Automation notes Documentation

Release 0.9.0

Abed

2019-06-26 13:22:58

Automation notes

Siemens PLC (TIA Portal), CoDeSys, Beremiz, IEC 61131-3, ABB Robot

Warning: 2019-06-26 13:22:58

Note: **Scientia potentia est**

Warning: Work in progress

CHAPTER 1

Basics

Warning: Work in progress

1.1 Basics

In any automatic industrial line the following are present:

- Sensors
- Actuators
- Controller
- SCADA

Other components maybe are available, but the first 3 components are the heart of automation.

This system can be compared to human beings. Usually a human have one actuator, one controller and five senosrs (actually the human being sensors are more than 5). This system acquire information from outside via sensors (eye, nose, skin, ears, tongue,. . .). The brain, controller, elaborate these information and send commands to muscles (actuators).

1.1.1 Sensors and actuators

Industrial Sensors are those devices that acquire information from the field. Typically the signals are digital (e.g. switch, proximity sensor) or analog (e.g. height sensors, pressure gauge). Also a camera (vision system) can be classified as a sensor.

Actuators are mainly driven by electric, pneumatic and oleo-dynamic power. These actuators are mainly motors and valves.

1.1.2 Programmable logic controller: PLC

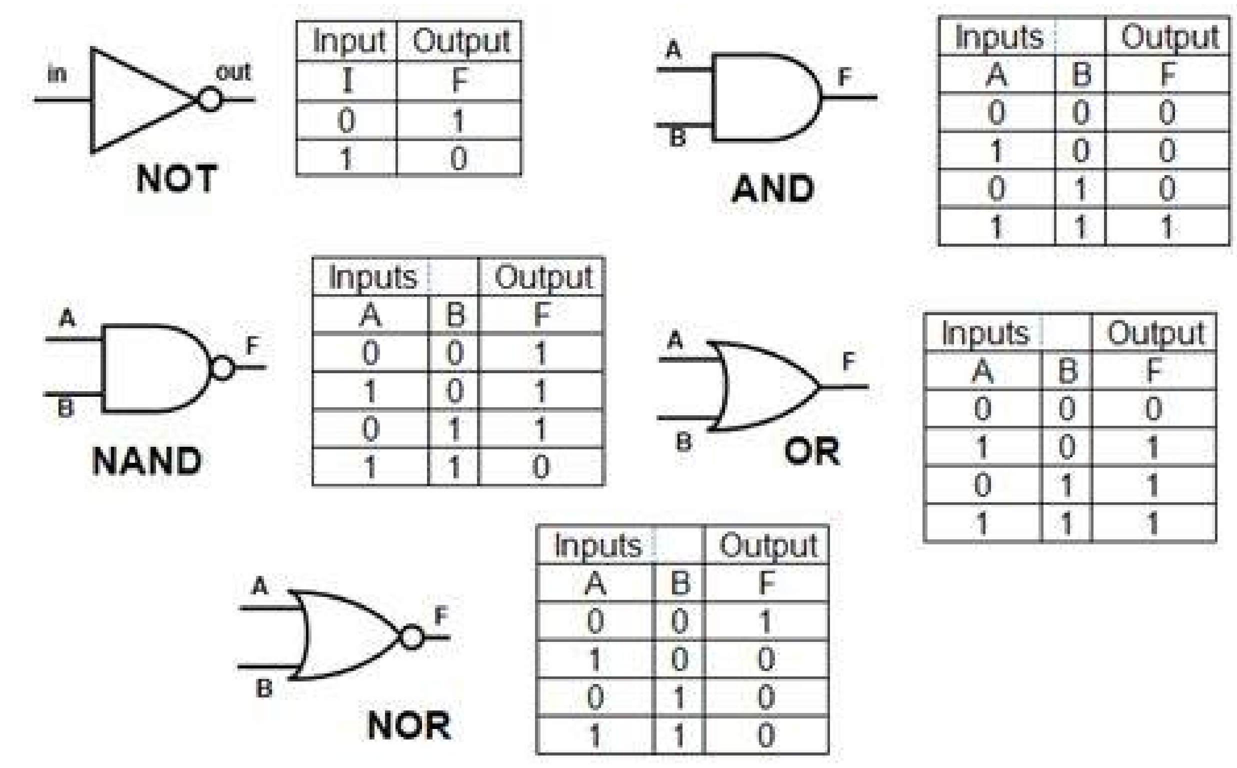
In this era, hard wiring is not any more necessary. Sensors and actuators can be interfaced to a controller, via cables, fieldbuses or any other communication protocol. PLC and microcontroller based solutions are the main controllers in industrial fields. PLC are programmed Usually in Ladder or in ST. Microcontrollers are programmed in C language. At the heart of a PLC, ther is a microcontroller, where is present a firmware to facilitate to programming.

1.2 Programming

1.2.1 Programming principles

Boolean algebra

Any CPU or microcontroller basically understand only logic operations. Main logic operations are `AND`, `OR`, and `NOT`. The following table resume the operations of these operators.



Input	Output
I	F
0	1
1	0

Inputs		Output
A	B	F
0	0	0
1	0	0
0	1	0
1	1	1

Inputs		Output
A	B	F
0	0	1
1	0	1
0	1	1
1	1	0

Inputs		Output
A	B	F
0	0	0
1	0	1
0	1	1
1	1	1

Inputs		Output
A	B	F
0	0	1
1	0	0
0	1	0
1	1	0

Fig. 1: Truth table and logic gates

In PLC ladder language is based on logic operations. More on this argument later.

1.3 C language

1.3.1 C++ shell

C language is chosen for different reasons. It is the king of all programming languages.

In order to try the examples, you can use the online shell: http://www.cpp.sh/. These shell is mainly a C++ compiler. Since C++ is compatible with C, we will use it in order to avoid you to install the compiler on your computer.

Fig. 2: C++ online shell

The following code is the main function, the entry point of any C program. For now we are interested in the `main` function.

Listing 1: C program

```
#include <stdio.h>
int main()
{
  return 0;
}
```

Basic syntax

Any programming language borrow some concepts from mathematics: operations, variables, values and functions.

Operations are:

- Addition
- Subtraction
- Multiplications
- Division

Values can be:

- Integers: 1, 2, 50, -10, ...
- Real numbers: 0.2 , 1.5 , 2.5
-

Variables are like in mathematics, can hold numeric and non numeric values.

In C and other languages (not all), we must declare a variable before using it.

Listing 2: C program

```
#include <stdio.h>
int main()
{
    int a=10;
    int sum;

    sum = a+ 12;

    printf("the sum = %d",sum);

    return 0;
}
```

C language have different types of numeric variables:

- int

- double
- float

Flow control

The execution of a program is usually sequential, It begin from the first instruction until the last one. Sometime we need to change the flow of execution. In C we have different contructs for flow control:

- if else
- switch case
- for
- while

Following a simple program than compare 2 variables.

Listing 3: If statement

```c
#include <stdio.h>
int main()
{
    int a=10;
    int b=30;

    if (a == b)
    {
        printf("a is equal to b");
    }
    else if ( a > b )
    {
        printf("a is bigger than b");
    }
    else
    {
        printf("a is smaller than b");
    }

    return 0;
  }
```

An equivalent to `if` is the `switch`.

Listing 4: Switch statement

```c
#include <stdio.h>
int main()
{
    int a=10;

    switch(a){
        case 0:
            printf("a is %d", 0);
            break;
        case 10:
            printf("a is %d", 10);
            break;
```

(continues on next page)

(continued from previous page)

```
        default:
            printf("Value not present");

    }
    return 0;
}
```

Functions

Functions are useful to group instructions that can be used more than one time and to make the program more readable. In the follwing example, a function called `max` is created.

Listing 5: Fucntion

```
#include <stdio.h>

int max(int a, int b)
{
    if (a> b)
        return a;
    else
        return b;
}

int main()
{
    int num =10, num2=20;

    int m;

    m = max(num , num2);
    printf("the maximum is %d", m);

    return 0;
}
```

1.4 Operating systems and firmwares

CHAPTER 2

Siemens PLC

Warning: Work in progress

2.1 Siemens PLC first steps

Note: All project are written in **TIA portal v15**. The exercise can be in any version, also in step 7.

2.1.1 S7-1200 Overview

We will use S7-1200 PLC. The model that we will be using is 1215C direct current (DC). The advantage of S7-1200 is the price and the integrated IO.

As shown in the image this PLC have 14 digital inputs (DI) and 10 digital outputs (DQ) and 2 analog inputs (AI) and 2 analog outputs (AQ). It have also High speed counters (HSC) and Pulse generators (PWM).

2.1.2 New Tia Portal project

In this section we will create a new Tia Portal project and create a new device. The new device will be the PLC we see in the previous section.

Set Ip Address

After creating a new PLC, the first step is to set its IP address. To set the Ip address, you need to open the property dialog of the PLC. If you click on the PLC image you need to go to `PROFINET interface [X1]`, `Ethernet addresses`. If you click on the Ethernet ports on the PLC image you can see directly the entry `Ethernet addresses`.

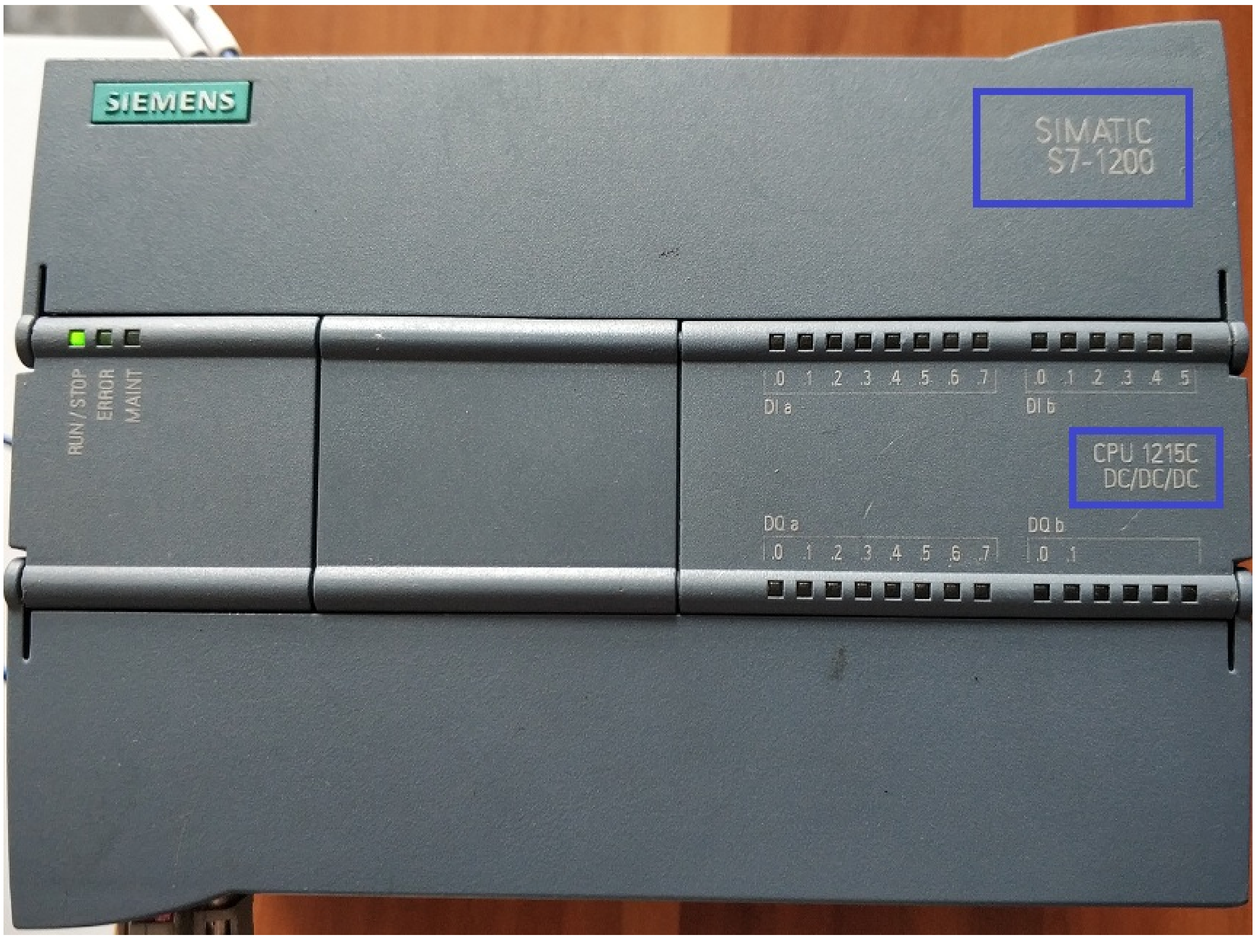


Fig. 1: Siemens S7-1200 PLC

Fig. 2: CPU 1215C DC/DC/DC 6ES7 215-1AG40-0XB0

Fig. 3: New TIA portal project
Create a new project and add S7-1200 PLC

Fig. 4: Set IP address

System and Clock memory

A clock in any CPU is necessary to provide timing. Select the PLC and in the property dialog check the 2 check boxes: `System memory bits` and `Clock memory bits`.

Fig. 5: System and Clock memory

Once these flags are checked, the PLC provide different system variables. For example `AlwaysTrue` is a variable that is always `true` i.e. have always value `1`. The variable `Clock_1Hz` is a variable that have the form of a square wave, where it is for `0.5s` is high and for `0.5s` low.

Tia portal navigation

Tia portal main windows is a dockable user interface. The following animation show how to navigate the main window.

Download configuration

Online and diagnostics

2.1.3 Simple Program

Lets suppose we wire a lamp to the first digital output of the PLC, labeled **DQa .0** on the PLC chassis. In the configuration of the PLC we give that output a name or a tag. The name can also be given in the PLC tags table. The following animation illustrate how to create a tag and write a small program in order to blink the lamp.

In this example we use the tag or variable `Clock_1Hz` in order to turn on and off the lamp, output, with a frequency of 1Hz. Remember, the clock have a wave square shape. If we want to blink the output with different timing, for example with a period of 2 seconds, the frequency that should be used is 1/2=0.5Hz. So `clock_0.5Hz` can be used.

`Download S7-1200 project`

2.1.4 S7-PLCSIM

2.1.5 Exercise S7-1500 HW configuration

`Download S7-1500 project`

2.2 Fundamental concepts

2.2.1 Memory Overview

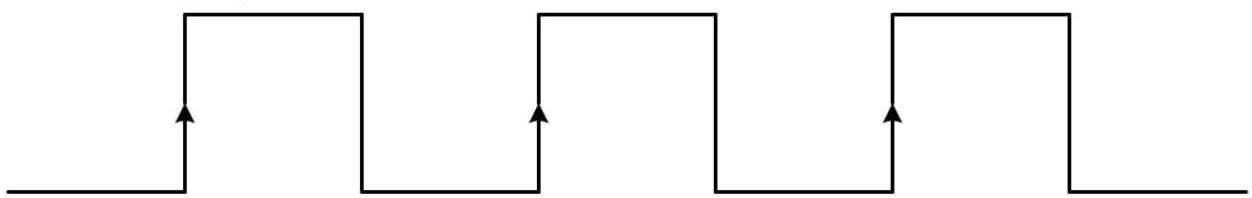


Fig. 6: CPU Clock
Remember that the time = 1/frequency

Fig. 7: TIA portal windows navigation

Fig. 8: Download configuration

Fig. 9: Online and diagnostics

Fig. 10: Blink an output with a frequency of 1Hz

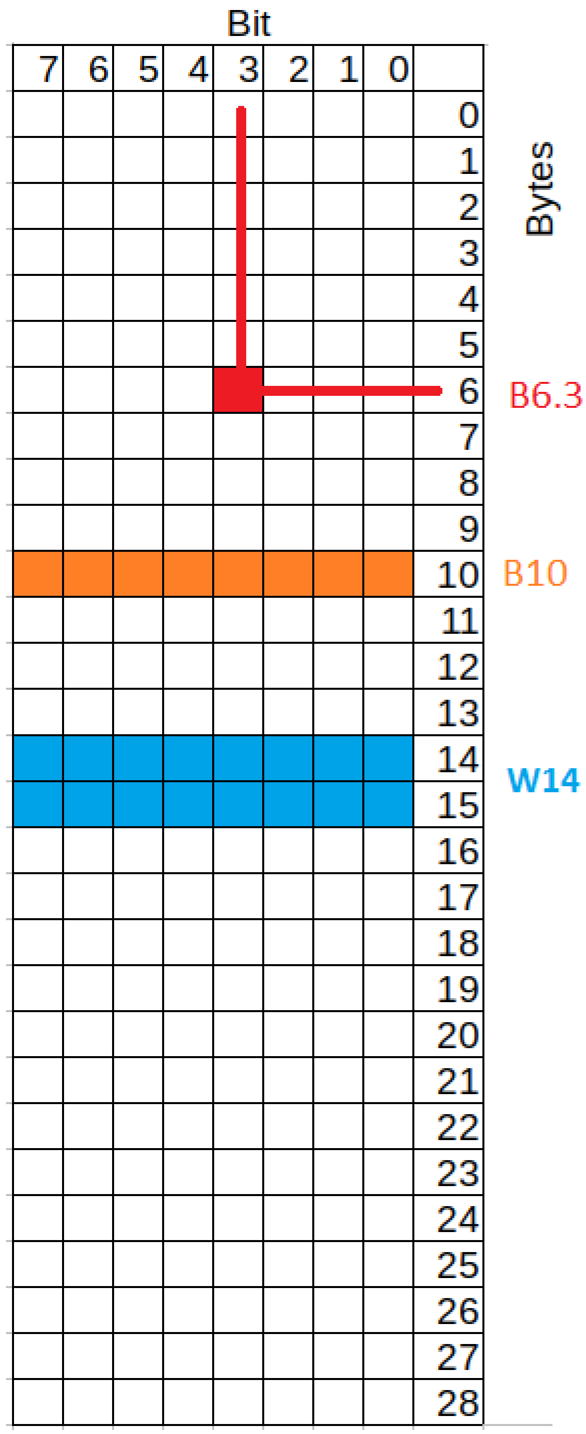


Fig. 11: Memory layout and addressing

Input and Output

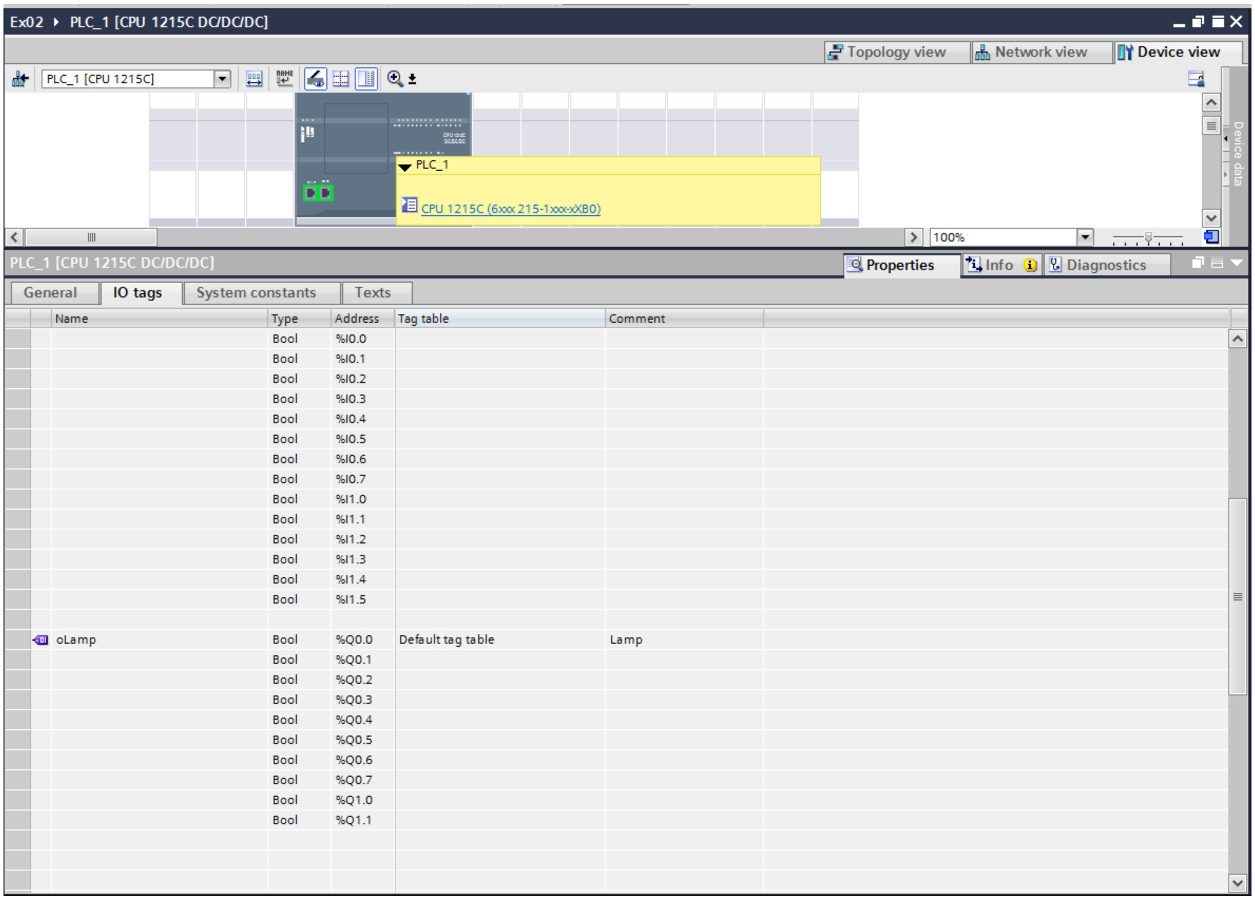


Fig. 12: S7-1200 integrated IO mapping

Fig. 13: PLC tags organization

Merker

Data Block

2.2.2 POU: Program Organization Unit

Organization Block

Function

Function Block

2.2.3 PLC programming languages

The standard IEC 61131-3 define 5 programming languages for PLC:

Name	Data type	Address
System_Byte	Byte	%MB1
FirstScan	Bool	%M1.0
DiagStatusUpdate	Bool	%M1.1
AlwaysTRUE	Bool	%M1.2
AlwaysFALSE	Bool	%M1.3
Clock_Byte	Byte	%MB0
Clock_10Hz	Bool	%M0.0
Clock_5Hz	Bool	%M0.1
Clock_2.5Hz	Bool	%M0.2
Clock_2Hz	Bool	%M0.3
Clock_1.25Hz	Bool	%M0.4
Clock_1Hz	Bool	%M0.5
Clock_0.625Hz	Bool	%M0.6
Clock_0.5Hz	Bool	%M0.7
<Add new>		

Fig. 14: Merker

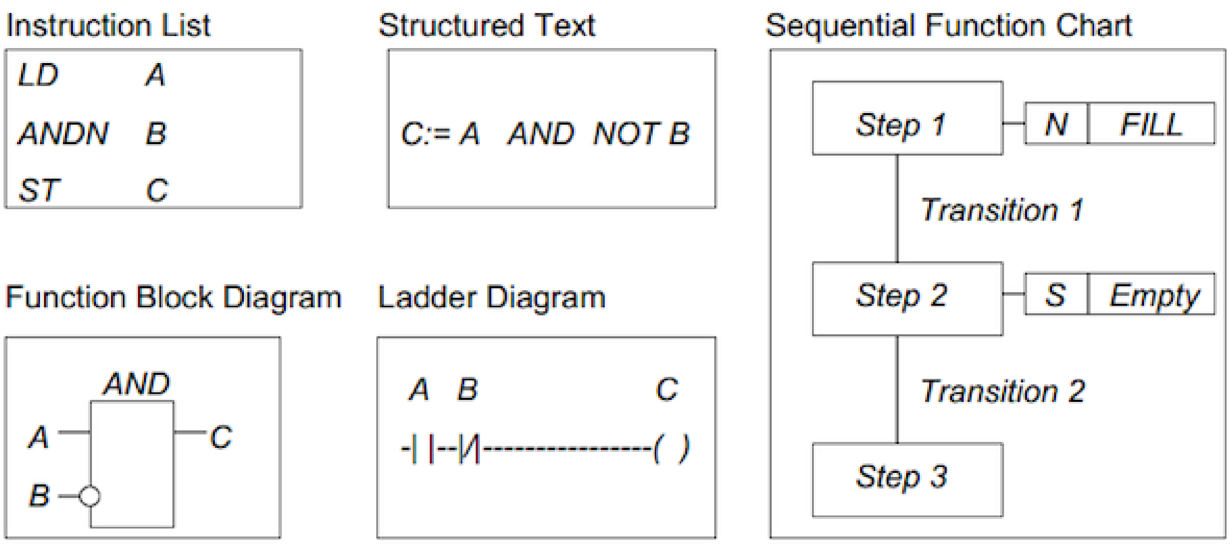
Fig. 15: Create new Data Block

Fig. 16: Using DB variables

Fig. 17: Organization Blocks

Fig. 18: Create and use a function as code organization

- IL: Instruction List (STL in Step7)
- LD: Ladder Diagram (LAD in step7)
- ST: Strucured Text (SCL in Siemens)
- SFC : Sequential Fucntion Chart
- FBD: Fucntion Block Diagram



2.3 Programming

`Download project Exercises.zip`

2.3.1 Basic operations

Contact and Coils

Trigger

Timers

Set Reset

2.3.2 SCL

If statement

Think about the `if` statement as you think in daily life. For example:

- If today is raining I take umbrella
- If it is cold I put a coat

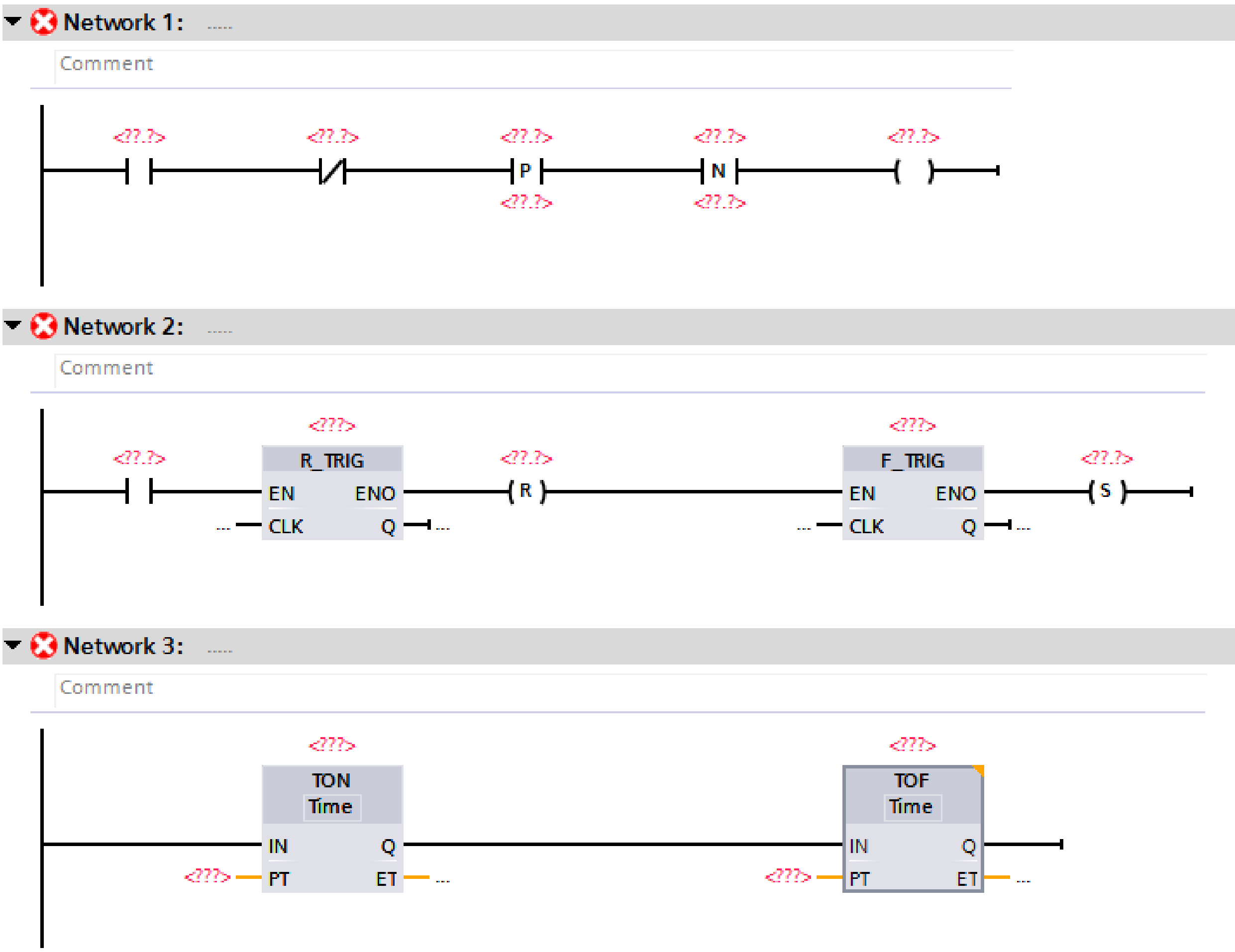
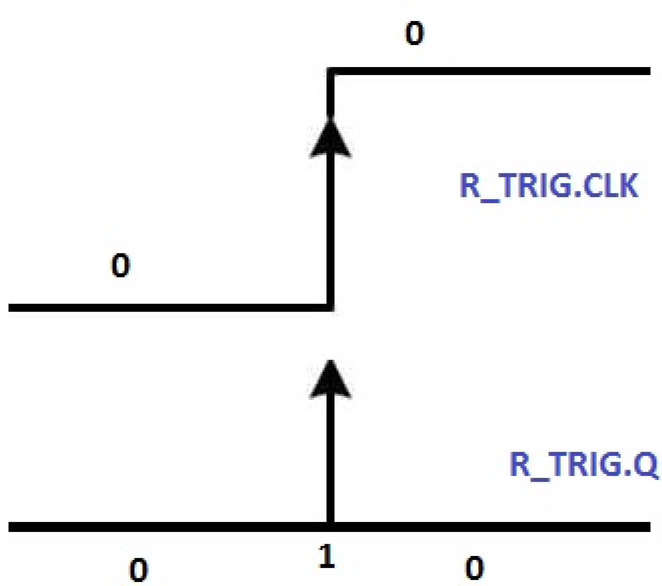


Fig. 19: Contact-Coil in ladder and its equivalent in SCL



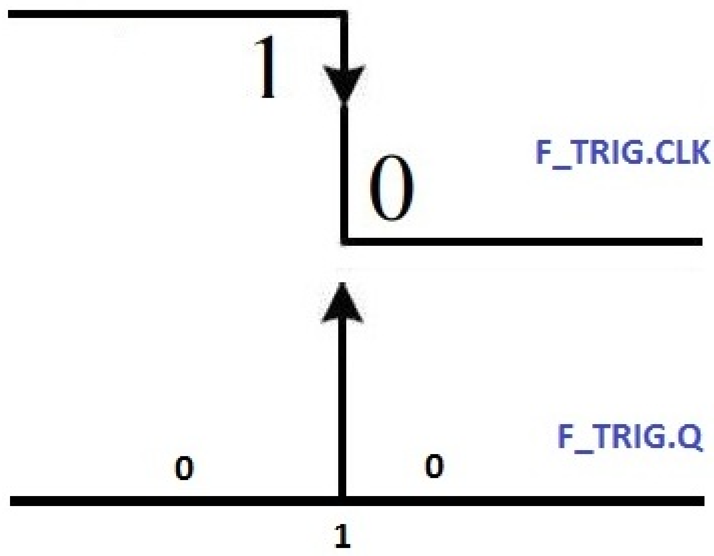


Fig. 20: R_TRIG positive signal edge in ladder

Fig. 21: R_TRIG positive signal edge in SCL

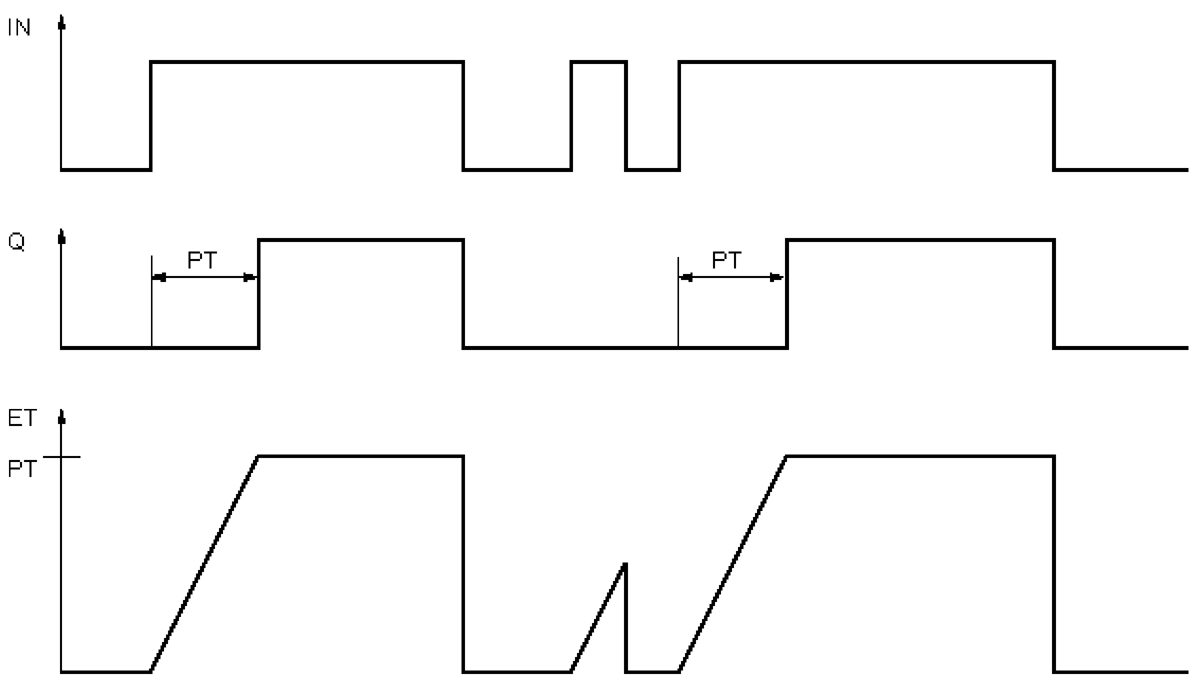


Fig. 22: TON (On delay) in ladder

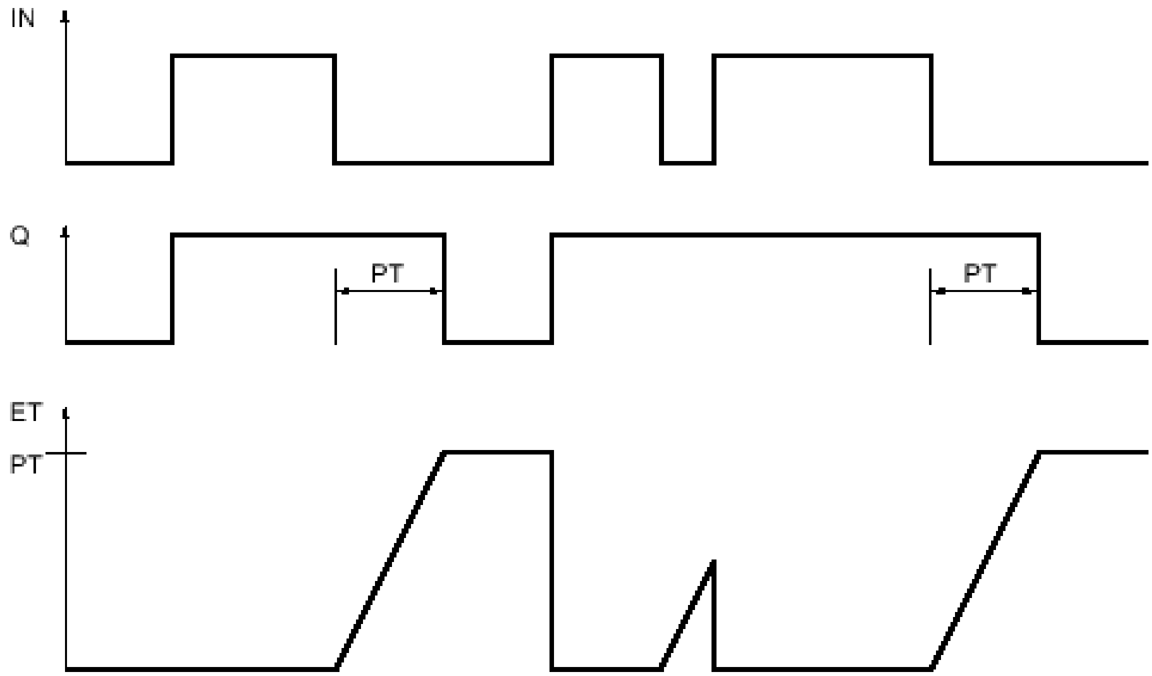


Fig. 23: TOF (Off delay) in ladder

Fig. 24: TON (On delay) in SCL

Fig. 25: Set Reset a signal

Fig. 26: Why the output didn't change value?

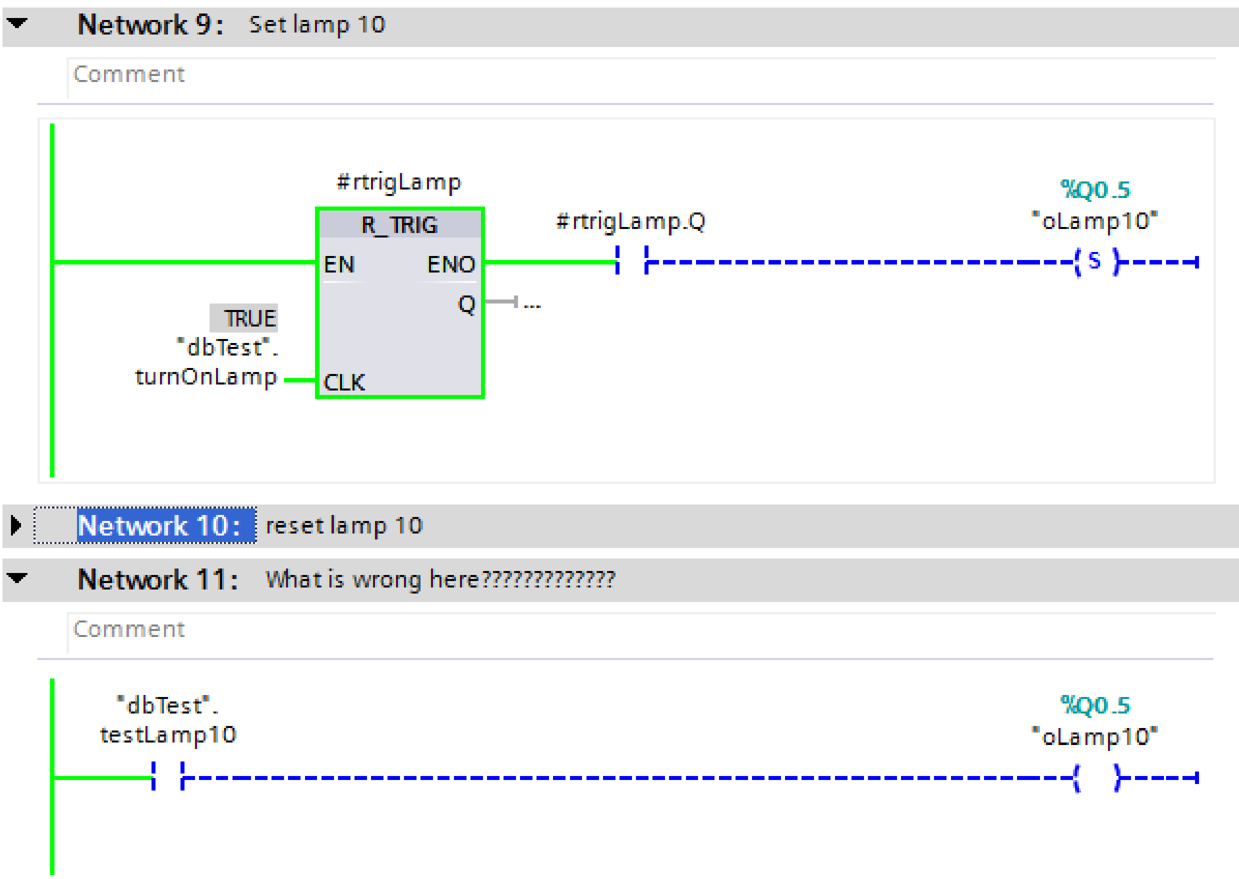


Fig. 27: What is wrong in this code ????

- I you find orange then buy, otherwise buy apple.

```
IF #a=5 THEN
    ;
END_IF;



IF #a = 10 THEN
    ;
ELSE
    ;
END_IF;

IF #a=11 THEN
    ;
ELSIF #a=12 THEN
    ;
ELSE
    ;
END_IF;
```

Fig. 28: If statement

Case statement

Case is like if, it check if the numerical value of the variable is present in the list, and execute the instruction corresponding to that value. For example let create a variable `day` of type `int`. The first day of the week is one the last day is seven. So If I want to make a decision tree, I list in the `case` statement days from 1 to 7, and for every value I do something:

- If day is 1 (Monday), I go to work
- If day is 2, I do something else
- ...
- If day is 6, I stay at home.

Remember that a case can be written also as an if.

The Case statement is more suitable in `state machine`. In Siemens there is no `enumeration` data type. In Tia portal siemens introduce CONSTANTS, so we can emulate an `enumeration`. It is more clear to have name than numbers. For example, is more clear to say Monday than day 1. And if Day 1 for me is Sunday? So is better to create a set of CONSTANTS with unique value and use them.

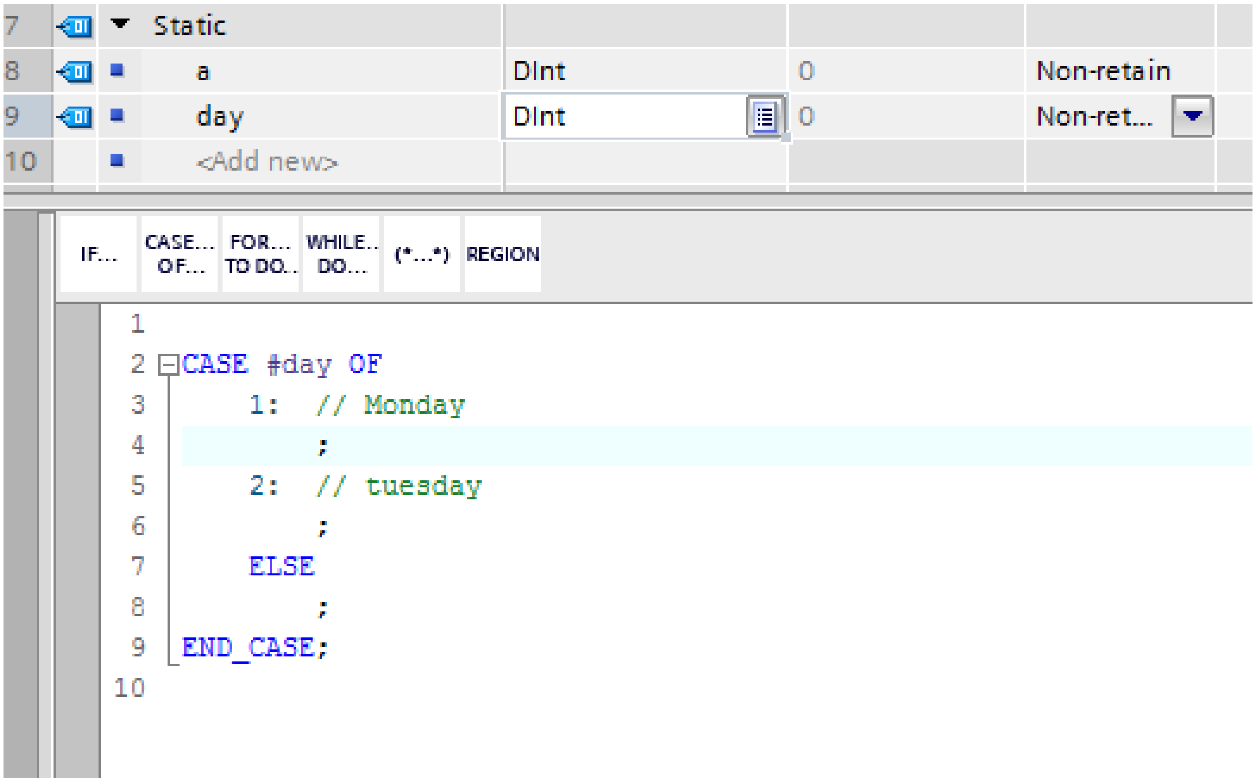


Fig. 29: Switch Case statement

```
int today;
const int MONDAY := 1;
const int TUESDAY := 2;
const int WEDNESDAY := 3;
const int THURSDAY := 4;
const int FRIDAY := 5;
const int SATURDAY := 6;
const int SUNDAY := 7;

CASE today OF
  MONDAY:
    I go to work;

  SATURDAY:
    I sleep more;

  ELSE:
    Error day is not recognized;

  END_CASE;
```

Loop

Try to avoid `for` and `while` in PLC programming if you don't know what are you doing. Infinite loops stop the plc.

2.4 Style guide

2.4.1 Project organization

Every project should have:

- README.md
- CHANGELOG.md
- Flowchart with yed, and converted in image(png or jpg)

The backup is projectName-Type-year-month-day-version-ProgrammerName.zip For example :

- Excersice01-PLC-2019-05-09-v0.0.1-Abed.zip
- A-JC-18-003-PLC-2019-05-09-v0.0.1-Abed.zip
- A-JC-18-003-ROBOT-2019-05-09-v0.0.1-Abed.zip

If in the same line have more than one robot, the robot id number should be the same as electrical drawings:

- A-JC-18-003-ROBOT01-2019-05-09-v0.0.1-Abed.zip
- A-JC-18-003-ROBOT02-2019-05-09-v0.0.1-Abed.zip
- A-JC-18-003-ROBOT03-2019-05-09-v0.0.1-Abed.zip
- A-JC-18-003-ROBOT04-2019-05-09-v0.0.1-Abed.zip

README

General informations about the project.

References

Special equipments

Short description about the workflow

CHANGELOG

The version is : major.minor.patch

The date is year-mont-day

[X.Y.Z] - aaaa-mm-dd Name(who) ### Added for new features. ### Changed for changes in existing functionality. ### Deprecated for soon-to-be removed features. ### Removed for now removed features. ### Fixed for any bug fixes. ### Security in case of vulnerabilities.

Flowchart or UML

Software used: https://www.yworks.com/products/yed/download

Every state machine should be illustrated in a chart (flowchart, uml,. . .).

2.4.2 Abbreviations

- Push button : pb, btn
- Lamp : lmp
- Limit switch : lsw
- Command : cmd
- Cylinder: cyl
- Table : tab
- Rotate : rot
- Robot : rob
- Machine : mach
- Panel view : hmi
- Actual : act
- Previous : prev
- Emergency : emrg, emr

Prefixes

- Input : i
- Output : q or o
- Analog input : ai
- Analog output : ao or aq
- Ethernet : eth
- Function block : FB
- Function : FC
- User data type: udt
- Structure: st

2.4.3 Names

S7 plc languages are not case sensitive, Button and button are the same variable.

Use `camelCase` for primitive data types: bool, word, dword, int, dint, real.

Use `PascalCase` for complex data types, and prefix them with the type:

- User defined data (udt, structures): udtConveyor, stConveyor
- AOI: AOI_Conveyor, AOI_Cylinder
- Function Block : FB_Conveyor, FB_Conveyor

The name of a variable should begin with the machine name, station name, component then function. For example: `conveyorMotorRun`, `conveyorMotorStop`, `conveyorLswPartPresent`.

`CONSTANT` variables in capital letters

Data blocks:

- Global data block: dbConveyor, dbRobot, dbCylinder
- Instance data block: idbConveyor, idbCylinder.

2.4.4 Rules

Rungs or segments must have a title

Rungs or segments should be commented in English, no Chinese nor other languages.

Every variable should have:

- Clear name
- Clear description
- If the variable is a signal, it should have the signal number as electrical drawing.

Every station have its own Function block, or own program in case of ControlLogix PLC.

Use state machine:

- Make state chart using OpenOffice draw or Yed software..
- Use unique numbers for states, use enumerations not numbers directly.

Cylinder:

> Cylinder states are: Opened, Closed. Cylinder commands are: Open, close. Don't use Forward, backward, up, down, left, right,. . .

2.4.5 Software organization

Functions (FB, FC) are the main building block of any program. The start point of S7 PLC is `OB1`, in `OB1` we should find only function calls. In `OB1` There is no business logic.

Every station should have is own main `FB` and global `DB` and instance `DB`. If the station have more then one component, every component should have its own `FB`. The components's `FB` should be instantiated in the `STAT` section of the parent `FB`. All functions and DBs of a station should be grouped in a folder.

`FB` that can be reused in different projects, should be placed in the `_Library` folder. A library with `FB` should be used.

Note: Follow example after training

2.5 Bad code

PLC programs usually are not structured well, neither follow best practice in software engineering. I notice that more than 95% of PLC programs are written in a horrible way, those are called bad code.

More experience a traditional PLC programmers have, more bad code he write. Reasons can vary from the leak of academic formation to other reasons. Even computer engineers write too bad code.

The main reason of bad code in PLC are come from the 2 dominant platforms: Siemens S7 and Allen Bradley PLCs. These platforms have a bad IDE and program organization. Even with Siemens new platform, TIA Portal, few things changed.

When someone begin to learn with these platforms, bad habits will accompany him for all his career. Using only one language or similar platform, is always a penalty.

A more advanced PLC based on CoDeSys and the standard IEC-61131, let you program a PLC like programming in C++. The IEC-61131 `ST` language have more features than Siemens and Allen Bradley PLCs. It support enumerations, classes, inheritances. Languages are variable name based, not address based like Siemens.

Tia portal become variable based, compared to the old Step 7. But Siemens keep function and data block numbers for an unknown reason. The reason can't be retro compatibility, if you open a project in Tia Portal 15, you can anymore open it in TIA portal 14.

2.5.1 Naming

2.5.2 Code reuse

A project have three cells, every cell have two rotating tables. The following snippet shown two function blocks without local variables for two tables in the same cell.

For the project the same logic was written six times at the beginning. During debugging a lot of malfunctioning were found. The six function blocks was modified again six times.

Another project have similar tables, the logic was written also 2 times for the 2 tables. In this project we can see also some difference in the program, even if the two tables should have the same logic.

At the end the logic of the turntable was written 8 times, and debugged more than 100 times. You can imagine how much time were wasted.

The logic of the same device in two different projects was written 2 times. If a function block with local variable was used, code duplication were avoided and time were saved.

2.5.3 General

In the following picture, a variable was assigned to other different variables, in different functions, before arriving to the output. During debugging is difficult to find any bug. Anyway this have no meaning.

When transferring data, e.g. from a recipe, group the variables in a `struct` and use block transfer. When dealing with assignments, it is better to use ST language than Ladder.

Two much conditions are present in this rung. When a rung become big, bigger than the screen it become difficult to debug.

2.6 Exercises

Note: We mean by function either FC or FB. Remember that an `FC` is a function without memory, it have only `temporary` variables. An `FB` is a function with memory, it have `static` variables.

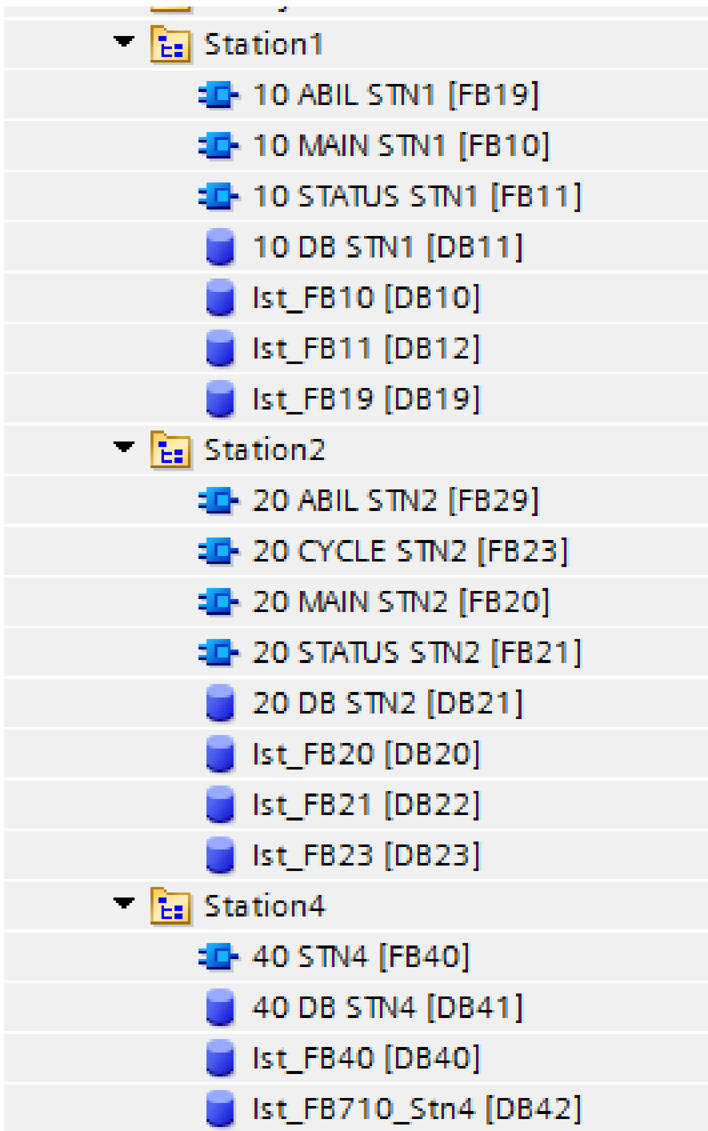


Fig. 30: Groups and functions without a good name

36	Tag_5	Bool	%M192.6	☑	☑
37	Tag_10	Bool	%M192.7	☑	☑
38	Tag_11	Bool	%M192.5	☑	☑
39	Tag_12	Bool	%M192.4	☑	☑
40	Tag_13	Bool	%M193.0	☑	☑
41	Tag_14	Bool	%M193.1	☑	☑
42	Tag_15	Bool	%M193.2	☑	☑
43	Tag_19	Bool	%M193.3	☑	☑
44	Tag_20	Bool	%M193.4	☑	☑
45	Tag_21	Bool	%M193.5	☑	☑
46	Tag_22	Bool	%M194.6	☑	☑
47	Tag_23	Bool	%M194.7	☑	☑
48	Tag_24	Byte	%IB23	☑	☑
49	Tag_27	Bool	%M194.0	☑	☑
50	Tag_30	Bool	%M194.1	☑	☑
51	Tag_31	Bool	%M194.2	☑	☑
52	Tag_32	Bool	%M194.3	☑	☑
53	Tag_33	Bool	%I51.5	☑	☑
54	Tag_35	Bool	%Q51.1	☑	☑
55	Tag_36	Bool	%Q51.0	☑	☑
56	Tag_37	Bool	%M300.0	☑	☑
57	Tag_41	Bool	%Q39.7	☑	☑
58	Tag_42	Bool	%M195.7	☑	☑
59	Tag_45	Bool	%M5000.1	☑	☑
60	Tag_49	Bool	%M1111.1	☑	☑
61	Tag_50	Timer	%T71	☑	☑

Fig. 31: Variables without name neither comment

2.6.1 Line equation

Analog signal need to be scaled to a physical unit in order to be understood. Usually analog sensors and actuators are modeled as linear systems. Write a function that map the value of an analog signal to a physical one (or from physical signal to analog one). For example, to map voltage to temperature, or to map current to pressure value, or to map a speed to voltage.

2.6.2 Rising edge

Write a function the detect the transition of a signal from 0 to 1. This function have the same functioning of the standard one `R_TRIG`.

2.6.3 Falling Edge

Write a function the detect the transition of a signal from 1 to 0. This function have the same functioning of the standard one `F_TRIG`.

2.6.4 Retentive TON

Write a function that count the time if a signal is 1. If the signal go to zero the function should stop counting. If the signal return to one, the function should continue to count from the previous value. Refer to the following timing diagram.

2.6.5 Blink

Write a function that toggle an output, with a determined frequency. The duty cycle of the signal can be tuned. Remember the duty cycle is the time (or percentage) of the time when the signal is high. In this exercise use time not

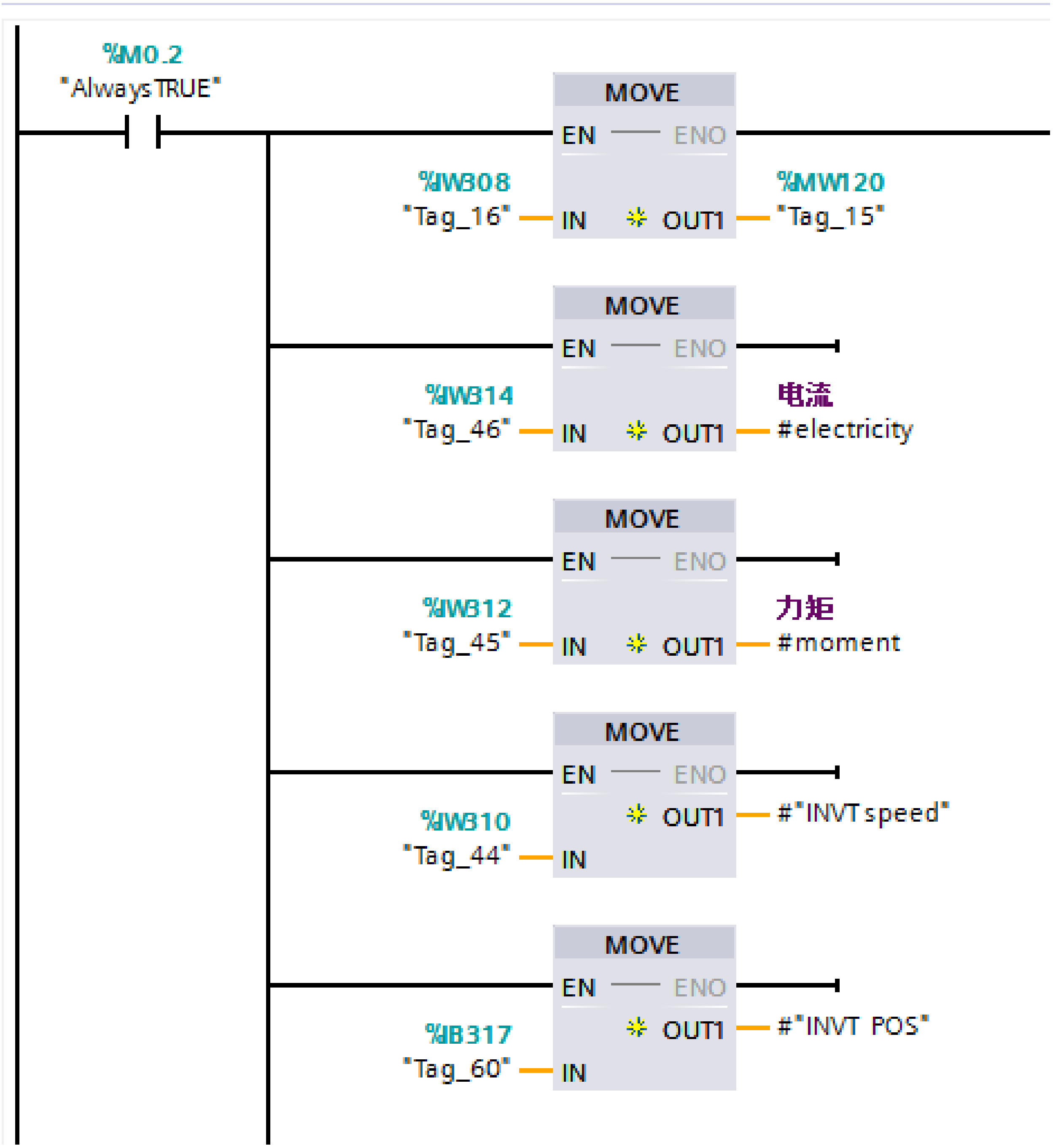


Fig. 32: How can remember the meaning of the variables?

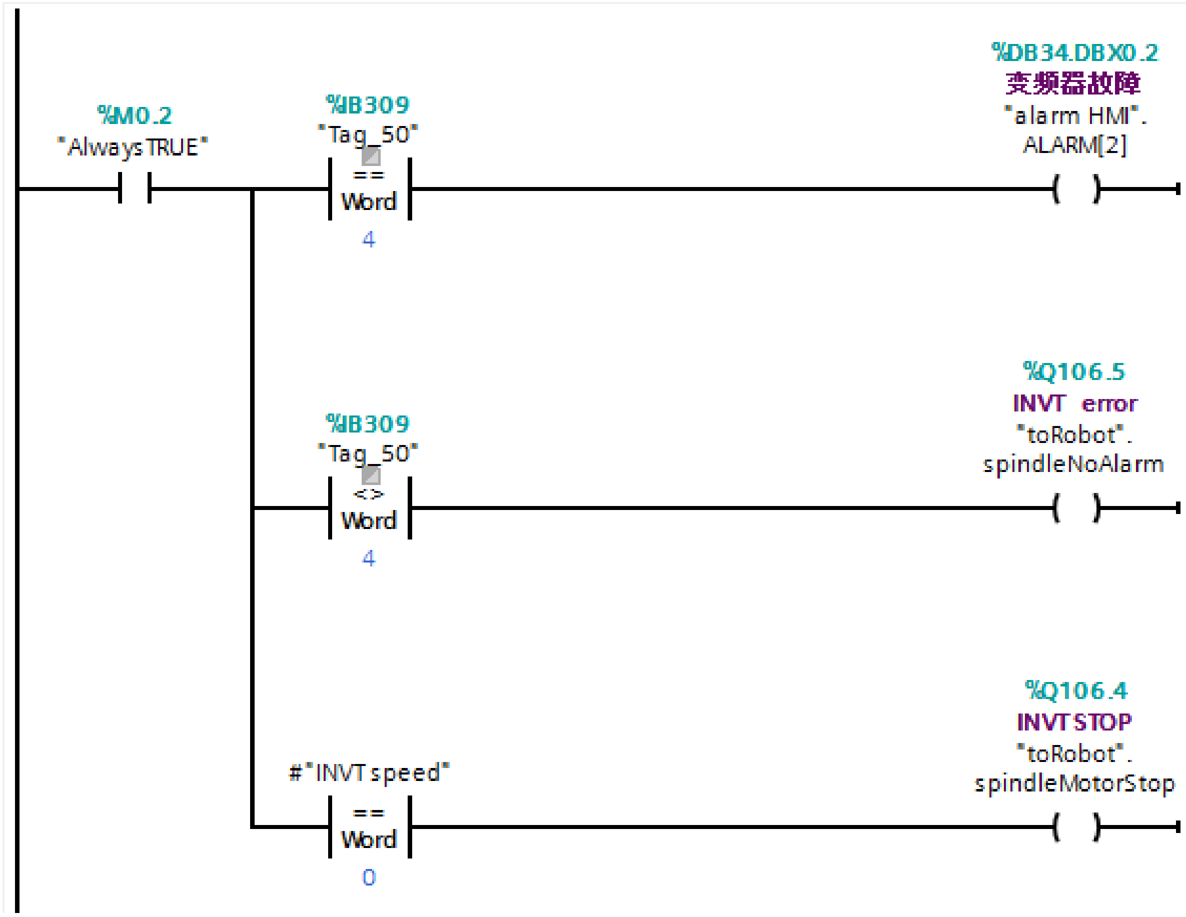


Fig. 33: How can remember the meaning of the variables?

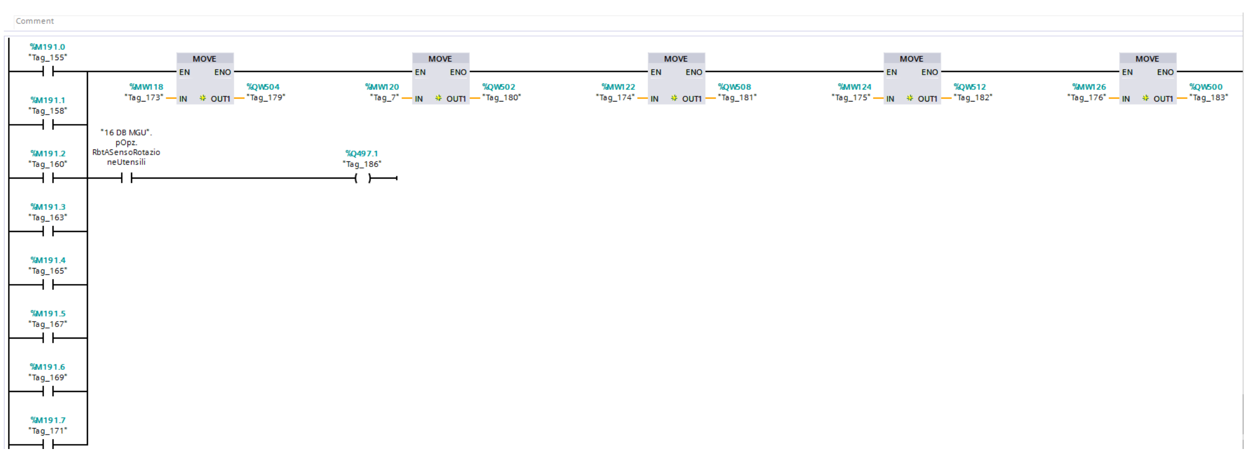


Fig. 34: No significant name

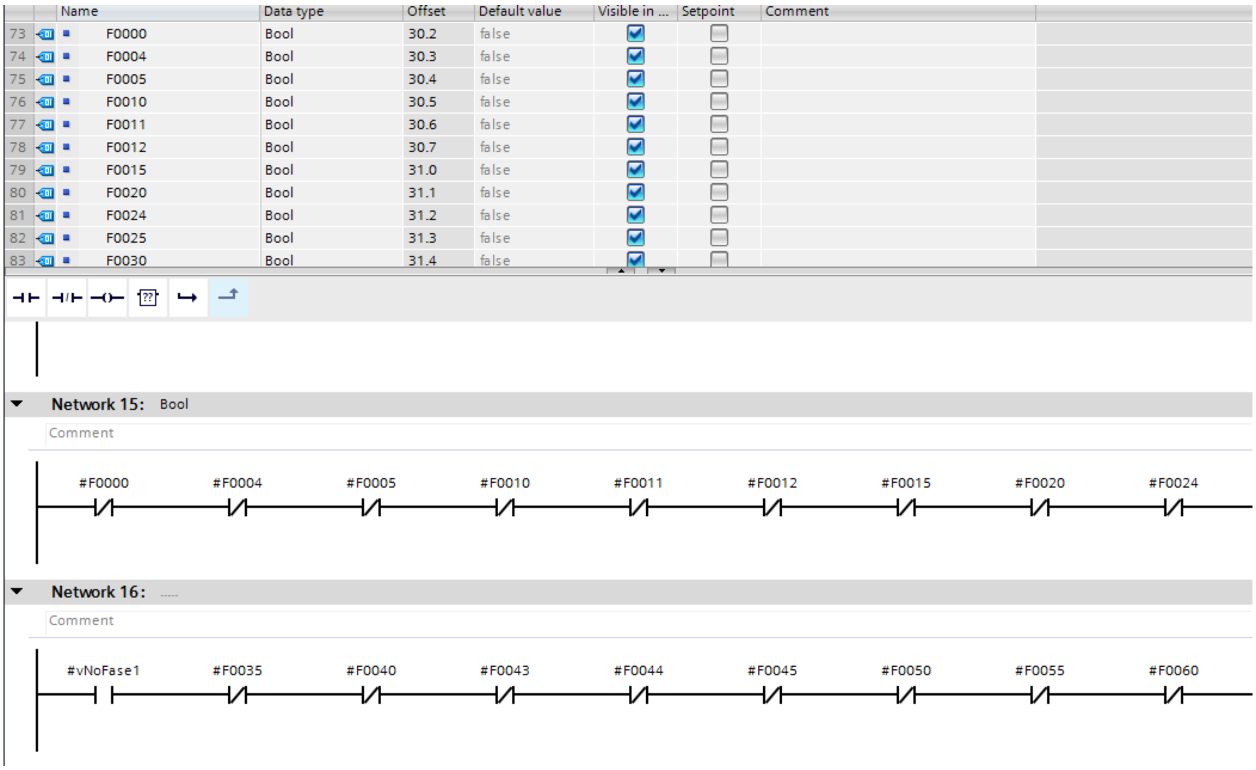


Fig. 35: State machine without state name neither comment

percentage.

2.6.6 Bi-stable cylinder

Write a function that control a cylinder. Imagine all digital input and outputs that are necessary to the correct functioning of the cylinder, as also any other signal or variable (not only physical input or output).

2.7 Solutions

Note: Complete and tested solution can be found in the OpenLib Library

`Download Exercises solutions TIA Portal 15`

Functions written in SCL can be exported and imported.

You can copy the code of the solution to a text file, save it with extension `.scl` then import it to TIA. Otherwise check the project file if you have Tia Portal version greater than 15.

2.7.1 Line equation

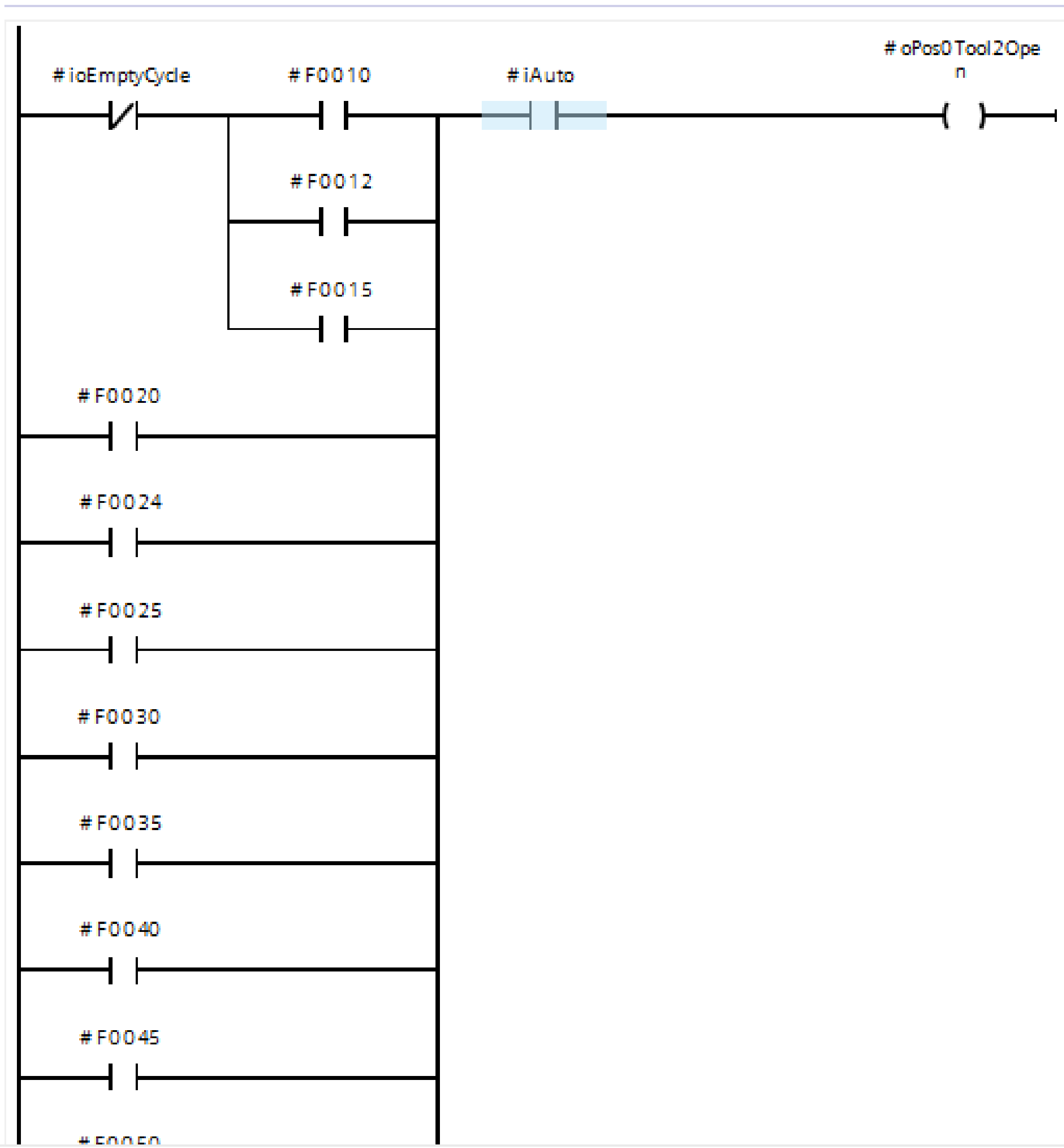


Fig. 36: State machine without state name neither comment

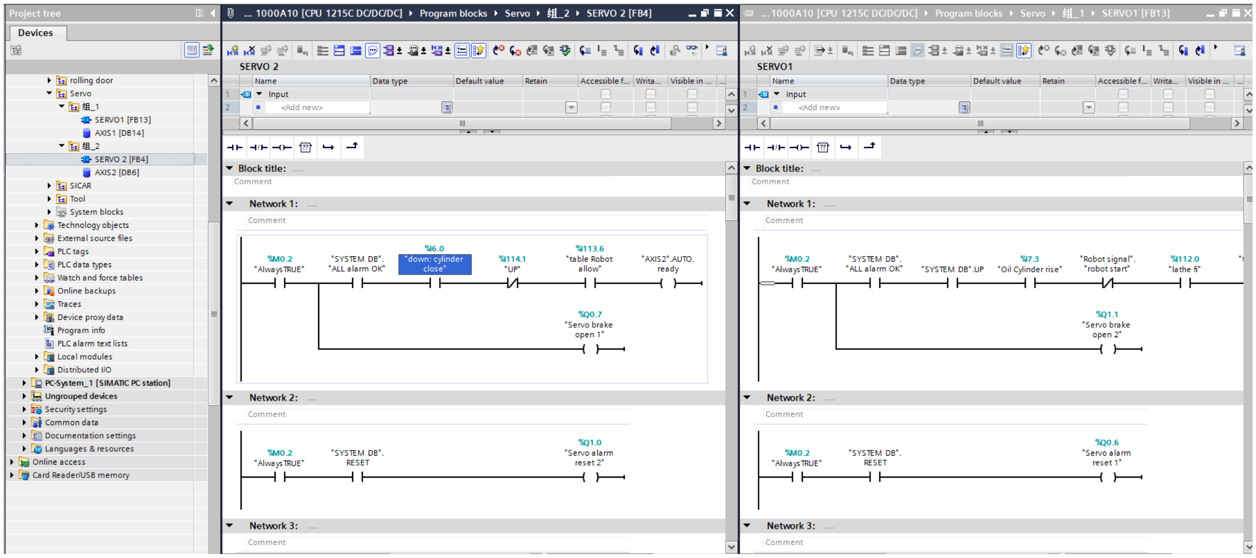


Fig. 37: Two tables in Project 1

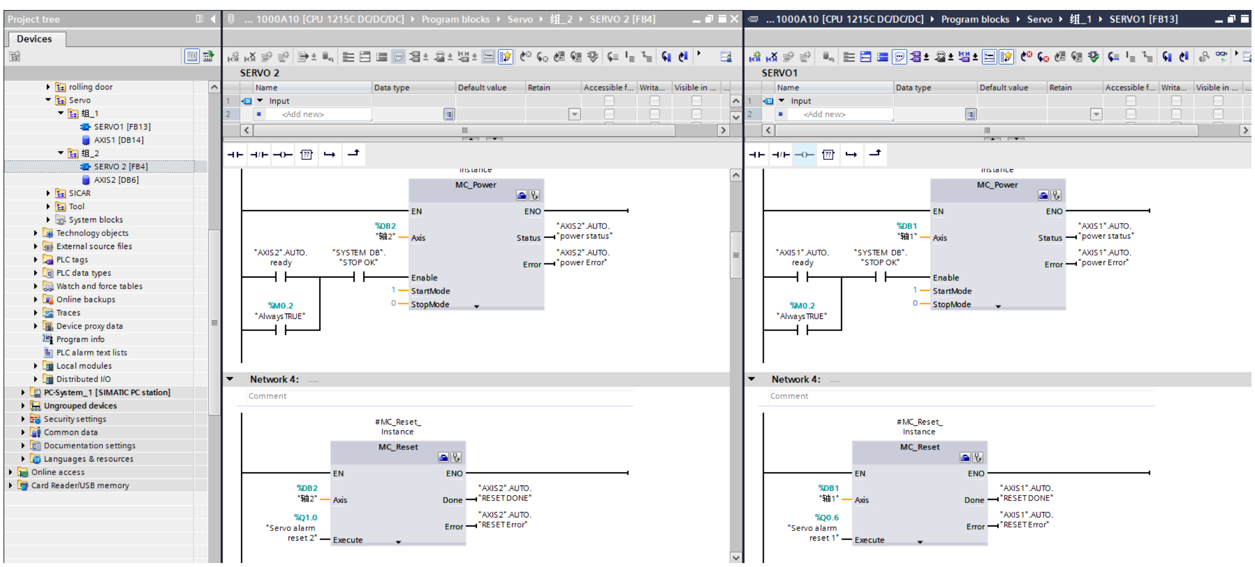


Fig. 38: Two tables in Project 1

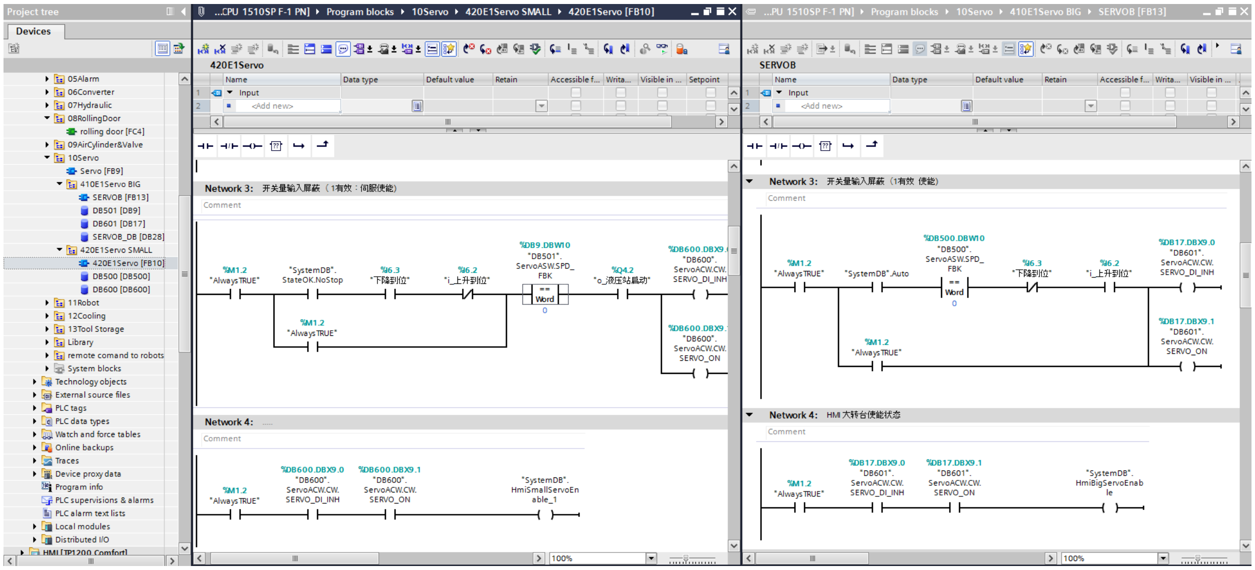


Fig. 39: Two tables in Project 2

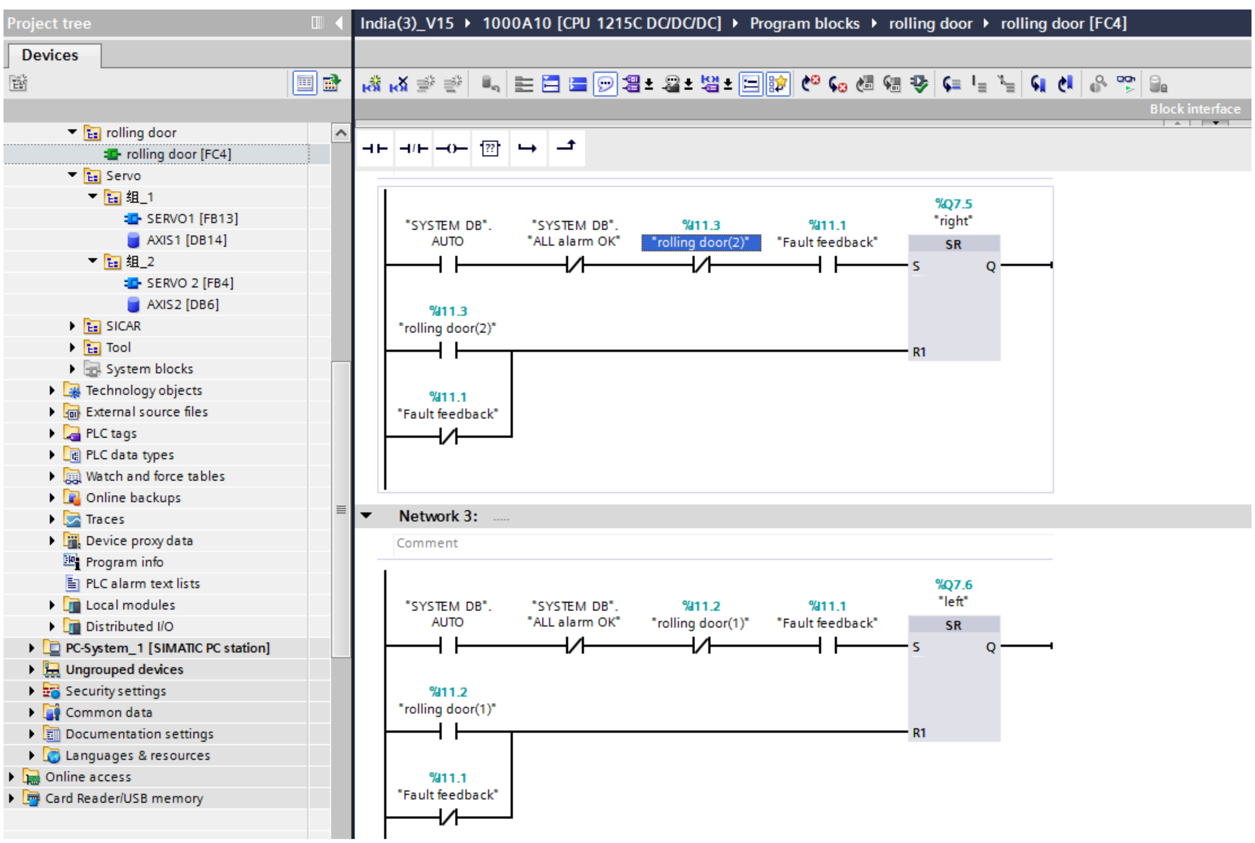


Fig. 40: Rolling shutter in Project 1

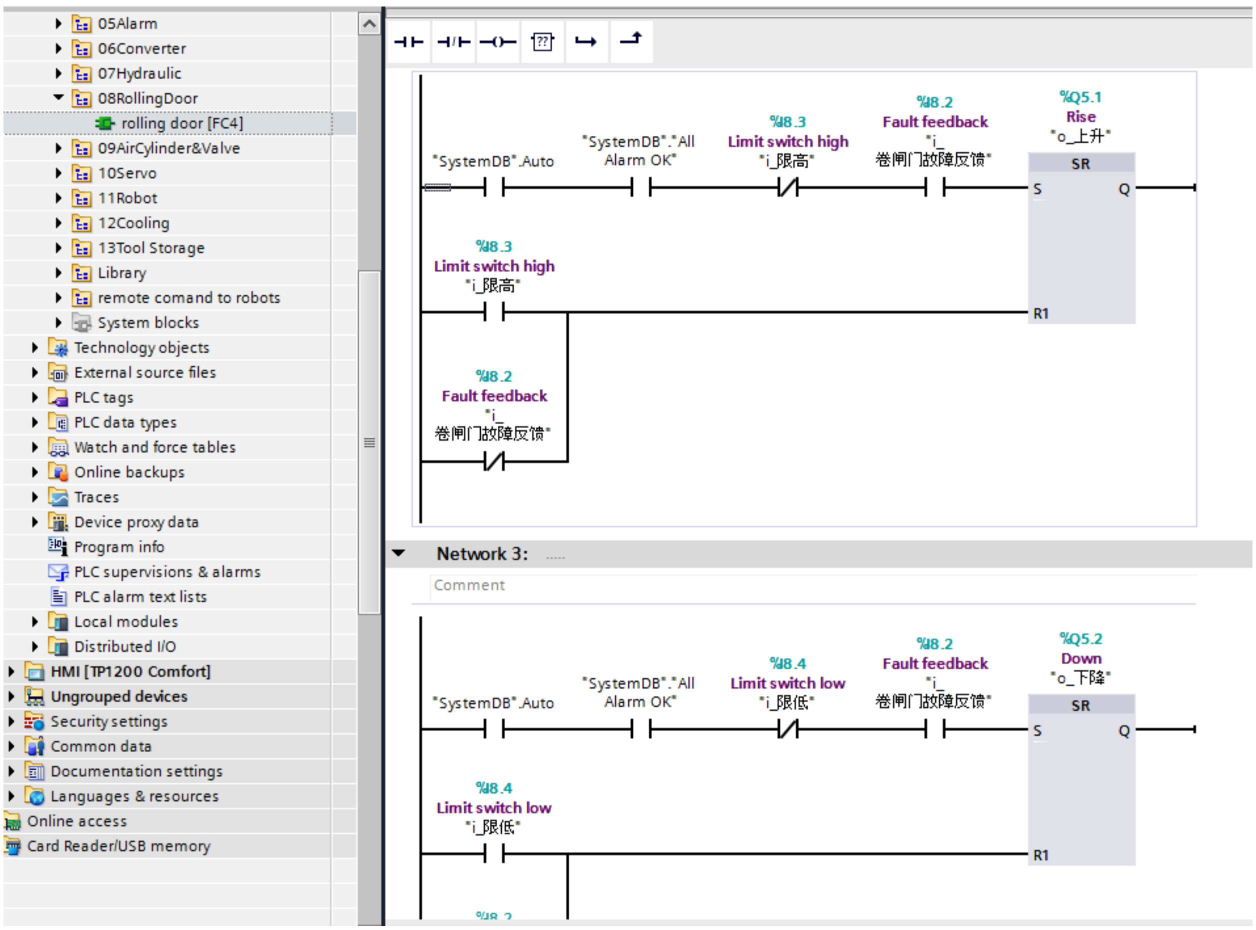
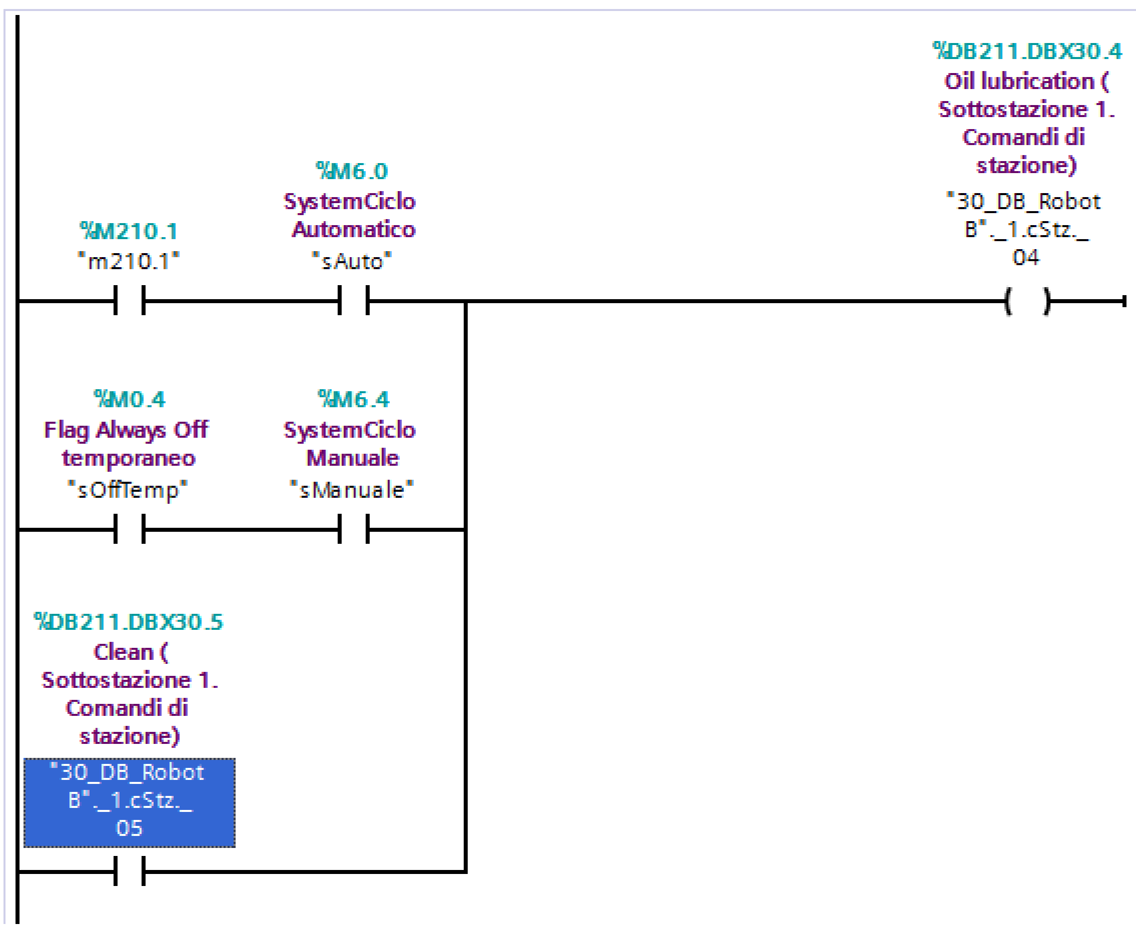
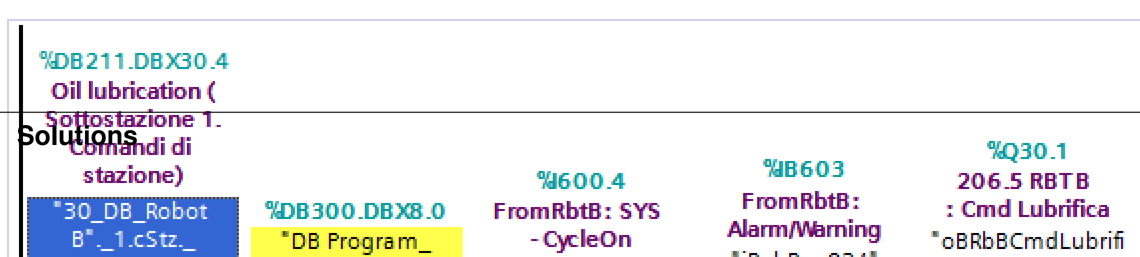


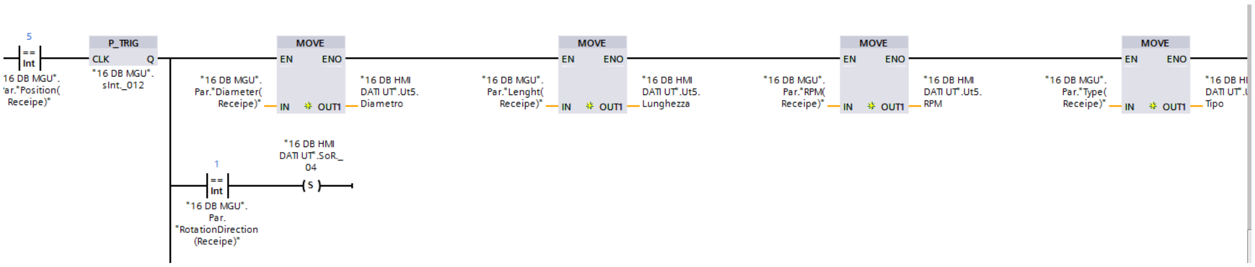
Fig. 41: Rolling shutter in Project 2

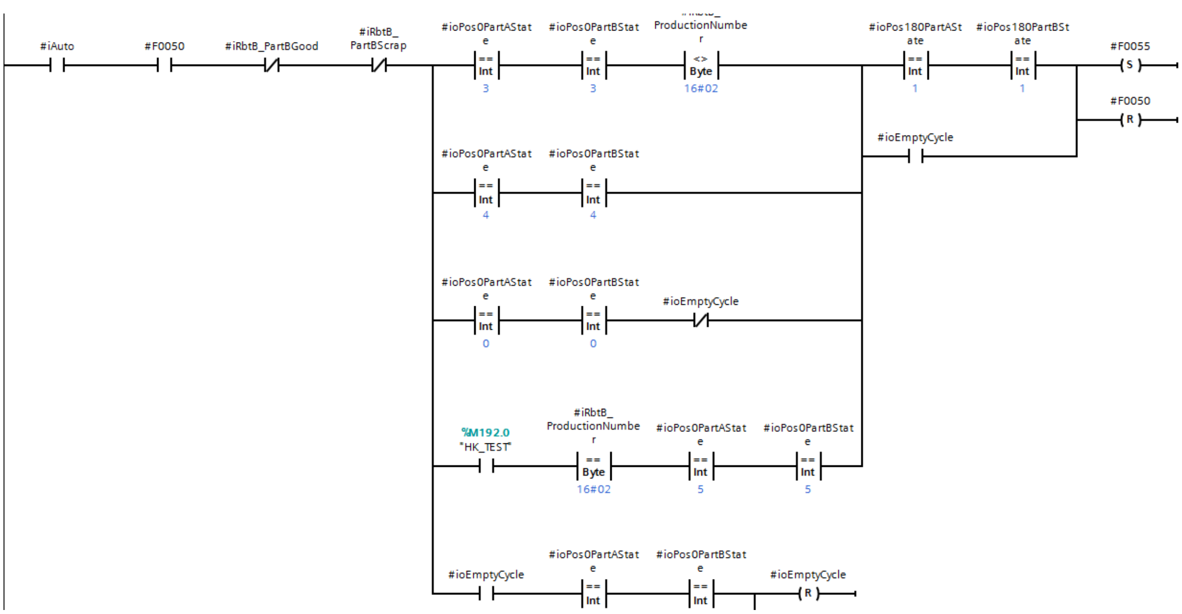

%I608.4
FromRbtB: Spindle - Clean
"iRobB_o068"
%DB211.DBX24.4
Spindle Clean from robot
"30_DB_Robot B"._1.rAuto._04

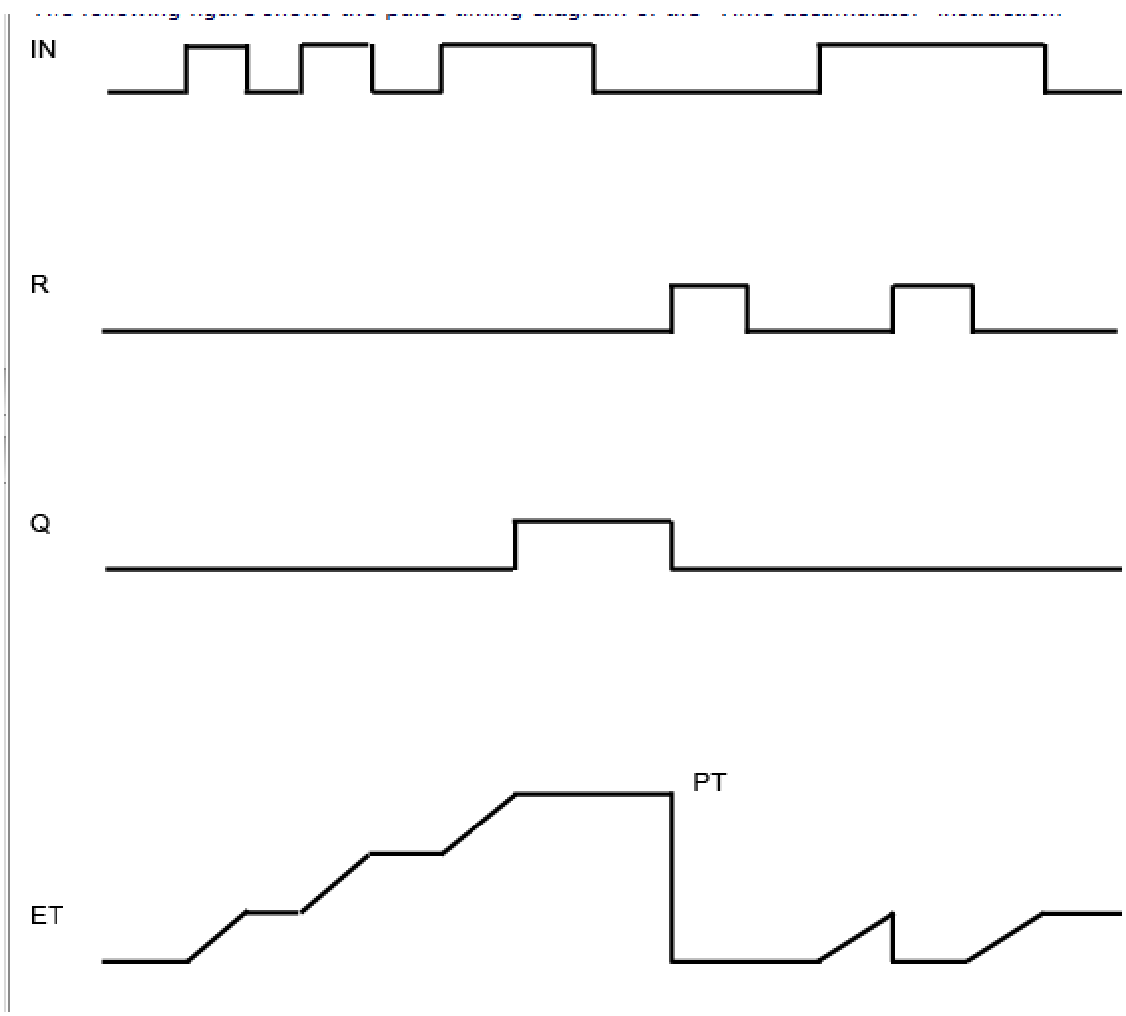

%DB211.DBX24.4
Spindle Clean from robot (Sottostazione 1. Richieste da Ciclo Automatico)
"30_DB_Robot B"._1.rAuto._04
%DB211.DBX30.5
Clean (Sottostazione 1. Comandi di stazione)
"30_DB_Robot B"._1.cStz._05


%DB211.DBX30.4
Oil lubrication (Sottostazione 1. Comandi di stazione)
"30_DB_Robot B"._1.cStz._04
%M210.1
"m210.1"
%M6.0
SystemCiclo Automatico
"sAuto"
%M0.4
Flag Always Off temporaneo
"sOffTemp"
%M6.4
SystemCiclo Manuale
"sManuale"
%DB211.DBX30.5
Clean (Sottostazione 1. Comandi di stazione)
"30_DB_Robot B"._1.cStz._05


%DB211.DBX30.4
Oil lubrication (Sottostazione 1. Comandi di stazione)
"30_DB_Robot B"._1.cStz._
%DB300.DBX8.0
"DB Program_
%I600.4
FromRbtB: SYS - CycleOn
%IB603
FromRbtB: Alarm/Warning
%Q30.1
206.5 RBTB : Cmd Lubrifica
"oBRbBCmdLubrifi


P_TRIG
CLK
Q
MOVE
EN
ENO
IN
OUT1
"16 DB HMI DATI UT".Ut5.Diametro
"16 DB HMI DATI UT".Ut5.Lunghezza
"16 DB HMI DATI UT".Ut5.RPM
"16 DB HMI DATI UT".SoR_04
"16 DB MGU".Par."Diameter(Receipe)"
"16 DB MGU".Par."Lenght(Receipe)"
"16 DB MGU".Par."RPM(Receipe)"
"16 DB MGU".Par."Type(Receipe)"
"16 DB MGU".Par."RotationDirection(Receipe)"
"16 DB MGU".sInt._012
Tipo
Int


#iAuto
#F0050
#iRbtB_PartBGood
#iRbtB_PartBScrap
#ioPos0PartAState
#ioPos0PartBState
#iRbtB_ProductionNumber
#ioPos180PartAState
#ioPos180PartBState
#F0055
#ioEmptyCycle
%M192.0
"HK_TEST"
Int
Byte
16#02



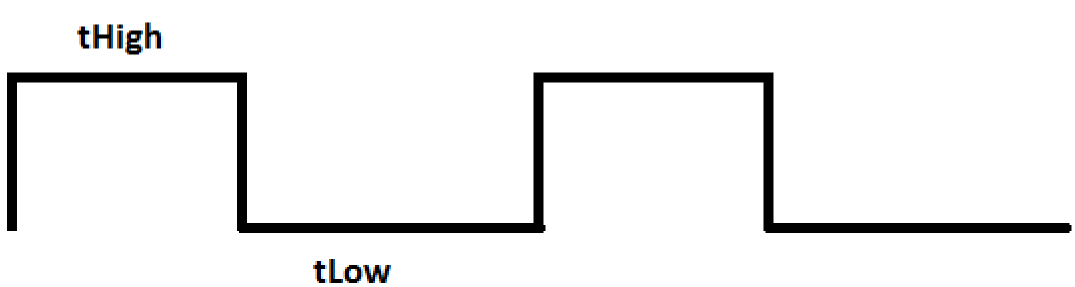


Fig. 42: Preset times tHigh (on) and tLow (off) can be set as desired

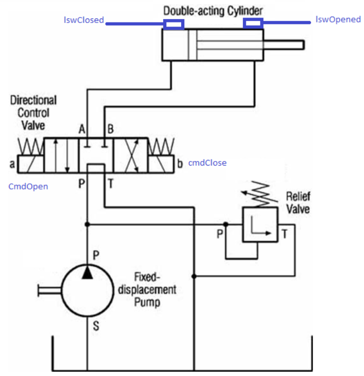


Fig. 43: Double acting cylinder

Fig. 44: Export a function (FB or FC) written in SCL to a file

```
FUNCTION "LineEquation" : Void
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
   VAR_INPUT
      x : Real;
      xA : Real;
      yA : Real;
      xB : Real;
      yB : Real;
   END_VAR

   VAR_OUTPUT
      y : Real;
   END_VAR

   VAR_TEMP
      m : Real;
   END_VAR


BEGIN
        // Analog input
        // x is Analog input (INT)
        // y is the physical meausre (REAL) (temperature, pressure,speed,....)

        // Analog output
        // x is the physical meausre (REAL) (temperature, pressure,speed,....)
        // y is Analog output (INT)

        #m := (#yA - #yB) / (#xA - #yA);

        #y := #m * (#x - #xA) + #yA;
END_FUNCTION
```

(continues on next page)

Fig. 45: Import an external source and generate the function

(continued from previous page)

Suppose we have a temperature sensor connected to the analog input of the PLC. The analog input read an `int`, 16-bit signed value between -32768 (-2^15) and 32767 (2^15 - 1).

Table A- 190 Analog input representation for voltage   12 bit ADC

System		Voltage Measuring Range						
Decimal	Hexadecimal	±10 V	±5 V	±2.5 V	±1.25V		0 to 10 V	
32767	7FFF	11.851 V	5.926 V	2.963 V	1.481 V	Overflow	11.851 V	Overflow
32512	7F00							
32511	7EFF	11.759 V	5.879 V	2.940 V	1.470 V	Overshoot range	11.759 V	Overshoot range
27649	6C01							
27648	6C00	10 V	5 V	2.5 V	1.250 V	Rated range	10 V	Rated range
20736	5100	7.5 V	3.75 V	1.875 V	0.938 V		7.5 V	
1	1	361.7 μV	180.8 μV	90.4 μV	45.2 μV		361.7 μV	
0	0	0 V	0 V	0 V	0 V		0 V	
-1	FFFF						Negative values are not supported	
-20736	AF00	-7.5 V	-3.75 V	-1.875 V	-0.938 V			
-27648	9400	-10 V	-5 V	-2.5 V	-1.250 V			
-27649	93FF					Undershoot range		
-32512	8100	-11.759 V	-5.879 V	-2.940 V	-1.470 V			
-32513	80FF					Underflow		
-32768	8000	-11.851 V	-5.926 V	-2.963 V	-1.481 V			

The s7-1200 AI data sheet show the mapping between tension (voltage) and corresponding numerical value.

The temperature sensor datasheet, will show the mapping between the tension and the temperature. In the PLC program we have to map from AI numerical value to tension, than from tension to temperature.

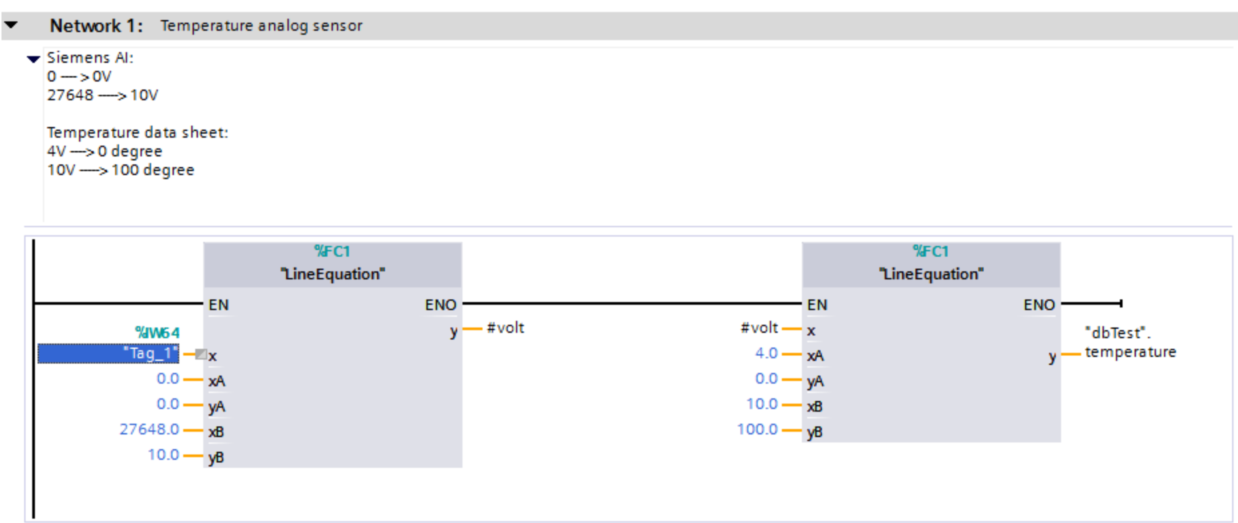


Fig. 46: Use example of linear function

2.7.2 Rising edge

2.7.3 Falling Edge

2.7.4 Retentive TON

2.7.5 Blink

```
FUNCTION_BLOCK "Blink"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
   VAR_INPUT
      enableDI : Bool;
      timeHigh : Time;
      timeLow : Time;
   END_VAR

   VAR_OUTPUT
      Q : Bool;
   END_VAR

   VAR
      timer_High {InstructionName := 'TON_TIME'; LibVersion := '1.0'} : TON_TIME;
      timer_Low {InstructionName := 'TON_TIME'; LibVersion := '1.0'} : TON_TIME;
      bOn {InstructionName := 'TON_TIME'; LibVersion := '1.0'} : TON_TIME;
      bOff {InstructionName := 'TON_TIME'; LibVersion := '1.0'} : TON_TIME;
   END_VAR


BEGIN
        #timer_High(IN := (#enableDI AND NOT #timer_Low.Q),
                    PT := #timeHigh);
        #timer_Low(IN := #timer_High.Q,
                   PT := #timeLow);

        #Q := #timer_High.Q;
END_FUNCTION_BLOCK
```

2.7.6 Bi-stable cylinder

A simple and functional solution in ladder is presented. A complete solution can be found in the library, and a state machine implementation can be found in the `state machine` chapter.

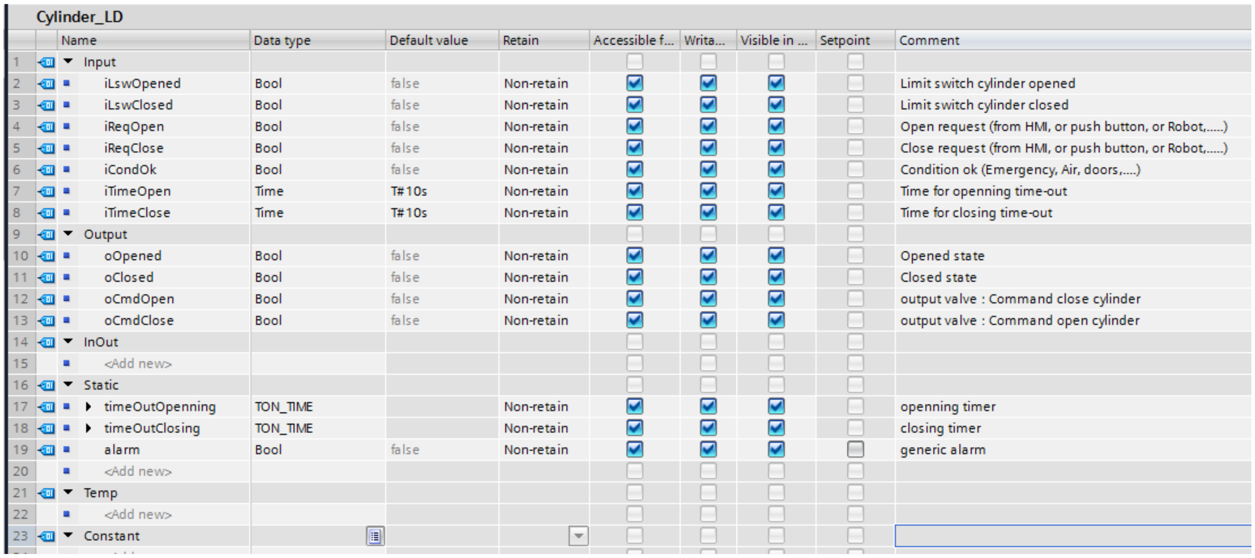
Physical IO may be:

- Two digital inputs: proximity sensors
- Two digital outputs: valve solenoid

Interaction with operators may be via physical push buttons, or software buttons (from HMI). The interaction may be with other devices like robots or the PLC itself depending on the plant. But from our point of view they are all the same, and we summarize them as open and close requests.

We can add also a stop request, and other things. But for now, we keep the solution simple.

The cylinder in normal operations, at rest, can be in a single state, or opened or closed.



Cylinder_LD

	Name	Data type	Default value	Retain	Accessible f...	Writa...	Visible in ...	Setpoint	Comment
1	▼ Input								
2	iLswOpened	Bool	false	Non-retain	✓	✓	✓		Limit switch cylinder opened
3	iLswClosed	Bool	false	Non-retain	✓	✓	✓		Limit switch cylinder closed
4	iReqOpen	Bool	false	Non-retain	✓	✓	✓		Open request (from HMI, or push button, or Robot,....)
5	iReqClose	Bool	false	Non-retain	✓	✓	✓		Close request (from HMI, or push button, or Robot,....)
6	iCondOk	Bool	false	Non-retain	✓	✓	✓		Condition ok (Emergency, Air, doors,....)
7	iTimeOpen	Time	T#10s	Non-retain	✓	✓	✓		Time for openning time-out
8	iTimeClose	Time	T#10s	Non-retain	✓	✓	✓		Time for closing time-out
9	▼ Output								
10	oOpened	Bool	false	Non-retain	✓	✓	✓		Opened state
11	oClosed	Bool	false	Non-retain	✓	✓	✓		Closed state
12	oCmdOpen	Bool	false	Non-retain	✓	✓	✓		output valve : Command close cylinder
13	oCmdClose	Bool	false	Non-retain	✓	✓	✓		output valve : Command open cylinder
14	▼ InOut								
15	<Add new>								
16	▼ Static								
17	▶ timeOutOpenning	TON_TIME		Non-retain	✓	✓	✓		openning timer
18	▶ timeOutClosing	TON_TIME		Non-retain	✓	✓	✓		closing timer
19	alarm	Bool	false	Non-retain	✓	✓	✓		generic alarm
20	<Add new>								
21	▼ Temp								
22	<Add new>								
23	▼ Constant								

Fig. 47: Variables and interface

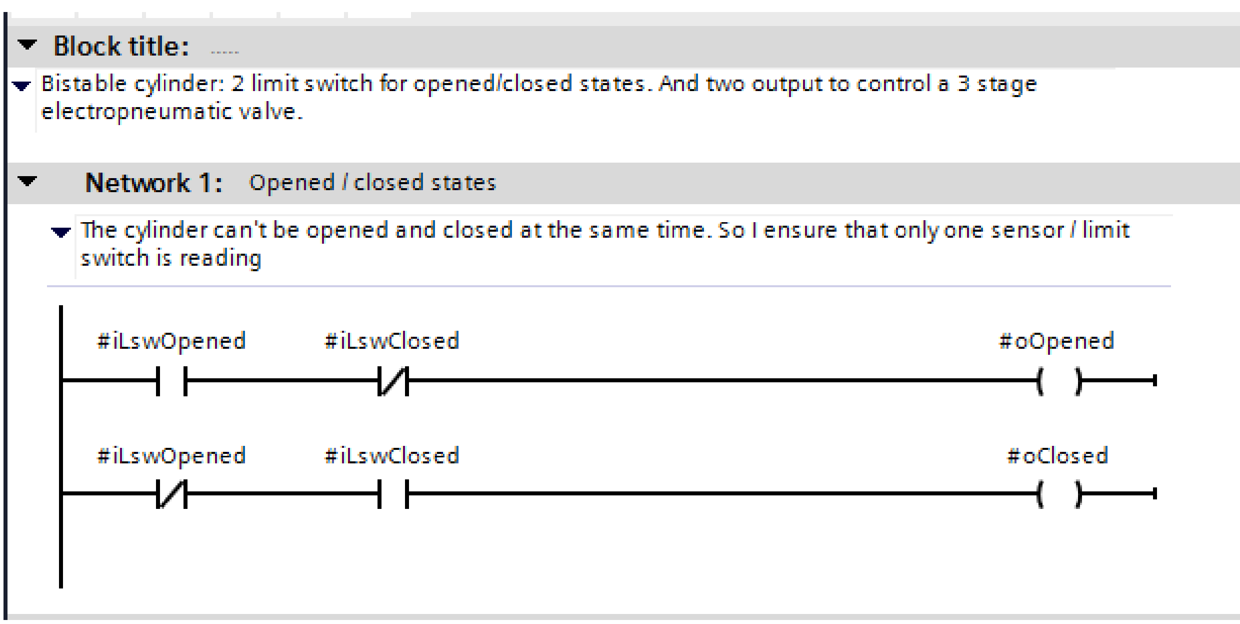


Fig. 48: States: Opened and closed

The cylinder can be opened, if it is not opened and receive a request to open. What if someone send the request to open and close in the same time? So we need to be sure to receive only one request.

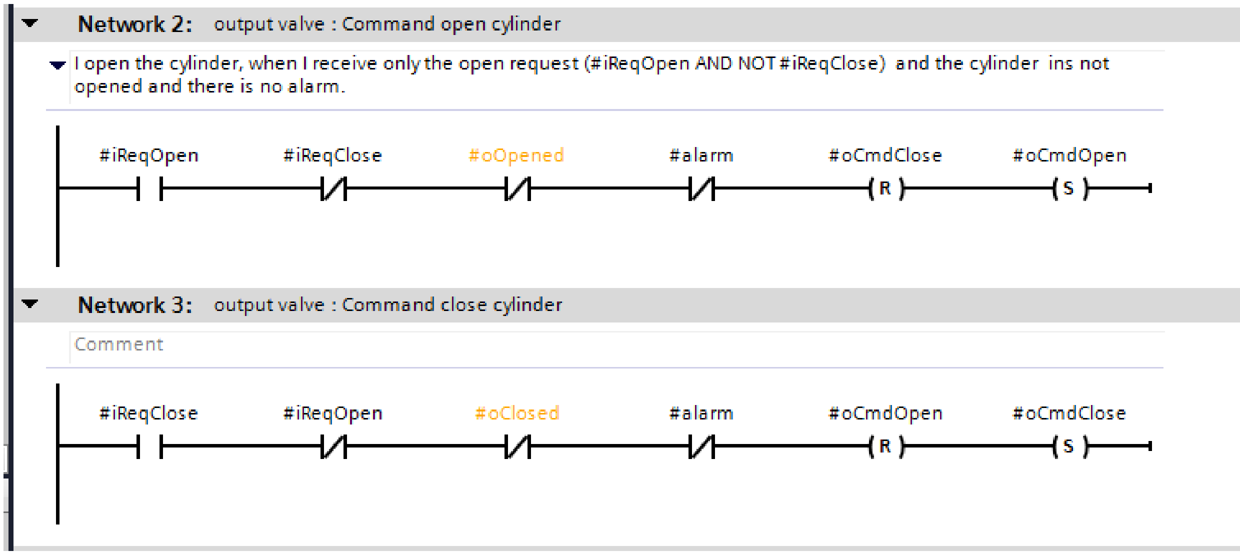


Fig. 49: Commands: open and close

The cylinder may not respond to our requests, maybe there is no compressed air. Or the command execution was interrupted, e.g. heavy load, or someone leave some object in middle of the way. The execution time for opening and closing may be different, e.g. the cylinder take more time to open because it push some heavy object, but while closing is free from any load.

When we send the opening request and we didn't get the opened state for a predefined time, we have an abnormal situation. Keep in mind, the predefined time is greater than the normal operating time, and it differ from application to application. For example, if the cylinder takes normally 5 seconds to open, we set the time to 7 seconds or 8 seconds for the time out.

When we get the time out signal, the commands should be resetted . . . .

2.8 State machine

Note: State machine diagram are drawing in yEd Graph editor from yWorks.

`Download Exercises solutions`

2.8.1 Concepts

A state machine have 2 componets:

- `State` represented as a circle.
- `Transition` represented as an arrow. The transition is the condition to change state.

For example a lamp may have 2 states: ON or OFF. The transition from one state to another is determined by a switch.

When writing software, first we begin with normal operations i.e. how the device should work, then we add abnormal situations. For example, we say a lamp may have only two states, in normal operations. But a lamp may be broken.

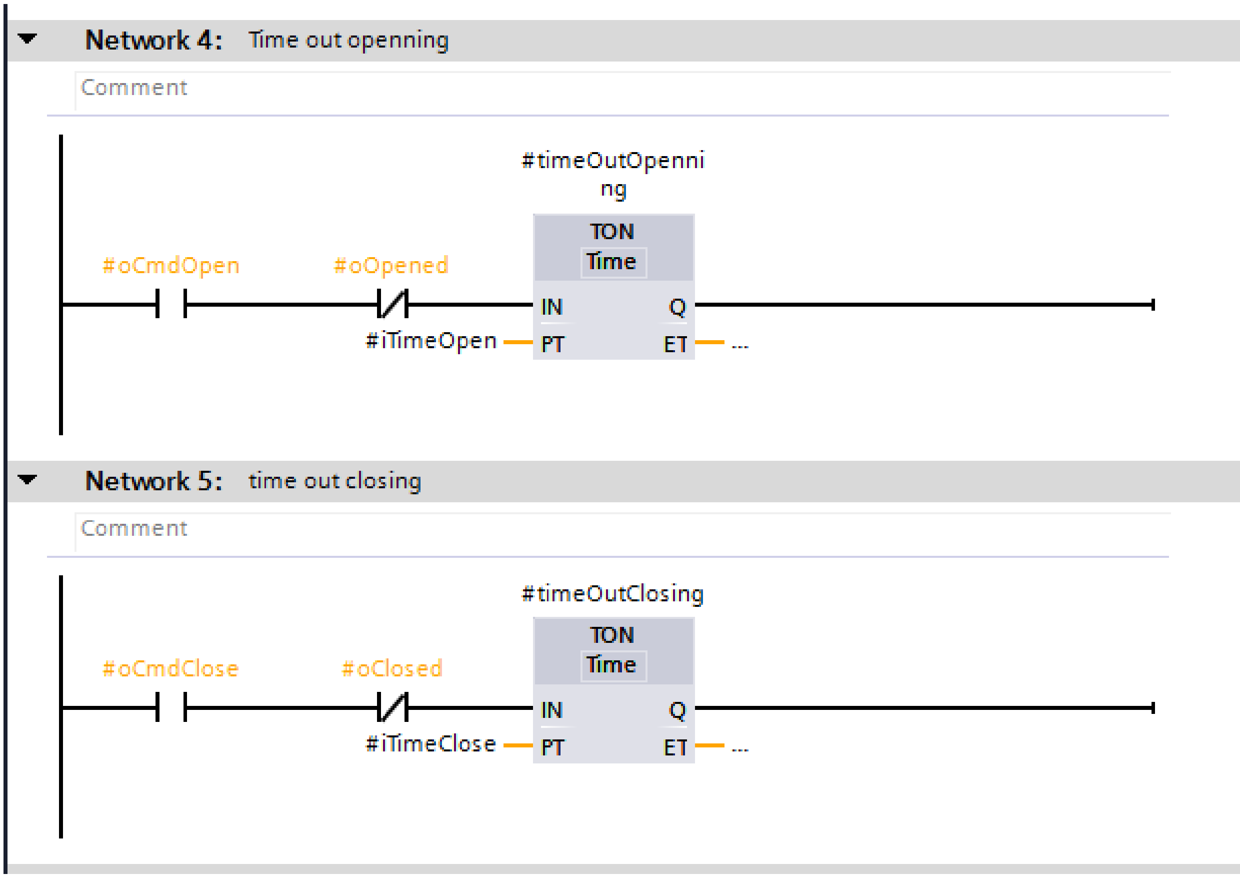


Fig. 50: Time outs: opening and closing

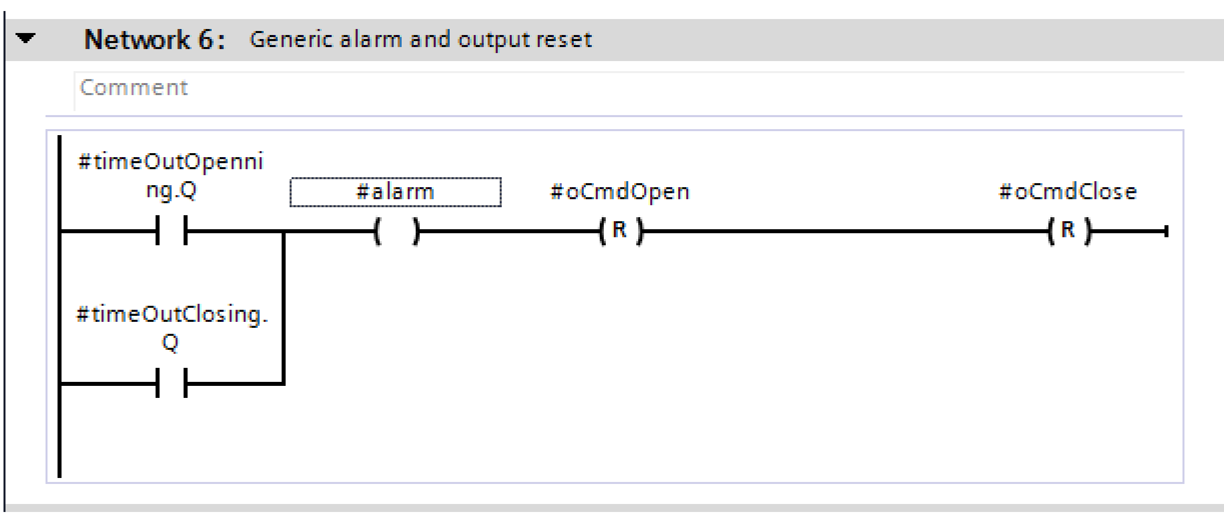


Fig. 51: Time outs: reset commands

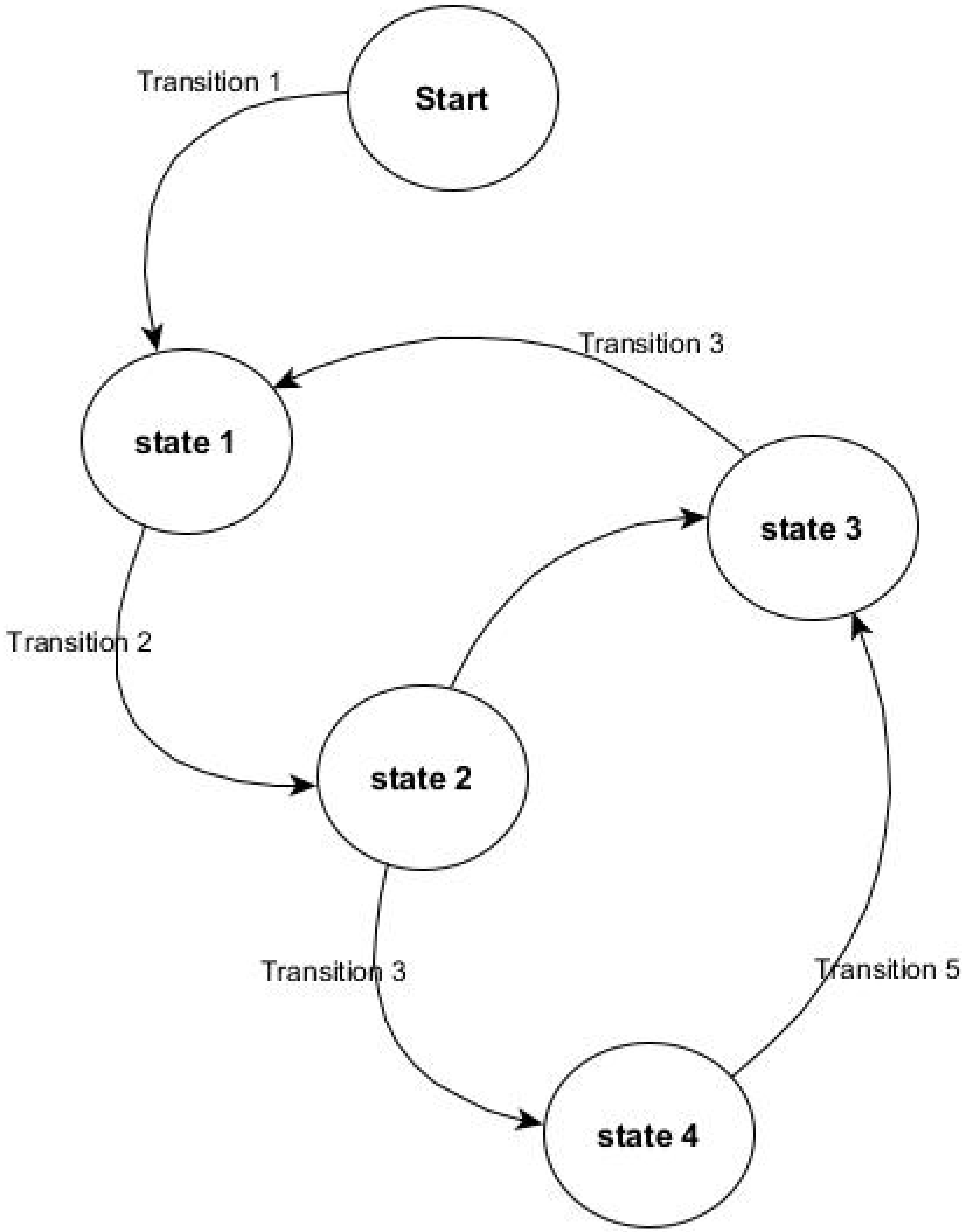


Fig. 52: State machine: States and Transitions

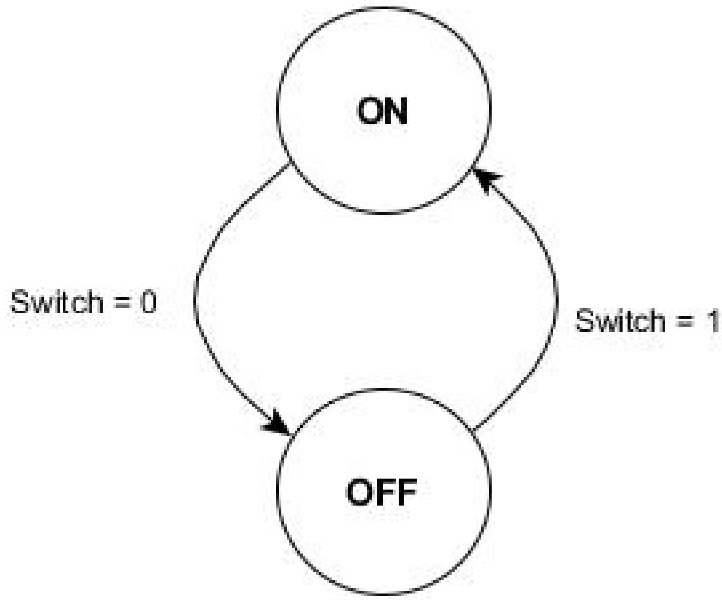


Fig. 53: Lamp states: ON or OFF

Now a simple lamp have three states. If we have a smart lamp (with internal diagnostic and MCU) the number of states may become more than three.

For example a pneumatic cylinder, can be opened or closed. It may also move in 2 directions, so it may have other 2 states, opening and closing. The cylinder may also be in a middle position, in our case we consider it as unknown position, it is in an alarm state.

The diagram show the states and transition from one state to another. As we can see, the cylinder can't go from `closed` to `opened` directly. To the `alarm` state we can arrive from any state.

We can make a transition from opening to closing directly. Suppose I was opening, but before to open completely I change idea, and want o close. But usually this is not the case when dealing, for example with a gripper that need to hold or leave an object. Anyway, depending on the application, transitions from one state to another can be considered or not.

2.8.2 Implementation

Siemens doesn't implement the `enumeration` data type. For better readability we emulate the enumeration data types by creating CONSTANTS with the name of the state.

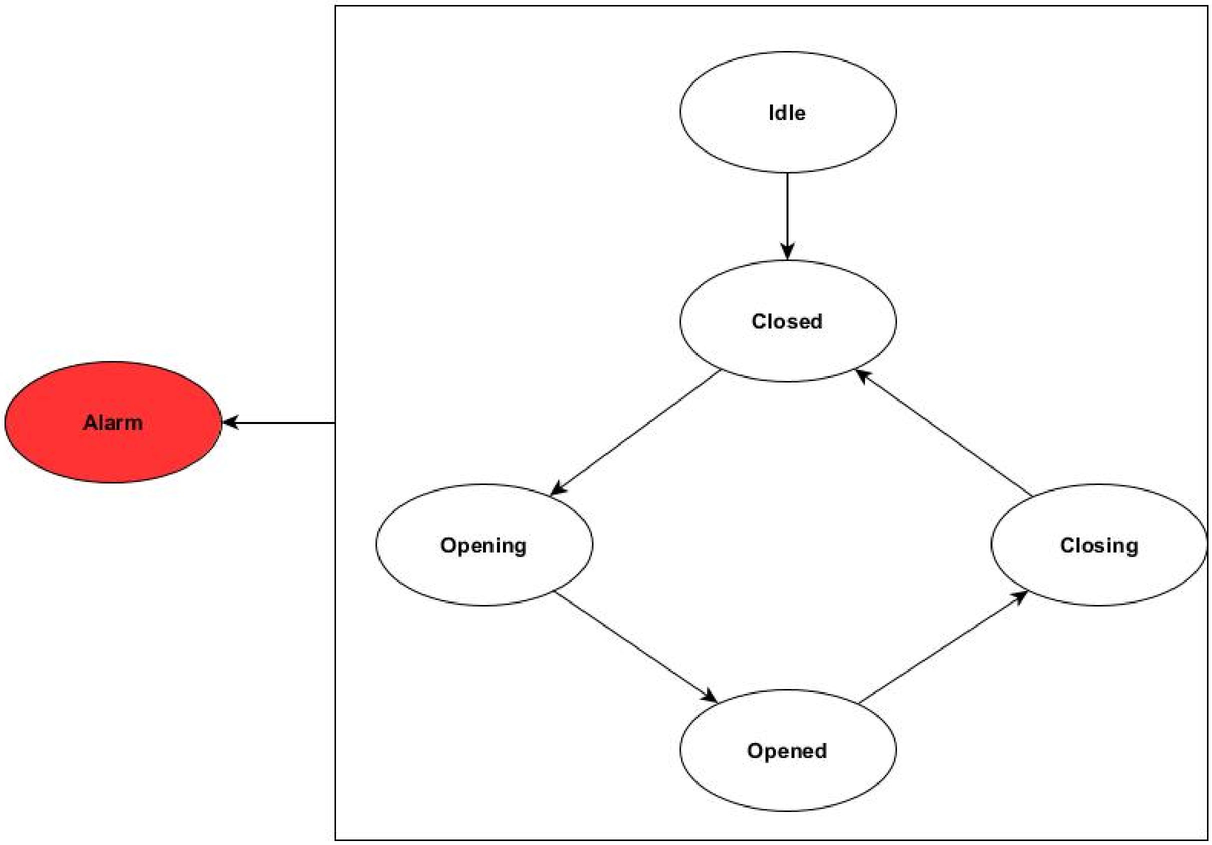
Compare the following two implementations:

Both implementations are valid and work. But one is more clear than the other, especially during debugging.

Every state should have a unique number. In the implementation the CONSTANTS variable will be used instead of its numeric value. Technically we don't care about the numeric value. It is enough that it is unique.

Implementation in ST

A code snippet is shown in this section, a complete and tested solution will be in the Library documentation. Note anyway that this version of code is already functional.



	Name	Data type	Default value	Retain	Accessible f...	Writa...	Visible in ...	Setpoint	Comment
1	Input				☐	☐	☐	☐	
2	iOpened	Bool	false	Non-retain	☑	☑	☑	☐	sensor cylinder opened
3	iClosed	Bool	false	Non-retain	☑	☑	☑	☐	sensor cylinder closed
4	iReqOpen	Bool	false	Non-retain	☑	☑	☑	☐	Open request (from HMI, or push button, or Robot,.....)
5	iReqClose	Bool	false	Non-retain	☑	☑	☑	☐	Close request (from HMI, or push button, or Robot,.....)
6	iCondOk	Bool	false	Non-retain	☑	☑	☑	☐	Condition ok (Emergency, Air, doors,....)
7	iTimeOpen	Time	T#10s	Non-retain	☑	☑	☑	☐	Time for openning time-out
8	iTimeClose	Time	T#10s	Non-retain	☑	☑	☑	☐	Time for closing time-out
9	Output				☐	☐	☐	☐	
10	oCmdOpen	Bool	false	Non-retain	☑	☑	☑	☐	output valve : Command close cylinder
11	oCmdClose	Bool	false	Non-retain	☑	☑	☑	☐	output valve : Command open cylinder
12	oActState	Int	0	Non-retain	☑	☑	☑	☐	Acutal state
13	oPrevState	Int	0	Non-retain	☑	☑	☑	☐	Previous stae
14	InOut				☐	☐	☐	☐	
15	<Add new>				☐	☐	☐	☐	
16	Static				☐	☐	☐	☐	
17	timeOutOpenning	TON_TIME		Non-retain	☑	☑	☑	☐	openning timer
18	timeOutClosing	TON_TIME		Non-retain	☑	☑	☑	☐	closing timer
19	Temp				☐	☐	☐	☐	
20	<Add new>				☐	☐	☐	☐	
21	Constant				☐	☐	☐	☐	
22	sIDLE	Int	1		☐	☐	☐	☐	Idle State
23	sALARM	Int	2		☐	☐	☐	☐	
24	sREADY	Int	10		☐	☐	☐	☐	
25	sCLOSED	Int	20		☐	☐	☐	☐	
26	sOPENING	Int	30		☐	☐	☐	☐	openning: during motion
27	sOPENED	Int	40		☐	☐	☐	☐	openned: already opened
28	sCLOSING	Int	50		☐	☐	☐	☐	

Fig. 54: State declared as CONSTANT variables with unique number or identifier. This interface is valid for implementation in SCL and in Ladder.

```
CASE #oActState OF
    #sIDLE:
        #oCmdClose := FALSE;
        #oCmdOpen := FALSE;
        IF #iOpened THEN
            #oPrevState := #oActState;
            #oActState := #sOPENED;
        ELSIF #iClosed THEN
            #oPrevState := #oActState;
            #oActState := #sCLOSED;
        ELSE
            #oPrevState := #oActState;
            #oActState := #sALARM;
        END_IF;
    #sALARM:
        IF #iReqClose THEN
            #oPrevState := #oActState;
            #oActState := #sCLOSING;
        ELSIF #iReqOpen THEN
            #oPrevState := #oActState;
            #oActState := #sOPENING;
        END_IF;

    #sCLOSED:
        IF #iReqOpen THEN
            #oPrevState := #oActState;
            #oActState := #sOPENING;
        END_IF;
    sOPENING:
        #oCmdOpen:=TRUE;
        #oCmdClose := FALSE;
        IF #iOpened THEN
            #oPrevState := #oActState;
            #oActState := #sOPENED;
```

Fig. 55: States are represented by CONSTANT variables.

```
CASE #oActState OF
    1:
        #oCmdClose := FALSE;
        #oCmdOpen := FALSE;
        IF #iOpened THEN
            #oPrevState := #oActState;
            #oActState := #sOPENED;
        ELSIF #iClosed THEN
            #oPrevState := #oActState;
            #oActState := #sCLOSED;
        ELSE
            #oPrevState := #oActState;
            #oActState := #sALARM;
        END_IF;
    5:
        IF #iReqClose THEN
            #oPrevState := #oActState;
            #oActState := #sCLOSING;
        ELSIF #iReqOpen THEN
            #oPrevState := #oActState;
            #oActState := #sOPENING;
        END_IF;

    10:
        IF #iReqOpen THEN
            #oPrevState := #oActState;
            #oActState := #sOPENING;
        END_IF;
    20:
        #oCmdOpen := TRUE;
        #oCmdClose := FALSE;
        IF #iOpened THEN
            #oPrevState := #oActState;
            #oActState := #sOPENED;
```

Fig. 56: States are represented by numeric value. A number by it self doesn't have any meaning.

State machine can be implemented using and if statement. But a `Switch-Case` statement is more suitable and more readable than an if statement.

For example when the cylinder is closed, it is in the closed state. So the variable `oActState` have the numeric value store in the constant `sCLOSED`. Using a `CASE` statement we can assign the logic depending on that state. For example, if the cylinder is in `sCLOSED` and receive the signal to open, a transition to the opening `sOPENING` state should be done

```
#sCLOSED:
  IF #iReqOpen THEN
      #oPrevState := #oActState;
      #oActState := #sOPENING;
  END_IF;
```

The previous code snippet change the value of `oActState` to `sOPENING` if the `iReqOpen` is true. So now the cylinder is in the opening state, where the cylinder should begin to move, so a command to the valve should be send

```
sOPENING:
  #oCmdOpen:=TRUE;
  #oCmdClose := FALSE;
  IF #iOpened THEN
      #oPrevState := #oActState;
      #oActState := #sOPENED;
  END_IF;
```

The cylinder begin to move, the output `oCmdOpen` to the valve is true. The cylinder still in this state until the signal `iOpened` became true.

A complete code snippet is shown here:

```
// Cylinder state machine
// best way to implement a state machine is using CASE statement
//
// Not complete

#timeOutOpenning(IN:= (#oActState = #sOPENING),
                 PT:=#iTimeOpen);

#timeOutClosing(IN:=#oActState = #sCLOSING,
                PT:=#iTimeClose);

IF #timeOutClosing.Q OR #timeOutOpenning.Q OR #iCondOk=FALSE THEN
    #oActState := #sALARM;
END_IF;

CASE #oActState OF
    #sIDLE:
        #oCmdClose := FALSE;
        #oCmdOpen := FALSE;
        IF #iOpened THEN
            #oPrevState := #oActState;
            #oActState := #sOPENED;
        ELSIF #iClosed THEN
            #oPrevState := #oActState;
            #oActState := #sCLOSED;
        ELSE
            #oPrevState := #oActState;
            #oActState := #sALARM;
```

(continues on next page)

(continued from previous page)

```
        END_IF;
    #sALARM:
        IF #iReqClose THEN
            #oPrevState := #oActState;
            #oActState := #sCLOSING;
        ELSIF #iReqOpen THEN
            #oPrevState := #oActState;
            #oActState := #sOPENING;
        END_IF;

    #sCLOSED:
        IF #iReqOpen THEN
            #oPrevState := #oActState;
            #oActState := #sOPENING;
        END_IF;
    #sOPENING:
        #oCmdOpen:=TRUE;
        #oCmdClose := FALSE;
        IF #iOpened THEN
            #oPrevState := #oActState;
            #oActState := #sOPENED;
        END_IF;
    #sOPENED:
        IF #iReqClose THEN
            #oPrevState := #oActState;
            #oActState := #sCLOSING;
        END_IF;
    #sCLOSING:
        #oCmdOpen:=FALSE;
        #oCmdClose := TRUE;
        IF #iClosed THEN
            #oPrevState := #oActState;
            #oActState := #sCLOSED;
        END_IF;
    ELSE  // Statement section ELSE
        #oPrevState := #sIDLE;
        #oActState:= #sIDLE;
END_CASE;
```

Time out are added for diagnostic purposes. When the cylinder still in the opening or closing state for more than the necessary time, the cylinder go to alarm state.

Of course the cylinder may stay in opened or closed state for indefinite time.

As you note, there is more code to write than the normal solution presented in the exercises chapter. Depending on the device we are controlling, the use of state machines may make the solution more or less complicated, but anyway more readable and easy to debug.

`Download FB cylinder in ST`

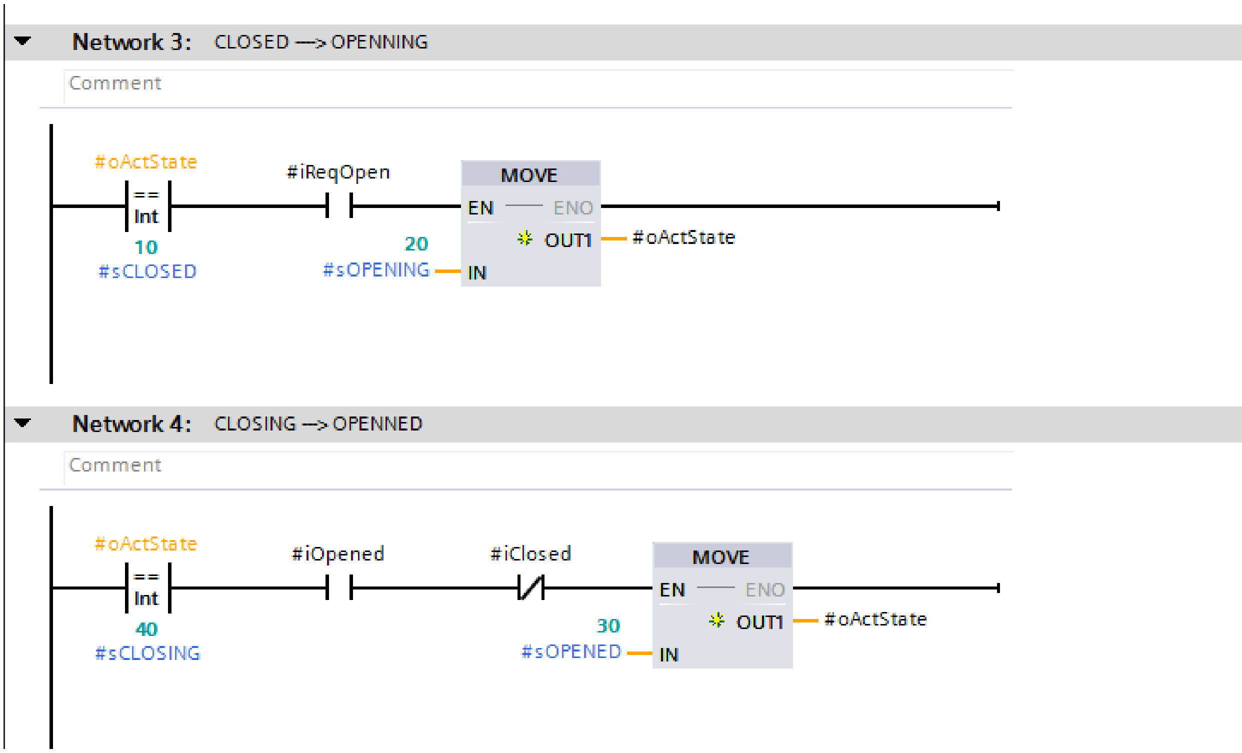
Implementation in Ladder

State machines are better implemented in textual language (ST, C, C++, etc.). Can be also implemented in Ladder Diagram, its implementation is slightly different.

Good implementation

As in ST every state is represented by a **unique number**. The implementation is divided in 2 stages:

- Transition from old state to new state
- Output assignment



Bad implementation

This implementation is absolutely to be a avoided. You will encounter a lot of implementations similar to it, without comment neither state names.

2.9 More exercises

2.9.1 State machine version of alternative motion

2.9.2 Access coordination

Write a program that control and manager the access of two robots to the same working station. Robot L put a part on the table (Load), Robot U take away the part from the table (UNload). On the table there is a sensor that check the presence of the part. The sensor is normally closed (No part or free=1, part present =0).

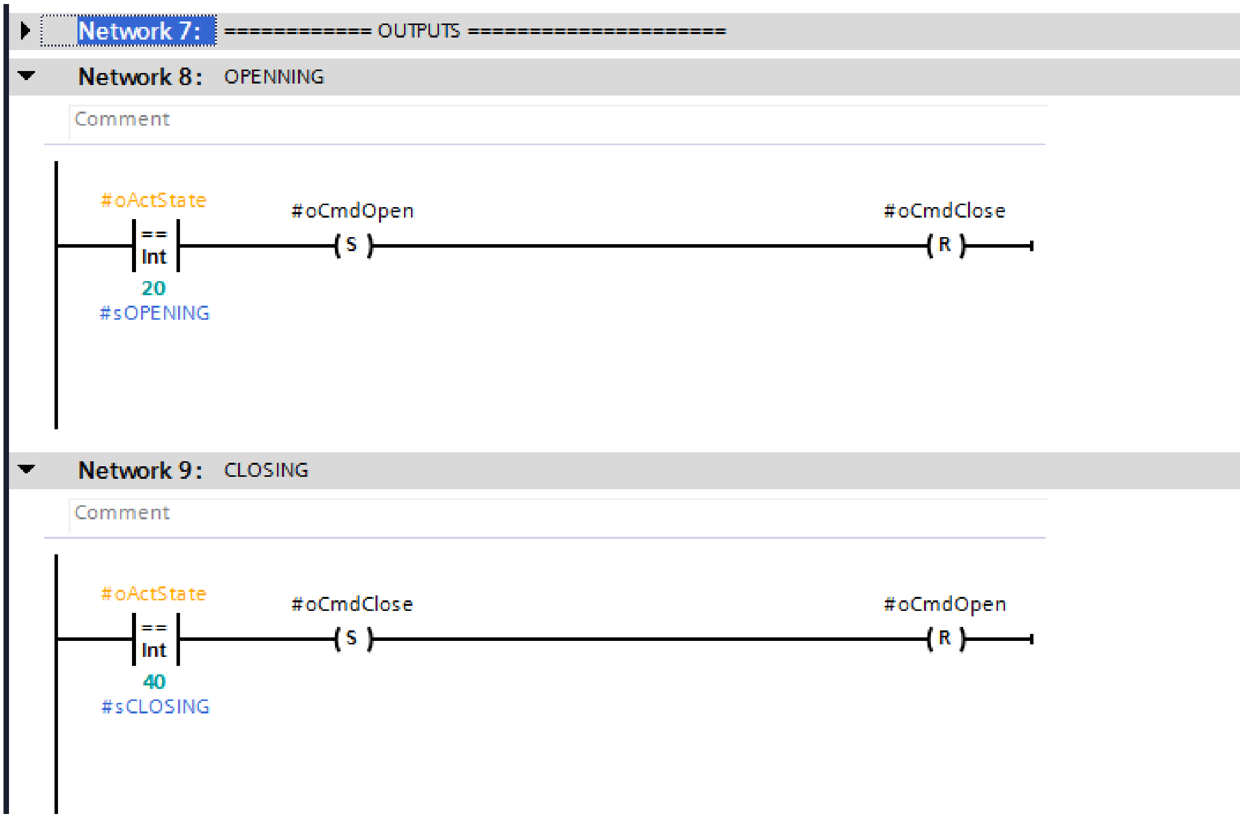


Fig. 57: Note that every state is represented by a boolean variable. The worst thing is that there is no comment neither a good variable name.

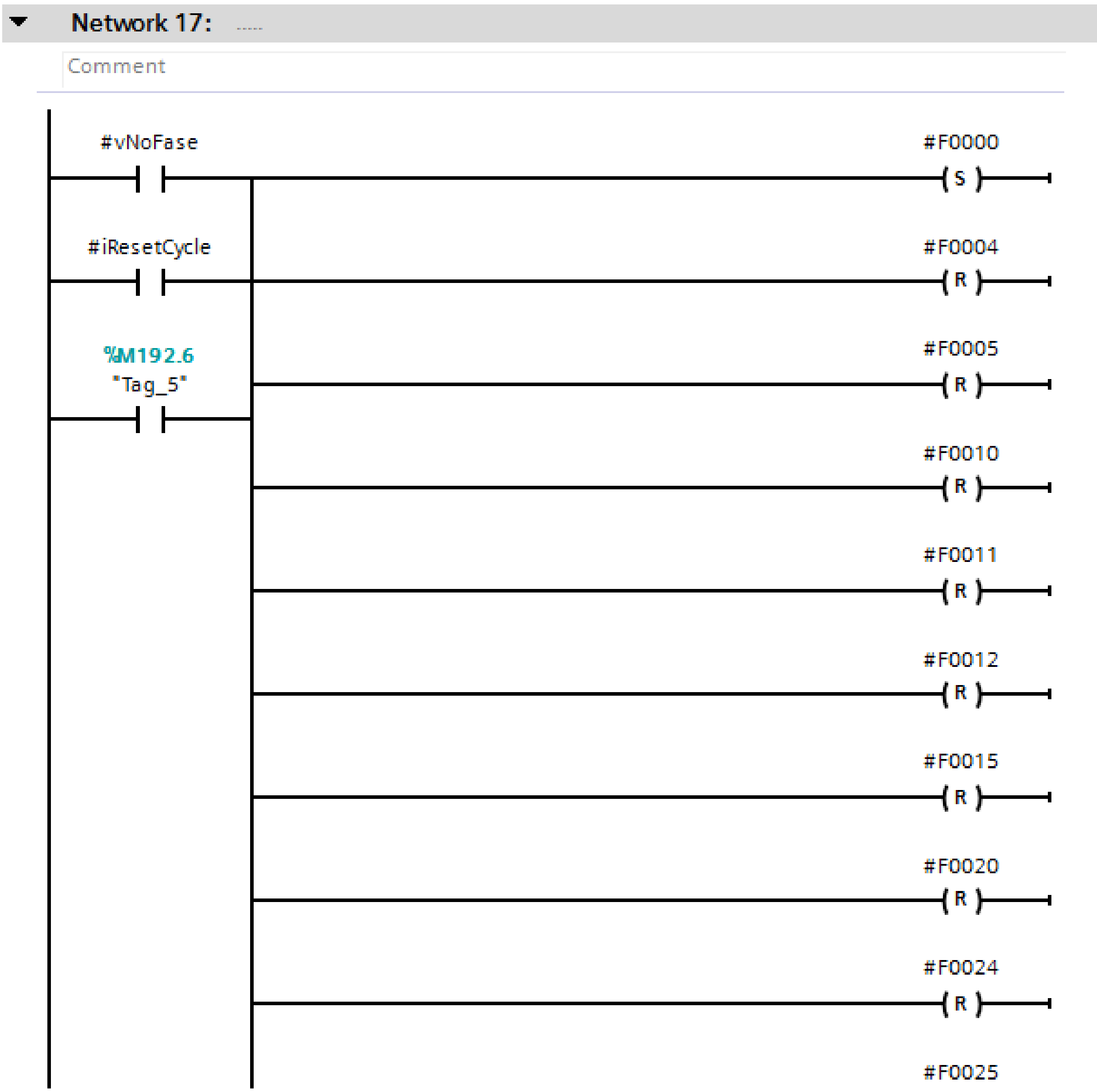


Fig. 58: During initialization a need to reset a lot of variables. If you forgot to reset some variable?

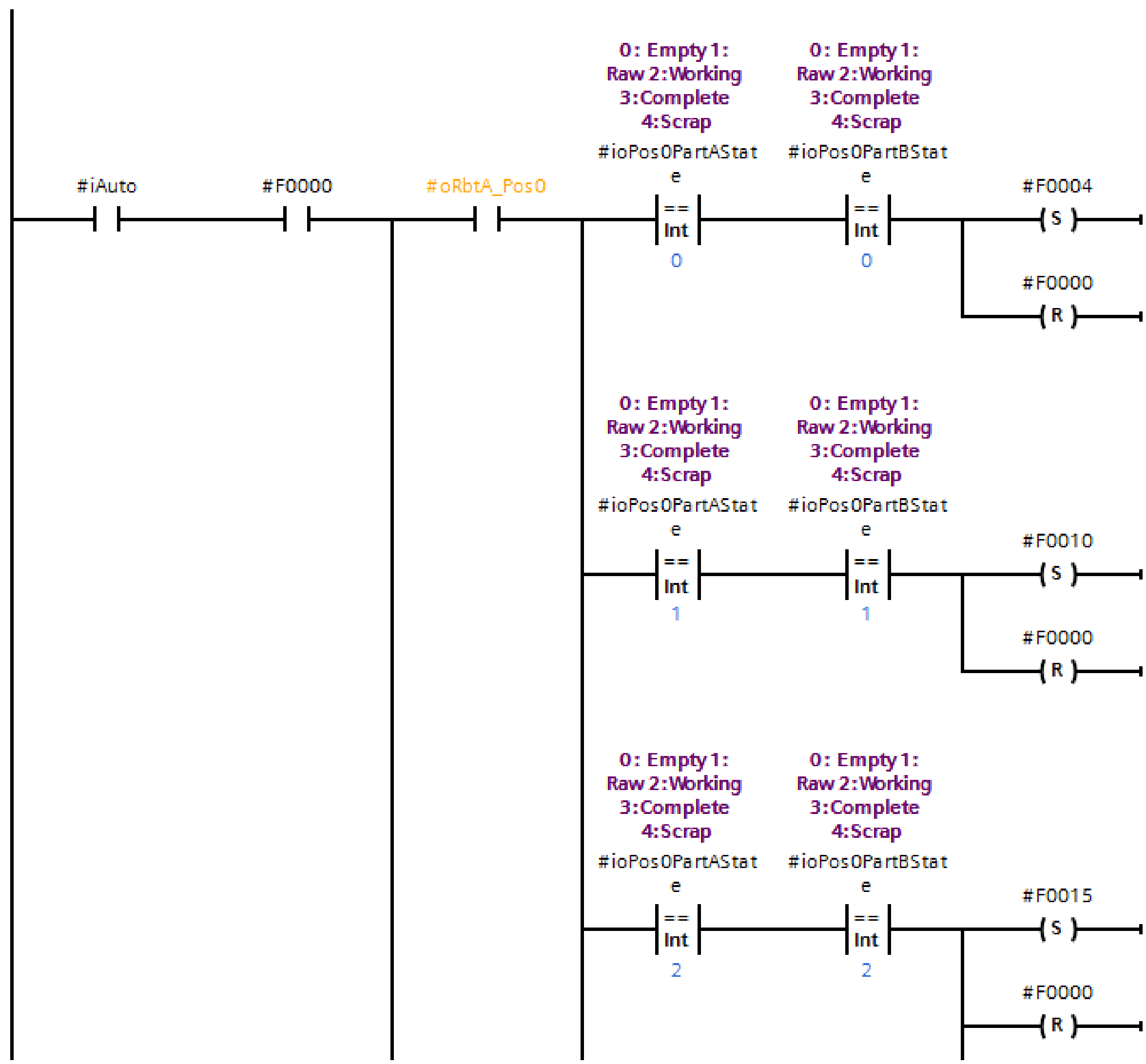


Fig. 59: At every transition to a new state you need to reset the old state.

2.9.3 Unloading conveyor

2.9.4 Vision system conveyor

2.9.5 Turn table

2.10 More exercises solutions

Note: Complete and tested solution can be found in the OpenLib Library

2.10.1 State machine version of alternative motion

2.10.2 Access coordination

2.10.3 Unloading conveyor

2.10.4 Vision system conveyor

2.10.5 Turn table

2.11 Create a library

Fig. 60: Create a new library

Fig. 61: Add function block (FB) and function (FC) to the library

Fig. 62: Modify a function and update the global library

Tia portal create a local copy of the functions from the `global library`. The functions are related to the `Project library`. When a function is modified is modified in the `Project library`. When modification is complete the `global library` can be updated from the `Project library`

A function can be separated from the library. Notice that the small triangle on the top right of the function icon disappear when the connection to the library is canceled.

`Download Library`

2.12 Simple project

`Download Exercises solutions`

The layout of this project is shown in the following image:

Fig. 63: Open an existing library

The process flow should be clear, the robot take a raw part from the turn table and put it in the CNC machine, in the loading position L. Then take the machined part from the unloading position U, and put it on the exit conveyor. The cycle continue in this way. The external position of the table is loaded by a person.

In the previous exercise we already write the function blocks for the conveyor and turn table, feel free to modify the logic if necessary, if you didn't consider some situation before. The goal of this project is to show how to organize the software.

the layout represent a `cell`. In this cell we can identify three `stations`: Turntable, machine and conveyor. Stations normally are independent from each others. For example the conveyor don't care neither need to know anything about the turntable neither the machine, and vice versa. The robot is the only connection between all stations.

From this point of view we can write the logic of every station independently from other stations. This is the approach taken also when writing the logic of very big production lines: break it down and you will see that a very complicated production line will be easy to implement. Every station have it sown defined job.

At this point we have one cell, three station and one robot. If the PLC control more than one cell, every cell should have its own folder. In this project, we have only one cell and one PLC. It is optional to create a cell folder. Keep in mind always future integration, so never limit your software to the current situation.

Three folders are created for every station and one folder for the robot. another folder can be created for general management of the cell.

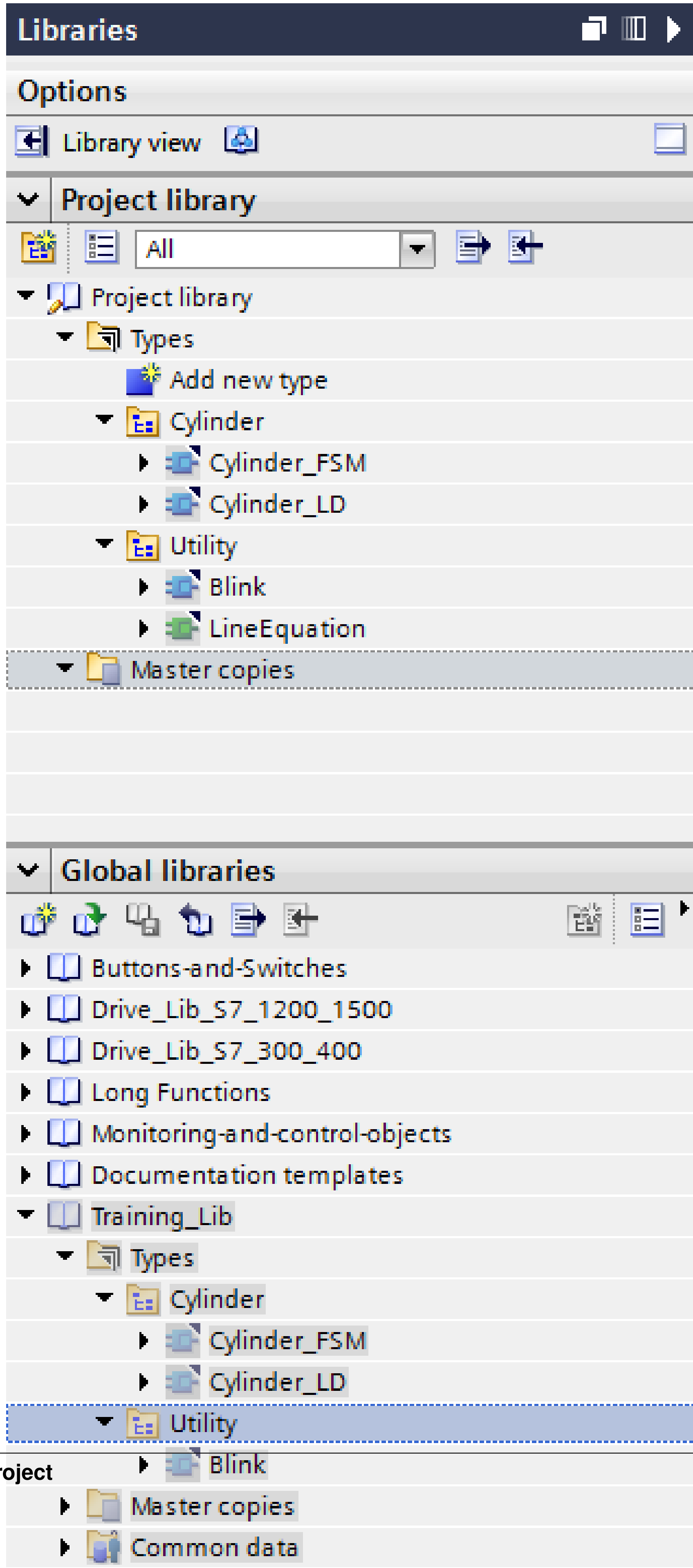
Every station has it own main function and main global data block and instance data block.

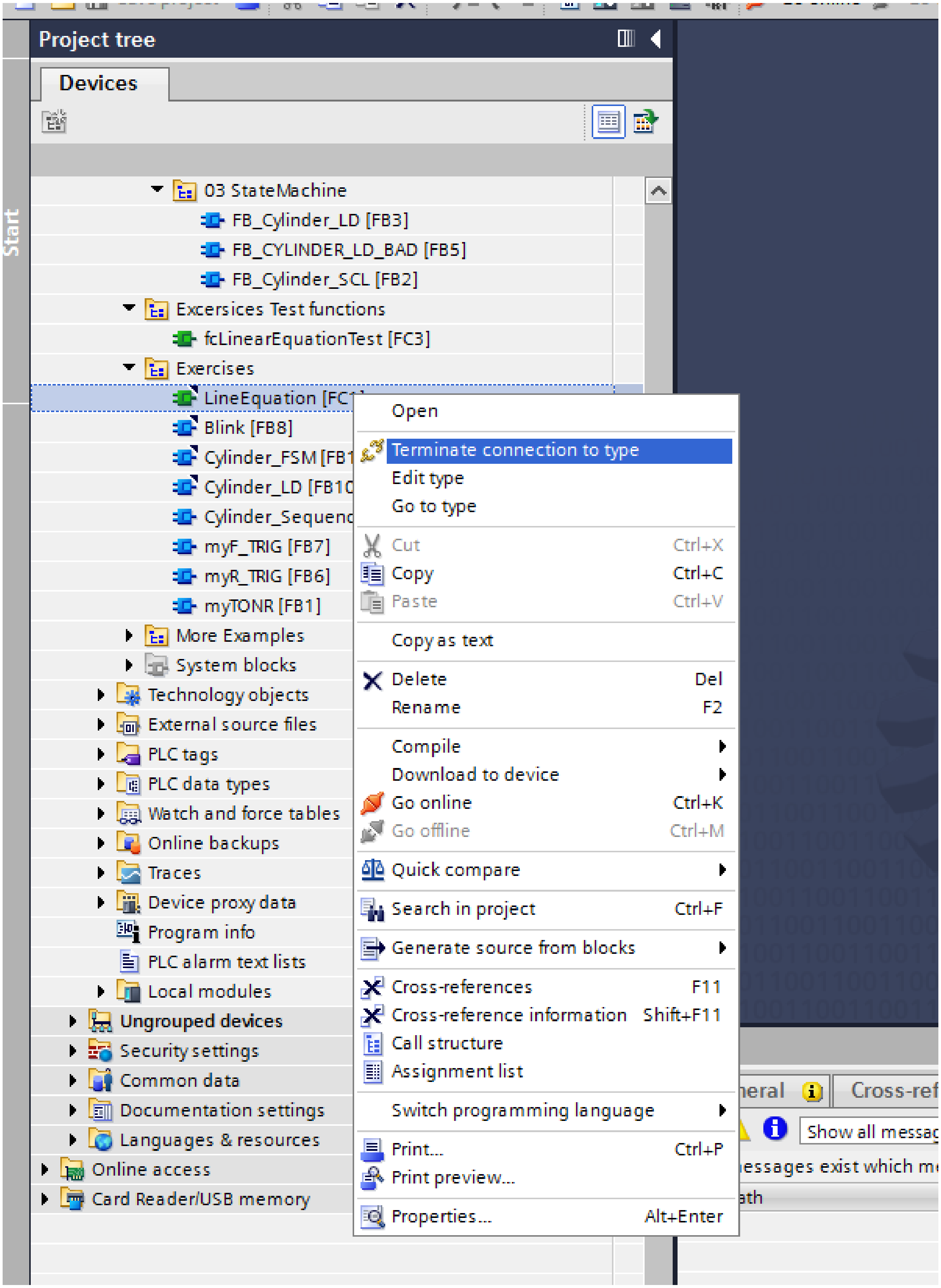
A folder called `_Library` is created, where general functions will be placed. In the previous chapter we see how to create an S7 library. We will add that library to the project and use conveyor and turntable FBs from it.

OB1 must contain only function calls. In the following image we can see the call to other functions. It is better to use SCL to call other functions because it is more compact. Notice the name of the function calls in SCL and in Ladder.

The main function blocks of the station, should not have input neither outputs. The call should appear on one line.

In the following sections we will examine every station. The developlent will be done without caring too much about where the physical IO are connected. As we think always in local variables at the beginning, we don't care where `I` and `Q` are assigned. Connecting those IO will be done when the logic is completed. Of course we need to know which IO we have in order to avoid to invent our own project. Electrical drawing should be always consulted.


Libraries
Options
Library view
Project library
All
Project library
Types
Add new type
Cylinder
Cylinder_FSM
Cylinder_LD
Utility
Blink
LineEquation
Master copies
Global libraries
Buttons-and-Switches
Drive_Lib_S7_1200_1500
Drive_Lib_S7_300_400
Long Functions
Monitoring-and-control-objects
Documentation templates
Training_Lib
Types
Cylinder
Cylinder_FSM
Cylinder_LD
Utility
Blink
Master copies
Common data


Project tree
Devices
03 StateMachine
FB_Cylinder_LD [FB3]
FB_CYLINDER_LD_BAD [FB5]
FB_Cylinder_SCL [FB2]
Excersices Test functions
fcLinearEquationTest [FC3]
Exercises
LineEquation [FC1]
Blink [FB8]
myF_TRIG [FB7]
myR_TRIG [FB6]
myTONR [FB1]
More Examples
System blocks
Technology objects
External source files
PLC tags
PLC data types
Watch and force tables
Online backups
Traces
Device proxy data
Program info
PLC alarm text lists
Local modules
Ungrouped devices
Security settings
Common data
Documentation settings
Languages & resources
Online access
Card Reader/USB memory
Open
Terminate connection to type
Edit type
Go to type
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Copy as text
Delete Del
Rename F2
Compile
Download to device
Go online Ctrl+K
Go offline Ctrl+M
Quick compare
Search in project Ctrl+F
Generate source from blocks
Cross-references F11
Cross-reference information Shift+F11
Call structure
Assignment list
Switch programming language
Print... Ctrl+P
Print preview...
Properties... Alt+Enter
Start

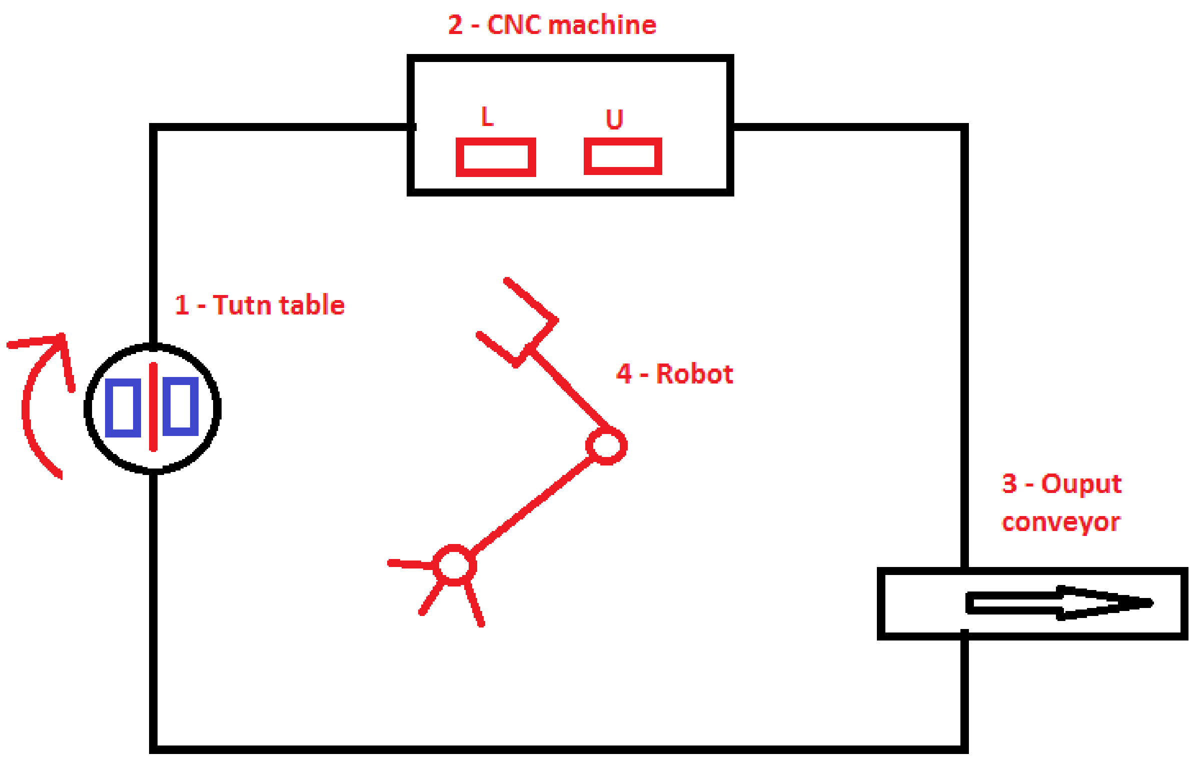
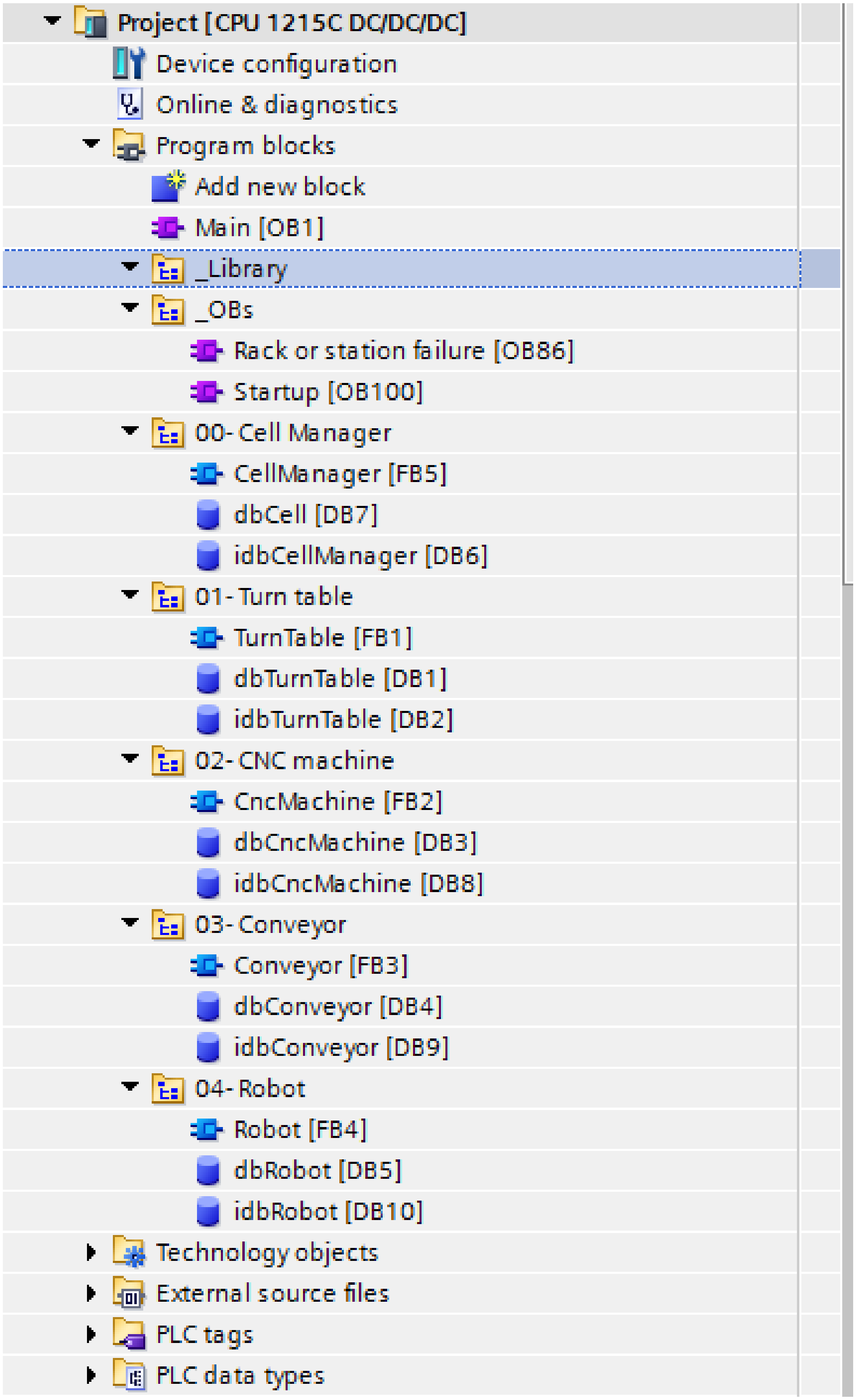
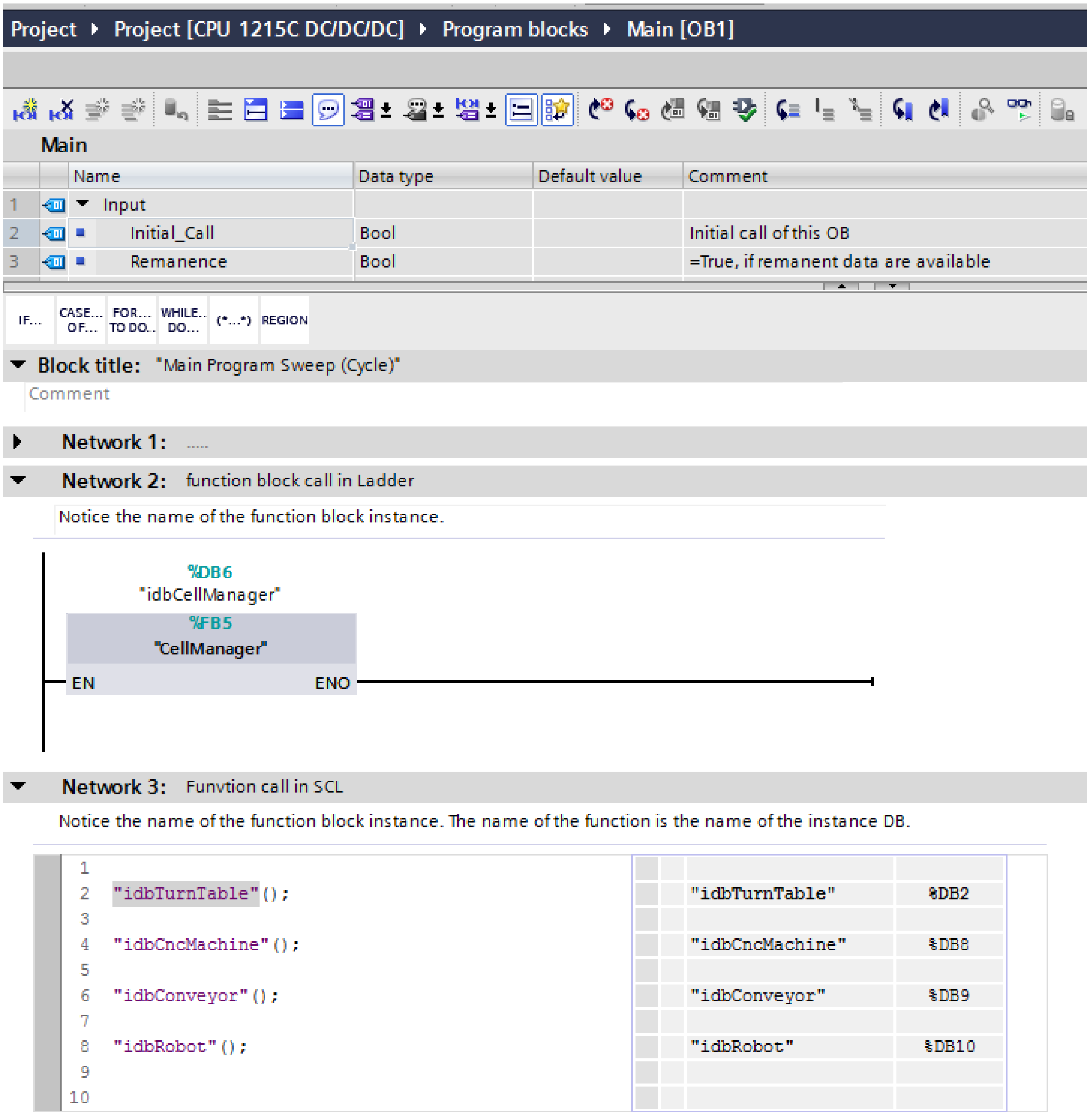


Fig. 65: Turn table (1) with 2 positions, CNC machine (2) with 2 position for loading and unloading, exit conveyor (3) and industrial robot (4)


Project [CPU 1215C DC/DC/DC]
Device configuration
Online & diagnostics
Program blocks
Add new block
Main [OB1]
_Library
_OBs
Rack or station failure [OB86]
Startup [OB100]
00- Cell Manager
CellManager [FB5]
dbCell [DB7]
idbCellManager [DB6]
01- Turn table
TurnTable [FB1]
dbTurnTable [DB1]
idbTurnTable [DB2]
02- CNC machine
CncMachine [FB2]
dbCncMachine [DB3]
idbCncMachine [DB8]
03- Conveyor
Conveyor [FB3]
dbConveyor [DB4]
idbConveyor [DB9]
04- Robot
Robot [FB4]
dbRobot [DB5]
idbRobot [DB10]
Technology objects
External source files
PLC tags
PLC data types


Project ▸ Project [CPU 1215C DC/DC/DC] ▸ Program blocks ▸ Main [OB1]
Main
Name | Data type | Default value | Comment
1 Input
2 Initial_Call | Bool | | Initial call of this OB
3 Remanence | Bool | | =True, if remanent data are available
IF... CASE... OF... FOR... TO DO... WHILE.. DO... (*...*) REGION
Block title: "Main Program Sweep (Cycle)"
Comment
Network 1: .....
Network 2: function block call in Ladder
Notice the name of the function block instance.
%DB6
"idbCellManager"
%FB5
"CellManager"
EN
ENO
Network 3: Funvtion call in SCL
Notice the name of the function block instance. The name of the function is the name of the instance DB.
"idbTurnTable"();
"idbCncMachine"();
"idbConveyor"();
"idbRobot"();
"idbTurnTable" %DB2
"idbCncMachine" %DB8
"idbConveyor" %DB9
"idbRobot" %DB10

2.12.1 Turn table

2.12.2 CNC machine

2.12.3 Exit conveyor

2.12.4 Robot

2.13 Complete project

2.13.1 Layout and process flow

2.13.2 Electrical Drawing

2.13.3 IO tags from electrical drawing

2.13.4 PLC-Robot Interface

2.13.5 Program structure

CHAPTER 3

CoDeSys

Warning: Work in progress

3.1 IEC 61131-3

3.2 CoDeSys

CHAPTER 4

S7 Library

OpenLib documentation TIA Portal v15

4.1 Operating mode

4.2 Utility

4.2.1 Linear equation

4.2.2 Swapping

4.3 Drives and inverters

4.4 Actuators

4.4.1 Bi-stable cylinder

4.5 Conveyors

4.5.1 Unloading conveyor

4.5.2 Vision system conveyor

4.6 Turn tables

4.6.1 Turn table

Note: Knowledge is Power
