MEIM | MASTER IN ENTREPRENEURSHIP INNOVATION MANAGEMENT IN COLLABORATION WITH MIT SLOAN

IN COLLABORATION WITH MIT MANAGEMENT SLOAN SCHOOL

UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

MASTER MEIM 2021-2022

Industrial Automation: The IEC 611131-3 standard SCADA and Human Machine Interface

Sara Dubbioso

A cura del prof. Lorem Ipsum

Prof. Di Economia e Management all'Università degli Studi di Napoli Parthenope

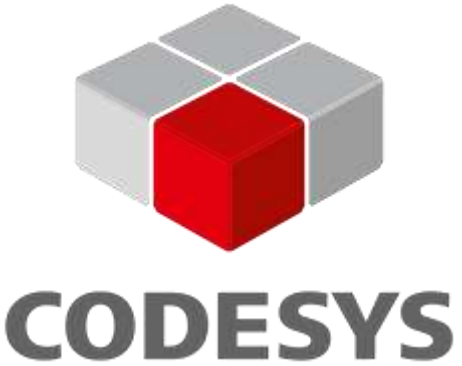
Lecture overview

- Introduction to Programmable Logic Controllers (PLCs)
  - Brief history of PLCs
  - PLC components and operation mode
- The IEC-61131-3 standard
  - Constant, variables and types
  - Main components
  - Programming Languages
- SCADA
  - Human Machine-Interface (HMI)

Software overview

CODESYS

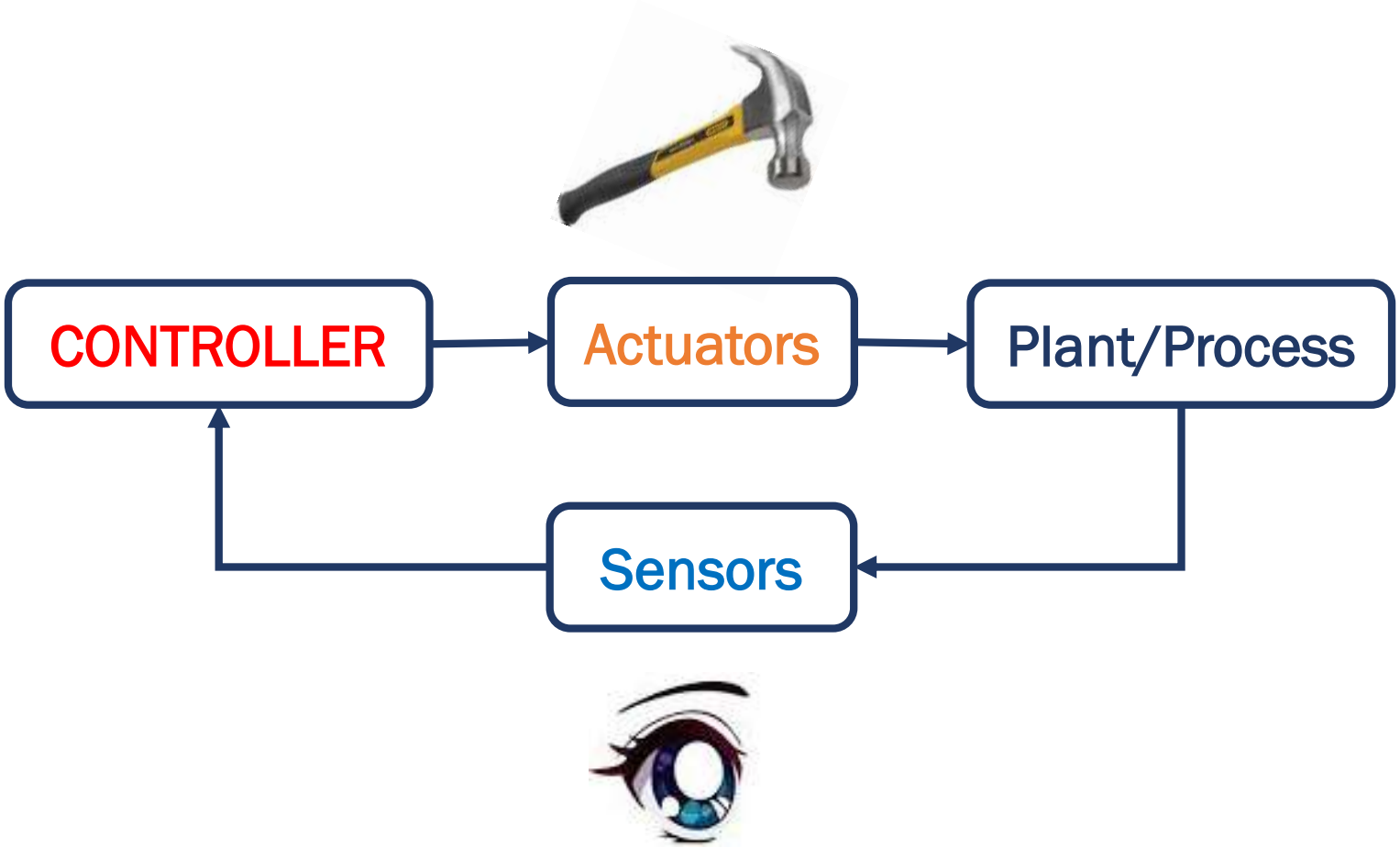
- Integrated **development environment** for programming controller applications according to the standard IEC 61131-3
- Download: https://store.codesys.com/en/



FACTORY IO

- 3D **factory simulation** for learning automation technologies
- offers scenes inspired by typical industrial applications
- Download (30 days trial): https://factoryio.com/start-trial

Control System



- The input devices (**SENSORS**) sense what is happening in the process
- The controller decides what to do about it
- The output devices (**ACTUATORS**) manipulate the process to achieve the desired result

Programmable Logic Controller (PLC)

A **general-purpose controller**, applicable to many different types of process control applications

- the end-user can program, or instruct, the PLC to do virtually any control function imaginable
- specialized for **logic and sequential control**

PLCs are often **used in factories and industrial plants** to control motors, pumps, lights, fans, circuit breakers and other machinery.

- They can connect to IT network and implement complex functionalities
- Can be seen as PC manufactured to work in industrial environments

PLC brief history

Before PLCs (the early to mid 1900s) automation was usually done using complicated **electromechanical relay circuits**

- Slow signal processing
- Bulky wired switchboard
- No flexibility
- High portability cost
- Not easy re-programmability
- Hard trouble shooting

On a basic level, **electromechanical relays** function by magnetically opening or closing their electrical contacts when the coil of the relay is energized. Can be used to represent logic variables

PLC brief history

The raising of digital electronics comes with **new requirements** for controllers

- Easley re-programmable
- Scalability and easy maintainability (modular architecture)
- Robust design
- Low dimension, consumption and cost

In1968 the first programmable logic controller came along **to replace** complicated relay circuitry in industrial plants

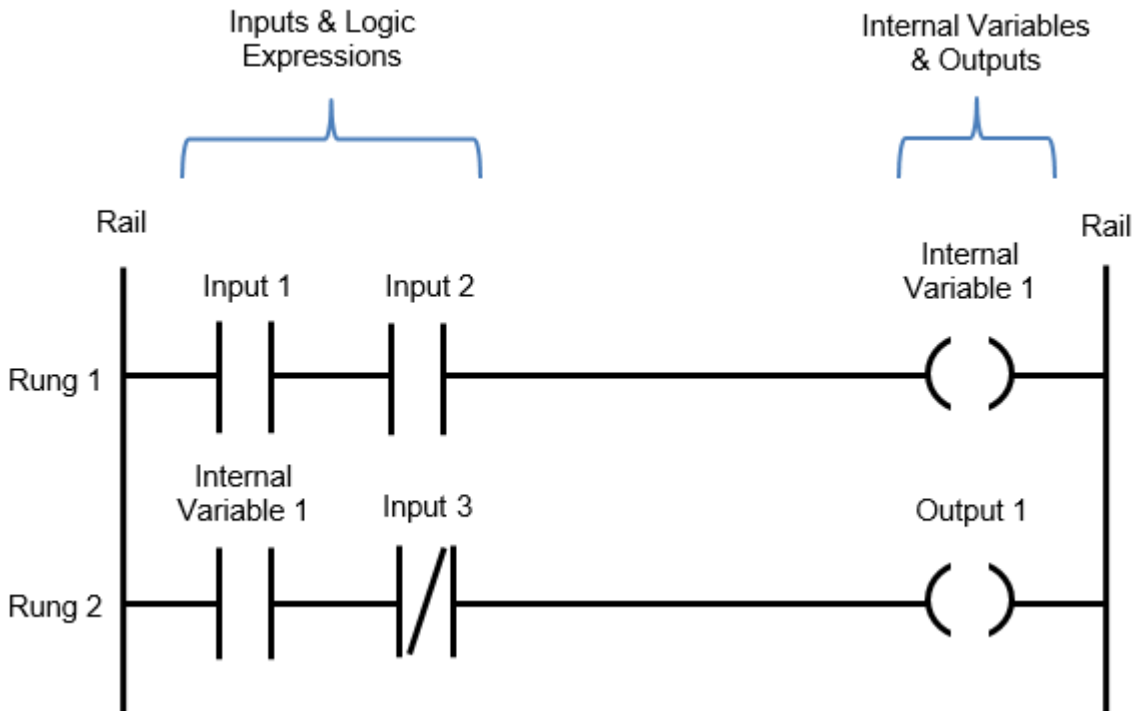
- **Modicon 084** from *Bedford Associates* (now *Schneider Electric*)

PLC brief history

In the mid‘70 came out the first **PLC based on a microprocessor** (the 8080 from *Allen-Bradley*)

- Dimension reduction
- Increase memory size
- Higher number of I/Os
- Introduction of higher-level programming languages
- Comunnication network with peripheral devices and computers
- I/O modules for analog variables

PLC brief history



The legacy of PLCs as relay-replacements is probably most evident in their traditional programming language: **Ladder Diagram**.

- Mimics the control circuit schematics
- Easily programmable by plant engineers and technicians that were already familiar with relay logic and control schematics

The IEC 61131 standard

No-standard for PLC manifacture and programming

- Programming terminals were single purpose
- Programming software were developed by third party companies (not from PLC manufacturer themselves)
- Custom and hardware-based programming languages were used by developers

1993: International Electrotechnical Commission (IEC) published the **IEC 61131 standard** for PLC hardware and software requirements

- It is composed by several parts

The IEC 61131 standard

- 1st part: General information [IEC 61131-1]
- 2nd part: Equipment requirements and tests [IEC 61131-2]
- 3rd part: Programming languages [IEC 61131-3]
- 4th part: User guidelines [IEC 61131-4]
- 5th part: Communications [IEC 61131-5]
- 6th part: Fuzzy control programming [IEC 61131-7]
- 7th part: Guidelines for the application and implementation of programming languages for programmable controllers [IEC 61131-8]

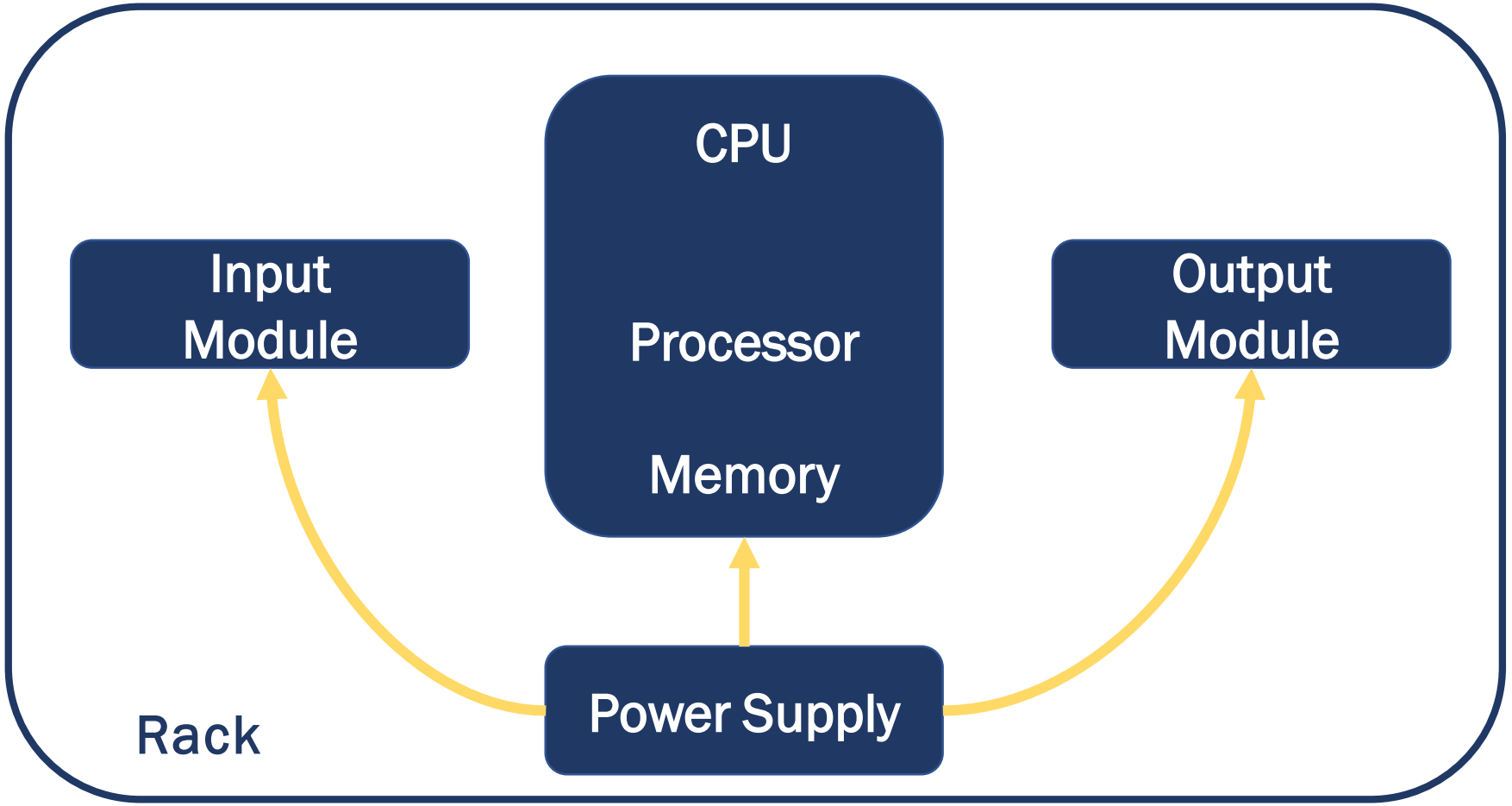
The IEC 61131-3 standard

- 1st part: General information [IEC 61131-1]
- 2nd part: Equipment requirements and tests [IEC 6113
- 3rd part: Programming languages [IEC 61131-3]
- 4th part: User guidelines [IEC 61131-4]
- 5th part: Communications [IEC 61131-5]
- 6th part: Fuzzy control programming [IEC 61131-7]
- 7th part: Guidelines for the application and implementa languages for programmable controllers [IEC 61131-8]

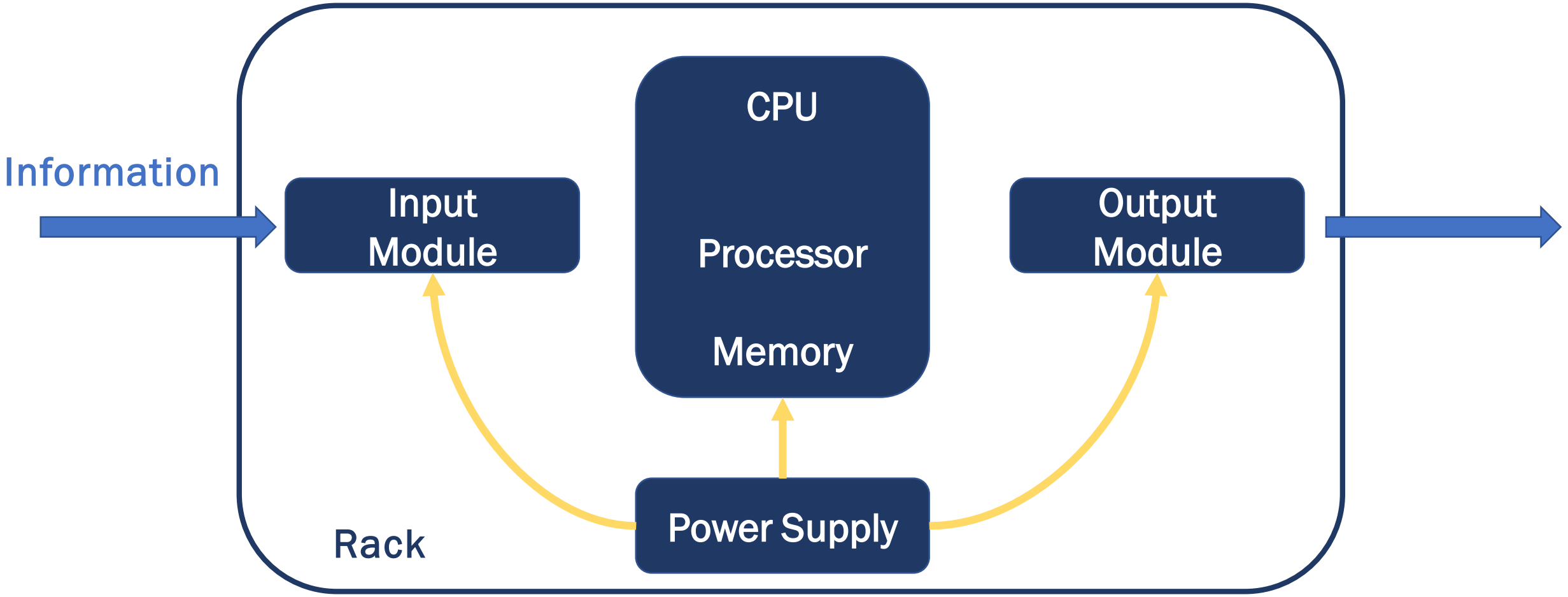
It specifies the **syntax, semantics and display** for a suite of five PLC programming languages:

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

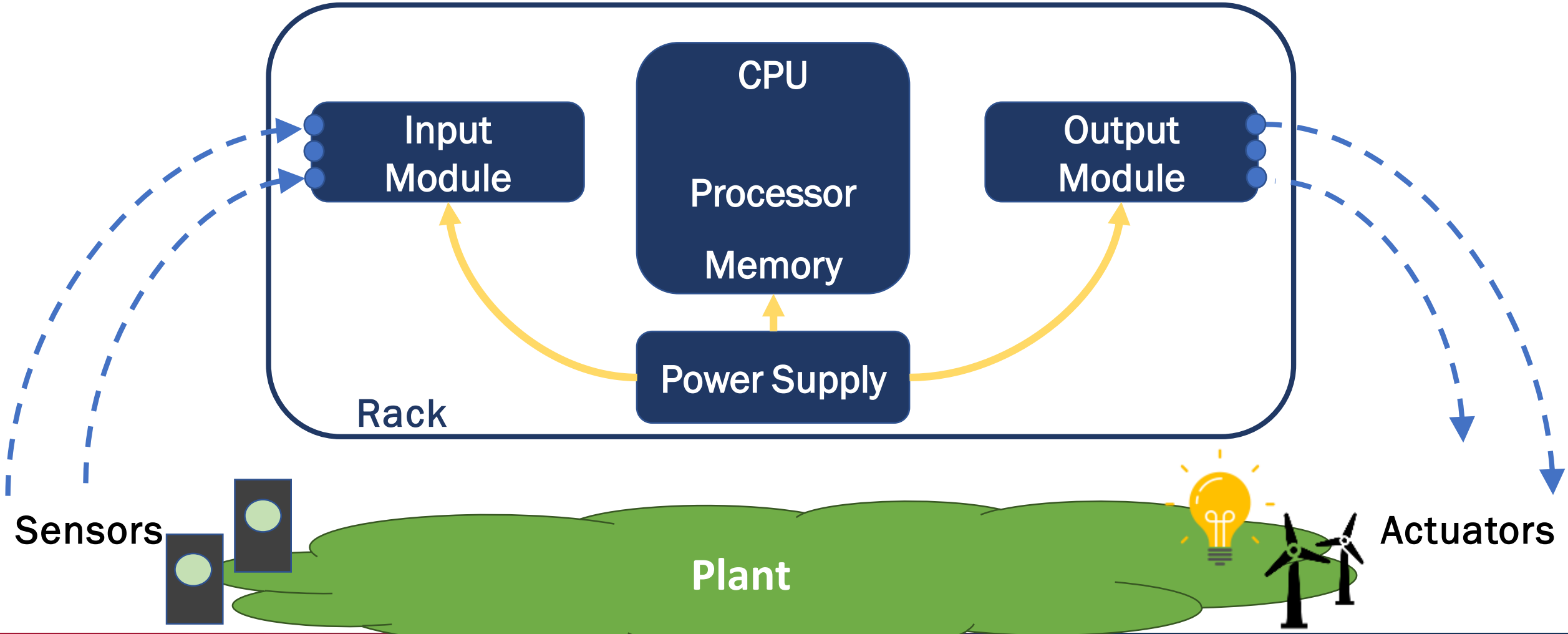
Components of a PLC



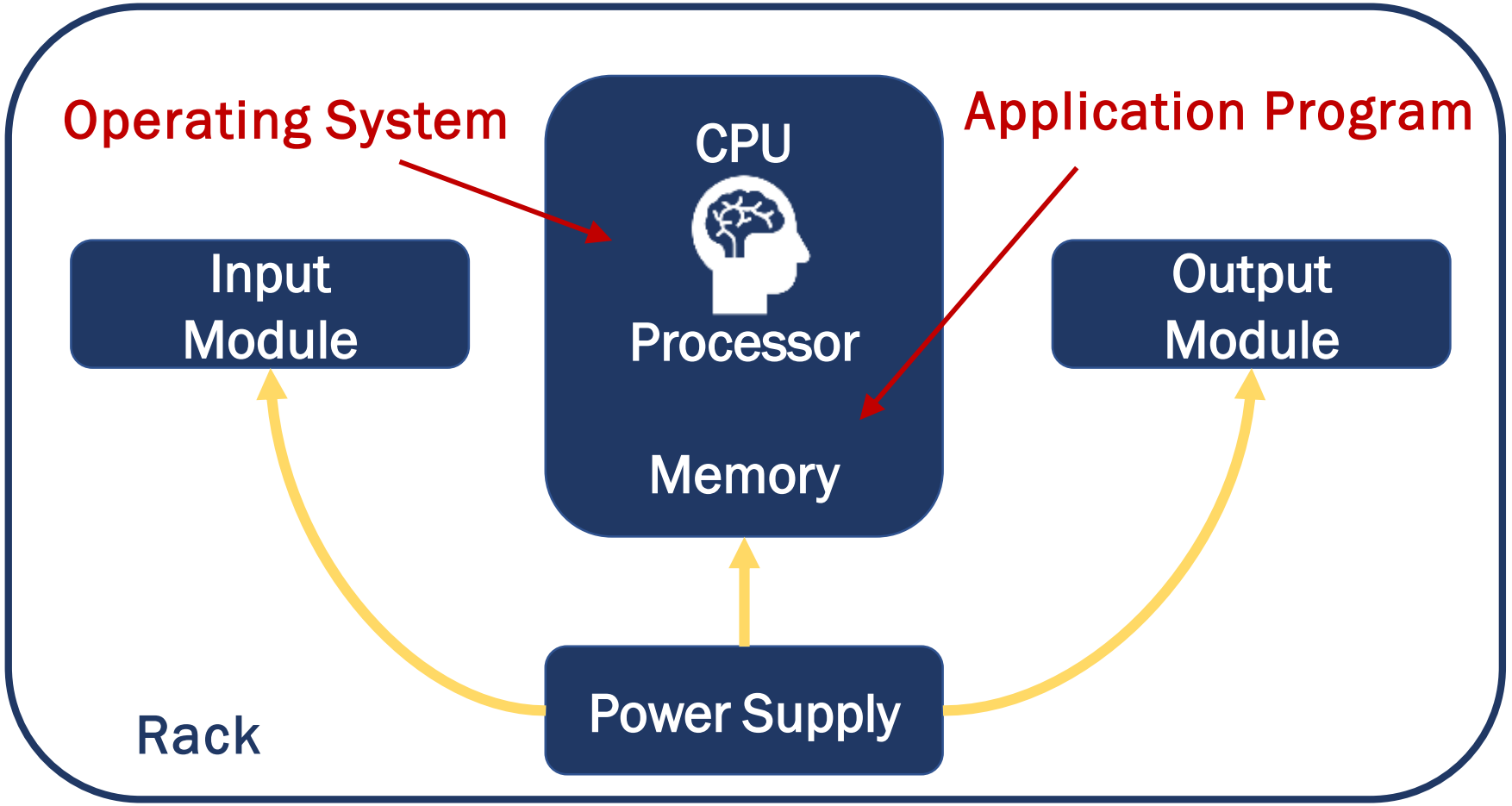
The IO modules



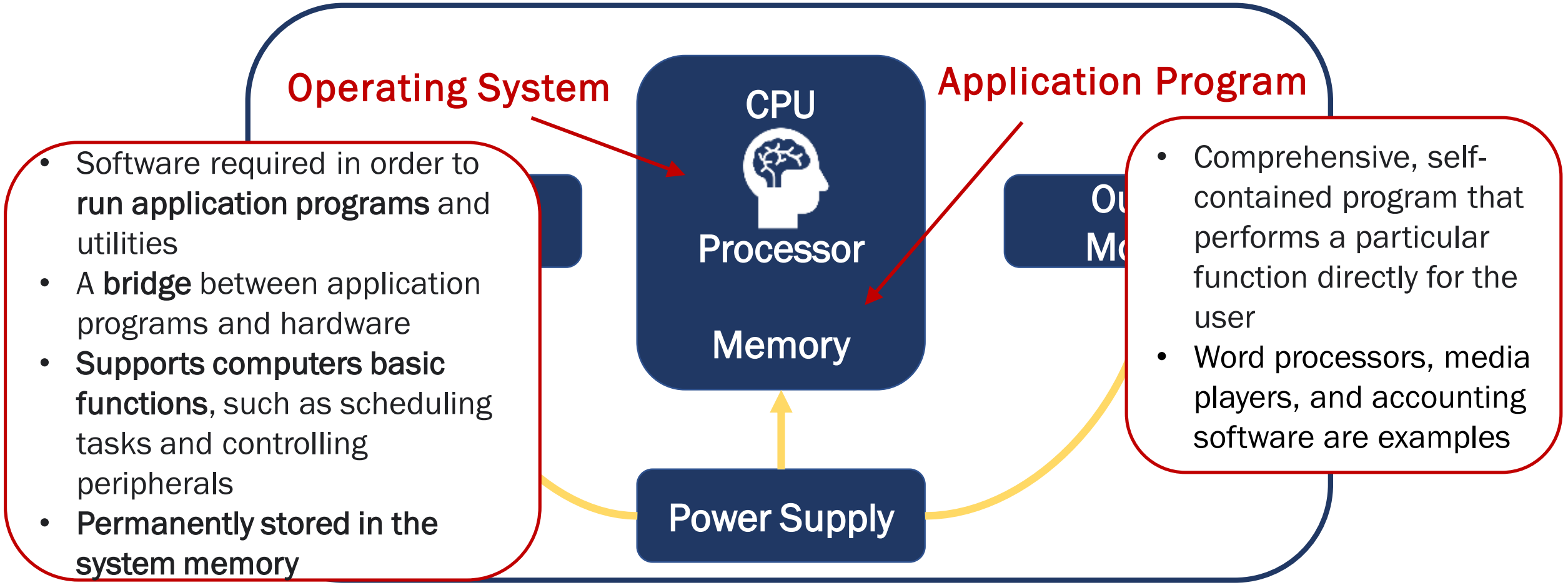
The IO modules



Central Processing Unit (CPU)



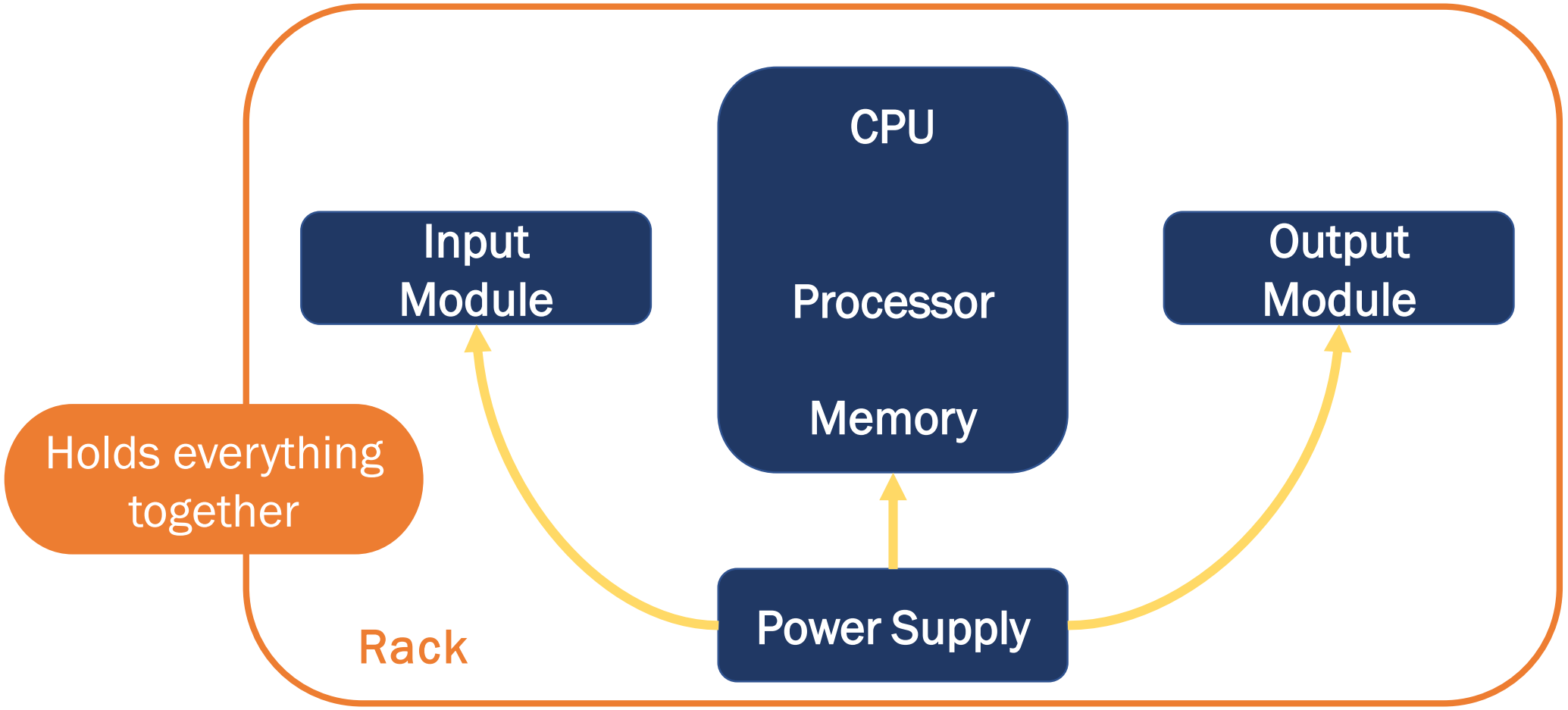
Central Processing Unit (CPU)



- Software required in order to **run application programs** and utilities
- A **bridge** between application programs and hardware
- **Supports computers basic functions**, such as scheduling tasks and controlling peripherals
- **Permanently stored in the system memory**

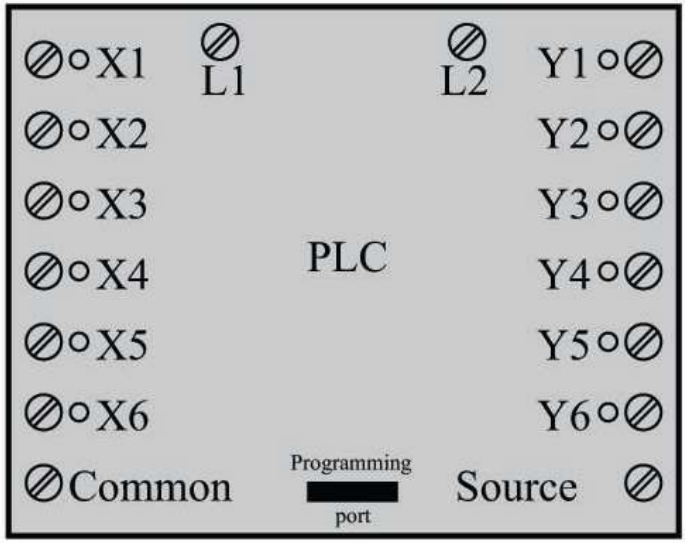
- Comprehensive, self-contained program that performs a particular function directly for the user
- Word processors, media players, and accounting software are examples

Rack and power supply



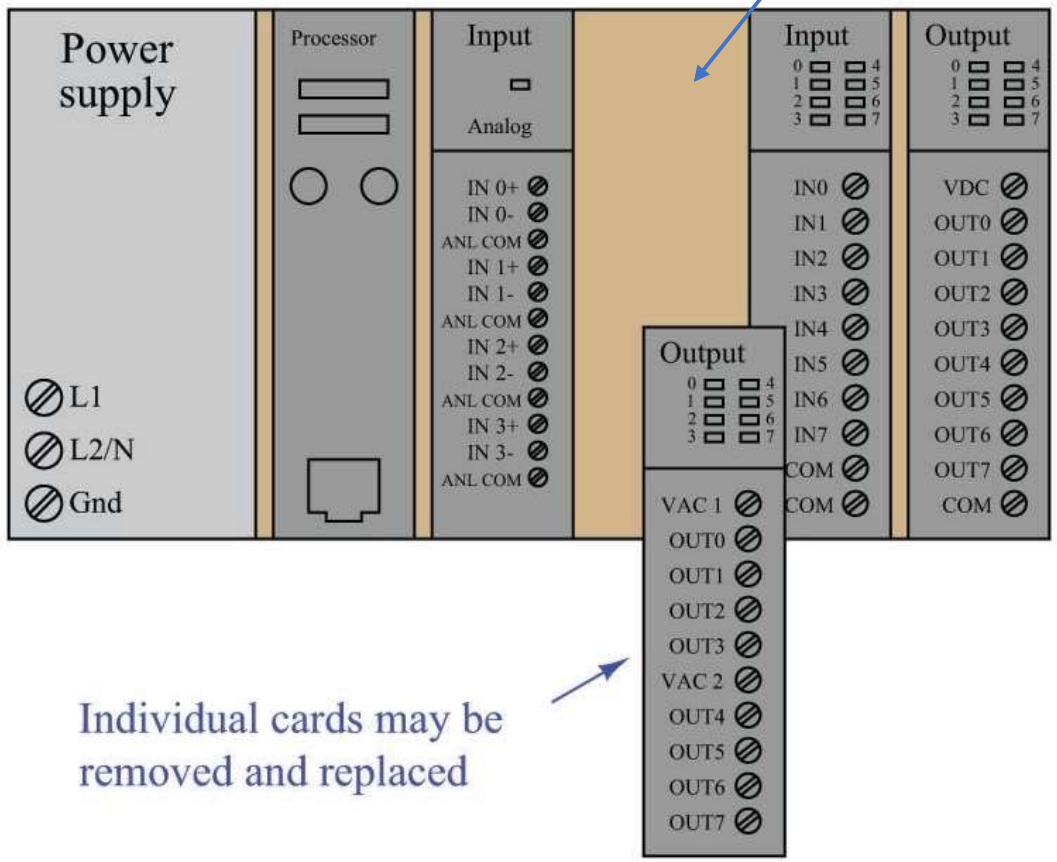
Components of a PLC

Monolithic PLC



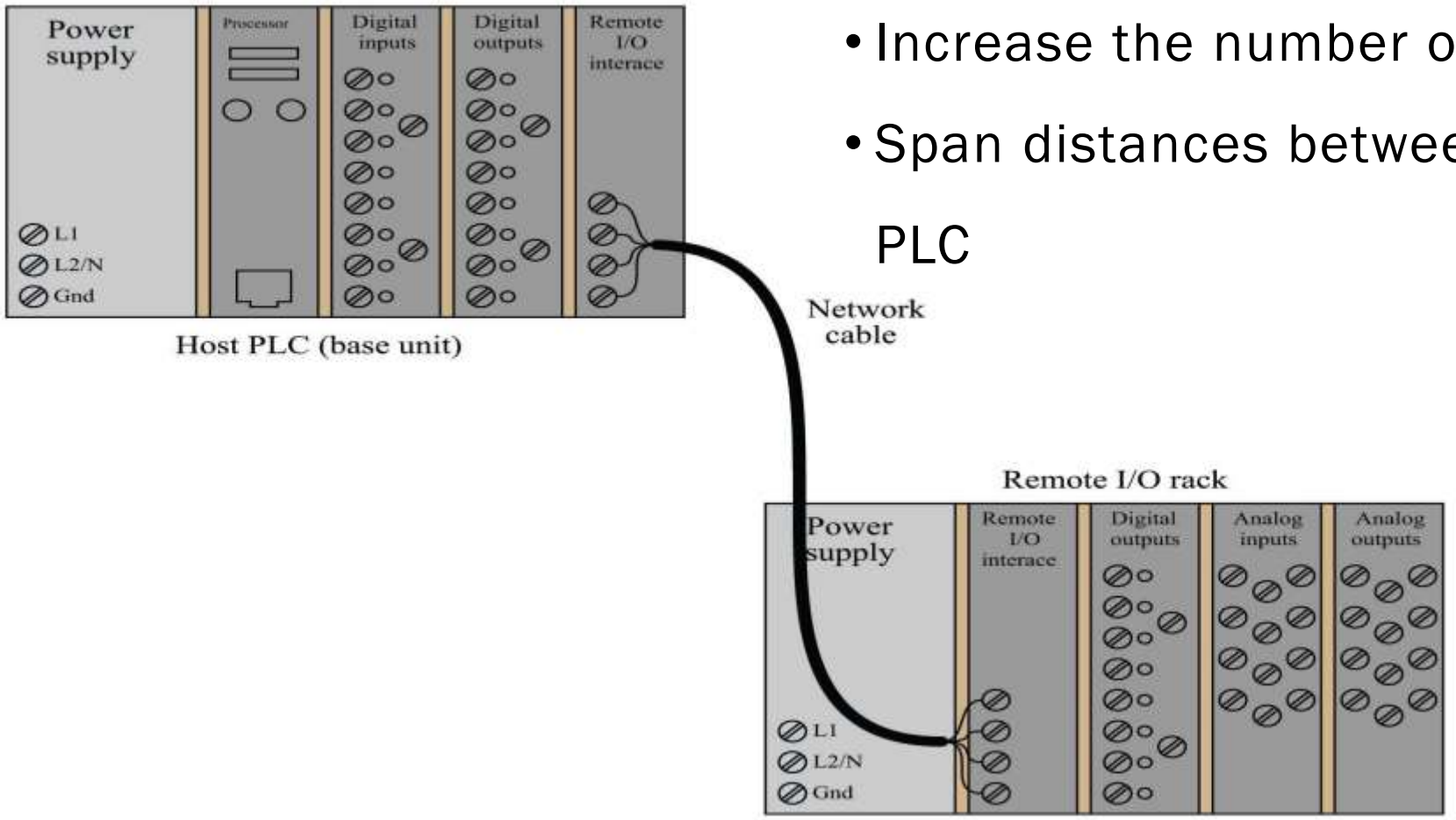
All I/O is fixed in one PLC unit

Modular ("rack-based") PLC



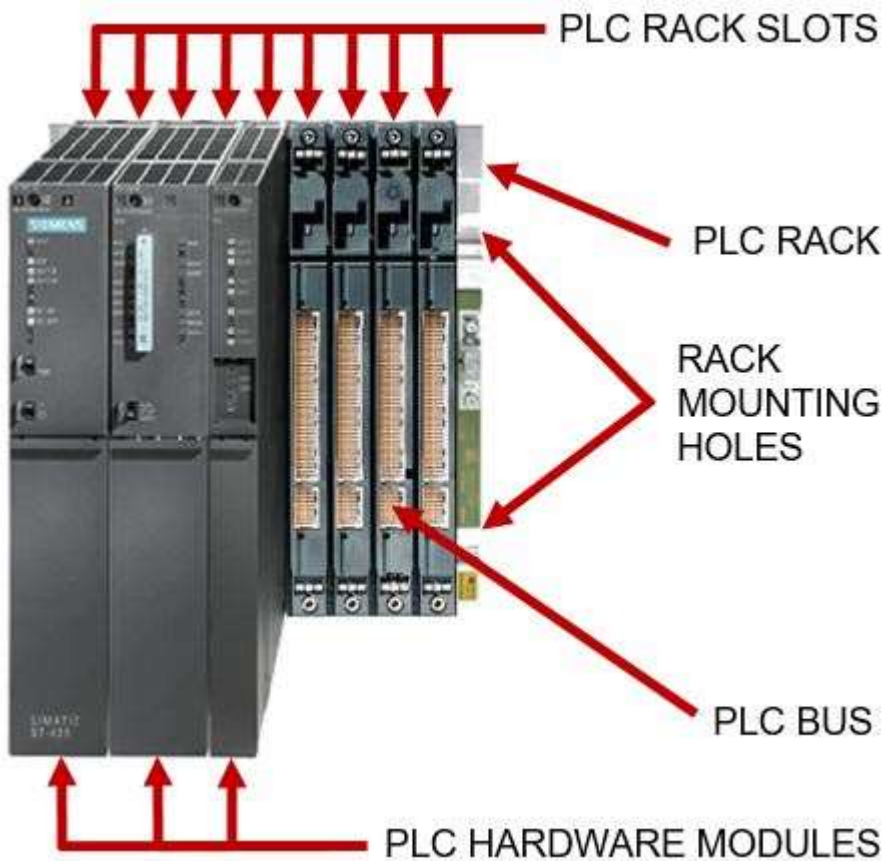
Individual cards may be removed and replaced

Components of a PLC



- Increase the number of I/O channels
- Span distances between the plant and the PLC

Components of a PLC



MEIM | MASTER IN ENTREPRENEURSHIP INNOVATION MANAGEMENT IN COLLABORATION WITH MIT SLOAN

Components of a PLC

PLC operation

Execution mode

- **Periodic**
  (e.g., every 20 ms)
- **Cyclic**
  (the program restart when finished)
- **Event driven**
  (instructions executed when a specific even occurs)

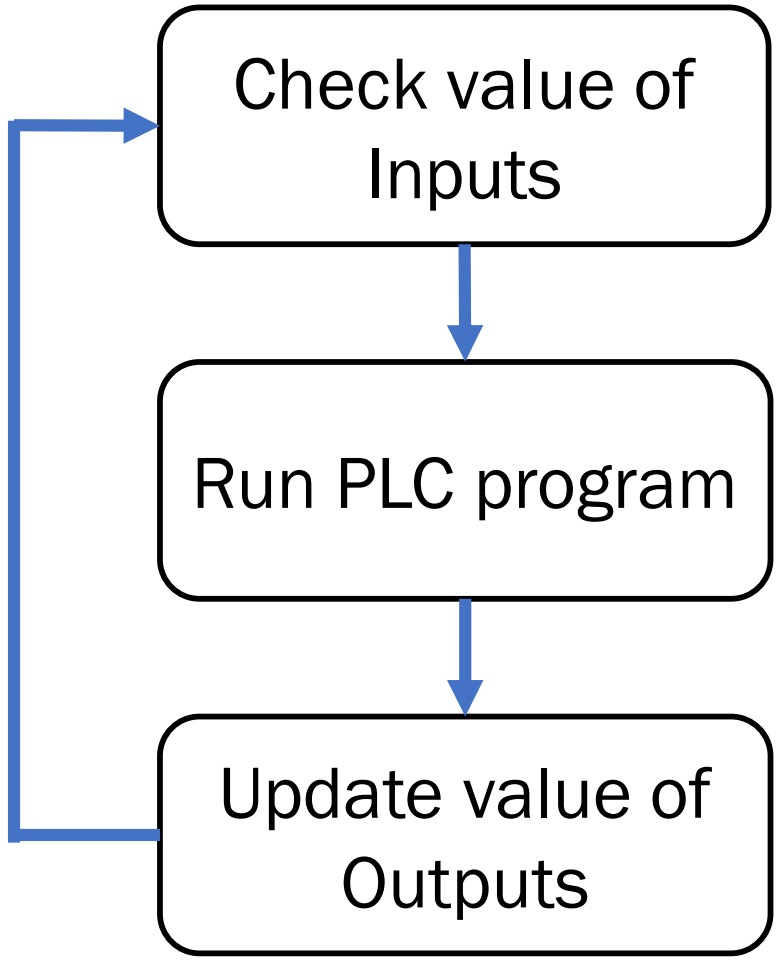
PLC operation

The scan cycle is the cycle in which the PLC

- gathers the inputs
- runs your PLC program
- and then updates the outputs.

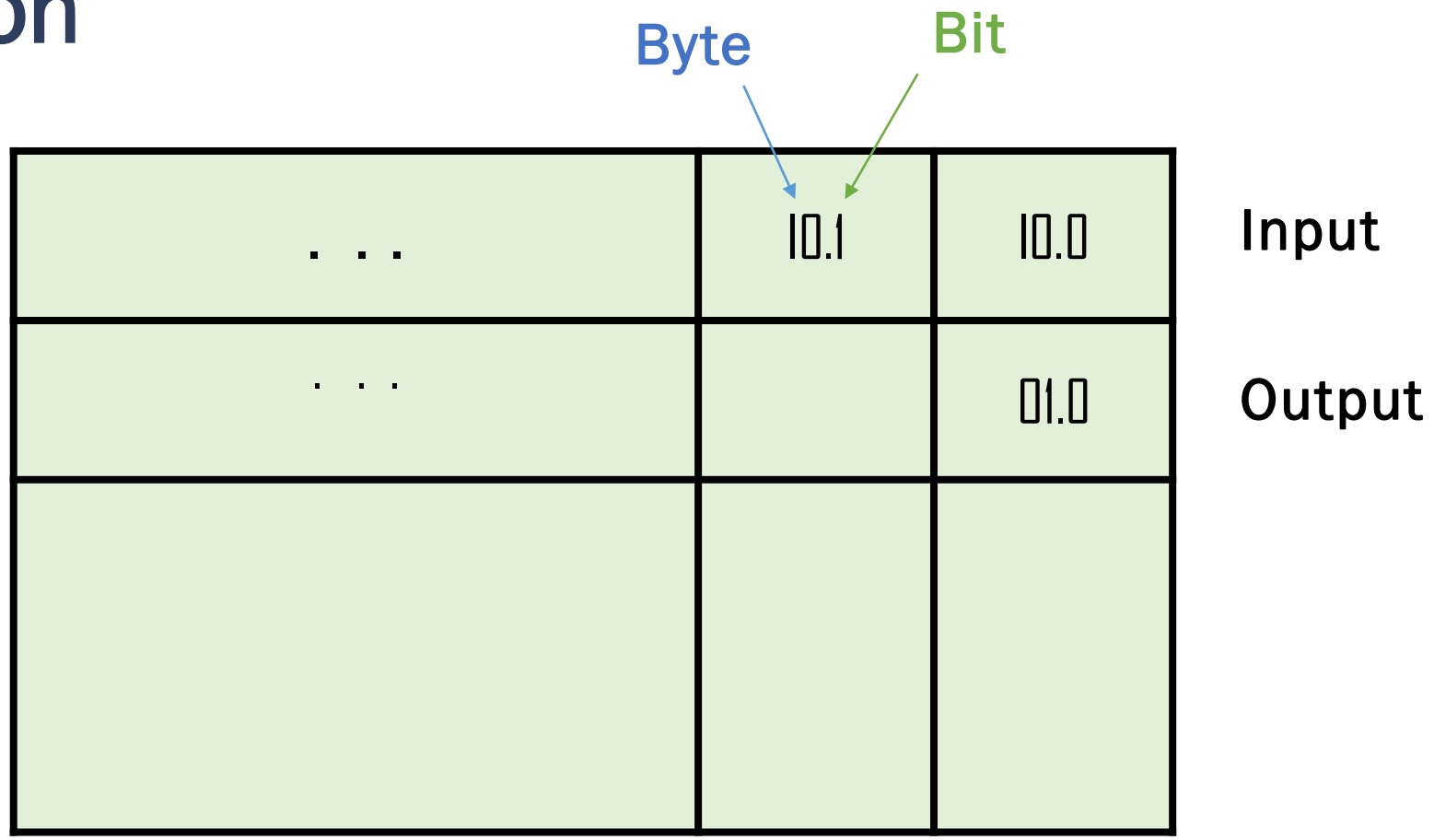
The scan time is amount of time it takes for the PLC to make one scan cycle

- often measured in milliseconds (ms)



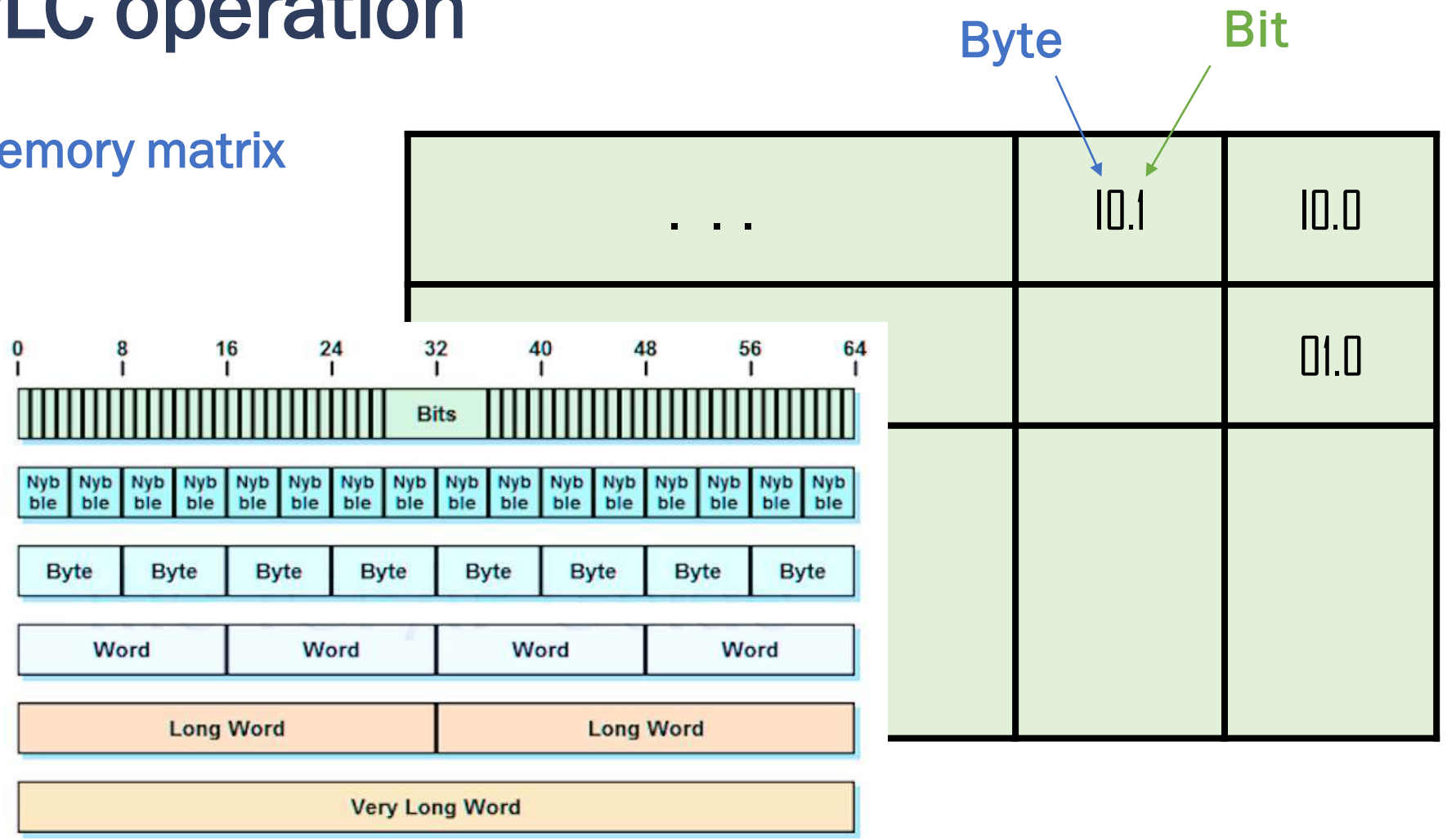
PLC operation

Memory matrix



PLC operation

Memory matrix



The IEC 61131-3 standard

PLC programming is regulated by the IEC 61131 standard, 3rd part. It aimed at a software design:

- Comprehensible
- Modular
- Well structured
- Portable

The standard provides **a benchmark** for both manufacturers and user

- **describes the PLC programming languages**
- comprehensive concepts and **guidelines for creating PLC projects**.

The IEC 61131-3 standard

Texual languages:

- Instruction List (IL)
- Structured Text (ST)

Graphic laguages:

- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Sequential Functional Chart (SFC)

Variables, Data Types and Common Elements

Data inside a PLC program can be represented by means of **variables**

Variables are **declared in textual form**

- Declarations essentially consist of **an identifier** as well as information about **the data type** used
- The standard defines some **predefined types**

What is a variable? A variable is any characteristic, number, or quantity that can be measured or counted.

```
VAR
   Start : BYTE;      (*declaration of variable "Start" with data type BYTE *)
END_VAR
```

Predefined data type

Integers:

- INT integers (16 bits, from -2^{15} to 2^{15})
- UINT unsigned integers (16 bits, from 0 to 2^{16} -1)

Reals

- REAL reals (32 bit, from - 10^{38} to 10^{38})
- LREAL long real (64 bits)

Time variables

- TIME duration (e.g., T#1d3h5m12s50ms)
- DATE, TIME_OF_DAY date and time of the day
- DATE_AND_TIME date and time together

Predefined data type

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	1	1	0	0	1	0

$(1\ x\ 32) + (1\ x\ 16) + (1\ x\ 2) = 50$

Integers:

- INT integers (16 bits, from -2^{15} to 2^{15})
- UINT unsigned integers (16 bits, from 0 to 2^{16} -1)

Reals

- REAL reals (32 bit, from - 10^{38} to 10^{38})
- LREAL long real (64 bits)

Time variables

- TIME duration (e.g., T#1d3h5m12s50ms)
- DATE, TIME_OF_DAY date and time of the day
- DATE_AND_TIME date and time together

Predefined data type

String:

- STRING string of characters

String of bits

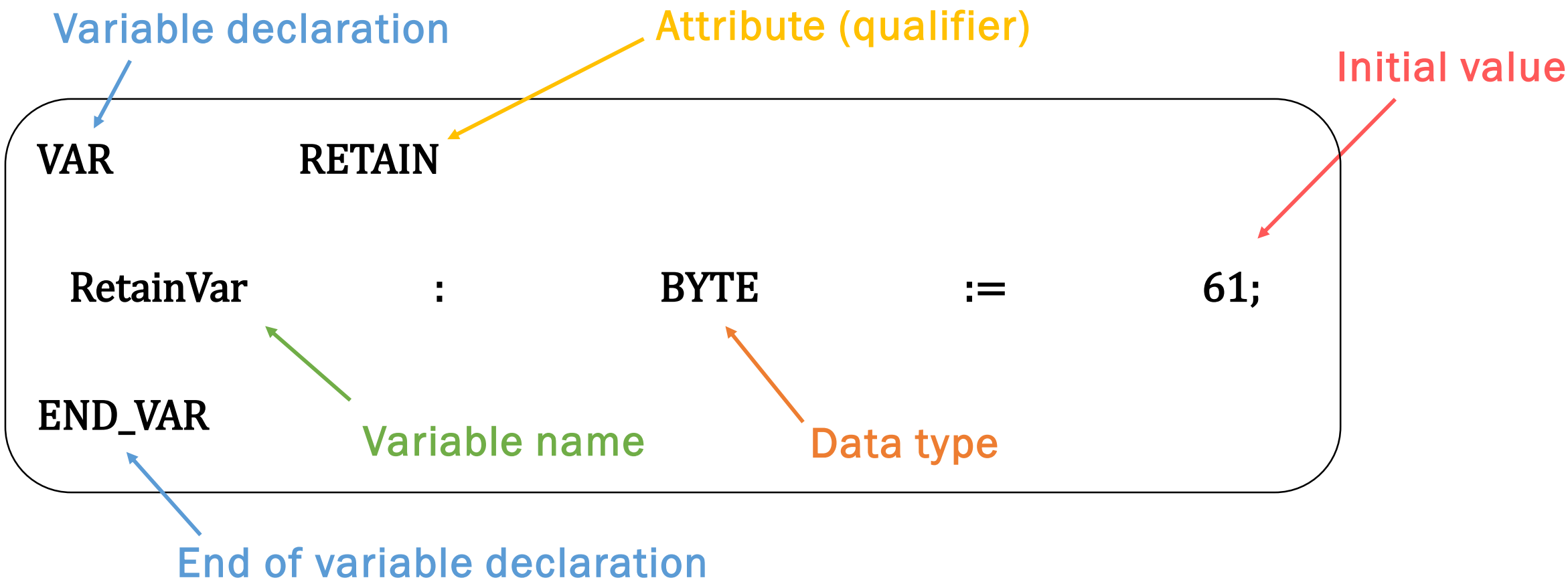
- BOOL single bit or logic variable
- BYTE 8 bits
- WORD 16 bits
- DWORD 32 bits
- LWORD 64 bits

Predefined data type

Generic data type:

- ANY any type
- ANY_NUM any numeric type
- ANY_INT any types of integer (signed and unsigned)
- ANY_REAL any types of real
- ANY_DATE any time variables
- ANY_BIT any string of bits

Attributes to variables



Declaration and initialization

Declarations

Variables are declared at the beginning of programs, functions and function blocks

- Key words: VAR . . . END_VAR

Initialization

Non ambiguity principle is fundamental: a variable must always be declared with an initial value. Can be a default one:

- 0 for numeric variables
- Null string
- 01/01/0001 for date variables

Attributes to variables

Attributes, or qualifiers, are additional properties can be assigned to variables

- RETAIN the values are to be retained during a loss of power
- CONSTANT the values are not allowed to be changed during program execution
- AT specifies the memory location

Comments (* this is a comment *)

Function declaration is ST

```
FUNCTION RealAdd: REAL                          (* function heading *)
VAR_INPUT                                       (* variable type "input" *)
Inp1, Inp2: REAL;                               (* variable declaration *)
END_VAR                                         (* end of variable type *)
RealAdd := Inp1 + Inp2 + 7.456E-3;              (* ST statement *)
END_FUNCTION                                    (* end of the function *)
```

Types of variables

Inputs

Typically associated with the state of a sensor (read-only) or inputs to a POU

- Key words: VAR_INPUT. . . END_VAR

Output

Typically associated with actuators or return variables from POU

- Key words: VAR_OUTPUT. . . END_VAR

Input/Output

Refers to POU external variable but that is still editable

- Key words: VAR_IN_OUT. . . END_VAR

Internals

Refers to temporary data or support variable

Global variables

A variable is accessible only internally to the POU in which it was declared (apart from I/O ones)

Global variables

- Key words: VAR_GLOBAL. . . END_VAR

It is accessible by the POUs internal to the one where they were declared. To be used it must be declared:

- Key words: VAR_EXTERNAL. . . END_VAR

Directly represented variables

Is possible to specify the memory areas. They come with a specific notation

- %ABxxx
  - A is the location prefix
    - **I** for input
    - **Q** for output
    - **M** for generic memory area
  - B is the data size
    - **X** (or no specified) for one bit
    - **B** for a byte (8 bit)
    - **W** for a word (16 bit)
    - **D** for a double word (32 bit)
    - **L** for a long word (64 bit)
  - **xxx** multi-digit hierarchical address (are dependent on the manufacturer)

Directly represented variables

Is possible to specify the memory areas. They come with a specific notation

- %ABxxx
  - A is the location prefix
    - **I** for input
    - **Q** for output
    - **M** for generic memory area
  - B is the data size
    - **X** (or no specified) for one bit
    - **B** for a byte (8 bit)
    - **W** for a word (16 bit)
    - **D** for a double word (32 bit)
    - **L** for a long word (64 bit)
  - **xxx** multi-digit hierarchical address (are dependent on the manufacturer)

- %IX01.1
  - Input area
  - one bit
  - 1st byte
  - 1st bit

- %MB04
  - Memory area
  - one byte
  - 4th byte

Derived data type

Is possible to create new data types derived from the predefined ones

- Key words: TYPE . . . END_TYPE

Initial value	The variable is given a particular initial value
Enumeration	The variable can assume one of a specified list of names as a value
Range	The variable can assume values within the specified range
Array	Several elements of the same data type are combined into an array
Structure	Several data types are combined to form one data type

Derived data type

TYPE

```
Temperature : REAL := 20;          (*initial value*)

Color : (red, yellow, green);      (* enumeration *)

Sensor : INT (-56..128);           (* range *)

Measure : ARRAY [1..45] OF Sensor; (* array *)

...
```

Derived data type

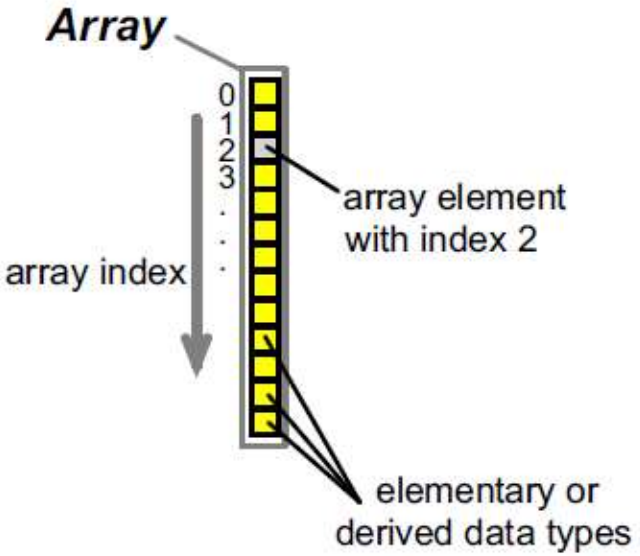
```
Testbench : STRUCT                    (* structure *)
    Place : UINT;                     (* elementary data type *)
    Light : Color:= red;              (* enumerated data type with initial value *)
    Meas1 : Measure;                  (* array type *)
    Meas2 : Measure;                  (* array type *)
END_STRUCT;
END_TYPE
```

Arrays and structures

Arrays are directly consecutive data elements of the same data type in memory

- Can be access by index → Meas1[2]

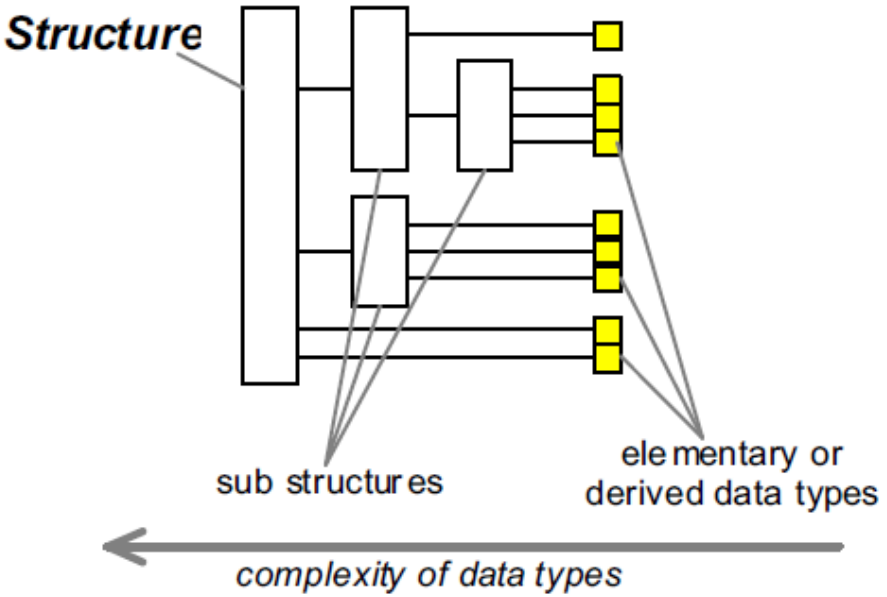
```
TYPE
    Meas_1Dim : ARRAY [1..45] OF Sensor;
    Meas_2Dim : ARRAY [1..10,1..45]OF Sensor;
END_TYPE
```



Arrays and structures

Data structures can be built up hierarchically

- Key words: STRUCT . . . END_STRUCT
- Can be access by field → Testbench.Place




```
TYPE
    temperature_sensor: STRUCT
        value : temperature;
        last_calibration: DATE;
        calibration_interval: TIME;
        max_value: REAL := 100.0;
        diagnostic : BOOL;
    END_STRUCT
END_TYPE
```

The IEC 61131-3 standard

Texual languages:

- Instruction List (IL)
- Structured Text (ST)

Graphic laguages:

- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Sequential Functional Chart (SFC)

The standard is just **a reference**. It is not mandatory. Remember that you can find PLC programmed with languages **not compliant** with the standard (especially if old PLC)

Instruction List (IL)

Instruction List IL is a convenient **assembler-like** programming language.

IL is a line-oriented language: An **instruction**, which is an executable command for the PLC, is described in exactly one line

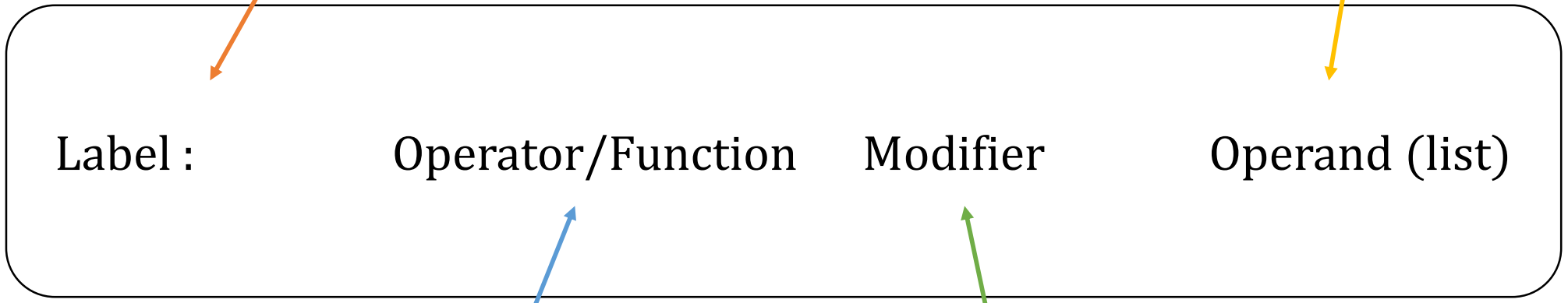
- An instruction consists of an **operator** (or a function) plus a number of **operands** (parameters)
- Can come with a **label** or a **modifier**

The operators implicitly refer to an **accumulator** called the “Current Result” (CR), in addition to the indicated operand

Instruction List (IL)

Jump label in order to reach this instruction

None, one or several constants or variables for the operator



IL operator or function name

Operator extension

IL operators

- `LD` load (operand -> accumulator)
- `ST` store (accumulator -> operand)
- `S` set (set a logic variable to 1)
- `R` reset (reset a logic variable to 0)
- `AND/&, OR, XOR` (logic operators)
- `ADD, SUB, MUL, DIV` (math operators)
- `GT, GE, EQ, NE, LE, LT` (comparison operators)
- `JMP` (jump to the label specified by the operand)
- `CAL` (function/function block call)
- `RET` (return from a function/function block)

IL example

Cyclic AND between two Boolean variables

```
MRun:   LD      %IX3.0      (* Load bit from I/O in the accumulator*)
        AND     a           (* AND of the accumulator with the variable a*)
        ST      r           (* Store value of the accumulator in the variable r*)
        JMP     MRun        (* Jump to the instruction labeled with MRun *)
```

IL modifiers

- `N` Negation of operand (the operand is negated before carrying out the instruction)
- `(` Nesting levels by parenthesis
- `C` Conditional execution of operator (if `CR = TRUE`, or `CR = FALSE` if combined with `N`). Can be used with `JMP, CAL, RET`

IL example

XOR between two Boolean variables + conditional jump

```
MRun:   LD      a       (* load the value of a in the accumulator *)
        ANDN    b       (* AND of the accumulator with the negation of b *)
        OR      ( b     (* OR of the accumulator with the result
        ANDN    a           of the expression between brackets *)
                )
        ST      e       (* store the value of the accumulator in e*)
      JMPC      MRun    (*jump if the result in the accumulator is 1*)
```

Structured Text (ST)

ST is called a High-Level language (similar to Pascal)

An **ST algorithm** is divided into several **ST statements**.

- Statements are separated by semicolons (;)
- A statement is used to compute and assign values, control the command flow, and call or leave a POU

The part of a statement that combines several variables and/or function calls to produce a value is called an **expression**

- An expression consists of **operands** and associated **ST operators**

ST operators

- Assignment `:=`
- Terminator `;`

- **Math operators**
  - `+` (Addition)
  - `-` (Subtraction)
  - `*` (Multiplication)
  - `/` (Division)
  - `MOD` (Modulo)
  - `**` (Exponentiation)

- **Logic operators**
  - `AND` or `&`
  - `OR`
  - `XOR` (exclusive OR)
  - `NOT` (complement)
- **Relational operators**
  - `<` and `<=`
  - `>` and `>=`
  - `=` (equality)
  - `<>` (inequality)

ST statements

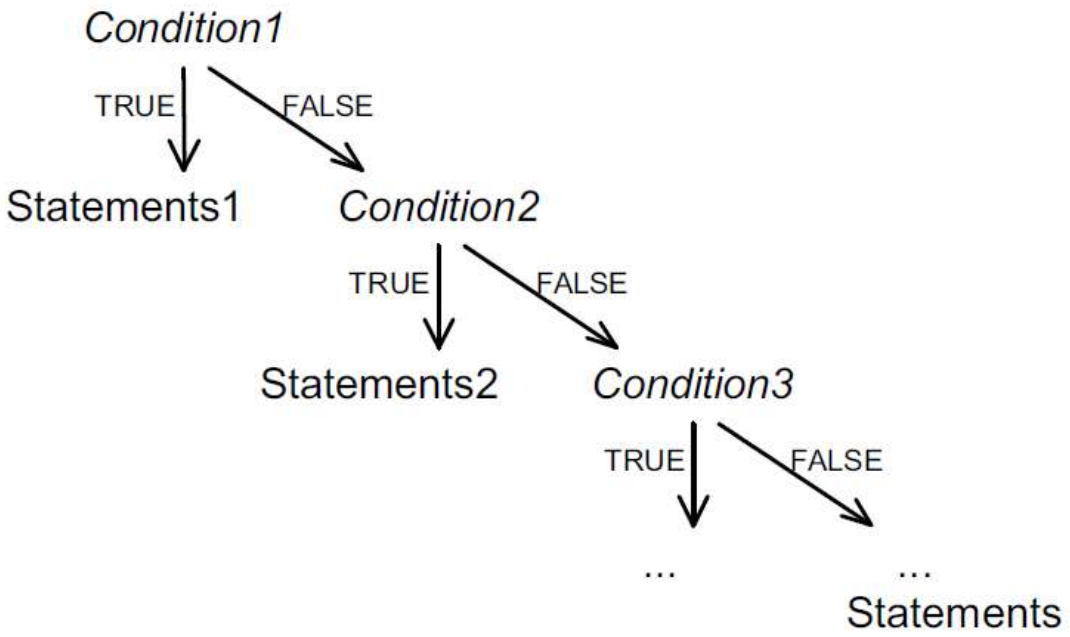
- **Selection**
  - `IF`
  - `CASE` (Multi-selection)
- **Iteration**
  - `FOR`
  - `WHILE`
  - `REPEAT`
  - `EXIT` (Premature termination of an iteration statement)
- `Call of a FB`
- `RETURN` (Leave the current POU and return to the calling POU)

ST does not include a jump instruction (GOTO). All conditional jumps can also be programmed via an IF structure

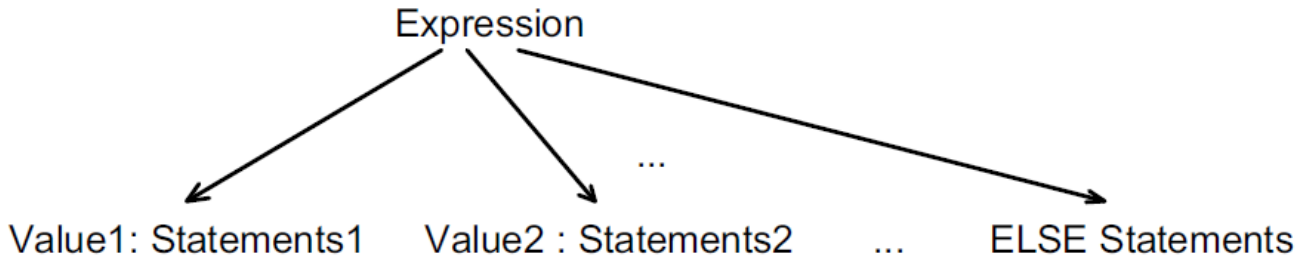
Selection statements

These two statement types are used to execute specific statements depending on a Boolean condition.

- Selection



- Multi-selection



Selection statement IF

```
IF Condition1 THEN
   Statements1;                    (* Execute Statements1, if Condition1 is TRUE,
                                      continue after End_IF  otherwise: *)

ELSIF Condition2 THEN
   Statements2;                    (* Execute Statements2, if Condition2 is TRUE,
                                      continue after End_IF  otherwise: *)

ELSIF Condition3 THEN

...

ELSE Statements;                   (* Execute Statements if no previous condition
                                      evaluates to TRUE *)

END_IF;                            (* End of IF *)
```

Selection statement IF

Types definition

```
TYPE
    impulse : UINT(0..1000);
    state : (stop, run, fault, wait);
    temperature : REAL := 20.0;
    temperature_sensor: STRUCT
        • value : temperature;
        • last_calibration: DATE;
        • calibration_interval: TIME;
    END_STRUCT
END_TYPE
```

Selection statement IF

Variables definition

```
VAR
        enable : BOOL;
        count : impulse;
        valve_state : state;
        thermometer : temperature := 0.0;
        thermocouple1, thermocouple2 : temperature_sensor;
END_VAR
```

Selection statement IF

IF selection

```
IF enable & (count < 100) THEN
      count := 100;
END_IF

IF enable THEN
      thermocouple1.value := thermocouple2.value;
ELSE
      thermocouple1.value := 0.0;
END_IF;
```

Selection statement IF

IF selection

```
IF count <= 1 THEN
      valve_state := stop;
ELSEIF conteggio < 6 THEN
      valve_state := run;
ELSEIF conteggio < 50 THEN
      valve_state := wait;
ELSE                                    (* count >= di 50 *)
      valve_state := fault;
END_IF;
```

Multi-selection statement CASE

```
CASE VarInteger OF
   1:       Statements1; (* Execute Statements1, if VarInteger is TRUE,
                            continue after End_CASE otherwise:
   *)

   2,3:     Statements2; (* Execute Statements2, if VarInteger is 2 or 3,
                            continue after End_CASE otherwise: *)

   10..20: Statements3; (* Execute Statements3, if VarInteger is between
   10                       and 20, continue after End_CASE otherwise:
   *)

ELSE     Statements;      (* Execute Statements if no comparison
succeeded *)

END_CASE;                 (* End_of_CASE *)
```

Multi-selection statement CASE

CASE selection

```
CASE count OF
   1 : valve_state := stop;
   2, 3, 4, 5 : valve_state := run;
   6..50 : valve_state:= wait;
ELSE
   valve_state := fault;
END_CASE;
```

The Program Organisation Unit (POU)

A POU is an encapsulated unit

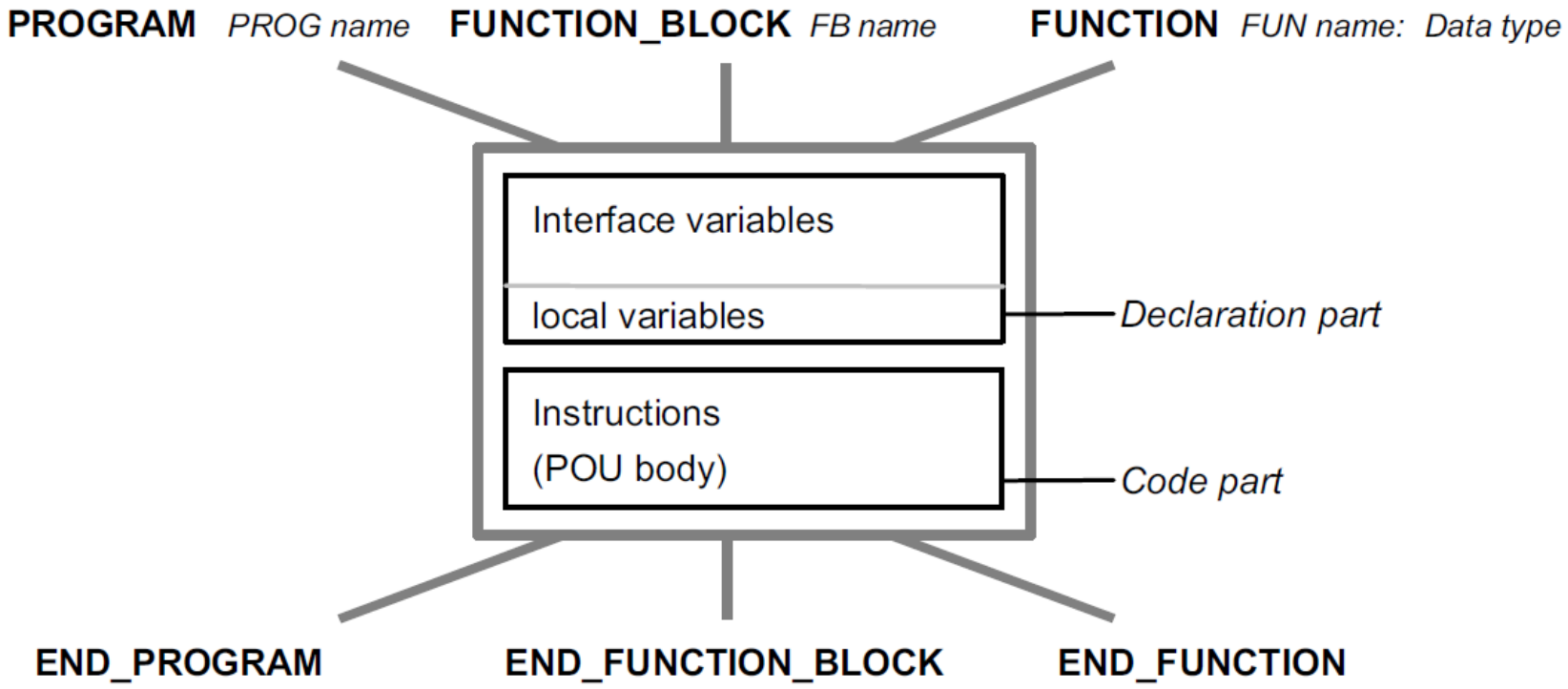
- It can be **compiled independently** of other program parts
- It can be **re-used** inside the program
- It facilitates **modularization** of tasks

POU type	Keyword	Meaning
Program	PROGRAM	Main program. It can include assignment to I/O, global variables, and access paths
Function Block	FUNCTION_BLOCK	Block with input and output variables; can have static variables (with memory)
Function	FUNCTION	Has no static variables (without memory)

Components of a POU

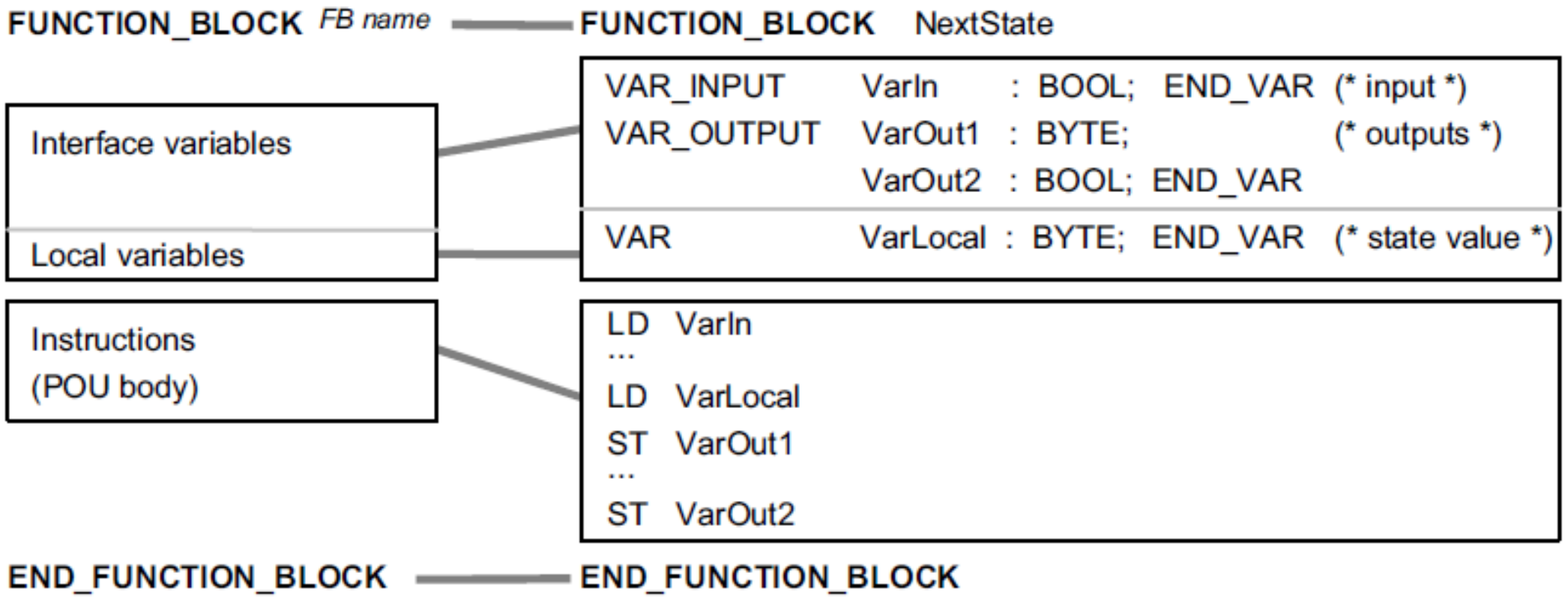
POU consists of the following elements:

- **POU type and name** (and data type in the case of functions)
- **Declaration part** with variable declarations
- **POU body** with instructions



Components of a POU

Function block in IL



Functions

Functions have any number of input and output parameters and exactly **one function (return) value**

- It can be of **any data type**, including derived data types
- It is the **name of the function itself**, and can be used in expressions

A function works **without memory**

- It always return the **same result** when provided with the same input parameters (does not depend on internal variables)

Function can only call other function (**recursion is not allowed**)

Functions declaration

Declarations and instructions can be programmed in graphical or textual form

Textual declaration

```
FUNCTION function_name: type          (* type of the function *)
   VAR_INPUT                          (* input variable definition *)
         ....;
   END_VAR
   ...                        (* other variables definitions *)
   ...                        (* function body *)
                              (* a value must be assigned to function_name *)
END_FUNCTION
```

Functions declaration

```
FUNCTION threshold_saturated: REAL
   VAR_INPUT
   data, lim_threshold, lim_saturation: REAL;
   END_VAR

   IF ABS(data) < lim_threshold THEN
        threshold_saturated:= 0.0;
   ELSE
        threshold_saturated:=
        MIN(MAX(data, - lim_saturation), lim_saturation);
   END_IF
END_FUNCTION
```

Functions declaration

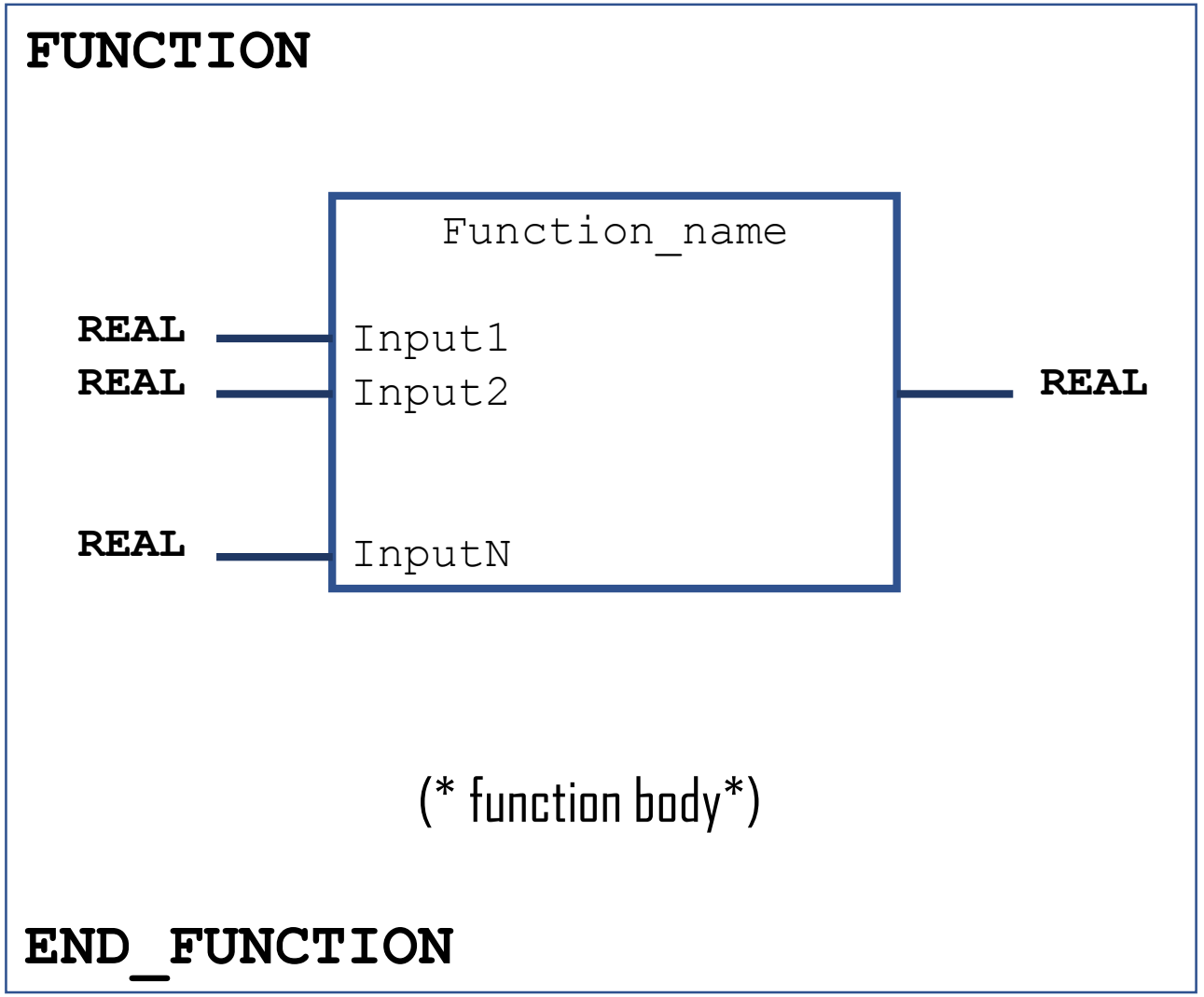
Textual declaration

- The body of the function can be programmed in any language of the standard (except SFC)
- Only local and input and output variables are permitted (no external, no global, no retain, no direct access)

Functions declaration

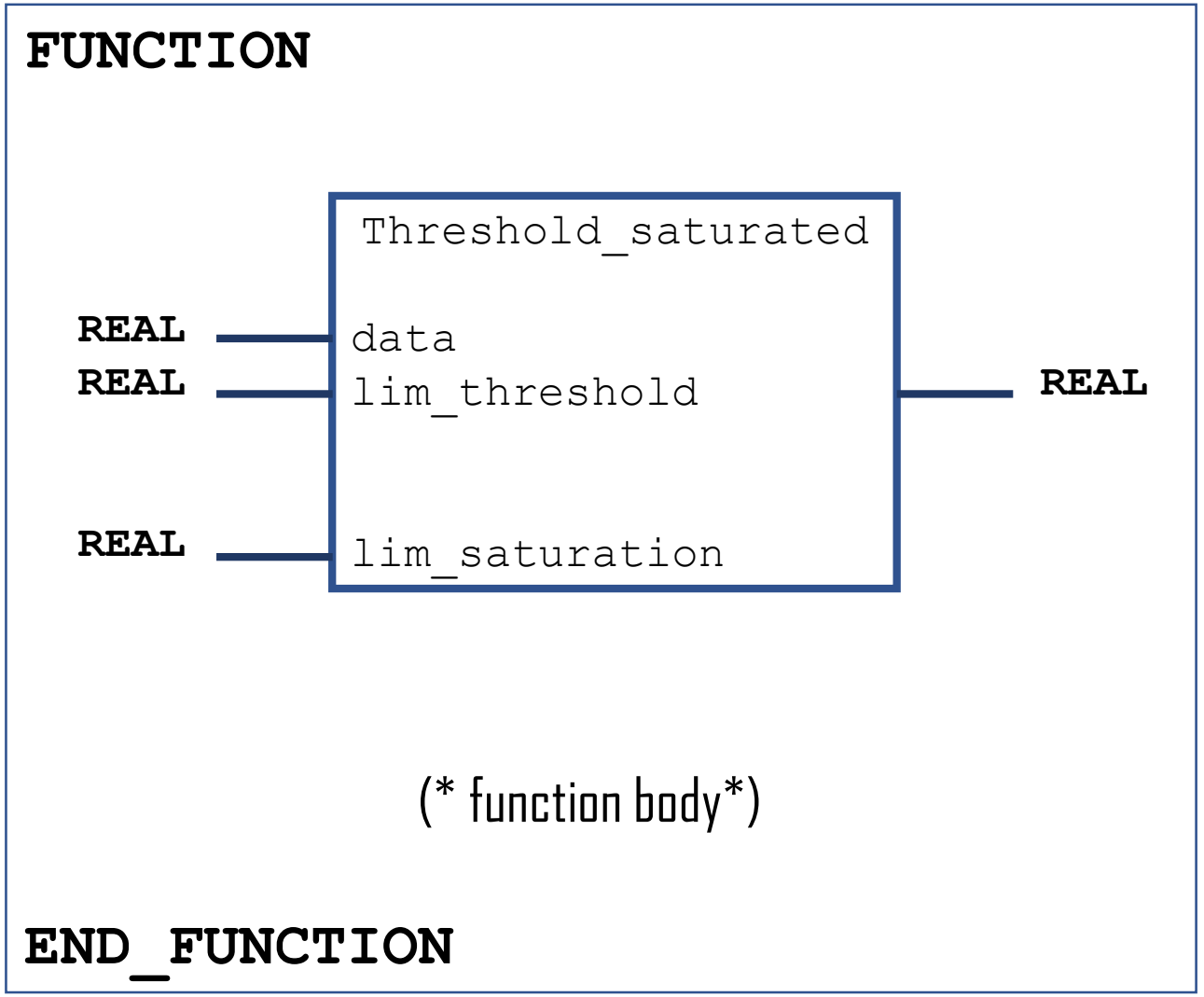
Graphical declaration

- The function interface is a **rectangular block**
- Internal variable must be declared
- The function body programmed in one of the standard language



Functions declaration

Graphical declaration



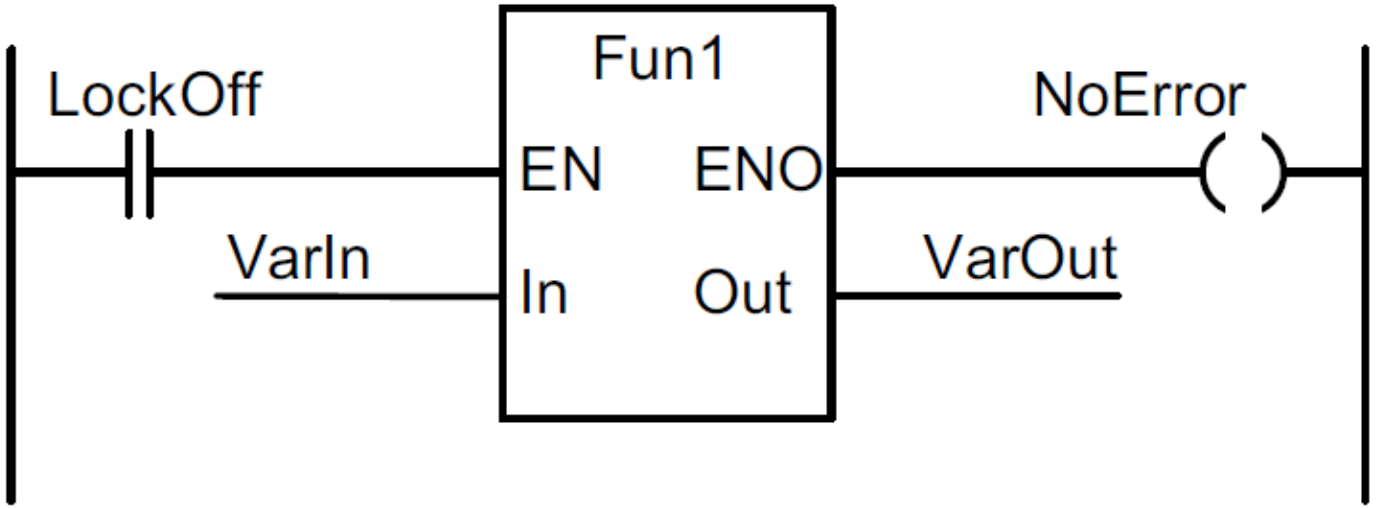
Functions declaration

Graphical declaration

In the ladder diagram LD, functions have a special feature

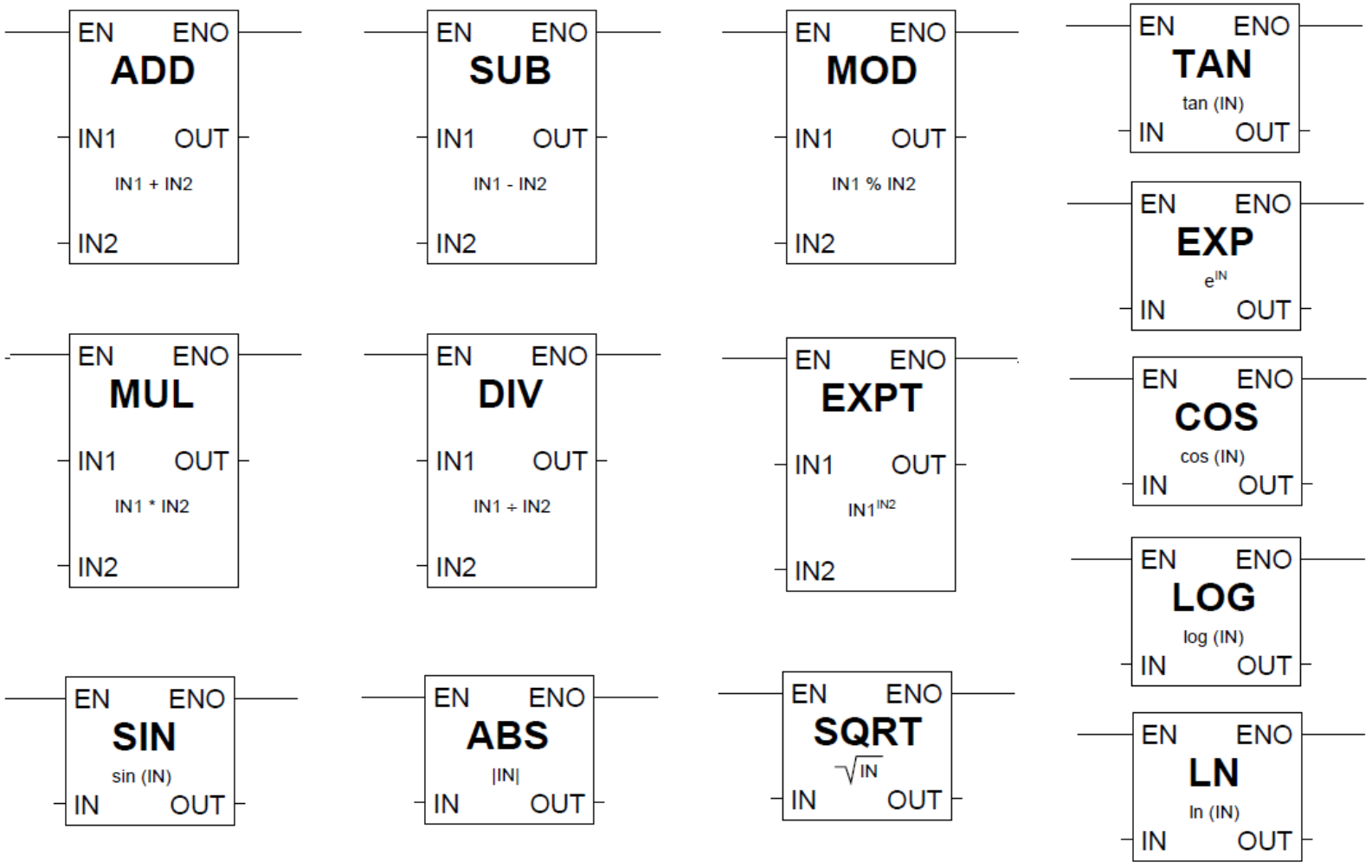
- Boolean input **EN** (Enable In)
- Boolean output **ENO** (Enable Out)

Can be used in **concatenated functions call**



Functions

- There are some predefined functions (ADD, SUB, MAX, MIN, MUX, LIM, GT/LT/GE/LE/EQ/NE, AND/OR/NOT/XOR, LN, INSERT…)



Function Block

FBs can be assigned parameters and has static variables (work **with memory**).

- When invoked with the same input parameters, will yield values which **depend also on an internal state**
- The value of its internal (VAR) and external (VAR_EXTERNAL) variables **are retained from one execution** of the function block **to the next**

Declarations and instructions can be programmed in graphical or textual form

- Keywords: FUNCTION_BLOCK … END_FUNCTION_BLOCK
- The body of the function can be programmed in any language of the standard (except SFC)

FBs can call functions and other FBs (**recursion is not allowed**)

Function Blocks declaration

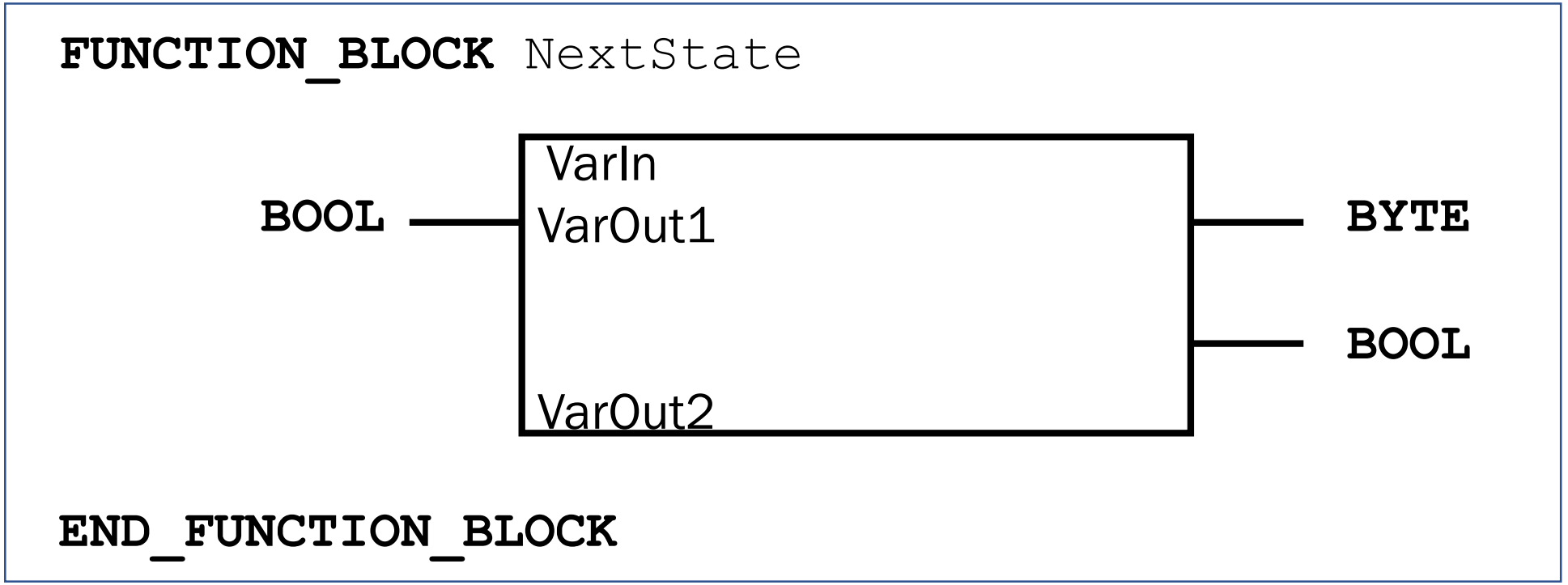
Textual declaration

```
FUNCTION_BLOCK function_block_name:
   VAR_INPUT                                   (* input variable definition *)
         ....;
   END_VAR
   ...                                  (* other variables definitions *)
   ...                                  (* function body *)

END_FUNCTION_BLOCK
```

Functions block declaration

Graphical declaration

- The function block interface is a **rectangular block**



Functions block instances

When a function block is defined also an **instance is create** (like variables)

- After instantiation **an FB can be used (as an instance)** and called within the POU in which it is declared
- Can have GLOBAL and RETAIN attribute

Instantiation is the creation of variables by the programmer by specifying the variable's name and data type in the declaration.

```
Motor1, Motor2   :   MotorType;   (* FB instance *)
```

Names of FB instances

FB type (user-defined)

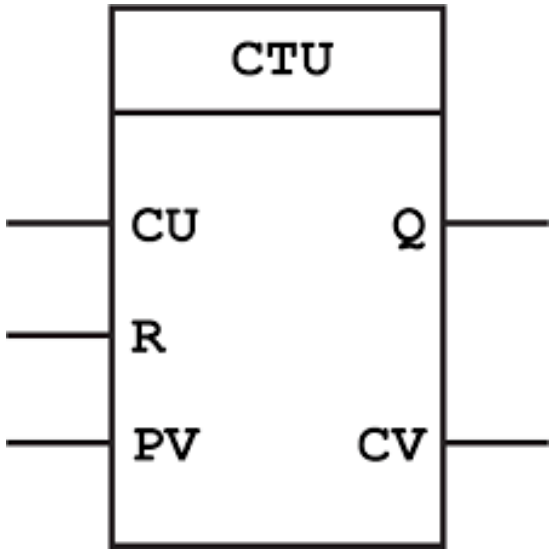
Functions block instances

The concept of instantiation results in **structured variables**

- describe the **FB calling interface** like a data structure
- the **user can** only **access** to the **input/output parameters**
- local or external variables are kept hidden.

Declaration of an up counter (standard FB)

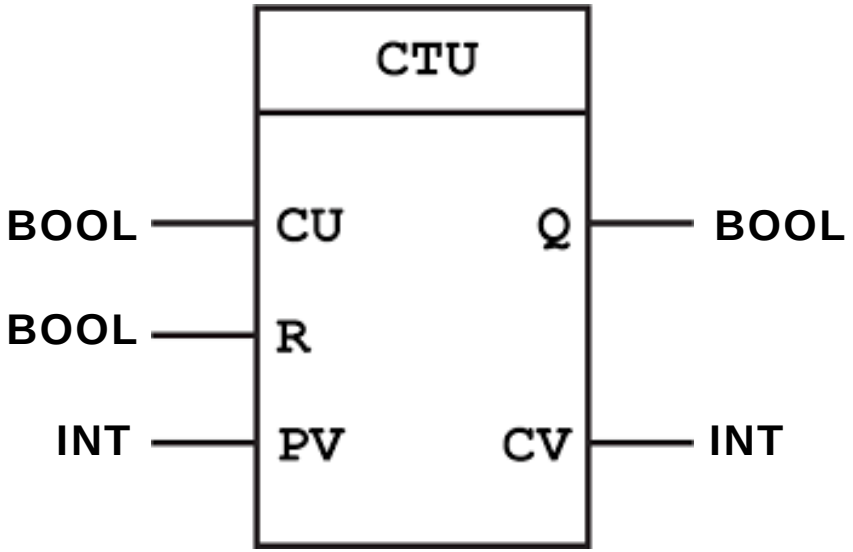
```
VAR
   Counter : CTU;          (* up counter *)
END_VAR
```



Functions block instances

Data structure of an FB instance of FB type CTU

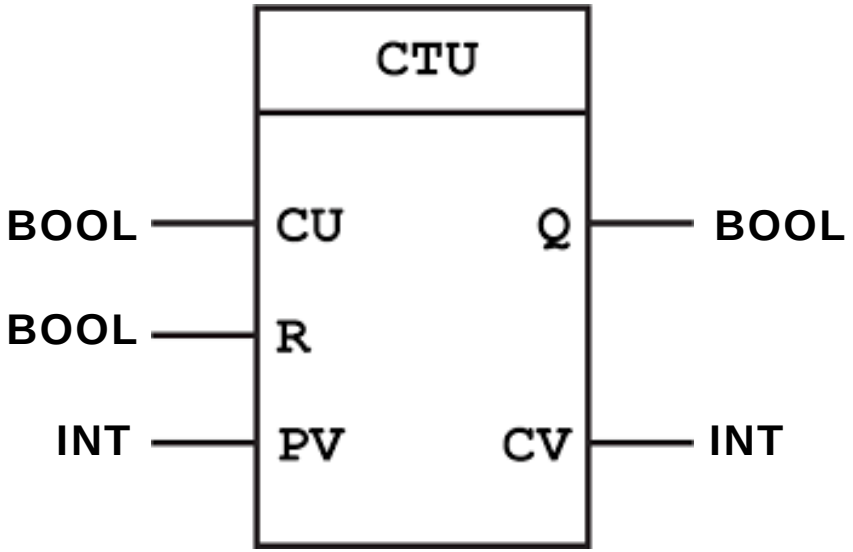
```
TYPE CTU :
   STRUCT
      (* inputs *)
      CU  : BOOL;      (* count up *)
      R   : BOOL;      (* reset *)
      PV  : INT;       (* preset value *)
      (* outputs *)
      Q   : BOOL;      (* output up *)
      CV  : INT;       (* current value *)
   END_STRUCT;
END_TYPE
```



Functions block instances

Up counter, FB type CTU

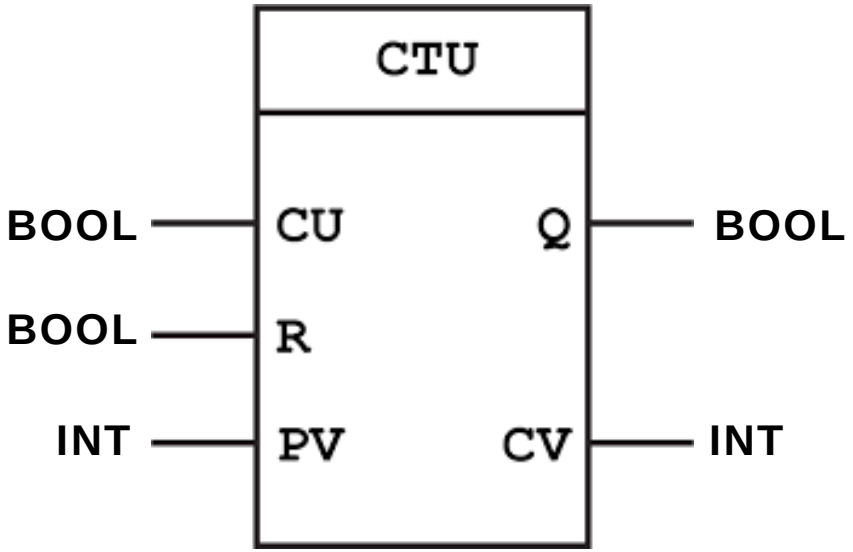
```
FUNCTION_BLOCK CTU
    VAR_INPUT
        CU : BOOL R_TRIG;    (* rising edge trigger*)
        R : BOOL;            (* reset signal *)
        PV : INT;            (* max counting value *)
    END_INPUT_VAR

    VAR_OUTPUT
        Q : BOOL;            (* end of count *)
        CV : INT;            (* counter courrent value*)
    END_VAR
    …
```



Functions block instances

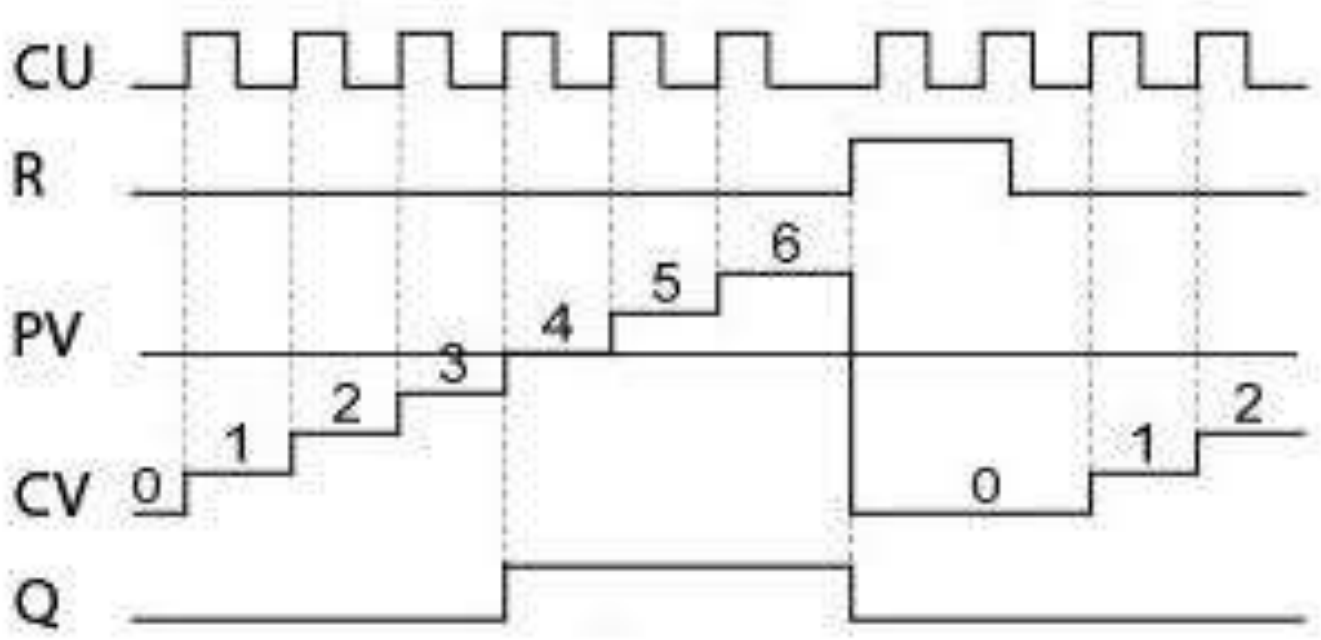
Up counter, FB type CTU

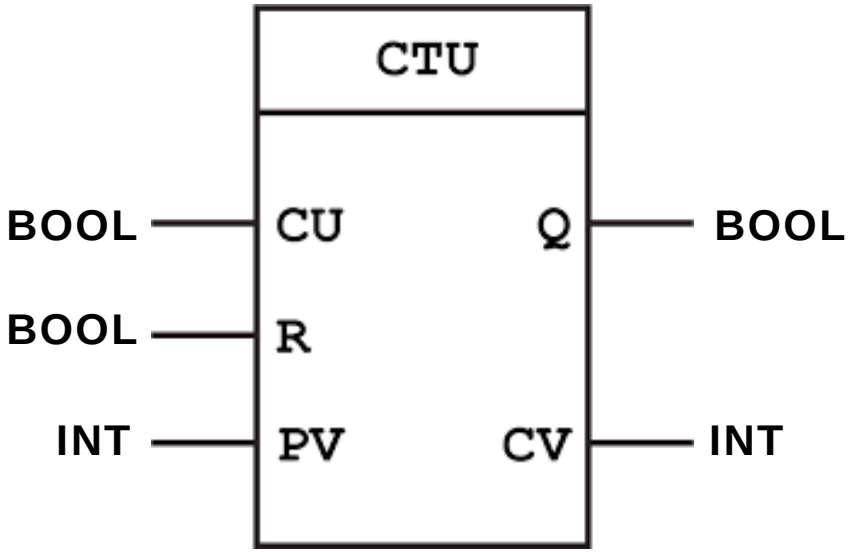
```
VAR_RETAIN
    AUX : BOOL := 0;
END_VAR

IF R THEN
        CV := 0;
ELSEIF CU AND (CV < PV) THEN
        CV : = CV + 1;
ENDIF
Q = (CV = PV);

END_FUNCTION_BLOCK
```



Functions block instances

Up counter, FB type CTU

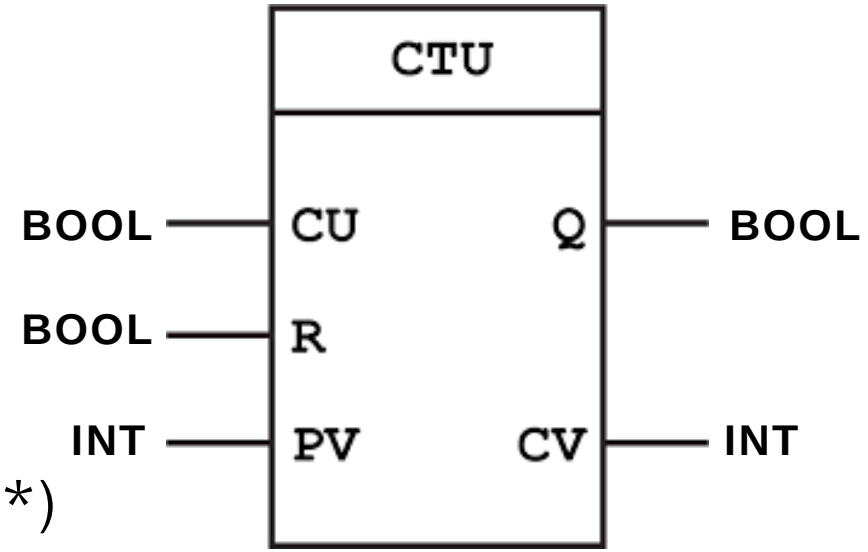




Functions block instances

Parameterization and invocation of the up counter in IL

```
LD    34
ST    Counter.PV      (* preset count value *)
LD    %IX7.1
ST    Counter.CU      (* count up *)
LD    %M3.4
ST    Counter.R       (* reset counter *)
CAL   Counter         (* invocation of FB *)
LD    Counter.CV      (* get current count value *)
```



Programs

The main program

- All variables of the whole program that are assigned to physical I/O addresses of the PLC (%Q, %I, %M) can be access and must be declared in this POU
- In all other aspects it behaves like an FB

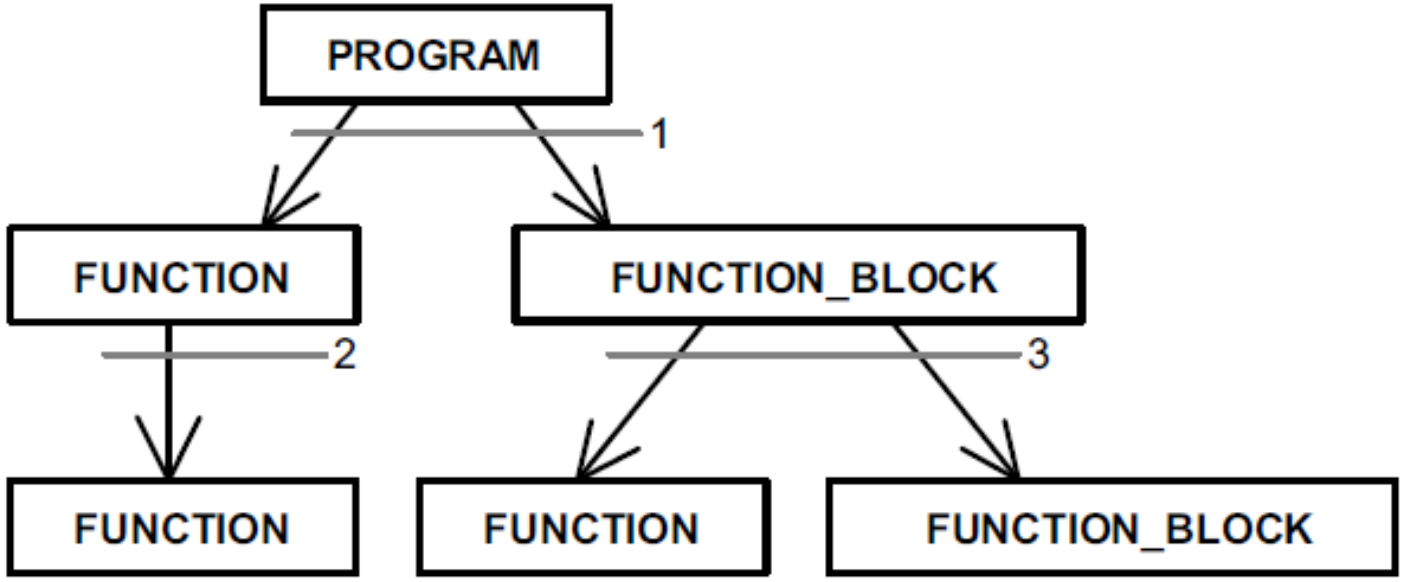
Program declaration

```
PROGRAM name
   VAR_INPUT...END_VAR
   VAR_OUTPUT...END_VAR
   ...(* other variables and body*)
END_PROGRAM
```

Mutual calls of POUs

Recursive calls are invalid

- it would not be possible for the programming system to calculate the maximum memory space needed



Function Block Diagram (FBD)

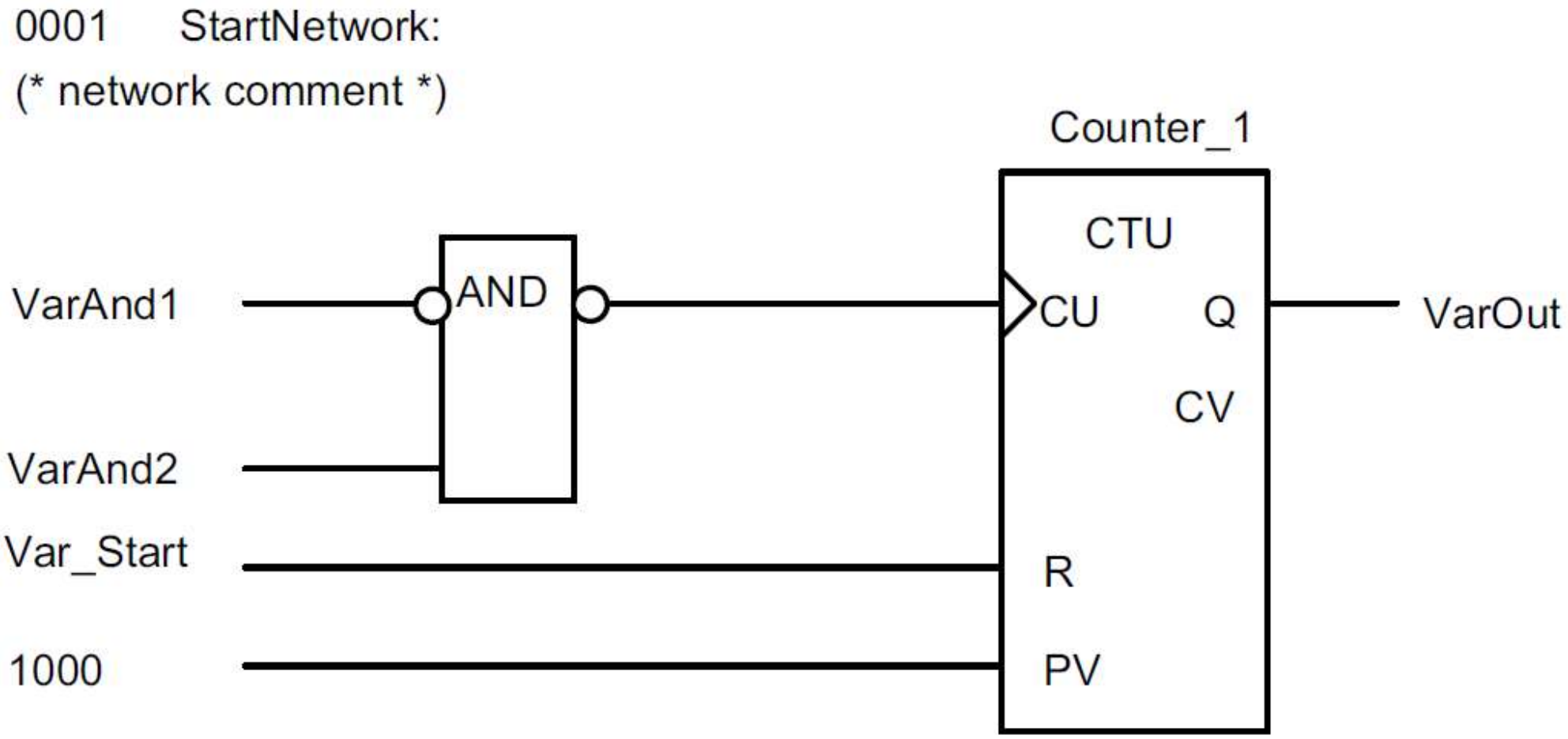
The graphical elements of an FBD network include **rectangular boxes and control flow** statements connected by horizontal and vertical lines

Each box can be seen as a **black box**

- Processes some inputs and returns some outputs
- Can be implemented in different language

Signals **travel along the connections** between the boxes (from left to right)

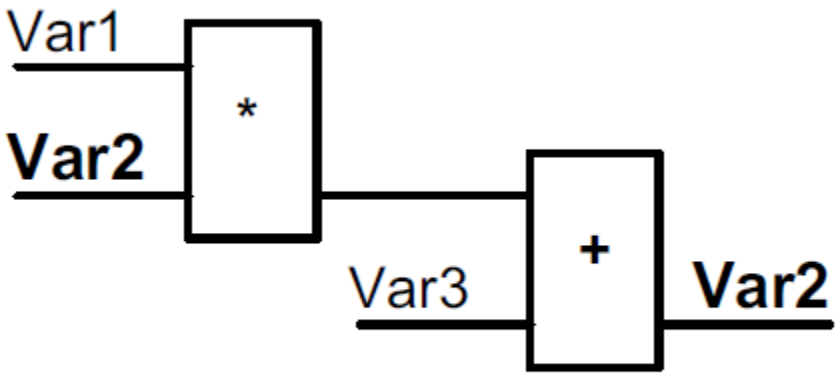
Function Block Diagram (FBD)

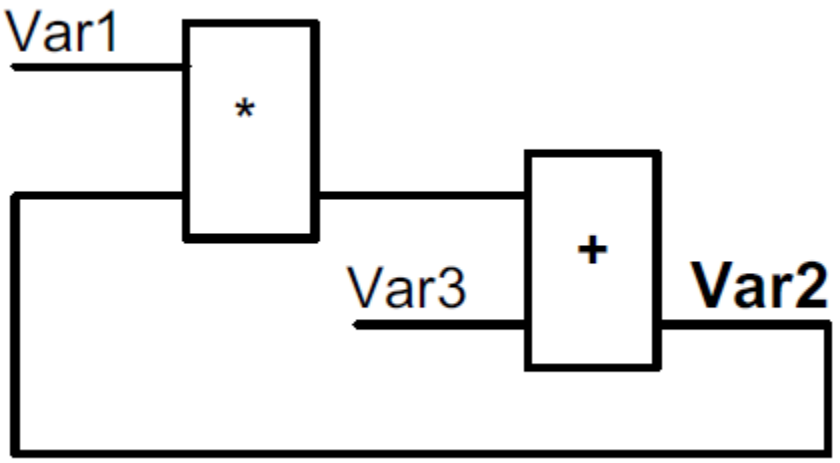


Function Block Diagram (FBD)

The value of an output parameter **can flow back to an input** parameter of the same network

- lines are called **feedback paths**
- the associated variables are called **feedback variables**





Sequential Function Chart (SFC)

SFC is born for implementing sequential control algorithms. It was defined to

- **break down a complex program** into smaller manageable units
- **describe the control flow** between these units

SFC has been derived from well-known techniques like **Petri-net** (discrete-event system)

- it is possible to design **sequential and parallel processes**

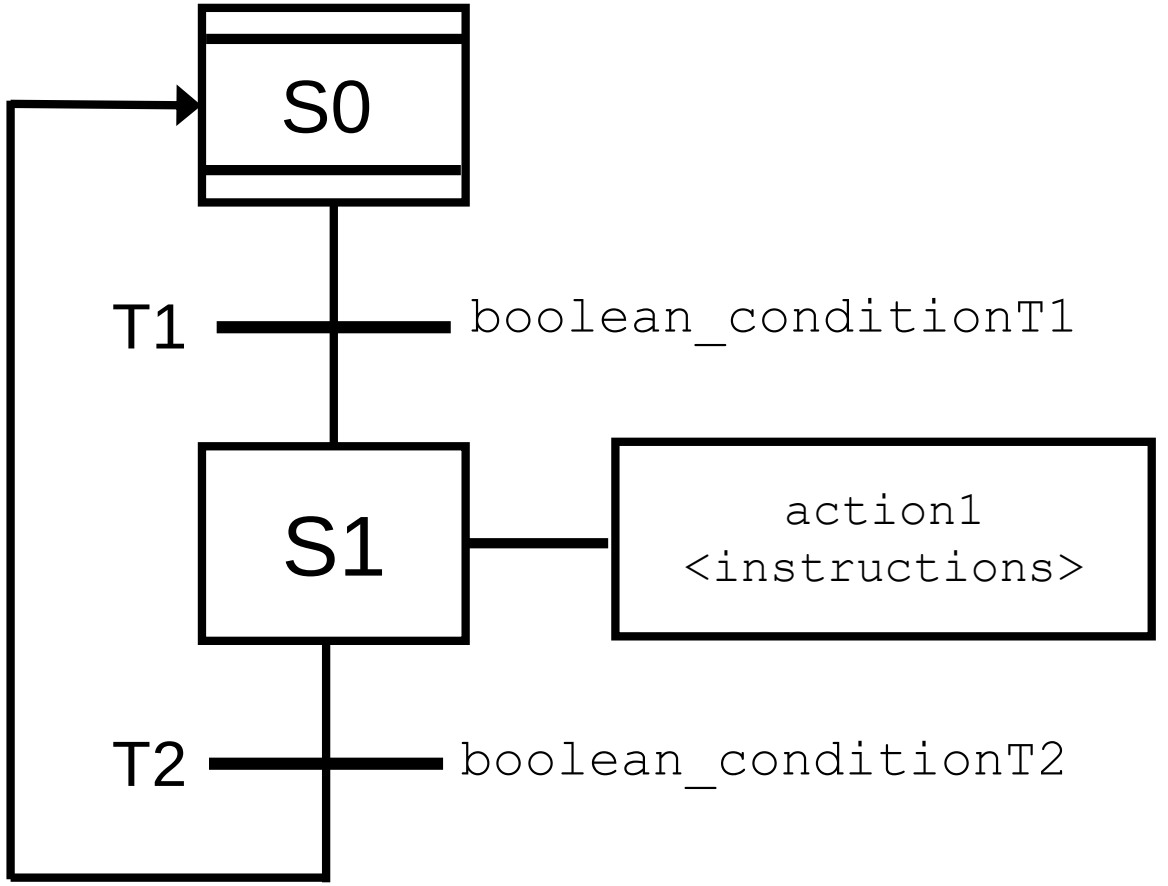
Main advantage → the **implementation** of a control system **matches the description of the desired process behavior**

SFC elements

- Steps
- Transitions
- Actions

Transition condition may be programmed in IL, ST, LD or FBD, but must produce a Boolean value. When evaluated to TRUE

- stops the step that was active
- activates the next successor step(s).



Ladder Diagram (LD)

The language comes from the field of **electromechanical relay systems**

- Transition from wired logic to programmable logic controller
- designed for processing Boolean signals

The name comes from the **ladder network shape**

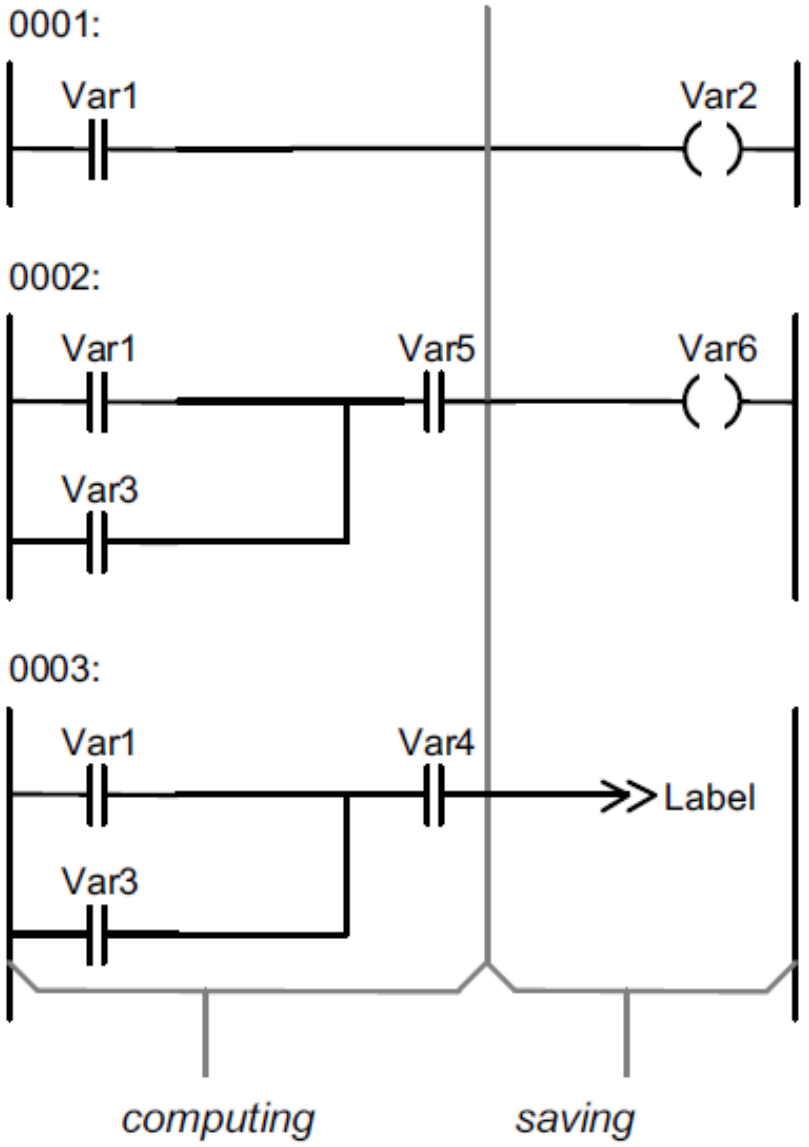
- **bounded by** so-called **power rails** on the left and on the right
- describes the power **flow through the network from left to the right** and instructions are executed from top to bottom

Ladder Diagram (LD)

REMEMBER:

- From **left (test area)** to the **right (action area)**
- From **top to bottom**

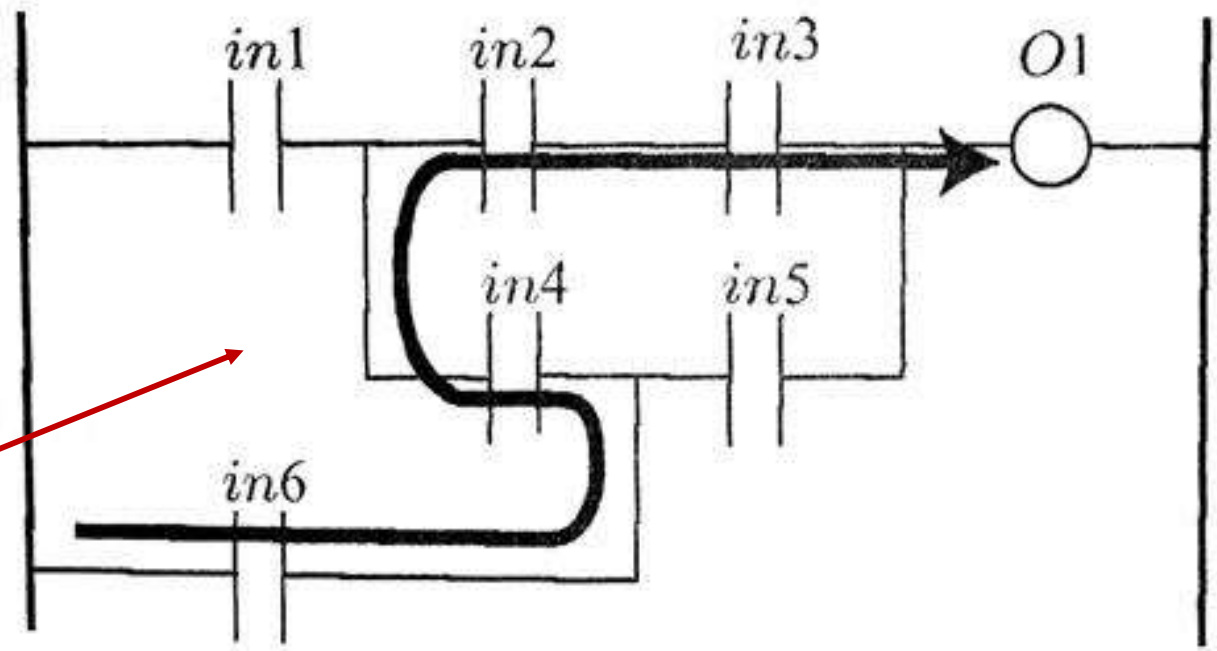
A **coil**, in the **saving area** (right side), is fed if a combination of **contacts** in the **computing area** (left side) allows the passing of network power



Ladder Diagram (LD)

REMEMBER:

- From **left (test area)** to the **right (action area)**
- From **top to bottom**

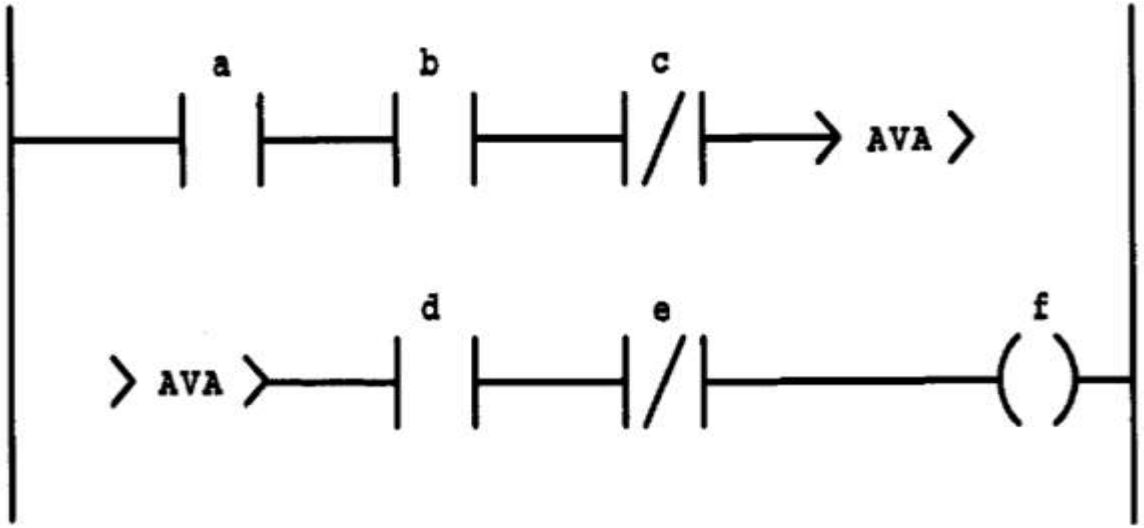


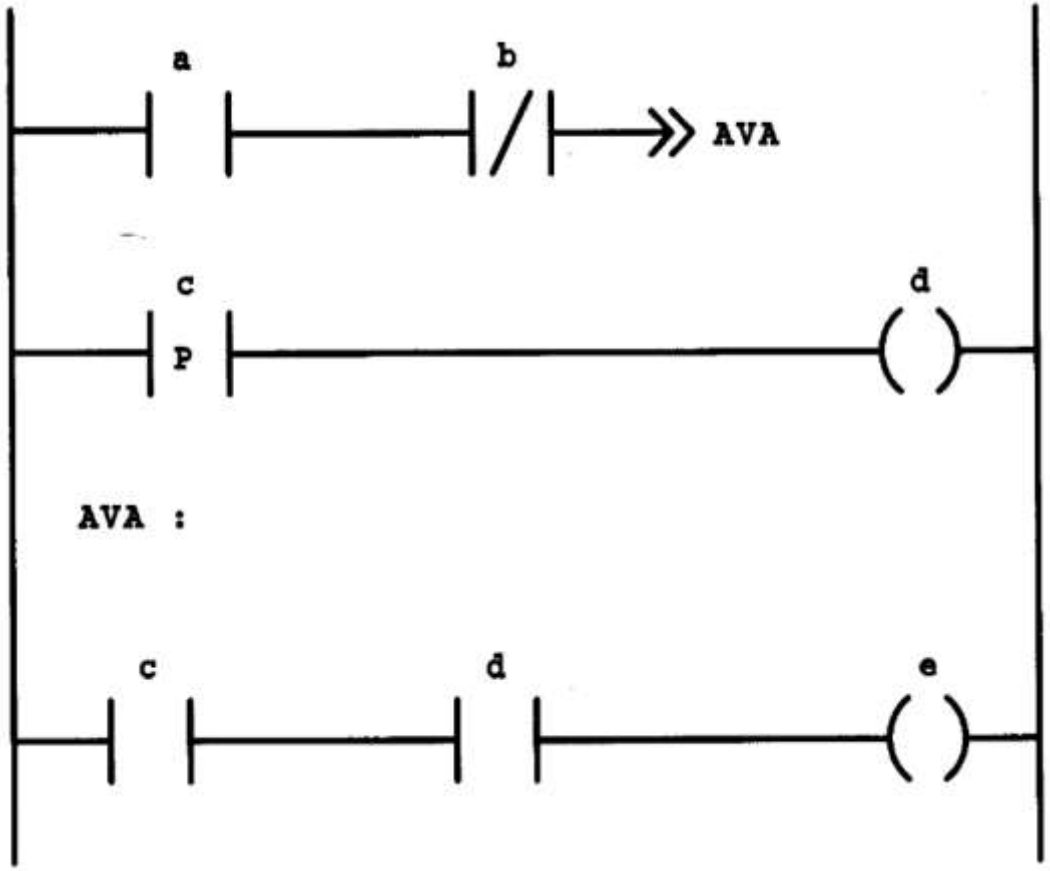
NOT ALLOWED FLUX

Ladder Diagram (LD)

Execution control with labels

- Jump to a target rung
- Split long rung



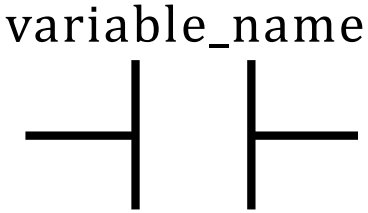


LD principal components

Contacts

Can be associated with Boolean variables (bits)

- **Internal**
- **External** (e.g., sensor)



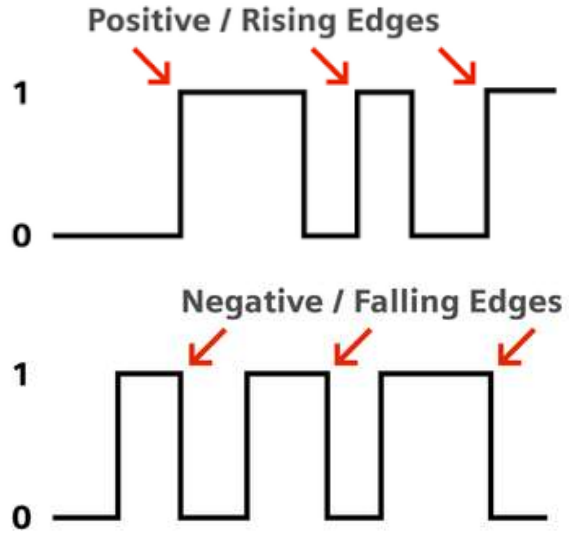
Are used to **evaluate the value of the variables** with which they are associated (usually positioned on the top)

A closing contact in the evaluation area (left side) allows the power flow to the saving area (right side)

LD contacts

- | | **normally open**
  - is closed if the associated bit is 1 (TRUE)
- |/| **normally closed**
  - is closed if the associated bit is 0 (FALSE)
- |P| **positive transition sensing**
  - is closed when the associated bit goes from 0 to 1 (FALSE → TRUE)
- |N| **negative transition sensing**
  - is closed when the associated bit goes from 1 to 0 (TRUE → FALSE)

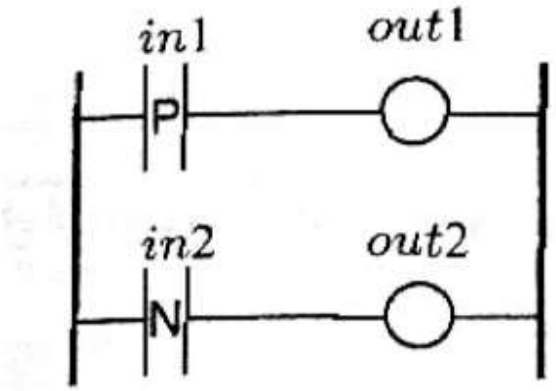
LD contacts



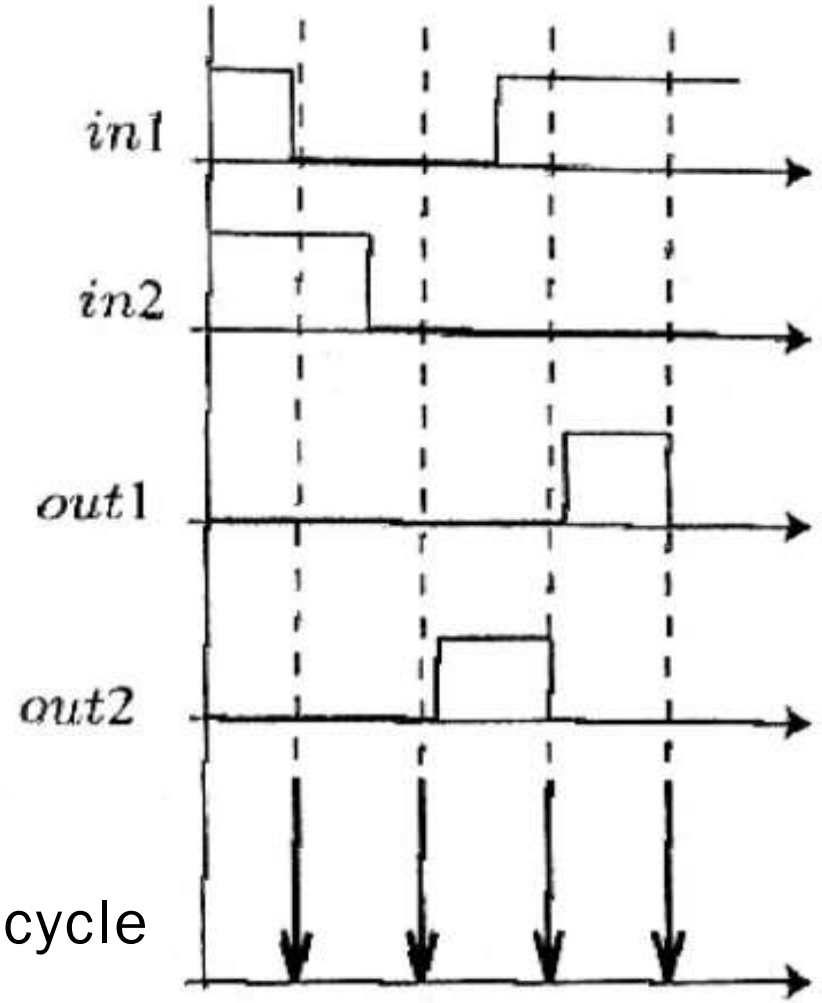
- **| | normally open**
  - is closed if the associated bit is 1 (TRUE)
- **|/| normally closed**
  - is closed if the associated bit is 0 (FALSE)
- **|P| positive transition sensing**
  - is closed when the associated bit goes from 0 to 1 (FALSE → TRUE)
- **|N| negative transition sensing**
  - is closed when the associated bit goes from 1 to 0 (TRUE → FALSE)

LD contacts

Transition sensing contacts



During a scan cycle, the PLC is ‘blind’



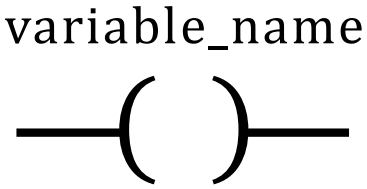
Scan cycle

LD principal components

Coils

Can be associated with Boolean variables (bits)

- **Internal**
- **External** (e.g., actuator)



Are used to **operate on the value of the variables** with which they are associated (usually positioned on the top)

- When the coil is fed it can change the value of the associated variable

LD coils

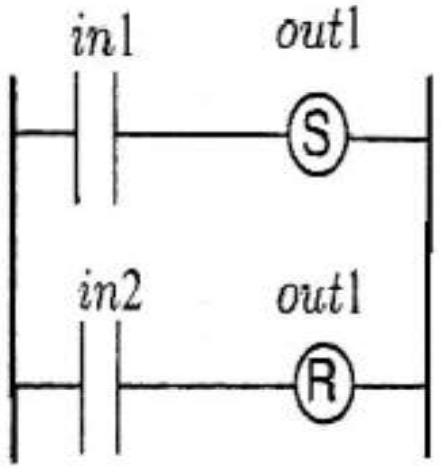
- ( ) **coils**
  - if fed the associated bit is set to 1, otherwise is 0
- (/) **negated coil**
  - if fed the associated bit is set to 0, otherwise is 1
- (S) **SET coil**
  - if fed the associated bit is set to 1 and retains the value 1 even when the coil is not fed anymore
- (R) **RESET coil**
  - if fed the associated bit is set to 0 and retains the value 0 even when the coil is not fed anymore

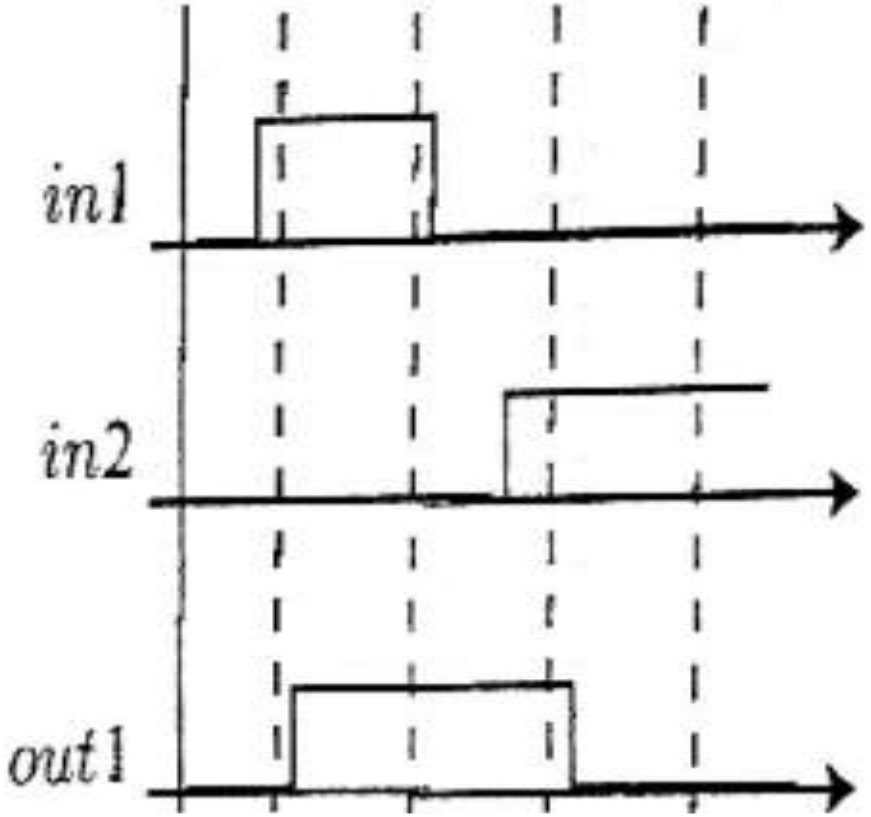
LD coils

- (P) **positive transition sensing**
  - the associated bit is set to 1 when the coil goes **from fed to unfed** (for one scan cycle)
- (N) **negative transition sensing**
  - the associated bit is set to 1 when the coil goes **from unfed to fed** (for one scan cycle)
- (M) (SM) (RM) **with retain**
  - As ( ), (S), (R) but the variable value is retained in case of power loss
  - Similar to defining the associated variable RETAIN

LD coils

SET and RESET coils





After a SET coil, there must be a RESET coil associated with the same variable

SCADA

Supervisory Control And Data Acquisition

A SCADA system is a collection of both **hardware and software components** that allows **supervision and control** of plants, both **locally and remotely**

- Examines, collects and processes data in real-time

They facilitate the interaction of the **operator** of the process with the **field devices** through **Human Machine Interface (HMI)**

- Dashboard that allows an operator to communicate with a machine, computer program, or system

SCADA

Industrial automation hierarchy

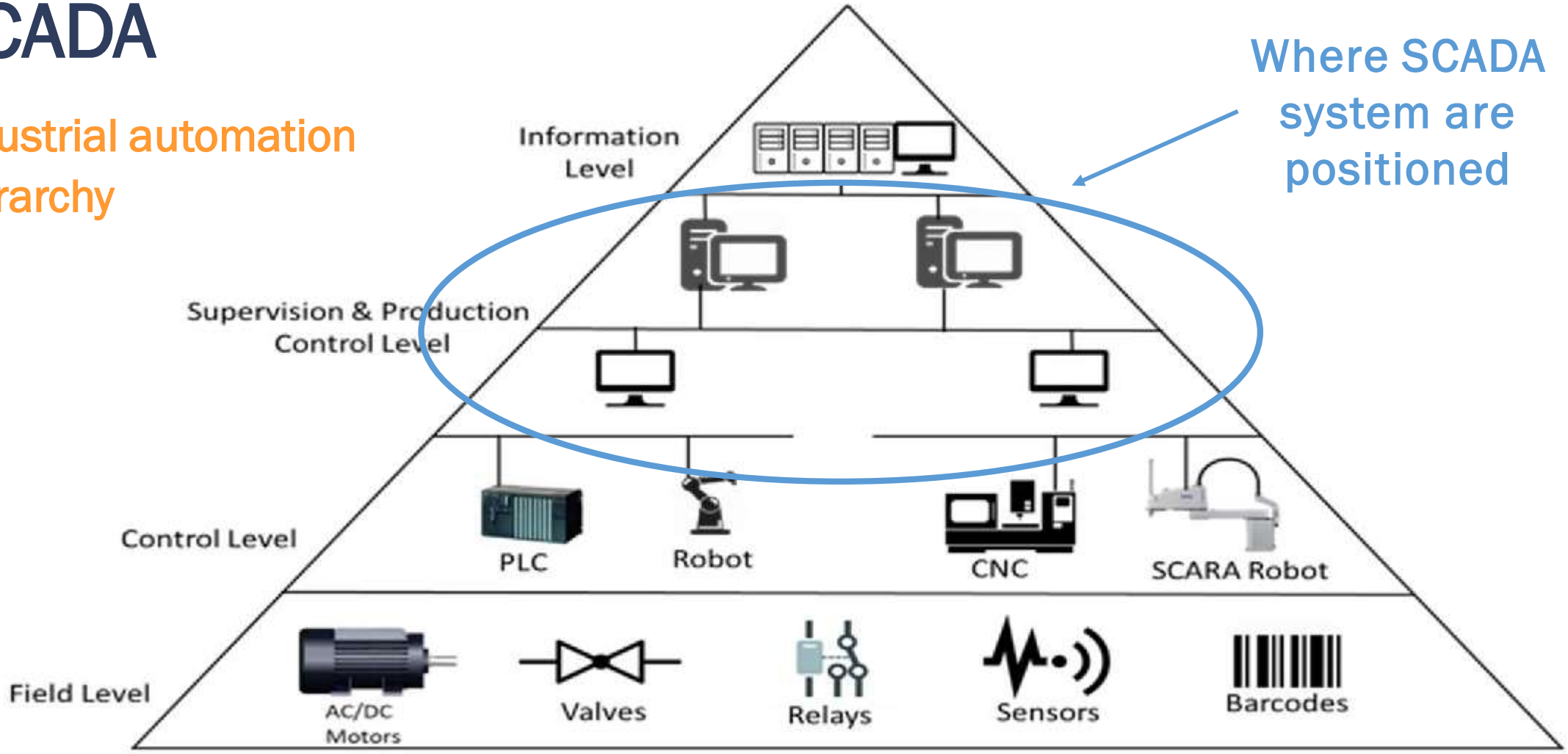
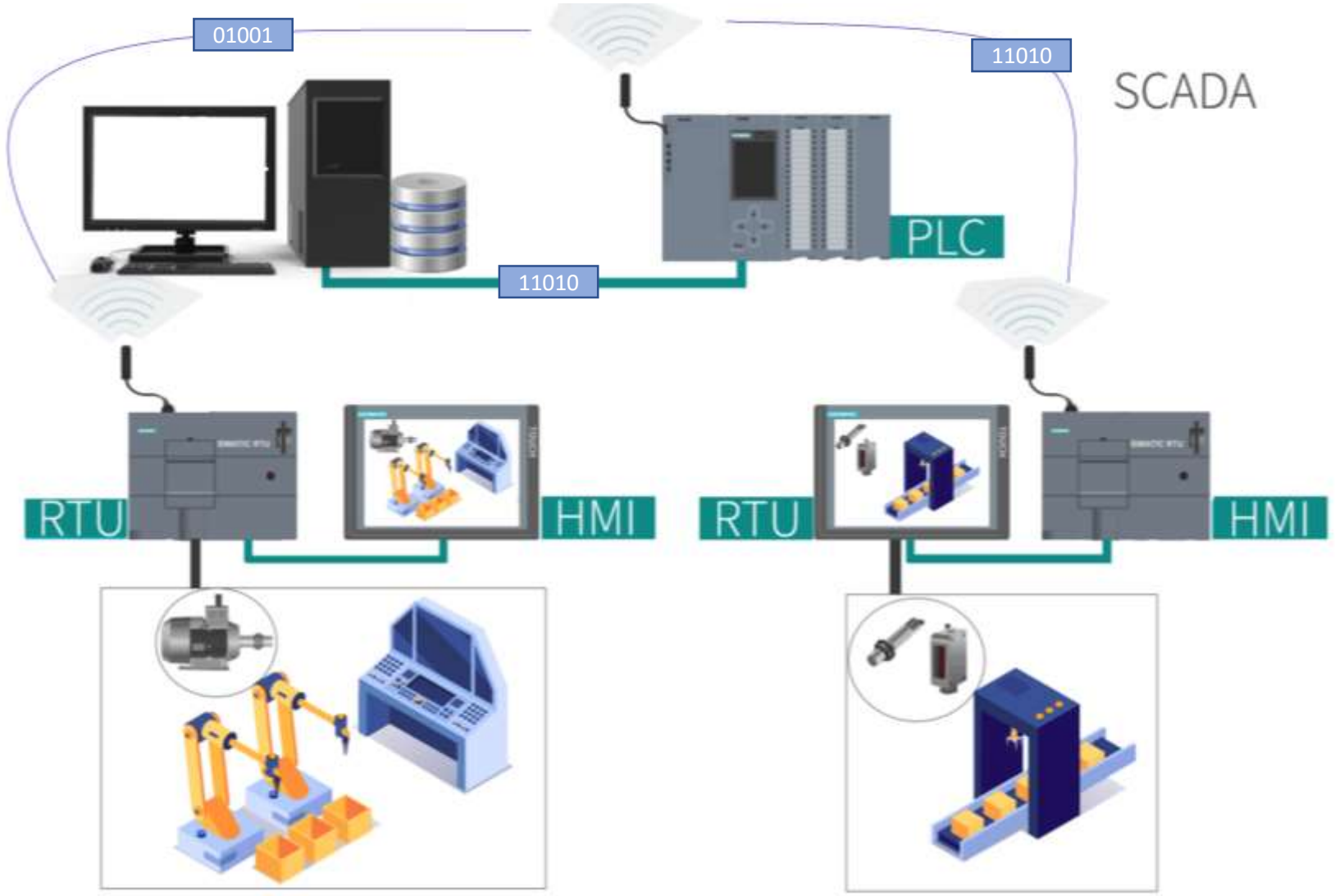


Fig. 2. Hierarchy of industrial automation and control systems

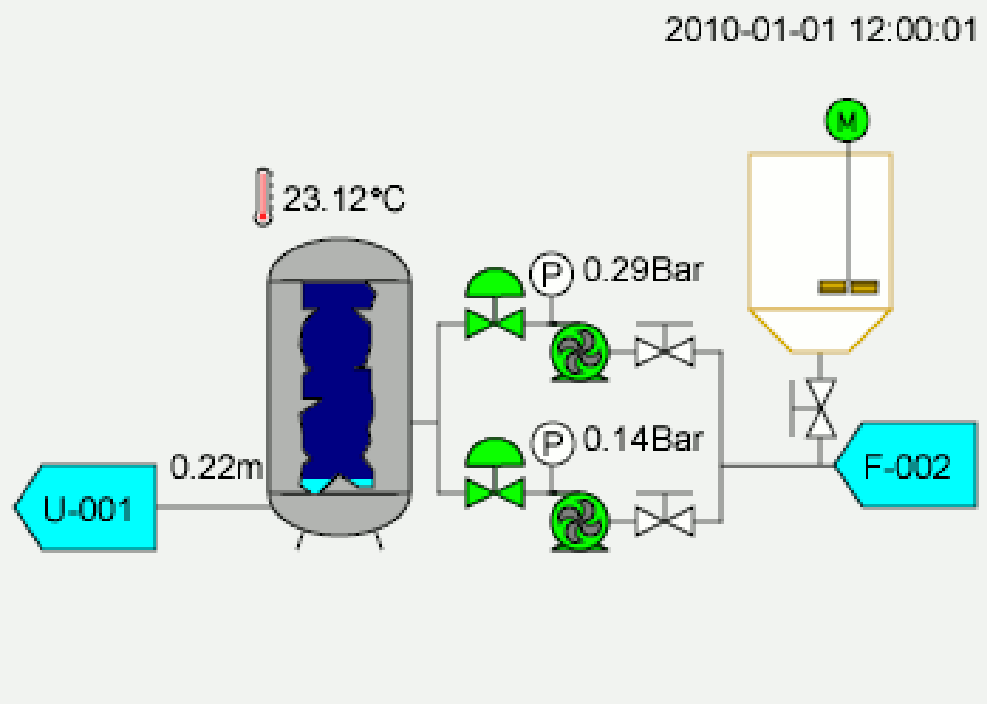
SCADA

The communication data is **routed from the PLC to the SCADA** computers, where the software **interprets and display** the data

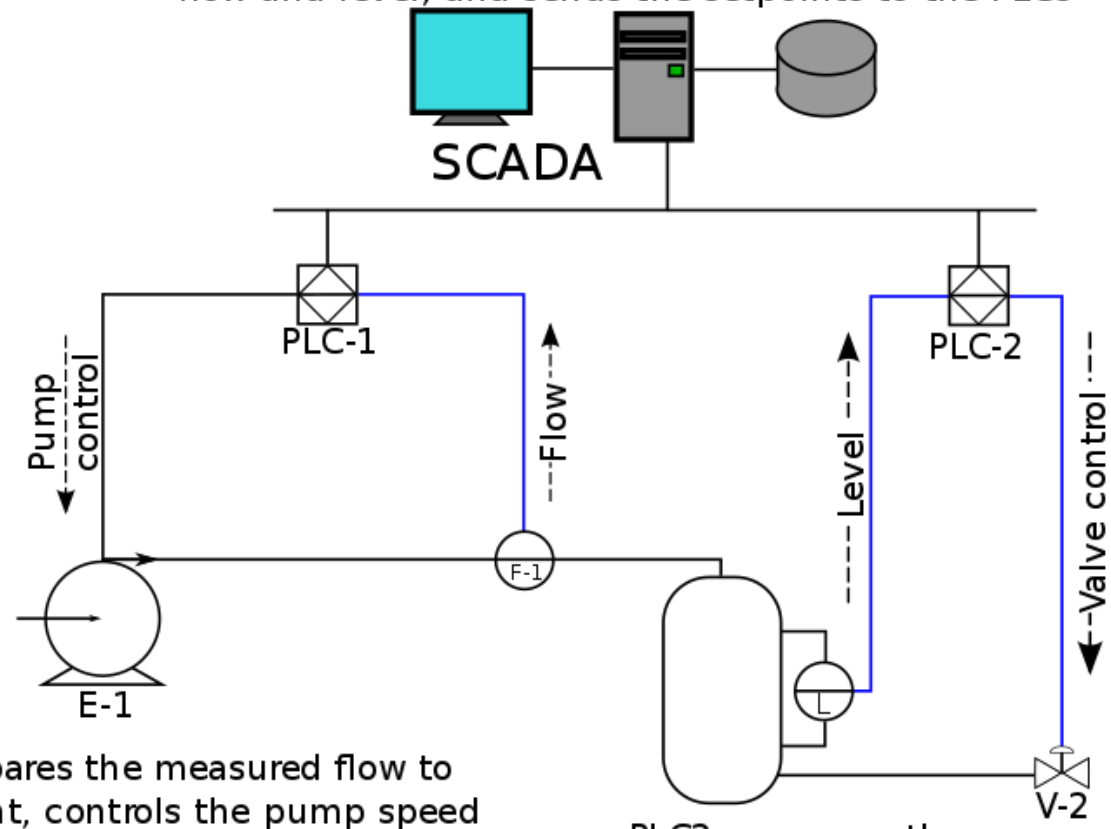


SCADA

SCADA simple system



The SCADA system reads the measured flow and level, and sends the setpoints to the PLCs



PLC1 compares the measured flow to the setpoint, controls the pump speed as required to match flow to setpoint

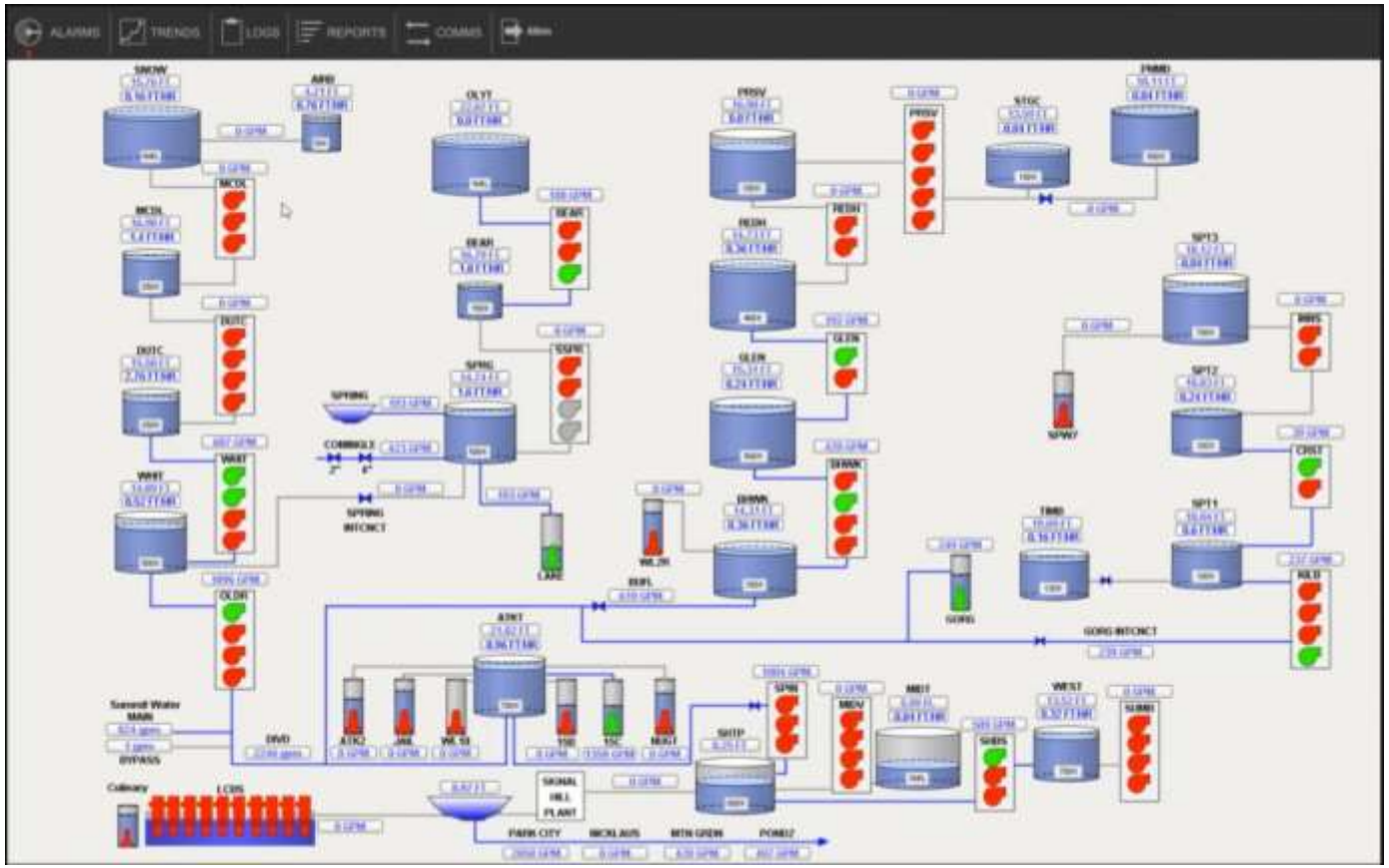
PLC2 compares the measured level to the setpoint, controls the flow through the valve to match level to setpoint

SCADA functionalities

Data representation

A SCADA system can show the collected data to the plant operator using HMI

- **Synoptic chart** with static or dynamic elements
- **Control panel** to interact with the field devices



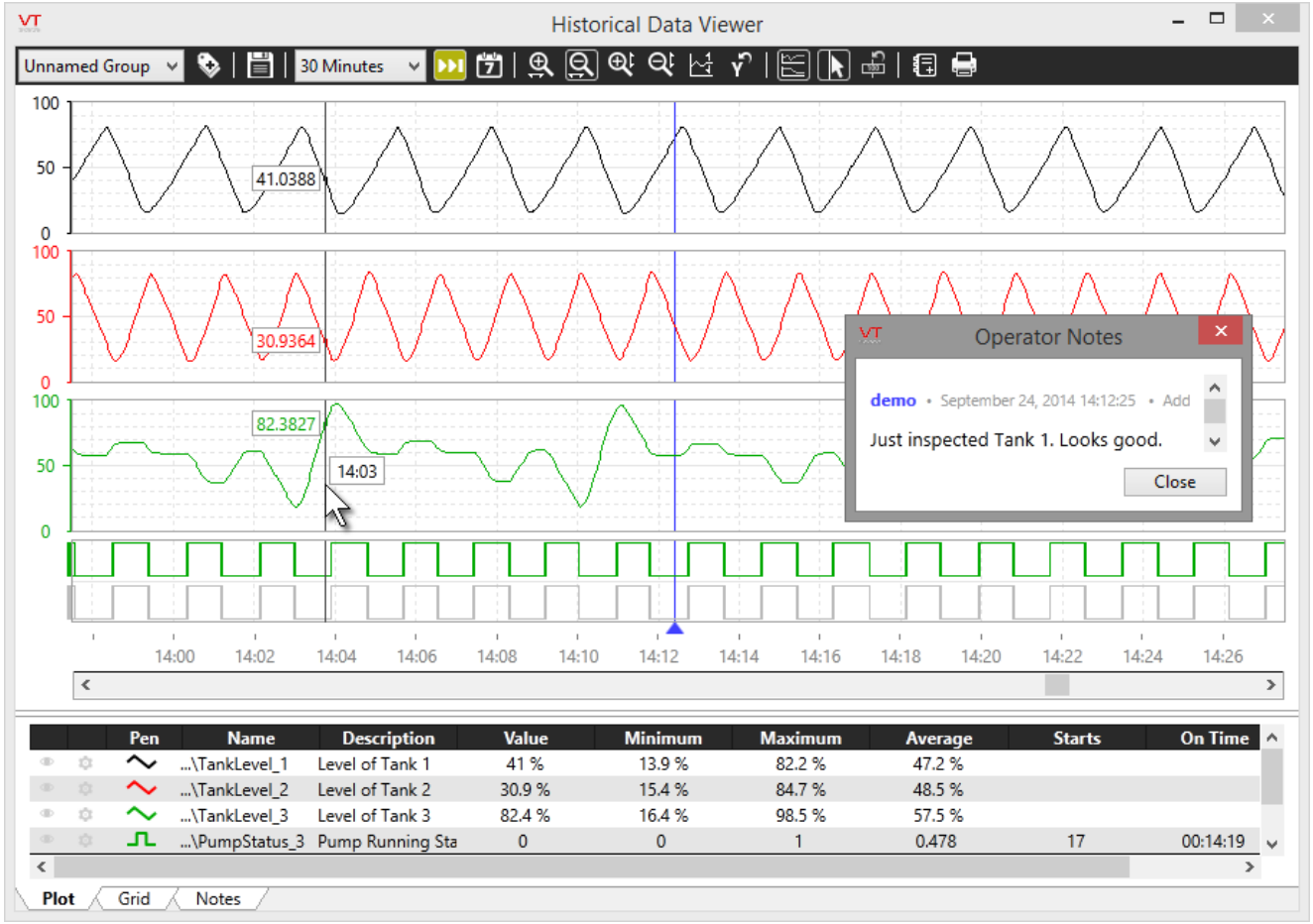
SCADA functionalities

Data history

The SCADA system stores the data collected from the plant, into a type of **time-series database**

The stored data can then be used to

- **display trends** of process data on charts
- **create reports**
- perform **data analysis**



SCADA functionalities

Alarms handling

A SCADA system allows **programming the conditions** in which an **alarm** should be activated

- It **notifies the operator** about the occurrence of the alarms (flashing lights, sirens, email,...)

Alarms can be

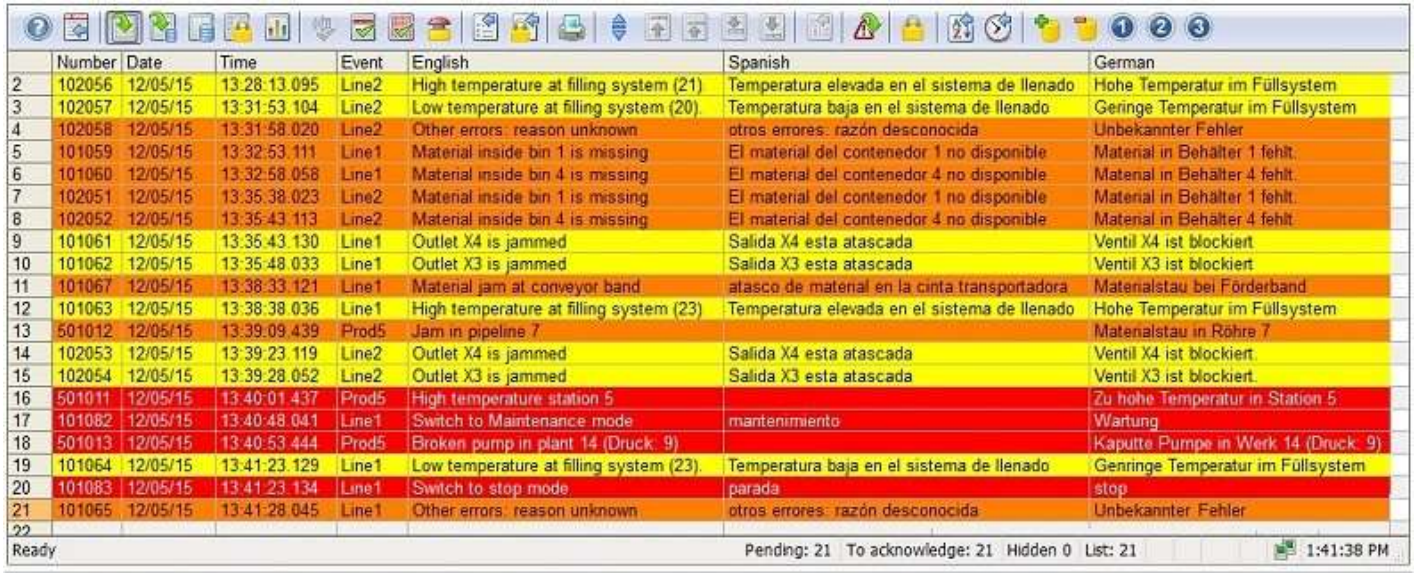
- **system defined** or **user defined** alarms
- **discrete** or **analog** alarms

SCADA functionalities

Alarms handling

When an alarm is raised, it is possible **to acknowledge** it

- Register the alarm
- Reset the alarm
- Silence alarm notification



	Number	Date	Time	Event	English	Spanish	German
2	102056	12/05/15	13:28:13.095	Line2	High temperature at filling system (21)	Temperatura elevada en el sistema de llenado	Hohe Temperatur im Füllsystem
3	102057	12/05/15	13:31:53.104	Line2	Low temperature at filling system (20)	Temperatura baja en el sistema de llenado	Geringe Temperatur im Füllsystem
4	102058	12/05/15	13:31:58.020	Line2	Other errors: reason unknown	otros errores: razón desconocida	Unbekannter Fehler
5	101059	12/05/15	13:32:53.111	Line1	Material inside bin 1 is missing	El material del contenedor 1 no disponible	Material in Behälter 1 fehlt.
6	101060	12/05/15	13:32:58.058	Line1	Material inside bin 4 is missing	El material del contenedor 4 no disponible	Material in Behälter 4 fehlt.
7	102051	12/05/15	13:35:38.023	Line2	Material inside bin 1 is missing	El material del contenedor 1 no disponible	Material in Behälter 1 fehlt.
8	102052	12/05/15	13:35:43.113	Line2	Material inside bin 4 is missing	El material del contenedor 4 no disponible	Material in Behälter 4 fehlt.
9	101061	12/05/15	13:35:43.130	Line1	Outlet X4 is jammed	Salida X4 esta atascada	Ventil X4 ist blockiert.
10	101062	12/05/15	13:35:48.033	Line1	Outlet X3 is jammed	Salida X3 esta atascada	Ventil X3 ist blockiert.
11	101067	12/05/15	13:38:33.121	Line1	Material jam at conveyor band	atasco de material en la cinta transportadora	Materialstau bei Förderband
12	101063	12/05/15	13:38:38.036	Line1	High temperature at filling system (23)	Temperatura elevada en el sistema de llenado	Hohe Temperatur im Füllsystem
13	501012	12/05/15	13:39:09.439	Prod5	Jam in pipeline 7		Materialstau in Röhre 7
14	102053	12/05/15	13:39:23.119	Line2	Outlet X4 is jammed	Salida X4 esta atascada	Ventil X4 ist blockiert.
15	102054	12/05/15	13:39:28.052	Line2	Outlet X3 is jammed	Salida X3 esta atascada	Ventil X3 ist blockiert.
16	501011	12/05/15	13:40:01.437	Prod5	High temperature station 5		Zu hohe Temperatur in Station 5
17	101082	12/05/15	13:40:48.041	Line1	Switch to Maintenance mode	mantenimiento	Wartung
18	501013	12/05/15	13:40:53.444	Prod5	Broken pump in plant 14 (Druck: 9)		Kaputte Pumpe in Werk 14 (Druck: 9)
19	101064	12/05/15	13:41:23.129	Line1	Low temperature at filling system (23)	Temperatura baja en el sistema de llenado	Genringe Temperatur im Füllsystem
20	101083	12/05/15	13:41:23.134	Line1	Switch to stop mode	parada	stop
21	101065	12/05/15	13:41:28.045	Line1	Other errors: reason unknown	otros errores: razón desconocida	Unbekannter Fehler
22							

Ready Pending: 21 To acknowledge: 21 Hidden 0 List: 21 1:41:38 PM

SCADA functionalities

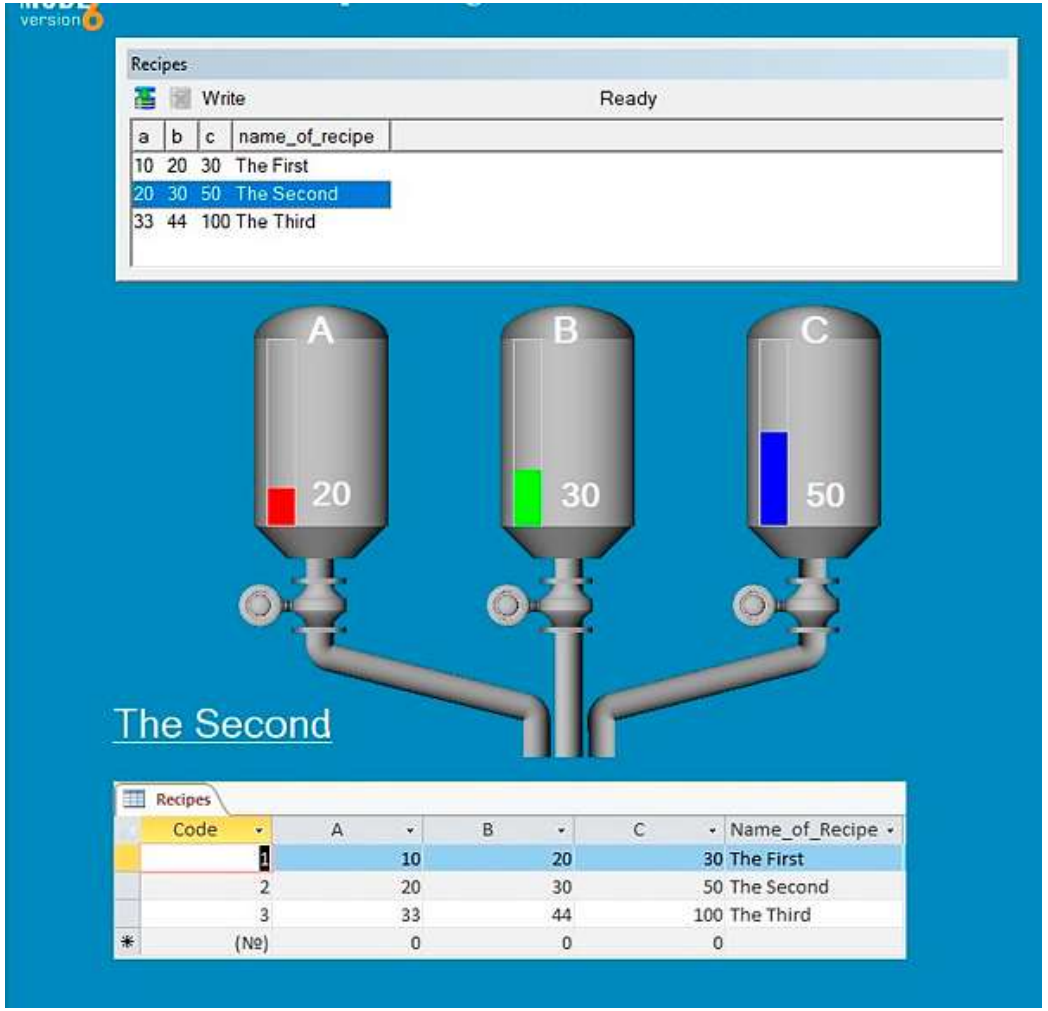
Recipes management

A SCADA system can execute a **predefined sequence of operations** called ‘recipe’

- Can be used in **batch production** systems.

The recipes can be executed:

- cyclically
- triggered by an event (e.g., an alarm)
- after an operator request



MASTER MEIM 2021-2022

Thank you