Chapter IV : PLC Programming

Chapter IV: Programming a PLC

In this chapter, we will first provide an overview of the different programming languages used for PLCs. Then, we will briefly introduce the basic concepts of Ladder programming and explain how to translate a Grafcet into contact language. Additionally, we will define linear and structured programming under SIMATIC S7 and present the memory zones and variables supported by the SIMATIC S7-1200. Through various examples provided at the end of this chapter, we will learn to program Grafcets in Ladder language, use IEC timers, and convert analog signals into numerical values. An appendix is included at the end of the chapter to describe in detail bitwise combinatorial operations.

1. PLC Programming Languages

The **IEC (International Electrotechnical Commission)** officially recognizes five PLC programming languages as defined in the **IEC 61131-3** standard. These languages are categorized into graphical and textual languages:

Graphical Languages:

- **LD**: Ladder Diagram (Contact Schematics)
- **FBD**: Function Block Diagram (Logic Diagrams)
- **SFC**: Sequential Function Chart (Grafcet)

Textual Languages:

- **IL**: Instruction List
- **ST**: Structured Text

Each PLC programming language has its own advantages and disadvantages depending on the task. The vast majority of industrial automation projects utilize a single language to accomplish the required task.

1.1. Ladder Diagram (LD)

This is the most popular PLC programming language, designed to replace wired relay control systems. Ladder languages have limitations, as you can only use predefined blocks. However, you can program the majority of control systems using Ladder diagrams alone.

1.2. Function Block Diagram (FBD)

The second most popular PLC programming language is the Function Block Diagram. In this language, program blocks are interconnected to create a complete program. Many commands used in Ladder logic are also used in FBD, but it is often easier to read and conceptualize.

1.3. Sequential Function Chart (SFC)

The SFC concept is simple: an action block remains active until the transition step below it is triggered. The transition step contains all the conditions that must be fulfilled for the next block to activate. For projects with repeatable steps or tasks that can be divided into smaller units, SFC is the easiest language to implement.

1.4. Instruction List (IL)

This language consists of multiple lines of code, with a single instruction per line. It is read from top to bottom and left to right. The instruction list is very straightforward, as each line is executed sequentially. Once you learn the mnemonics (e.g., Load = LD, Start = ST), it becomes an excellent language for creating compact and efficient code tailored to application needs.

1.5. Structured Text (ST)

Structured Text closely resembles BASIC or C programming. It is ideal for control systems requiring mathematical computations or complex tasks. Trigonometry, calculations, and data analysis can be implemented much more easily in this language than in Ladder diagrams.

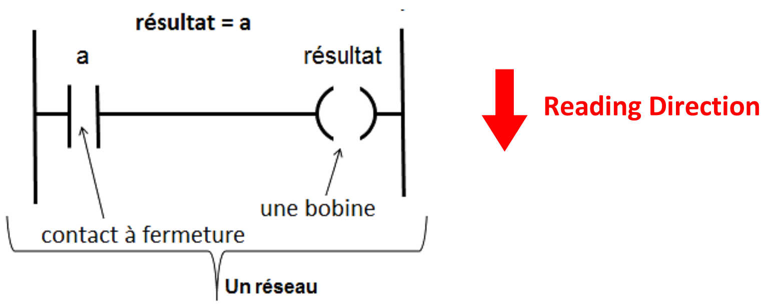
Note: In the following sections, we will focus exclusively on Ladder language for Siemens PLCs from the SIMATIC S7-1200 and 1500 series.

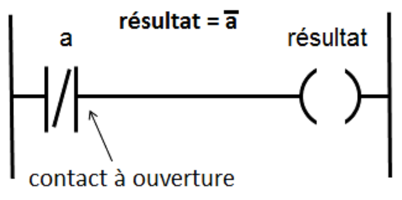
2. Ladder Language in SIMATIC S7-1200

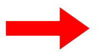
This language is well-suited for describing combinatorial functions, creating function blocks, or programming Grafcets. Its name, "Ladder," comes from the resemblance of its programs to a ladder. Ladder programs are read from top to bottom, with value evaluation occurring from left to right. They consist of **networks** where all the left-hand contacts must be true for the output coils to be energized. Each input signal is represented by a contact (switch) that is either normally closed or normally open.

Note: There can only be one coil per network.

Example:







2.1. Basic Combinatorial Operations in Ladder Language

Combinatorial operations on bits utilize the binary system. For contacts and coils, '1' indicates activated or energized, while '0' indicates deactivated or de-energized.

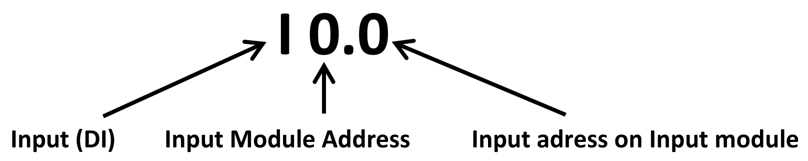
Bitwise combinatorial operations evaluate the signal states (1 and 0) and combine them based on Boolean logic. The result of these combinations is either 1 or 0, referred to as the **Logical Result (LR)**. Below, we will illustrate some logical equations in Ladder language.

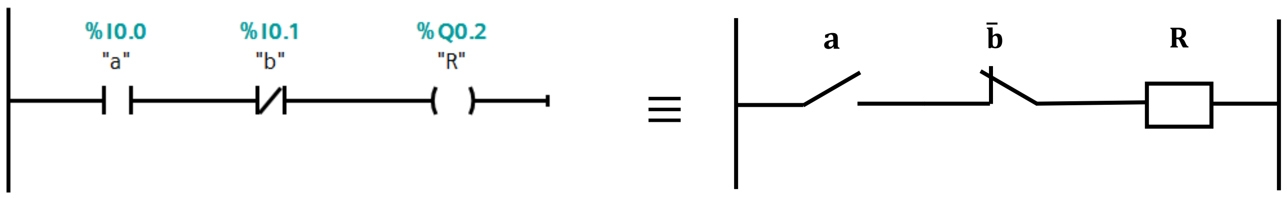
- ### Logical 'AND' Function

Let **I0.0** and **I0.1** be two discrete inputs (DI) on a PLC input module, represented by the Boolean variables **'a'** and **'b'**, respectively.

Let **Q0.2** be a discrete output (DO), represented (renamed) by the Boolean variable **'R'**.

The equation **Q = a·b** can be expressed in Ladder language through the following graphical representation:





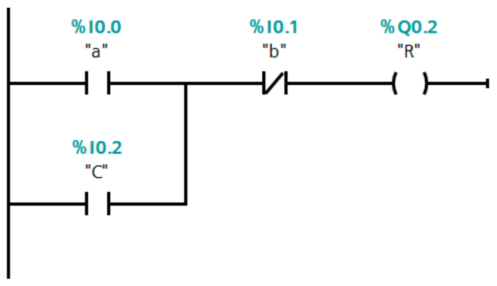
$$\boxed{R = a \cdot \bar{b}}$$

The output Q0.2 is high (coil energized) if and only if I0.0 is high and I0.1 is low. In other words, R = 1 if a = 1 and b = 0.

- ### Logical 'OR' Function

Let **I0.0**, **I0.1**, and **I0.2** be three inputs of a PLC, represented by the Boolean variables **a**, **b**, and **c**, respectively. The output **Q0.2** is denoted by the variable **R**.

The equation **R = (a + c) · $\bar{\mathbf{b}}$** is expressed in Ladder language through the following graphical representation:



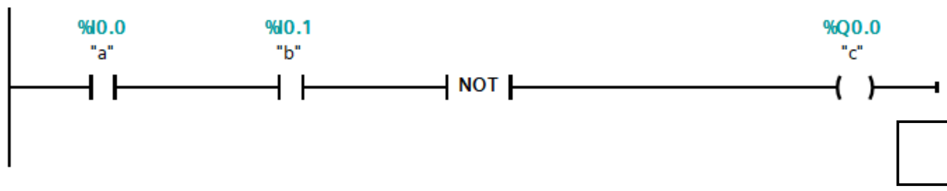
$$\boxed{\mathbf{R} = (\mathbf{a} + \mathbf{c}) \cdot \bar{\mathbf{b}}}$$

The output Q0.0 is active if input I0.0 is high, or if I0.2 is high and I0.1 is low.

- **Logical Negation**

Let **I0.0** and **I0.1** be two inputs of a PLC, represented by the Boolean variables **a** and **b**, respectively, and an output **Q0.0**, renamed **c**.

The equation $\mathbf{c} = \overline{\mathbf{a} \cdot \mathbf{b}}$ is expressed in Ladder language through the following graphical representation:



$$\boxed{\mathbf{c} = \overline{\mathbf{a} \cdot \mathbf{b}}}$$

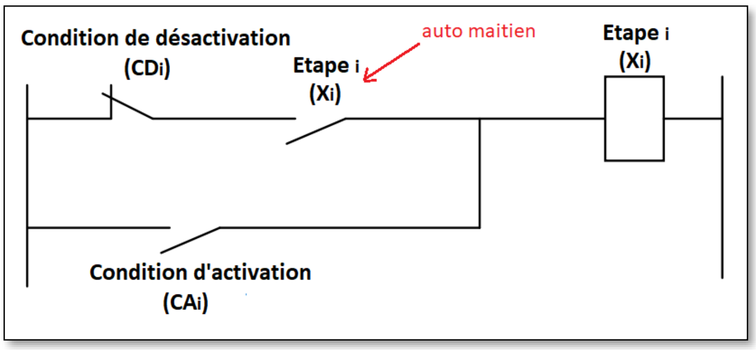
3. Conversion from GRAFCET to Ladder Language (LADDER)

In certain cases, it can be useful to translate a GRAFCET into LADDER language. GRAFCET steps can be viewed as memory functions, each having an activation condition (AC) and a deactivation condition (DC).

- **Activation Condition (AC):** A step is activated if the immediately preceding step is active AND the associated transition condition is satisfied.
- **Deactivation Condition (DC):** A step is deactivated if the activation condition of the next step is validated.

The activation and deactivation equations for each step take the following form:

$$\mathbf{X_i} = \overline{\mathbf{CD_i}} \cdot \mathbf{X_i} + \mathbf{CA_i} \qquad \mathbf{(4.1)}$$



To fully understand the operation of logical equation (4.1), we have established its truth table.

N°		x_i (Actuel)	CD_i	CA_i	x_i (Future)	Observation
0	Désactivée	0	0	0	0	State of x_i maintained
1		0	0	1	1	Activation of x_i : 0 → 1
2		0	1	0	0	State of x_i maintained
3		0	1	1	1	Activation of x_i : 0 → 1 (dominant activation condition)
4	Active	1	0	0	1	State of x_i maintained
5		1	0	1	1	State of x_i maintained
6		1	1	0	0	Desactivation of x_i : 1 → 0
7		1	1	1	1	State of x_i maintained (Active)

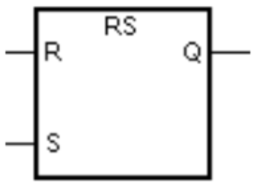
Table 4.1. Truth Table for the Activation/Deactivation Equation of Steps

- When step x_i is desactivated and both activation and desactivation conditions are simultaneously true ($CD_i = 1$ et $CA_i = 1$) step x_i becomes active ($x_i: 0 \rightarrow 1$).
- When step $\boldsymbol{x_i}$ is active and both activation and deactivation conditions are simultaneously true ($\boldsymbol{CD_i = 1}$ et $\boldsymbol{CA_i = 1}$) step $\boldsymbol{x_i}$ maintains its active state ($\boldsymbol{x_i: 1 \rightarrow 1}$). This confirms the crossing rule number 3 (refer to the Grafcet course).

Rule 3: When a step must be simultaneously activated and deactivated, it remains active.

Truth Table of the SR Flip-Flop

Q_n (Actuel)	R	S	Q_{n+1} (Future)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	-
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	-



Symbol of the SR Flip-Flop

The output equation of the RS flip-flop can be determined using the Karnaugh map.

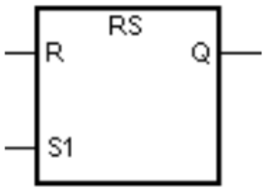
$$Q_{i+1} = \overline{R} \cdot Q_i + S \qquad (4.2)$$

Note: If both inputs are activated simultaneously (S=R=1S = R = 1), this can cause an undefined or unstable state in the output.

The activation and deactivation equation of each step (4.1) is similar to the output equation of the SR flip-flop (4.2). The only difference arises during the simultaneous activation of both inputs (R=S=1R = S = 1). To make the two equations identical, the SR flip-flop can be replaced with an RS1 flip-flop, which forces the flip-flop's output to 1 when both inputs are active simultaneously.

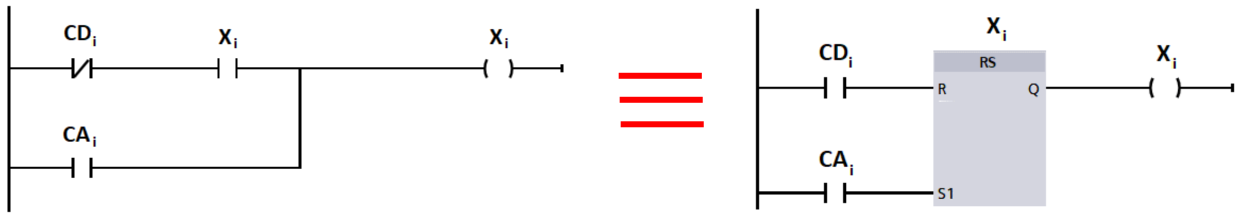
Truth Table of the RS1 Flip-Flop

Q_n (Actuel)	R	S	Q_{n+1} (Future)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



Truth Table of the RS1 Flip-Flop

Representation of a Step Using an RS1 Flip-Flop



4. Description of Bitwise Combinatorial Operations

- **Normally Open Contact**

Normally Open Contact	Operand type	Description	Memory Area
<opérande> ---| |---	Bool	The contact is closed if the operand value equals 1, allowing current to flow through it. If the operand value equals 0, the contact is open, and no current flows through it.	I , Q , M , D , L

- Normally Closed Contact

Normally Closed Contact	Operand type	Description	Memory Area
<opérande> ---| / |---	Bool	The contact is closed if the operand value equals 0, allowing current to flow through it. If the operand value equals 1, the contact is open, and no current flows through it.	**I , Q , M , D , L**

- Assignment

Assignment	Operand type	Description	Memory Area
<opérande> ---( )	Bool	This operation functions like a coil in a relay diagram. If energy reaches the coil, the operand is set to 1. If energy does not reach the coil, the operand is set to 0.	**I , Q , M , D , L**

- Negation of Assignment

Assignment	Operand type	Description	Memory Area
<opérande> ---(/)	Bool	This operation inverts the logical result and assigns it to the specified operand. When the coil input equals "1," the operand is set to 0. When the coil input equals "0," the operand is set to the logical state "1."	**I , Q , M , D , L**

- Invert RLO

Assignment	Operand type	Description	Memory Area
---|NOT|---	Bool	**Invert the Logical State** When the logical state "1" is present at the input of the instruction, its output provides the logical state "0." If the logical state is "0" at the input of the instruction, its output provides the logical state "1."	**I , Q , M , D , L**

- Setting and Resetting a Bit

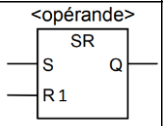
Setting a Bit

Assignment	Operand type	Description	Memory Area
<opérande> **---(S)**	Bool	**Activate Output** When S (Set) is activated, the operand value is set to 1. When S is not activated, the operand remains unchanged.	**I , Q , M , D , L**

Resetting a Bit

Assignment	Operand type	Description	Memory Area
<opérande> **---(R)**	Bool	**Deactivate Output** When R (Reset) is activated, the operand value is set to 0. When R is not activated, the operand remains unchanged.	**I , Q , M , D , L**

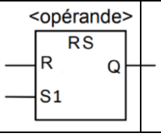
- Bascule 'mise à 1/mise à 0'

Assignment	Operand type	Description	Memory Area
	Bool	**Flip-Flops with Set/Reset** The SR operation is a flip-flop with reset priority, where the reset dominates. If both the set (S) and reset (R1) signals are true, the operand value will be 0.	**I , Q , M , D , L**

Truth Table of the SR Flip-Flop

S	R1	Q_{n+1}	operand
1	0	1	1
0	1	0	0
0	0	Q_n	Q_n
1	1	0	0

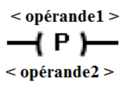
- Reset/Set Flip-Flop

Assignment	Operand type	Description	Memory Area
	Bool	**Flip-Flops with Reset/Set:** The RS operation is a flip-flop with set priority, where the set dominates. If both the set (S1) and reset (R) signals are true, the operand value will be 1.	**I , Q , M , D , L**

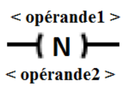
Truth Table of the RS Flip-Flop

S1	R	Q_{n+1}	operand
1	0	1	1
0	1	0	0
0	0	Q_n	Q_n
1	1	1	1

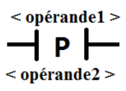
- Setting the Operand on Rising Edge of the Signal ---(P)---

Assignment	description	Memory Area
	**Setting the Operand on Rising Edge:** When a rising edge is detected, the instruction sets <operand1> to the logical state "1" for one program cycle. In all other cases, the operand provides the logical state "0." This instruction compares the current RLO (Result of Logical Operation) to the previous RLO stored in an edge memory (<operand2>).	operand1 : output operand2 : input **I , Q , M , D , L**

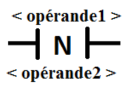
- Setting the Operand on Falling Edge of the Signal ---(N)---

Assignment	description	Memory Area
	**Setting the Operand on Falling Edge:** When a falling edge is detected, the instruction sets to the logical state "1" for one program cycle. In all other cases, the operand provides the logical state "0." This instruction compares the current RLO (Result of Logical Operation) to the previous RLO stored in an edge memory ().	operand1 : output operand2 : input **I , Q , M , D , L**

- Detecting Rising Edge of an Operand ---|P|---

Assignment	Description	Memory Area
	**Detecting Rising Edge of an Operand:** The instruction detects a change from "0" to "1" in the logical state of <operand1>. It compares the current logical state of <operand1> with the previous state, stored in an edge memory (<operand2>). When the instruction detects a change from "0" to "1," it identifies it as a rising edge. When a rising edge is detected, the instruction output provides the logical state "1." In all other cases, the output logical state is "0."	operand1 : Input operand2 : output **I , Q , M , D , L**

- Detecting Falling Edge of an Operand ---|N|---

Assignment	description	Memory Area
	**Detecting Falling Edge of an Operand:** The instruction detects a change from "1" to "0" in the logical state of . It compares the current logical state of with the previous state, stored in an edge memory (). When the instruction detects a change from "1" to "0," it identifies it as a falling edge. When a falling edge is detected, the instruction output provides the logical state "1." In all other cases, the output logical state is "0."	operand1 : output operand2 : input **I , Q , M , D , L**