EPFL

PLC Programming

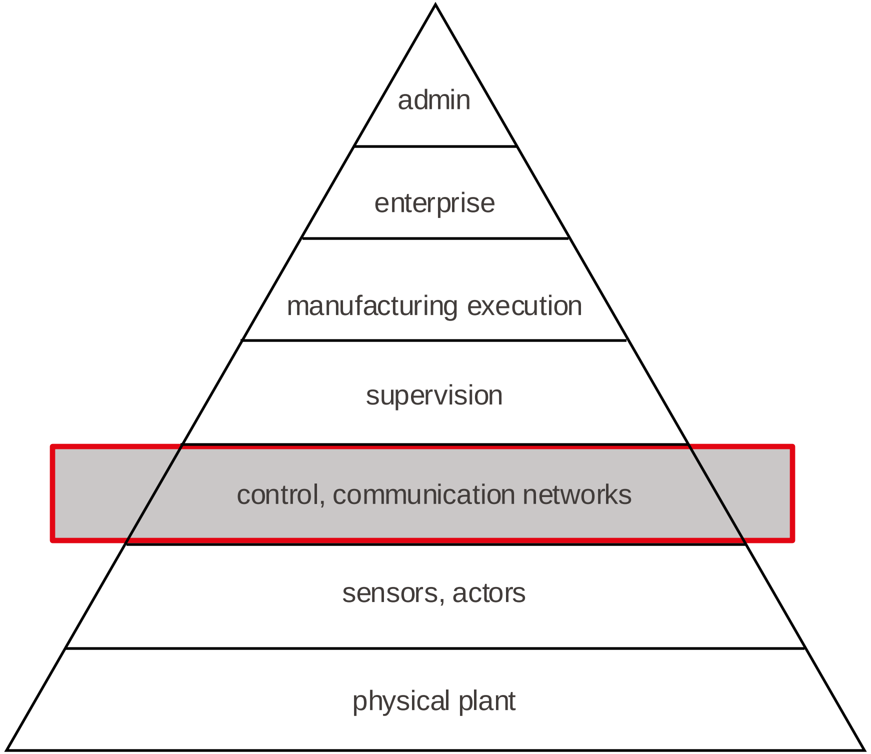
Dr. Philipp Sommer

Spring 2025

Join at
slido.com
#776 576

[Picture: Pierre75000, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=80946678]

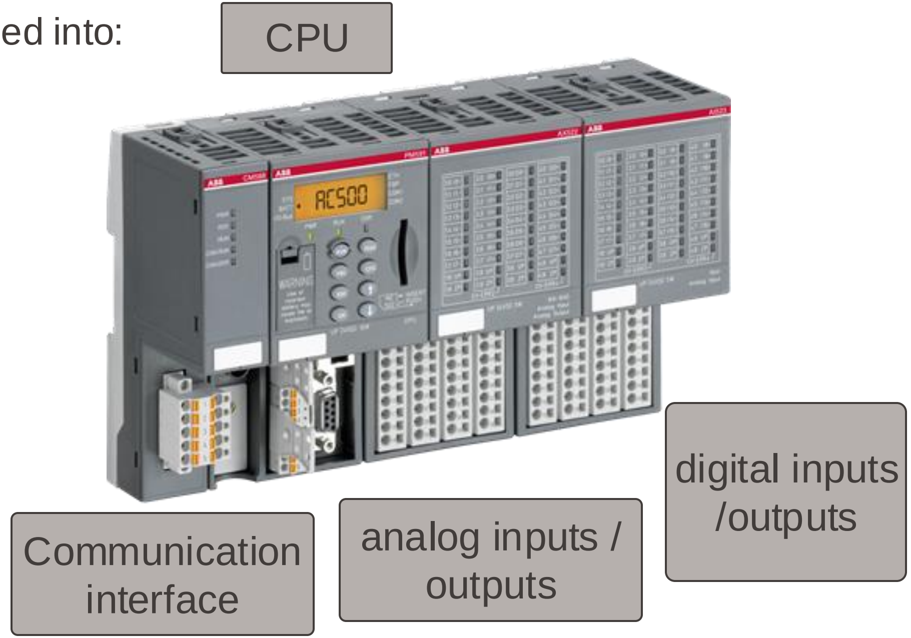
Programmable Logic Controllers (PLCs) in the Automation Pyramid



Programmable Logic Controller (PLC)

Main components in a PLC can be categorized into:

- Processor (CPU, etc.)
- Input (digital, analog, etc.)
- Output (digital, analog, etc.)
- Communication interface



CPU

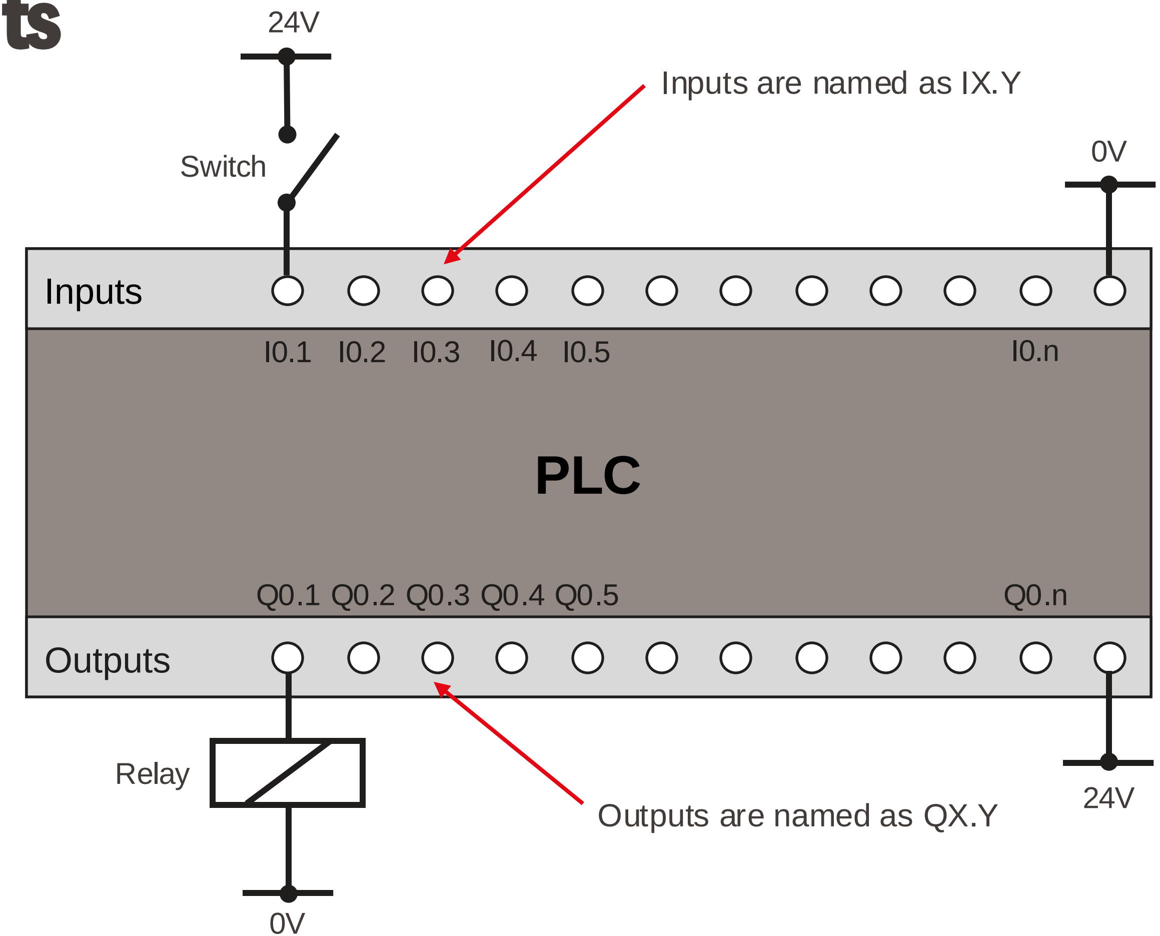
Communication interface

analog inputs / outputs

digital inputs /outputs

https://new.abb.com/plc/programmable-logic-controllers-plcs/ac500

PLC Inputs and Outputs

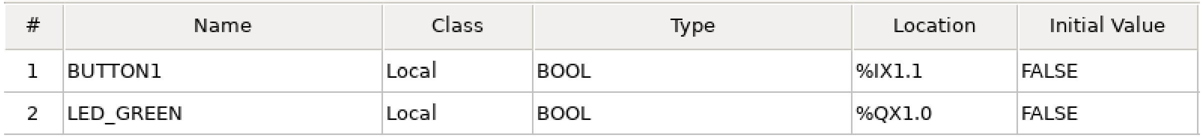


Inputs are named as IX.Y

Outputs are named as QX.Y

Connecting to Inputs / Outputs

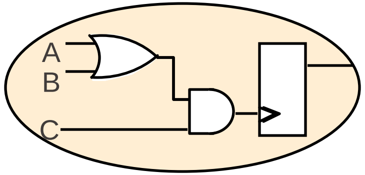
- All program variables must be declared with name, type and initial value.
- A variable may be connected to an input or an output, giving it an I/O address.
- Several properties can be set: initial value, fall-back value, store at power fail,…



#	Name	Class	Type	Location	Initial Value
1	BUTTON1	Local	BOOL	%IX1.1	FALSE
2	LED_GREEN	Local	BOOL	%QX1.0	FALSE

Matching the analog and binary world

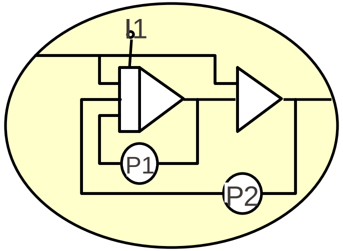
- Binary World
  - Relay control, pneumatic sequencer



combinatorial　sequential

discrete processes

- Analog World
  - Pneumatic and electromechanical controllers



regulation, controllers

continuous processes

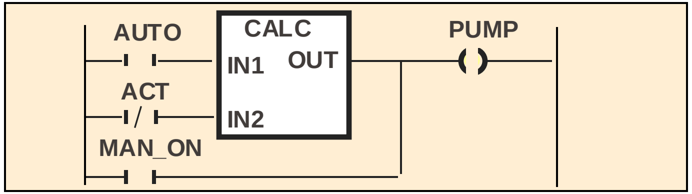
PLC

How to program PLCs?

- Originally PLCs are intended to be used by non-software engineers
  - Ladder Diagram (LD): Graphical programming language similar to electric circuit diagrams
- IEC (International Electrotechnical Commission) 61131-3 standard defines 5 programming languages adopted by most PLC manufacturers:
  - Textual languages:
    - Instruction List (IL)
    - Structured Text (ST)
  - Graphical languages:
    - Ladder Diagram (LAD)
    - Function Block Diagram (FBD)
    - Sequential Function Chart (SFC)

IEC 61131-3 programming languages

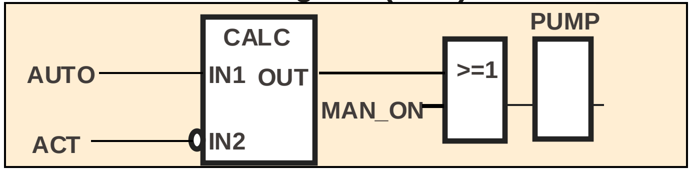
Ladder Diagram (LD)



Instruction List (IL)

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

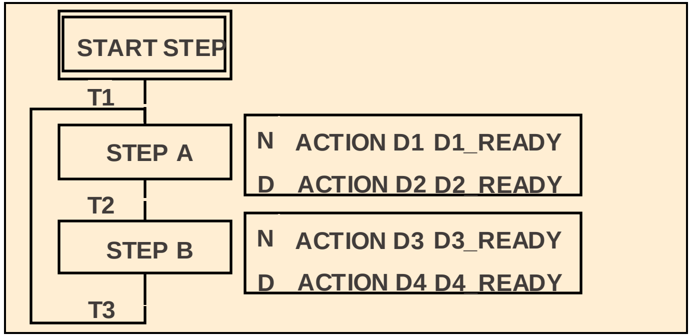
Function Block Diagram (FBD)



Structured Text (ST)

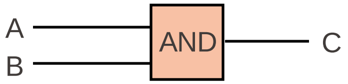
```
VAR CONSTANT X : REAL := 53.8;
Z : REAL; END_VAR
VAR aFB, bFB :  FB_type; END_VAR

bFB(A:=1, B:=‘OK’);
Z := X - INT_TO_REAL (bFB.OUT1);
IF Z>57.0 THEN aFB(A:=0, B:=“ERR”);
END_IF
```

Sequential Flow Chart (SFC)

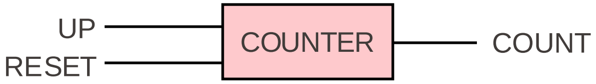


Types of Program Organisation Units (POUs)

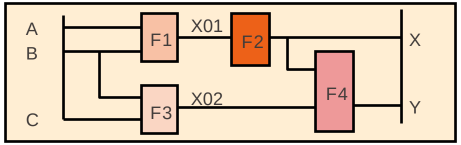
- Functions:
  - are part of the base library
  - have no memory
  - Examples: AND gate, adder, multiplier, selector, ...



- Function Blocks (FB):
  - are part of the base library
  - may have a memory ("static" data)
  - may access global variables (side-effects!)
  - Examples: counter, filter, integrator, ...



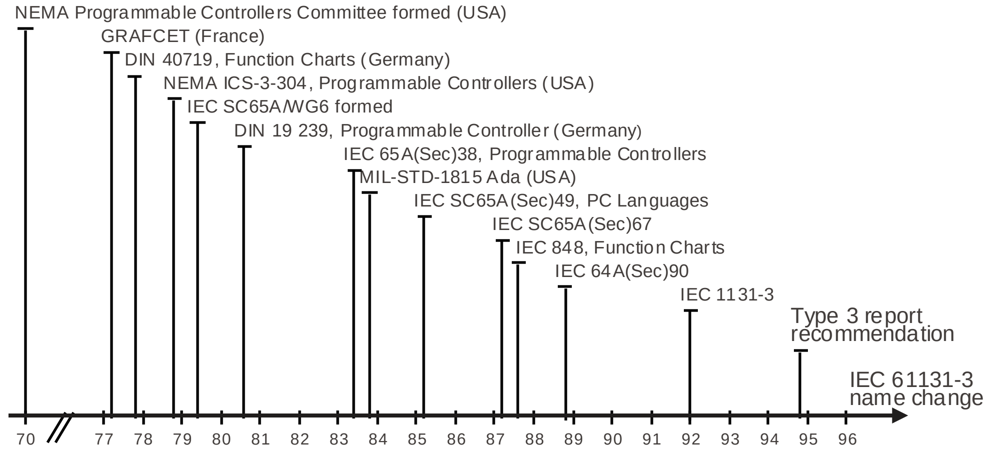
- Programs (compound blocks):
  - user-defined or application-specific blocks
  - may have a memory
  - may be configurable
  - Examples: PID controller, overcurrent protection, motor sequence (a library of compound blocks may be found in IEC 61804-1)



The long way to the IEC 61131-3 standard

- PLC industry needs agreement on:
  - Data types (operations may only be executed on appropriate types)
  - Programming languages
  - Software structure (program organization units for modularity, encapsulation)
  - Execution

it took 20 years to make that standard…



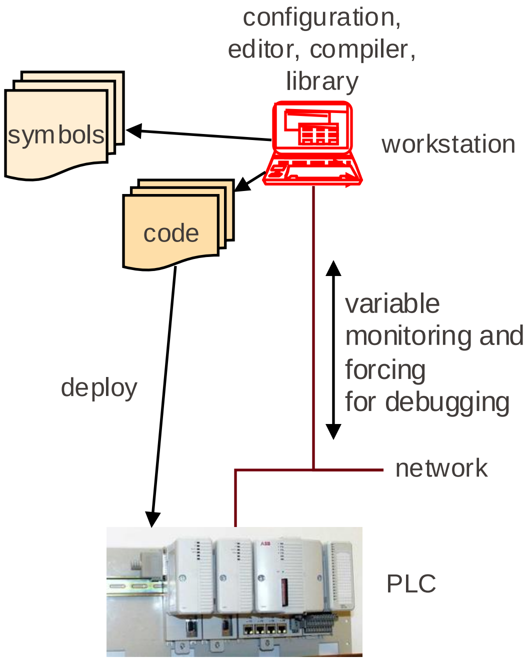
Source: Dr. J. Christensen

Why using IEC 61131-3 for PLCs?

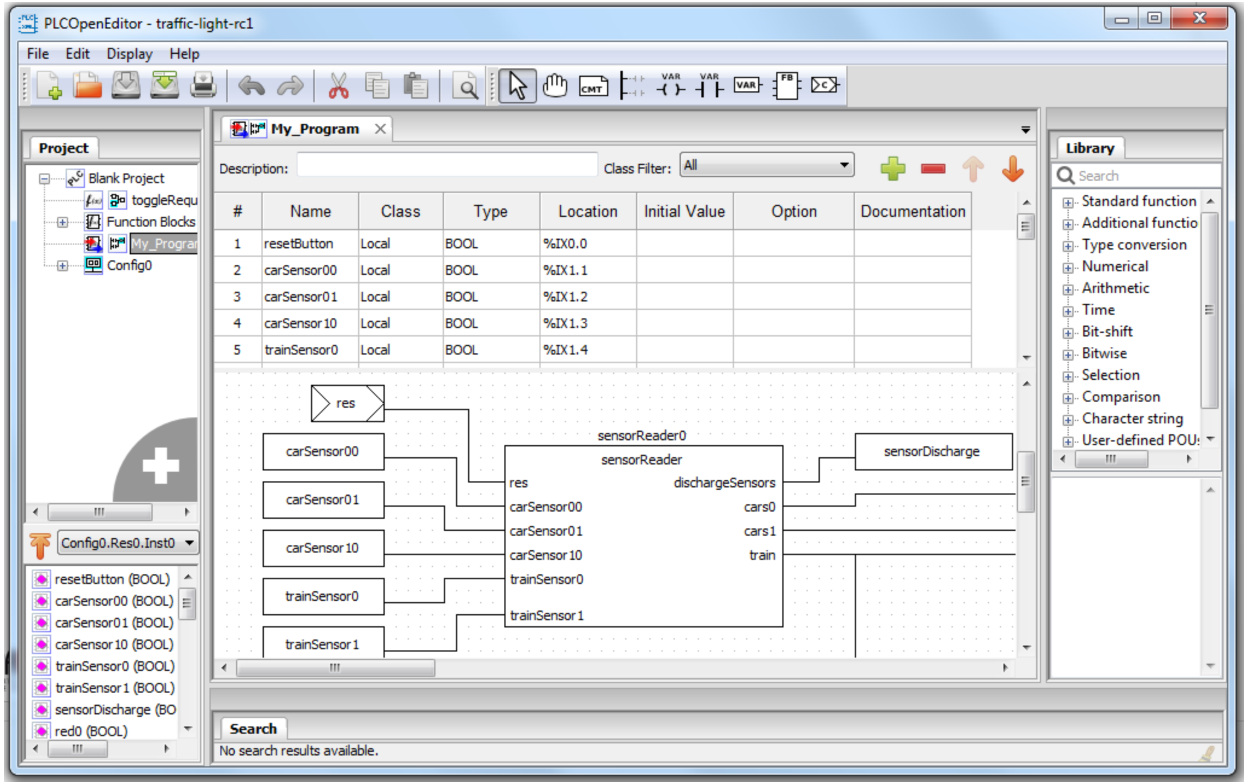
- IEC 61131-3 are the most important automation languages in industry. Most PLCs on the market support it.
- Improved interoperability between programming languages (standardized data types)
- Modularization, component reuse
- More information:
  - http://www.plcopen.org/pages/tc1_standards/downloads/plcopen_iec61131-3_feb2014.pptx
  - http://www.plcopen.org/pages/pc2_training/downloads/index.htm

Programming environment capabilities

- A PLC programming environment allows the programmer to:
  - program using one of the IEC 61131 languages
  - define the variables (name and type)
  - bind the variables to the input/output (binary, analog)
  - run simulations
  - download programs and firmware to the PLC
  - upload from the PLC (if provided, rare)
  - monitor the PLC
- Examples:
  - ABB ControlBuilder, Siemens Step 7, CoDeSys, OpenPLC


configuration, editor, compiler, library
symbols
workstation
code
deploy
variable monitoring and forcing for debugging
network
PLC

IDE Example: OpenPLC Editor



Description:

Class Filter: All

#	Name	Class	Type	Location	Initial Value	Option	Documentation
1	resetButton	Local	BOOL	%IX0.0			
2	carSensor00	Local	BOOL	%IX1.1			
3	carSensor01	Local	BOOL	%IX1.2			
4	carSensor10	Local	BOOL	%IX1.3			
5	trainSensor0	Local	BOOL	%IX1.4			

Search

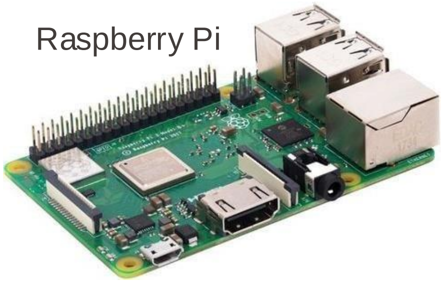
No search results available.

http://www.openplcproject.com

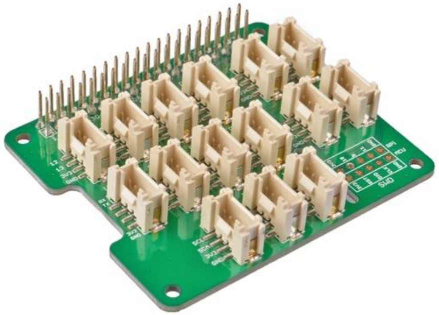
OpenPLC Runtime for Raspberry Pi

- Open-Source PLC Runtime Environment

Raspberry Pi



+



Grove Base Hat

+

LED (red, green, blue)

Push Button

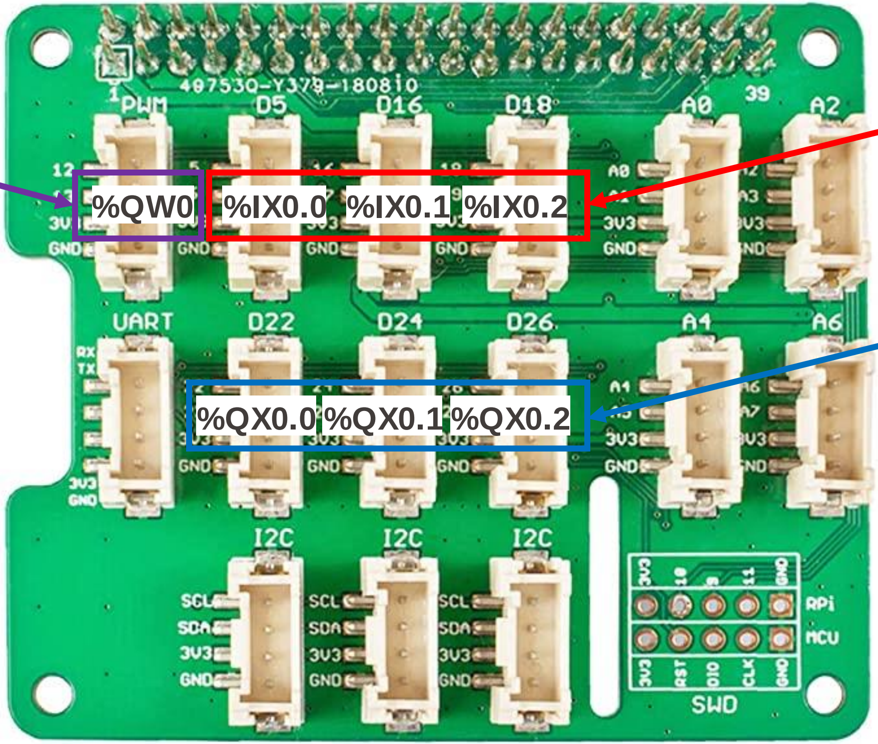
Switch

openPLC

Runtime

Raspberry Pi: Inputs and Outputs

Grove Base Hat for Raspberry Pi



1x Analog Output

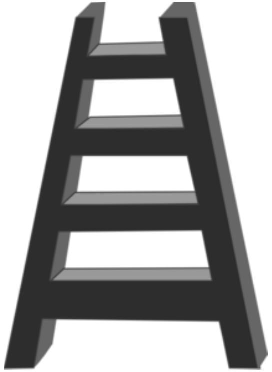
3x Digital Input

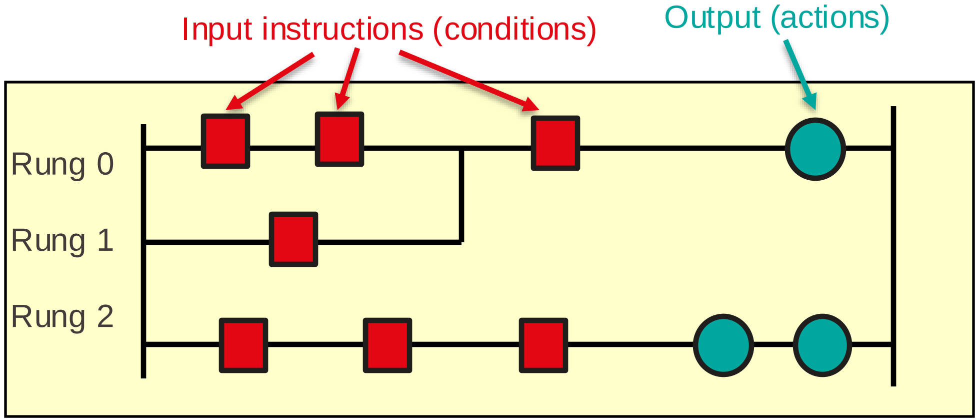
3x Digital Output

1. Ladder Diagrams (LD)

Ladder Diagrams (LD)

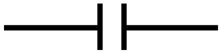
- Ladder Diagrams is the oldest programming language for PLC
  - based on relay intuition of electricians
  - widely used
  - not recommended for large new projects

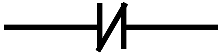


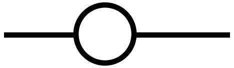


Ladder Diagrams (LD)

- The contact plan or "ladder diagram" language allows an easy transition from the traditional relay logic diagrams to the programming of binary functions:
  - well suited to express combinational logic
  - not suited for process control programming (there are no analog elements)
- The main Ladder Diagrams symbols represent the elements:


make contact


break contact


relay coil

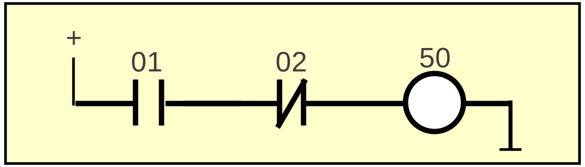
Ladder Diagrams: Series and Parallel Contacts

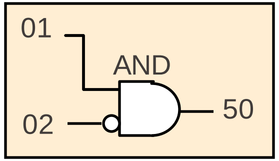
Binary combinations are expressed by series and parallel relay contact:

Ladder Diagrams representation

"logic" equivalent

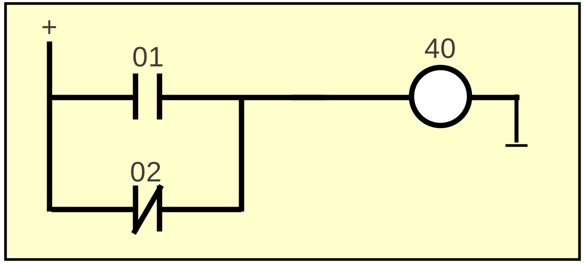
Series

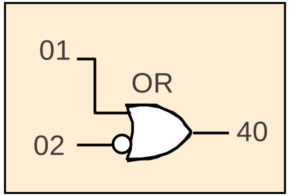




Coil 50 is active (current flows) when 01 is active AND 02 is not.

Parallel

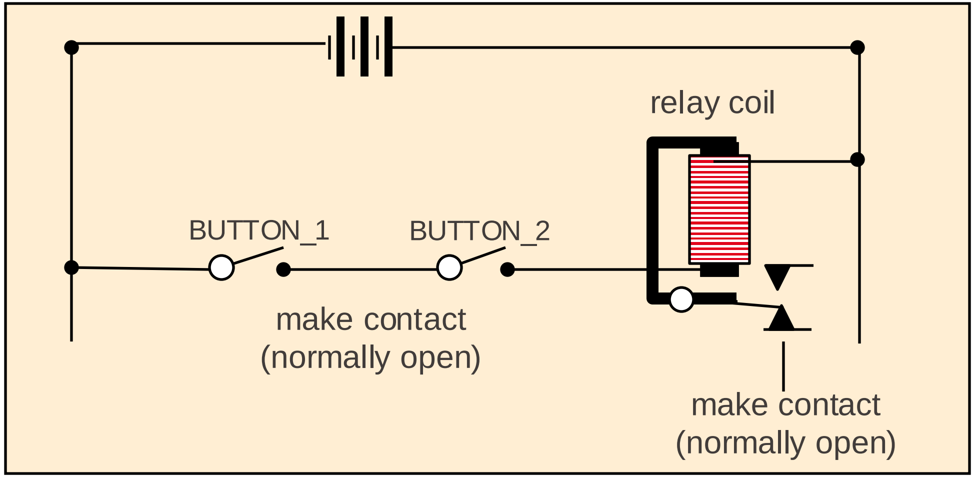




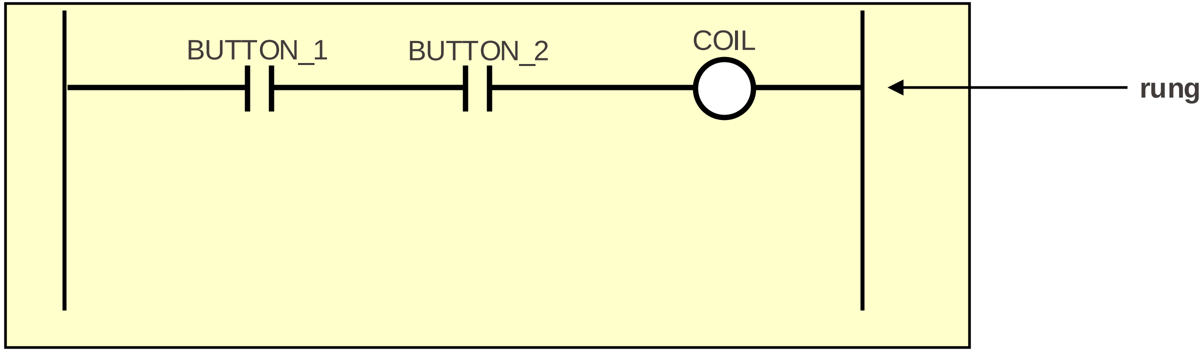
Coil 40 is active (current flows) when 01 is active OR 02 is not.

Ladder Diagrams: Example

origin:
Electrical circuit



corresponding
ladder diagram

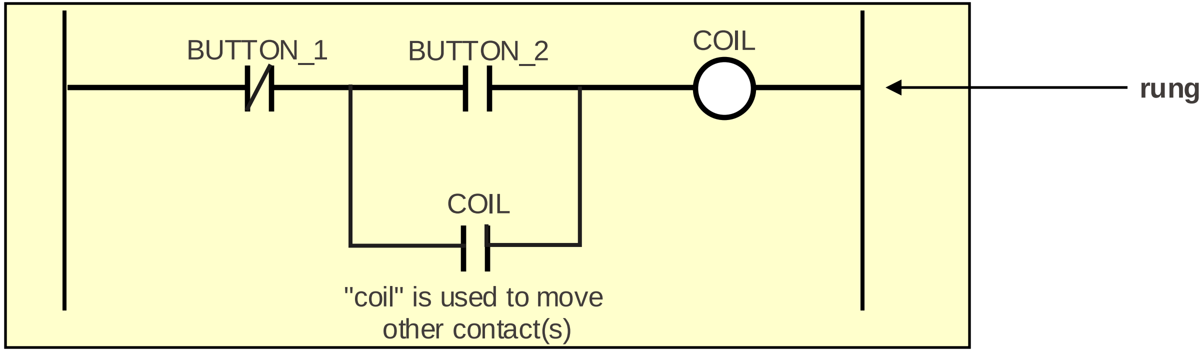


Exercise: Ladder Diagram

Join at
slido.com
#776 576

- What is the behavior of the following ladder diagram?
  - What happens when Button 1 is pressed?
  - What happens when Button 2 is pressed?

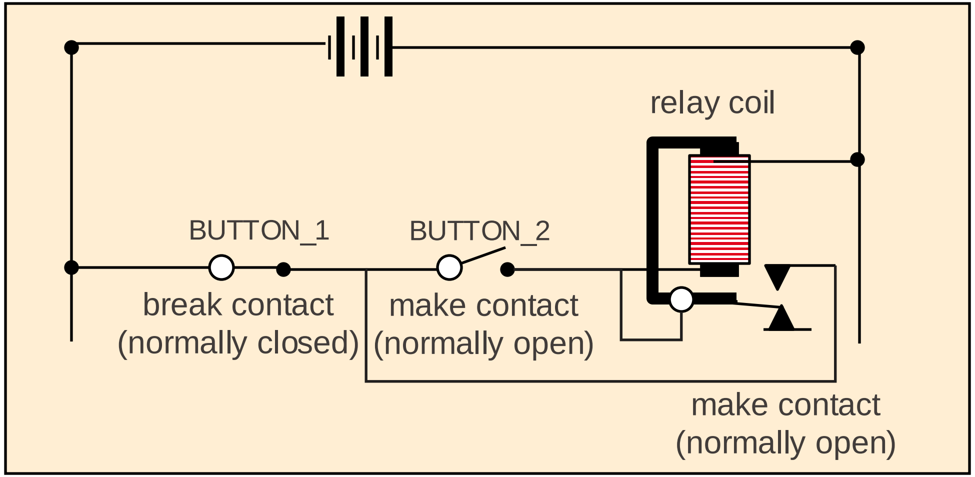
corresponding
ladder diagram



"coil" is used to move
other contact(s)

Solution: Ladder Diagram for Start/Stop

origin:
Electrical circuit



relay coil

BUTTON_1

BUTTON_2

break contact
(normally closed)

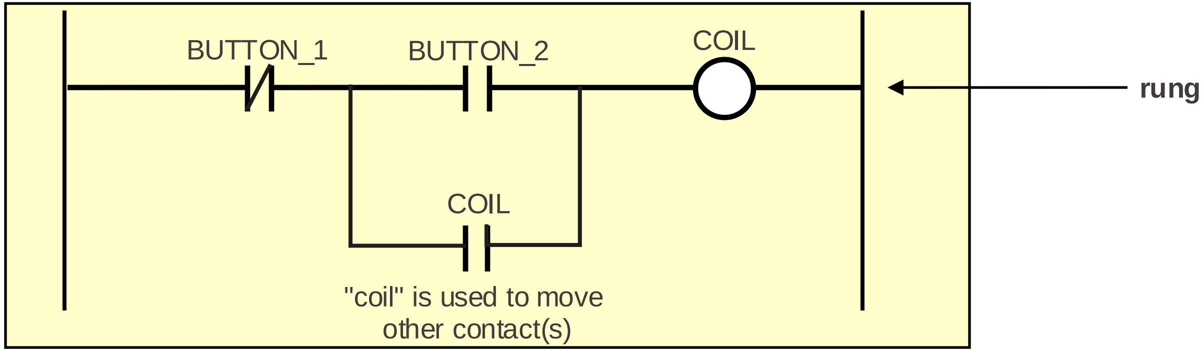
make contact
(normally open)

make contact
(normally open)

BUTTON_1 = STOP

BUTTON_2 = START

corresponding
ladder diagram



BUTTON_1

BUTTON_2

COIL

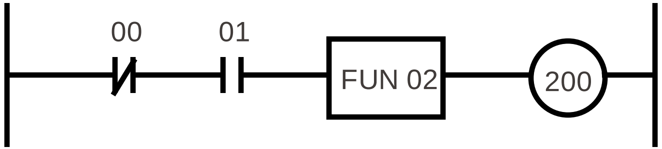
COIL

"coil" is used to move
other contact(s)

rung

Ladder Diagrams: Advanced Functions

- Ladder Diagrams stems from time of relay technology.
  - As PLCs replaced relays, not everything could be expressed in relay terms.
- Language was extended to express functions.
  - Intuition of contacts and coil gets lost.



literal expression:

200:= !00 & 01 FUN 02

- More or less hidden control of the flow destroys the freedom of side effects and makes programs difficult to read.

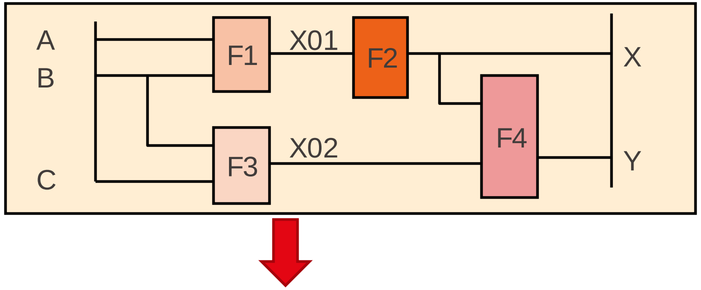
Ladder Diagrams: Shortcomings

- Ladder Diagrams don’t provide:
  - sub-programs (blocks)
  - data encapsulation
  - structured data types
- Not suited to make reusable modules.
- IEC 61131 does not prescribe the minimum requirements for a compiler / interpreter such as number of rungs per page nor does it specify the minimum subset to be implemented.
- Therefore, it should not be used for large programs made by groups of people
  - It is very limited when considering analog values
  - → used mostly in manufacturing, not in process control

2. Functional Block Diagram (FBD)

Execution of function blocks

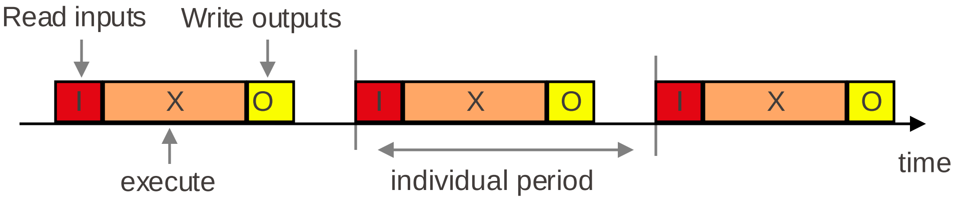
- The function blocks are translated to machine language (intermediate code, IL), that is either interpreted or compiled to assembly language
- Blocks are executed in sequence, normally from upper left to lower right
- The sequence is repeated every *t* ms.



function	F1
input1	A
input2	B
output	X01
	F2
	X01
	X
	F3
	B
	C
	X02
	F4
	X
	X02
	Y

Input-output of function blocks

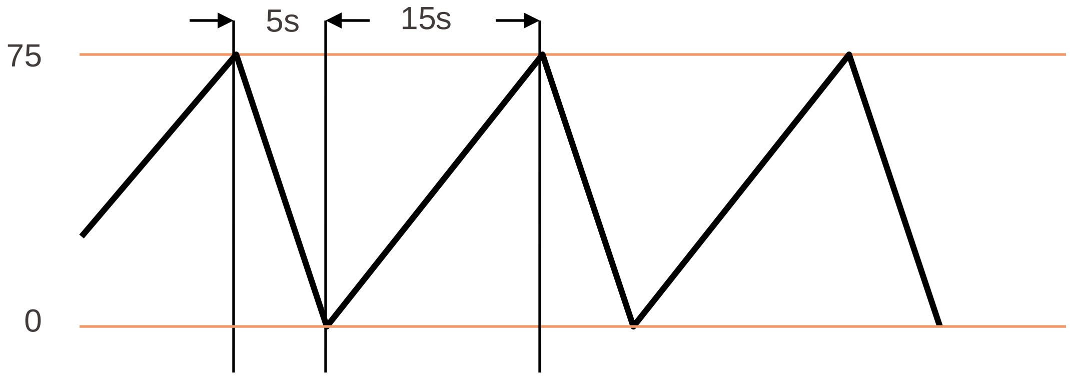
- Executed cyclically:
  - all inputs are read from memory or plant (possibly cached)
  - segment is executed
  - results are written into memory or to plant (possibly to cache)



- Order of execution of the blocks does not matter
- For speed it can help to impose execution order on blocks
- Different segments may be assigned different periods

Exercise: Asymmetric Sawtooth Wave

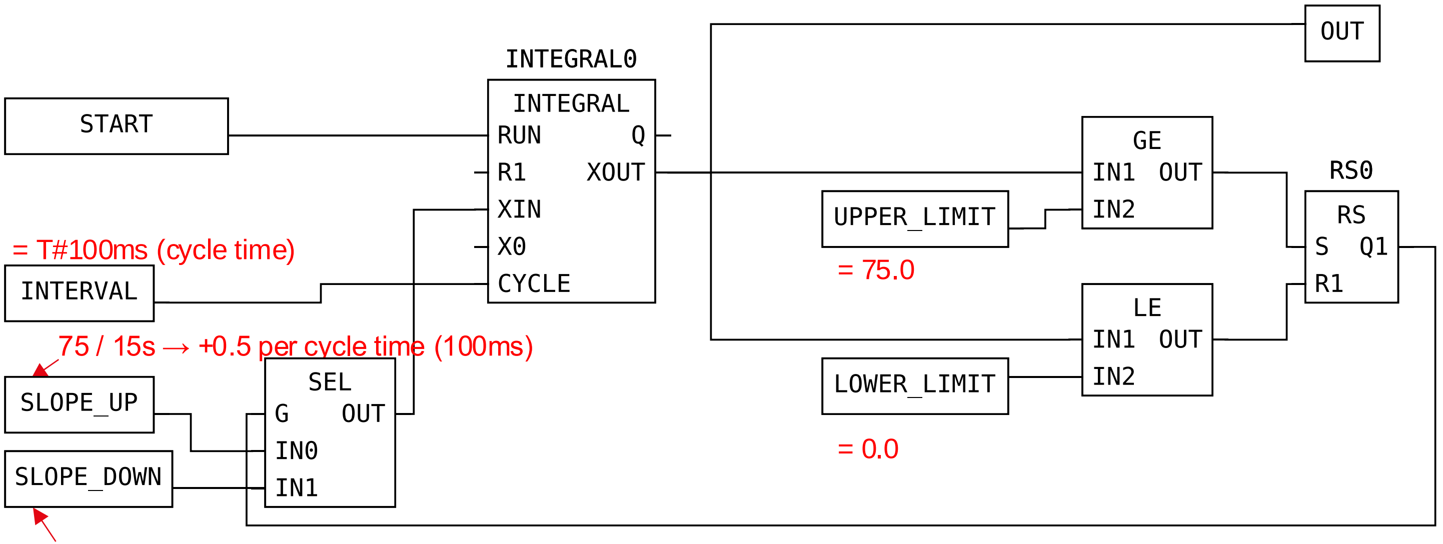
Build an asymmetric sawtooth wave generator for constant input with IEC 61131 function blocks.



Hints:

- Compute the slopes
- Use integrators, comparators, flip-flops and selectors

Solution: Asymmetric Sawtooth Wave with FBD



= T#100ms (cycle time)

75 / 15s → +0.5 per cycle time (100ms)

= 75.0

= 0.0

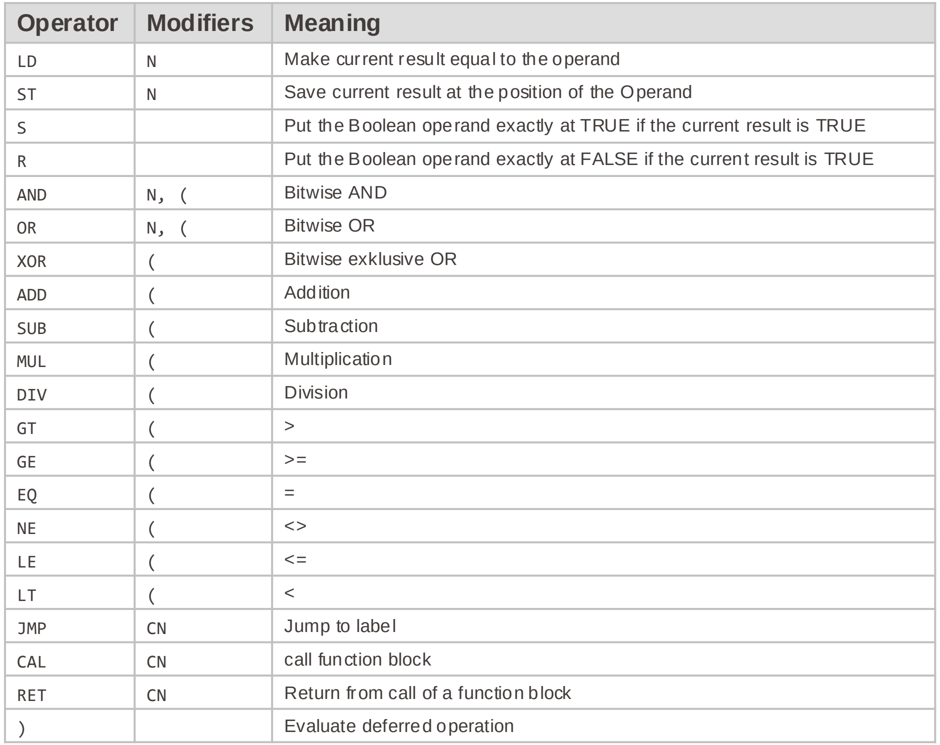
75 / 5s → -1.5 per cycle time (100ms)

Note: This is just one of many possible solutions.

3. Instruction List (IL)

Instruction Lists (1)

- Instruction lists is the machine language of PLC programming
- It has 21 instructions
- Three modifiers are defined:
  - "N" negates the result
  - "C" makes it conditional
  - "(" delays it.
- All operations relate to accumulator.



Operator	Modifiers	Meaning
`LD`	`N`	Make current result equal to the operand
`ST`	`N`	Save current result at the position of the Operand
`S`		Put the Boolean operand exactly at TRUE if the current result is TRUE
`R`		Put the Boolean operand exactly at FALSE if the current result is TRUE
`AND`	`N, (`	Bitwise AND
`OR`	`N, (`	Bitwise OR
`XOR`	`(`	Bitwise exklusive OR
`ADD`	`(`	Addition
`SUB`	`(`	Subtraction
`MUL`	`(`	Multiplication
`DIV`	`(`	Division
`GT`	`(`	>
`GE`	`(`	>=
`EQ`	`(`	=
`NE`	`(`	<>
`LE`	`(`	<=
`LT`	`(`	<
`JMP`	`CN`	Jump to label
`CAL`	`CN`	call function block
`RET`	`CN`	Return from call of a function block
`)`		Evaluate deferred operation

Source: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplccontrol/html/TcPlcCtrl_Languages%20IL.htm

Instruction Lists (2)

- Accumulator-based programming:
  1. First, values are loaded into the accumulator (LD instruction)
  2. Then, operations are executed with first parameter taken out of accumulator and second parameter of operand.
  3. Result put in the accumulator, from where it can be stored (ST instruction)
- Conditional executions or loops are supported by comparing operators like EQ, GT, LT, GE, LE, NE and jumps (JMP, JMPC, JMPCN, for the last two the accumulators value is checked on TRUE or FALSE)
- Syntax:
  - each instruction begins on a new line and contains an operator and, depending on the type of operation, one or more operands separated by commas
  - before an instruction there can be a label, followed by a colon (:), as target for jumps
  - use brackets to define order of execution
  - comments must be placed last

Instruction Lists: Example

```
LD TRUE         (*load TRUE into the accumulator*)
ANDN BOOL1      (*execute AND with the negated value of the BOOL1 variable*)
JMPC mark       (*if the result was TRUE, then jump to the label "mark"*)

LDN BOOL2       (*load the negated value of BOOL2 into the accumulator*)
ST RES          (*store the content of the accumulator in RES*)
JMP continue    (*jump to label “continue"*)

mark:
LD BOOL2        (*save the value of *)
ST RES          (*BOOL2 in RES*)

continue:
…
```

Instruction Lists: Shortcomings

- With Instructions Lists (IL) once can write the most efficient code, but only for specialists.
- In general, IL should not be used because of:
  - provides no code structuring
  - is machine-dependent
  - weak tool support (IDE)

Instruction Lists: Exercise

Join at **slido.com** **#776 576**

```
LD temp1                    (* load temp1 *)
GT temp2                    (* test if temp1 > temp2 *)
JMPCN Greater               (* jump if NOT true to Greater *)
LD speed1                   (* load speed1 *)
ADD 200                     (* add constant 200 *)
JMP End                     (* jump unconditionally to End *)

Greater: LD speed2          (* load speed2 *)
End: ST speed3              (* store result in speed3 *)
```

Question: What is the resulting value of the variable **speed3** for the following input?

```
temp1 = 10
temp2 = 5
speed1 = 50
speed2 = 100
```

4. Structured Text (ST)

Structured Text

- Structured Text is an imperative language similar to Pascal (`If`, `While`, etc..)
  - Iteration loops: `REPEAT..UNTIL, WHILE..DO`
  - Conditional execution: `IF..THEN..ELSE, CASE`
- The variables defined in ST can be used in other languages
- ST is used for complex data manipulation and to write blocks
- Caution: writing programs in structured text can breach real-time rules!

Structured Text Examples

```
IF tank.temp > 200 THEN
       pump.fast :=1;
       pump.slow :=0;
       pump.off :=0;
ELSIF tank.temp > 100
THEN
       pump.fast :=0;
       pump.slow :=1;
       pump.off :=0;
ELSE
       pump.fast :=0;
       pump.slow :=0;
       pump.off :=1;
END_IF;
```

```
IF( Switch_0 AND  Switch_1 ) THEN
       Start_Motor := 1;
       Start_Count := Start_Count + 1;
END_IF;
```

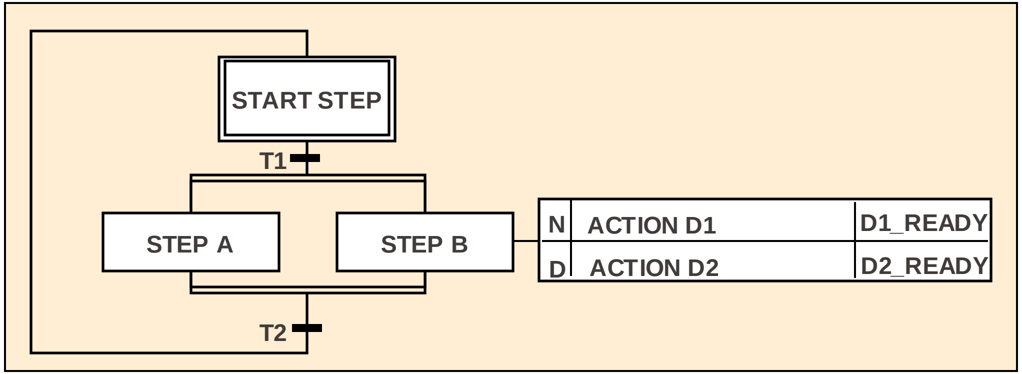
```
IF( BUTTON_1 ) THEN
       LED_RED := 1;
ELSE
       LED_RED := 0;
END_IF;
```

[http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm007_-en-p.pdf]

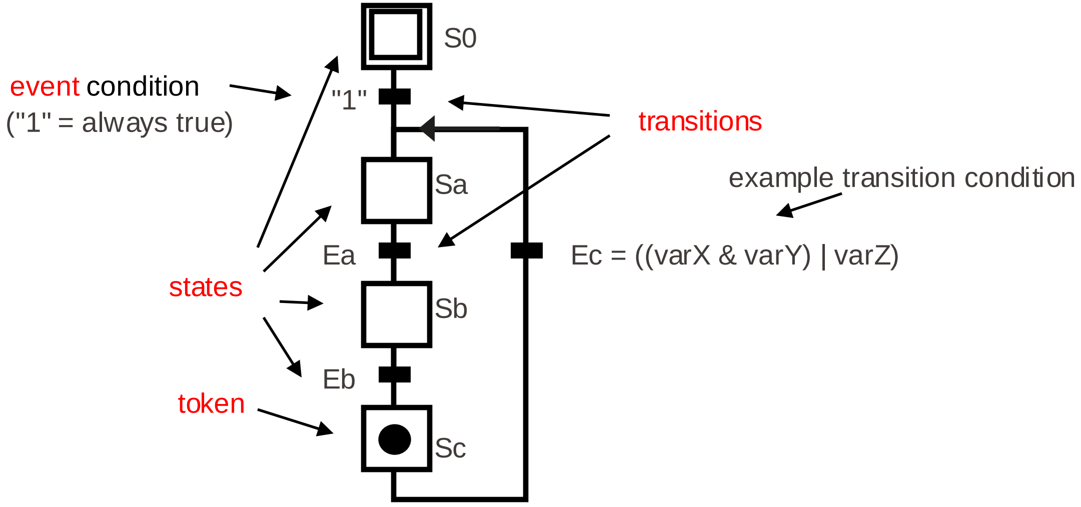
5. Sequential Function Chart (SFC)

SFC (Sequential Function Chart)

- Describes sequences of operations and interactions between parallel processes.
- Derived from Grafcet and SDL (Specification and Description Language, used for communication protocols), mathematical foundation lies in Petri Nets.



SFC: Elements

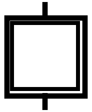


- Program consists of states connected by transitions.
- A state is activated by a token (the corresponding variable becomes TRUE).
- Token leaves state when transition condition (event) on state output is true.
- Only one transition takes place at a time.
- Execution period is configuration parameter (task to which program is attached)

Rule: there is always a transition between states, there is always a state between transitions.

SFC: Initial state

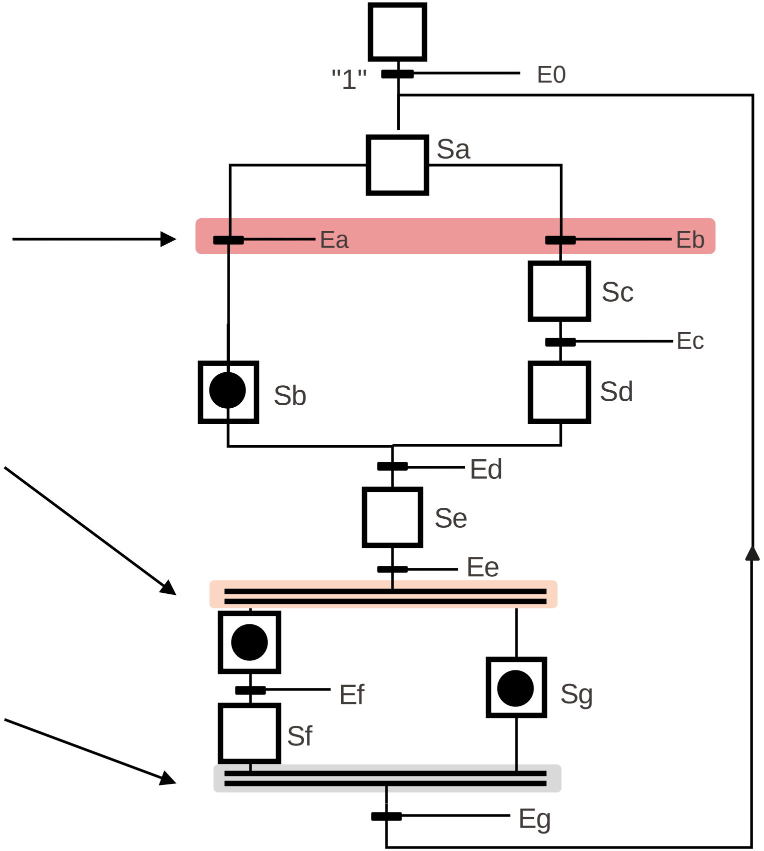
- State which come into existence with a token are called initial states.
- All initial states receive exactly one token, the other states receive none.
- Initialization takes place explicitly at start-up.
- In some systems, initialization may be triggered in a user program (initialization pin in a function block).



Initial State

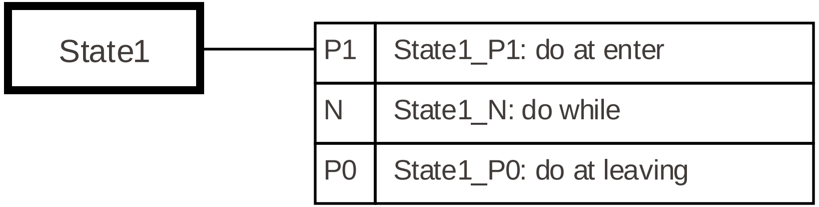
SFC: Switch and parallel execution

- Token Switch
  - Token crosses the first active transition (at random if both Ea and Eb are true)
  - Note: transitions are after the switch
- Token Forking
  - Token is replicated to all connected states when the transition Ee is true
  - Note: transition is before the fork
- Token Join
  - Single token is forwarded when all connected states have tokens and transition Eg is true.
  - Note: transition is after the join



SFC: P1, N and P0 actions

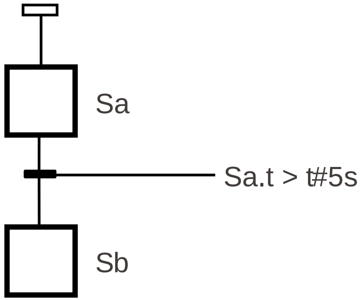
- P1 (pulse raise) action is executed once when the state is entered
- P0 (pulse fall) action is executed once when the state is left
- N (non-stored) action is executed continuously while the token is in the state
- P1 and P0 actions could be replaced by additional states.
- The actions are described by a code block written e.g. in Structured Text.



P1	State1_P1: do at enter
N	State1_N: do while
P0	State1_P0: do at leaving

Special action: the timer

- Rather than define a P0 action “reset timer….”, there is an implicit variable defined as <state name>.t that express the time spent in that state.



Flow Charts vs Function Blocks

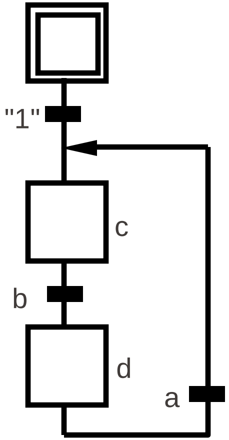
- Many PLC applications mix continuous and discrete control.
  - Function blocks: Continuous time control
  - Sequential flow charts: Discrete time control
- A PLC may execute alternatively function blocks and flow charts.
- Communication between these program parts is possible.

A flow chart taken as a whole can be considered a function block with binary inputs (transitions) and binary outputs (states).

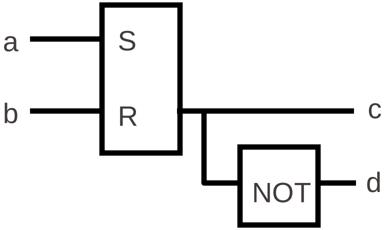
Flow Charts or Function Blocks?

A task can sometimes be written indifferently as function blocks or as flow chart. The application may decide which representation is more appropriate:

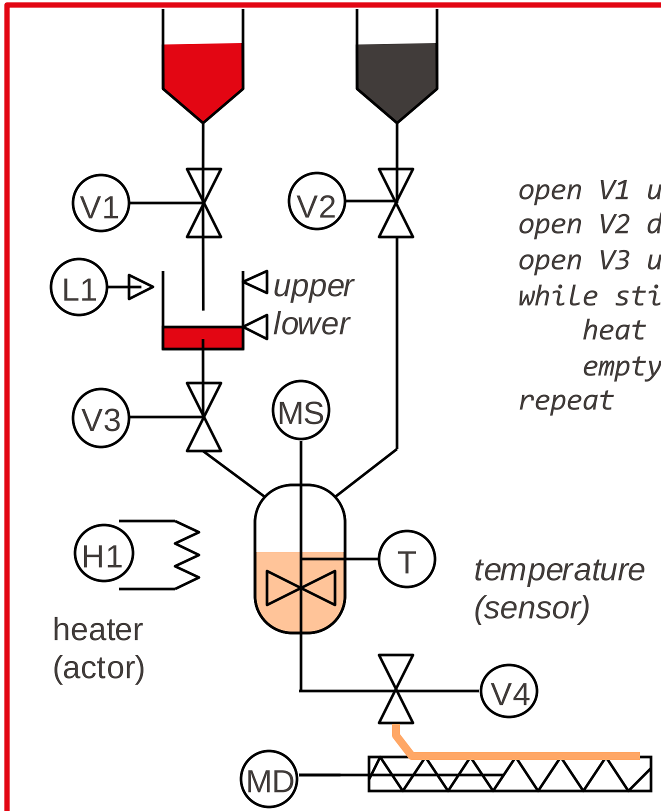
Flow Chart



Function Block



Exercise: Write the SFC for this task

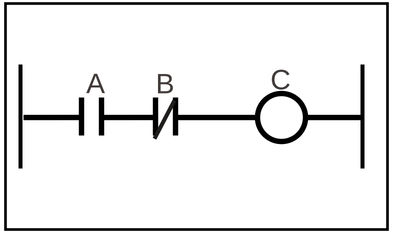


```
open V1 until tank’s L1 indicates upper level
open V2 during 25 seconds
open V3 until the tank’s L1 indicates it reached the lower level
while stirring:
    heat mixture during 50 minutes while stirring
    empty the reactor while the drying bed is moving
repeat
```

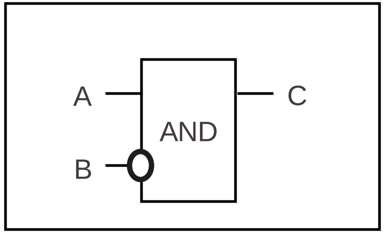
Summary

PLC Programming Languages (IEC 61131-3)

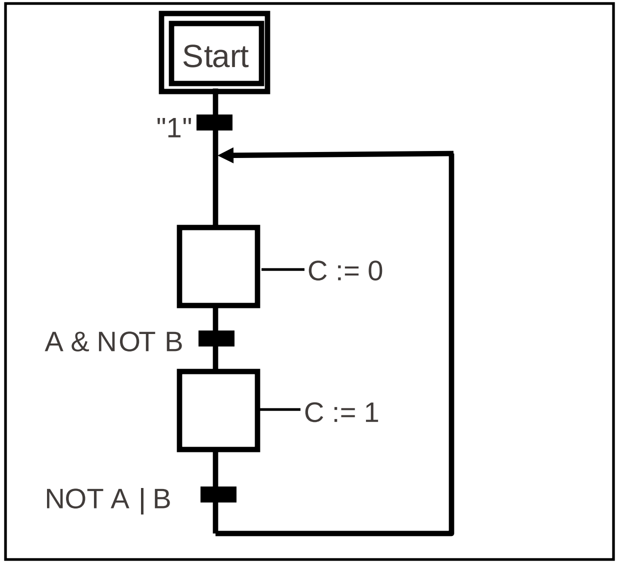
Ladder Diagram (LD)



Function Block Diagram (FBD)



Sequential Function Chart (SFC)



Instruction List (IL)

```
LD      A
ANDN    B
ST      C
```

Structured Text (ST)

```
C:= A AND NOT B
```

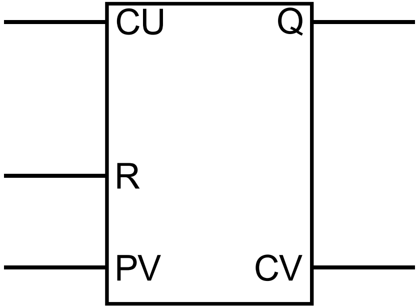
Limitations of IEC 61131

- No support to distribute execution of programs over several devices.
- No support for event-driven operation.
  Note: Blocks may be triggered by a Boolean variable
- No object orientation in structured text.

The next standard: IEC 61499

- IEC 61499 was first published in 2005
- Extends IEC 61131 with an event-driven model
- Application can be distributed over several PLCs

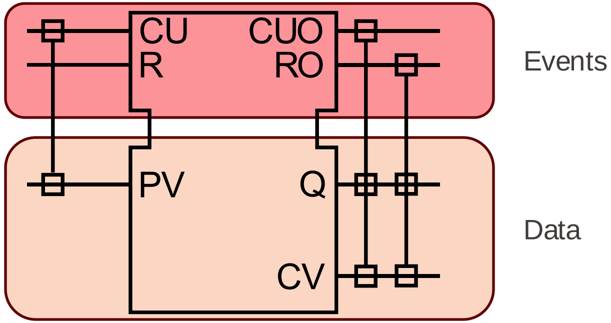
Counter Up



IEC 61131

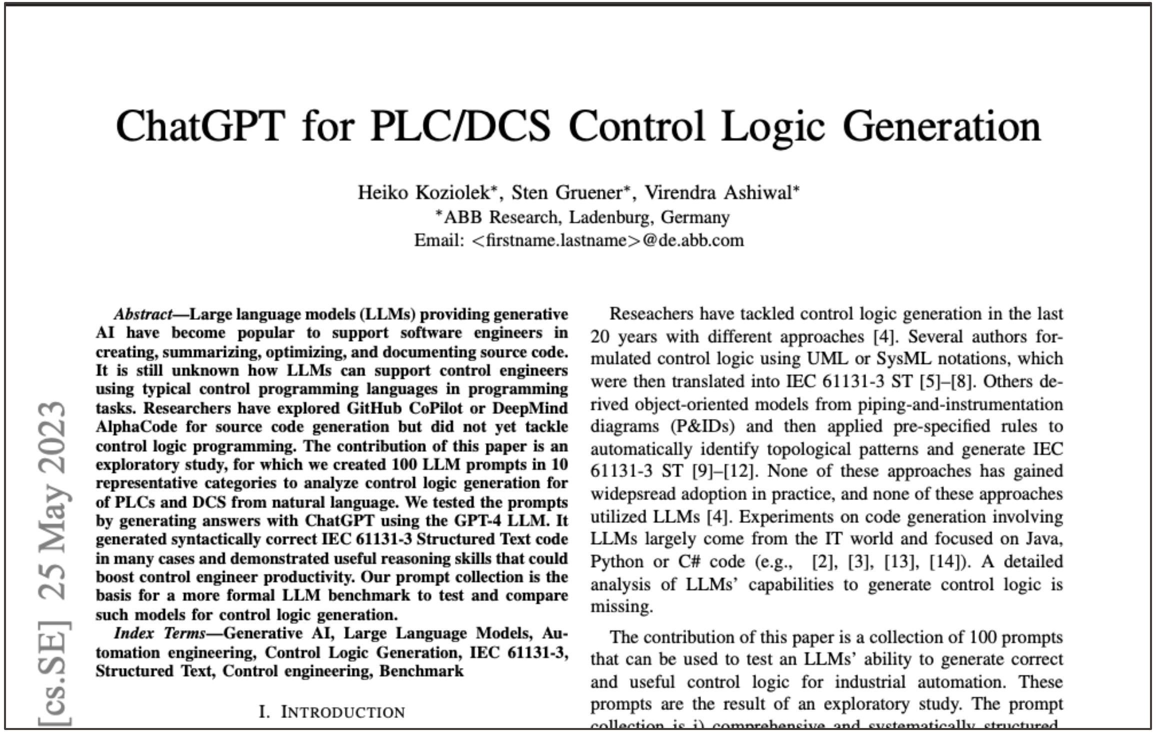
CU = Counter Up
R = Reset
PV = counter limit value
CV = current counter value
Q = TRUE when counter reaches limit

Counter Up



IEC 61499

Learning PLC Programming?



ChatGPT for PLC/DCS Control Logic Generation

Heiko Koziolek*, Sten Gruener*, Virendra Ashiwal*
*ABB Research, Ladenburg, Germany
Email: <firstname.lastname>@de.abb.com

Abstract—Large language models (LLMs) providing generative AI have become popular to support software engineers in creating, summarizing, optimizing, and documenting source code. It is still unknown how LLMs can support control engineers using typical control programming languages in programming tasks. Researchers have explored GitHub CoPilot or DeepMind AlphaCode for source code generation but did not yet tackle control logic programming. The contribution of this paper is an exploratory study, for which we created 100 LLM prompts in 10 representative categories to analyze control logic generation for of PLCs and DCS from natural language. We tested the prompts by generating answers with ChatGPT using the GPT-4 LLM. It generated syntactically correct IEC 61131-3 Structured Text code in many cases and demonstrated useful reasoning skills that could boost control engineer productivity. Our prompt collection is the basis for a more formal LLM benchmark to test and compare such models for control logic generation.

Index Terms—Generative AI, Large Language Models, Automation engineering, Control Logic Generation, IEC 61131-3, Structured Text, Control engineering, Benchmark

I. Introduction

Reseachers have tackled control logic generation in the last 20 years with different approaches [4]. Several authors formulated control logic using UML or SysML notations, which were then translated into IEC 61131-3 ST [5]–[8]. Others derived object-oriented models from piping-and-instrumentation diagrams (P&IDs) and then applied pre-specified rules to automatically identify topological patterns and generate IEC 61131-3 ST [9]–[12]. None of these approaches has gained widepsread adoption in practice, and none of these approaches utilized LLMs [4]. Experiments on code generation involving LLMs largely come from the IT world and focused on Java, Python or C# code (e.g., [2], [3], [13], [14]). A detailed analysis of LLMs' capabilities to generate control logic is missing.

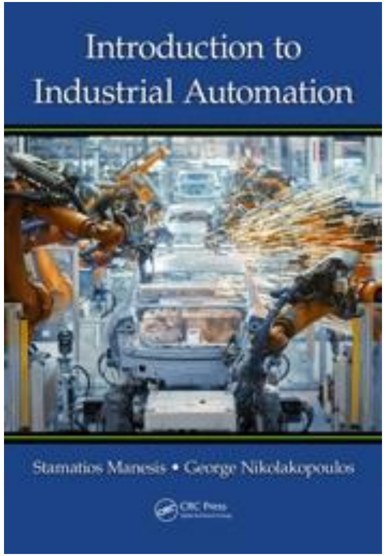
The contribution of this paper is a collection of 100 prompts that can be used to test an LLMs' ability to generate correct and useful control logic for industrial automation. These prompts are the result of an exploratory study. The prompt collection is i) comprehensive and systematically structured

Assessment

1. Which programming languages are defined in IEC 61131-3?
2. What are the advantages of using a PLC instead of using traditional relay logic?
3. How are inputs and outputs to the process treated in a function block language?
4. Why is Java/Python/Matlab not used to program PLCs?
5. Why is OpenPLC on the RaspberryPi not suited for industrial automation tasks?

Literature

- Stamatios Manesis, George Nikolakopoulos, “Introduction to Industrial Automation”, CRC Press


Introduction to Industrial Automation
Stamatios Manesis • George Nikolakopoulos
CRC Press