Defined Functions

- Defining the Function

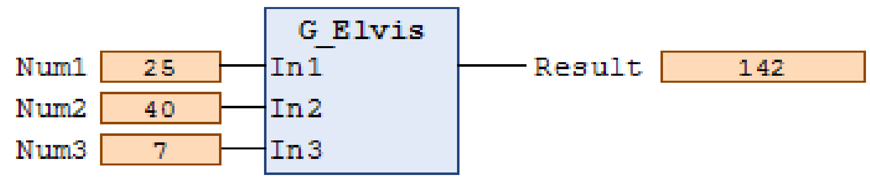
```
FUNCTION G_ELVIS: INT
VAR_INPUT
    In1, In2, In3 : INT;
END_VAR

G_ELVIS := 70 + In1 + In2 + In3;

END_FUNCTION
```

- Calling the Function (in FBD):

```
PROGRAM G_Calling_Elvis
VAR
    Num1     :INT := 25;
    Num2     :INT := 40;
    Num3     :INT := 7;
    Result   :REAL;
END_VAR
```

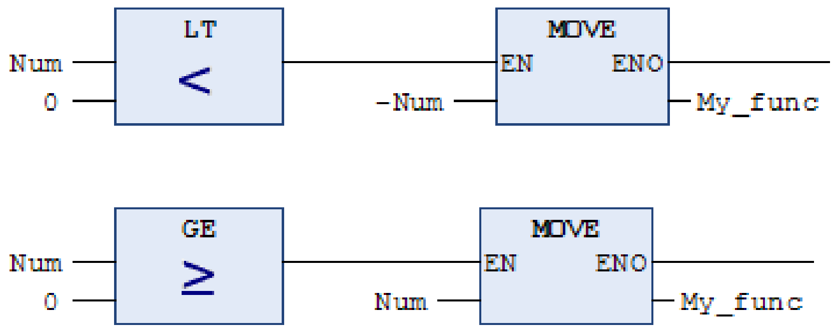


- In Run-time:



Another Function example

```
FUNCTION My_func : INT
VAR_INPUT
        Num :INT;
END_VAR
```



ST-code:

```
IF Num < 0 THEN
    My_func_ST := -Num;
ELSE
    My_func_ST := Num;
END_IF;
```

Yet another Functions-example

```
FUNCTION Find_roots : REAL
(* Declarations *)
VAR_INPUT
    A, B, C: REAL;
END_VAR
VAR
    Root: REAL;
    Nroots: USINT;
END_VAR
VAR_OUTPUT
    X1, X2: REAL;
    Info: STRING;
END_VAR
```

Note! Multiple outputs is not allowed in version 1 of the standard

```
(* Function code *)
Root := B*B -4*A*C;
IF Root < 0.0 THEN
    Nroots := 0;
    X1 := X2 := STRING_TO_REAL('NaN');
    Info := 'No real roots';
ELSIF Root = 0.0 THEN
    Nroots := 1;
    Info := 'Concurrent roots';
    IF a <> 0 THEN
        X1 := X2 := (-B+SQRT(Root)) / (2*A);
    ELSE
        X1 := X2 := 0;
    END_IF
ELSE
    Nroots := 2;
    X1 := (-B-SQRT(Root)) / (2*A);
    X2 := (-B+SQRT(Root)) / (2*A);
    Info := 'Two real roots';
END_IF;
```

Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

POUs (Program Organization Units)

Function

Function Block

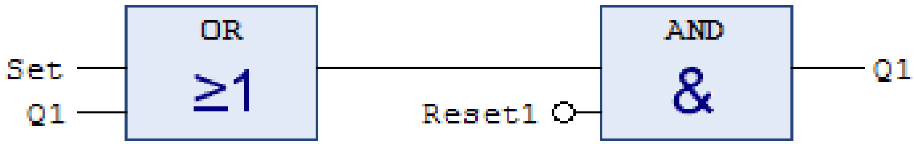
Program

Configuration

User-defined FBs in FBD

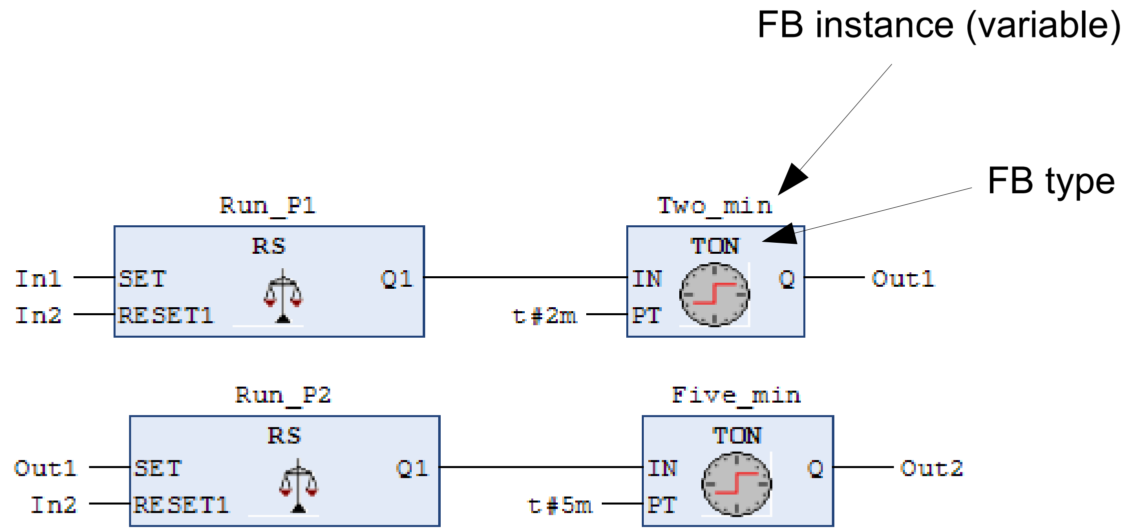
- The FB in this example has 2 input-variables and 1 output-variable, all of data type BOOL.
- You can clearly see the similarity with LD-code.

```
FUNCTION_BLOCK My_RS

VAR_INPUT    (* In-variable *)
    Set: BOOL;
    Reset1: BOOL;
END_VAR

VAR_OUTPUT   (* Out-variable *)
    Q1: BOOL;
END_VAR
```



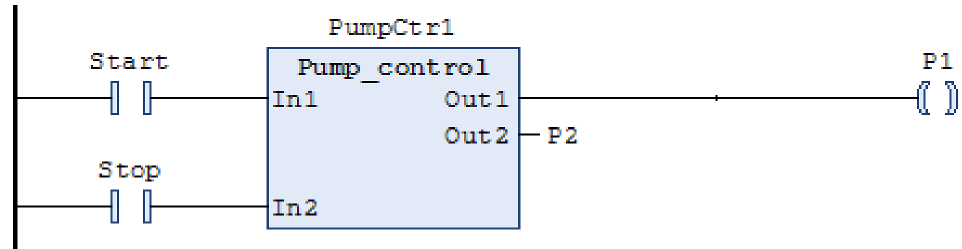
User-defined FB

```
FUNCTION_BLOCK Pump_control
VAR_INPUT
    In1, In2        :BOOL;
END_VAR
VAR_OUTPUT
    Out1, Out2      :BOOL;
END_VAR
VAR
    Run_P1, Run_P2      :RS;
    Two_min, Five_min   :TON;
END_VAR
```



- As we have seen, when using (calling) a FB in a graphical language, a rectangular symbol will be generated automatically, with the name of the FB, and the number of inputs and outputs that was declared using VAR_INPUT and VAR_OUTPUT.

```
PROGRAM X_Calling
VAR
    Start, Stop         :BOOL;
    P1, P2              :BOOL;
    PumpCtr1          : Pump_control;
END_VAR
```

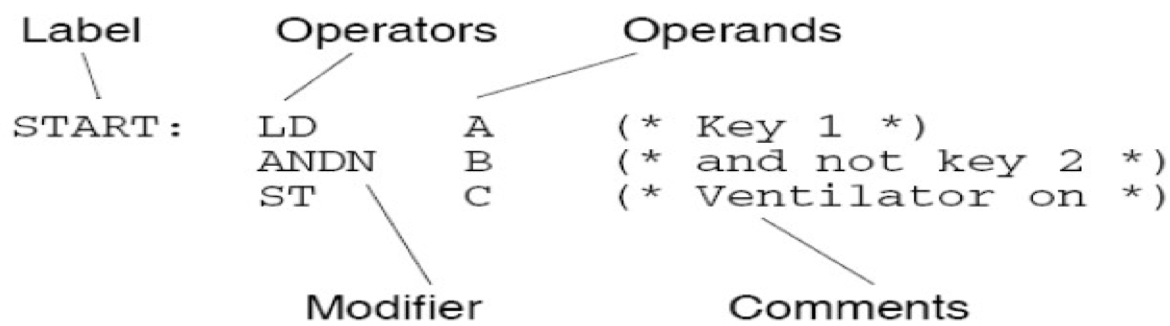


Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
  - LD – Ladder Diagram
  - FBD – Function Block Diagram
  - IL – Instruction List
  - ST – Structured Text
  - SFC – Sequential Function Chart
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

IEC 61131-3 Standard Architecture

- IL – Instruction List
  - Mainly for historical reasons
  - Somewhat cryptic, even though powerful...



  - Supports parer

```
LD %IX0
AND(
   LD %IX1
   OR %IX2
)
```

```
LD    %IX0
AND(  %IX1
   OR %IX2
)
```

IEC 61131-3 Standard Architecture

- IL

Operator	Modifier	Operation
LD	N	Set current result equal to operand
ST	N	Store current result to operand location
S		Set operand to 1 if current result is Boolean 1
R		Reset operand to 0 if current result is Boolean 1
AND	N	(Logical AND
&	N	(Logical AND
OR	N	(Logical OR
XOR	N	(Logical Exclusive OR
NOT		Logical Negation (one's complement)
ADD	(	Addition
SUB	(	Subtraction
MUL	(	Multiplication
DIV	(	Division
MOD	(	Modulo-Division

IEC 61131-3 Standard Architecture

- IL

Operator	Modifier	Operation
MOD	(	Modulo-Division
GT	(	Comparison: >
GE	(	Comparison: >=
EQ	(	Comparison: =
NE	(	Comparison: <>
LE	(	Comparison: <=
LT	(	Comparison: <
JMP	C N	Jump to label
CAL	C N	Call function block (See table 53)
RET	C N	Return from called function, function block or program
)		Evaluate deferred operation

Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
  - LD – Ladder Diagram
  - FBD – Function Block Diagram
  - IL – Instruction List
  - ST – Structured Text
  - SFC – Sequential Function Chart
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

IEC 61131-3 Programming Languages
ST – Structured Text

- An ST program is a sequence of any of the following statements:
  - Assignment statement
  - FB invocation
  - Control Flow Statements
  - Iteration Statements

- In any of the above statements, we may find
  - expressions

```
PROGRAM Foo
  VAR
    light : BOOL;
    counter : count_up;
    I, X, Y, Z: INTEGER;
  END_VAR

  (* Program Body in ST *)
  light := NOT light;
  counter (light);
  IF (counter.count > 30-X)
    THEN ...
  END_IF
  FOR I:=Y*fact(z) TO 8+Z-y DO
    ...
  END_FOR
END_PROGRAM
```

IEC 61131-3 Programming Languages
ST – Structured Text

- Assignment Statements:

  variable := expression

> **The variable and the expression MUST be of the same data type!**

- Expression:
  - Defined recursively either as (binary expression)

    expression <operator> expression
  - or... (unary expression)

    <operator> expression
  - or a... variable, function invocation, constant, enumerated value

```
PROGRAM Foo
  VAR
    light : BOOL;
    counter : count_up;
    I, X, Y, Z: INTEGER;
  END_VAR

  (* Program Body in ST *)
  light := NOT light;
  counter (light);
  IF (counter.count > 30-X)
    THEN ...
  END_IF
  FOR I:=Y*fact(z) TO 8+Z-y DO
    ...
  END_FOR
END_PROGRAM
```

IEC 61131-3 Programming Languages
ST – Structured Text

- Assignment Statements:
  variable := expression

- Expression:
  - Defined recursively either as
    (binary expression)
    expression <operator> expression
  - or...
    (unary expression)
    <operator> expression
  - or a...
    variable, function invocation, constant, enumerated value

- Unary Expressions
  - -   (<=> multiply by -1)
  - NOT   (boolean negation)

- Binary Expressions
  (in order of decreasing precedence)
  - **   (power expression)
  - *, /, MOD   (multiplication, division, modulo operation)
  - +, -   (addition, subtraction)
  - <, >, <=, >=   (comparison operations)
  - =, <>   (equal, not equal)
  - &, AND   (boolean AND)
  - XOR   (boolean exclusive OR)
  - OR   (boolean OR)

IEC 61131-3 Programming Languages
ST – Structured Text

- FB Invocation statements:
  - Formal Invocation

```
FBinstance(
    input_par := expression,
    output_par => variable,
    NOT output_par => variable
)
```

In any order!

  - Non-Formal Invocation

```
FBinstance(
    expression,
    variable,
    variable
)
```

Same order as declared in FB being invoked.

Input parameters!

Output parameters!

```
FUNCTION_BLOCK PID_t
  VAR_INPUT Error: Real; END_VAR
  VAR_OUTPUT out: Real; END_VAR
  VAR_INPUT P, I, D: Real; END_VAR
  ...
END_FUNCTION_BLOCK

PROGRAM Oven
  ...
  PID(P:=42, I:=4, D:=3,
      Error:=Err, Out=>PWM_Out);
  PID(Err, PWM_Out, 42, 4, 3);
  PID(Err);
  pwm_out := PID.out;
  ...
END_PROGRAM
```

The syntax for Function Invocation is the same as that for FBs!

IEC 61131-3 Programming Languages
ST – Structured Text

- Control Flow Statements:
  - RETURN
  - IF boolean_expression THEN
    statement_list
    ELSIF boolean_expression THEN (optional)
    statement_list
    ELSE (optional)
    statement_list
    END_IF
  - CASE expression OF
    elem1 : statement_list
    elem2 .. elem3 : statement_list
    elem3, elem4 : statement_list
    ELSE statement_list (optional)
    END_CASE

```
PROGRAM Foo
  VAR
    B1, B2 : BOOL;
    I, X, Y, Z: INTEGER;
  END_VAR

  (* Program Body in ST *)
  IF B1 XOR B2
    THEN x:=9;
  END_IF

  CASE I/4 OF
    1: x:=42;
    2,4..7: x:=y+z;
  ELSE ...
  END_CASE
END_PROGRAM
```

IEC 61131-3 Programming Languages
ST – Structured Text

- Iteration Statements:
  - FOR variable := expression TO expression $\underbrace{\text{BY expression}}_{\text{optional}}$ DO
    statement_list
    END_FOR
  - WHILE boolean_expression DO
    statement_list
    END_WHILE
  - REPEAT
    statement_list
    UNTIL boolean_expression
    END_REPEAT

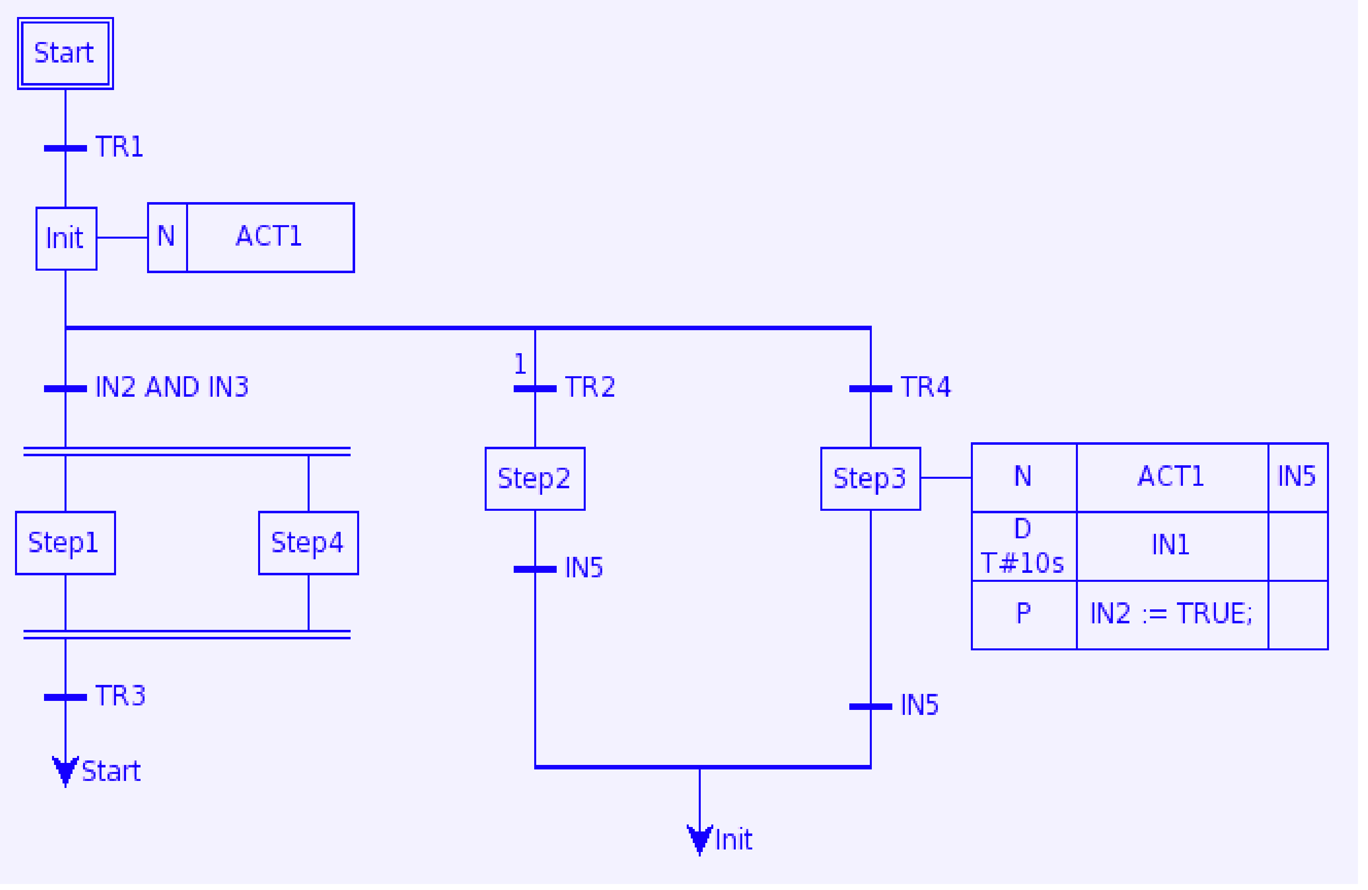
```
PROGRAM Foo
   ...
  FOR I:=Y*fact(z) TO 8+Z-y BY -2 DO
    ...
  END_FOR

  WHILE %IX4.3 DO
    ...
  END_WHILE

  REPEAT
    ...
  UNTIL %IX4.3 AND NOT %IX2.4
END_PROGRAM
```

Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
  - LD – Ladder Diagram
  - FBD – Function Block Diagram
  - IL – Instruction List
  - ST – Structured Text
  - SFC – Sequential Function Chart
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

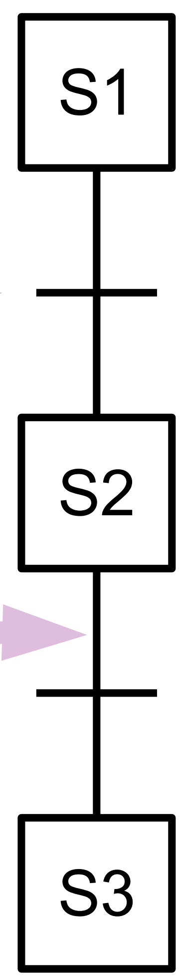
IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- An SFC represents a state machine
- Multiple states may be active at any one time
- Sequence of event activation is defined by directed links
- Transitions define when the sequences should occur
- SFC is based on the Grafcet standard IEC 60848.
  (the IEC 61131-3 standard itself references IEC 60848).



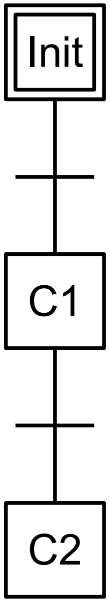
IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

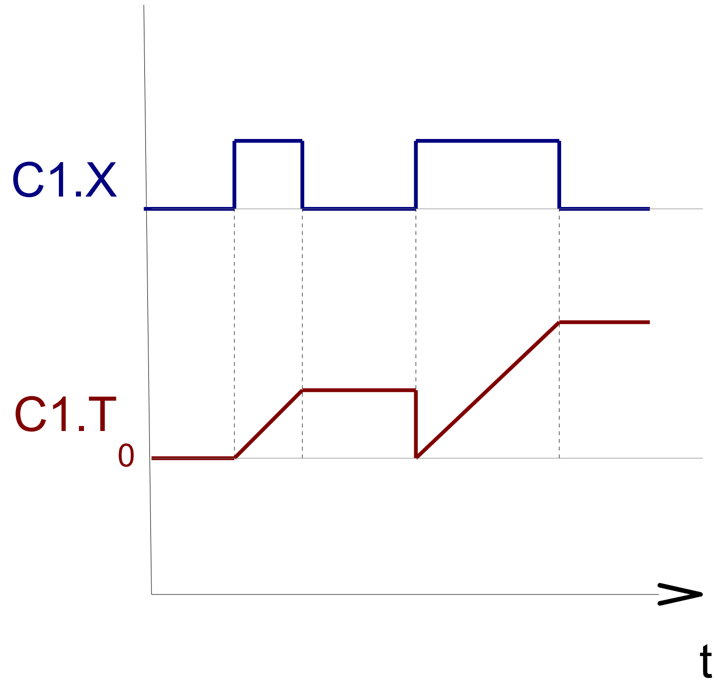
- An SFC is composed by:
  - Steps
    - Represent the system state
    - At any given time, every step is either active or inactive.
    - Multiple steps may be active simultaneously
    - Each step is given a unique identifier (same rules as variable names)
  - Transitions
    - Represent conditions that define when the SFC's state will evolve
  - Directed Links
    - Define sequence of step activation/deactivation when a transition fires
    - Always go from top to bottom (unless otherwise specified)
    - Always enter a step from the top and leave from the bottom



IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Steps
  - Initial step...
    - Is the step that is active when the SFC is first started
    - Represented by a double edged box
    - Each SFC must have exactly one initial step.
  - Associated with each step there is a boolean variable step_name.X
    - Will have the value TRUE when the step is active, and FALSE otherwise
    - eg. C1.X, Init.X
  - Associated with each step there is a variable step_name.T of type TIME
    - Will store how long the step has been active
    - eg. C1.T, Init.T





Scope of automatic variables S.X and S.T is limited to POU containing the SFC.

Automatic variables S.X and S.T are read-only

IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Examples of permitted use of step variables

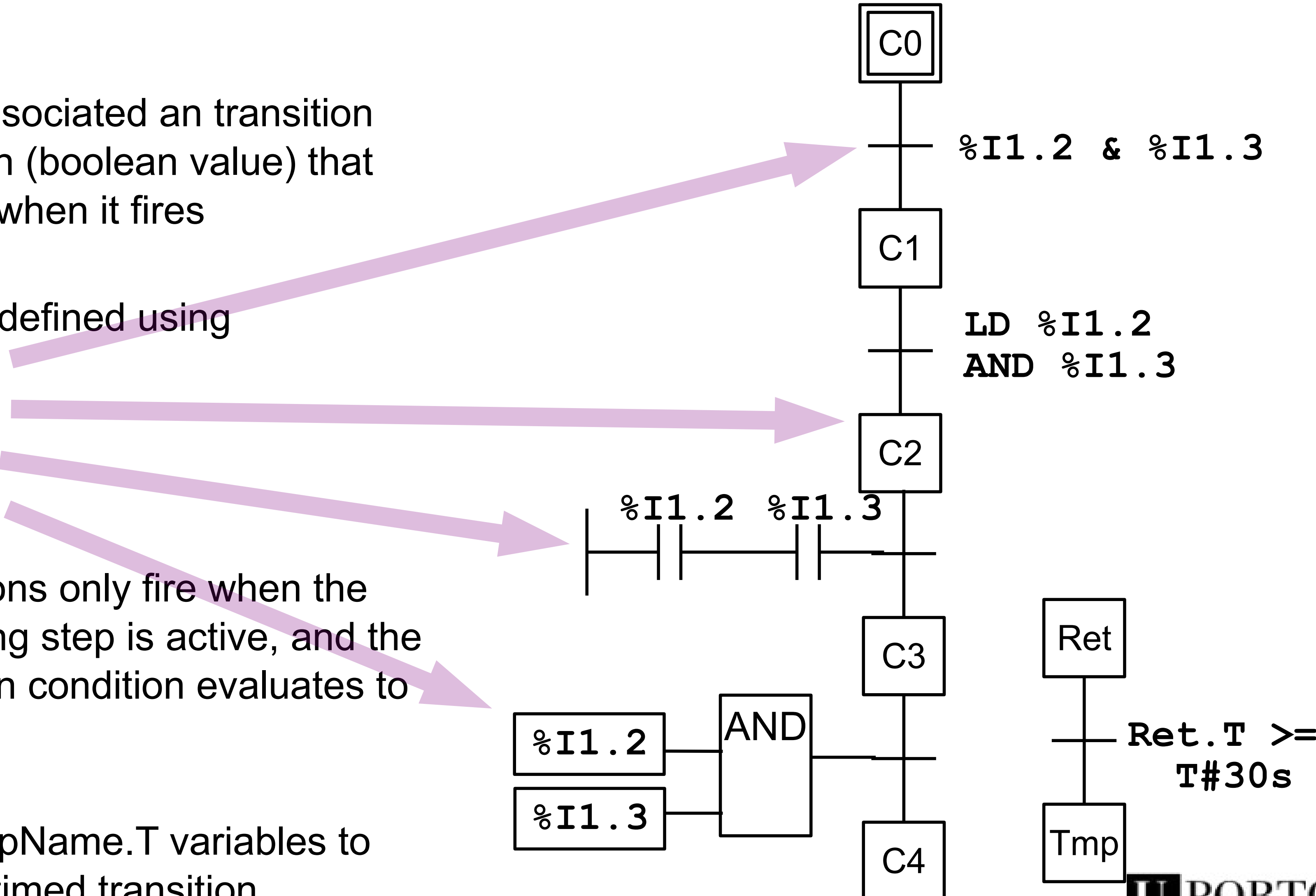
```
IF Stir.X OR (Drain.X AND %IX1.0) THEN
    %QX2.5 := TRUE;
ELSE
    %QX2.5 := FALSE;
END_IF;
WHILE Warm_up.T < t#20s DO
    Calculate;
END_WHILE;
```

- Examples of illegal use of step variables

```
IF Stir.X OR (Drain.X AND %IX1.0) THEN
    Warm_up.X := 1;      ←Illegal
END_IF;
IF %IX1.12 THEN
    Fill.T := t#45s;     ←Illegal
END_IF;
```

IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Transitions
  - Have associated an transition condition (boolean value) that defines when it fires
  - May be defined using
    - ST
    - IL
    - LD
    - FBD
  - Transitions only fire when the preceding step is active, and the transition condition evaluates to TRUE.
  - Use StepName.T variables to specify timed transition

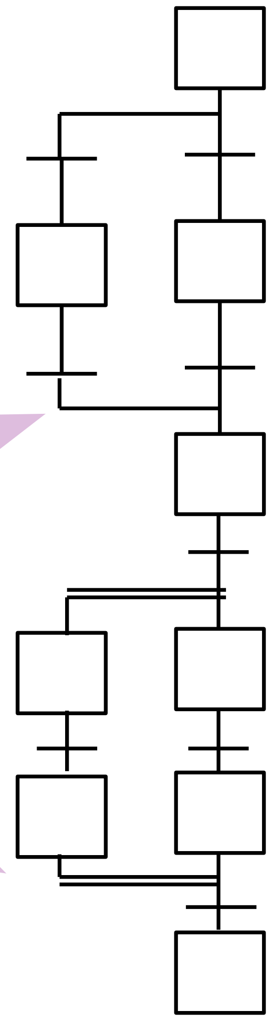


IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Directed Arcs
  - Define the step activation sequence

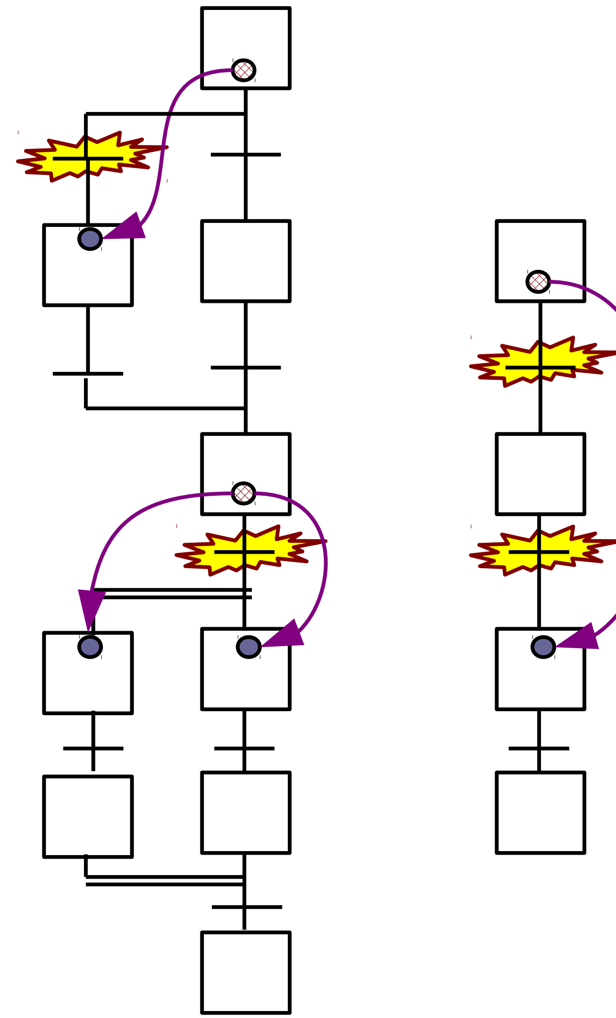
  - Basic structure rules:
    - Between any two steps, we must find a transition
    - Between any two transitions, we must find a step

    - Or sequences always have more than one transition
    - And sequences only have one transition



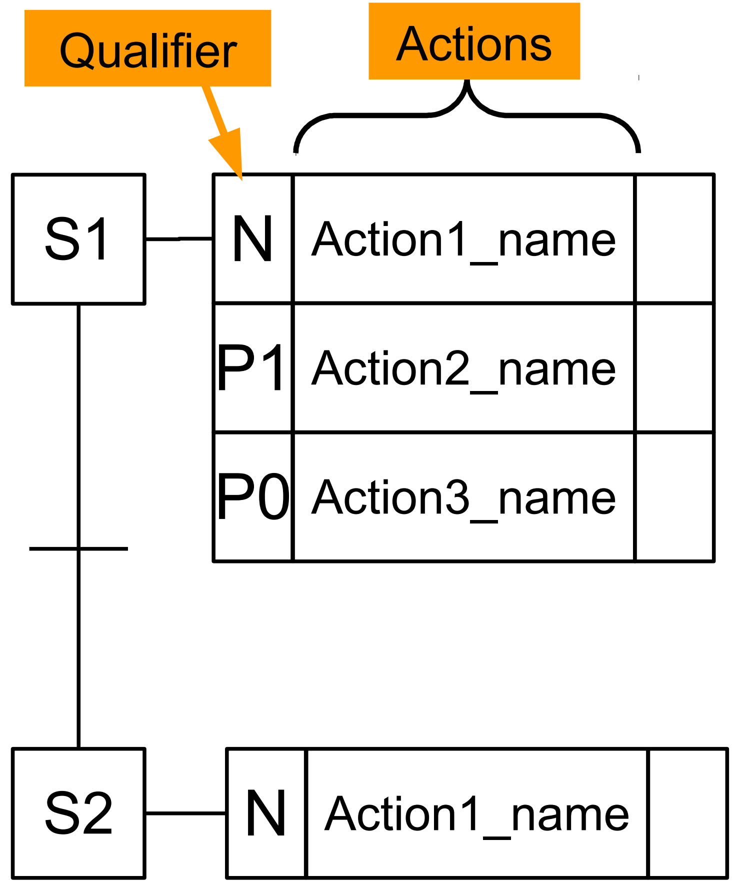
IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Rules of Evolution
  - When a transition fires...
    - De-activate all inbound steps
    - Activate all outbound steps
  - In a divergence situation, it is an error if multiple transitions may fire simultaneously.
  - When several transitions may be fired simultaneously, these are fired simultaneously (within the timing constraints of the PLC).
    - Few IEC61131-3 implementations follow this rule.



IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Actions
  - We can associate 0, 1 or more actions to each step
  - Execution of Actions will depend on whether the associated step is active, and its qualifier.
  - Actions will reference code written in IL, LD, ST, FBD, SFC, or simply a variable (POU output or internal variable)
  - Automatic variables (S1.X, S1.T) may be used inside the actions.



Actions written in SFC allow the definition of hierarchical SFCs!!

Mário de Sousa (msousa@fe.up.pt), 2011

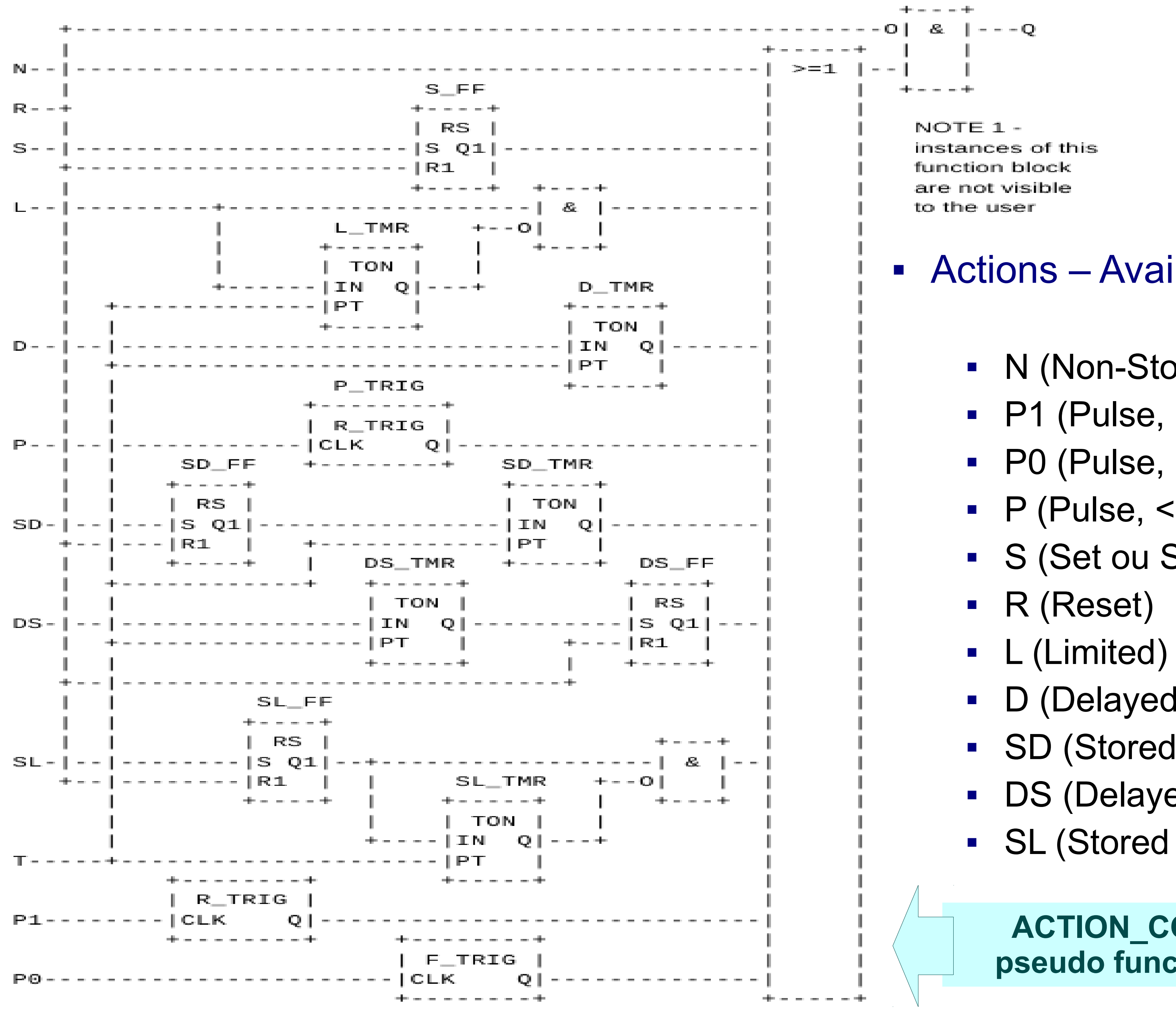


Figure 15b - ACTION_CONTROL function block body without "final scan" logic

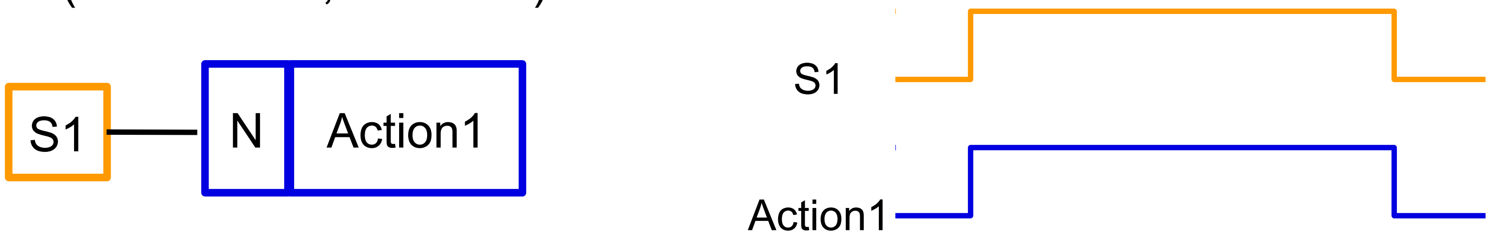
Actions – Available Qualifiers

- N (Non-Stored)
- P1 (Pulse, subida)
- P0 (Pulse, descida)
- P (Pulse, <=> to P1)
- S (Set ou Stored)
- R (Reset)
- L (Limited)
- D (Delayed)
- SD (Stored and time Delayed)
- DS (Delayed and Stored)
- SL (Stored and time Limited)

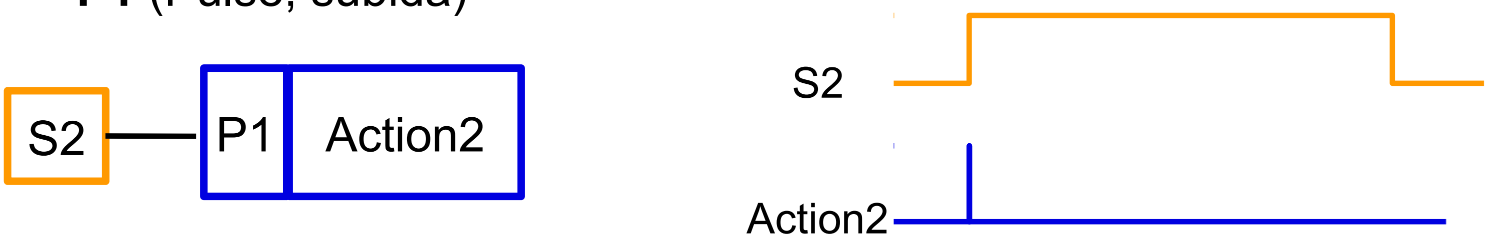
ACTION_CONTROL pseudo function block

SFC – Action Qualifiers

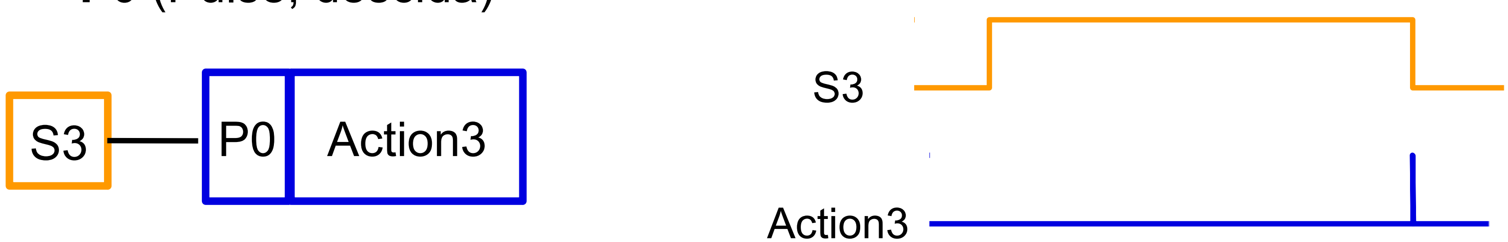
N (Non-Stored, contínua)



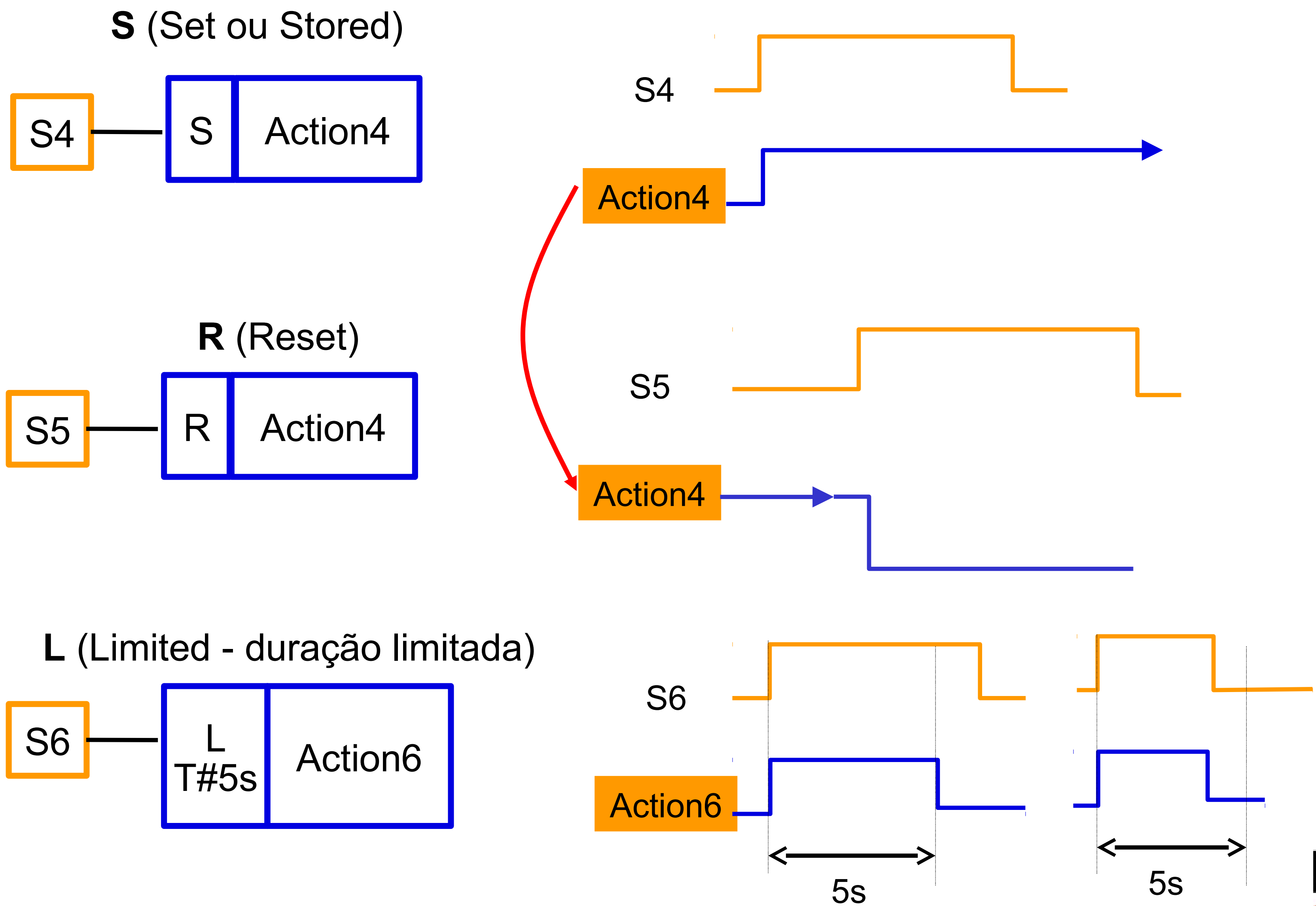
P1 (Pulse, subida)



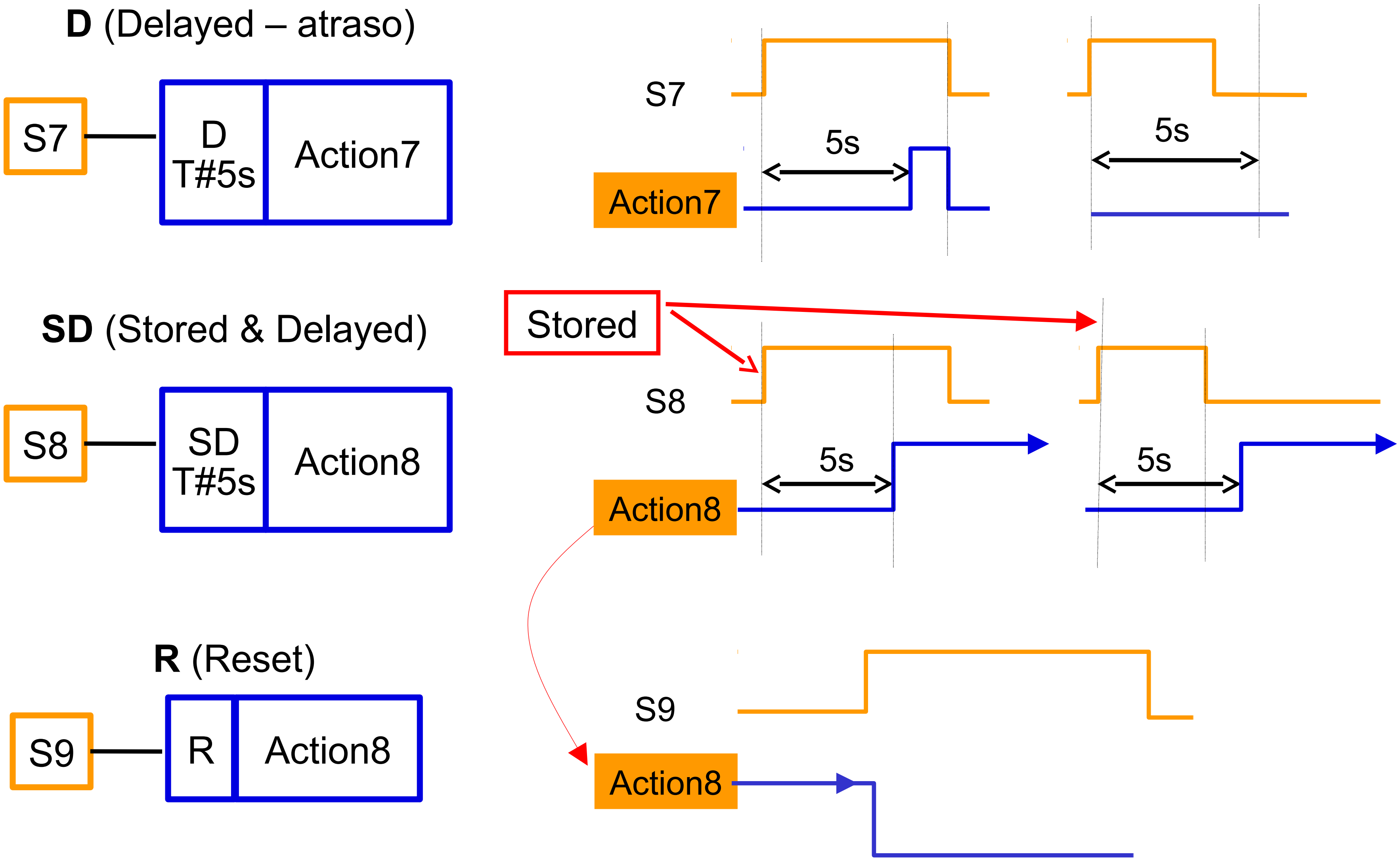
P0 (Pulse, descida)



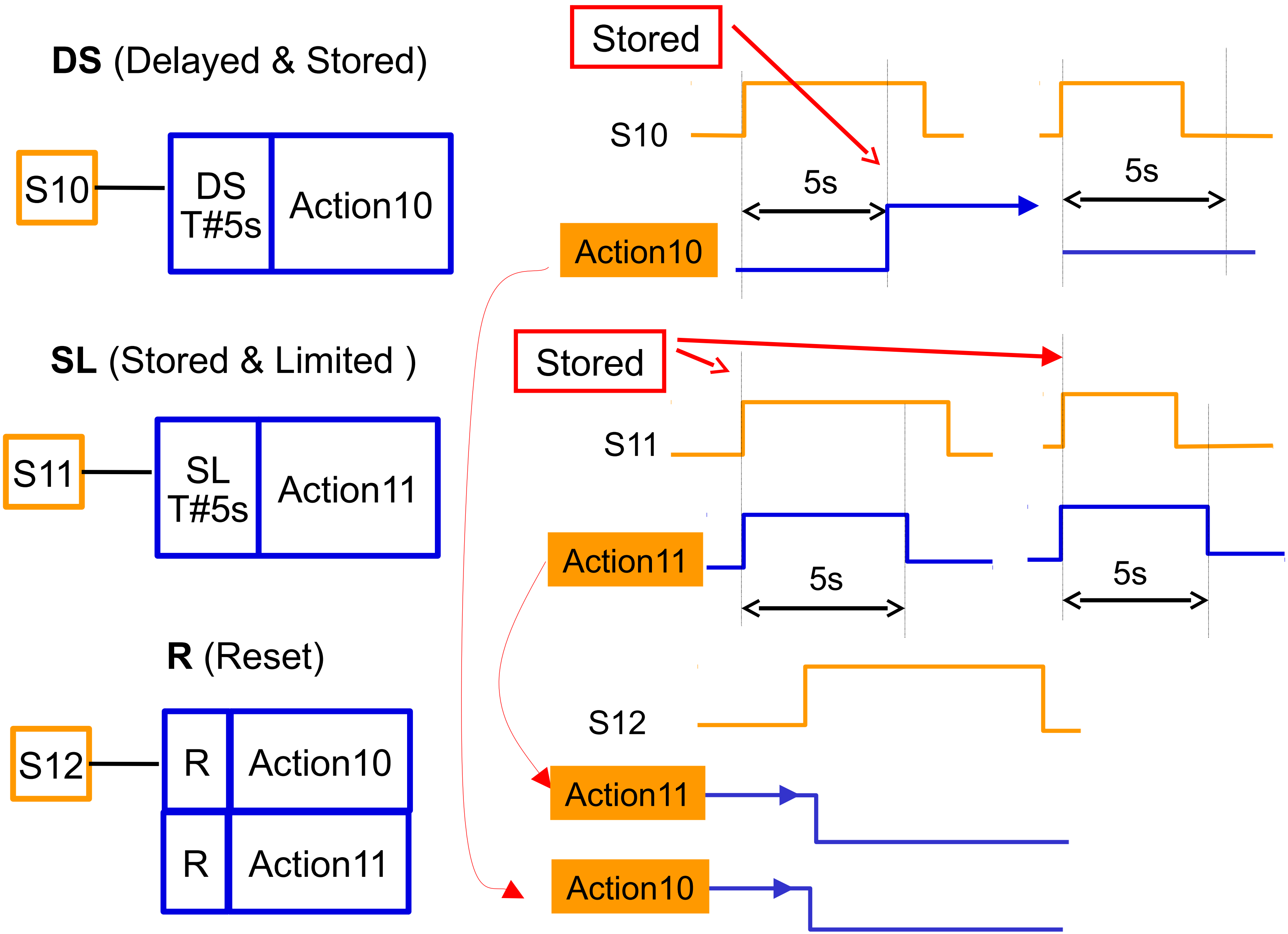
SFC – Action Qualifiers



SFC – Action Qualifiers

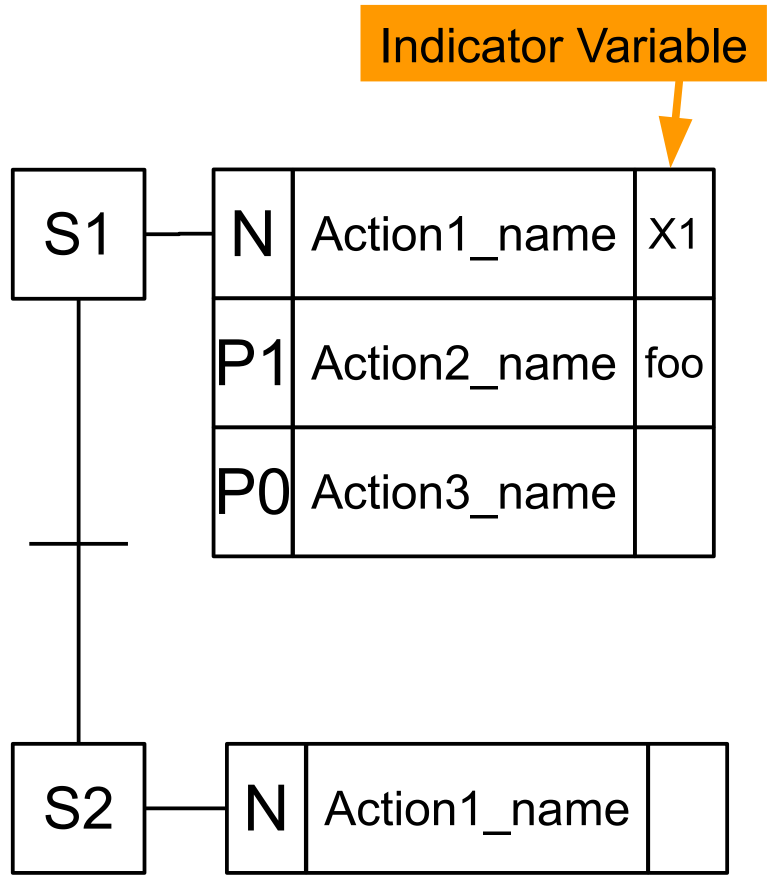


SFC – Action Qualifiers



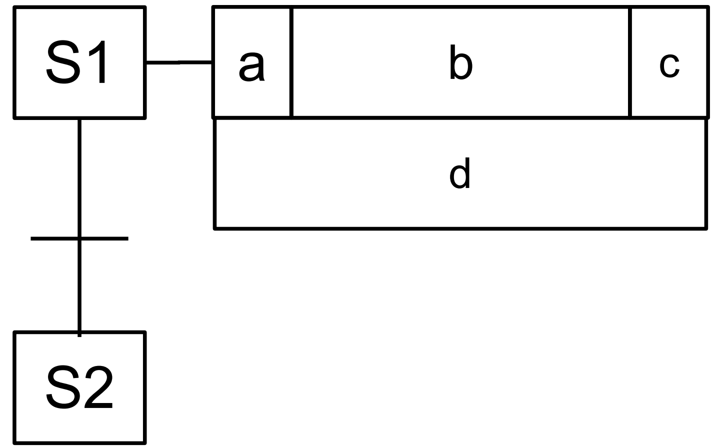
IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Indicator Variable
  - Each action block may be associated with a boolean Indicator Variable
  - action block is active → Indicator Variable becomes TRUE
  - action block is inactive → Indicator Variable becomes FALSE
  - The same variable may be used as Indicator variable in several action blocks, but standard does not mandate resulting boolean value.



IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Action Blocks
  - a: Qualifier
  - b: action name
  - c: boolean indicator
    - Variable that will mimic the value of the Action_Control block's output.
  - d: Action, using one of the following languages → IL, ST, LD, FBD, SFC.

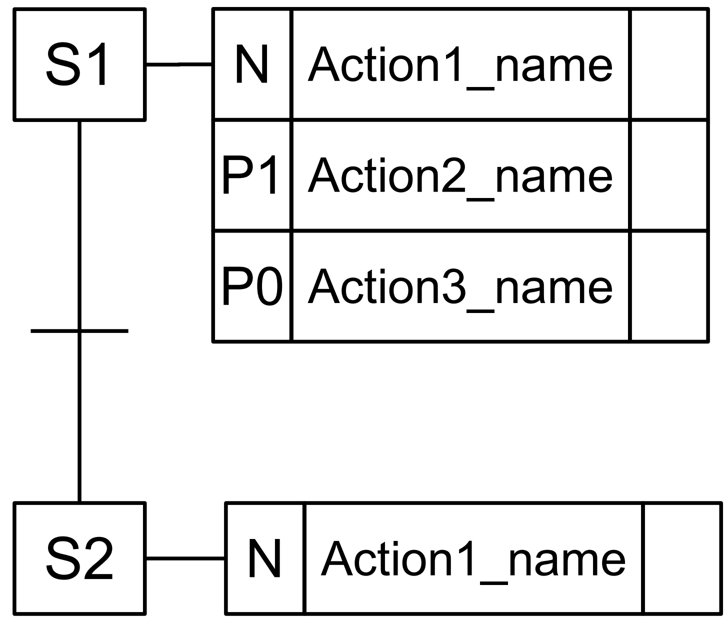


IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Action Blocks

  - The same action action may be used in several distinct action blocks!
    - The Boolean input to the action's ACTION_CONTROL block shall have the value BOOL#1 if it has at least one active association, and the value BOOL#0 otherwise.

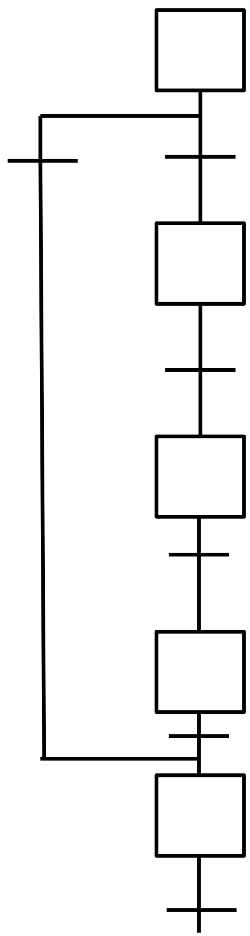
  - Each action has an automatic boolean variable **Action_Name.Q** that will mimic the value of the Action_Control_Block's output.



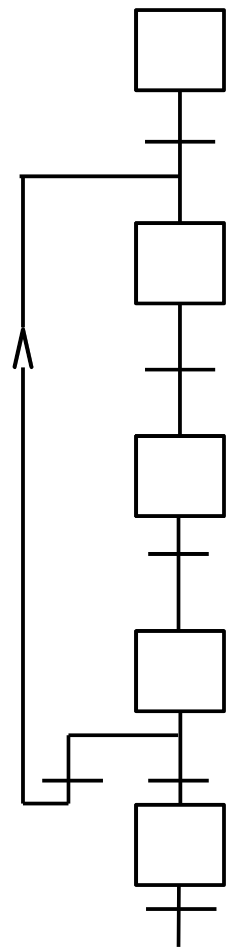
An action may use the condition action_name.Q=FALSE to determine that it is being executed for the final time during its current activation.

IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

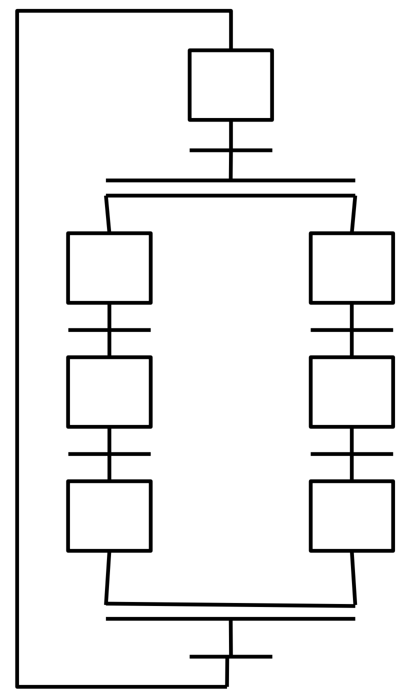
Sequence Skip



Loop

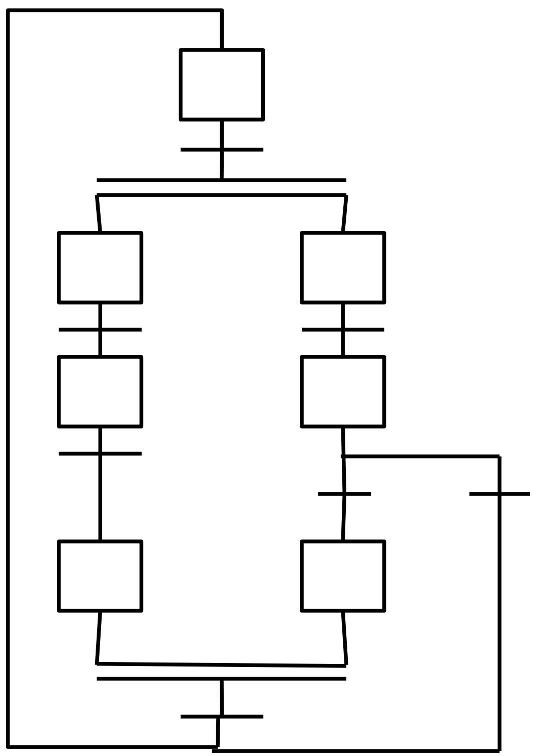


Parallelism / Concurrency

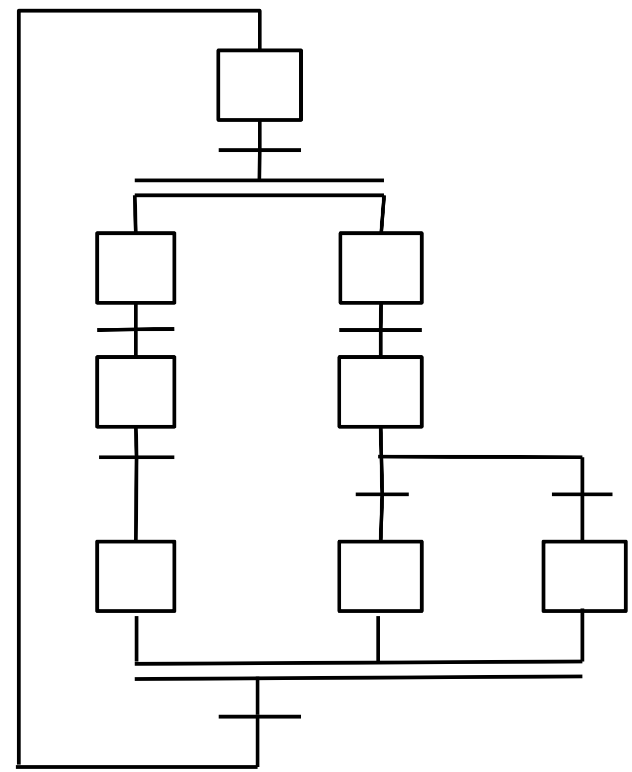


IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

An unsafe SFC

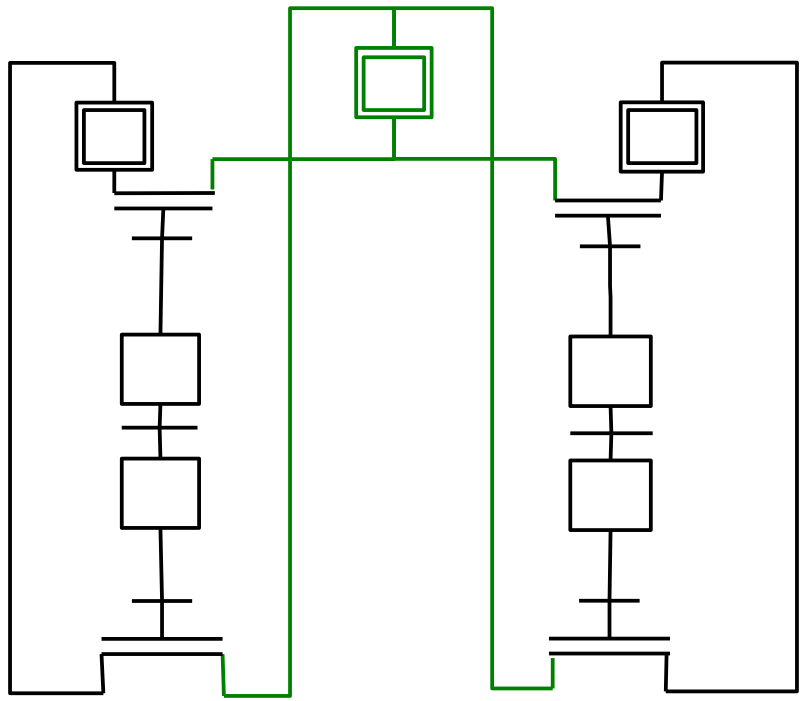


An unreachable SFC



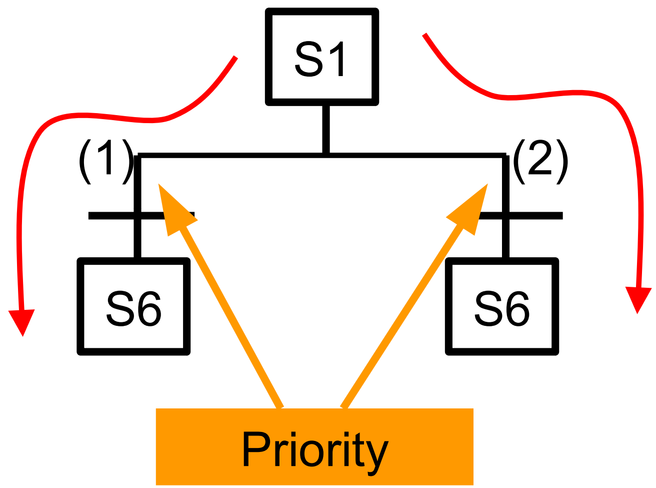
IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

Resource Sharing

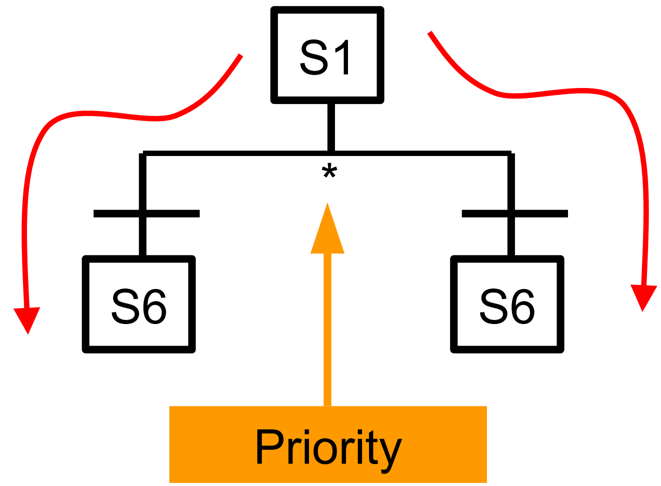


IEC 61131-3 Programming Languages
SFC – Sequential Function Chart

- Divergent Paths
  - Conditions should be mutually exclusive
  - Priorities may be applied to each branch
  - Try to avoid ambiguous situations, as the standard itself is ambiguous in the defined semantics



Lower numbers have higher priority



Priority is lft to right

Overview of IEC 61131-3

- Introduction to IEC 61131 and Related Standards
- IEC 61131-3 Standard Architecture
- Programming Languages
- IEC 61131-3 Execution Model
- Standard Functions and Function Blocks
- Examples

Standard Functions

- Standard Type Conversion Functions

```
*_TO_**        (eg: INT_TO_REAL, TIME_TO_REAL, DATE_TO_STRING, …)

*_BCD_TO_**    (*  -> BYTE, WROD, DWORD, LWORD)
  **_TO_BCD_*  (** -> USINT, UINT, UDINT, ULINT)

TRUNC          (INPUT -> REAL, LREAL;  OUTPUT -> ANY_INT)
```

NOTE 1
Unlike user defined functions,
Standard functions may be overloaded!

The function will determine the data type of the INPUT and/or OUTPUT, and produce the correct conversion

NOTE 2
Conversion from REAL or LREAL to SINT, INT, DINT or LINT, follows IEC 60559 (i.e. round to nearest integer, or, if equally near, to nearest even integer)

TRUNC always truncates towards 0.

Standard Functions

- Standard Single Parameter Numerical Functions

Name	I/O TYPE	Description
	General Functions	
ABS	ANY_NUM	Absolute Value
SQRT	ANY_REAL	Square Root
	Logarithmic Functions	
LN	ANY_REAL	Natural Logarithm
LOG	ANY_REAL	Logarithm base 10
EXP	ANY_REAL	Natural exponential
	Trigonometric Functions	
SIN	ANY_REAL	Sine in radians
COS	ANY_REAL	Cosine in radians
TAN	ANY_REAL	Tangent in radians
ASIN	ANY_REAL	Principal arc sine
ACOS	ANY_REAL	Principal arc cosine
ATAN	ANY_REAL	Principal arc tangent

Standard Functions

- Standard Arithmetic Functions

Name	Symbol	I/O TYPE	Description
	Extensible Arithmetic Functions		
ADD	+	ANY_NUM	OUT := IN1 + IN2 + ...
MUL	*	ANY_REAL	OUT := IN1 * IN2 * ...
	Non-extensible Arithmetic Functions		
SUB	-	ANY_REAL	OUT := IN1 - IN2
DIV	/	ANY_REAL	OUT := IN1 / IN2
MOD		ANY_REAL	OUT := IN1 modulo IN2
EXPT	**		Exponent.OUT:= $IN1^{IN2}$
MOVE	:=	ANY_REAL	OUT := IN

Extensible => may have many inputs

e.g.:
average:=ADD(a1, a2, a3, a4, a5) /5

Standard Functions

- Standard Arithmetic Functions

Name	Symbol	I/O TYPE	Description
	Extensible Arithmetic Functions		
ADD	+	ANY_NUM	OUT := IN1 + IN2 + ...
MUL	*	ANY_REAL	OUT := IN1 * IN2 * ...
	Non-extensible Arithmetic Functions		
SUB	-	ANY_REAL	OUT := IN1 - IN2
DIV	/	ANY_REAL	OUT := IN1 / IN2
MOD		ANY_REAL	OUT := IN1 modulo IN2
EXPT	**		Exponent.OUT:=IN1 ^{IN2}
MOVE	:=	ANY_REAL	OUT := IN

These functions are all overloaded!

All input parameters and output parameter must be of the same data type

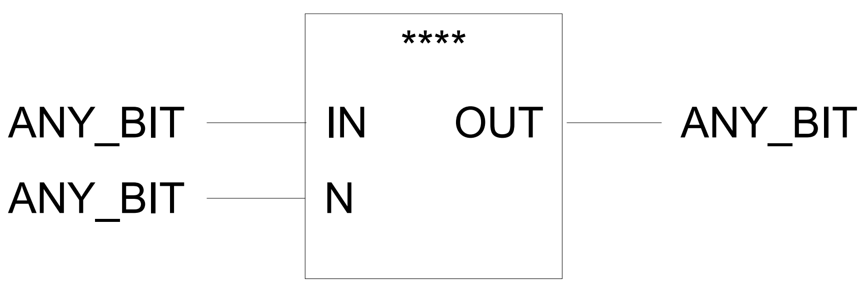
They also have non-overloaded versions

e.g.: ADD_INT, MUL_REAL, ...

Standard Functions

- Bit-String Functions

Name	I/O TYPE	Description
SHL	ANY_BIT	Left shift N bits, zero-filled
SHR	ANY_BIT	Right shift N bits, zero-filled
ROR	ANY_BIT	Right rotate N bits, circular
ROL	ANY_BIT	Natural Logarithm



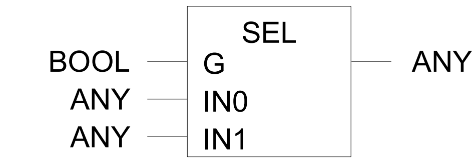
Standard Functions

- Bitwise Boolean Functions

Name	Symbol	I/O TYPE	Description
AND	&	ANY_BIT	OUT:=IN1 AND IN2 AND....
OR	>=1	ANY_BIT	OUT:=IN1 OR IN2 OR....
XOR	=2k+1	ANY_BIT	OUT:=IN1 XOR IN2 XOR....
NOT		ANY_BIT	OUT:= NOT IN1

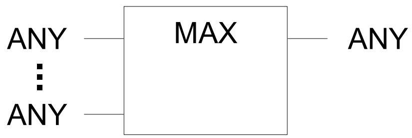
Standard Functions

- Selection Functions



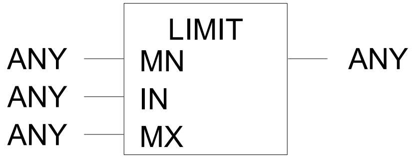
Binary selection.

```
IF     G THEN returns IN1
IF NOT G THEN returns IN0
```



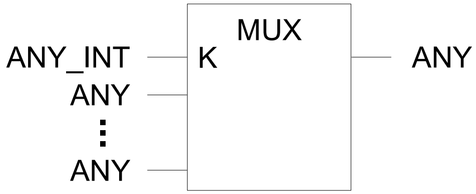
Returns highest value. Extensible.

Returns lowest value. Extensible.



Limiter.

```
returns MIN(MAX(IN, MN), MX)
```

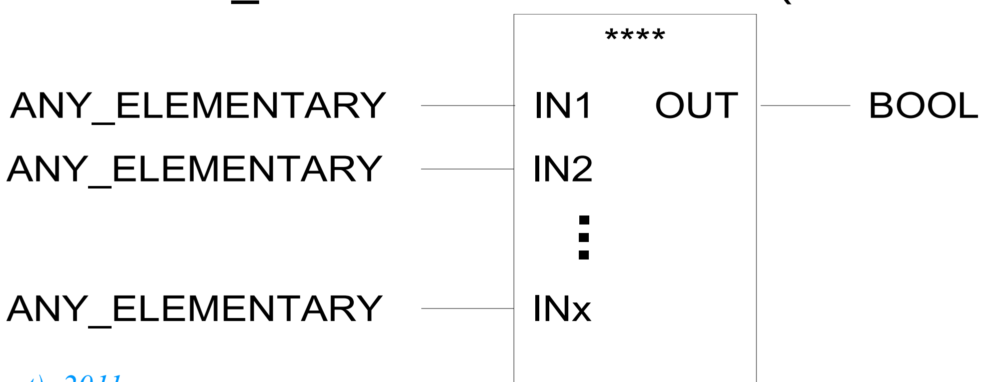


Many to 1 multiplexer.
Select input indexed by K (starts at 0).

Standard Functions

- Comparison Functions

Name	Symbol	IN TYPE	Description
	Extensible Comparison Functions		
GT	>	ANY_ELEMENTARY	OUT:= (IN1>IN2) AND(IN2>IN3) AND ....
GE	>=	ANY_ELEMENTARY	OUT:= (IN1>=IN2) AND(IN2>=IN3) ....
EQ	=	ANY_ELEMENTARY	OUT:= (IN1=IN2) AND(IN2=IN3) AND ....
LE	<=	ANY_ELEMENTARY	OUT:= (IN1<=IN2) AND(IN2<=IN3) ....
LT	<	ANY_ELEMENTARY	OUT:= (IN1<IN2) AND(IN2<IN3) AND ....
	Non-extensible Comparison Functions		
NE	<>	ANY_ELEMENTARY	OUT:= (IN1<>IN2)



Standard Functions

- Character String Functions

Inputs	Function	Output	Description
STRING	LEN	INT	Length of a string IN `LEN('IEC61131-3')` returns 10
STRING → IN ANY_INT → L	LEFT	STRING	L leftmost characters of string IN `LEFT('IEC61131-3', 3)` returns 'IEC'
STRING → IN ANY_INT → L	RIGHT	STRING	R rightmost characters of string IN `RIGHT('IEC61131-3', 3)` returns '1-3'
STRING → IN ANY_INT → L ANY_INT → P	MID	STRING	L characters, starting at P, of string IN `MID('IEC61131-3', 3, 5)` returns '131'
STRING ⋮ STRING	CONCAT	STRING	Joining of strings. Extensible. `CONCAT('IEC', '61131', '-', '3')` returns 'IEC61131-3'

Standard Functions

- ## Character String Functions (continued)

Inputs	INSERT	Output
STRING	IN1	STRING
STRING	IN2	
ANY_INT	P	

Insert IN2 into IN1, after position P

`INSERT('IEC-3', '61131', 3)` returns 'IEC61131-3'

Inputs	DELETE	Output
STRING	IN1	STRING
ANY_INT	L	
ANY_INT	P	

Delete L characters, starting at P, from string IN1

`MID('IEC61131-3', 3, 5)` returns '131'

Inputs	REPLACE	Output
STRING	IN1	STRING
STRING	IN2	
ANY_INT	L	
ANY_INT	P	

Replace L characters of IN1, starting at P, by IN2.

`REPLACE('IEC61131-3', '60848', 7, 4)` returns 'IEC60848'

Inputs	FIND	Output
STRING	IN1	ANY_INT
STRING	IN2	

Find the position of the first occurrence on IN2 in IN1

`FIND('ABCABC', 'BC')` returns 2

Standard Functions

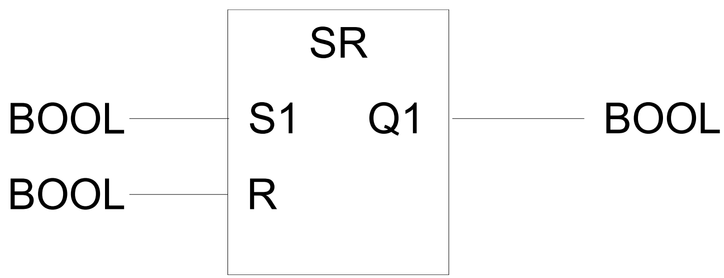
- Date and Time Manipulation Functions

Name	Symbol	IN1 TYPE	IN2 TYPE	OUT Type
ADD	+	TIME	TIME	TIME
ADD	+	TIME_OF_DAY	TIME	TIME_OF_DAY
ADD	+	DATE_AND_TIME	TIME	DATE_AND_TIME
SUB	-	TIME	TIME	TIME
SUB	-	DATE	DATE	TIME
SUB	-	TIME_OF_DAY	TIME	TIME_OF_DAY
SUB	-	TIME_OF_DAY	TIME_OF_DAY	TIME
SUB	-	DATE_AND_TIME	TIME	DATE_AND_TIME
SUB	-	DATE_AND_TIME	DATE_AND_TIME	TIME
MUL	*	TIME	ANY_NUM	TIME
DIV	/	TIME	ANY_NUM	TIME
CONCAT		DATE	TIME_OF_DAY	DATE_AND_TIME

Don't forget: DATE_AND_TIME_TO_TIME_OF_DAY and DATE_AND_TIME_TO_DATE

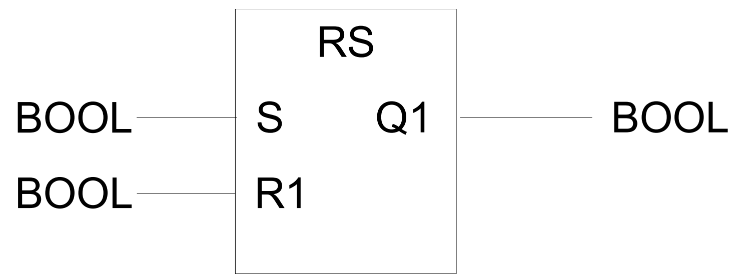
Standard Function Blocks

- RS Flip-Flops



Set dominant RS Flip-Flop

Q1 := S1 OR (R AND NOT Q1)

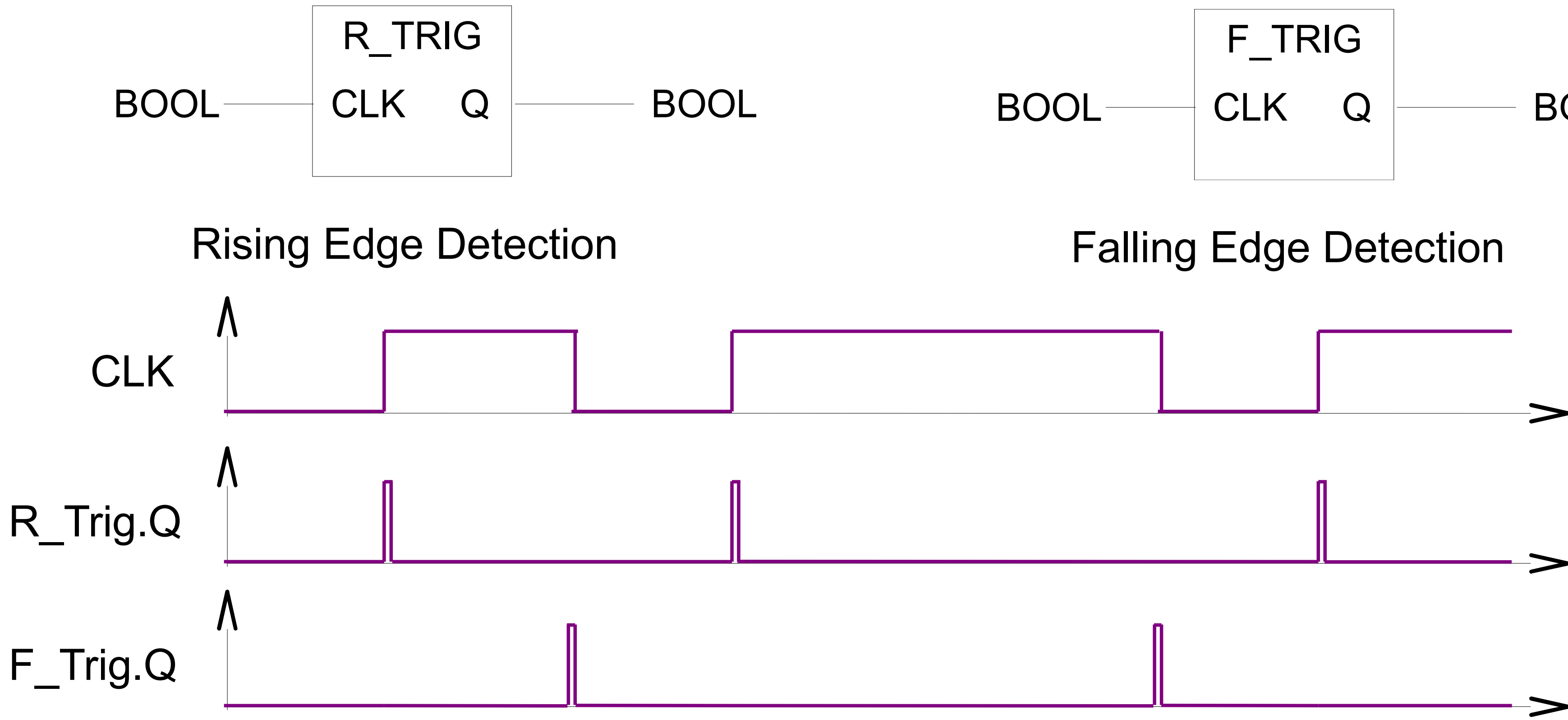


Reset dominant RS Flip-Flop

Q1 := NOT R1 AND (S OR Q1)

Standard Function Blocks

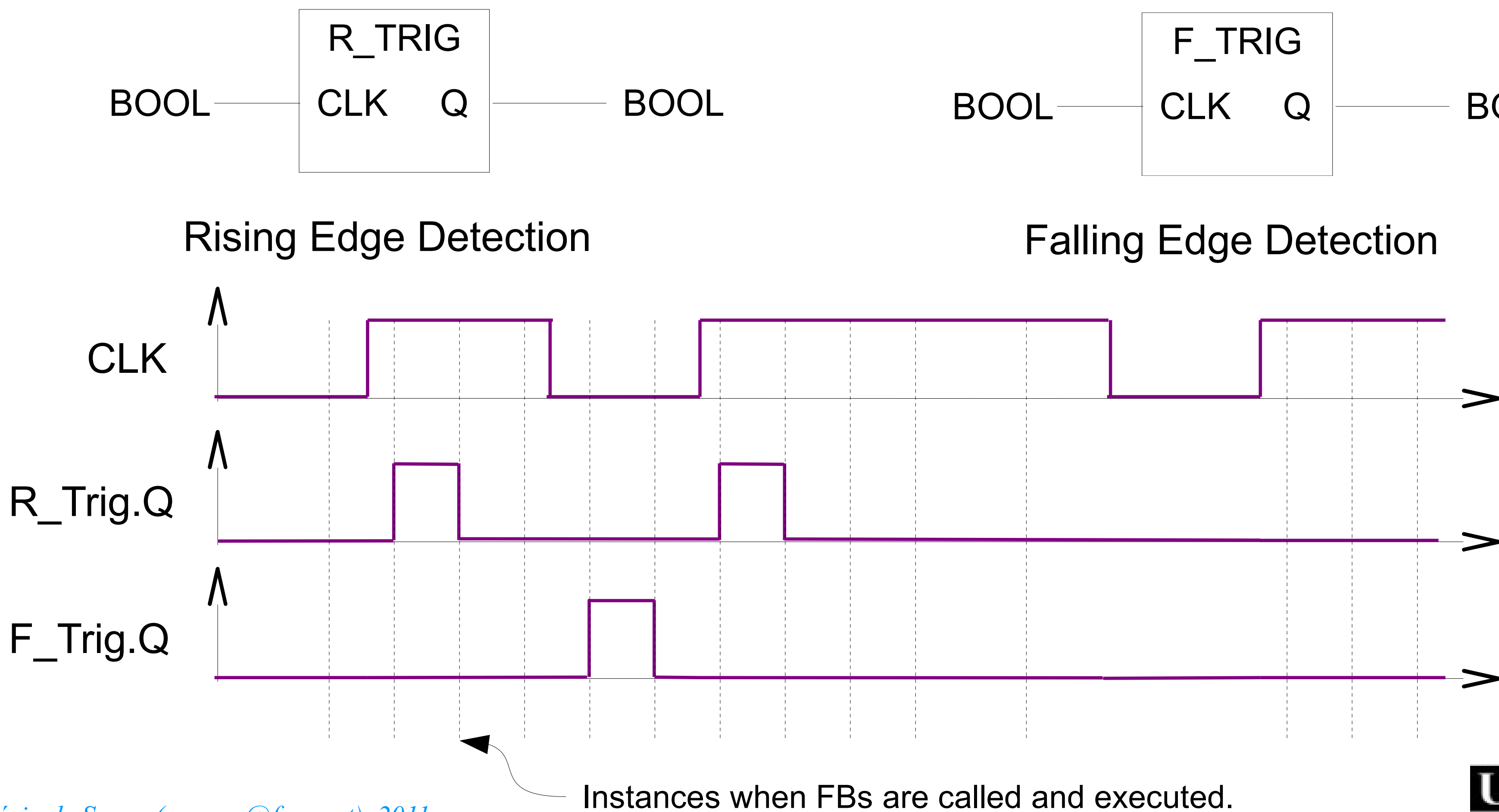
- Edge Detection



Trigger outputs Q will remain at TRUE between two consecutive invocations.
This is not necessarily a short time (as displayed in above chart).

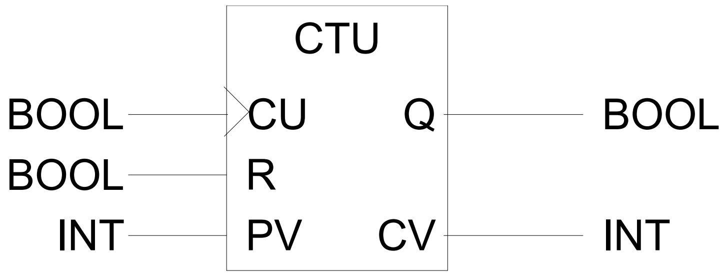
Standard Function Blocks

- Edge Detection



Standard Function Blocks

- Counters



Up Counter

CU – Count Up on Rising Edges
R – Reset counter (CV:=0)
PV – Preset Value
Q – Output := (CV >= PV)
CV – current value

```
IF R THEN CV := 0 ;
ELSIF Re(CU) AND (CV < PVmax)
    THEN CV := CV+1;
END_IF ;
Q := (CV >= PV) ;
```

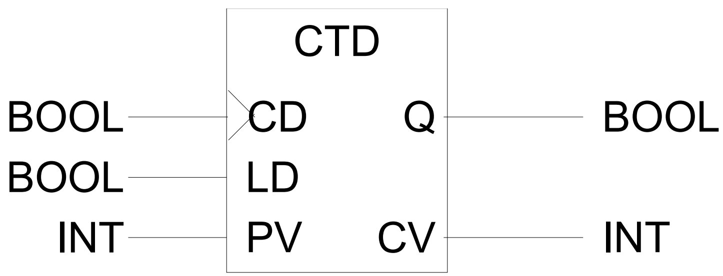
Pvmin, Pvmax:
implementation dependent parameters

Also available in:

CTU_DINT, CTU_LINT, CTU_UDINT, CTU_ULINT

Standard Function Blocks

- Counters




```
IF LD THEN CV := PV ;
ELSIF Re(CD) AND (CV > PVmin)
      THEN CV := CV-1;
END_IF ;
Q := (CV <= 0) ;
```

Pvmin, Pvmax:
implementation dependent parameters

Down Counter

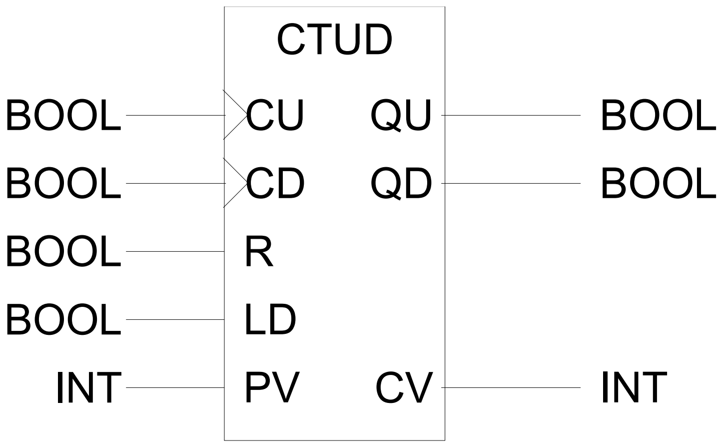
CD – Count Down on Rising Edges
LD – Load Preset value (CV:=PV)
PV – Preset Value
Q – Output := (CV <= 0)
CV – current value

Also available in:

CTD_DINT, CTD_LINT, CTD_UDINT, CTD_ULINT

Standard Function Blocks

- Counters



Up/Down Counter

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

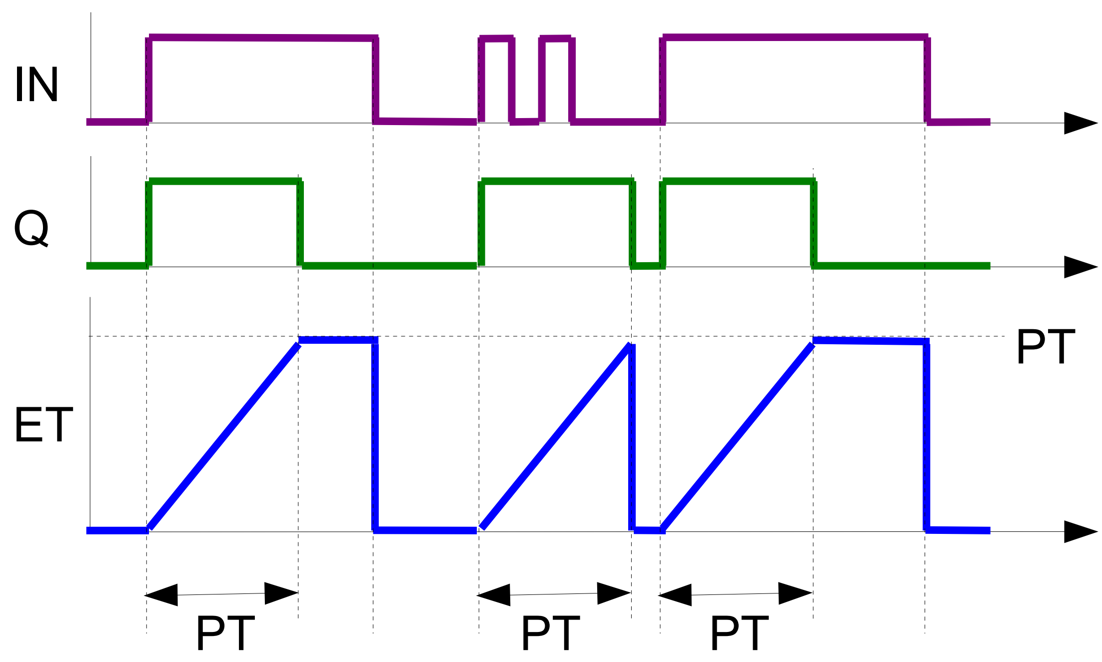
Also available in:

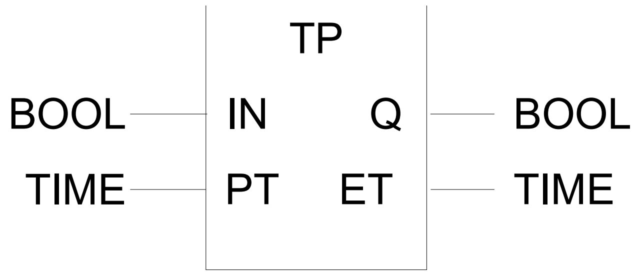
CTUD_DINT, CTUD_LINT, CTUD_ULINT

Pvmin, Pvmax:
implementation dependent parameters

Standard Function Blocks

- Timers: TP – Pulse Timer



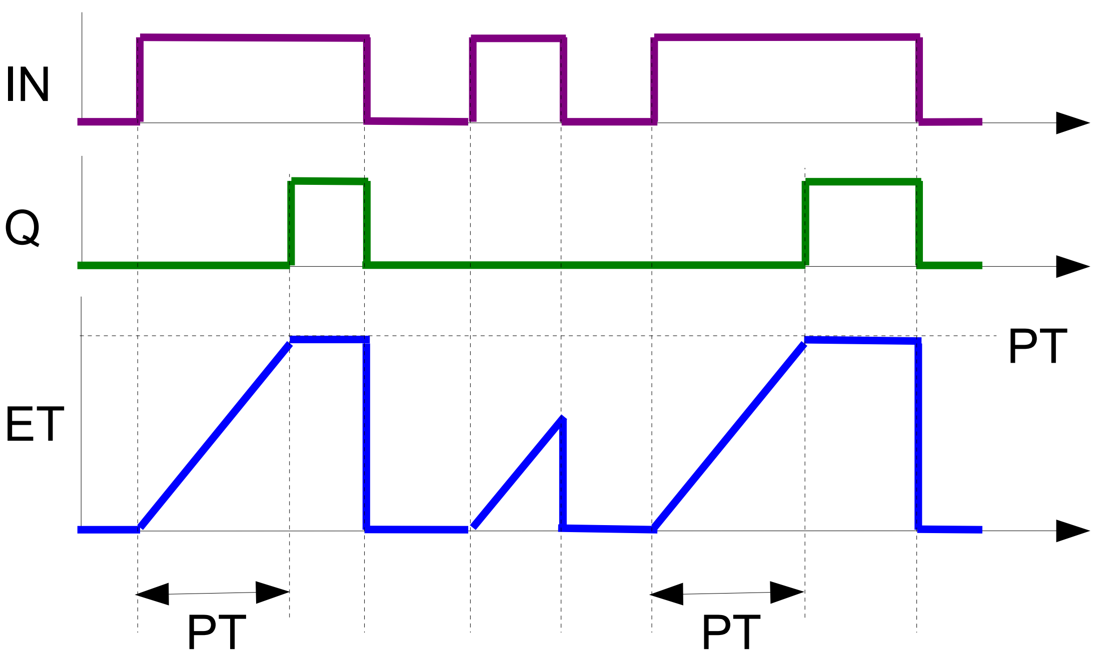


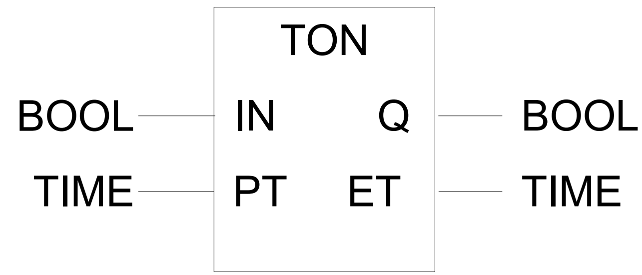
PulseTimer

PT – Pulse Time
ET – Elapsed Time

Standard Function Blocks

- Timers: TON – On Delay Timer



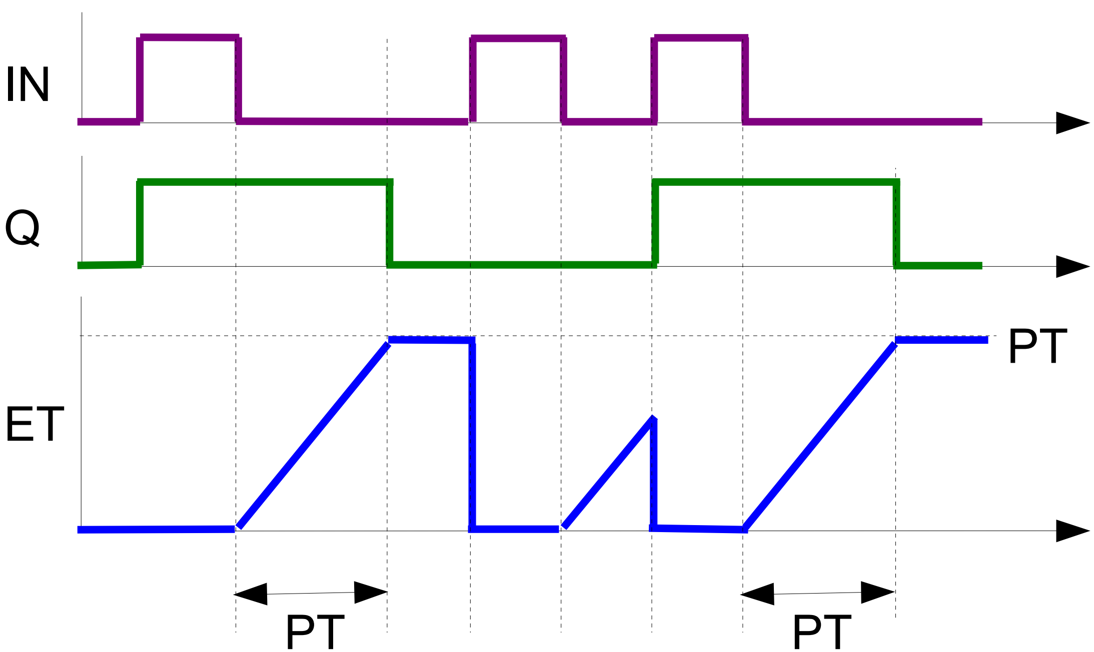


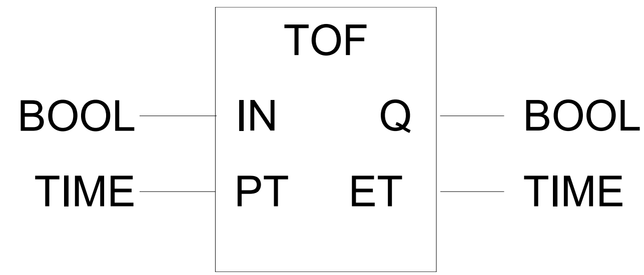
On Delay Timer

PT – Pulse Time
ET – Elapsed Time

Standard Function Blocks

- Timers: TOF – Off Delay Timer





Off Delay Timer

PT – Pulse Time
ET – Elapsed Time

Standard Function / Function Blocks

- EN / ENO
  - Standard Function and Standard FBs may have the parameters

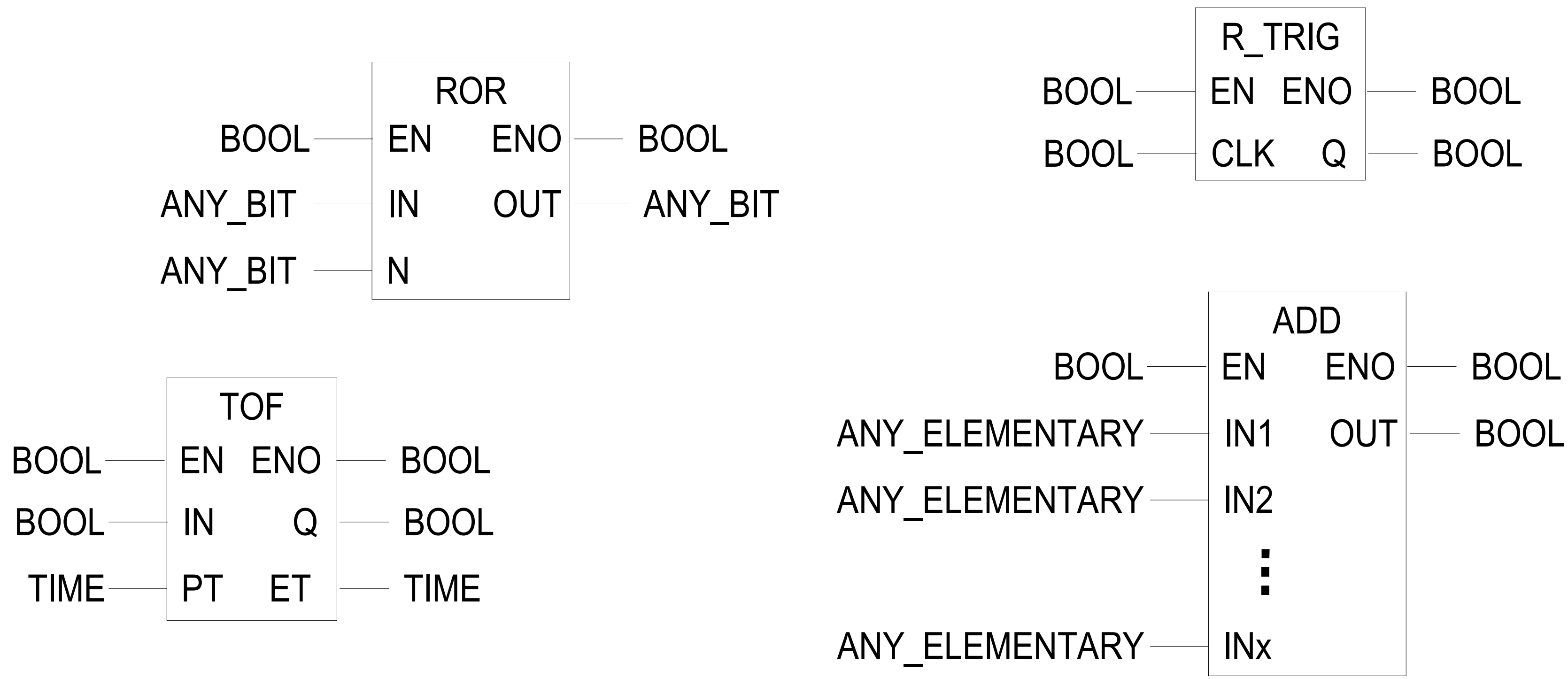
```
VAR_INPUT  EN : BOOL:=1; END_VAR
VAR_INPUT  ENO: BOOL;    END_VAR
```

  - User may also decide to declare them in his Functions / Fbs
  - These Parameters are Optional!
    - May not be present in every IEC61131-3 implementationENO
  - EN
    - Disables the execution of the Function/FB when set to FALSE
  - ENO
    - If EN is FALSE, or on error in the Function / FB execution, ENO is set to FALSE
    - If EN is TRUE, and Function / RB executes correctly, ENO is set to TRUE

Note default value of TRUE!

Standard Function / Function Blocks

- EN / ENO (examples)



**When used, these
are always the first two parameters.**

They show up at the top of the block!

Overview of IEC 61131-3

Questions?

(preferably in English)

Kysymyksiä?

Questions?

Otázky?

Questions?

질문?

Въпроси?

質問ですか？

Spørgsmål?

Domande?

Vragen?

الأسئلة؟

Spørsmål?

Perguntas?

Pitanja?

¿Preguntas?

Fragen?

问题？

Ερωτήσεις;

Frågor?

Pytania?

Întrebări?

Вопросы?