CHAPTER 1

INTRODUCTION TO THE PLC PROGRAMMING AND THE IEC 61131-3 STANDARD

Many PLC programming standards have been suggested over the years. Suggestions have come from various national and international committees that had the goal of developing a common interface for programmable controllers. In 1979, an international working group of PLC experts who were tasked to come up with a first draft for a comprehensive PLC standard proposed a first draft.

After the document appeared in 1982, it was decided that the standard was too comprehensive to be collected into a single document. The original working group was therefore split up into five different working groups, each of which dealt with its portion of the standard.

The five parts consisted of:

1. General information
2. Hardware and requirements for testing
3. Programming languages
4. User interface
5. Communications

The first standard on programming languages (Part 3) was published in March 1993 and was designated IEC 61131-3. Other additions were published in 2002 and the third, and provisionally last, appeared in 2013. The standard, which is currently followed to a greater or lesser degree by most of the major PLC manufacturers, includes various programming languages:

1. Structured Text—ST
2. Function Block Diagram—FBD
3. Ladder Diagram—LD

4. Instruction List—IL
5. Sequential Function Chart—SFC

LD, SFC, and FBD are graphical programming languages, while IL and ST are text-based languages. It should be noted that the order in which they are listed above corresponds to the order in which they are described in the standard. When or how often the languages are used is irrelevant.

The textual language ST will be followed by the graphical languages LD and FBD. In addition, SFC will be thoroughly discussed since this language is especially designed for programming sequential controls and for organizing program code in general. This chapter contains a brief presentation of all the languages in the standard, and the four languages mentioned above will be thoroughly treated in their own chapters. The IL programming language will not be described extensively since is based upon an industry evaluation of which language is most efficient in use and which is the most widespread.

1 Number Systems

Many of the fundamental operations and signal-processing routines performed by a PLC require that the programmer have a fundamental knowledge of and understanding of binary numbers, logical (Boolean) quantities, and Boolean algebra. We can use logical functional expressions to describe instructions and actions and use Boolean algebra to simplify these for implementation. For more advanced programmers, there are also techniques and systematic presentations that can be used in order to structure both the tasks that the PLC is to solve and the operational mode of the program itself. In what follows our intention is to cover the necessary knowledge concerning these subjects.

A basic knowledge of different number systems and conversions among these systems is necessary in connection with programming PLCs. Which format the values and numbers are stored in and processed in varies between different PLCs, but most of them use one or more standard formats. These formats can be binary numbers (BCD coded), hexadecimal, and octal, in addition to decimal members. Here, we will discuss the formats that are used most frequently.

1.1 The Decimal Number Systems

A brief presentation of the decimal number system is useful to facilitate understanding of the structure of the other number systems. The decimal number system or the 10-base system, as it is also called, operates with the number 10 as the base number. Fundamentally, this means that this number system uses 10 different numerals to write all numbers. Furthermore, the base number is the factor between positions of the numerals.

Example 1. As is well known, a decimal number can be split up as shown below:

$$3647 = 3 \times 10^3 + 6 \times 10^2 + 4 \times 10^2 + 7 \times 10^0 = 3000 + 600 + 40 + 7$$

As we see, each different numeral has a defined weight depending upon its placement in the number (thousands, hundreds, tenths, units). △

1.2 The Binary Number System

All digital equipment such as computers is based on binary numbers. These are numbers that are built up out of only two different numerals, namely, 0 and 1. In digital equipment, the numeral 0 is represented by a voltage of 0 V, while the numeral 1 is represented by a constant non-zero voltage level (5 V, 12 V and so on...). The word binary comes from the fact that each numeral has two states. All binary numbers can be expressed by means of only the two numerals 0 and 1. The binary number system is also called the 2-base system. The base number in that number system is therefore 2.

Example 2. Any random binary number can be split up as shown below:

$$\begin{aligned} 11010101 &= 1\times 2^7 + 1\times 2^6 + 0\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 \\ &= 1\times 128 + 1\times 64 + 0\times 32 + 1\times 16 + 0\times 8 + 1\times 4 + 0\times 2 + 1\times 1 \\ &= 128 + 64 + 0 + 16 + 0 + 4 + 0 + 1 = 213 \end{aligned}$$

Here, each numeral has a particular weight depending upon its placement in the number and there is a factor of 2 (the base number) between each position in the number. △

1.3 The Hexadecimal Number System

Numbers that originally are in binary form are often displayed and transmitted in another form, namely, as hexadecimal numbers. The hexadecimal number system is also used in PLCs and has the base number of 16. This means that there are 16 different symbols that are used in this number system. These are

0, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **A**, **B**, **C**, **D**, **E**, **F**

As we see, the letters A through F are used instead of the numbers 10–15.

Example 3.

$$\mathbf{E3C8} = 14\times 16^3 + 3\times 16^2 + 12\times 16^2 + 8\times 16^0 = 58312 \text{ (decimal)}$$

There is a factor of 16 (the base number) between each position in the number. △

The hexadecimal number system is often used to represent binary numbers because the numbers can be expressed much more compactly. For example, a four-place hexadecimal number corresponds to a binary number of 16 bits (a word). Table 1.1 contains the numbers from 0 to 15 in decimal, binary, and hexadecimal form. The binary number 1111, which is 15 in decimal and **F** in hexadecimal, is the largest number that can be expressed with four bits. The standard register size1 in most PLCs is therefore 16 bits. The largest number that we can store in a register address is 1111 1111 1111 1111 which corresponds to **FFFF** in hexadecimal.

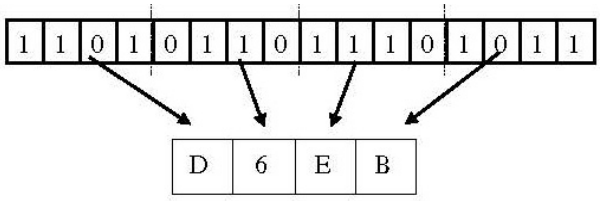
The corresponding decimal number is

$$\mathbf{FFFF} = 15\times 16^3 + 15\times 16^2 + 15\times 16^1 + 15\times 16^0 = 65535$$

Decimal	Binary	Hexadecimal
0	0000	**0**
1	0001	**1**
2	0010	**2**
3	0011	**3**
4	0100	**4**
5	0101	**5**
6	0110	**6**
7	0111	**7**
8	1000	**8**
9	1001	**9**
10	1010	**A**
11	1011	**B**
12	1100	**C**
13	1101	**D**
14	1110	**E**
15	1111	**F**

Table 1.1: Examples of decimal, binary, and hexadecimal numbers

If bipolar values, that is, values that could be either positive or negative, are to be stored, the Most Significant Bit (MSB) is used as a sign bit. In that case, values between -2^{15} and $2^{15} - 1$, that is, between -32768 and 32767, can be stored by using 16 bits. As mentioned previously, sometimes hexadecimal numbers are used as an alternative to the binary form when values are to be shown on the display/screen or when values are to be transmitted. That is to say that a 16-bit number is divided into groups of four numerals. Each of the groups is represented by a symbol between 0 and F.



1.4 Binary-Coded Decimal Numbers

Numbers in binary-coded decimal (BCD) form are not so different from numbers in binary form. The difference is that for each numeral (0 through 9) the decimal number is represented by four bits. This means that the numbers $0 - 9$ are precisely like the numbers $0 - 9$ in binary form. If we study a four-bit number, however, we have six bit combinations that are not valid in BCD form. These are the combinations for the hexadecimal numbers $\mathbf{A} - \mathbf{F}$ (decimal $10 - 15$). The BCD system provides a convenient way of handling large numbers that need to be input to or output from a PLC. As you can see from looking at the various number systems, there is no easy way to go from binary to decimal and back. The BCD

system provides a means of converting a code readily handled by humans (decimal) to a code readily handled by the equipment (binary). PLC thumb wheel switches and LED displays are examples of PLC devices that make use of the BCD number system. The BCD system uses 4 bits to represent each decimal digit. The 4 bits used are the binary equivalents of the numbers from 0 to 9. In the BCD system. the largest decimal number that can be displayed by any four digits is 9. The BCD representation of a decimal number is obtained by replacing each decimal digit by its BCD equivalent. To distinguish the BCD numbering system from a binary system, a BCD designation is placed to the right of the units digit. A thumb wheel switch is one example of an input device that uses BCD. Figure 1.1 shows a single-digit BCD thumb wheel. The circuit board attached to the thumb wheel has one connection for each bit's weight plus a common connection. The operator dials in a decimal digit between 0 and 9, and the thumb wheel switch outputs the equivalent 4 bits of BCD data. In this example, the number eight is dialed to produce the input bit pattern of 1000. A four-digit thumb wheel switch, similar to the one shown, would control a total of 16 (4 x 4) PLC inputs.

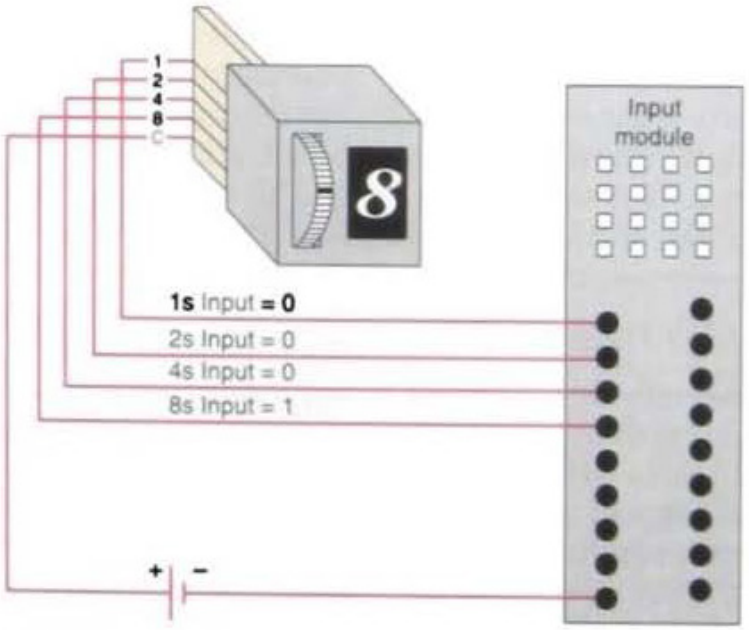


Figure 1.1: BCD Thumbwheel example

Example 4. The decimal number 6923 will be stored in BCD form as 0110 1001 0010 0011. Note that this is entirely different from the binary notation. In binary form, the same number would be represented as 0001 1011 0000 1011 (try the conversion yourself) while the hexadecimal number would be **1B0B**. △

What are the consequences of this? First, it limits how large a number we can operate within 16-bit registers. The largest number in BCD form would then be 9999. However, the advantage is that in calculation operations with larger integers, the content of two adjacent addresses, each address holding a four-digit number, can be considered as a single number, that is, an eight-place number. For example, we can then perform the multiplication

99999999 × 99999999. The result of this operation is 9.99×10^15, and the result will occupy four addresses or 64 bits. In most cases, such large numbers will be more than sufficient.

In summary, all digital equipment such as a PLC performs calculations and stores values in binary form. However, it is not particularly user-friendly to work with and read binary numbers. Therefore, most digital equipment operates with several different number systems where one of them is, of course, the decimal number system. All PLCs of recent vintage use the decimal number system in the user interface. In the next Figure you can compare all the previously described numerical systems

NUMERIC VALUES IN DECIMAL, BINARY, BCD, AND HEXADECIMAL REPRESENTATION

Decimal	Binary	BCD	Hexadecimal
0	0	0000	0
1	1	0001	1
2	10	0010	2
3	11	0011	3
4	100	0100	4
5	101	0101	5
6	110	0110	6
7	111	0111	7
8	1000	1000	8
9	1001	1001	9
10	1010	0001 0000	A
11	1011	0001 0001	B
12	1100	0001 0010	C
13	1101	0001 0011	D
14	1110	0001 0100	E
15	1111	0001 0101	F
16	1 0000	0001 0110	10
17	1 0001	0001 0111	11
18	1 0010	0001 1000	12
19	1 0011	0001 1001	13
20	1 0100	0010 0000	14
126	111 1110	0001 0010 0110	7E
127	111 1111	0001 0010 0111	7F
128	1000 0000	0001 0010 1000	80
510	1 1111 1110	0101 0001 0000	1FE
511	1 1111 1111	0101 0001 0001	1FF
512	10 0000 0000	0101 0001 0010	200

Figure 1.2: Comparison between Decimal, Binary, BCD and Hexadecimal number systems

2 Digital Logic

In order to be able to program PLCs more efficiently and to reduce the risk of ambiguous program algorithms, it is an advantage to master digital logic. Most often, outputs and internal variables are controlled by a combination of states of other variables and input signals. A little example could be to start a pump when a start switch is activated but only if the level in the tank is lower than a certain value.

In this section, we will therefore present the basic logical functions and how calculations can be performed in a computer or PLC based on Boolean algebra.

There are three basic logical functions: AND, OR, and NOT. In addition, there is a function frequently used called Exclusive Or (XOR) along with combinations of AND, OR, and NOT.

These basic functions are shown below along with their symbols, functional expressions, and functional tables (truth tables). The symbols are used in drawing and designing digital circuits and in the graphical programming language in the Function Block Diagram (FBD) standard.

The functional expressions are used when the algorithms for combinatorial controls are to be described with text. The functional tables, or truth tables as they are also called, describe the structure of the logical functions. For AND and OR, the operation is also illustrated with simple circuit diagrams consisting of a switch, a lamp, and a battery. In the following, functions are presented with two inputs but there can often be three or more.

All of these basic logical functions are found as electronic components in the form of integrated circuits (ICs). These can be used to construct simple controls when there is no need to be able to modify them in the future.5 Each IC can contain several ports of the same type. How many depends upon how many inputs each port has (Figures 1.3 and 1.4).

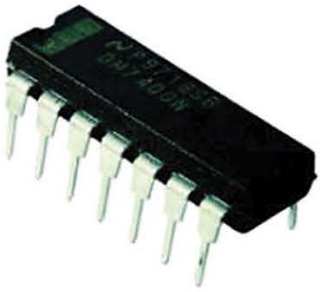


Figure 1.3: Integrated circuit (IC)

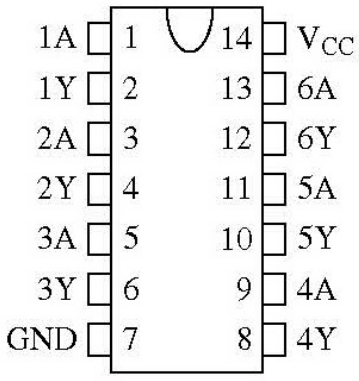


Figure 1.4: Pin configuration for SN7408N

2.1 AND

As the name indicates, AND functions in such a way that the output from the function is logically high (1) when both (all) inputs are logically high. Otherwise, the output is logically low (0). The functional expression reads thus: F equals A and B. The expression could also be read as $F = A\text{AND}B$, but it is common to use the times sign instead of AND. The AND symbol can be graphically represented according to the IEC standard (Figure 1.5) or the US standard (Figure 1.6) The truth table is

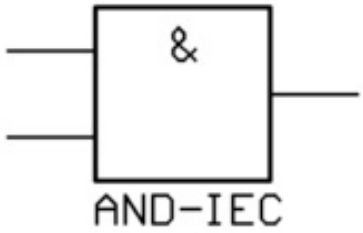


Figure 1.5: AND symbol (IEC standard)

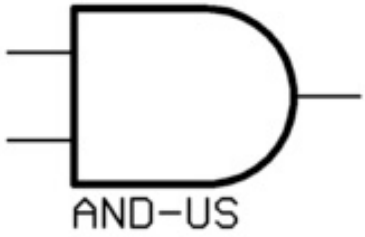


Figure 1.6: AND symbol (US standard)

A	B	$A \,\text{AND}\, B = A \cdot B$
0	0	0
1	0	0
0	1	0
1	1	1

Table 1.2: AND truth table

2.2 OR

The output of an OR is logically high when one or more of the inputs are logically high. The functional expression reads thus: $F = A \,\text{OR}\, B$. Note the use of the plus sign in the functional expression. The OR symbol can be graphically represented according to the IEC standard (Figure 1.7) or the US standard (Figure 1.8) The truth table is

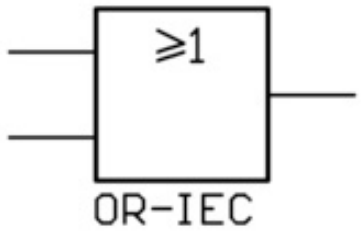


Figure 1.7: OR symbol (IEC standard)

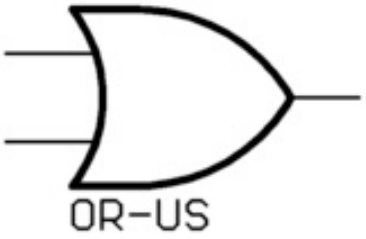


Figure 1.8: OR symbol (US standard)

A	B	$A \,\mathrm{OR}\, B = A + B$
0	0	0
1	0	1
0	1	1
1	1	1

Table 1.3: OR truth table

2.3 NOT

This is an inverter. The output variable state is the inversion (opposite) of the state of the input variable. We read: $F = \text{NOT}A$. Note the use of the inversion sign (bar symbol). The NOT symbol can be graphically represented according to the IEC standard (Figure 1.9) or the US standard (Figure 1.10) The truth table is

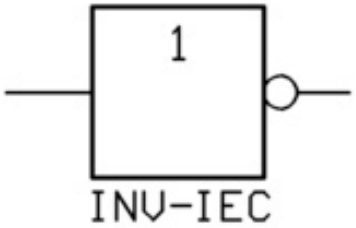


Figure 1.9: NOT symbol (IEC standard)

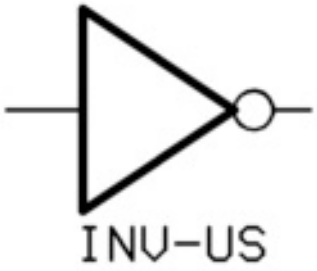


Figure 1.10: NOT symbol (US standard)

A	NOT $A = \overline{A}$
0	1
1	0

Table 1.4: NOT truth table

2.4 NAND

NAND is a combination of NOT and AND. The output from such a function is the inverse of the output from an AND. This means that the output is logically low when all inputs are logically high. Otherwise, the output is logically high. The NAND symbol can be graphically represented according to the IEC standard (Figure 1.11) or the US standard (Figure 1.12)

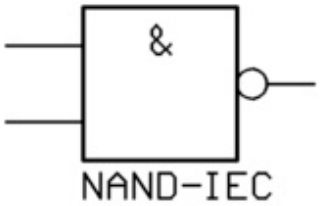


Figure 1.11: NAND symbol (IEC standard)

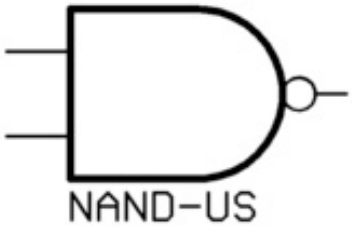


Figure 1.12: NAND symbol (US standard)

The truth table is

A	B	$A \,\mathrm{NAND}\, B = \overline{A \cdot B}$
0	0	1
1	0	1
0	1	1
1	1	0

Table 1.5: NAND truth table

2.5 NOR

A combination of NOT and OR. As the functional expression shows, the output variable is equal to the inverse of the output from an OR. The NOR symbol can be graphically represented according to the IEC standard (Figure 1.13) or the US standard (Figure 1.14)

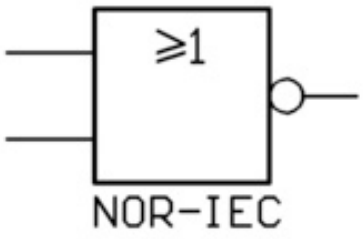


Figure 1.13: NOR symbol (IEC standard)

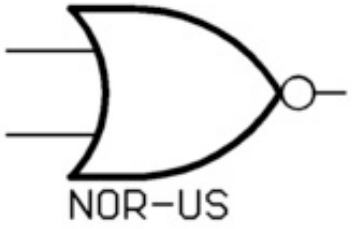


Figure 1.14: NOR symbol (US standard)

The truth table is

A	B	$A \,\mathrm{NOR}\, B = \overline{A+B}$
0	0	1
1	0	0
0	1	0
1	1	0

Table 1.6: NOR truth table

2.6 XOR

This is a special variant of OR that is called XOR. In contrast to an ordinary OR, the output here is logically high when only one of the inputs is logically high. The XOR symbol can be graphically represented according to the IEC standard (Figure 1.15) or the US standard (Figure 1.16) The truth table is

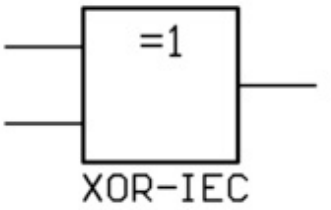


Figure 1.15: XOR symbol (IEC standard)

Figure 1.16: XOR symbol (US standard)

A	B	$A \,\mathrm{XOR}\, B = A \oplus B$
0	0	0
1	0	1
0	1	1
1	1	0

Table 1.7: XOR truth table

With these basic building blocks, one can construct many useful (and useless) circuits such as those shown in the following examples. Note that there are many types of ready-made digital circuits on the market: from the simple types that are presented here to advanced circuits such as arithmetic units, memory circuits, converters, and microprocessors. It is entirely practical to build anything at all based on so-called off-the-shelf components. For larger hardware-based controls, such as those for a washing machine, for example, it is more usual to employ programmable circuits (EPROM, PAL, etc.) or microprocessors.

Example 5. The circuit in Figure is an adder that adds two single-bit numbers. The result of the addition appears as the two bits F_1 and F_2, where F_1 is the Least Significant Bit (LSB). In order to understand the operation of the circuit, it is easiest to set up a function table for

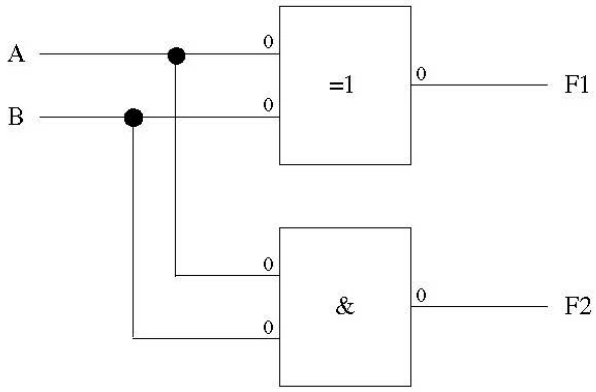


Figure 1.17: One bit adder

B	A	F_2	F_1	Decimal sum
0	0	0	0	$0+0=0$
0	1	0	1	$0+1=0$
1	0	0	1	$1+0=1$
1	1	1	1	$1+1=2$

the circuit (the last column shows the sum in decimal form): In this example, the circuit was given beforehand, something that unfortunately seldom happens in practice. Most often, one takes the desired function or operation as a starting point and then designs the circuit or codes from that. △

Example 6. In a chemical processing facility, liquid chemicals are used in production. The chemicals are stored in three different buffer tanks. A level sensor located in each tank gives a logical high signal when the level in the tank in question falls below a lower limit. We will design a digital circuit that provides a logical high alarm signal when the level in at least two of the tanks gets too low. If we call the three level sensors A, B, and C and the alarm signal from the circuit is F, we can set up a functional table for the circuit: It is actually not

A	B	C	F
0	0	0	0
1	0	0	0
0	1	0	0
0	0	1	0
1	1	0	1
1	0	1	1
0	1	1	1
1	1	1	1

necessary to set up such a table for this simple problem, but it can be nice for the sake of an overview. One possible implementation of a circuit for this control is shown in Figure 1.18. In the following section, we will see how this is developed. △

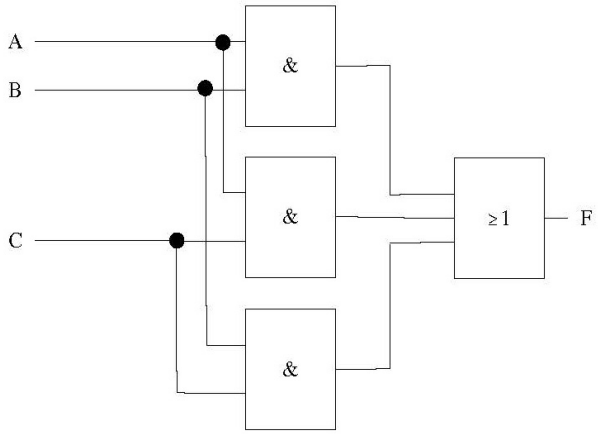


Figure 1.18: Chemical Plant Monitoring Logical Network

3 Boolean Design

3.1 Logical Functional Expressions

Logical functions are expressions that describe the behavior or the desired mode of operation of the combinatorial circuit. Such logical expressions can also be useful to describe control algorithms for instructions and actions in a PLC program. Functional expressions arise either directly from a descriptive problem statement or as a result of processing and simplification by means of Boolean algebra. Functional expressions describe for what combinations of input variables the output in question should be logically high.

Example 7. For the circuit in Example 6 it is a good idea to minimize the expression by thinking through a little logic. Based on the description of the operation of the control, we can analyze the problem as follows:

If two of the tanks are too low a level, that is, if for example signals A and B are logically high, the alarm should go off (F should be logically high).

Furthermore, we can say that because the alarm should go off no matter which two tanks have a low level, it is unimportant whether the third tank has a low level or not. This means that the last condition that should result in an alarm, the fact that all three level sensors are giving signals (A and B and C) can be neglected. The other conditions also cover this. Accordingly, we can set up the following functional expression for the alarm output F:

$$F = A \cdot B + A \cdot C + B \cdot C \tag{1.1}$$

Note that the expression for an active alarm in the example above is actually a minimized (simplified) functional expression. If we had taken the output point in the functional table alone, that is, without thinking practically about the process, we would have set up the following functional expression:

$$F = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C} + A \cdot B \cdot C \tag{1.2}$$

This method of expressing functions is called Sum of Products (SOP). If you compare the expression with the table in Example 6, you will see that for each combination of input signals

where F is logically high, we can write an AND expression for the combination. After that, we just add up (OR operation) all the expressions. △

The functional expression in eq. (1.2) and the simplified functional expression in eq. (1.1) are actually completely identical in the sense that they describe the same logical function. The difference shows up only when we are going to implement the function in the form of a digital circuit or a program code. This tells you right away that it is the minimized expression that we will prefer.

Now, it is not always easy to undertake such a minimization based on only an understanding of the process or on practical evaluations. Often, there is a requirement for a more methodical and mathematical approach. Such a minimization can in many cases be made simply based on knowledge of simple logical algebra, or Boolean algebra, as it is also called. This is discussed in the next section, but before we go on to that, we will study one more example:

Example 8. When you have a completed simplified functional expression, it is easy to design the actual circuits or to program code in Function Block Diagram (FBD). Assume that we have

$$F = A \cdot \bar{B} + \overline{\bar{A} \cdot C}$$

The circuit or FBD code for this function is then as shown in Figure 1.19. △

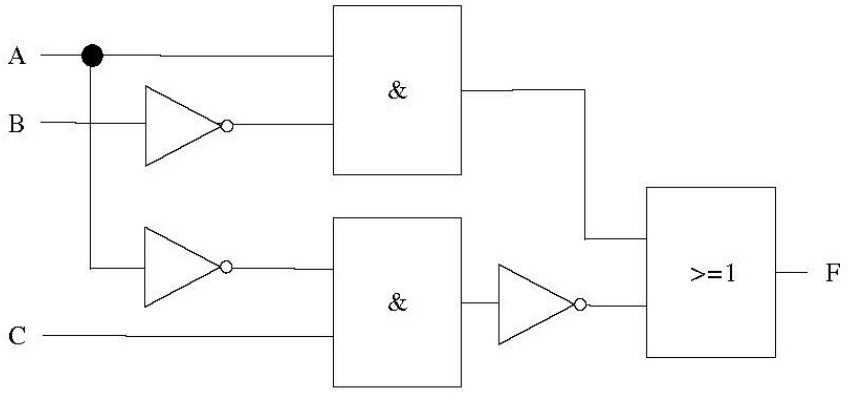


Figure 1.19: FBD for Example 8

3.2 Boolean Algebra

A logical variable, signal, or quantity can assume two values or states. Various designations are used in different contexts:

- 1 or 0;
- Logical high or logical low (or just high or low);
- 5 V/0 V (or possibly 12 V/0 V);

- TRUE/FALSE.

In the PLC context, the words discrete quantity or digital quantity are often used, but the designation Boolean variable or signal is also common. The branch of mathematics that deals with Boolean quantities is called Boolean algebra.

In the following, we will briefly study the basic rules of Boolean algebra and show how these can be used to minimize logical expressions.

Boolean Rules

The following rules are based on one or two variables, but the rules naturally apply generally, that is, for an undetermined number of variables and for cases where the members are not single variables but rather the products or sums of variables.

We begin with Boolean addition: Consider A is a Boolean variable that can have the values 0 or 1. Based upon our newly acquired knowledge of an OR operation, we can set up the following rules (compare these with functional tables for an OR port):

$$\begin{aligned} A+0 &= A \\ A+1 &= 1 \\ A+\bar{A} &= 1 \\ A+A &= A \end{aligned}$$

the next are for an AND port

$$\begin{aligned} A\cdot 0 &= 0 \\ A\cdot 1 &= A \\ A\cdot \bar{A} &= 0 \\ A\cdot A &= A \end{aligned}$$

In addition to these fundamental equations, there are the usual rules of calculation that you have learned in mathematics, such as calculating with parentheses and that the order of factors is irrelevant.

Example 9. In Example 6, we arrived at a logical function for when an alarm should become active by means of thinking practically. The expression was actually a simplified version of the following functional expression:

$$F = \bar{A}\cdot B\cdot C + A\cdot \bar{B}\cdot C + A\cdot B\cdot \bar{C} + A\cdot B\cdot C$$

which can be arranged as, thanking to the rules

$$\begin{aligned} F = \bar{A}\cdot B\cdot C + A\cdot \bar{B}\cdot C + A\cdot B\cdot \bar{C} + A\cdot B\cdot C + A\cdot B\cdot C + A\cdot B\cdot C = \\ = A\cdot B\cdot \left(\bar{C}+C\right) + B\cdot C\cdot \left(\bar{A}+A\right) + A\cdot C\cdot \left(\bar{B}+B\right) = \\ = A\cdot B\cdot 1 + B\cdot C\cdot 1 + A\cdot C\cdot 1 = \\ = A\cdot B + B\cdot C + A\cdot C \end{aligned}$$

Here is a minimization obtained by reverse use of the rule $A+A=A$. In this way, the term $A\cdot B\cdot C$ can be written three times. The purpose of this is to utilize the fact that the term $A\cdot B\cdot C$ has two signals in common with each of the three other terms. Normally, we would not write down all of the terms several times since this takes time and space, but just use this fact in this simplification, namely, that terms can be used several times. △

De Morgan Theorem

Another rule that is very useful is the so-called De Morgan theorem. This rule is used to convert expressions that contain negations. The most practical way to describe this theorem is mathematically:

$$\overline{A \cdot B} = \bar{A} + \bar{B} \text{ and } \overline{A + B} = \bar{A} \cdot \bar{B}$$

These relationships can be easily demonstrated by setting up a functional table. (Try it yourself!) The theorem is useful and can also be used for larger expressions where the terms consist of groups of variables, such as those shown in the next example.

Example 10. Let us reduce the following expression

$$Y = \left(\overline{A \cdot \bar{B} + C}\right) \cdot \left(\bar{A} \cdot C + B\right)$$

We have, using the De Morgan theorem

$$Y = \left(\overline{A \cdot \bar{B}} \cdot \bar{C}\right) \cdot \left(\bar{A} \cdot C + B\right) =$$

$$= \overline{A \cdot \bar{B}} \cdot \left(\bar{A} \cdot C \cdot \bar{C} + B \cdot \bar{C}\right) =$$

$$= \left(\bar{A} + \overline{\bar{B}}\right) \cdot \left(\bar{A} \cdot 0 + B \cdot \bar{C}\right) =$$

$$= \left(\bar{A} + B\right) \cdot B \cdot \bar{C} =$$

$$= \left(\bar{A} \cdot B + B \cdot B\right) \cdot \bar{C} =$$

$$= \left(\bar{A} \cdot B + B\right) \cdot \bar{C} =$$

$$= \left(\bar{A} + 1\right) \cdot B \cdot \bar{C} =$$

$$= 1 \cdot B \cdot \bar{C} = B \cdot \bar{C}$$

△

As we see, this expression can be reduced significantly. Note also that the signal A disappeared during the simplification. This means that the value of A did not have any significance for the resulting output Y.

Example 11.

$$F = \left(A + \bar{B}\right) \cdot \left(A + C \cdot \bar{D}\right) = A \cdot A + \bar{B} \cdot A + A \cdot C \cdot \bar{D} + \bar{B} \cdot C \cdot \bar{D} =$$
$$= A + A \cdot C \cdot \bar{D} + \bar{B} \cdot A + \bar{B} \cdot C \cdot \bar{D} = A \cdot \left(1 + C \cdot \bar{D} + \bar{B}\right) + \bar{B} \cdot C \cdot \bar{D} =$$
$$= A + \bar{B} \cdot C \cdot \bar{D}$$

△

4 Brief Presentation of the PLC programming Languages

4.1 Traditional PLC main weakness

All PLC manufacturers have used LD as one of the programming languages, but each manufacturer has previously had its own dialect. This means that, for standardization purposes, there have been relatively major differences from one type of PLC to another:

- The use of symbols and programming capabilities varied from one type of PLC to another. This meant that one had to learn a new dialect when one changed brands of PLC.

- It was difficult to structure the programs and to build hierarchical structures. Most PLCs supported a limited number of subroutines, but did not support the use of program blocks in LD. If one cannot group the code into blocks with input and output parameters, it is nearly impossible to make good structures that connect various program code blocks.

- The use of only global variables and addresses meant that the programmer had to be careful and screen a part of a program from being influenced by another part. The capability for encapsulation of the individual program parts is important in making a good, legible, and durable code that is also easy to modify later.

- Arithmetic operations were also difficult to implement. Most manufacturers had previously implemented this possibility only by using their own arithmetic blocks.

- Just as an example, Figure 1.20 shows a program code to add two numbers with the Omron C200H PLC. Numbers larger than 9999 (BCD) had to be stored in several addresses in this PLC. In the example, the number stored in addresses IR020 and IR021 are added to the number located in DM0020 and DM0021. The result remains in the addresses DM0030, DM0031, and DM0032. (IR is the input register and DM is the data memory register.)

- Reuse of program code was also difficult when the code could not be stored in separate blocks with input and output parameters. Often, one had a need to utilize the same code again several times. In many traditional LD-based PLCs, reuse is difficult in the best case.

- There were limited possibilities for exercising control of how the execution of the program took place. In most PLCs, the execution of the program was by continual scanning of the program. How much time the processor took for each such cycle was primarily determined by the size of the program application. However, one frequently found a need to be able to control the updating speed. This made it simpler to structure the code by splitting up the code into several programs that were run at different times and with different cycle times.

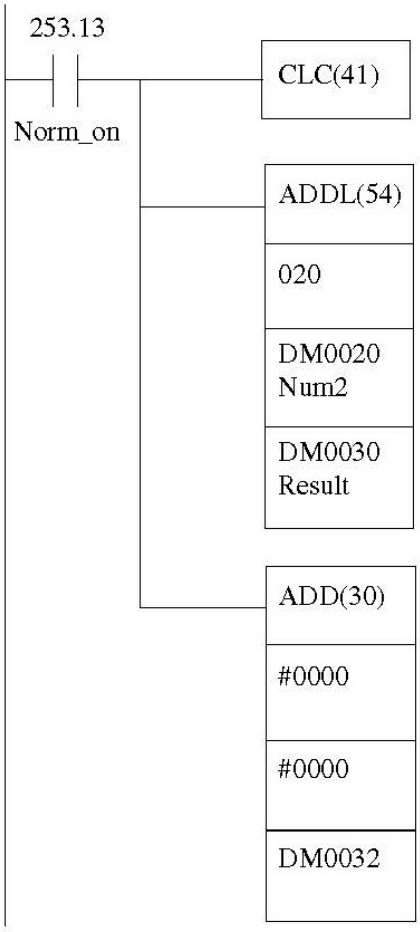


Figure 1.20: Arithmetic operations in LD (Omron C200H)

4.2 Improvements with IEC 61131-3

One of the aims of standardization is for everyone working with PLCs to understand one another better. An example of this would be that someone who is working on a project would be able to communicate better with the programmers of the PLC. It would also be easier to make alterations in the program that someone else has written no matter what PLC it applied to. The threshold for users would be lower and the training time reduced. The group working with development of the standard studied programming languages from many major manufacturers. Then they set up suggestions for languages that included the most essential features from the individual dialects. In addition to a clear definition of the languages, the standard covers several other aspects such as:

- Addressing
- Execution
- Data formats/data structures
- Use of symbols

- Sequential control
- Connections between languages

Some improvements from the standard:

- It is becoming simpler to build structured programs and collect programs hierarchically. Smaller program parts can be encapsulated into separate program elements that can be coupled in a hierarchical structure. The main program can be split up into separate parts, each of which can have its own execution conditions. Smaller program parts that require faster execution can be placed into their own folders in the program structures. The same is true of program code that is to be executed only if an abnormal condition occurs.
- The possibility of cyclic execution (fixed interval between scans) in which the cycle time can be configured means that the programmer achieves better control. Different program sections can be executed and updated at different times, which also contributes to better structure and simpler control.
- Reuse of program code becomes possible to a greater degree. The standard is not an absolute set of rules. Among other things, it means that code written for a type of PLC cannot be directly imported into a PLC from a different manufacturer, but the code can still be rewritten easily with minor modifications.
- Buying and selling products and services have become simpler since the competence of users has become more generalized. It is been simpler for users who know PLCs from one manufacturer to learn to get around in a programming language from a different manufacturer.

4.3 One the implementation of the Standard

It is important to be clear that the IEC 61131-3 standard is not an absolute standard. Manufacturers who want to adopt the standard do not need to follow the standard slavishly. This would have made the standard to comprehensive and detailed and would have limited the possibility for manufacturers to develop software (and hardware) with competitive advantages over other manufacturers.

Instead, the standard defines a comprehensive set of guidelines. These are summarized in 76 tables in the document (International Electrotechnical Commission, 2013). Manufacturers determine for themselves to what extent they will follow the guidelines. This also means that there can still be relatively large differences with respect to programming, user interface, graphics, etc. between the various manufacturers' systems even though they have all been certified by the standardizing organization.

As we see, there is a relatively large degree of freedom associated with the right to assert that a system meets the standard. However, there is an unavoidable requirement that the documentation makes clear what is and what is not in line with the standard. This may be done by reference to the individual items in the individual tables.

The documentation for the system must therefore contain a statement of conformity such as "This system conforms to the guidelines in IEC 61131-3 in the following properties." The information in the table must be taken directly from the relevant subparagraphs in (International Electrotechnical Commission, 2013).

Example 12. The following table has been taken from the documentation for the programming tool PL7 Pro from Telemecanique (Schneider Electric, 2002 and 2004).

Table number	Characteristic number	Description of characteristics
59	1	Left power rail
59	2	Right power rail
60	1	Horizontal link
60	2	Vertical link
61	1	Open contact
61	3	Closed contact
61	5	Positive transition contact detector
61	7	Negative transition contact detector
62	1	Coil
62	2	Negated coil
62	3	SET (latch) coil
62	4	RESET (unlatch) coil

The table refers to subparagraphs concerning graphical elements in the LD programming language. △

4.4 Program Organization Unit

Before introducing the PLC programming languages it of interest to define the Program Organization Unit (POU) term which is extensively used in the PLC jargon. A **Program Organization Unit (POU)** refers to a modular component of a control program. POUs help in structuring and organizing the control logic of PLC applications. They are the fundamental building blocks in the development of PLC programs, and each POU is designed to carry out a specific function or task.

There are three main types of POUs in PLC programming:

- **Function (FC)**: A reusable block of code that performs a specific task and returns a single result. Functions do not retain any internal data between calls (stateless).
- **Function Block (FB)**: Similar to a function but with internal memory. Function blocks can store data between calls (stateful), which makes them suitable for tasks that require memory of previous states.
- **Program (PRG)**: This is the main unit of the control system, where the overall logic is written. Programs call other POUs (functions and function blocks) and orchestrate the control flow.

POUs are essential for creating well-structured, maintainable, and reusable PLC code. They allow the breakdown of complex tasks into smaller, manageable parts.

4.5 ST (Structured Text) Language

As the name indicates, ST is a text-based language. It is, in contrast to ILs (the other text based language in the standard), a high-level language where many operations and instructions can be performed with a single command line. If we were to compare it to other high-level languages, ST most resembles C. ST has been specially developed to program complex arithmetic functions, manipulate tables, and work with word objects and text. The example below contains ordering of values, conditional instructions, a FOR-loop, and the declaration of a variable. (The example is written with the programming tool CODESYS.)

```
PROGRAM PLC_PRG
VAR
    Index           : INT := 1;
    Parameter       : ARRAY [0..10] OF REAL;
    Data AT %MW5    : ARRAY [0..10] OF REAL;
END_VAR

IF %MX0.1 THEN
    %MW0 := 0;
    %MX0.2 := %MX20.0;
ELSIF NOT %MX3.0 THEN
    %MW0 := 10;
END_IF;

FOR Index := 1 TO 10 BY 1 DO
    Parameter[Index] := Data[Index];
    Index := Index - 1;
END_FOR;
```

Figure 1.21: ST Example text

4.6 FBD (Functional Block Diagram) Language

FBD is a graphical language which, described in a very simplified way, is based on connecting functions and Function-blocks (Figure 1.22). The language includes, among other things, use of standard logical function such as AND, OR, NOT, etc. and function blocks (FBs) such as timers and counters, but self-constructed functions and FBs can also be defined. Many who have little knowledge of digital electronics therefore think that this is a great language to use or at any rate to get started with. It can be practical to use the language to program logical algorithms (Boolean functions) and control functions such as regulator structures and the like. However, the language does not offer anything that cannot be executed in ST, but it gives a better overview (at any rate, for smaller programs).

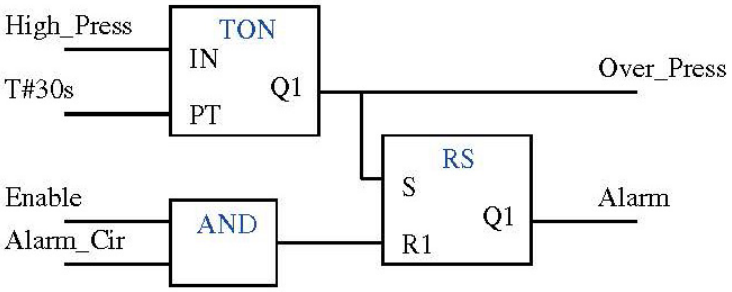


Figure 1.22: FBD Example text

4.7 LD (Ladder Diagram) Language

Ladder Diagram, or just plain LD, is still used to a large degree by many PLC programmers. This is despite the fact that both ST and SFC are more efficient languages in most contexts. The reason that LD is still used so much is that it is simple to understand and that it is based upon traditional electrical wiring diagrams (relay diagrams). The well-established language has, for a long time, continued to maintain its presence because new engineers and technicians still have to learn to read it, understand it, and apply it. Some people also apply the concept of relay diagram to program code written in LD.

LD basically consists of a set of instructions that execute the most basic types of control functions: logic, time control, and counting, as well as simple mathematical operations.

An example of program code is shown in Figure 1.23. The example of code contains standard elements such as contactors (NO and NC), coils, flank-detecting contacts, and a timer.

Most PLC manufacturers nowadays make it possible to perform advanced additional functions in LD, often integrated with other languages such as FBD and ST. For smaller controls, LD can therefore be a fine choice of programming language.

The basic functions that are needed in order to implement smaller applications can be learned relatively quickly and the graphical presentation can be understood intuitively.

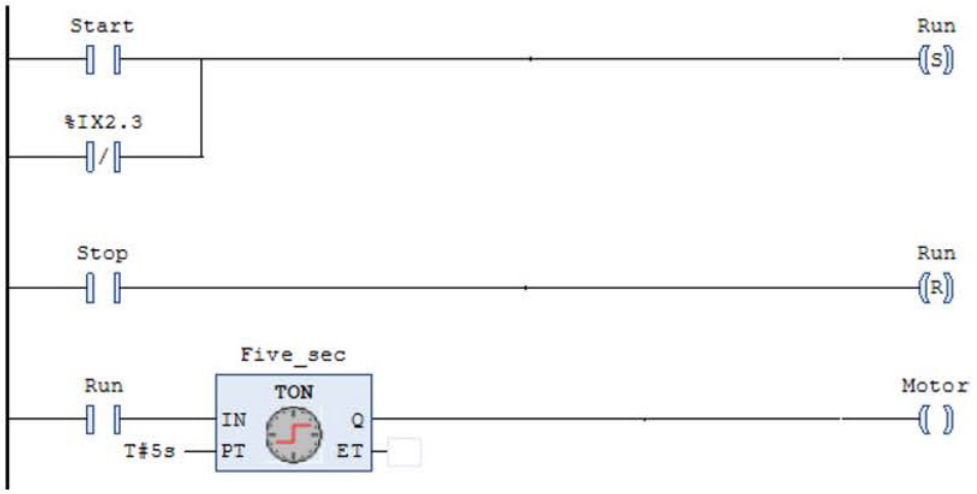


Figure 1.23: LD Example

4.8 IL (Instruction List) Language

IL is an assembler-like low-level language. Even though there are disadvantages associated with the use of a low-level language such as IL, the advantage of the language is that it does not require much computer power.

The reason that the language continues to be used is that the language, together with LD, has existed longer than the other languages in the standard.

Many older PLCs can be programmed only with IL/LD. There can therefore be cases where one can use IL, for example, when program code written in IL is taken from an old PLC for modification or analysis.

The IL language, however, has limited capabilities and applications because it is hard to learn and not very comprehensible when programming tasks are numerous or complex.

On older PLCs code in IL (or LD) be programmed and transferred to the CPU via a special panel. It was not necessary (or possible), as it is today, to connect a PC for programming and diagnostics.

An example of such an older PLC that has a programming panel is the Omron model C20, which is shown in the picture below.

Example of code in IL:

```
LD     run
ST     ttimer1.IN
LD     counter
GE     5 (* IF counter >= 5, *)
JMPC   next (* jump to next *)
CAL    timer1(PT:=t#10m)
LD     timer1.Q
ST     motor
next:
```

4.9 SFC (Structured Functional Chart) Language

SFC is a graphical tool that is ideal for programming sequential controls and implementing state-based control algorithms (Figure 1.24). SFC is actually not a programming language in the traditional sense, but a more graphic approach for structuring program code. It is also brilliantly adapted for this. All the other languages in the standard can be used together with SFCs, and it is necessary for at least one of them to be able to implement all the necessary transitions and actions.

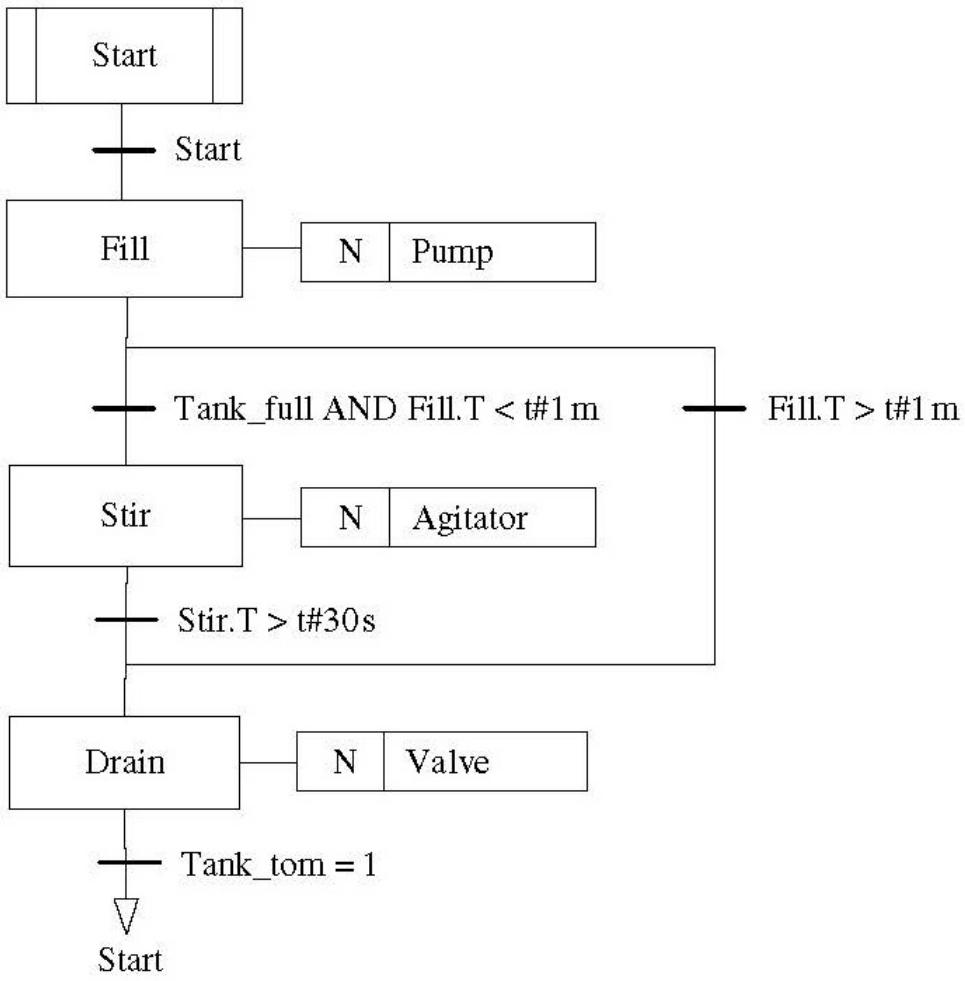


Figure 1.24: SFC Example

5 IEC 61131-3: Common Language Elements

It is one of the goals of the standard that all five languages should be able to integrate, so that, for instance, a Program Organization Unit (POU) written in ST can contain a call of a POU written in LD or that a program code with associated variable declarations and comments can be converted from one programming language to another.

In order for it to be possible to implement this, there are many concepts and elements in the standard that follow defined rules that are common to all the languages. This applies to:

- Identifiers
- Keywords
- Comments
- Literals
- Addressing
- Data types
- Variables

5.1 Identifiers, Keywords, and Comments

This group of common elements deals with permitted use of characters, either as identifiers, variables, program names, or for comments. For example, some particular combinations of characters are not permitted to be used as identifiers. Such combinations are called keywords and are processed as such.

Identifiers

An identifier is a fancy word for name. Many elements must be given a name before or during the programming. This applies to programs, variables, user-defined functions and functional blocks, and steps and actions (SFC) etc.

The standard requires that, as a minimum, the first six characters in a name should be tested for uniqueness by the hardware. This means that the system must be able to distinguish between the variable names **Motor1** and **Motor2** (six characters), but not necessarily between the variables **Switch1** and **Switch2** (seven characters).

However, it is freely up to the manufacturer to implement a higher number and most systems can distinguish names with a length that is much higher than six.

Other guidelines in the standard applicable to identifiers are:

- Interpretation of identifiers must be independent of character case. For example, the system should interpret Sensor, sensor, and sEnSoR as the same identifier.
- Identifiers may not contain a space.
- They must begin with a letter or an _ (underscore).
- They may not end with an underscore or have two sequential underscores.
- Numerals are permitted, but not first in the identifier.

As an example **AbCDe**, **_ABCdE**, **AB_CDE**, **A_2_3** are allowed identifiers whereas **A_B__C** (two adjacent _), **1_A_B** (numeral first), **A_B CD** (space), **AbCDe_** (ends with _) are not allowable. (Note that keywords cannot be used as identifiers.)

Since it varies from one system to another in how many characters are permitted to use in identifiers, it makes sense to be rather modest in the selection of identifiers. Even though most systems will probably support it, it is probably a good idea to avoid identifiers such as **This_is_a_long_identifier**. It quickly becomes difficult to keep track of such long identifiers and it naturally takes longer to enter them.

5.2 Keywords

Keywords are unique combinations of characters that are reserved from being used as identifiers, since they are only to be used as syntactic elements in programs. The standard remains open for national standardization organizations to translate keywords and publish a national list in place of the list that is published in International Electrotechnical Commission (2013).

It seems doubtful that this is actually taking place. Here are some examples of keywords described in the standard:

- TRUE, FALSE
- IF...THEN...ELSIF...ELSE...END_IF
- AND, OR, NOT, MOD, XOR
- FUNCTION...END_FUNCTION
- VAR...END_VAR

As we see, only uppercase letters are used in the keywords. Normally, the system is not sensitive to the use of upper- or lowercase letters when writing keywords. Generally, the system will automatically correct and display only uppercase letters in the editor and then in a particular color to clearly differentiate the keywords from other words and identifiers.

In practice, the complete list of reserved words (that is words that are not permitted to be used as identifiers) is much longer than the list found in International Electrotechnical Commission (2013). The reason for this is that manufacturers offer many predefined functions and functional blocks that are assigned unique identifiers that are reserved against use by programmers.

5.3 Comments

Comments can be used everywhere, in all POUs and in any programming language. The purpose of comments is partly to make it easier for the programmer to keep track of his/her own program code and partly to make it easier for others to read and understand the code.

Frequent use of comments is a good habit to get into. The standard formulates several requirements for how comments are to be implemented. The comments here have been written in *italics* for clarification:

- Comments should be enclosed by (* and *), that is, parentheses and asterisks, both before and after the comment itself: *(* This is a lengthy comment, long enough that it may well extend over more than one line of written code *).*
- Comments may be placed anywhere at all in the code, but not in the middle of a variable name or the like.
- Use of nested comments is permitted as long as (* and *) come in pairs, as in this example: (* This (* is *) legal *) but (* this (* is illegal *).
- A comment may contain all characters.
- The standard also defines the alternative3 character combinations /* and */.
- A one-line comment can be indicated following the character combination // as here:
- *//This is a comment on a single line.*

It is also permitted to use (* and *) for a one-line comment. Note that the number of characters in a single comment is dependent upon an implementation dependent parameter. The code example below contains various applications of comments. Since all characters are permitted within comments, there can be several asterisks between (* and *).

```
(********* RS FLIP-FLOP *********)
(* Example of function block that
implements a reset-dominant flip-flop *)
(*****************************)
FUNCTION_BLOCK RS
   // Declares input variable:
   VAR_INPUT
     Set :    BOOL;
     Reset :  BOOL;
   END_VAR
   //Declares output variable:
   VAR_OUTPUT
     Out : BOOL;
   END_VAR
   Out := NOT Reset AND (Set OR Q1);  (*The FB's program code *)
END_FUNCTION_BLOCK
```

Figure 1.25: ST with comments

5.4 About Variables and Data Types

In traditional PLC systems, only global addresses are used, and these often have fixed locations in memory. The disadvantage of this is that the users themselves must be careful that no conflicts occur when they are written to the same addresses from different parts of the program or from different programs. In other words, the user monitors which addresses he/she has used and to what extent they do not overlap with other addresses.

The only advantage with fixed addresses is that the user does not have to declare the variable. With IEC 61131-3, all addresses that will be used must be declared7 in the form of variables. If you do not specify something else, all variables will be declared as local variables within the individual POUs. Then they will be accessible only within the POU where they were declared. Then there will be no conflict with variables declared in a different POU, even though the same variable name has been used.

Nor is the type of data that an address can contain fixed in traditional PLCs. Since the data type is not declared, the same address or address area can be used for floating-point numbers in one part of program code, for instance, and the four integers in another part of the code.

This is possible since the same area in memory is often used for different types of data so that the user must be careful that there is no overlap between addresses that contain different types of data.

These two objects have different data types but refer to the same address location (same memory area). When using both objects in the program, logical errors in the form of assignment of content will occur because assignment of content to one object will change the other object.

With IEC 61131-3, data types are declared explicitly when the variables are declared. If any conflict arises among various data types, for instance, when a variable that contains data of the floating-point type (REAL) is assigned to a variable that contains data of the whole-number type (INT), this will be considered as a syntax error. The compiler will therefore deliver a message about it.

5.5 Pragma and Literals

A pragma instruction can be used to affect the properties of one or more variables with respect to compilation or precompilation processes. This means that a pragma influences the generation of the code. It can also be used as another type of comment if it does not have a valid prefix that the compiler recognizes.

Both syntax and semantics are implementation dependent so that the use of pragmas is entirely up to the manufacturer of the system to define. The only requirement is that it be enclosed in curly brackets of the type , both of which are on the same line. A common application in programming is to use them to provide information in the code that is to be displayed on a screen during the run.

Literals

In an original grammatical context, the word literal means “verbatim” or “literally,” but in the digital context, a literal is a “nameless constant” This somewhat mystical concept refers to two conditions: the word constant indicates that it deals with constant values (in contrast to variables where the content can change). Nameless refers to the fact that this is a value that is provided directly in the program code instead of being declared beforehand.

In the IEC 61131-3 standard, the concept of literal has a significance that is in line with its digital significance since it deals with how values are assigned. The value format can be numbers, text strings, or time, and there are, naturally enough, rules for how various types

of values are entered and how the software will interpret the values. The standard defines three main types of literals:

- Numerical literals: Numerical values of the integer and floating-point types
- Text strings: Sequences of characters
- Time literals: Values such as duration, time of day, or date

When values are assigned to a variable, the format and range of value depends upon the data type of the variable. It is therefore natural to group literals according to data type.

5.6 Data Types

Depending upon what task the PLC is to perform, there will be a requirement for many different types of data. For example, there may be Boolean (BOOL) variables, perhaps associated with digital I/O, various types of integers (INT, UINT, DINT, etc.), floating-point objects (REAL), or types for management of time (TIME).

While a variable name identifies the storage location of a variable and the variable type indicates, for instance, whether the variable is global or local, the data type indicates what type of values (or literals) the variable can have. This is also significant for what operations can be undertaken with the variable in question and how the contents of the variable are stored.

When a variable is declared, the data type must be declared at the same time. The standard naturally defines guidelines for how variables and data types are to be declared and ranges of values for the individual data types. The declaration of a variable, together with its properties (as a data type), is done in a separate declaration field within the individual POU.

The declaration is made in the same way, no matter which programming language is being used otherwise in the POU.

The standard defines a set of elementary data types. These will be predefined in PLCs that adhere to the standard. In addition, the standard contains guidelines on how the system can implement user-defined data types.

Numerical data types

Figure 1.26 describes all the basic integer and floating-point types that are defined in International Electrotechnical Commission (2013), while Figure 1.27 describes data types in bitstring format. It is not certain that the manufacturer has chosen to implement all of these, particularly since some are identical in practice. The tables show, for each data type, information on associated keywords, the number of bits each element accepts, and the resulting possible range of values. As we see, a distinction is made between integer types with and without a sign. In many contexts, such as counter values, there is no requirement for negative numbers, and it is therefore an advantage to be able to use separate types for these. Then the system will give an error message or warning for a negative result, and at the same time, the positive value range will be larger for the same number of bits in memory. The purpose of similar types with different lengths (number of bits) is also a question of resources. There is

Format	Data type	Number of bits	Value range	Initial value
Integer (w/sign)[a]	*SINT*	8	–128 to + 127	0
	INT	16	–32 768 to + 32 767	0
	DINT	32	-2^{31} to + 2^{31}-1	0
	LINT	64	-2^{63} to + 2^{63}-1	0
Positive integer	*USINT*	8	0 to 255	0
(unsigned)[b]	*UINT*	16	0 to 65 535	0
	UDINT	32	0 to 2^{32}-1	0
	ULINT	64	0 to 2^{64}-1	0
Floating-point	*REAL*	32	$\pm 10^{\pm 38}$	0.0
numbers[c]	*LREAL*	64	$\pm 10^{\pm 308}$	0.0

Figure 1.26: PLC Numerical Data Types

Format	Data type	Number of bits (*N*)	Value range
Boolean	*BOOL*	1	0/FALSE
Bit strings	*BYTE*	8	16#00[a]
	WORD	16	16#0000
	DWORD	32	16#0000_0000
	LWORD	64	16#0000_0000_0000_0000

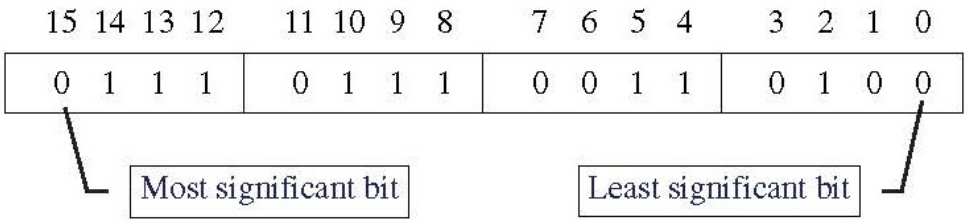
Figure 1.27: PLC Numerical Data Types in Bitstring Format. The prefix 16# indicates that this is a literal in hexadecimal form.

no reason for using more bits than are necessary to store the values in question. If a variable is going to be used for counting the number of bottles in a crate, for instance, it is sufficient to use the data type USINT.

The standard also defines a group of data types that are called bit strings (Figure 1.27). In this group, we find a very fundamental data type, namely, BOOL. This type is used for variables associated with digital inputs and outputs as well as status and memory flags.

The other bit-string formats BYTE, WORD, DWORD, and LWORD correspond in many ways to data types for positive integers, SINT, INT, DINT, and LINT. The reason for defining these bit-string data types is because they can be used to store binary information. This can be advantageous during communications with external units and instruments for storing and setting various status bits (flags).

Another application is management of multiple Boolean objects efficiently, where each individual bit can represent a digital output or a signal to a stepping motor. In the next chapter, we will see that there are many functions that are defined in the standard in order to be able to manipulate the content of bit-string variables. The data type WORD is also used for declaration of variables associated with digital inputs and outputs. Many conventional PLCs also use the types BYTE, WORD, and so forth, and an address with the length of 16 bits is also traditionally often designated as a word. The example below shows an illustration of the content in a word.



For integers with signs (data type SINT), the most significant bit (MSB) is used for the sign (0 = positive and 1 = negative).

Numerical Literals

Numerical literals can also be classified into the main types integers and floating-point numbers. The integers group thus also includes the bit-string types BYTE, WORD, DWORD, and LWORD. The standard defines several formats for assigning and representing integer values, but it is not certain that all manufacturers have implemented all types. An integer value can be entered directly in the form of a decimal number, for instance, 235. If desirable, the user can also enter (and display) integer numbers in binary, octal, or hexadecimal form. In order to distinguish these formats from one another, the base number is given, followed by the character # before the value in question. If the value is not preceded by a base number and #, the value is automatically interpreted as a decimal number. Take a look at Figure

Decimal	Binary	Octal	Hexadecimal
0	2#00000000	8#000	16#00
37	2#0010_0101	8#45	16#25
–14	–2#00001110 (or 2#11110010)	–8#16 (or 8#362)	–16#0E (or 16#F2)
12_534	2#00110000_11110110	8#030366	16#30f6

Figure 1.28: Integer Literal

1.28, how large a number can be entered and stored depends upon the number of bits that are available (8, 16, 32, or 64). For instance, for the numbers 0.37 and -14 (decimal), it is sufficient with eight-bit data types (such as SINT), while the number 12 534 (decimal) requires a minimum of a 16-bit data type.

As the figure shows, there is a guideline for the use of the underline character for dividing up long numbers in order to improve legibility. This is particularly useful for entering and representation of binary numbers. And, as with the definition of identifiers, no distinction is made here between uppercase and lowercase letters.

For negative numbers, the figure also shows the complementary numbers (in parentheses). Even though it is possible to enter the numbers preceded by a sign, they are stored in the two's complement format in the PLC. This is the way that negative numbers are handled. Two's complement implies that all bits in the binary representation of the value are inverted and then the number is added to 1 (binary).

Think about another example, integer -14 is equal to -00001110 in binary. The two's complement of this number then becomes $11110001 + 1 = 11110010$. Negative numbers in octal and hexadecimal form can also be expressed using the two's complement form

Floating point numbers

The word floating point originates from the way these numbers are represented in the PLC (or in a computer). The number is stored in two parts, the mantissa and the exponent, according to the following formula:

$$\text{Floating-point number} = \text{Mantissa} \times 10^{\text{Exponent}}$$

512.0 can also be written as 5.12×10^2, 12532 can be written as 1.2532×10^4, 0.125 can be written as 1.25×10^{-1}. What is "floating" here is the decimal point in the mantissa because this is moved after the number has been entered. Often the letter E is used instead of the base number 10 for representing the number. For example, 12532 can look like $1.2532E + 4$, and 0.00001234 is written as $1.234E - 5$. How large or small the floating-point numbers have to be before this display format is used depends upon the implementation. The accuracy and range of floating points depends upon how many bits are used to represent the mantissa and how many are used to represent the exponent. The range of values for a 32-bit floating-point number (REAL) is from 10^{-38} to 10^{38} for positive numbers and correspondingly for negative numbers (-10^{38} to -10^{-38}). This means that 6 bits are used for storage of the exponent, one bit for the sign and the remainder for the mantissa. The use of floating-point numbers is more accurate than simply operating with integers. Floating point is also used for intermediate storage and for the results of arithmetic calculations.

Data Types for Time and Duration

The standard defines some distinct data types that are not ordinary data types in the basic programming languages. These data types are especially designed for control of time and duration (Table 6.3). In industrial control systems, there is often a requirement for monitoring the duration of events and actions, for instance, at what time of day or on which day of the week actions should be performed. With these specially designed data types such as TIME, these events can be programmed in a more structured way. Notice that, the data type

Data type	Description	Initial value
TIME	Duration	T#0s
LTIME[a]	Duration	LTIME#0s
DATE	Calendar date	–[b]
LDATE[c]	Calendar date	LDATE#1970-01-01
TIME_OF_DAY or *TOD*	Time of day	TOD#00:00:00
DATE_AND_TIME or *DT*	Date and time of day	–

Figure 1.29: Time and Duration

LTIME is a 64-bit integer with signa and the resolution is in nanoseconds. The similar data types DATE, TOD, and DT are used for many different purposes. It may be to activate and terminate actions according to time of day or to particular dates. This is useful for programming building automation such as air-conditioning and lighting.

Another example of use of these data types is for reporting purposes. There may be requirements for storing the date and time when an alarm was activated or when an operational stoppage took place. If power fails, there may be various actions that should be performed when power is restored, depending upon how long it was out.

Time Literals

The standard permits many ways of entering and displaying time and duration. All time literals must have a prefix that indicates the type, followed by the character #. The actual time follows this. The following are used for specification of time and duration:

- **d** for days
- **h** for hours
- **m** for minutes
- **s** for seconds
- **ms** for milliseconds

Time literals must be entered in the proper order: days, hours, minutes, seconds, milliseconds. The guidelines permit both uppercase and lowercase letters, negative values, use of underscore and decimal point, and both short and long forms of prefixes. Below are some examples of correct literals for variables of the type TIME. The following examples make easy to understand the timing syntax

- T#25s
- T#-25s (negative time)
- T#12.4ms
- t#12h
- T#12h23m42s
- t#12h_23m_42s_67ms
- TIME#45m
- time#4m_20s

Note that it is possible, if the manufacturer so permits, that the most significant part of the time literal can include overflow. For instance, a time can be entered as T#29h25m. This would be the equivalent of T#1d_5h_25m.

Entering and display of literals for data types DATE, TOD, and DT must also follow a particular order. For the data type DATE, the literal should follow the form:

DATE or D # Year - Month no. - Day no.

Literal for the data type TIME_OF_DAY (TOD):

TIME_OF_DAY or TOD # Hours : Minutes : Seconds

(Note that hours, minutes, and seconds are separated by a colon, while year, month, and day are separated by a hyphen.)

Literal for the data type DATE_AND_TIME (DT):

DATE_AND_TIME or DT # DATE-literal - TOD-literal

Some examples

- DATE#2007-05-31
- D#1968-11-25
- time_of_day#08:45:00
- TOD#17:30:45
- DATE_AND_TIME#1814-05-17-13:45:00
- dt#2007-08-01-12:30:00

Text Strings

The last of the elementary data types is text strings represented by the keywords CHAR, WCHAR, STRING, and WSTRING.11 All these data types are used to manage letters and other characters. The difference between CHAR and WCHAR and between STRING and WSTRING depends only upon the way the content is interpreted and stored. CHAR and STRING are text in ASCII format, while WCHAR and WSTRING are text in Unicode format. This last is an expansion from the first version of the standard and was introduced in order to be able to handle a greater variety of characters. This is useful if one needs to enter special characters that are not found in ASCII and for handling many languages other than English. The reason that the Unicode character set contains more characters is that each character occupies two bytes of data or 16 bits, while each character in ASCII occupies only one byte To help the compiler help you catch possible errors in the code, there is a difference in entering the literals in the two formats, so that ‘ ’ is used around CHAR and STRING types and “ ” around WCHAR and WSTRING.

Description	Data type	Number of bytes per character	Initial value
A single character	*CHAR*	8	'$00'
	WCHAR	16	"$0000"
Text strings of variable length	*STRING*	8	''
	WSTRING	16	""

Figure 1.30: String Type

Example:

aString := 'This enters a STRING'

aWString := "This enters a WSTRING"

The first 127 characters in ASCII and Unicode are otherwise the same, so that if you do not have any special requirement, it is recommended that you use the data type STRING for text variables. When declaring a variable of the STRING type, the programmer, if desired, can also enter the length of the variable, that is, the maximum number of characters that the variable can contain. If no length is stated, a default length will be used. The maximum permitted length of text strings depends upon the implementation. Typical applications of this data type are found in dialogues with the operator's panel and HMI or for sending data to printers. In the standard, there are several defined functions for management of text strings such as finding the length of the string, inserting characters in a string, and deleting characters from a string. In addition to ordinary letters and characters, the standard defines some special combinations that can be used to format text for display or printout. These special combinations, which are not themselves displayed on the screen or printout, consist of a dollar sign ($) followed by the letters L, N, P, R, or T.

Formatting code	Significance for printout
$L	Line feed + carriage return (new paragraph)
$N	New line
$P	New page
$R	Return key
$T	Tab key

Figure 1.31: Special Characters

User Defined Data Types

In addition to the elementary data types and any special data types defined by the manufacturer, it is possible to define one's own data types. These are called derived data types

since they are derived from (that is, based upon) the elementary data types.

The purpose of user-defined data types is to be able to obtain a more structured code, particularly in those cases where it is natural to group several I/Os of various data types by defining a new class of I/O composed of elementary data types as members.

Derived data types are defined by means of the keywords TYPE and END_TYPE. It is possible that a development tool will permit only definitions of one type within each set of the keywords TYPE and END_TYPE.

The defined types will be accessible to the entire project (globally). In this way, global and local variables can be declared based upon the new data type. Sometimes, there is a requirement to be able to process sets of several variables or values in a structured way (ARRAY and STRUCT) or simply to define some additional different properties for the selected data type, for instance, to limit the value range.

After we have obtained a little experience in programming, the use of the elementary data types will be, well, elementary. Structured data types (STRUCT) are a little more complicated and are seen traditionally as not useful in the PLC programming. Nevertheless, they offer an important contribution that improves the capability of producing well-structured program code.

Example 13. Assume that the PLC is to be used for monitoring and controlling a cold-storage site. The storage consists of several different freezer rooms, but each freezer room is equipped completely identically, with a freezer unit that is to be controlled and temperature and pressure that are to be monitored.

Instead of declaring one variable for each individual I/O in each freezer room, one can first make a self-defined structured data type called, for instance, Freezer, where all I/Os associated with a room are members of the new data type. Then one can declare a variable of the type Freezer in each of the freezer rooms in the storage site. △

As mentioned earlier, it is possible to define new data types based upon an elementary type but with a limited range of values. This can be done as follows:

```
TYPE
          HoleNumber :  INT (-800..200);
END_TYPE
```

Here, a new data type called `HoleNumber` is defined on the basis of the elementary data type INT, but with a limited value range from -800 through 200.

Enumeration is a user-defined data type that is based upon user-defined text constants. The constants are referred to as enumeration values. In enumeration, the program developer enters a list of permitted text strings that a variable can assume. Defining an enumerated data type follows the same syntax as statement of a value range:

```
TYPE
          Color :  (Green, Yellow, Red);
END_TYPE
```

Here, the data type `Color` is defined, which can take on one of three possible values: “Green,” “Yellow,” or “Red.” A variable can then be declared based upon the new data type

Color. Unless otherwise specified, the variable will then initially have the first value in the list, in this case the value "Green."

Note also that the specified permitted values are compatible with the use of integers. This implies that instead of operating with the values directly, one can use numbers to identify them. In the example above, the value "Green" is automatically assigned the value 0, since nothing else is specified, and "Yellow" takes the integer 1 and "Red" takes the integer 2. This means that it is possible to use variables with an enumerated data type in control structures that loop. If desired, other numerical values can be assigned to the enumerated values:

```
TYPE
          Card:  (Jack := 11, Queen := 12, King := 13, Ace := 14);
END_TYPE
```

A common derived data type is *Arrays*. This is not actually a user-defined data type since arrays can also be defined directly in the declaration field in a POU or in the list of global variables. However, similar to limitation of a value range, one can define a data type and use it in declaring a variable. The individual elements in the array can be often elementary data type or a user-defined data type. It is possible to define one-, two-, and three-dimensional arrays. The standard suggests the following syntax:

```
TYPE
          Tab_1dim :  ARRAY [lower..upper] OF DATATYPE;
          Tab_2dim :  ARRAY [lower1..upper1, lower2..upper2] OF DATATYPE;
END_TYPE
```

The only requirement placed upon the lower and upper boundaries is that they are integers that lie within the value range of DINT. A lower boundary can well be 437, for instance, as long as the upper boundary is higher. In other words, the numbers that are used for the elements are not important but the array's dimension is. Some examples:

```
TYPE  One_dim:    ARRAY  [0..9]              OF  USINT;  END_TYPE
TYPE  Two_dim:    ARRAY  [0..9, 1..4]        OF  USINT;  END_TYPE
TYPE  Three_dim:  ARRAY  [1..3, 1..4, 0..5]  OF  USINT;  END_TYPE
```

Up until now, we have seen a series of elementary data types and how such data types can be organized in array form. Sometimes, there is a requirement for more complex data structures, where several different data types appear as subelements within a comprehensive data type. These subelements can be any one of the aforementioned data types, including enumerated types and arrays.

Structured data types are declared within the keywords TYPE, STRUCT and END_STRUCT, END_TYPE with the following syntax:

TYPE Name_of_datatype:
STRUCT
<Declaration of datatype 1>;
<Declaration of datatype 2>;
...
<Declaration of datatype n>
END_STRUCT
END_TYPE

An example

```
TYPE ANALOG_SIGNAL:
   STRUCT
     Raw_value :  WORD;
     Scaled_value :  REAL;
     Min_raw :  INT (-32767..0);
     Max_raw :  UINT (0..32768);
   END_STRUCT
END_TYPE
```

5.7 Variables

There is a fundamental difference between addressing in elder PLCs and declaring and using variables in PLCs that follow the standard. In conventional systems, only addresses that are global and have fixed locations in memory are used. Global means that the addresses are accessible from all parts of the program and from all programs in the PLC. Fixed location means that the user specifies what portion of the memory the addressed object uses. The memory can be located in the CPU, can be in the form of a separate memory card (e.g., a flash memory), or could be built into an input or output module. Those who have experience with PLCs from different manufacturers know that the absence of a standard has meant differences in addressing syntax between different types of PLCs. Before (or during) programming, it is necessary to enter elements for storage of data by specifying what type of data it applies to (integer, floating point, text, etc.) and giving names to these elements with logical, reasonable identifiers. This is called *declaration* of variables and is done at the beginning of all POUs. What the declaration editor looks like depends upon the implementation. It can be in tabular form, or, as here, in a text-based form

```
VAR
   Temp_ref    :  INT := 70;
   Deviation   :  REAL;
END_VAR
```

We see that the declaration of variables begins with indicating the type of variable by using the correct keyword. For instance, there are local variables, global variables, and input and output variables. Normally, a local variable is declared by using the keyword VAR. Heterogeneous data type variables can be declared together

```
VAR
   Start          :  BOOL := TRUE;
   Alarm          :  BOOL;
   MV             :  REAL := 48.5;
   Temp_ref       :  INT := 70;
   Denomination   :  STRING := 'Degrees';
   Light          :  Color := Yellow;
   Time1          :  TOD;
   Time2          :  TIME := time#70m_30s;
   Date1          :  DATE := DATE#2007-06-18;
END_VAR
```

It is also possible to declare several variables of the same data type in succession.

```
VAR
   Value_1, Value_2, Value_3   :  INT;
END_VAR
```

The compiler helps reduce the use of erroneous data types by checking data types when the variables are used. If, for instance, an attempt is made to assign a value of the REAL type to a variable of the BOOL type, the compiler will give an error message.

In the previous examples it can noticed that variables of different data types can be assigned initial values (or starting values). These values overwrite the default values that the individual data types are originally assigned. These default values are, for instance, 0 for integer data type, FALSE or 0 for data type BOOL, and '', an empty string for data type STRING. It is possible to define data types on the basis of elementary (integer) types but with a limited range of values. It is actually not necessary to first define a new data type in order to be able to use the new type in declaration of a variable. Such a value limitation can be imposed directly during the declaration of variables:

```
VAR
   Hole_Num  :  INT (-800..200);
   Pos_Num   :  UINT (0..10000);
END_VAR
```

Constants and Retain

Sometimes, there is a requirement to enter and store values that should not be changed by the program code. This is achieved by using the qualifier CONSTANT after entering the VAR keyword:

```
VAR CONSTANT
   Setpoint    :  INT := 75;
END_VAR
```

It seems a little odd to mix the two keywords VAR and CONSTANT together, but this is the way it is defined in the standard. Constants are used to retain parameters and setting such as duration, number, time of day, etc. It is easier and more structured if such values can be declared as a group instead of entering them directly into the program code. Both local and global variables can be declared as constants, that is, the keyword CONSTANT can also be used as an attribute of the variable type VAR_GLOBAL.

Another important qualifier is RETAIN. This is used on variables that should retain their value during an out-of-control situation such as power failure or, for that matter, during a controlled shutdown such as a reboot of the PLC. When the PLC is in RUN again, the values that the variables had before the stop will be used as processing progresses.

```
VAR RETAIN
   Stored_Value  :  WORD;
END_VAR
```

RETAIN can also be used for the types VAR_GLOBAL and VAR_OUTPUT.

Local vs. Global Variables

One of the most notable properties of variables in IEC 61131-3 is that variables can be declared locally within the POU in question. This means that the same identifier can be used again as a name for a variable in another POU as long as the variable is declared as local within its own POU. In addition to it being useful in practice to be able to use the same identifier in several places, this also reduces the risk of undesired overwriting of data.

The variables that are declared in the foregoing examples are all local. This is characterized by the group types that are used in the declarations: VAR – END_VAR. Sometimes, it is desirable and necessary to use global variables, variables that are accessible from several POUs within the resource or for several resources (PLCs).

Global variables are not declared within a POU as local variables are, but rather are declared at a higher configuration level. The format for declaration is like that for local variables, it is only that another group type is used:

```
VAR_GLOBAL
   ItemCount    :  UINT;
   AlarmLight   :  BOOL;
END_VAR
```

Two global variables are declared in this example: one variable of the data type UINT called ItemCount and a variable of the Boolean data type called AlarmLight. These two global variables are now declared, but they cannot be used without further programming. In order to have access to a global variable from a POU, the standard says that the POU where

the variable will be used must contain a form of declaration for the same global variable. This is done by using the group type VAR_EXTERNAL:

```
VAR_EXTERNAL
   ItemCount    :  UINT;
   AlarmLight   :  BOOL;
END_VAR
```

It can seem unnecessary to have to declare the same variable several times, but the reason for this is to keep the programmer from accidentally using a local identifier when he/she has forgotten that the name had already been used as the name for a global variable. It is not certain that this was a deliberate reference to the existing global variable and that the user had intended to assign a completely different data type and value to the variable. If the compiler had accepted the use of this identifier without it having been declared again, it could happen that the programmer had overlooked this inconsistency.

Input and Output Variables

This has nothing to do with a PLC's physical inputs and outputs, but rather with the variables that are used for reading or transmission of parameters to and from a POU. Three types are defined: VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT. You can use these variables when you program a POU that will be called from another POU. If a variable in the POU that is called is declared as VAR_INPUT, the call from the other POU can contain data that will be used in execution of the POU that was called. By declaring a variable as VAR_OUTPUT, the POU that is called can return values back to the POU that made the call. Example 6.21 shows the variable declarations to the functional block CTU (Count Up). This FB performs counting of positive flanks, that is, 0–1 transitions in a signal/variable. (The program code for the counter is not included here, but a symbol for the functional block is shown for the sake of illustration.)

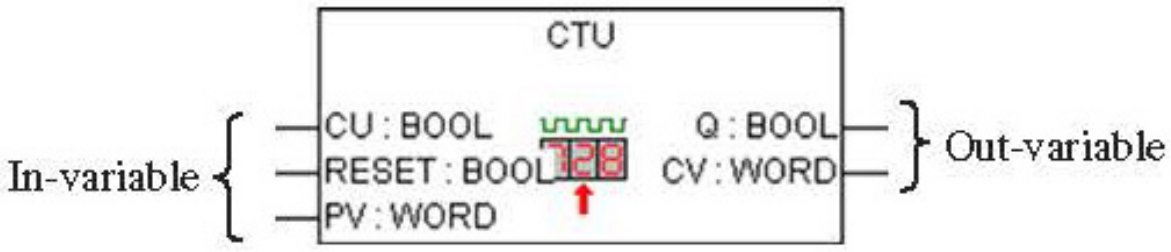


Figure 1.32: CTU Module

As we see, the declaration contains variables of both types VAR_INPUT and VAR_OUTPUT. A call of this FB must therefore contain three variables (arguments); these will be coupled to the variables CU, RESET, and PV. The state Q and current value CV will be returned to the POU that the call came from. In addition to these two variable types, there is a type

```
FUNCTIONBLOCK CTU
                    (* CV increases by 1 each time CU has a rising flank.  *)
                    (* Q becomes TRUE when CV reaches the value of PV. *)
  VAR_INPUT         (* Declares input variable:  *)
    CU              :  BOOL; (* Count Up *)
    RESET           :  BOOL; (* Sets counter value CV to 0 *)
    PV              :  WORD; (* Desired quantity *)
  END_VAR
  VAR_OUTPUT        (* Declares output variable:  *)
    Q               :  BOOL; (* Count Up *)
    CV              :  WORD; (* Current Value *)
  END_VAR
  VAR               (* Declares a local variable:  *)
    M               :  BOOL; (* Count Up *)
  END_VAR
  :
  :                 (* Program code for the function block *)
  :
END_FUNCTION_BLOCK
```

of variable that functions as both an input variable and an output variable simultaneously. This type of variable is declared within the POU by the keyword VAR_IN_OUT. In using this type, the called POU will not only receive values from external variables, as with the use of VAR_IN, but will also receive the actual memory location. In other words, the called POU can change the value of the variables that were used in the call.

Variable Addressing

Even though the standard introduces variables, the standard still permits use of direct addressing, that is, reference to specific memory regions. This can take place in one of two ways: either by using addresses directly in the program or by assigning symbolic names to the addresses in the declaration field. In other words, conventional addressing is still possible. The structure for how a data element is addressed is shown in the next Table. As we see in the Figure, all addresses start with a percent sign (%),followed by a location prefix (a letter). The location prefix indicates whether the memory region is associated with inputs (I), outputs (Q), or an internal memory (M). Next follows a size prefix that indicates the length of the storage location that the address refers to. This is indicated by X, B, W, D, or L for 1, 8, 16, 32, or 64 bits, respectively. This has nothing to do with data types directly, in the sense that the prefix only indicates the size of the storage area and not which type of data it is possible to store there. Let us consider the example in figure

Using direct memory addresses can be a little risky. Aside from the legibility of the code becoming significantly worse than with the use of variables or symbolic addresses, there is the risk of referring to memory locations that overlap one another.

For example, there is the Boolean address %M4.15 for the MSB in the address %MW4, and the address %MW50 contains the two-byte addresses %MB100 and %MB101.

%	1st prefix	2nd prefix	Specific location	Meaning
	I			Input
	Q			Output
	M			Memory
		None or X		Boolean : 1 bit
		B		Byte : 8 bit
		W		Single word : 16 bit
		D		Double word : 32 bit
		L		Long word : 64 bit
			u, v, w, x, y	Hierarchically arranged location Possible meanings: u - rack, v – module, w – channel, x – word, y – bit

Figure 1.33: Addressing structure for direct representation of data elements

Address	Meaning
%MX0.0	Bit 0 (LSB) in memory location 0
%M0.0	----------"----------
%MB8	Memory byte 8
%MW12	Memory word 12
%MD45	Double word at memory location 45
%ML14	Quadruple word at memory location 14

Figure 1.34: Address examples

Figure 1.35 illustrates this concept.

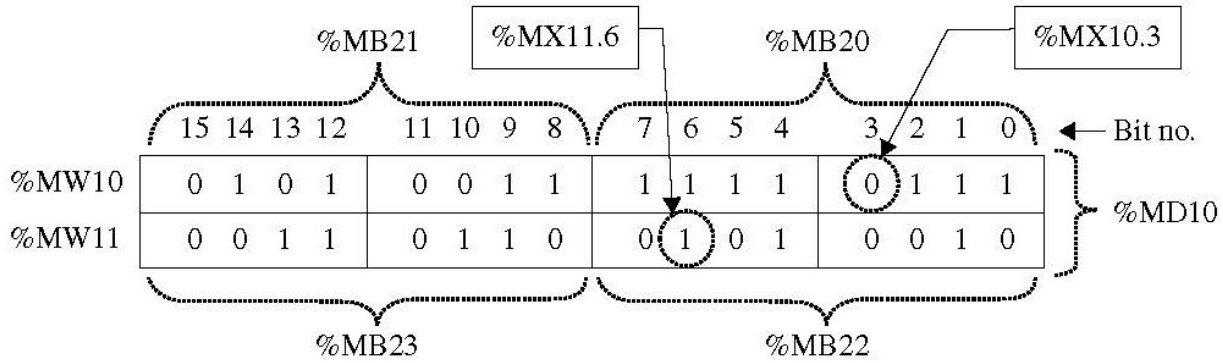


Figure 1.35: Illustration of overlapping between bit, byte, word, and double word

Here, there are represented two memory locations, word no. 10 (%MW10) and word no. 11 (%MW11). Each location is 16 bits in range, where each individual bit can be addressed.

This overlapping of references to memory regions means that there will be addresses that cannot be used in programming. If you use direct addressing, it is therefore smart to have a consistent system for the use of memory: For example, you can decide that memory locations 0 through 4 are used for Boolean objects (this gives 5 * 16 = 80 accessible objects) and that locations 5–19 will be used for bytes, 20–29 for memory words, and so forth.

If several double and long (quadruple) word addresses are to be used, it is important to skip over some memory locations so that overlap does not occur. For example, you can avoid using double-word addresses sequentially. That is, you would not simultaneously use the addresses %MD0, %MD1, and %MD2, but rather skip every other location and, for instance, use %MD0, %MD2, and %MD4. The reason for this is that the double-word %MD4 includes the word addresses %MW4 and %MW5, and the double-word %MD5 includes the words %MW5 and %MW6. Word address %MW5 is therefore contained in both of the sequential double-word addresses and will therefore be overwritten by both of the subsequent double addresses. (Similarly, neither can the address %MW5 be used.)

Figure 1.36 illustrates this.

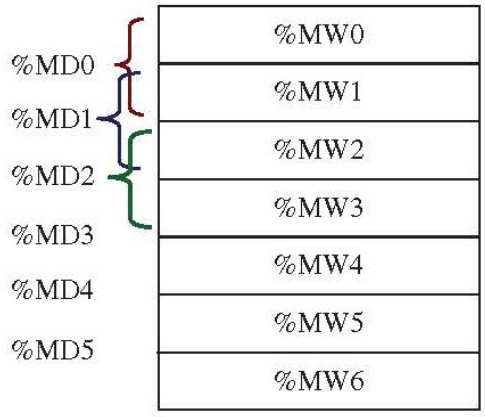


Figure 1.36: Illustration—address overlapping

I/O Addressing: If the address refers to data elements in input or output memory, it is also necessary to specify which input or output it applies to. This is done by adding some numbers after the prefixes. The numbers can, for example, indicate module number and channel number. The actual structure of this location reference is implementation dependent, but it is a requirement that it have a hierarchical structure. This means that the number farthest to the left indicates the highest level in the address structure with successively lower levels continuing toward the right.

%IX1.5	Digital input. The numbers can represent channel 5 in module 1
%Q2.4.12	Digital output, for example, in rack 2, module 4, channel 12
%IW12	Input word no. 12, for example, an analog input
%IW3.2	Analog input, for example, channel 2 in module 3
%QW5.2.4.7	Output word. Network address 5, rack 2, module 4, channel 7

Figure 1.37: Example: Direct I/O Addressing

Variable versus I/O-Addresses

Direct addressing can naturally be used together with variables by having an identifier connected to a particular data element. This can also be done for internal memory locations, but it is most practical and absolutely necessary to associate variables with the inputs and outputs of the PLC. Such a connection between fixed addresses and identifiers can be obtained by using the keyword AT in the declaration.

As an example we see here that the Boolean input address %IX2.4 is assigned to the variable name Dig_in, while the Boolean output address %QX3.5 is assigned to the variable Dig_out. A_in and A_ out are variables associated with an analog input and an analog output, respectively. Notice that it is permitted to assign an object with a "shorter" data

```
VAR
   Dig_in AT %IX2.4 :  BOOL := TRUE;
   Dig_out AT %QX3.5 :  BOOL;
   A_in AT %IW3.2 :  WORD;
   A_out AT %QW4.1 :  WORD;
END_VAR
```

type to an object of a "longer" data type. The basis for this is that memory locations are only storage locations and as long as the memory location has room (enough bits) to store the object, everything will go well. On the contrary, the assignment of an object that has a data type that requires more space than the assigned memory location can provide is naturally not permitted. This will therefore trigger a syntax error in the compiler.

Unspecified I/O Address Sometimes, one would like to declare variables that are to be assigned to I/O without specifying their exact addresses. When you are programming a FB or a program that is to be reused in another context or if you are programming in accordance with the requirements of the client, you will not know the exact address that the variables should be assigned to. This information is not available until all I/O modules have been installed and configured.

In such cases, you can undertake a partial assignment by specifying how the variables are to be mapped to an I/O, without stating the actual address. When you or others at a later time are going to use the program, the address can be specified at the configuration level by use of the keyword VAR_CONFIG.

As an example the following declaration shows how one assigns incomplete I/O addresses by use of AT and %I* or %Q*. (Note: Even though the example shows declaration in a program, it is more useful for function blocks for reasons that we will come back to the next chapter.)

```
PROGRAM Whichio
   VAR
    Input AT %I* :  BOOL;
    Output AT %Q* :  BOOL;
END_VAR
```

When you have installed all the I/O modules and configured them, then you can finish the application in by adding a new Global Variable List (GVL), where you specify the exact addresses (note the reference to the current POU (Whichio) where the variables will be declared and used):

```
VAR_CONFIG
   Whichio.Input AT %IX2.3 :  BOOL;
   Whichio.Output AT %QW3.1 :  WORD;
END_VAR
```

Multielement Variables

Declaration of multielement variables is done in a similar way to declaration of simple variables. As mentioned previously, arrays can be declared directly with declaration of variables. It is not necessary to define a data type for arrays, even though this is possible. Example shows both forms: direct declaration and declaration based upon predefined data types. The variables younameit and sowhat are declared on the basis of previosly declared array data types, while the variables Oddnum and Values are declared directly as arrays

```
TYPE
   Two_dim :  ARRAY [1..2, 1..5] OF INT := [10, 20, 30, 40, 6(50)];
END_TYPE


TYPE
   Three_dim :  ARRAY [0..3, 2..5, 1..4] OF REAL;
END_TYPE


VAR
   Oddnum :  ARRAY [0..9] OF SINT := [1, 3, 5, 7, 11, 13, 17, 19, 23, 29];
   Values :  ARRAY [1..3, 1..4] OF INT;
   younameit :  Two_dim;
   sowhat :  Three_dim := [10(3.14)];
   whoops :  ARRAY [1..2, 1..2, 1..2] OF BOOL := [0, 0, 1, 0, 1, 1, 0, 1];
END_VAR
```

Arrays: The example above shows how arrays can be initialized (be given an initial content). This can be done either by defining data types or, more commonly, by declaration of a variable. The advantage of the latter method is that several variables can use the same data type but have different initial values.

Because initial values must be entered, all of the elements in the array will be given default values for the data type in question. This is the case with the variable Values in the example.

It is also possible to partially assign initial values. The variable sowhat (data type Three_dim) is declared with initial values for only the first 10 elements. The remaining 54 elements are automatically set to 0. The syntax that is used for giving specific values to the first 10 elements in the variable sowhat is in line with what the standard recommends that manufacturers implement in order to give several successive elements the same value: `NumberOfElements(value)`.

Refer again to the Example, where six elements in the data type Two_dim are set equal to 50. Such a repetition factor can also be used to initiate more complex numerical sequences. For example, 3(2,5,7) is the same as 2, 5, 7, 2, 5, 7, 2, 5, 7.

Arrays are well suited to read, transfer, store, and use large quantities of data in a structured simple way. In the program code, one can read from and write to individual elements in arrays by indicating the index of the element in question. For a variable called My_table,

declared as a one-dimensional array, this syntax is: `My_table[i]` where i gives the element number.

In what follows it is shown the syntax for assigning new values to all the elements in the arrays Oddnum, younameit, and whoops that were declared. (Note: The array elements can naturally be assigned variables instead of values.) Note that for multidimensional arrays, that is, arrays that have more than one set of array boundaries, the indices are specified in a significant order, where the array boundaries farthest to the left have the highest significance. In other words, the first index varies slowest and the last index varies fastest when we move through the array.

Indirect Addressing: Many traditional PLC systems operate with indirect addressing for efficient management of large quantities of data. In indirect addressing, the actual address location depends upon the value of the variable (index). Using arrays covers this similarly by indexing the array elements indirectly, such as here:

```
num1 := 3;

num2 := 8;

num3 := 122;

younameit[num1, num2] := num3;
```

Here, the element younameit[3,8] is set equal to 122.

```
Values[2,3] := 5*younameit[1,4] - Oddnum[6] + 300;
```

This shows an arithmetic expression in the ST programming language where the array element `Values[2,3]` gets the value `5*40 - 13 + 300 = 487`.

Data structures: Data structures are powerful data types because the program code can be built up in a highly structured way by declaration of carefully structured data types. The syntax for declaring variables based on structured data types is the same as for variables based on the elementary data types

```
TYPE
   Camera   : (OK, LabelError, Leakage);
END_TYPE
TYPE Productdata :    (* Name of the structured datatype *)
   STRUCT
      PictureResult : Camera;          (* Sub-element *)
      Weight        : REAL;            (* ------"------- *)
      ID            : UINT (0..10000); (* ------"------- *)
   END_STRUCT
END_TYPE
```

And the variable can be declared as follows

```
VAR
   M, N : UINT;;
   Product :  ARRAY[1..100] OF Productdata;
END_VAR
```

Here, a variable is being declared as a one-dimensional array of type Productdata. Each element in the array will therefore consist of three subelements: PictureResult, Weight, and ID.

It is fully possible to access subelements in a data structure. In this way, one can read values and write in new values. The syntax is as follows:

```
FOR m:=1 TO 100 DO
   IF Product[m].PictureResult = OK THEN
    IF (Product[m].Weight > 240.0) AND (Product[m].Weight < 260.0) THEN
      OK_Product[m] := Product[m].ID;
    END_IF
   END_IF
END_FOR
```

5.8 Functions

A function is defined as a POU that yields the same result every time it is called (executed). This implies that a function does not have any memory. The result from a function call is most often one single value, but it can also be a matrix or a structure of many values if the input argument is of such a data type. Examples of standard functions are SIN (sine), COS (cosine), SQRT (square root), ADD (add), and SHL (shift left).

Many of the functions that are defined in the standard belong to the group of operators in the Structured Text (ST) programming language and using the function is therefore called performing operations on an operand. Example: Here, SQRT is an operator and B is an

```
A := SQRT(B).
```

operand. The answer is stored in A. Many of the standard functions also have their own operator symbols that are used in ST instead of the function names. This includes + (ADD), $*$ (MUL), and $\geq$ (GE).

Operators and other standard functions will normally be implicitly recognized by the development tool. If the tool does not recognize a particular function, this may mean that it belongs to a library that must be associated with the project.

The structure of a function is the same as that of programs and Function Blocks (FB, we will see in the next section the structure of this POUs): At the top, there is a declaration field, and below, there follows a program code field (implementation field). The declaration field, naturally enough, will contain declarations of all variables that are used in the code. This takes place in the same way no matter which programming language is used in the program code field. Next Figure shows the use of functions in both the text-based language ST and the graphical language FBD.

The functions that are used in the example are the arithmetic function MUL (multiply); the comparison function GE (Greater than or Equal to $\geq$); the bit-string functions AND, OR, and NOT; and the type-conversion function WORD_TO_BOOL.

The example shows that the use of functions does not have to be declared. This is an indirect result of their not having any memory. This does not apply to FBs, so here an instance of the FB RS is declared (more about this later).

Layout and design of the graphical representations of the functions can vary from one tool to another, but some requirements are imposed by the standard. Some of these require that the blocks should be square or rectangular, signal flow should be from left to right, the function name or symbol should appear within the block, and the $\circ$ should be used as an inversion symbol.

Standard Functions

Next Figure shows standard functions that are common to all programming languages. These functions can also be implemented as operators in ST. For functions where the standard does not specify such operator symbols, the symbols are given in parentheses following the function names.

The practical difference between using operators and function names is that the use of function names requires that operands3 must be given as arguments in parentheses follow-

```
PROGRAM SomeFunctions
VAR // Declaring variables and an instance of the FB RS.
  On, Off, Light  : BOOL;
  A, B            : WORD;
  OneFB           : RS;
END_VAR

(***** Program code in ST: *****)

(* Instruction that uses several standard functions: *)
On := WORD_TO_BOOL(A OR NOT B) AND ((A*B) ≥ 5);

(* Call of function block oneFB: *)
OneFB(Set := On , Reset1 := Off, Q1 => Light);

(************************************************)

(***** Program code in FBD: *****)
```

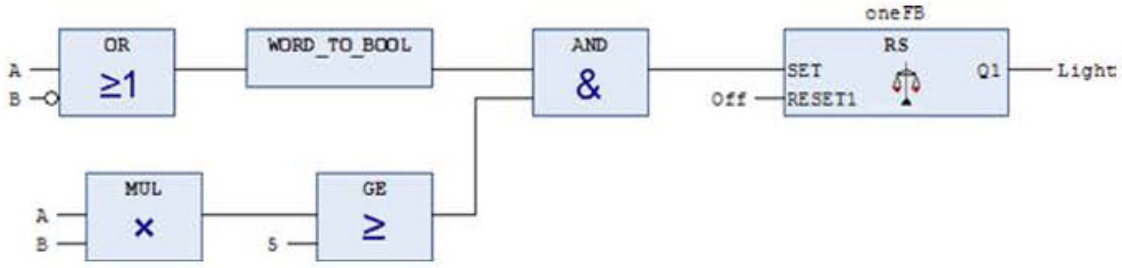


Figure 1.38: Use of Functions in ST and FBD

ing the name. Some functions perform operations between operands, but others perform operations on one or more operands, for instance, LOG(A) and MAX(A,B,C).

Assignment

One of the simplest instructions we can perform is to set the content of one variable equal to the content of another variable.

Suppose that you wish to activate a pump connected to the digital output %Q2.3 when a sensor connected to the digital input %I1.8 gives a logical high value.

Function type	Function name	Comments
Arithmetic	ADD (+), MUL (*), SUB (–), DIV (/), MOD, EXPT (**), MOVE (:=)	MOVE is used for assignment in LD and FBD
Numerical	ABS, SQRT LN, LOG, EXP SIN, COS, TAN, ASIN, ACOS, ATAN	General Logarithmic Trigonometric
Bit-string operations	SHL, SHR, ROR, ROL AND (&), OR, XOR, NOT	Bit shift operations Boolean operations
Selection	SEL, MUX, MAX, MIN, LIMIT	
Comparison	GT (>), GE (≥), EQ (=), LE (<), LT (≤), NE (≠)	
Type conversion	Syntax: **Type1_TO_Type2** Examples: INT_TO_REAL, INT_TO_WORD, BOOL_TO_STRING REAL_TO_INT, STRING_TO_TIME, DT_TO_STRING + many others	Which conversions are supported depends upon implementation
Text-string operations	LEN, LEFT, RIGHT, MID, CONCAT, INSERT, DELETE, REPLACE, FIND	

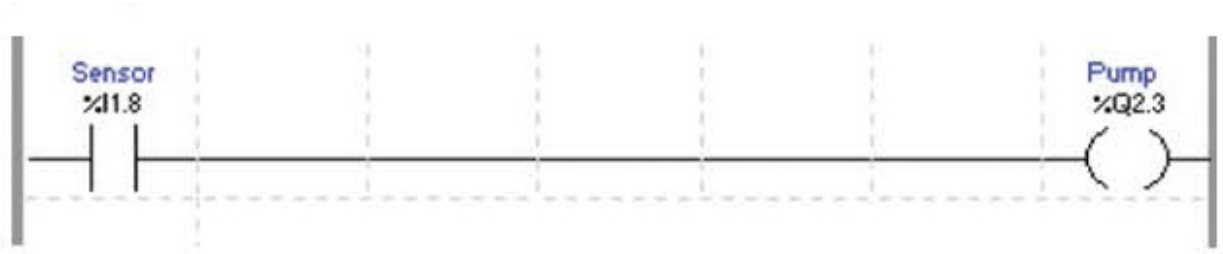
Figure 1.39: Standard Functions

```
%Q2.3 := %I1.8.
```

Or, with symbols and variables

```
Pump := Sensor.
```

The corresponding instruction in LD would look about like this:

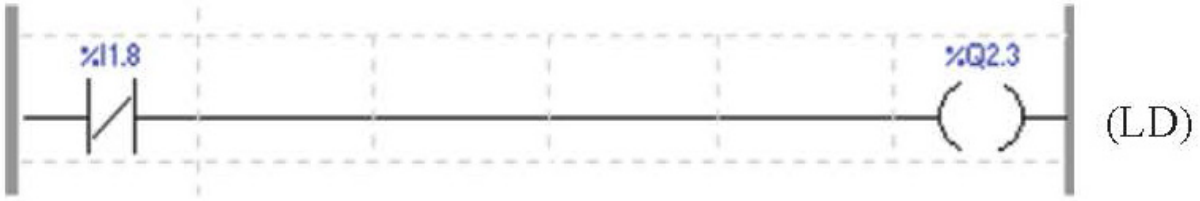


This is called assignment, and such instructions are used to transfer the content on one operand to another. The operands do not need to be of the BOOL data type; they may be any data type, including structured and array.

The operator := belongs to the group of arithmetic functions, and the function name is MOVE. In the graphic languages FBD and LD, the function is represented by a single graphic block with one input and one output:

Boolean Operations

This group covers use of the operators AND, OR, XOR, and NOT. Above, we assigned a Boolean address to another Boolean address. If the output address is to have the value TRUE when the input address has the value FALSE, we would have had to program the following (in LD and ST, respectively): By using the functions AND, OR, XOR, and NOT, we can




```
%Q2.3:=NOT %I1.8;
```

(ST)

implement all common Boolean operations. Note that the operators can also be used for bit strings such as BYTE and WORD.

The functions are expandable with respect to the numbers of operands and inputs: A

```
%Q2.0 := Var1 OR Var2 OR (%I1.0 AND %I1.1 AND NOT %MX5.2);
```

graphic representation in FBD/LD will generally look like the following (the function NOT can have only one input value):

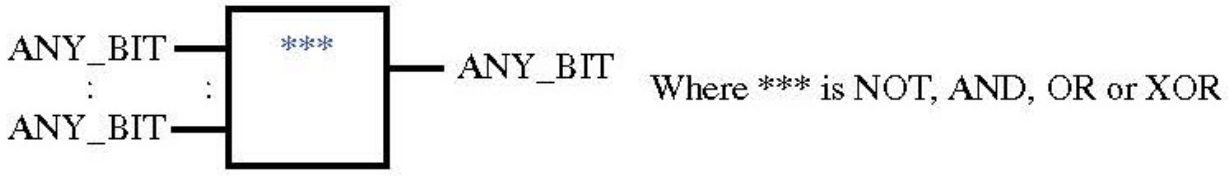


Figure 1.40: Boolean graphical Block

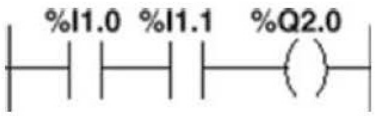
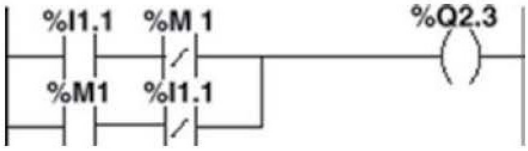
Function	LD	ST
AND (&)		%Q2.0 := %I1.0 AND %I1.1;
OR		%Q2.3 := %I1.1 OR %M1 ;
XOR		%Q2.3 := %I1.1 XOR %M1 ;

Figure 1.41: Boolean Functions Examples in different languages

Arithmetic Functions

This group of operators and functions is used to perform arithmetic operations between two operands. Figure 1.42 shows an overview. The graphic representation that is used with these functions in FBD or LD is generally the same for all these functions. They differ only in the function name or the operator symbol in the box. IN1 and IN2 can be numbers or

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

Figure 1.42: Arithmetic Functions

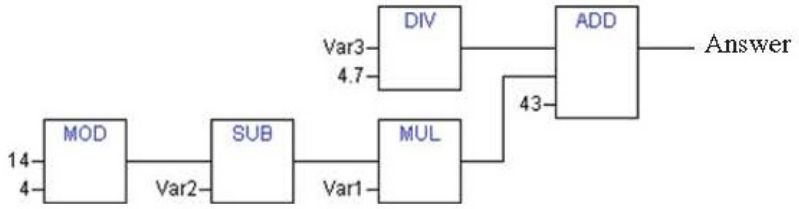
Figure 1.43: Arithmetic function block

variables of the types ANY_NUM or ANY_BIT (generic numerical or bit data type). The functions ADD and SUB also apply with the data type TIME. The symbol above shows only two inputs, but the functions ADD and MUL can be used for an arbitrary number of inputs, while the function MOVE has only one input.

An example

```
Answer := 43 + Var1*((14 MOD 4) - Var2) + Var3/4.7; (* ST-code:  *)
```

```
(* Same expression in FBD code: *)
```




```
(* … and in LD code: *)
```

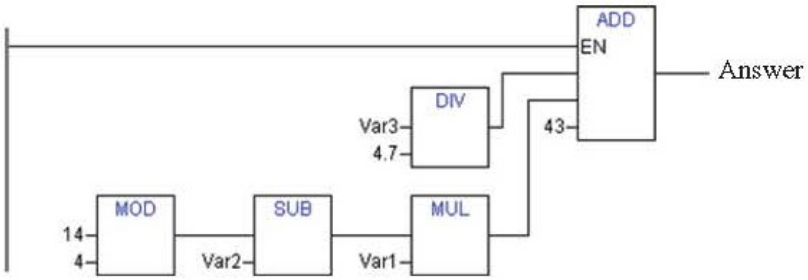


Figure 1.44: FBD and LD Code Numerical function implementation

As we see, the LD code and the FBD code are nearly identical here. The only difference is that the code in LD must be triggered (activated) via an Enable input, since LD is based on the use of a "conductor rail" on the left side (see next EN/ENO). (MOD stands for modulo, which is an operation that yields the remainder from division. Here, the operation to be performed is 14 MOD 4. 14 divided by 4 equals 3 with 2 remainder. The result is thus 2.)

Overflow

Be aware of the risk of overflow when you are working with arithmetic and numerical instructions. If the result is stored in operand of data type INT, for example, the result must lie within the limits of -32768 and +32768. If the result lies outside these limits, you have overflow. The values that are stored in the result can have any value at all, and this is naturally undesirable. The programmer must take care that the resulting values are asserted to a maximum (or possibly minimum) if overflow has occurred. If the danger of overflow is present, you should naturally evaluate the use of other data types or alternative code in a different way. Some PLCs have system addresses that monitor for overflow, among other things.

5.9 Comparison

The comparison operators are used to compare the values of two or more operands. The syntax in ST is as follows (operator is one of those listed in Figure 1.45):

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

Figure 1.45: Comparison Operators

```
Out := OP1 operator OP2 operator OP3 operator ...  operator OPn;
```

The graphic representation that is used with these functions in FBD or LD is, generally, the same for all these functions. They differ only in the function name or the operator symbol in the box. The inputs (operands) IN1 and IN2 can be numbers or variables of a data type

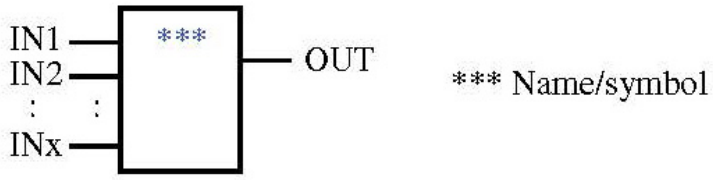
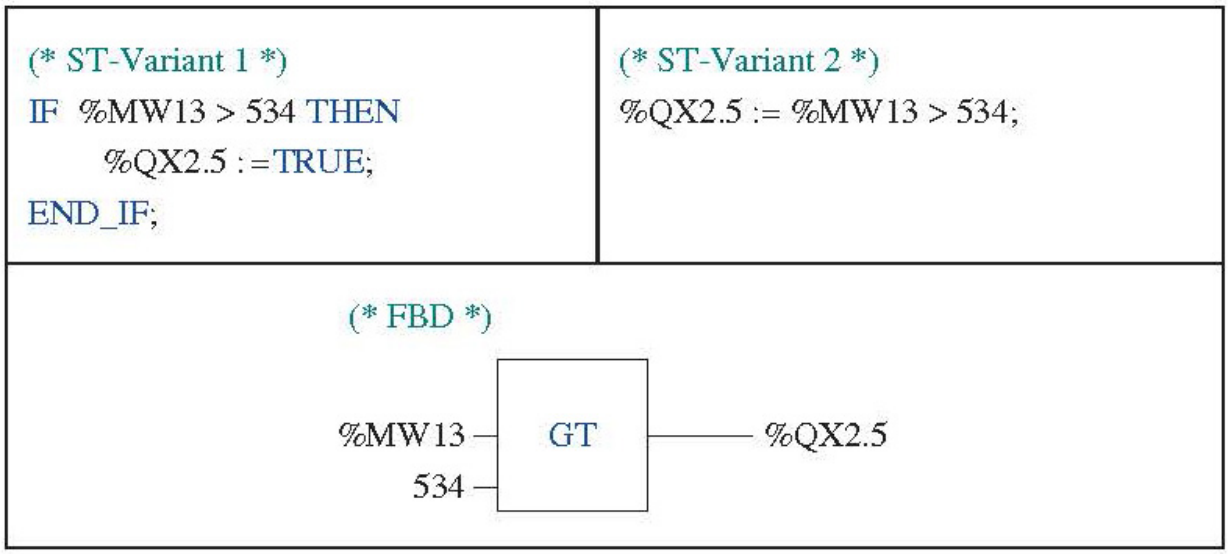


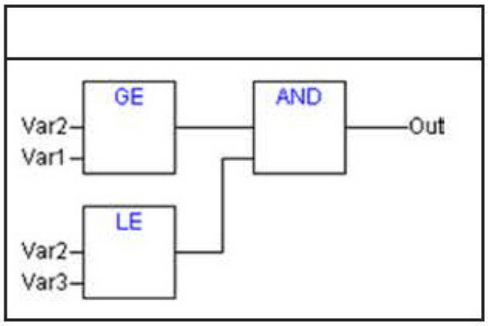
Figure 1.46: Comparison Operator, graphical representation

that belongs to the class (generic type) ANY_ELEMENTARY. When using functions on bit strings of differing lengths, the length of the shortest string is increased by filling with zeros from the right. From the descriptions in the tables, we can understand that the order of the operands is important, just as it is in connection with the inputs to the graphic blocks. For example, if you want the output signal to be TRUE when Num1 is greater than Num2, you must place Num1 at the top of the block GT.

An example: the codes in ST below show two different possible codes for the same instruction, which is to set a Boolean output to TRUE if the result of a comparison is true. The same instruction is also shown in FBD code. We see that variant 2 of the ST code is



directly comparable with the use of the function in a graphic language. This is a compact way of testing the output of a comparison that is worthwhile noting because it saves a little coding in ST. Below, there is an example of comparison with more than two operands.

Structured text	Graphic language
Out := (Var1≥Var2) AND (Var2≤Var3);	

Numerical Operations

We have previously examined a group of functions and operators that are used for arithmetic calculations. The functions that are presented here are also used for calculations, but what is special about these is that they perform operations on a single operand, rather than between operands. The syntax in ST is as follows:

```
OP1:  = Function (OP2)
```

Here is a possible graphical representation in FBD/LD. Name is the name of the function.

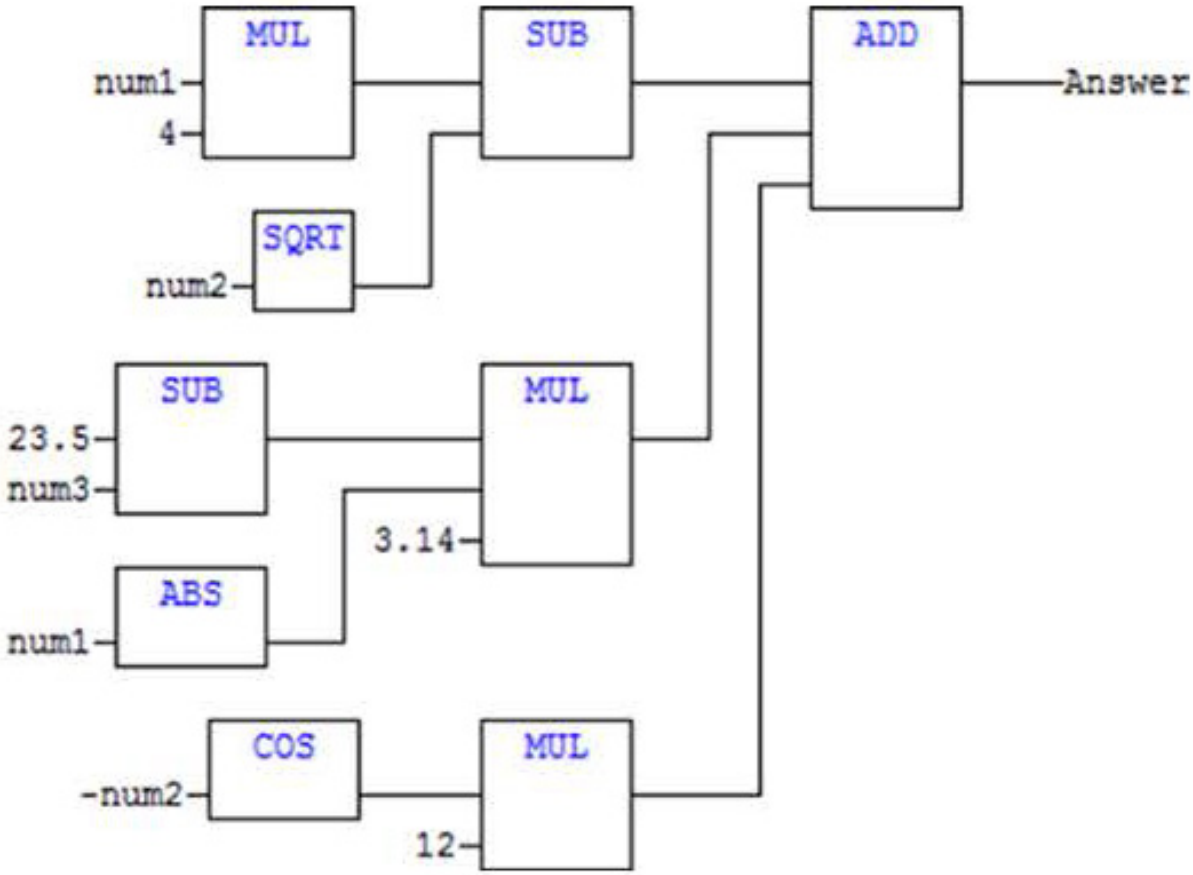
Function name	Data type	Description
ABS	ANY_NUM	Absolute value
SQRT	ANY_REAL	Square root
LN	----"----	Natural logarithm
LOG	----"----	Base-10 logarithm
EXP	----"----	Natural exponential (e^x)
SIN	----"----	Sine (radians)
ASIN	----"----	Arc sine (inverse sine)
COS	----"----	Cosine (radians)
ACOS	----"----	Arc cosine (inverse cosine)
TAN	----"----	Tangent (radians)
ATAN	----"----	Arc tangent (inverse tangent)

Figure 1.47: Numerical functions

Numerical expressions are most often used in combination with numerical operations, Boolean instructions, and arithmetic operations, in addition to comparison operations. There is no limit to the number of operators and operands that can be used. Furthermore, an arithmetic sign can be placed in front of an operand or an operation on a single operand without needing to use parentheses.

As an example, look at the following expression written in ST ad Graphical Language

```
(* ST-code:  *)
    Answer        :=  num1*4 - SQRT(num2) + ABS(num1)*3.14*(23.5 - num3)
                      + 12*COS(-num2);
(* FBD/LD code:  *)
```

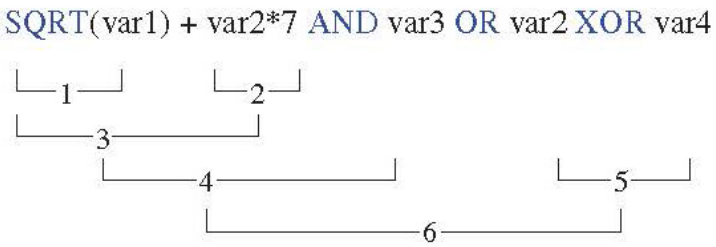


The priority of execution in the evaluation of a numerical expression is:

1. Parentheses;
2. Operations performed on an operand, for instance, SIN(X) or ABS(Y);
3. Negation (-) and complement (NOT);
4. $*$, /, MOD;
5. $+$, -;
6. $<$, $>$, $\leq$, $\geq$;

7. $=$, $\neq$;
8. AND;
9. XOR;
10. OR.

In order to control the order in which the operations are to be performed, one must therefore use parentheses. If you are in doubt, it is much better to use many parentheses than to risk an erroneous result from a calculation. At any rate, there is no limitation on the number of parentheses.



Type Conversion

By far, the largest group of functions is those that perform conversion between different data types. To begin with, one can convert between all the elementary data types (ANY_ELEMENTARY), even though in practice there is seldom or never a requirement for some of these conversions.

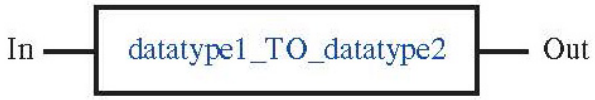
Conversions are most often used in numerical calculations where different data types occur in the expressions. The reason for this is that analog input values often are stored in addresses of the WORD type and if these are going to be used in numerical expressions with a high requirement for accuracy, there is a need for conversion to floating-point numbers.

Note that you generally should not convert from a larger to a smaller data type, for instance, from WORD to BYTE or from DINT to INT. In the worst case, this results in an error, and in the best case, you run the risk of losing information.

Syntax for the conversion functions in structured text is as follows:

```
Out := datatype1_TO_datatype2(In);
```

In the graphic languages FBD and LD, the symbol will be a single block with one input and one output: We are not going to provide examples of all possible conversions because this



would be a too lengthy task.

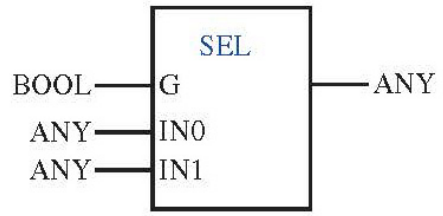
Instead, we will show a selection of examples (in ST).

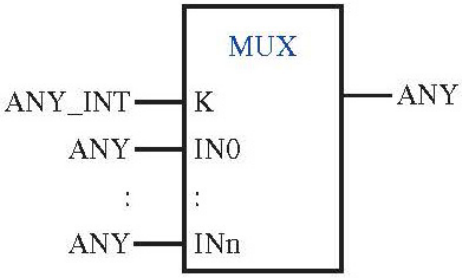
```
B   :=  BOOL_TO_INT(TRUE);       (* Result:  1 *)
O   :=  BOOL_TO_STRING(TRUE);    (* Result:  'TRUE' *)
RI  :=  BOOL_TO_TIME(TRUE);      (* Result:  T#1ms *)
N   :=  BOOL_TO_TOD(TRUE);       (* Result:  TOD#00:00:00.001 *)
G   :=  BOOL_TO_DATE(FALSE);     (* Result:  D#1970-01-01 *)
J   :=  REAL_TO_INT(7.5);        (* Result:  J = 8 *)
A   :=  REAL_TO_INT(7.4);        (* Result:  A = 7 *)
C   :=  REAL_TO_INT(-7.5);       (* Result:  C = -8 *)
K   :=  REAL_TO_STRING(35.27);   (* Result:  K = '35.27' *)
B   :=  TRUNC_INT(-23.6);        (* Result:  B = -23 *)
B   :=  REAL_TO_INT(-23.6);      (* Result:  B = -24 *)
```

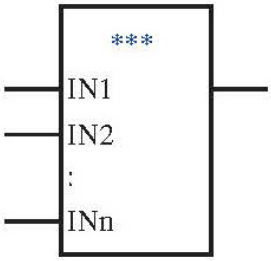
Selection

This is a special group of functions with the common feature that one of the inputs is assigned to the output. The difference among the selection functions is what criterion is used to select among the inputs. See the Figure.

Name	Graphic symbol	Explanation and example

SEL		Use it to select which inputs that shall be assigned to the output:
		G = False (0) gives Out := IN1
		G = True (1) gives Out := IN2
		Example:
		`Out := SEL(1, A, B)`
		`(*Gives Out = B *)`

MUX		Multiplexing resembles SEL, except that MUX is used to select one out of many inputs. Which input is selected is determined by the integer value of K (0, 1, 2, 3,...)
		Example:
		`Out := MUX(0, A, B, C, D)`
		`(*Gives Out = A *)`

MAX
MIN
		*** - MAX or MIN
		These functions are used to select which of the inputs has the largest or smallest value
		Example:
		`Out := MAX(5, 14, 8)`
		`(*Gives Out = 14 *)`

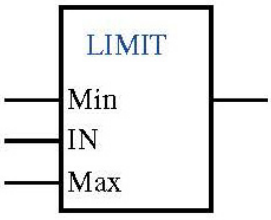
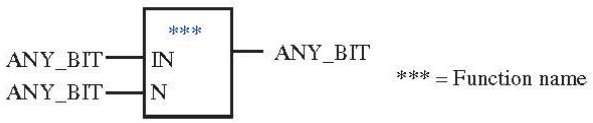
LIMIT		LIMIT is a limiter of values. The user provides a lower limit (Min) and an upper limit (Max). If the value of (IN) exceeds the upper limit, the value Max is returned. If the value of (IN) is below the lower limit, the value Min is returned. Example: `MV_out := LIMIT(0, MV, 32767)`
	Permitted data types are ANY_ELEMENTARY	

Figure 1.48: Functions for selection

Bit-string functions

These are a group of classic functions that are used for everything from control of stepping motors to monitoring of states. Most people will connect these functions with shift registers. The standard defines 4 functions that all are based on shifting bit strings a desired number of bits toward the left or toward the right. These are the functions SHL, SHR, ROL, and ROR.

The structure of the input arguments is equivalent for all four. (Even though these func-



tions can be used for all data types belonging to the generic ANY_BIT, there is, of course, little meaning to using them on Boolean objects.)

Function name	Description
SHL	Shift left: Execution of the function implies that the content in a bit string is shifted N places to the left, with the vacancies being filled by zeros
SHR	Shift right: Execution of the function implies that the content in a bit string is shifted N places to the left, with the vacancies being filled by zeros from the left. If N is greater than the number of bits in IN, the result is an empty bit string. (The same is true for the function SHL)
ROL	Rotate left: The content of a bit string is shifted N places to the left when the function is executed. Bits that fall out on the left side are filled in again on the right side
ROR	Rotate right: The content of a bit string is shifted N places to the right when the function is executed. Bits that fall out on the left side are filled in again on the left side

Table 1.8: Bit String functions

```
PROGRAM Shift
VAR
  bait   : BYTE := 2#10110100;                    (* = B4 Hex *)
  ord    : WORD := 2#0000000010110100;            (* = B4 Hex *)
  bait_shl, bait_shr, bait_rol, bait_ror :  BYTE;
  ord_shl, ord_shr, ord_rol, ord_ror :      WORD;
  n      : INT := 2;
END_VAR

bait_shl  := SHL(bait, n);
ord_shl   := SHL(ord, n);
bait_shr  := SHR(bait, n);
ord_shr   := SHR(ord, n);
bait_rol  := ROL(bait, n);
ord_rol   := ROL(ord, n);
bait_ror  := ROR(bait, n);
ord_ror   := ROR(ord, n);
END_PROGRAM
```

Result from run:

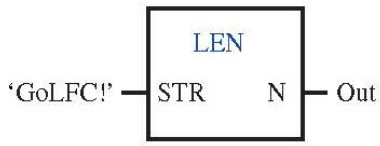
```
bait_shl = 2#11010000
ord_shl = 2#0000001011010000

bait_shr = 2#00101101
ord_shr = 2#0000000000101101

bait_rol = 2#11010010
ord_rol = 2#0000001011010000

bait_ror = 2#00101101
ord_ror = 2#0000000000101101
```

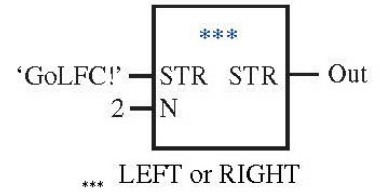
Text to String Functions

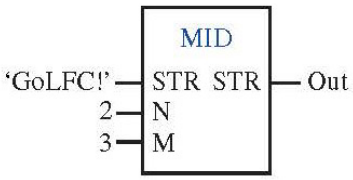
Several of the functional groups that we have presented so far can also be used on text strings, that is, the data types CHAR/WCHAR and STRING/WSTRING. This applies to selection functions, functions for type conversion, and functions for comparison. The groups of functions that are reviewed here are specially designed for use on text strings. These are the functions LEN, LEFT, RIGHT, MID, CONCAT, INSERT, DELETE, REPLACE, and FIND.

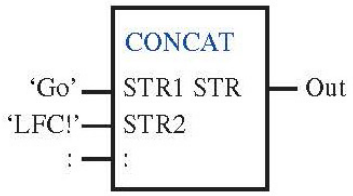
Figure 1.49 contains possible graphic symbols and explanation and examples of the use of the functions. Because of lack of space in the table, some abbreviations are used: STR, STR1, and STR2 have data type ANY_STRING, and N and M are ANY_INT.

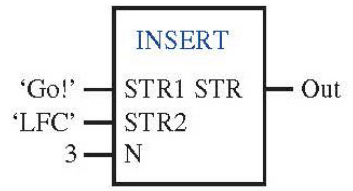
Name	Graphic symbol and example	Explanation and example in ST

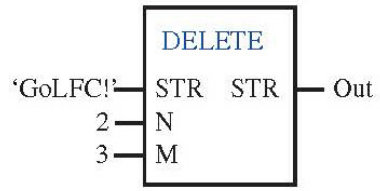
LEN		Returns the length of a text string (number of characters in the string)
		Example:
		`Out := LEN('GoLFC!')`
		`(*Gives Out = 6 *)`

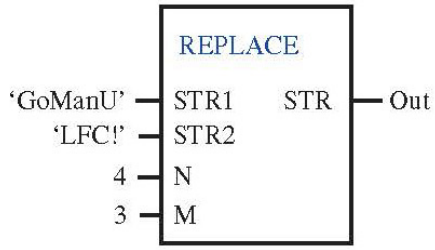
LEFT
RIGHT
		Returns a desired number of char (N), starting from the left or right
		Examples:
		`Out := LEFT('GoLFC!', 2)`
		`(*Gives Out = 'Go' *)`
		`Out := RIGHT('GoLFC!', 4)`
		`(* Gives Out = 'LFC!' *)`

MID		Returns a desired number of characters, N, starting with character number M from the left
		Example:
		`Out := MID('GoLFC!', 2, 3)`
		`(*Gives Out = 'oL' *)`

CONCAT		The function performs a concatenation of several text strings
		Example:
		`Out := CONCAT('Go', 'LFC!')`
		`(*Gives Out = 'GoLFC!' *)`

INSERT		The function is used to insert a text string (STR2) into another text string (STR1) in the position following character number N
		Example:
		`Out := INSERT('Go!', 'LFC', 3)`
		`(*Gives Out = 'GoLFC!' *)`

DELETE		The functions used to delete And characters from a string, beginning with character number M from the left
		Example:
		`Out := DELETE('GoLFC!', 2, 3)`
		`(*Gives Out = 'GFC!' *)`

REPLACE		The function replaces N characters from the string STR1 with the string STR2, beginning with character number M from the left
		Example:
		`Out := REPLACE('GoManU', 'LFC!',`
		`4, 3)`
		`(*Gives Out = 'GoLFC!' *)`

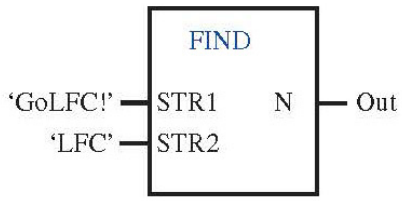
FIND		Returns the starting position for a partial string STR2 in the string STR1. If STR2 is not found in STR1, the value 0 is returned
		Example:
		`Out := FIND('GoLFC!', 'LFC');`
		`(*Gives Out = 3 *)`

Figure 1.49: Text to String Functions

Defining new functions

When making programs for automated processing, you normally get by with the predefined functions and operators provided by the manufacturer, but occasionally, there is a need to define your own functions.

Defining a new function begins with a declaration where the function is assigned an identifier (given a name) and a data type. Following that, the input variables are declared. These are used for conversion of arguments when the function is called. An example

```
(*** Defining the function:  ***)
FUNCTION My_func:  INT              (*Declaration part*)
   VAR_INPUT
   Num :  INT;
   END_VAR

    IF Numb < 0 THEN                (*Implementation part*)
     My_func := -Num;
    ELSE
     My_func := Num;
    END_IF
END_FUNCTION
```

We see that the declaration begins with keyword FUNCTION, followed by the function name and data type. The data type must be consistent with the format of the result of the function call, since the function name also acts as a variable where the result of the instructions is stored. VAR_INPUT is used to declare variables that will receive values transferred in the function call (often called arguments). Here, we have only one such variable or operand, namely, the variable Num.

In the code portion of the function, we have here a control structure of the type IF-THEN ELSE-END_IF. There we check whether the value in the argument is negative. In that case, the value is inverted and returned as the result (My_func := -Num). The definition of the function closes with END_FUNCTION.

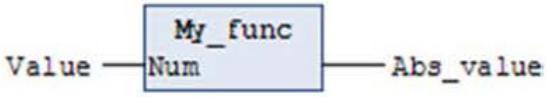
In this example, we use only the variable type VAR_INPUT in the function, but it is also permitted to use the variable types VAR_OUT and VAR_IN_OUT. Internal variables, or variables that are used only within the function itself, can also be declared by using the keyword VAR.

The code below shows how we can use (call) our new function in the program. Here, we show how the code for the call looks in both ST and FBD.

In a graphic language such as FBD, the function will automatically appear as a rectangular box with a line for each of the input variables on the left side and (usually) a line on the right side. Since our function has only one argument and gives only one value as a result, there is only one line on each side. Similarly, in ST, there will be only one operand (the variable Value).

```
PROGRAM PLC_PRG
VAR
   Value      : INT;
   Abs_value  : INT;
END_VAR

(*Function call in ST: *)
Abs_value:= My_func(Value);

END_PROGRAM
```

```
(*Function call in FBD: *)
```



EN/ENO

The syntax for activating the code in the LD programming language is based on the idea that all instructions must be activated by being connected to a "power rail" that graphically is a vertical line at the left of the instructions. This power rail is constantly "TRUE". This implies that the graphics symbols we wish to use in LD must have an Enable input that can be connected to this power rail. We saw this previously in Figure 1.44. Functions that are made for use in LD have such an Enable input called EN. In addition, they have an Enable output called ENO that usually has the same state as EN.

The way this works depends, in part, on implementation but can be as follows:

- Only when EN has the state TRUE (1) the function is called and executed.
- Nothing happens when EN is FALSE (0).
- How ENO works depends partly upon how the manufacturer has chosen to implement the function.

If an error occurs during execution of the function, ENO should be reset to FALSE. If not, then one of the following can be implemented:

1. ENO is like EN at any time. Then the purpose of ENO is just to continue EN to the EN input of the next functions.
2. ENO is set TRUE or FALSE depending upon the result of the execution.

In order to illustrate the principle, we will expand the function we defined in the previous Example by adding EN/ENO so that it can be used in the LD language. (Since the code in this example is so simple, ENO is used here only for continuing EN.)

```
(*** Defining the function:  ***)
FUNCTION My_func:  INT              (*Declaration part*)
   VAR_INPUT
   EN : BOOL := TRUE;
   Num :  INT;
   END_VAR
   VAR_OUTPUT
   ENO: BOOL;
   END_VAR

    IF Numb < 0 THEN                (*Implementation part*)
     My_func := -Num;
    ELSE
     My_func := Num;
    END_IF
    ENO := EN;
END_FUNCTION
```

5.10 Function Blocks

In contrast to functions, function blocks(FBs) can have internal memory. This means that the result of a call to a FB can depend not only upon the arguments (input values) but also on the values of the FB's own internal variables. In other words, repeated calls of a FB with identical arguments do not necessarily give the same output values.

When a function is called from a POU, the function returns a response in the form of a return value (or an array of values). This is not what happens with a call of a FB. Instead, the result of running a FB is stored in its own output variable. FBs can thus have more than one output

IEC 61131-3 defines a set of FBs that, together with the standard's functions, cover the basic operations one associates with PLCs. These FBs are:

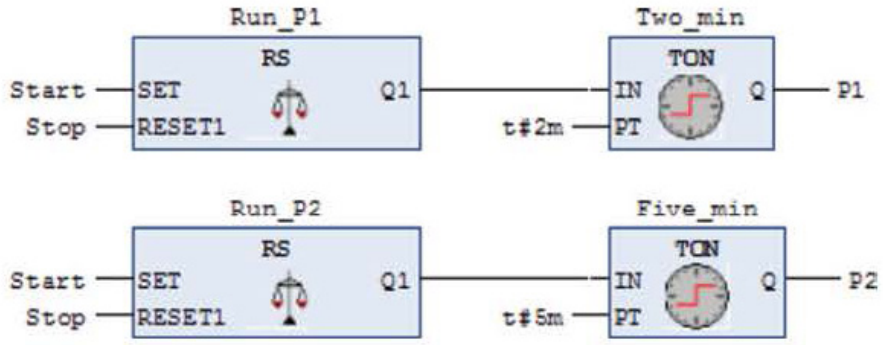
- Flank detection: R_TRIG and F_TRIG
- Bistable (flip-flop): SR and RS
- Timer (time delay): TON, TOF, TP
- Counter: CTU, CTD, CTUD

Here, we will study all of these FBs and explain how they work. You will find examples of the FBs in the section devoted to LD language.

In order to be able to use a FB in a POU, an instance of the FB must be declared in the POU. Instances are named multiple uses of a FB type. This sounds somewhat cryptic, but the point is that since a FB has memory and properties that are associated to it by use, one must declare a new instance of every FB that one uses even if their are of the same type. Suppose that in your program you have to use two RS flip-flops to start (Set) and to stop

(Reset) two pumps; P1 and P2. In addition, we will use two timers to provide a delay in starting the pumps of two and five minutes, respectively. In order to be able to start and stop each pump independently, you must declare an instance of each SR flip-flop and on instance of each timer. Declaration is made with the same syntax with which you declare variables, except that you indicate the type of FB instead of data type:

```
PROGRAM Call_of_FBs
VAR
      Start, Stop        : BOOL;
      Run_P1, Run_P2     : RS;// Declares two instances of a RS
      Two_min, Five_min  : TON;// and two instances of a Timer
      P1, P2             : BOOL;
END_VAR
(* Program code: *)
```




```
END_PROGRAM
```

When you declare such instances, you can name them anyway you want, so long as it is a permitted name. Personally, I like to name timers according to the length of time delay (such as five_min) and counters according to the value that they will count up to (e.g., Fifty). Regarding the other two categories of FBs, you will normally use many more of them in a program. For these, I therefore recommend simple, but at the same time fairly descriptive, names:

1. RS1, RS2, etc. for bistable of the type RS
2. SR1, SR2, etc. for bistable of the type SR
3. RE1, RE2, etc. for positive flank triggers (R_TRIG)
4. FE1, FE2, etc. for negative flank triggers (F_TRIG)

(RE and FE are acronyms for rising edge and falling edge, respectively.)

Once an instance has been declared, the FBs are made accessible for use in the code field in the POU. (It is also possible to declare an instance of a FB globally.) At any rate, the instance can now be used in a POU via a call.

The code above, programmed in CODESYS, shows use of our FB instances in FBD. Notice that the name of the instance will be at the upper edge of the symbol.

In Structured Text, we call instances of FBs by writing the name of the instance together with arguments. A call of Run_P1 can appear as follows:

```
Run_P1 (Set := Start, Reset1 := Stop, Q1 => P1);
```

Note that it is possible to use FBs without all inputs associated with variables. In the code on the previous page, the ET outputs are not associated to anything. Any inputs that are not associated will use their initial values in the execution of the block. In the following, we will study the way all of the FBs work in the standard. We will study the use of the FBs later when we review LD language.

FBs for Flank Detection

In many applications, we would like to perform an instruction only once, for example, precisely when a condition is met. For a PLC, this is not natural. It wants to check conditions repeatedly (every scan) and perform an instruction every time as soon as the condition is satisfied. The time between every time the condition is checked is what is called the scan time. This can be 20 ms, for example. Therefore, if a condition is met, the associated instructions will be performed at every scan, that is to say, every 20 ms.

Finding out exactly when a Boolean condition is satisfied is the same as detecting when the Boolean condition changes state from FALSE to TRUE (0 to 1) or vice versa, from TRUE to FALSE. This is called flank detection.

The standard defines to FBs for flank detection, R_TRIG and F_TRIG. These FBs operate in the same way as the flank triggers P and N in LD as we will see. Both blocks have one input and one output, both Boolean.

The graphic symbols for the blocks are shown in Figure 1.50. R_TRIG is used to detect a rising edge on the Boolean input variable or the Boolean expression associated with the input. When that happens, the output Q becomes TRUE and then remains TRUE for a time equal to the scan time, that is, until the next time the block is evaluated. F_TRIG

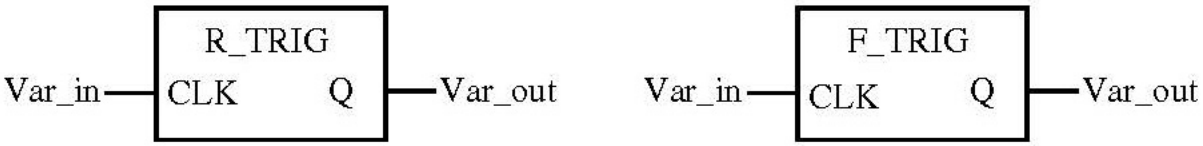


Figure 1.50: Function blocks for flank detection

is used to detect a falling edge on a Boolean variable or a Boolean expression. In the PLC scan where the input signal becomes FALSE, the output Q becomes TRUE. It then remains TRUE until the next scan (next execution of the block) (see Figure 1.51).

Bistable Elements

The Bistable elements SR and RS or memories, as they may also be called, are equivalent with the retention elements S and R in LD.

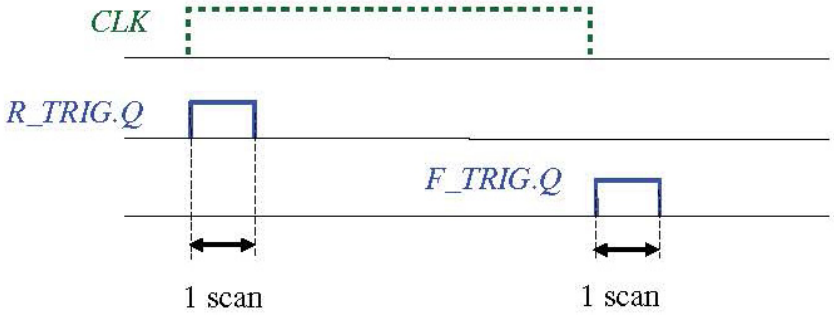


Figure 1.51: Diagram for the input and output signals of the block

Figure 1.52 shows a possible graphic representation of the blocks as they can appear in the graphical languages LD and FBD.

All variables must be of the type BOOL. The working principles are as follows: A logical high signal on the Set input (Var1 = TRUE) causes output Q1 to be set TRUE. It then remains TRUE even though the signal on the Set input becomes FALSE. The output can only be FALSE again when it receives a TRUE value on the Reset input.

The reason that these are also called memories is precisely that the output retains its value even though the state at the Set input becomes FALSE.

The difference between SR and RS first becomes noticeable when both Set and Reset are TRUE:

- SR is Set dominant. This means that Set has a higher priority than Reset and the output goes TRUE even if Reset is also TRUE.
- RS is Reset dominant, which means that the output goes FALSE when Reset goes TRUE, whether Set is also TRUE.

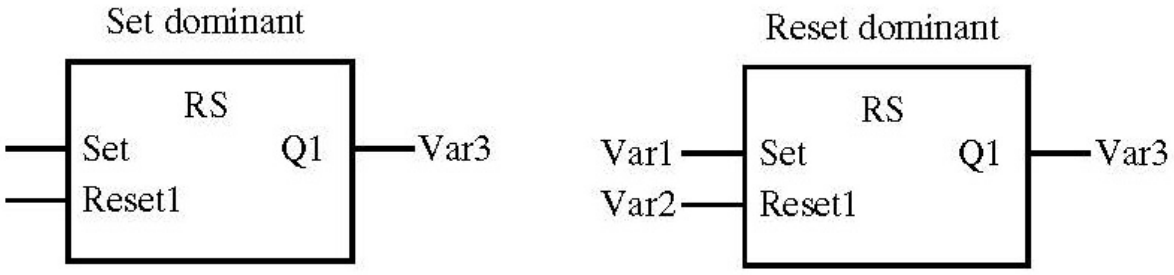


Figure 1.52: Bistable Elements

(Note that a 1 is attached to the dominant input variable as a reminder to the user for which of the two blocks is being used.) The Boolean quantities associated with the Set and Reset inputs do not need to be simple Boolean variables, but may also be Boolean expressions

Timers

A common element in traditional electrical installations is the time clock in its various variants. These are found in both mechanical and electronic implementations. There is also a time clock in the PLC, but there it is present as software. These are called a timer.

The purpose of using a timer, briefly stated, is to be able to change the state of the Boolean address at a desired time after some criterion has been met.

The symbol for a timer is shown in Figure 1.53

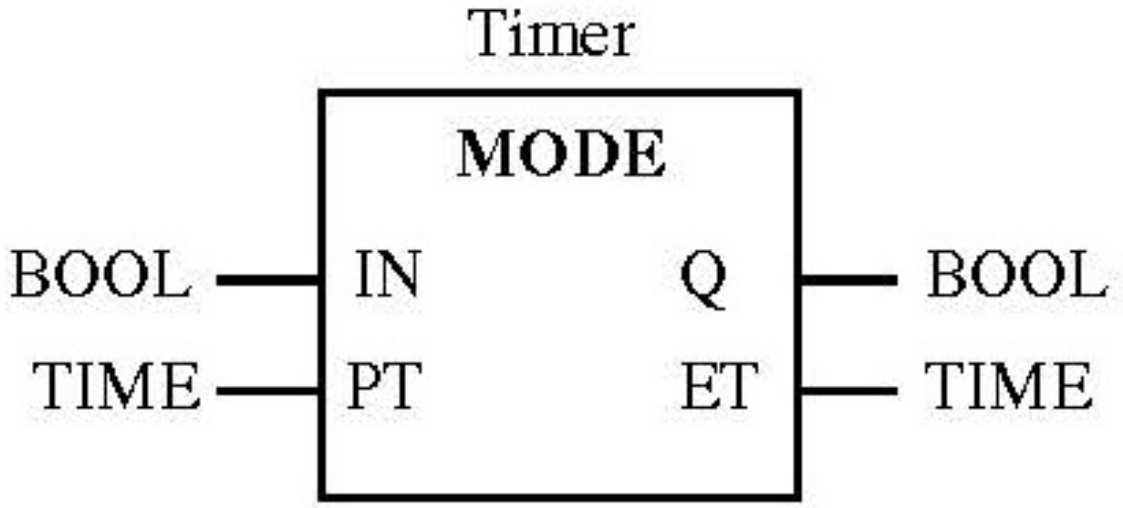


Figure 1.53: Timers Symbol

Inputs and Outputs

- Input variable IN: When the signal at IN changes state (rising edge for TON and TP, falling edge for TOF), the timer will start.
- Timer output Q: The state of output Q does not depend upon the input only but also upon the mode selected (TON, TOF, or TP) (see the following text).
- Input variable PT: Here, a variable of data type TIME is associated. This may be a time given directly in a standard time format (for instance, t#2m30s). PT is a predetermined time that indicates the desired delay.
- Output variable ET: A variable of type TIME that contains the current value in the timer, that is, the time that has elapsed since the timer was activated. When the content of ET equals the content of PT, the timer's output Q changes state.

The standard defines three different modes for control of the state of output Q:

- TON: The output Q gets TRUE a user-specified time (content of PT) after the condition at the input is satisfied (TRUE signal on IN). In order for the output to change state, the input must be TRUE for at least as long as the predefined time.

- TOF: The output gets TRUE immediately when the condition at the input is satisfied and thus when IN becomes TRUE. After the input becomes FALSE again, the output will becomes FALSE after a user-specified time, provided that the input has not gone TRUE again in the meantime (ET resets when IN gets TRUE).
- TP: The output Q goes logically high when input IN goes high and stays TRUE for the user-specified time PT, whether or not IN becomes FALSE in the meantime. (ET resets only when both the input and the output are logically low.)

Figure 1.54 shows a sequence diagram that illustrates the three modes. The solid lines

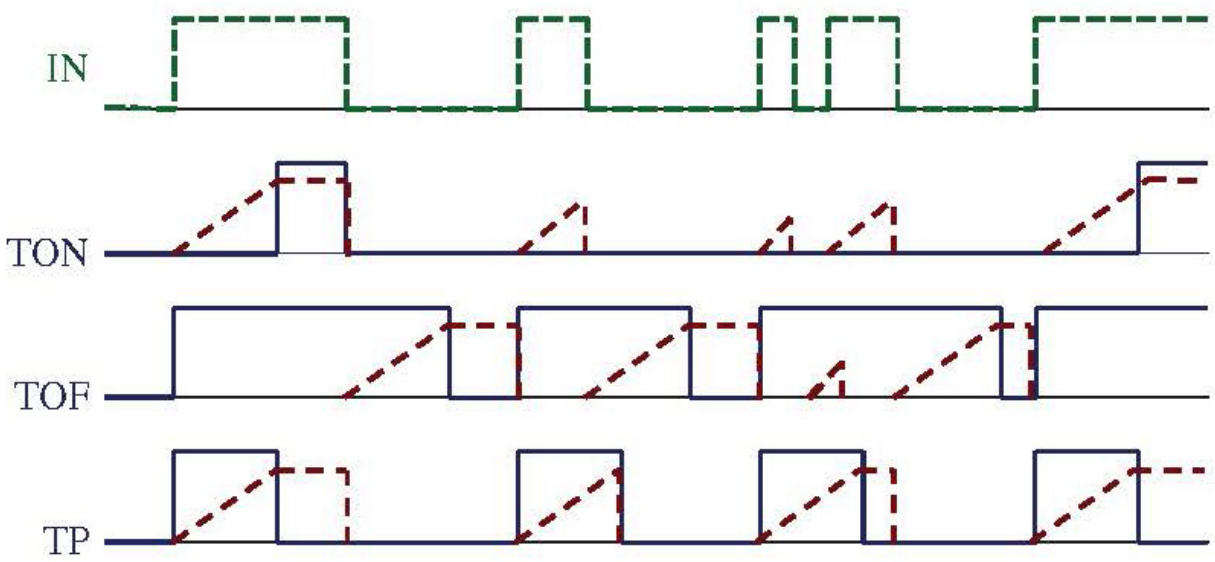


Figure 1.54: Illustration of the difference between the various modes of timer

show the state of output Q for a timer in each of the three modes. The dotted lines at the top of the figure show the state of the input IN. The figure also shows the values in ET (marked —-), which contains the current value in the timer. The output status changes when ET is equal to PT.

Counters

Counters are FBs that have some similarities in principle with timers. They have an input variable containing the desired number of pulses that are to be counted, an output variable that contains the counters' present value, and a Boolean output the changes state from FALSE to TRUE when the desired count is reached. The standard defines three types of counters for counting up, counting down, and up/down counting: CTU, CTD, and CTUD. It is unclear why three different FBs are defined for counting when the CTUD block covers all requirements for counting. It probably has some connection with tradition. At any rate, in the following, we will study the symbols and characteristic parameters for all three types.

Up-Counter: The up-counter's parameters and variables are shown in Table 1.9 (see Figure 1.55 for the graphic symbol) *Down-Counter*: The symbol and operation of a down-counter

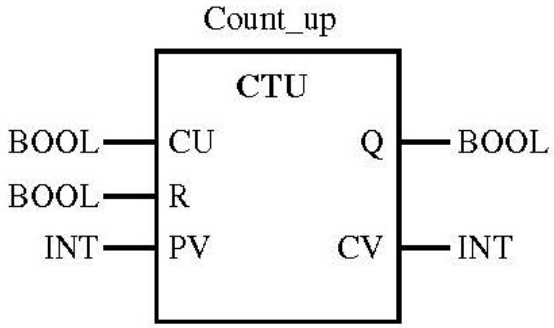


Figure 1.55: Up Counter

CU	Count Up	These are pulses (rising edges) of the input to be counted
PV	Preset Value	Integer that contains a desired preset value
CV	Current Value	Integer type of value whose content increases with one unit for each rising edge at CU
R	Reset (input)	Counter reset so that CV = 0 when R becomes TRUE
Q	Done (output)	Is set to TRUE when counting is done (CV = PV)

Table 1.9: Characteristic parameters for an up-counter

(Figure 1.56) is very much like that of an up-counter except for two things: on rising edges at the counter input, which is now designated CD, the counter counts downward from a preset value (PV). When CV equals 0, the counter is finished, and Q goes logically high. The reset input for CTU has been replaced here by a load input (LD). When LD is TRUE, CV is set to equal PV.

The down-counter's parameters and variables are shown in Table 1.10.

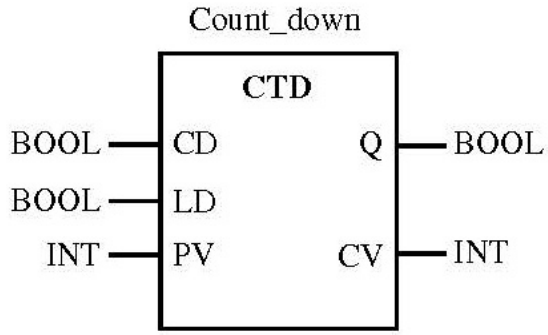


Figure 1.56: Down Counter

CD	Count Down	It is the number of pulses at this input that is to be counted
PV	Preset Value	Integer that contains a desired preset value
CV	Current Value	Integer type of value whose content decreasese by one unit for each rising edge at CD
LD	Load	Counter reset so that CV = PV when LD becomes TRUE
Q	Output	Is set to TRUE when counting is done (CV = 0)

Table 1.10: Characteristic parameters for a down-counter

Up/Down-Counter: The third and last FB for counting is called CTUD and implements a combination of CTU and CTD. The block's input and output variables are the same as those in CTU or CTD (Figure 1.57).
Operation

- Counting: The inputs CU and CD are scanned in turn. With a rising edge at CU, the current value (CV) increases by 1. With a rising edge at CD, the value CV decreases by 1.
- Reset: When input R is set to TRUE, the CV is set to 0.
- Load: When input LD is set to TRUE, CV is set to PV.
- QU returns TRUE when CV $\geq$ PV.
- QD returns TRUE when CV = 0.

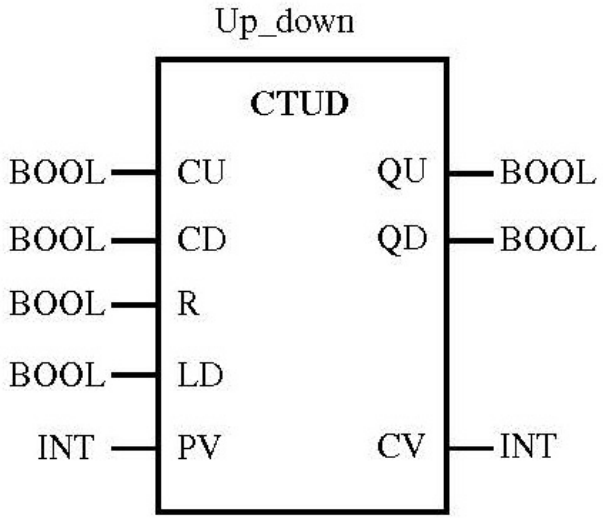
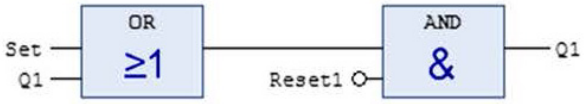


Figure 1.57: Up-Down Counter

5.11 Defining new FBs

The example shows a definition for the standard's FB RS. The declaration and code for a FB is enclosed by the keywords FUNCTION_BLOCK and END_FUNCTION_BLOCK. This particular block has two input variables and one output variable, all type BOOL. This FB does not have any local variables.

```
(* Example of a function block that implements a Reset dominant flip-flop.  *)
FUNCTION_BLOCK RS
   VAR_INPUT //Input variable
   Set :  BOOL;
   Reset1 :  BOOL;
   END_VAR
   VAR_OUTPUT
   Q1:  BOOL;
   END_VAR
(* Function block program code (in FBD): *)
```




```
END_FUNCTION_BLOCK
```

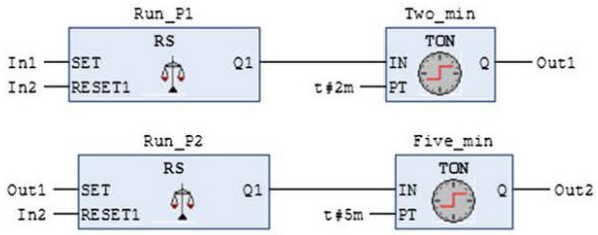
When a FB is used in the graphic languages, a graphic symbol is generated automatically, based upon the name of the block, the number of inputs (VAR_INPUT), and the number of outputs (VAR_OUTPUT).

An important point about FBs is that they are very useful for encapsulating code. If you write code for others, for instance, for a client, and have developed a special code that you perhaps would use again in several programs and which you do not want to distribute to others, you can store this code in the form of a FB.

You can then use the FB in a program, where it will do precisely the same job as though the code in the block had been part of the program code. The major difference is that no one can see the code that is stored in the FB, and you have thereby protected this section of the code.

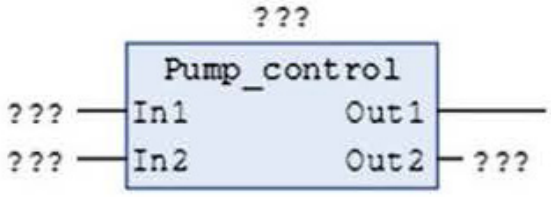
In the beginning of the FB section, we saw a little program where we controlled two pumps with the help of a pair of RS flip-flops and two timers. Let us now fix that code so that it is a FB rather than a program:

```
(* Example of a function block that implements a Reset dominant flip-flop.  *)
FUNCTION_BLOCK Pump_Control
   VAR_INPUT
   In1, In2 :  BOOL;
   END_VAR
   VAR_OUTPUT
   Out1, Out2:  BOOL;
   END_VAR
   VAR
   Run_P1, Run_P1:  RS;
   Two_min, Five_min:  TON;
   END_VAR
```




```
END_FUNCTION_BLOCK
```

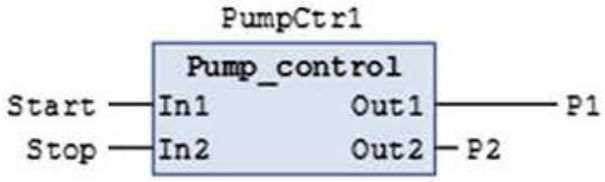
If you compare this code with the code a the beginning of the FB sectiom, you will see that they are identical. The difference is in the declaration field, the fact that this is a POU of the FB type instead of a POU of the program type. In the program, we can use as many FBs as we care to, of any type, and when we use them, we will not see the code that lies within the FB; we only have to know what it does and how we put it to work in the program. Since you can use a FB several times, and perhaps for various purposes, you should use more general variable names in the declaration. Here, I have used In1 and In2 on the input variables and Out1 and Out2 on the output variables. Now, if we make a program, we can declare an instance of our new FB (Pump_control) and use it in a code. As mentioned, it will automatically generate a symbol in graphic language, like this:



Now we see the use of our new FB in a program. Like all other FB, we must declare an instance of our FB in the declaration field where we give the instance a name. The instance can then be used in the code field, where we can associate other variables (or expressions) to

the inputs and outputs of the instance. The goal of concealing the part of the code that lies in the FB has been achieved, in addition to the advantage that it is simple to reuse the code in several programs or several times in the same program.

```
PROGRAM Calling_Pump_control
   VAR
   Start AT %IX2.3 :  BOOL;
   Stop AT %IX2.4 :  BOOL;
   P1 AT %QX5.0 :  BOOL;
   P2 AT %QX5.1 :  BOOL;
   PumpCtr1 :  Pump_control;
   END_VAR
```

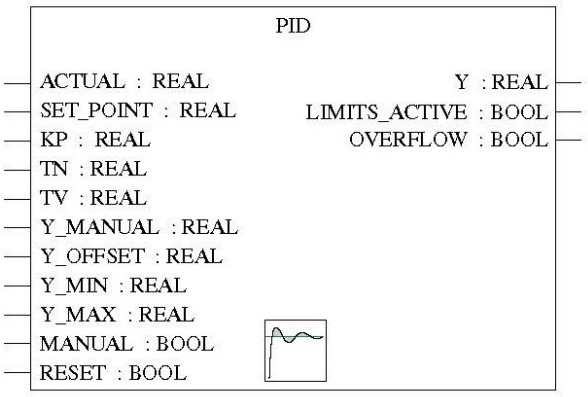



```
END_PROGRAM
```

Non-standardized Function Block Section

As mentioned earlier, the development tool that you use will offer many more functions and FBs then are defined in the standard. In addition, users and third-party suppliers develop new FBs that they often make available on the Internet. Before you get started programming your own FB for a particular purpose, it can therefore be a good idea to look around a little in the programming tool library or search on the Net. The following example describes a PID function block.

```
FUNCTION_BLOCK PID
   VAR_INPUT
    ACTUAL :REAL;          (* Actual value (PV - process variable) *)
    SET_POINT :REAL;      (* Desired value, set point *)
    KP :REAL;             (* Proportionality const.  (P) *)
    TN :REAL;             (* Integral time (I) in sec *)
    TV :REAL;             (* Derivative time (D) in sec*)
    Y_MANUAL :REAL;       (* Y is set to this value as long as MANUAL=TRUE *)
    Y_OFFSET :REAL;       (* Offset for manipulated variable *)
    Y_MIN :REAL;          (* Minimum value for manipulated variable *)
    Y_MAX :REAL;          (* Maximum value for manipulated variable *)
    MANUAL :BOOL;         (* TRUE: manual:  Y is not influenced by controller, FALSE: co
    RESET :BOOL;          (*Set Y output to Y_OFFSET and reset integral part *)
   END_VAR
   VAR_OUTPUT
    Y :REAL;              (* Manipulating variable *)
    LIMITS_ACTIVE :BOOL;  (* TRUE if value exceed Y_MIN, Y_MAX *)
    OVERFLOW :BOOL;       (* Overflow in integral part *)
   END_VAR
   VAR
    CLOCK :TON;
    I :INTEGRAL;          (* Integral and Derivative are FBs*)
    D :DERIVATIVE;
    TMDIFF :DWORD;
    ERROR :REAL;
    INIT :BOOL:=TRUE;
    Y_ADDOFFSET :REAL;
   END_VAR
```



Programs

The third type of POU in the standard—a program—is defined as:

...a logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a PLC system.

Said a little differently (but perhaps not better): A program consists of addresses, variables, constants, functions, FBs, and control structures combined in a logical way so that it constitutes a runnable code that solves a control problem.

As we see, it is difficult, and perhaps meaningless, to define what a program is. The following chapters on programming languages will, we hope, provide a good understanding of what a program is and how program code is constructed with various programming languages.

To begin with, we will be satisfied with the guidelines that the standard specifies. Declaration and use of the program is identical with that specified for FBs, except for the following:

- A program is bounded by the keywords PROGRAM and END_PROGRAM.
- , A program is at a higher structural level than FBs. This means that a program can contain instances of FBs, but an FB cannot contain programs.
- A program can, in addition to containing functions and instances of FBs, also call other programs.

A program can call other programs. You can structure your application better by splitting up the code into several POUs of the program and FB types, where each POU handles its share of the control. This is important in larger applications and makes it simpler to maintain and structure the code. Another advantage is that you can reuse snippets of code by importing programs and FBs that you have previously developed for other projects.

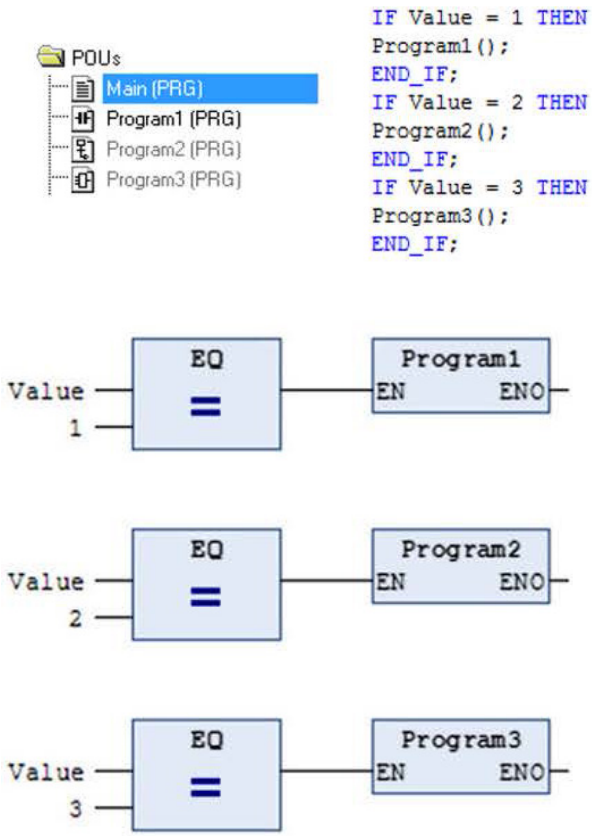
When a program is called, any changes in values and variables will be retained for the next time the program is called. This is different from calling up a FB, where only the variables in the current instance of the FB are changed. As we have seen, you can declare several instances of the same FB in a program. The values that are changed affect only the current instance and are therefore only significant in the next call of the same instance.

In the example we show how the code in a program can call other programs in the application. What conditions are used in calling the individual programs will naturally depend upon the project in question. If it is a sequential structure, for example, the next program can be called when the previous program has carried out its part. In this example, it has been done simply by having the programs called as a result of the value in the integer variable value. Here the code in the application is divided into four different programs. A main program, called main, calls the other three programs. We see that the individual programs can be written in different languages (Program1, LD; Program2, SFC; and Program3, FBD).

There is often a requirement to use the result from an instruction, a network in LD, or a POU to control other POUs. This can easily be solved by sending the result from the instruction or network to a global variable that is then used to control the execution of other POUs.

It is also possible to do this directly by the use of Enable (EN) and Enable Out (ENO) as we previously saw.

If these variables are used in a function, FB, or program, the execution of the POU takes place in accordance with the following rules:



1. If the value to input EN is FALSE (0), none of the instructions that are defined in the POU are executed, and the output ENO is set to 0.
2. If the value to EN is TRUE (1), the defined instructions are executed, and the output ENO is set to 1 as soon as the execution of the instructions is completed successfully (with no errors).
3. For FBs, all outputs (VAR_OUTPUT) will retain their values from the previous call if EN is set to 0.

6 Ladder Diagram (LD) Language

Programming with LD has traditionally been the most widely used programming language for PLCs. Another concept that is used for code written in LD is relay diagram. Even though other, and in many instances more efficient, programming languages have gradually appeared, LD continues to be widely used. There are many reasons for this:

- It has been around for a long time.
- The language is graphic.
- It is relatively easy to grasp.

Previous editions of LD had a basic set of instructions sufficient to be able to perform most of the fundamental types of control functions such as logic, time control, and counting, along with simple mathematical operations. Most of the PLC manufacturers currently provide PLCs and programming tools that make it possible to perform advanced additional functions in LD, often integrated with other languages such as FBD and ST.

The basic functions that are needed in order to implement smaller control systems can be learned relatively quickly, and the graphic presentation can be understood intuitively. For beginners in programming, and when smaller applications are to be developed, LD is therefore a fine choice of programming language.

6.1 Program Structure

Figure 1.58 shows a sketch in principle of how a LD code is structured. Both sides of the code are bounded by vertical lines that we can call power rails. The rail on the left always has the state TRUE. You can consider that to be a voltage (+24 V as an example) connected to the rail.

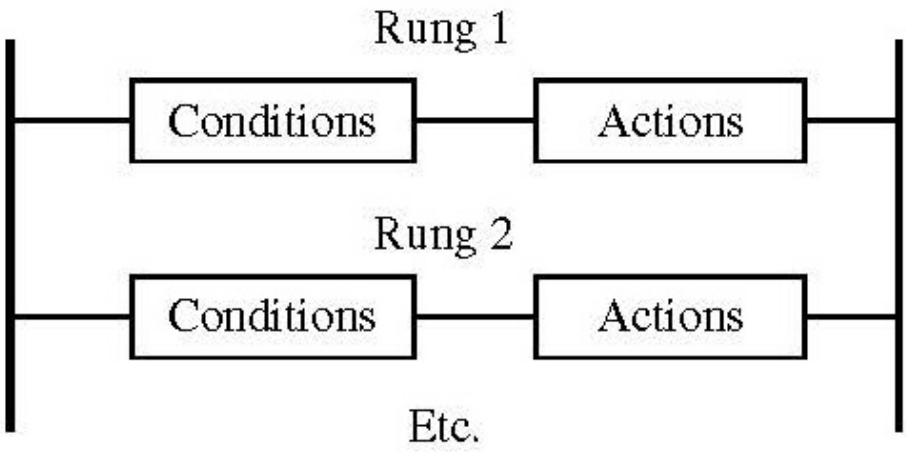


Figure 1.58: LD code structure

The rail at the right has no defined logical state under the standard. It can therefore be considered as implicit, and not all producers implement a vertical right line in the development tool.

Generally, a code in an LD is based upon the following principles:

If a condition or a combination of conditions is satisfied, then one or more actions (events, instructions) will be performed.

We can call a set of conditions with associated actions a rung[1]. A program will thus consist of one or more such concatenated rungs that are being executed sequentially by the PLC. Here is an example of an LD code:

[1]Some versions of the LD documents are referring to the term *networks* instead of rungs

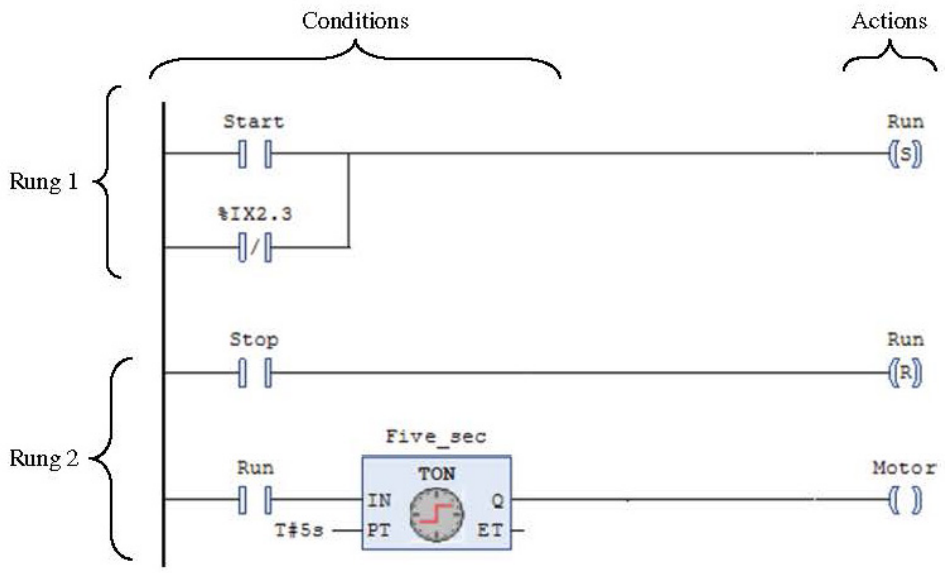


Figure 1.59: LD code example

Contacts and Conditions

The conditions in Figure 1.59 will most often be a contact or a combination of contacts. A contact is a graphic element (-||-) that is associated with a Boolean variable or a Boolean address.

Even though there is naturally no current going through these logical rungs, we can understand them better if we consider a contact to be like an ordinary light switch: If it is closed, current goes through the contact. If it is open, the flow is interrupted.

There are several types of contact symbols. The two basic variants are normally open (NO) and normally closed (NC). The symbols for the two types associated with the variable Var_A are shown in Figure 1.60. In other words: A sensor that is connected to the digital

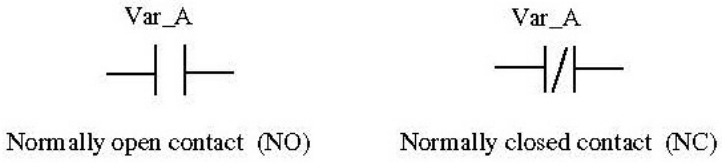


Figure 1.60: Contact types

input that has this address will "go hot" through the contact when the sensor gives a logically high signal.

A normally closed contact functions just opposite of this; the contact is closed (conducts current) when the state of the sensor is FALSE, and the contact is open when sensor state is logically TRUE.

6.2 Coils and Actions

An action or *instruction* in LD can be various things. For example, they can perform an arithmetic calculation, jump to another place in the program code, or change the state of Boolean addresses.

This last event comes in under the group of Boolean instructions, and the graphic symbol that is used for this is called a coil (see Figure 1.61).

Figure 1.61: Coil types

Coils are placed at the right of conditions, which are placed at the left in the code. As mentioned earlier, we can analyze the rungs by thinking how they conduct current and which contacts are on or off. The job of a coil is to transfer the result (the Boolean state) from a condition on the left side of the coil to the Boolean variable or address that is associated with the coil.

In the same way that contacts can be associated with Boolean input addresses, a coil can be associated with an address to a physical output where it is connected to actuators such as magnetic valves, lights, alarms, relays, and so on.

Example 14. Assume the following: A sensor gives a high signal when the fluid level in the tank reaches a certain level. The sensor is connected to an input with address %IX1.8. When the fluid level reaches the sensor, a pump should start so that the fluid is emptied out of the tank. The pump is connected to a relay output with the address %QX2.3. The program for this in LD can look like the one shown in Figure 1.62.

The open contact here constitutes the condition and the coil constitutes the action or instruction that is to be carried out when the condition is satisfied. We can read the program in this way:

- When the state of address %IX1.8 is TRUE, the state of %QX2.3 should also be TRUE. Therefore, when the sensor gives a logically high signal, the pump should operate (assuming that the pump in the electrical layout is connected so that it runs when the relay is closed).

- When the state of address %IX1.8 is FALSE, the state of %Q2.3 should also be FALSE. Therefore, when the sensor gives a logically low signal, the pump should not operate.

△

Figure 1.62: A really simple LD program

Graphic Elements, an overview

Since the PLC program obvious is not a physical hookup of switches, but rather logical functions, there is no physical situation that limits the functions that are possible to be programmed. This means that different and more complex functions than those used in relay controls can be programmed in a PLC.

In the previous sections, a series of standard functions and function blocks have been presented. All of the function blocks can be implemented in LD as well as several of the standard functions.

Some of the function blocks are so central in programming that the standard defines specific graphic symbols in LD for them. This applies to the bistable function blocks SR and RS and the edge-detection function blocks R_TRIG and F_TRIG:

- SR and RS are implemented in LD as a Set coil -(S)- and a Reset coil -(R)-.
- R_TRIG and F_TRIG are implemented in LD as -|P|- and -|N|- contacts that detect rising and falling edges, respectively.

All of the graphic LD elements defined in the standard are collected in Figure 1.63.

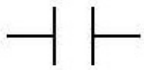
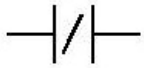
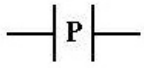
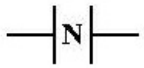
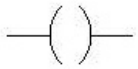
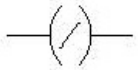
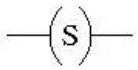
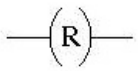
Designation		Symbol	Function
Test element	Normally open contact		The contact closes when the associated Boolean object becomes TRUE
	Normally closed contact		The contact closes when the associated Boolean object becomes FALSE
	Flank-detecting contacts		Rising edge: The contact is closed only in the scan (see Section 1.3.3) during which the associated Boolean object changes state from 0 to 1
			Falling edge: The contact is closed in the scan where the associated Boolean object changes state from 1 to 0
Connections	Horizontal		To connect elements in series
	Vertical		To connect elements in parallel
Action element	Direct coil		The associated Boolean object is set to the same state as the state of the left side of the coil
	Inverse coil		The associated Boolean object gets the inverse of the state of the left side of the coil
	On-coil		The associated Boolean object is set to TRUE when the state of the left side of the coil is TRUE
	Off-coil		The associated Boolean object is set to FALSE when the state of the left side of the coil is TRUE
	Conditional jump to another rung	→ Label	Enables jump to another named rung in the program (the POU) When a jump is activated: 1. The active rung is interrupted 2. The named rung is activated
	Return to call	**< RETURN >**	If a function or function block is programmed in LD, this is used to return to the POU that called up the function or block[2]

Figure 1.63: LD symbols collection

6.3 Boolean Operations

We have previously presented a brief overview of all the Boolean functions in the IEC standard. In structured text, these are represented with their individual operators and special graphic symbols are used in FBD.

One group of standard functions are the Boolean operations AND, OR, XOR, and NOT. In LD, such operations can be implemented by combining the basic graphical elements (contacts).

AND/OR Conditions

In many situations, there are more complex conditions that must be satisfied before an instruction is performed. AND-conditions and OR-conditions and combinations of these are very common. If one is clear about what logical conditions must be met for these events to take place, it is easy to implement them in a PLC program.

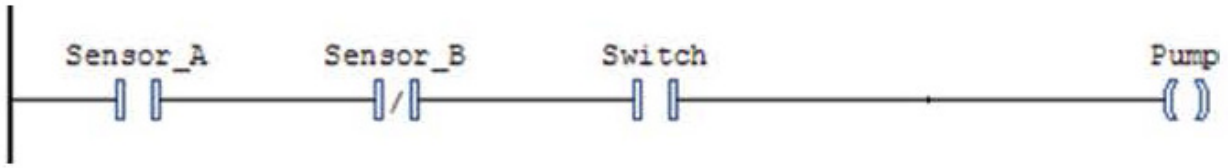


Figure 1.64: AND LD example

Example 15. [AND Condition]:

We can read the program code in Figure 1.64 thus:

IF Sensor_A = TRUE AND Sensor_B = FALSE AND Switch is switched ON THEN The pump will run.

We can express it thus in logical form:

$$\texttt{Pump} := (\texttt{Sensor\_A}) \cdot \left(\overline{\texttt{Sensor\_B}}\right) \cdot (\texttt{Switch})$$

△

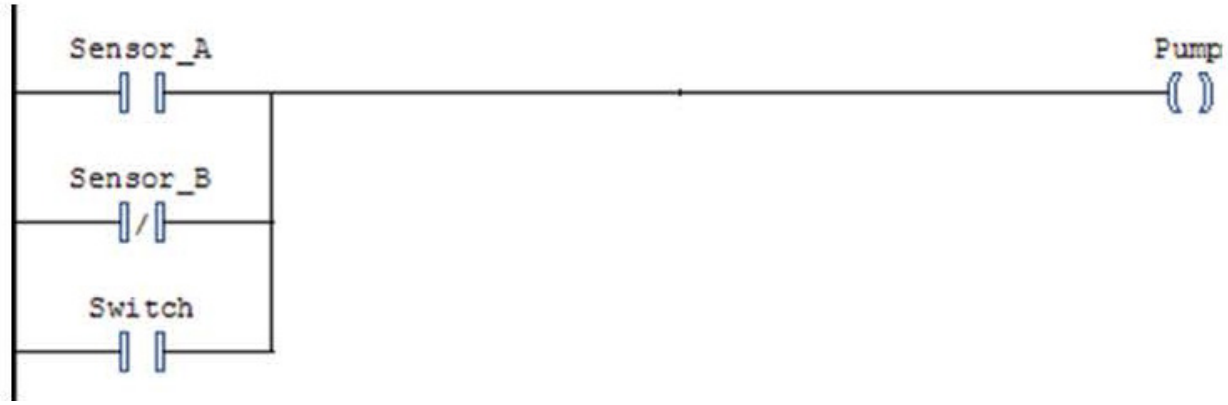


Figure 1.65: OR LD example

Example 16. [OR Condition]:

We can read the program code in Figure 1.65 thus:

IF Sensor_A = TRUE OR Sensor_B = FALSE OR Switch is switched ON THEN The pump will run.

We can express it thus in logical form:

$$\texttt{Pump} := (\texttt{Sensor\_A}) + \left(\overline{\texttt{Sensor\_B}}\right) + (\texttt{Switch})$$

△

Example 17. [XOR Condition]:
Now, it is obvious that any logical function can be implemented in a PLC where all possible combinations of AND-conditions and OR-conditions can be exploited. For example, here, Figure 1.66 shows how a 2-input Exclusive OR can be implemented.

A	B	G	$G = A \oplus B = A \cdot \bar{B} + \bar{A} \cdot B$
0	0	0	
1	0	1	
0	1	1	
1	1	0	

Table 1.11: XOR Table

Of course, the logical expressions can become complicated. In general, one must be sure to include enough of the conditions that are necessary in order for the process to be controlled to behave as one wish. Many inexperienced programmers, however, have a tendency to use unnecessarily complex conditions. It can therefore be sensible to begin by using the techniques and methods for reducing the Boolean expressions, such as those previously introduced. △

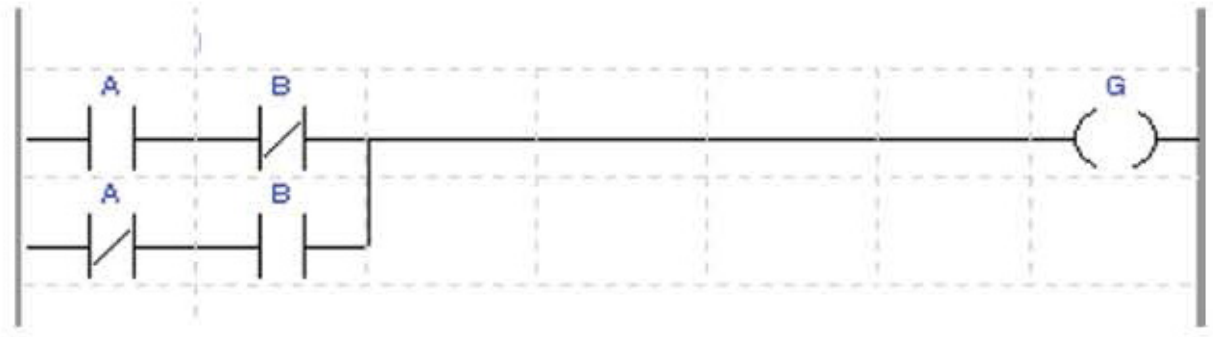


Figure 1.66: XOR LD example

Set/Reset Coils

A function that frequently comes up in an electrical facility is to start an electric motor with the help of a pulse switch (push button). Such a switch completes a control circuit, but only while the button is being held in, and then the current is interrupted when the button is released. This means that the signal from the pulse switch that is used as a condition for starting the motor is present only while the button is being held down. However, the motor should continue to run even though the signal from the push button disappears. How can we do that?

Electrically, this can be designed with the help of a contactor that will stay connected, after the control current turns it on, for as long as the operating current does not disappear. As soon as the operating current cuts off, the contactor will disconnect and can only be turned on again when the pulse switch is pressed in again. Here, we will look at ways to solve this problem in a PLC.

The value of the variables and addresses associated with the coil objects that we have used up till now will always be the direct result of the conditions that set them high or low. This means that if the result of the condition changes state, the variable associated with the coil object also changes state. Such programs are said to be purely combinatorial since they do not take into account any aspect of time such as when an event occurs, for instance.

Example 18. Let us assume that we have a pulse switch that is used to start a motor and another pulse switch that is used to stop the same motor. We want to make a kind of retention function that keeps the motor from stopping when we release the start button. Assume that both the start button and the stop button are both physically of the NO type. One possible solution is shown in Figure 1.67

Figure 1.67: Possible Implementation of a Retention Function

We have thus connected the start switch to address %I1.14 and associated this with a normally open contact. The stop switch is connected to the digital input %I1.15. Here, we use the symbol for a normally closed contact so that %I1.15 will be TRUE when the stop switch is not pressed. The condition for the motor to start is therefore that the start button is activated but the stop button is deactivated.

When this condition is satisfied, this state of address %Q2.1 is set TRUE, which is something that again means that a relay in the PLC's output block is closed so that the motor receives operating voltage.

When the start button is released, the motor will continue to run because the output address %Q2.1 is associated to a contact in parallel with the start button. Thus, the condition continue to be satisfied and will be until the stop button is operated. △

This is only one example where retention functions are needed. We often have the requirement that the program stores an action when a certain combination of values or states is present. The standard therefore defines two special coil symbols that can be used for this purpose (i.e., to implement memory[2]). With these coils, a Boolean value can be set high (S) and held high until it is reset (R).

- -(S)- : Set coil—Sets the associated Boolean address (high state).
- -(R)- : Reset coil—Resets the associated Boolean address (low state)

These coils are otherwise used in the same way as other coils. A contact, or several contacts in combination, are used to structure conditions that must be satisfied for a Boolean quantity (a digital output, for instance) to be set high. After that, the output in question will remain high even though the condition that set it high is no longer present. Now, we can control the motor in previous example with the following short code (Figure 1.68). Note that the

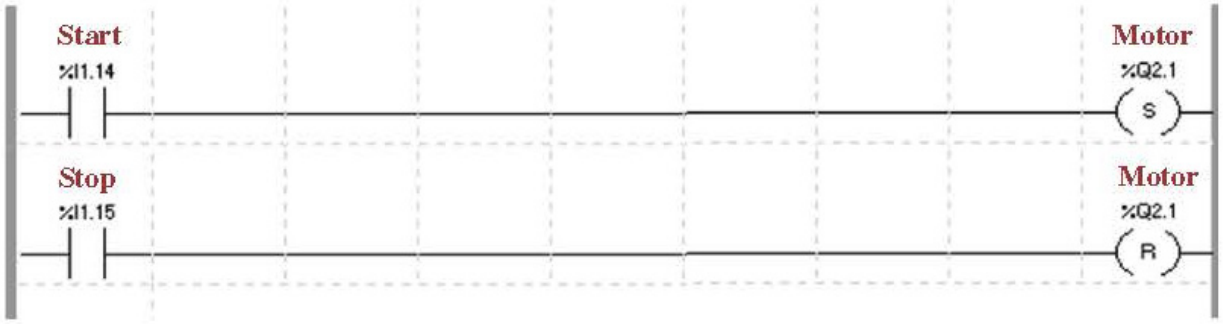


Figure 1.68: Latched Coils Implementation of the Retention Function

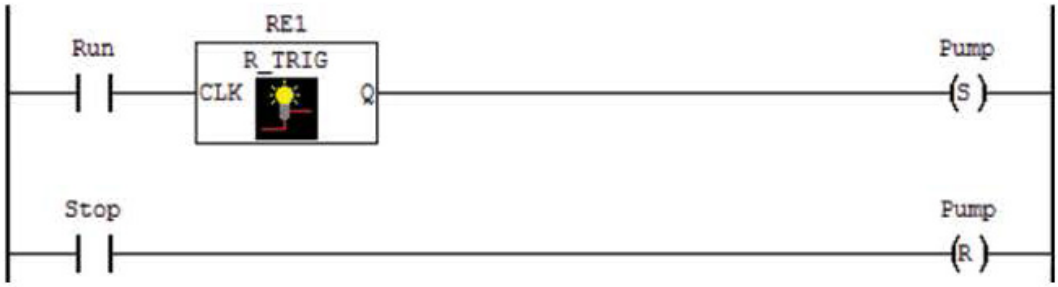
sequence of the LD rungs can be significant for how the code functions. We will study this in a devoted section.

Example 19. A pump is to start when a switch is turned from the Off-position to the Run-position. The pump is stopped by pressing a special stop button. In order for the pump to be able to start again after the stop button is released, the start switch must first be turned to the Off-position before the pump can be started again by twisting the switch to Run.

[2]Such coils are often called *Latched Coils.*

The following program implements this: Note the declaration: Use of function blocks, no

```
PROGRAM Rising
VAR
    Pump    AT %QX2.5     :BOOL;
    Run     AT %IX1.7     :BOOL;
    Stop    AT %IX1.8     :BOOL;
    RE1                   :R_TRIG;
    (*Here we declare an instance of the FB R_TRIG *)
END_VAR
```




```
END PROGRAM
```

Figure 1.69: Use of the function block R_TRIG and the Set/Reset coils

matter what type, requires that instances of the block be declared together with the POU's variable. In this example, an instance of the standard function block R_TRIG is declared.

The instance is here given the arbitrary name RE1. Legal names follow the general rules for identifiers.

Use of function blocks require the declaration of an instance for each and every one that is used. This applies even if the blocks are of the same type. A new instance must be declared for each use. △

Suppose that we want to manage the Stop button in the example earlier in the same way as the Start button so that it is still possible to start the motor again by turning the Start switch, even if someone continues to hold down the Stop button. Then, we must still declare another instance of the R_TRIG block: Even though the standard defines S- and R-coils in LD, you can use the function blocks SR and RS instead of these coils or in addition to them. As we will see later in the chapter, there can be an advantage in using these function blocks since the LD code becomes easier to read.

For example, the code in Figure 1.68 can also be implemented as shown in 1.71. Note that the block RS is Reset dominant, something that implies that it if both Start and Stop buttons are held in, the motor will stop.

```
PROGRAM Rising2
VAR
    Pump    AT %QX2.5    :BOOL;
    Run     AT %IX1.7    :BOOL;
    Stop    AT %IX1.8    :BOOL;
    RE1                  :R_TRIG;
    RE2                  :R_TRIG;
END_VAR
```

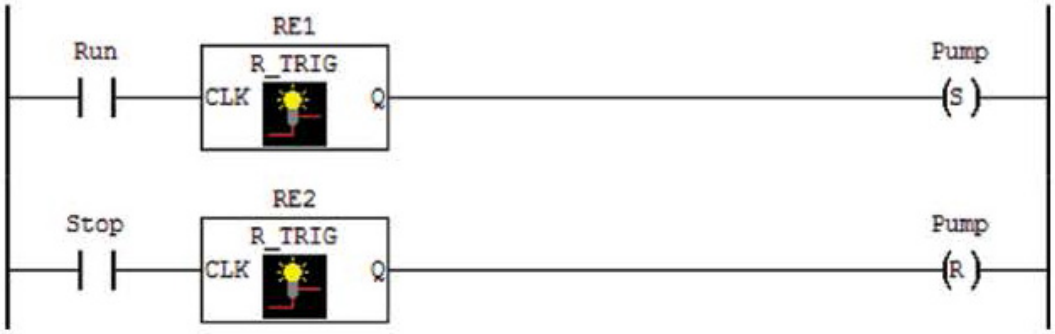


Figure 1.70: Use of a type SR/RS function block in LD

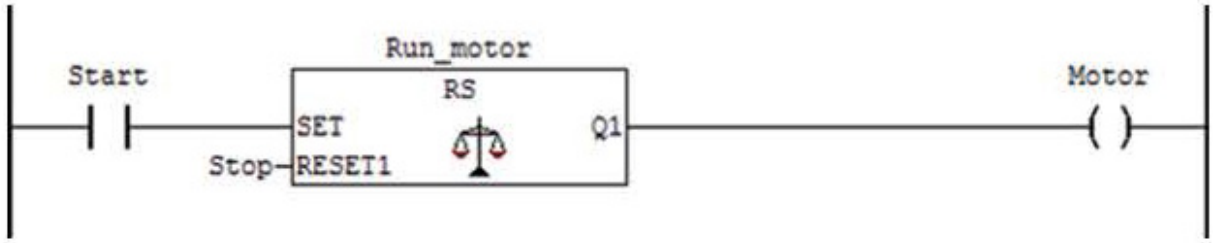


Figure 1.71: Use of a type SR/RS function block in LD

Edge Detecting Contacts

We have seen that the special coils Set and Reset in LD are used as an alternative to the function blocks SR and RS to implement memory. Similarly, there is also defined an alternative to the function blocks R_TRIG and F_TRIG for edge detection. These are implemented like special contacts -|P|-, -|N|-.

The contacts "close" and are held closed only during the program scan when the associated Boolean variable changes state. The type P contact detects a change in state from 0 to 1 (rising edge), while the N contact detects a transition from 1 to 0 (falling edge). In other words, the symbols can be used to set Boolean values TRUE/FALSE precisely when the condition is satisfied. An illustration of the operation of the flank contacts compared to an ordinary contact of the type NO is shown in Figure 1.72

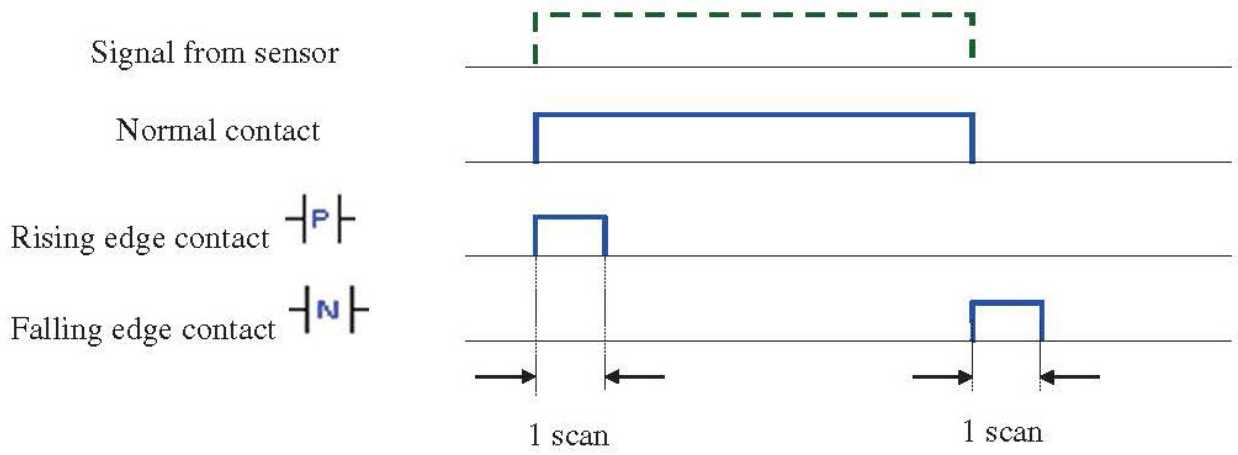


Figure 1.72: Operation of the flank-detecting contacts

Control of a Mixing Process

Figure 1.73 shows a process that is part of an industry known and loved by many. In this process, water is mixed with sugar and fermented barley in a predetermined ratio. The mixing process should proceed as follows: Assume that the tank is empty at the start. When the start button (`Start`) is pressed, magnetic valve `MV1` opens so that water runs into the tank.

- When the level reaches sensor `LT2`, the water supply is closed and the motor for the conveyor belt starts at the same time that the agitator (Stir) starts.
- When the level reaches transmitter `LT3`, the motor stops and the magnetic valve `MV2` at the outlet opens (the agitator will continue to run).
- When the level falls below `LT2`, the agitator is also stopped.
- When the level then falls below `LT1`, valve `MV2` is closed.

When the level in the tank reaches level transmitter `LT2`, the water supply is shut off and the motor for the conveyor belt starts at the same time that the agitator (stir) starts. When the level comes up to transmitter `LT3`, the motor will stop and the magnetic valve `MV2` on the outlet is opened (the agitator will continue to run). When the level comes below `LT2`, the agitator is also stopped.

The sequence can now be started anew by activating the start button. In Figure 1.74, we are drawing up a sequence diagram for the mixing process. With such a diagram in hand, it is simple to set up function expressions for each of the outputs and write the program code in LD language. The trick to getting a program code that is unambiguous, that is, where the function expressions do not conflict with one another, is to use memory (Set/Reset) and flank contacts. For example, we see that output `MV1` will go logically high when the start button is pressed (Start becomes logical 1). In other words, we want to set output `MV1` logically high when the Start signal changes state from 0 to 1. In LD code, this becomes: The function

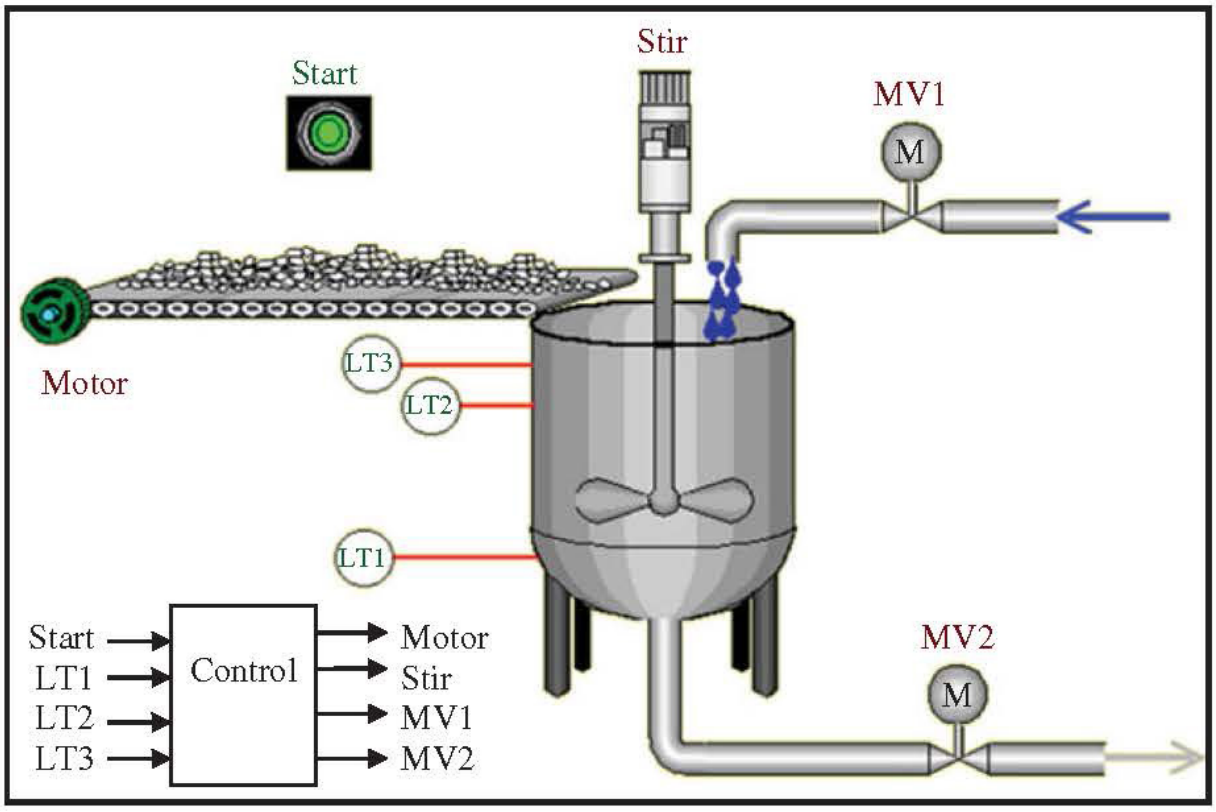


Figure 1.73: Mixing Process

expression for this code can be described thus:

$$\texttt{Set MV1} =\uparrow \texttt{Start}$$

The up arrow there symbolizes a positive flank. With this code, valve `MV1` will stay open until we reset the signal. From the diagram, we see that this will happen when `LT2` goes logically high:

$$\texttt{Reset MV1} =\uparrow \texttt{LT2}$$

The function expressions for setting and resetting the other outputs similarly become:

$$\texttt{Set motor} =\uparrow \texttt{LT2}, \quad \texttt{Reset motor} =\uparrow \texttt{LT3}$$

$$\texttt{Set stir} =\uparrow \texttt{LT2}, \quad \texttt{Reset stir} =\downarrow \texttt{LT2}$$

$$\texttt{Set MV2} =\uparrow \texttt{LT3}, \quad \texttt{Reset MV2} =\downarrow \texttt{LT1}$$

A complete code in LD for the process, including declaration of variables is shown below

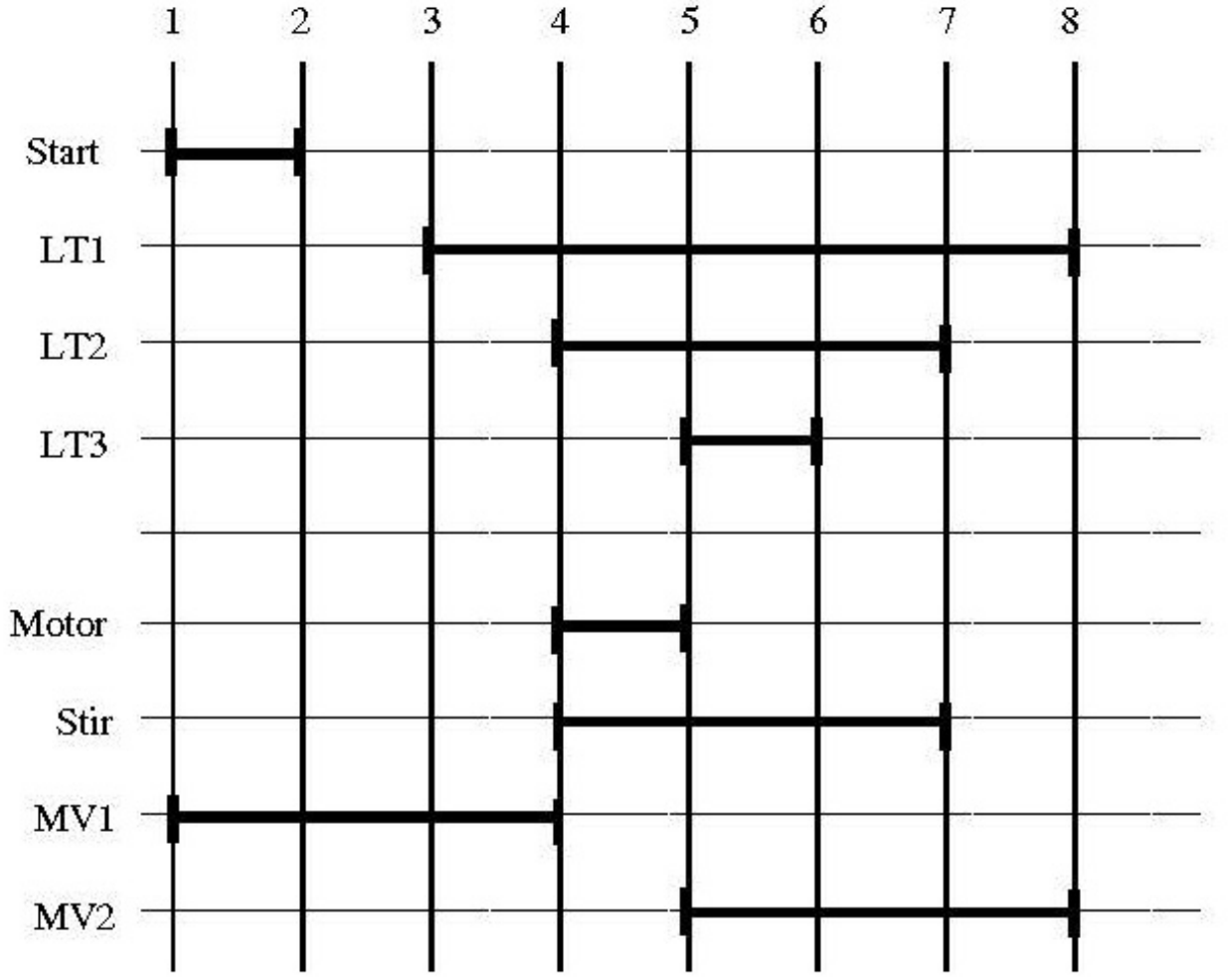
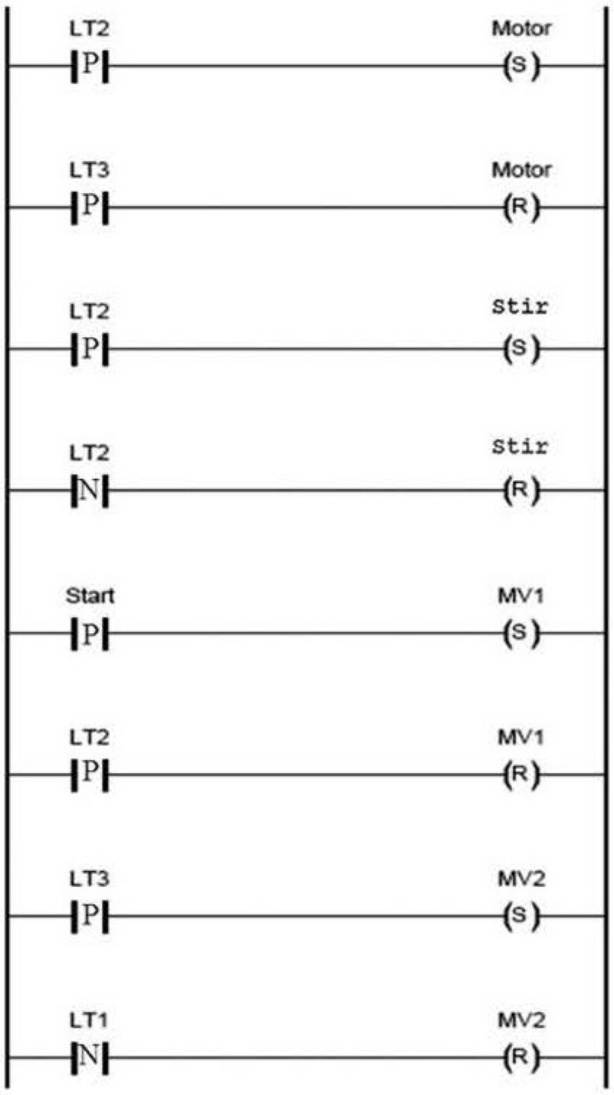


Figure 1.74: Sequence diagram for the Mixing process

```
PROGRAM Mixing process
VAR
    Start   AT %IX1.0:
      BOOL;
      LT1   AT %IX1.1:
        BOOL;
      LT2   AT %IX1.2:
        BOOL;
      LT3   AT %IX1.3:
        BOOL;
      Motor AT %QX2.0:
        BOOL;
    Stir    AT %QX2.1:
      BOOL;
      MV1   AT %QX2.2:
        BOOL;
      MV2   AT %QX2.3:
        BOOL;
END_VAR
```


LT2
P
Motor
S
LT3
P
Motor
R
LT2
P
Stir
S
LT2
N
Stir
R
Start
P
MV1
S
LT2
P
MV1
R
LT3
P
MV2
S
LT1
N
MV2
R

Use of edge detecting contacts in the program code is also significant for the duration of the state of the signal. Since the sequence in this example will start when the start button is activated, it is of no significance how long the operator holds the start button in. In order for the sequence for this process to start anew, the start button must be activated again so that the PLC registers a new positive flank.

6.4 Rules of execution and Ambiguous Situations

A program written in LD will be executed rung by rung from top to bottom. Each individual rung in the code is also evaluated from top to bottom, row by row, and in each row from left to right (see Figure 1.75). In accordance with these rules, the PLC will:

- Evaluate the logical state of each contact in accordance with the instantaneous values of variables or states to the inputs for the I/O modules.
- Update the Boolean objects associated with the coils.
- Go to another rung in the same program (jump or return).

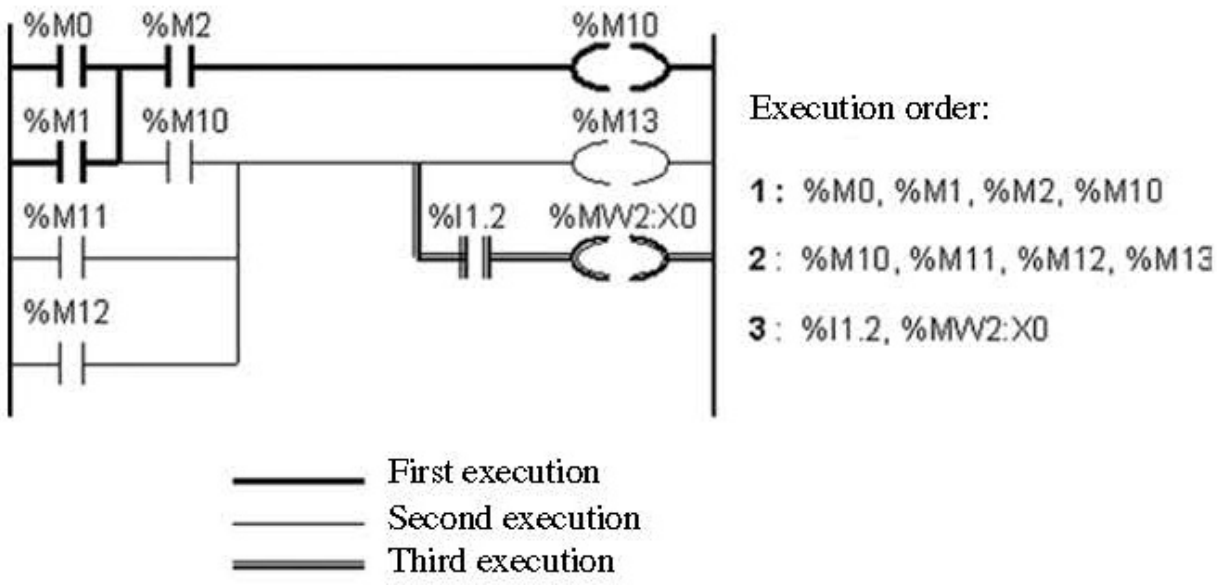


Figure 1.75: LD Execution Order Sequence

One Output: Several Conditions

A typical mistake that many beginners in LD programming make, especially when they are trying to make a program without planning it through first, is to insert code rungs that are in conflict with each other. The problem often occurs as a result of coils associated with the same address or variables used at several places in the code. Here, we will study a simple example that illustrates such a conflict.

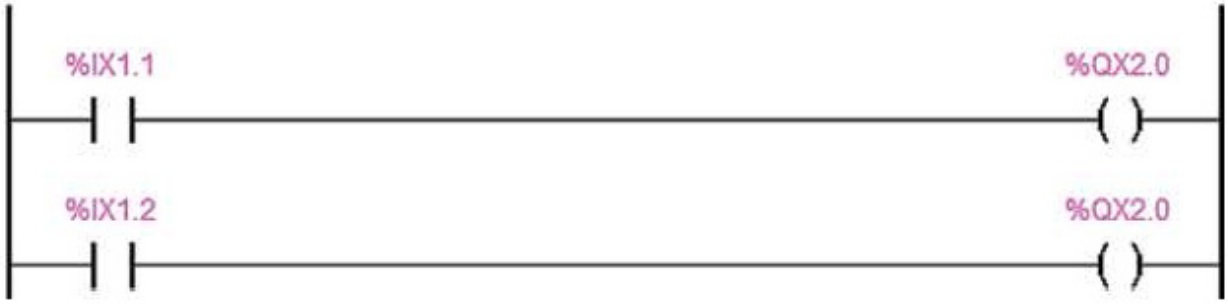


Figure 1.76: One Output, two rungs

Suppose that we have two switches that change between being on and off each time a button is pressed. If at least one of them is turned on, a motor should start:

So, if one of the input %IX1.1 or %IX1.2 gets the state TRUE, the output %QX2.0 becomes TRUE and the motor will start. This looks so simple that it's impossible that it won't work, right? Actually, this rung will not function satisfactorily the way we have designed it here.

Why? Well, because we have a conflict between the rungs since we have used the same address (%QX2.0) on two coils in the program. Splitting the conditions for control of the state of an output or variable in this way is a programming technique that should never be used, even if the program behaves as desired! What problems that can arise are not always easy to determine because it depends upon the logical expressions and the states of the variables that appear in the expression.

This is a result of the way the PLC scans (executes) the program: from left to right and from top to bottom. Furthermore, the physical outputs will not be updated until the entire program code has been executed.

Assume that switches 1 and 2 above the output point in the rung are off. Then the PLC will also set the output logically low. Let's look at two possible scenarios:

1. We turn on switch 1 so that address %IX1.1 gets the state TRUE. When the program is executed, the network is analyzed as follows:

   - %IX1.1 is TRUE therefore %QX2.0 becomes TRUE.
   - %IX1.2 is FALSE therefore %QX2.0 will be set FALSE.

   In other words: When the PLC updates the outputs, the output will remain low, even though the desired function was that the output should be set high when one or both of the inputs is high.

2. Now we turn on switch 2 so that address %I1.2 becomes TRUE. Then the following occurs:

   - %I1.1 is FALSE therefore %Q2.0 becomes FALSE.
   - %I1.2 is TRUE therefore %Q2.0 will be set TRUE.

   We see that no matter whether switch 1 is off or on, the motor will react only to the state of switch 2.

How can we fix a simple program that does the job satisfactorily? Okay, we collect the conditions for control of the output.

Figure 1.77: One Output, two rungs, corrected code

Order of execution relevance

A PLC operates with respect to its basic working rules: Read input data, perform program code, and update outputs. Here, we will study an example to see what significance this has for the way our program codes work.

We can hope that then we will see the difference between the values and states of the PLC physical inputs and outputs and the contents of the addresses or variables that are associated with those same inputs and outputs. This is an extremely important point that many (including instructors and teachers) can struggle to understand.

Two light switches (NO) are connected, each to a discrete input whose address is assigned the variables `SW1` and `SW2`. The following code is implemented in a PLC that is in Run mode:

1. What do you think will happen to the lights when switch 1 is turned on and switch 2 is off?

   - Switch 2 is turned off. Variable `SW2` will then have the state FALSE. So, there is no condition satisfied for the state of variable L1 to become TRUE.
   - Switch 1 is turned on. Therefore, the state of variable `SW2` is changed to TRUE (because of rung number two).
   - Since the state of `SW2` now is TRUE, the state of L2 will also become TRUE. Therefore, Light no. 2 will light but not Light no. 1.

2. What do you think will happen to the lights when switch 2 is turned on and switch 1 is off?

   - Switch 2 is turned on, and the variable `SW2` will have the state TRUE. Therefore, the condition for setting variable `L1` to TRUE is satisfied.
   - Switch 1 is turned off so that the state of the variable `SW2` is changed to FALSE and the state of `L2` will be set to FALSE. Therefore, Light no. 1 will light but not Light no. 2.

6.5 Labels and Jumps

If desired, labels can be used to identify rungs in the code. These are used when you need to jump from one rung to another in the program. Labels are located up on the left corner of the rung, right next to the power rail.

With jumps, the symbol $\rightarrow$ `Label` is used, where `Label` is an arbitrary name (so long as it follows the rule for valid identifiers). A jump can be unconditional or conditional. In unconditional jumps, the jump is directly connected to the rail on the left side, possibly to a contact where you write TRUE instead of the variable name.

In conditional hops, you program a condition as an argument. This condition can be a Boolean variable or a combination of Boolean variables and comparisons. Each label can be used only once within the same POU. If you want to return after the rung you jumped to have been performed, you can insert $\leftarrow$ `RETURN` $\rightarrow$ at the right of the rung.

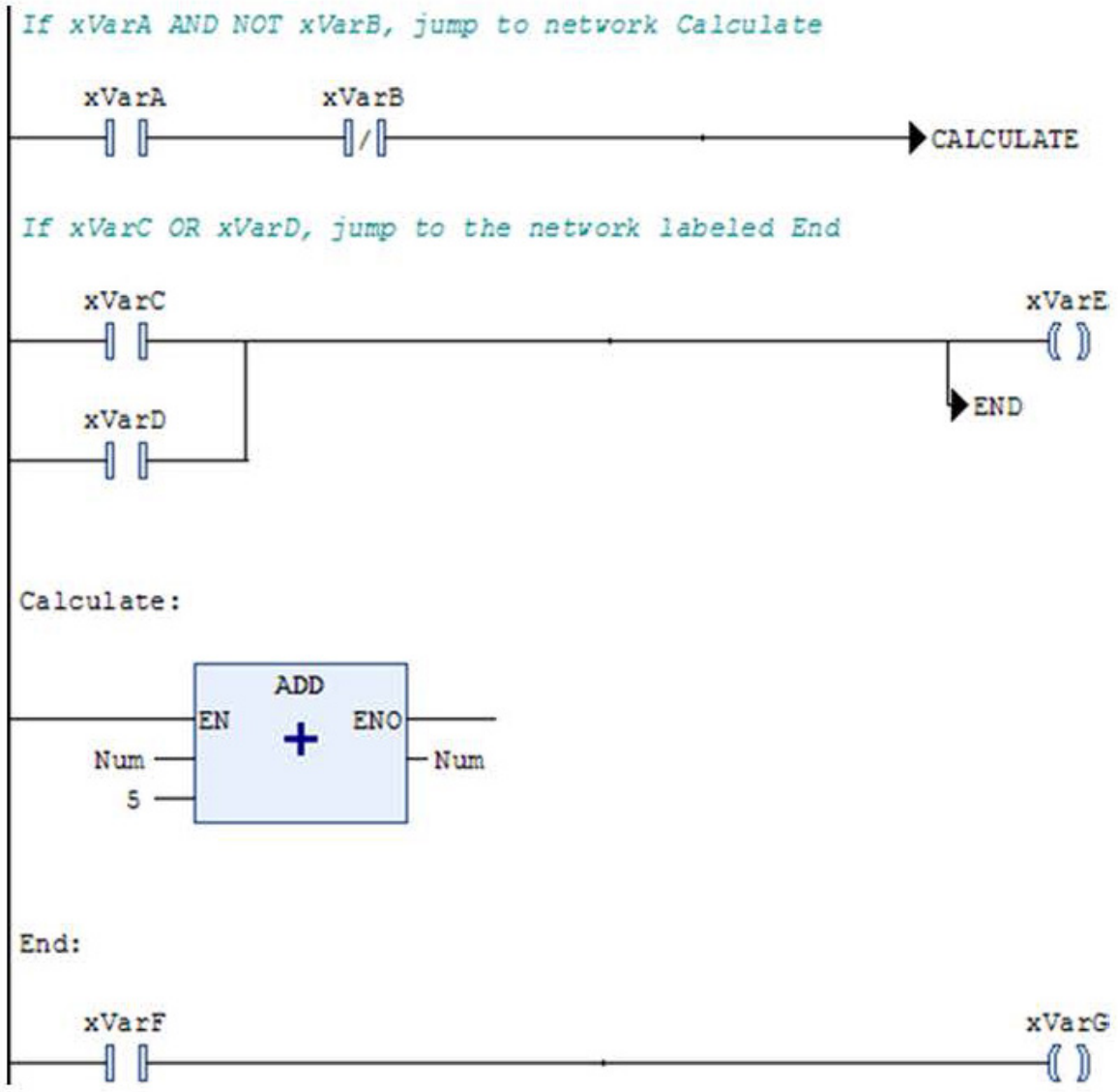


Figure 1.78: LD program with Labels, Jumps and Comments

It is also possible to associate comments to every individual rung. An example of use of labels and comments is shown in Figure 1.78. Here, rung 3 has received the label "calculate" and rung 4 the label "end." A jump has been placed in both the first and the second rungs, in addition to comments. If the condition in the first rung is satisfied, rung 3 is called. Rung 2 is jumped over and is never executed.

If the condition in rung 1 is not satisfied, rung 2 is performed as usual. If either variable `xVarC` or `xVarD` is TRUE, `xVarE` is set TRUE, and in addition, the program jumps to rung 4. In this case, rung 3 is not executed

6.6 Use of standard functions in LD

In the program example earlier, the function ADD is used (here to increment the content in the integer variable Num by 5 each time the rung is executed). So even though LD is a language that originally was designed for programming logical (Boolean) conditions for control of discrete outputs, it is fully capable of writing program algorithms that involve other data types.

For example, there is often a need to compare values of an analog signal with a particular value in order to perform actions based upon the result of the comparison. An example of this is to turn off a heating element when a temperature reaches a desired value. In order to perform this and other tests in LD, one can use functions that are defined in the standard. The concept of EN and ENO were previously introduced in subsection 5.9.

Remember that EN is an acronym for Enable and is an extra input argument in the standard's defined functions and function blocks. When the state of this input is TRUE, the function is performed. It is this EN input that makes it possible to integrate functions in LD (Figure 1.79). Even though functions can be integrated into the LD code, there are

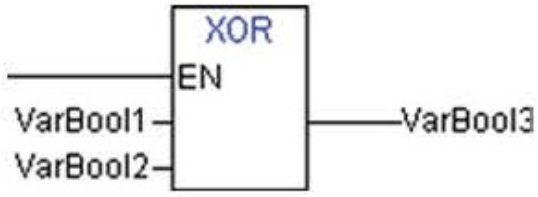


Figure 1.79: XOR function with EN input

other (and probably better) ways to solve problems that require use of arithmetic functions, functions for comparison, text-strings, etc. It is a matter of utilizing the strength that lies in the seamless interaction among POUs. If there is a requirement to perform a number of calculations, these can be written in a separate POU that is programmed in Structured Text.

There are several ways to integrate functions in LD. Some manufacturers have chosen to use graphic blocks with the same symbols as function block diagram (FBD). Execution of these blocks is then controlled by means of a conditional activation of the EN input.

Another possible implementation is by means of a combination of graphic blocks and text-based blocks where it is possible to write the code in Structured Text. Both variants are illustrated in Figures 1.80 and 1.81.

Figure 1.80: Use of EN blocks with FBD symbol

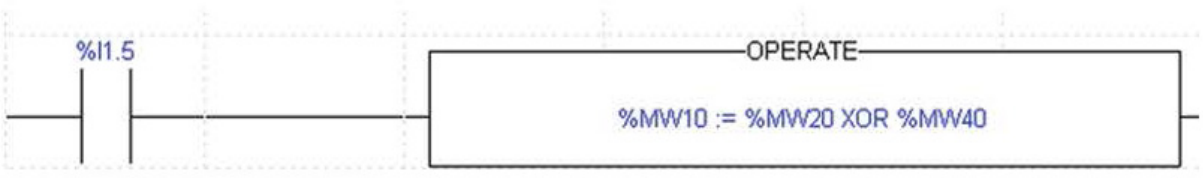


Figure 1.81: Integration of functions by use of a text-based block

6.7 Development of FBs in LD

Using FBs in LD is simple and is done in the same way no matter what type of function block is involved. An instance of the block must be declared in the declaration field in the POU. This is usually quickly accomplished in most development tools or takes place automatically when the block is added to the code. It is not always necessary to assign fixed connections to all inputs and outputs of the function block since FB inputs and outputs can be addressed indirectly.

Example 20. Let us use a timer in the example of motor control that was shown in Figure 1.68. Assume that the control is to satisfy the following problem: We start and stop the motor with pulse switches, and we want to have 5 seconds delay before the motor is switched on. One possible implementation of the program is shown in Figure 1.82. The timer can be

```
PROGRAM MotorControl
VAR
  Start      AT %IX0.0    :BOOL;
  Stop       AT %IX0.1    :BOOL;
  Motor      AT %QX0.0    :BOOL;
  Run                     :BOOL;
  Five_sec                :TON;
END_VAR
```

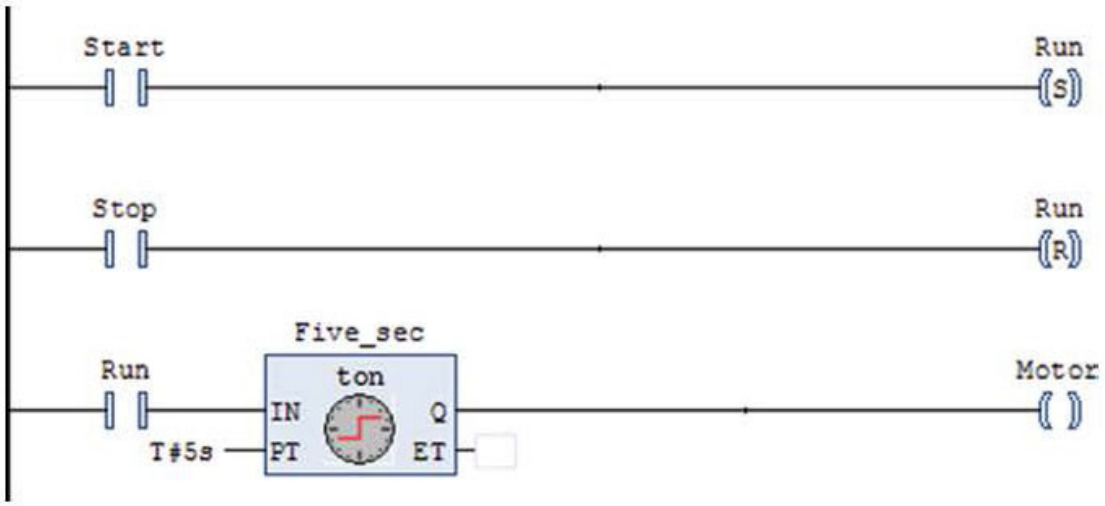


Figure 1.82: Program example with use of timer

used for many purposes. In Figure 1.83 in the following text, you will see how you can make your own function block that generates a pulse train, where the user determines the duration of the high and low periods △

Example 21. Since this is a function block, we must declare the different classes of variable. Input and output variables will constitute "connections" when the block is used in another POU. Any internal variables are used to implement the block's function and operation and will not be visible to anyone using the function block later. Also, notice the following two things:

- The two input variables of the type TIME below are both given an initial value of 1 second. When the block is used, it will function (with the output 1 second on and 1 second off) even though the user has not stated any time for these inputs.
- The object reference of the timer outputs (TimerName.Q), which is a Boolean variable, is associated with contact symbols.

```
FUNCTION_BLOCK      Pulse
VAR_INPUT
   StartPulseTrain  :BOOL;
   TimeOn           :TIME     := t#1s;
   TimeOff          :TIME     := t#1s;
END_VAR
VAR_OUTPUT
   PulseTrain       :BOOL;
END_VAR
VAR
   Timer1           :TON;
   Timer2           :TON;
END_VAR
```

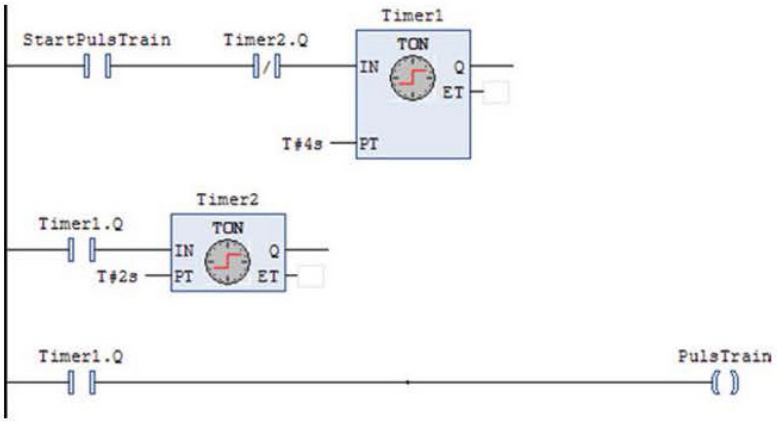
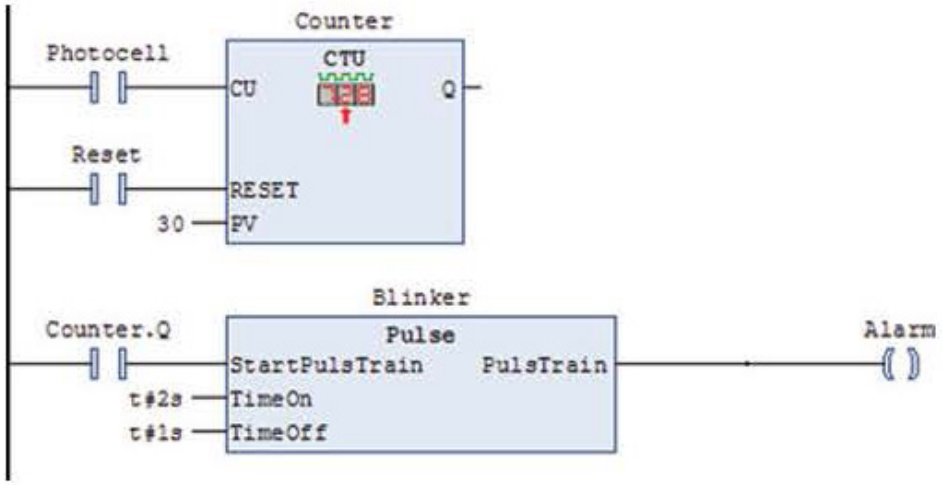


Figure 1.83: Self programmed FB in LD program

△

Example 22. Assume that we want to set an alarm when 30 items have passed a photocell on a conveyor belt. The alarm is a light that is to blink 2 seconds on and 1 second off. When a Reset button is activated, the alarm and the counter are set to zero. One possible program code that implements this could be as follows: Note the declaration of "Blinker." As soon as

```
PROGRAM ItemCounting
VAR
    Photocell       AT %IX1.0    :BOOL;
    Reset           AT %IX1.1    :BOOL;
    Band            AT %QX2.0    :BOOL;
    Alarm           AT %QX2.1    :BOOL;
    Counter         :CTU;
    Blinker         :Pulse;
END_VAR
```




```
END_PROGRAM
```

you have programmed an FB, it will be available for use in another POU. You can, as with the function blocks of the standard, declare as many instances as you want of one and the same FB, as long as each instance has a unique name.

Here, we have only one instance of our new FB "Pulse," and we have chosen to call it "Blinker." The graphic design of the block is generated automatically, and the number of inputs and outputs is a direct result of the variables that we have declared under VAR_INPUT and VAR_OUTPUT in the declaration of the FB.

△

6.8 Structured Programming in LD

If a process is sequential or mainly combinatoric, this can have significance for the selection of programming languages. Even though sequential control is perhaps easiest to program in sequential function chart (SFC), there is no obstacle to the use of LD, FBD, or even ST. Since LD is the language that has traditionally been used most often, it is very common to use this language for specifically sequential systems as well. We have seen an example of a mixing process. This process had a sequential nature, but we solved it in LD in a simple way by diligent use of flank contacts and Set and Reset coils. The sequence diagram that we already have prepared will naturally be helpful in the example.

It can be a challenge in sequential controls, particularly when the complexity and scope become larger, in that conflicts can arise that are associated with conditions that are ambiguous. This means that there can occur cases of the same condition that at times in the sequence cause a different event than that which was caused at a different time in the sequence. In other words, it can be necessary to take into account what phase of the sequence we find ourselves in. As we shall see, this is not significant if the program code is implemented with SFC. The same philosophy that lies behind SFC can be used in constructing program codes in one of the standard's other languages.

The technique is to use a Boolean variable (a flag) or a memory (bistable flip-flop) of the type RS for each of the states in the sequential process. When the conditions for the program to go from one state to the next are satisfied, the previous state is reset simultaneously.

After the code for activating and deactivating states has been written, the state variables (or outputs from RS flip-flops if they have been used for states) are used as conditions to control actions and instructions that are to be performed.

In order to demonstrate the method, we will again use the mixing process that we previously studied in. Let us consider a flowchart for this process

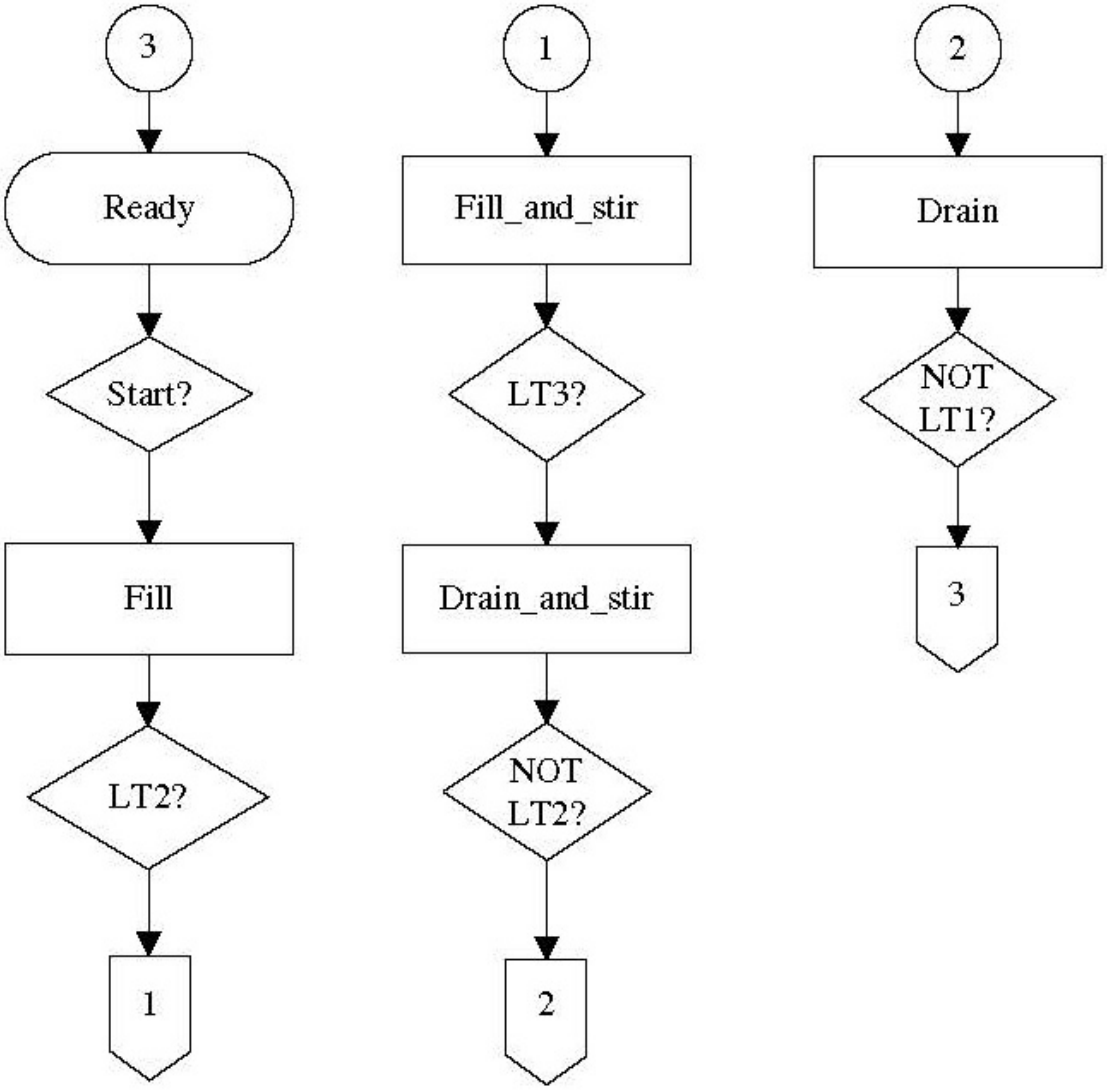


Figure 1.84: Flowchart for the Mixing Process

There, we used ordinary language to describe the states. If we now are to write a code based on the flowchart, we must use identifiers that are permitted and that will be accepted by the software. Examples of such identifiers can be Fill, Stir, Warm_up, and so on. The same is true of transitional conditions that also can be specified more concretely. It is recommended that these be written in pseudocode or as logical expressions. In order to make the transition to code simpler, it is therefore advisable to design the flowchart in an implementation-friendly language, while at the same time trying to use state names that describe what is to happen in the states in question.

Looking at the flowchart, we can easily identify states in the process as the rectangular blocks in the flowchart, in addition to the initial state "Ready." We start by declaring all states and the other variables. If this had been a physical process, we would naturally have associated addresses to the I/O variables. The program code for activation and deactivation

```
VAR
  Ready, Fill, Fill_and_Stir, Drain_and_Stir, Drain    : BOOL;
  Start, LT1, LT2, LT3, MV1, MV2, Motor, Stirrer       : BOOL;
  RE1, RE2, RE3  : R_TRIG;
  FE1, FE2       : F_TRIG;
END_VAR
```

of the states of the process is shown in Figure 1.85 (The code is written in CODESYS v2.3.x, which does not have flank contacts in the library. That is the reason for using the function blocks R_Trig and F_Trig.)

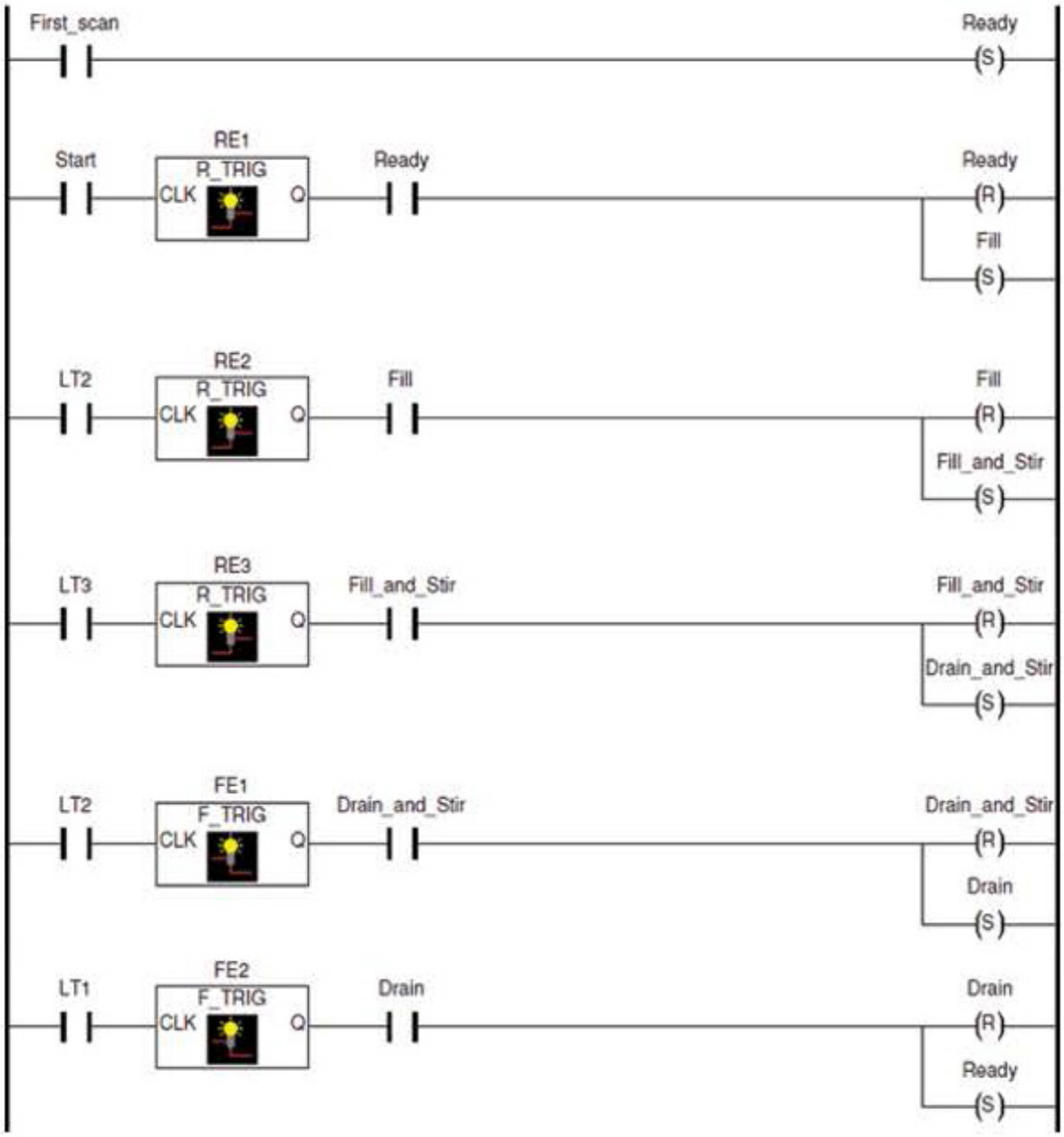


Figure 1.85: Controlling states

In the code, we see that the flags for states are set and reset successively downward. In order for the code to be unambiguous, the same flags are also used for a portion of the conditions for activating and deactivating the states. For example, the state "Fill" is activated only when the previous state (Ready) is active. When starting the program, it is necessary to get the program to activate the state Ready. This can be solved by giving Ready the initial value TRUE. Alternatively, we can do as we have done here, namely, use a special system flag that is available in all (?) PLCs. Individual manufacturers have defined a function for this, but others have made the flag accessible via a fixed system address. A common designation

for this flag is First Scan because the flag has the state TRUE only during the course of the first scan. If you do not have access to such a flag, you can straight away declare a Boolean object that you initially set to TRUE. Right at the end of the program code, you set the object to FALSE in this way: When the code for activation and deactivation of the

states is written, the code for control of outputs is built up by the use of state flags as conditions (Figure 1.86): The sequence in this example was simple and had few states. Use

Figure 1.86: Program code based on the state flags

of this methodology in this example also resulted in more code than we previously ended up. Nevertheless, this method of proceeding is absolutely preferable, particularly for sequential processes. Not only does a methodical approach make it simpler to develop the code, there is a major benefit in code that is guaranteed to be unambiguous and, one hopes, free of errors if you master this technique.

A better alternative to using coils of the Set and Reset type is to use flip-flops. This does not result in less coding, but the clarity of the code improves considerably. As always, it is wise to be consistent during programming. In the following examples, I have used only

Reset-dominant memories. The conditions for changing state are implemented on Set inputs. The output of the RS block for the next state is always used on Reset inputs. In this way, the current state is deactivated at the same instant that the next state is activated.

Flowchart vs RS based LD code

In order to clarify the transition from the flowchart to an LD code that is structured around RS flip-flops, we can study a section of the flowchart in Figure 1.85. Here, we see the RS flip-flop that represents the state Fill along with the conditions for Set and Reset of the state. The rules for coding of a state are:

- Previous states, together with the transition conditions, constitute the conditions for Set inputs to the RS flip-flop. If there are several previous states, we get conditions in parallel at the Set input.
- The next state, and only that, constitutes the conditions for Reset of the state. If there are several next states, we get conditions in parallel at the Reset input (Figure 1.87).

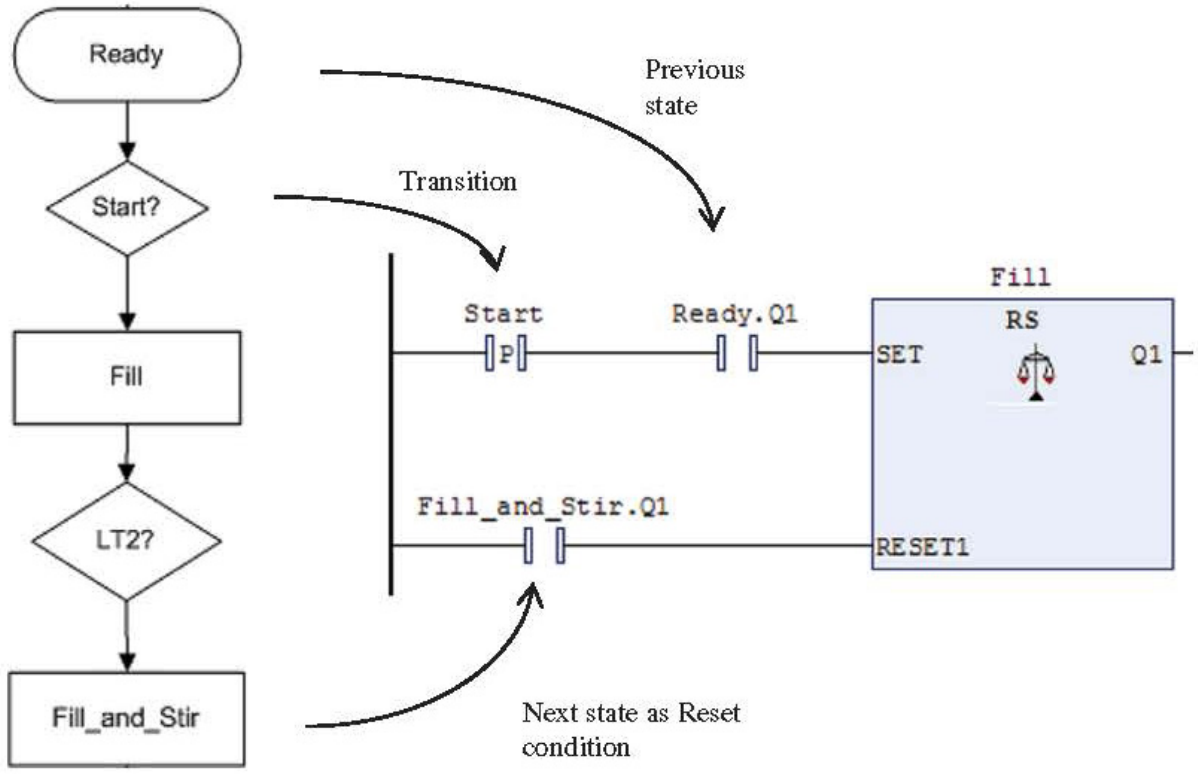


Figure 1.87: From Flowchart to LD Code

Let's take a look at the mixing process. Figure below shows an alternative code to the code in Figure 1.85. We see that the clarity of the state structure is improved. Also, notice the declaration and the use of the Boolean variable First_scan. (The variable is set FALSE at the end of the code.) This code was developed with CODESYS v3.4, which does have flank contacts (-|P|- and -|N|-).

```
Program MixingProcess
Var
  First_scan  :BOOL      :=TRUE;
  Ready, Full, Fill_and_stir, Drain_and_Stir, Drain    :RS;
  Start, LT1, LT2, LT3, Motor, Stirrer, MV1, MV2       :BOOL;
END_VAR
```