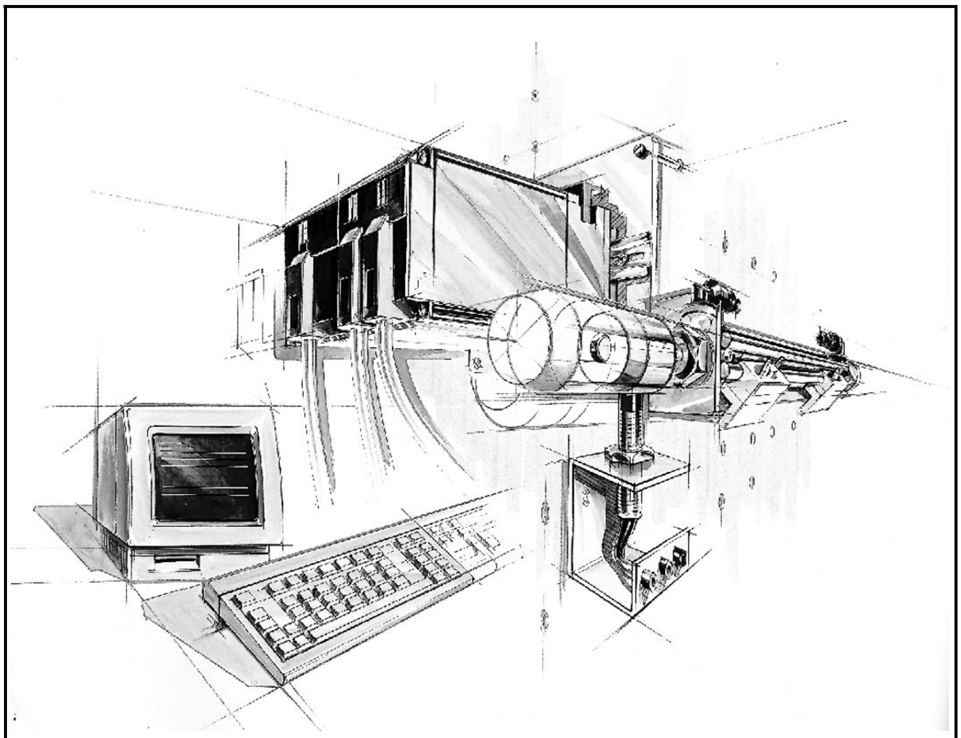
Learning System for Automation and Communications

# Programmable logic controllers



Basic level TP301
**Textbook**

FESTO DIDACTIC

| | |
|---|---|
| Order No.: | 093311 |
| Description: | PLC LB GS |
| Designation: | D.LB-TP301-1-GB |
| Edition: | 07/1995 |
| Layout: | 26.07.95, F. Ebel |
| Graphics: | B. Böhland, D. Schwarzenberger |
| Authors: | R. Bliesener, F. Ebel, C. Löffler, B. Plagemann, H. Regber, E. v. Terzi, A. Winter |

## *Preface*

The programmable logic controller represents a key factor in industrial automation. Its use permits flexible adaptation to varying processes as well as rapid fault finding and error elimination.

This textbook explains the design of a programmable logic controller and its interaction with peripherals.

One of the main focal points of the textbook deals with the new international standard for PLC programming, the IEC-1131, Part 3. This standard takes into account expansions and developments, for which no standardised language elements existed hitherto.

The aim of this new standard is to standardise the design, functionality and the programming of a PLC in such a way as to enable the user to easily operate with different systems.

In the interest of continual further improvement, all readers of this book are invited to make contributions by way suggestions, ideas and constructive criticism.

February 1995 The authors

## *Table of Contents*

# Chapter 1

# The PLC in automation technology

## *1.1 Introduction*

The first **P**rogrammable **L**ogic **C**ontroller (PLC) was developed by a group of engineers at General Motors in 1968, when the company were looking for an alternative to replace complex relay control systems.

The new control system had to meet the following requirements:

- Simple programming
- Program changes without system intervention (no internal rewiring)
- Smaller, cheaper and more reliable than corresponding relay control systems
- Simple, low cost maintenance

Subsequent development resulted in a system which enabled the simple connection of binary signals. The requirements as to how these signals were to be connected was specified in the control program. With the new systems it became possible for the first time to plot signals on a screen and to file these in electronic memories.

Since then, three decades have passed, during which the enormous progress made in the development of micro electronics did not stop short of programmable logic controllers. For instance, even if program optimisation and thus a reduction of required memory capacity initially still represented an important key task for the programmer, nowadays this is hardly of any significance.

Moreover, the range of functions has grown considerably. 15 years ago, process visualisation, analogue processing or even the use of a PLC as a controller, were considered as Utopian. Nowadays, the support of these functions forms an integral part of many PLCs.

The following pages in this introductory chapter outline the basic design of a PLC together with the currently most important tasks and applications.

## *1.2 Areas of application of a PLC*

Every system or machine has a controller. Depending on the type of technology used, controllers can be divided into pneumatic, hydraulic, electrical and electronic controllers. Frequently, a combination of different technologies is used. Furthermore, differentiation is made between hard-wired programmable (e.g. wiring of electro-mechanical or electronic components) and programmble logic controllers. The first is used primarily in cases, where any reprogramming by the user is out of the question and the job size warrants the development of a special controller. Typical applications for such controllers can be found in automatic washing machines, video cameras, cars.

However, if the job size does not warrant the development of a special controller or if the user is to have the facility of making simple or independent program changes, or of setting timers and counters, then the use of a universal controller, where the program is written to an electronic memory, is the preferred option. The PLC represents such a universal controller. It can be used for different applications and, via the program installed in its memory, provides the user with a simple means of changing, extending and optimising control processes.

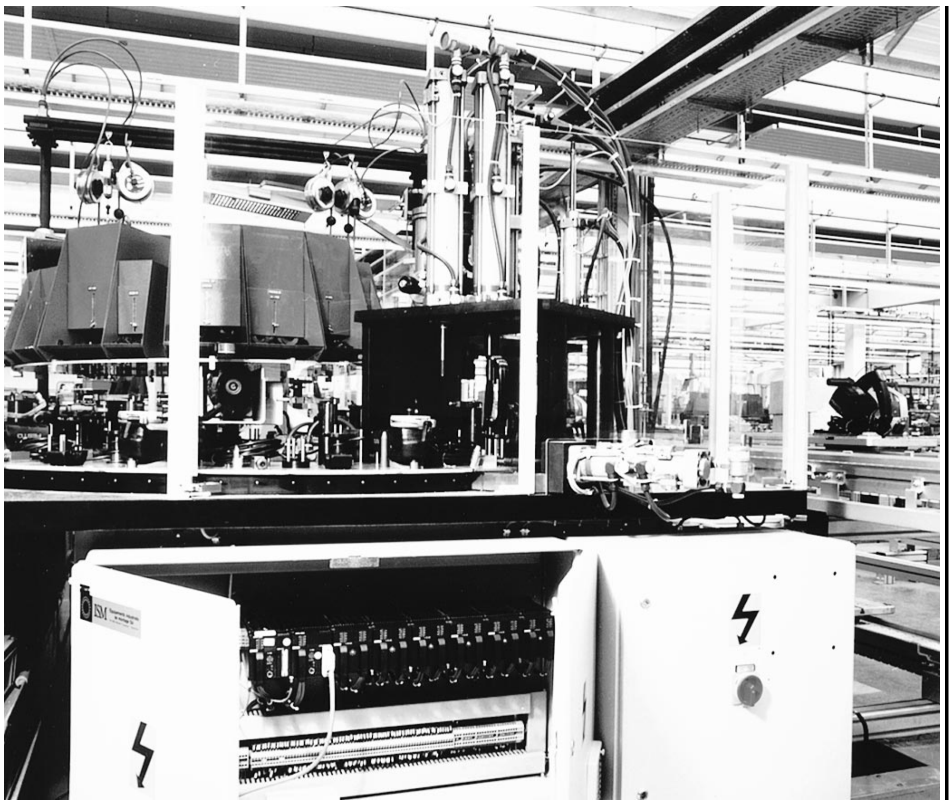


*Fig. B1.1: Example of a PLC application*

The original task of a PLC involved the interconnection of input signals according to a specified program and, if "true", to switch the corresponding output. Boolean algebra forms the mathematical basis for this operation, which recognises precisely two defined statuses of one variable: "0" and "1" (see also chapter 3). Accordingly, an output can only assume these two statuses. For instance, a connected motor could therefore be either switched on or off, i.e. controlled.

This function has coined the name PLC: **Programmable logic controller**, i.e. the input/output behaviour is similar to that of an electro-magnetic relay or pneumatic switching valve controller; the program is stored in an electronic memory.

However, the tasks of a PLC have rapidly multiplied: Timer and counter functions, memory setting and resetting, mathematical computing operations all represent functions, which can be executed by practically any of today's PLCs.

The demands to be met by PLC's continued to grow in line with their rapidly spreading usage and the development in automation technology. Visualisation, i.e. the representation of machine statuses such as the control program being executed, via display or monitor. Also controlling, i.e. the facility to intervene in control processes or, alternatively, to make such intervention by unauthorised persons impossible. Very soon, it also became necessary to interconnect and harmonise individual systems controlled via PLC by means of automation technology. Hence a master computer facilitates the means to issue higher-level commands for program processing to several PLC systems.

The networking of several PLCs as well as that of a PLC and master computer is effected via special communication interfaces. To this effect, many of the more recent PLCs are compatible with open, standardised bus systems, such as Profibus to DIN 19 245. Thanks to the enormously increased performance capacity of advanced PLCs, these can even directly assume the function of a master computer.

At the end of the seventies, binary inputs and outputs were finally expanded with the addition of analogue inputs and outputs, since many of today's technical applications require analogue processing (force measurement, speed setting, servo-pneumatic positioning systems). At the same time, the acquisition or output of analogue signals permits an actual/setpoint value comparison and as a result the realisation of automatic control engineering functions, a task, which widely exceeds the scope suggested by the name (programmable logic controller).

The PLCs currently on offer in the market place have been adapted to customer requirements to such an extent that it has become possible to purchase an eminently suitable PLC for virtually any application. As such, miniature PLCs are now available with a minimum number of inputs/outputs starting from just a few hundred Pounds. Also available are larger PLCs with 28 or 256 inputs/outputs.

Many PLCs can be expanded by means of additional input/output, analogue, positioning and communication modules. Special PLCs are available for safety technology, shipping or mining tasks. Yet further PLCs are able to process several programs simultaneously – (multitasking). Finally, PLCs are coupled with other automation components, thus creating considerably wider areas of application.

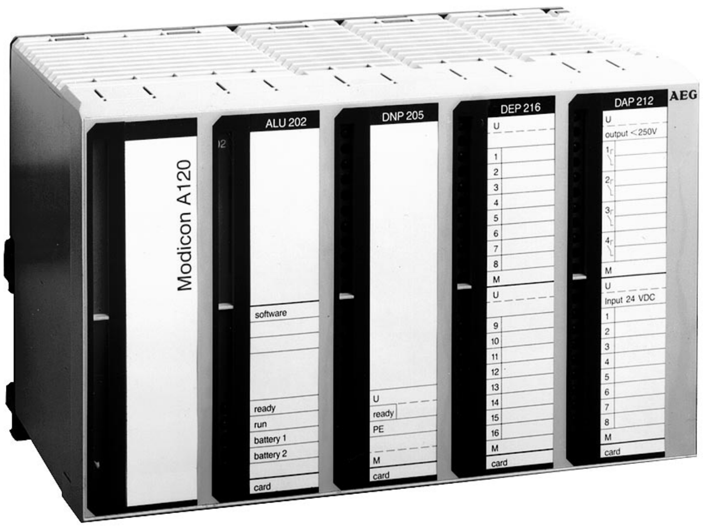


*Fig. B1.2:
Example of a PLC:
AEG Modicon A120*

## *1.3 Basic design of a PLC*

The term ’programmable logic controller’ is defined as follows by IEC 1131, Part 1:

*"A digitally operating electronic system, designed for use in an industrial environment, which uses a programmable memory for the internal storage of user-oriented instructions for implementing specific functions such as logic, sequencing, timing, counting and arithmetic, to control, through digital or analog inputs and outputs, various types of machines or processes. Both the PC and its associated peripherals are designed so that they can be easily integrated into an industrial control system and easily used in all their intended functions."*

A programmable logic controller is therefore nothing more than a computer, tailored specifically for certain control tasks.

Fig. B1.3 illustrates the system components of a PLC.

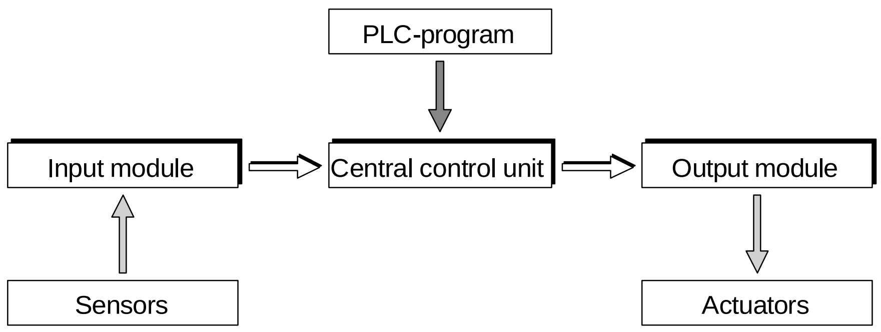


*Fig. B1.3: System components of a PLC*

The function of an input module is to convert incoming signals into signals which can be processed by the PLC and to pass these to the central control unit. The reverse task is performed by an output module. This converts the PLC signal into signals suitable for the actuators.

The actual processing of the signals is effected in the central control unit in accordance with the program stored in the memory.

The program of a PLC can be created in various ways: via assembler-type commands in ’statement list’, in higher-level, problem-oriented languages such as structured text or in the form of a flow chart such as represented by a sequential function chart. In Europe, the use of function block diagrams based on function charts with graphic symbols for logic gates is widely used. In America, the ’ladder diagram’ is the preferred language by users.

Depending on how the central control unit is connected to the input and output modules, differentiation can be made between compact PLCs (input module, central control unit and output module in one housing) or modular PLCs.

Fig. B1.4 shows the FX0 controller by Mitsubishi representing a compact PLC as an example.

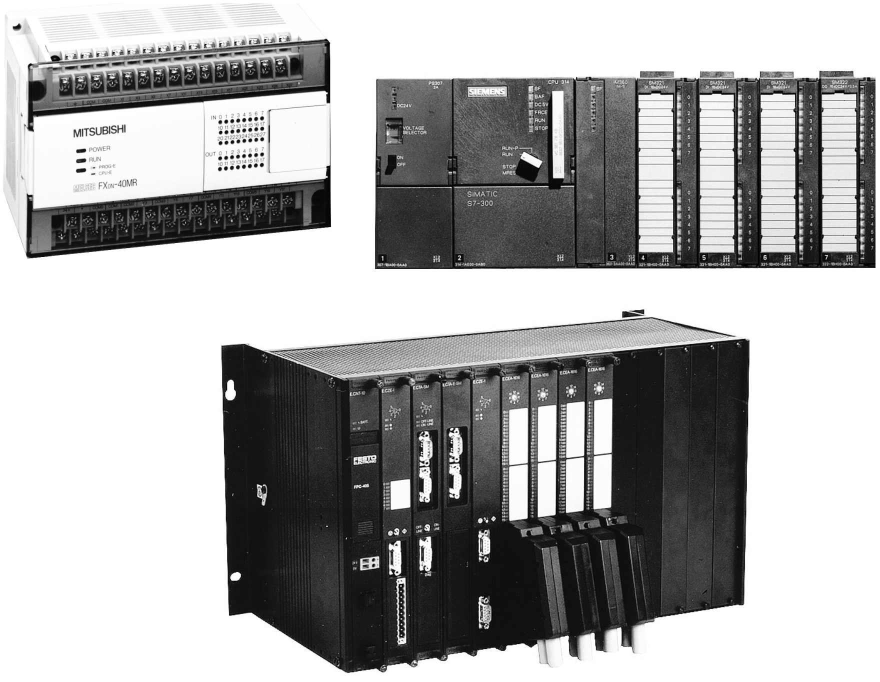


*Fig. B1.4:
Compact PLC (Mitsubishi FX0), modular PLC (Siemens S7-300), PLC plug-in cards (Festo FPC 405)*

Modular PLCs may be configured individually. The modules required for the practical application – apart from digital input/output modules which can, for instance, include analogue, positioning and communication modules – are inserted in a rack, where individual modules are linked via a bus system. This type of design is also known as series technology. Two examples of modular PLCs are shown in figs. B1.2 and B1.4. These represent the familiar modular PLC series by AEG Modicon and the new S7-300 series by Siemens.

A wide range of variants exists, particularly in the case of more recent PLCs. These include both modular as well as compact characteristics and important features such as spacing saving, flexibility and scope for expansion.

The card format PLC is a special type of modular PLC, developed during the last few years. With this type, individual or a number of printed circuit board modules are in a standardised housing. The Festo FPC 405 is representative of this type of design (Fig. B1.4).

The hardware design for a programmable logic controller is such that it is able to withstand typical industrial environments as regard signal levels, heat, humidity, fluctuations in current supply and mechanical impact.

## *1.4 The new PLC standard IEC 1131*

Previously valid PLC standards focussing mainly on PLC programming were generally geared to current state of the art technology in Europe at the end of the seventies. This took into account non-networked PLC systems, which primarily execute logic operations on binary signals. DIN 19 239, for example, specifies programming languages which possess the corresponding language commands for these applications.

Previously, no equivalent, standardised language elements existed for the PLC developments and system expansions made in the eighties, such as processing of analogue signals, interconnection of intelligent modules, networked PLC systems etc. Consequently, PLC systems by different manufacturers required entirely different programming.

Since 1992, an international standard now exists for programmable logic controllers and associated peripheral devices (programming and diagnostic tools, testing equipment, man-to-machine interfaces etc.). In this context, a device configured by the user and consisting of the above components is known as a PLC system.

The new IEC 1131 standard consists of five parts:

- Part 1: General information
- Part 2: Equipment requirements and tests
- Part 3: Programming languages
- Part 4: User guidelines (in preparation with IEC)
- Part 5: Messaging service specification (in preparation with IEC)

Parts 1 to 3 of this standard were adopted unamended as European Standard EN 61 131, Parts 1 to 3. As such, they also hold the status of a German standard.

The purpose of the new standard was to define and standardise the design and functionality of a PLC and the languages required for programming to the extent where users were able to operate using different PLC systems without any particular difficulties.

The next chapters will be dealing with this standard in greater detail. However, for the moment the following information should suffice:

- The new standard takes into account as many aspects as possible regarding the design, application and use of PLC systems.
- The extensive specifications serve to define open, standardised PLC systems.
- Manufacturers must conform to the specifications of this standard both with regard to purely technical requirements for the PLC as well as the programming of controllers.
- Any variations must be fully documented for the user.

After initial reservations, a relatively large group of interested people (PLCopen) has been formed to support this standard. A large number of major PLC suppliers are members of the association, i.e. Allen Bradley, Klöckner-Moeller, Philips, to mention a few. PLC manufacturers such as Siemens or Mitsubishi also offer control and programming systems conforming to IEC-1131.

The initial programming systems are already available in the market and others are being developed at the time of going to press. The norm therefore stands a good chance of being accepted and succeeding. Not least, it is hoped that this textbook will, to a certain extent, help to contribute to this.

# Chapter 2

# Fundamentals

## *2.1 The decimal number system*

Characteristic of the decimal number system used in general is the linear array of digits and their significant placing. The number 4344, for instance, can be represented as follows:

4344 = 4 x 1000 + 3 x 100 + 4 x 10 + 4 x 1

Number 4 on the far left is of differing significance to that of number 4 on the far right.

The basis of the decimal number system is the availability of 10 different digits (decimal: originating from the Latin 'decem' = 10 ). These 10 different digits permit counting from 0 to 9. If counting is to exceed the number 9, this constitutes a carry over to the next place digit. The significance of this place is 10, and the next carry over takes place when 99 is reached.

The number 71.718.711 is to be used as an example:

*Example*

| $10^7$ | $10^6$ | $10^5$ | $10^4$ | $10^3$ | $10^2$ | $10^1$ | $10^0$ |
|---|---|---|---|---|---|---|---|
| 7 | 1 | 7 | 1 | 8 | 7 | 1 | 1 |

As can be seen from the above, the significance of the "7" on the far left is 70.000.000 = 70 million, whereas the significance of the "7" in the third place from the right is 700.

The digit on the far right is referred to as the least significant digit, and the digit on the far left as the most significant digit.

Any number system can be configured on the basis of this example, the fundamental structure can be applied to number systems of any number of digits. Consequently, any computing operations and computing methods which use the decimal number system can be applied with other number systems.

## *2.2 The binary number system*

We are indebted to Leibnitz, who applied the structures of the decimal number system to two-digit calculation. As long ago as 1679, this created the premises essential for the development of the computer, since electrical voltage or electrical current only permits a calculation using just two values: e.g. "current on", "current off". These two values are represented in the form of digits: "1" and "0".

If one were to be limited to exactly 2 digits per place of a number, then a number system would be configured as follows:

| $2^7$=128 | $2^6$=64 | $2^5$=32 | $2^4$=16 | $2^3$=8 | $2^2$=4 | $2^1$=2 | $2^0$=1 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |

*Example*

The principle is exactly the same as that of the method used to create a decimal number. However, only two digits are available, which is why the significant place is not calculated to the base $10^x$, but to the base $2^x$. Hence the lowest significant number on the far right is$^0$ = 1, and of the next place $2^1 = 2$ etc. Because of the exclusive use of two digits, this number system is known as the binary or also the dual number system.

Up to a maximum of

$$2^8 - 1 = 256 - 1 = 255$$

can be calculated with eight places, which would be the number $1111\ 1111_2$.

The individual places of the binary number system can adopt one of the two digits 0 or 1. This smallest possible unit of the binary system is termed 1 bit.

In the above example, a number consisting of 8 bits, i.e. one byte, has been configured (in a computer using 8 electrical signals representing either "voltage available" or "voltage not available" or "current on" or "current off".) The number considered, $1011\ 0001_2$, assumes the decimal value $177_{10}$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $1 \times 2^7$ | $0 \times 2^6$ | $1 \times 2^5$ | $1 \times 2^4$ | $0 \times 2^3$ | $0 \times 2^2$ | $0 \times 2^1$ | $1 \times 2^0$ |
| = 128 | | + 32 | + 16 | | | | + 1 |
| = 177 | | | | | | | |

*Example*

## 2.3 The BCD code

For people used to dealing with the decimal system, binary numbers are difficult to read. For this reason , a more easily readable numeral representation was introduced, i.e. the binary coded decimal notation, the so-called BCD code (binary coded decimal). With this BCD code, each individual digit of the decimal number system is represented by a corresponding binary number:

Table B2.1: Representation of decimal numbers in BCD code

| | |
|---|---|
| $0_{10}$ | $0000_{BCD}$ |
| $1_{10}$ | $0001_{BCD}$ |
| $2_{10}$ | $0010_{BCD}$ |
| $3_{10}$ | $0011_{BCD}$ |
| $4_{10}$ | $0100_{BCD}$ |
| $5_{10}$ | $0101_{BCD}$ |
| $6_{10}$ | $0110_{BCD}$ |
| $7_{10}$ | $0111_{BCD}$ |
| $8_{10}$ | $1000_{BCD}$ |
| $9_{10}$ | $1001_{BCD}$ |

4 digits in binary notation are therefore required for the 10 digits in the decimal system. The discarded place (in binary notation, the numbers 0 to 15 may be represented with 4 digits) is accepted for the sake of clarity.

The decimal number 7133 is thus represented as follows in the BCD code:

$$0111\ 0001\ 0011\ 0011_{BCD}$$

16 bits are therefore required to represent a four digit decimal number in the BCD code. BCD coded numbers are often used for seven segment displays and coding switches.

## 2.4 The hexadecimal number system

The use of binary numbers is often difficult for the uninitiated and the use of the BCD code takes up a lot of space. This is why the octal and the hexadecimal system were developed. Three digits are always combined in the case of the octal number system. This permits counting from 0 to 7, i.e. counting in "eights".

Alternatively, 4 bits are combined with the hexadecimal number system. 4 bits permit the representation of the numbers 0 to 15, i.e. counting in "sixteens". The digits 0 to 9 are used to represent these numbers in digits, followed by the letters A, B, C, D, E and F where A = 10, B = 11, C = 12, D = 13, E = 14 and F = 15. The significant place of the individual digits is to the base 16.

| $16^3$=4096 | $16^2$=256 | $16^1$=16 | $16^0$=1 |
|---|---|---|---|
| 8 | 7 | B | C |

*Example*

The number $87BC_{16}$ given as an example therefore reads as follows:

$$8 \times 16^3 + 7 \times 16^2 + 11 \times 16^1 + 12 \times 16^0 = 34\ 748_{10}$$

## *2.5 Signed binary numbers*

Up to now, we have dealt solely with whole positive numbers, not taking into account negative numbers. To enable working with these negative numbers, it was decided that the most significant bit on the far left of a binary number is to be used to represent the preceding sign: "0" thus corresponds to "+" and "1" corresponds to "–".

Hence $1111\ 1111_2 = -127_{10}$ and $0111\ 1111_2 = +128_{10}$

Since the most significant bit has been used, one bit less is available for the representation of a signed number. The following range of values is obtained for the representation of a 16 digit binary number:

| ***Integer*** | ***Range of values*** |
|---|---|
| unsigned | 0 to 65535 |
| signed | -32768 to +32767 |

*Table B2.2: Range of values for binary numbers*

## *2.6 Real numbers*

Although it is now possible for whole positive and whole signed numbers to be represented with 0 or 1 , there is still the need for points or real numbers.

In order to represent a real number in computer binary notation, the number is split into two groups, a power of ten and a multiplication factor. This is also known as the scientific representation of digits.

The number 27,3341 is thus converted into 273 341 x $10^{-4}$. Two whole signed numbers are therefore required for a real number to be represented in a computer.

## 2.7 Generation of binary and digital signals

As has already become clearly apparent in the previous section, all computers and as such all PLCs operate using binary or digital signals. By binary signal, we understand a signal which recognises only two defined values.

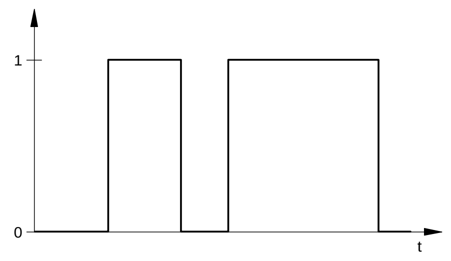


*Fig. B2.1: Binary signal*

These values are termed "0" or "1", the terms "low" and "high" are also used. The signals can be very easily realised with contacting components. An actuated normally open contact corresponds to a logic 1-signal and an unactuated one to a logic 0-signal. When working with contactless components, this can give rise to certain tolerance bands. For this reason, certain voltage ranges have been defined as logic 0 or logic 1 ranges.

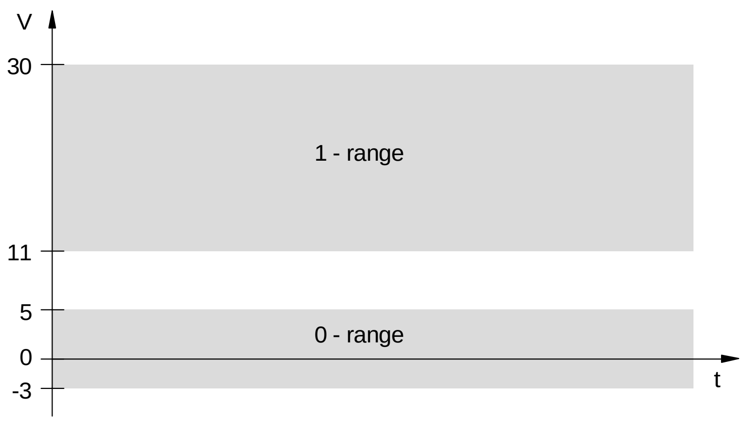


*Fig. B2.2: Voltage ranges*

IEC 1131-2 defines a value range of -3 V to 5 V as logic 0-signal, and 11 V to 30 V as logic 1-signal (for contactless sensors). This is binding for PLCs, whose device technology is to conform to IEC 1131-2. In current practice, however, other voltage ranges can often be found for logic 0- and 1-signal. Widely used are: -30 V to +5 V as logic 0, 13 V to 30 V as logic 1.

Unlike binary signals, digital signals can assume any value. These are also referred to as value stages. A digital signal is thus defined by any number of value stages. The change between these is non-sequential. The following illustration shows three possible methods of converting an analogue signal into a digital signal.

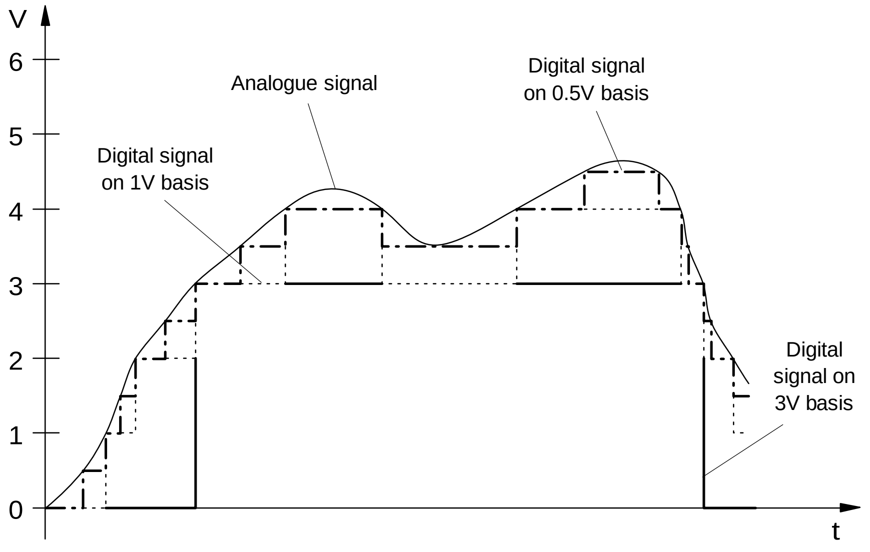


*Fig. B2.3:*
*Conversion of an analogue signal into a digital signal*

Digital signals may be formed from analogue signals. This method is for instance used for analogue processing via PLC. Accordingly, the analogue input signal within a range of 0 to 10 V is reduced into a series of step values. Depending on the quality of the PLC and the possible step height set, the digital signal would thus be able to operate in steps of value of 0.1 V, 0.01 V or 0.001 V. Naturally, the smallest range is selected in this instance in order for the analogue signal to be reproduced as accurately as possible.

One simple example of an analogue signal is pressure, which is measured and displayed by a pressure gauge. The pressure signal may assume any intermediate value between its minimum and maximum values. Unlike the digital signal, it changes continually. In the case of the processing of analogue values via a PLC, as described, analogue voltage signals are evaluated and converted.

On the other hand, digital signals can be formed by adding together a certain number of binary signals. In this way, again as described in the above paragraph, it is also possible to generate a digital signal with 256 step values.

*Example*

| ***Bit No.*** | ***7*** | ***6*** | ***5*** | ***4*** | ***3*** | ***2*** | ***1*** | ***0*** | ***Digital value*** |
|---|---|---|---|---|---|---|---|---|---|
| Example 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 187 |
| Example 2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 51 |
| Example 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

This process is for instance used to implement timer and counter functions.

# Chapter 3

# Boolean operations

## *3.1 Basic logic functions*

As described in the previous chapter, any computer and equally any PLC operates using the number system to the base 2. This also applies to the octal ($2^3$) and the hexadecimal systems ($2^4$). The individual variable can therefore assume only two values, "0" or "1". Special algorithms have been introduced to be able to link these variables – the so-called boolean algebra. This can be clearly represented by means of electrical contacts.

**Negation (NOT function)**
The push button shown represents a normally closed contact. When this is unactuated, lamp H1 is illuminated, whereas in the actuated state, lamp H1 goes off.

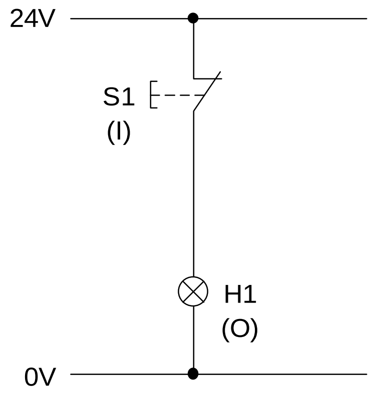


*Fig. B3.1: Circuit diagram*

Push button S1 acts as signal input, the lamp forms the output. The actual status can be recorded in a truth table:

| *I* | *O* |
|---|---|
| 0 | 1 |
| 1 | 0 |

*Truth table*

The boolean equation is therefore as follows:

$$\overline{I} = O \quad \text{(read: Not I equals O)}$$

The logic symbol is:

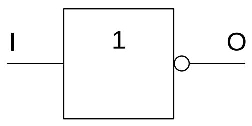


*Fig. B3.2:*
*NOT function*

If 2 negations are switched in succession, then these cancel one another.

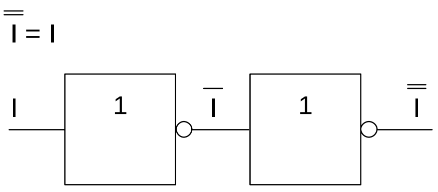


*Fig. B3.3:*
*2 logic NOT functions*

**Conjunction (AND-function)**
If two normally open contacts are switched in series, the actuated lamp is illuminated only if both push buttons are actuated.

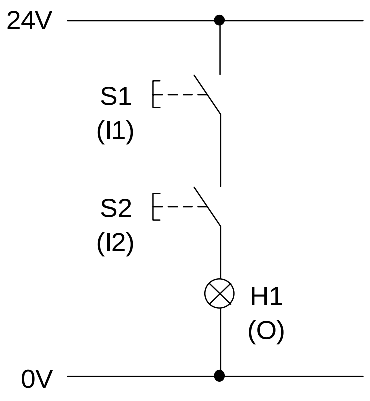


*Fig. B3.4:*
*Circuit diagram*

Truth table

| *I1* | *I2* | *O* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The truth table assigns the conjunction. The output assumes 1 only if both input 1 and input 2 produce a "1"-signal. This is referred to as an AND operation, which is represented as follows as an equation:

$$I1 \wedge I2 = O$$

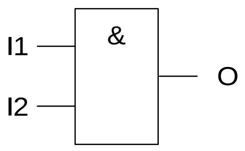


*Fig. B3.5: AND function*

In addition, the following algorithms apply for the conjunction:

$$a \wedge 0 = 0$$

$$a \wedge 1 = a$$

$$a \wedge \overline{a} = 0$$

$$a \wedge a = a$$

**Disjunction (OR-Function)**
Another basic logic function is OR. If the 2 normally open contacts are switched in parallel, then the lamp is illuminated whenever a least one push button is pressed.

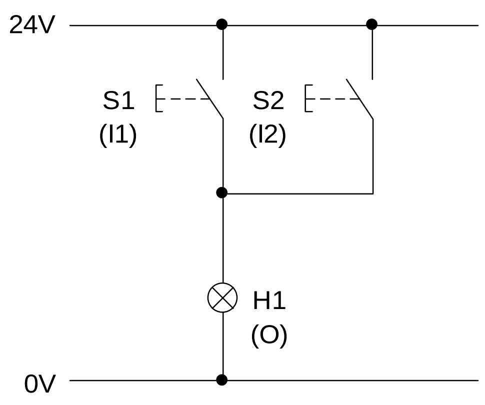


*Fig. B3.6:*
*Circuit diagram*

| *I1* | *I2* | *O* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

*Truth table*

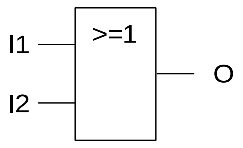


*Fig. B3.7:*
*OR function*

The logic operation is written in the form of the following equation:

$$I1 \vee I2 = O$$

The following algorithms also apply for the OR-operation:

$$b \vee 0 = b$$

$$b \vee 1 = 1$$

$$b \vee b = b$$

$$b \vee \overline{b} = 1$$

## 3.2 Further logic operations

The electrical realisation of a NOT-/AND-/OR-operation has already been described in section B3.1. Each of these operations can of course also be realised pneumatically or electronically. Boolean algebra also recognises the following logic operations. The following table provides an overview of these.

*Table B3.1: Logic connections*

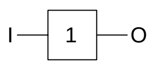
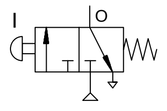
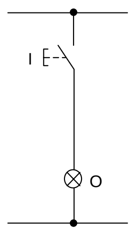
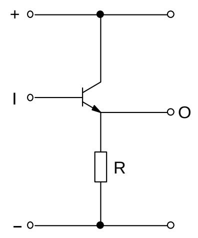
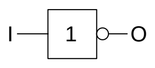
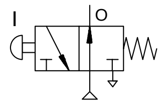
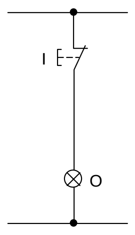
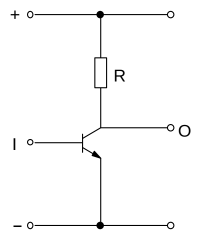
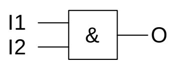
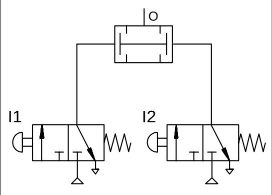
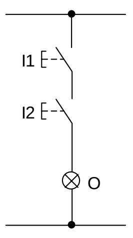
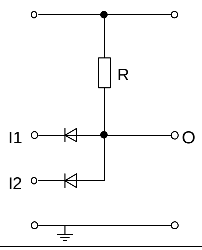
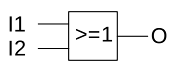
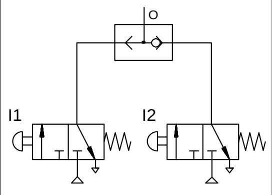
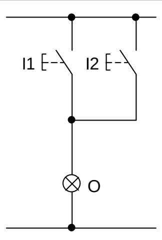
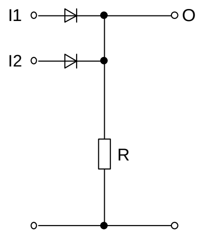
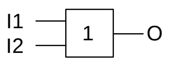
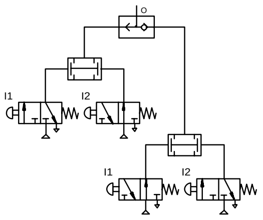
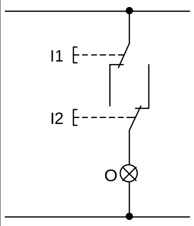
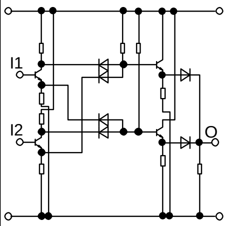
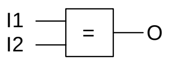
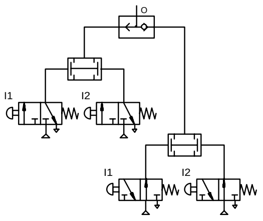
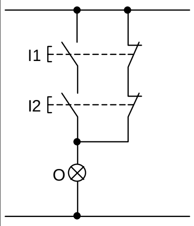
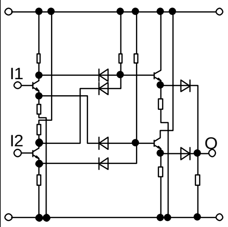
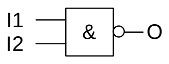
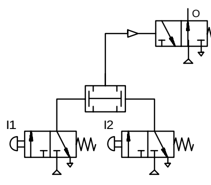
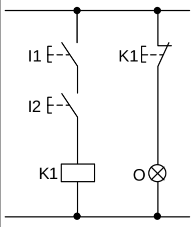
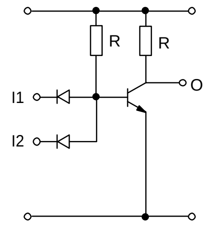
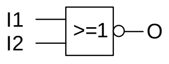
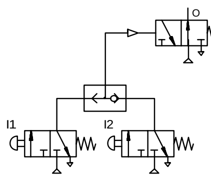
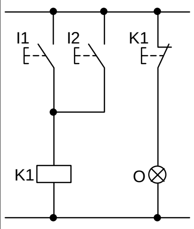
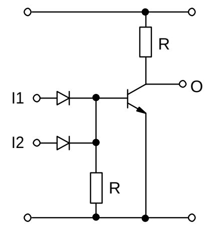
| Name | Equation | Truth table | log. symbols | pneumatic realisation | electr. realisation | electron. realisation |
|---|---|---|---|---|---|---|
| Identity | I = A | I \| O<br>0 \| 0<br>1 \| 1 |  |  |  |  |
| Negation | $\overline{I} = O$ | I \| O<br>0 \| 1<br>1 \| 0 |  |  |  |  |
| Conjunction | $I1 \wedge I2 = O$ | I1 \| I2 \| O<br>0 \| 0 \| 0<br>0 \| 1 \| 0<br>1 \| 0 \| 0<br>1 \| 1 \| 1 |  |  |  |  |
| Disjunction | $I1 \vee I2 = O$ | I1 \| I2 \| O<br>0 \| 0 \| 0<br>0 \| 1 \| 1<br>1 \| 0 \| 1<br>1 \| 1 \| 1 |  |  |  |  |

*Table B3.1:*
*Logic connections*
*(continued)*

| Name | Equation | Truth table | log. symbol | pneumatic realisation | electr. realisation | electron. realisation |
|---|---|---|---|---|---|---|
| Antivalence (exclusive OR) | $I1 \wedge \overline{I2} \vee \overline{I1} \wedge I2 = O$ | I1 \| I2 \| O<br>0 \| 0 \| 0<br>0 \| 1 \| 1<br>1 \| 0 \| 1<br>1 \| 1 \| 0 |  |  |  |  |
| Equivalence | $I1 \wedge I2 \vee \overline{I1} \wedge \overline{I2} = O$ | I1 \| I2 \| O<br>0 \| 0 \| 1<br>0 \| 1 \| 0<br>1 \| 0 \| 0<br>1 \| 1 \| 1 |  |  |  |  |
| NAND | $\overline{I1 \wedge I2} = O$ | I1 \| I2 \| O<br>0 \| 0 \| 1<br>0 \| 1 \| 1<br>1 \| 0 \| 1<br>1 \| 1 \| 0 |  |  |  |  |
| NOR | $\overline{I1 \vee I2} = O$ | I1 \| I2 \| O<br>0 \| 0 \| 1<br>0 \| 1 \| 0<br>1 \| 0 \| 0<br>1 \| 1 \| 0 |  |  |  |  |

## 3.3 Establishing switching functions

**Deriving boolean equations from the truth table**
Often, the logic operations shown in the previous section are not enough to adequately describe a status in control technology.

Very often, there is a combination of different logic operations. The logic connection in the form of a boolean equation can be easily established from the truth table.

The example below should clarify this:

**Sorting station task**
Various parts for built-in kitchens are to be machined in a production system (milling and drilling machine). The wall and door sections for certain types of kitchen are to be provided with different drill holes. Sensors B1 to B4 are intended for the detection of the holes.

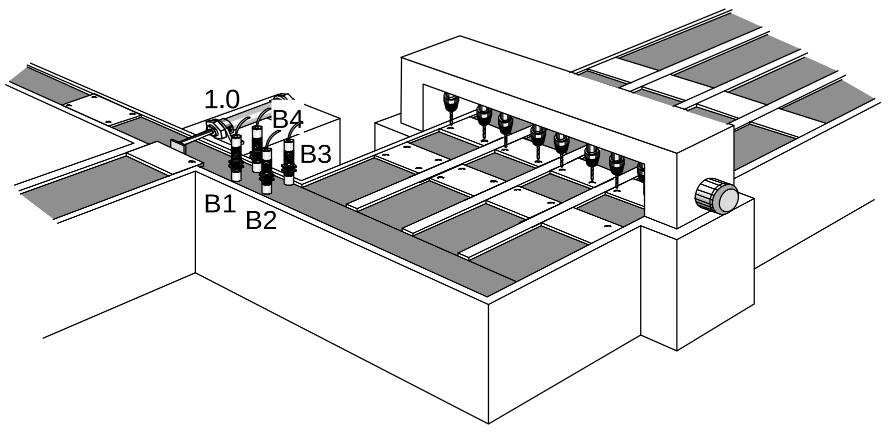


*Fig. B3.8: Sorting station*

Parts with the following hole patterns are for the ’Standard’ kitchen type. These parts are to be advanced via the double-acting cylinder 1.0.

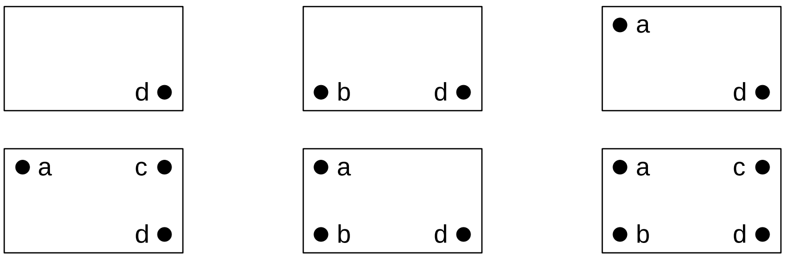


*Fig. 3.9:*
*Hole pattern of parts*

Assuming that a drilled hole is read as a 1-signal, the following truth table results.

| ***a*** | ***b*** | ***c*** | ***d*** | ***y*** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

*Truth table*

Two options are available in order to derive the logic equation from this table, which lead to two different expressions. The same result is obtained, of course, since the same circumstances are desribed.

**Standard form, disjunctive**
In the disjunctive standard form, all conjunctions (AND-operations) of input variables with the result 1, are carried out as a disjunctive operation (OR-operation). With signal status 0, the input variable is carried out as a negated operation and with signal status 1 as a non-negated operation.

In the case of the example given, the logic operation is therefore as follows:

$$\begin{aligned} y \quad = \quad & (\bar{a} \wedge \bar{b} \wedge \bar{c} \wedge d) \vee (\bar{a} \wedge b \wedge \bar{c} \wedge d) \vee (a \wedge \bar{b} \wedge \bar{c} \wedge d) \vee \\ & (a \wedge \bar{b} \wedge c \wedge d) \vee (a \wedge b \wedge \bar{c} \wedge d) \vee (a \wedge b \wedge c \wedge d) \end{aligned}$$

**Conjunctive standard form**
In the conjunctive standard form, all disjunctions (OR-operations) of the input variable producing the result 0, are carried out as a conjunctive operation (AND-operation). In contrast with the disjunctive standard form, in this instance, the input variable is negated with signal status "1" and a non-negated operation carried out with signal status "0".

$$\begin{aligned} y \quad = \quad & (a \vee b \vee c \vee d) \wedge (a \vee b \vee \bar{c} \vee d) \wedge (a \vee b \vee \bar{c} \vee \bar{d}) \wedge \\ & (a \vee \bar{b} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{c} \vee d) \wedge (a \vee \bar{b} \vee \bar{c} \vee \bar{d}) \wedge \\ & (\bar{a} \vee b \vee c \vee d) \wedge (\bar{a} \vee b \vee \bar{c} \vee d) \wedge \\ & (\bar{a} \vee \bar{b} \vee c \vee d) \wedge (\bar{a} \vee \bar{b} \vee \bar{c} \vee d) \end{aligned}$$

## 3.4 *Simplification of logic functions*

Both equations for the example given are rather extensive, with that of the conjunctive standard form being even longer still. This defines the criteria for using the disjunctive or conjunctive standard from: The decision is made in favour of the shorter form of the equation. In this case, the disjunctive standard form.

$$\begin{aligned} y \quad = \quad & (\bar{a} \wedge \bar{b} \wedge \bar{c} \wedge d) \vee (\bar{a} \wedge b \wedge \bar{c} \wedge d) \vee (a \wedge \bar{b} \wedge \bar{c} \wedge d) \vee \\ & (a \wedge \bar{b} \wedge c \wedge d) \vee (a \wedge b \wedge \bar{c} \wedge d) \vee (a \wedge b \wedge c \wedge d) \end{aligned}$$

This expression may be simplified with the help of a boolean algorithm.

The most important rules in boolean algebra are shown below:

| | |
|---|---|
| $a \vee 0 = a$ | $a \wedge 0 = 0$ |
| $a \vee 1 = 1$ | $a \wedge 1 = a$ |
| $a \vee a = a$ | $a \wedge a = a$ |
| $a \vee \overline{a} = 1$ | $a \wedge \overline{a} = 0$ |

**Commutative law**

$a \vee b = b \vee a$ $\qquad$ $a \wedge b = b \wedge a$

**Associative law**

$a \vee b \vee c = a \vee (b \vee c) = (a \vee b) \vee c$

$a \wedge b \wedge c = a \wedge (b \wedge c) = (a \wedge b) \wedge c$

**Distributive law**

$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

**De Morgan's rule**

$\overline{a \vee b} = \overline{a} \wedge \overline{b}$ $\qquad$ $\overline{a \wedge b} = \overline{a} \vee \overline{b}$

Applied to the above example, the following result is obtained:

$$\begin{aligned}
\mathbf{y} &= \overline{a}\,\overline{b}\,\overline{c}d \vee \overline{a}b\overline{c}d \vee a\overline{b}\,\overline{c}d \vee a\overline{b}cd \vee ab\overline{c}d \vee abcd \\
&= \overline{a}\,\overline{b}\,\overline{c}d \vee \overline{a}b\overline{c}d \vee a\overline{b}\,\overline{c}d \vee a\overline{b}cd \vee abd(\overline{c} \vee c) \\
&= \overline{a}\,\overline{c}d(\overline{b} \vee b) \vee a\overline{b}d(\overline{c} \vee c) \vee abd \\
&= \overline{a}\,\overline{c}d \vee a\overline{b}d \vee abd \\
&= \overline{a}\,\overline{c}d \vee ad(\overline{b} \vee b) \\
&= (\overline{a}\,\overline{c} \vee a)d \\
&= (\overline{c} \vee a)d \\
&= \boldsymbol{\overline{c}d \vee ad}
\end{aligned}$$

For reasons of clarity, the AND-operation symbol "$\wedge$" has been omitted in the individual expressions.

The basic principle of simplification is in the factoring out of variables and reducing to defined expressions. However, this method does require a sound knowledge of boolean algorithms plus a certain amount of practice. Another option for simplification will be introduced in the following section.

## 3.5 Karnaugh-Veitch diagram

In the case of the Karnaugh-Veitch diagram (KV diagram) the truth table turns into a value table.

| *a* | *b* | *c* | *d* | *y* | *No.* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 2 |
| 0 | 0 | 1 | 0 | 0 | 3 |
| 0 | 0 | 1 | 1 | 0 | 4 |
| 0 | 1 | 0 | 0 | 0 | 5 |
| 0 | 1 | 0 | 1 | 1 | 6 |
| 0 | 1 | 1 | 0 | 0 | 7 |
| 0 | 1 | 1 | 1 | 0 | 8 |
| 1 | 0 | 0 | 0 | 0 | 9 |
| 1 | 0 | 0 | 1 | 1 | 10 |
| 1 | 0 | 1 | 0 | 0 | 11 |
| 1 | 0 | 1 | 1 | 1 | 12 |
| 1 | 1 | 0 | 0 | 0 | 13 |
| 1 | 1 | 0 | 1 | 1 | 14 |
| 1 | 1 | 1 | 0 | 0 | 15 |
| 1 | 1 | 1 | 1 | 1 | 16 |

*Value table*

A total of 16 allocation options are available for the example, whereby the value table must also have 16 squares.

| | $\overline{cd}$ | $\overline{c}d$ | $c\overline{d}$ | $cd$ |
|---|---|---|---|---|
| $\overline{ab}$ | 1 | 2 | 3 | 4 |
| $\overline{a}b$ | 5 | 6 | 7 | 8 |
| $a\overline{b}$ | 9 | 10 | 11 | 12 |
| $ab$ | 13 | 14 | 15 | 16 |

*Fig. B3.10: Value table*

The results of the value table are transferred to the KV diagram according to the diagram shown below. In principle, representation is again possible in conjunctive or disjunctive standard form. The following, however, will be limited to the disjunctive standard form.

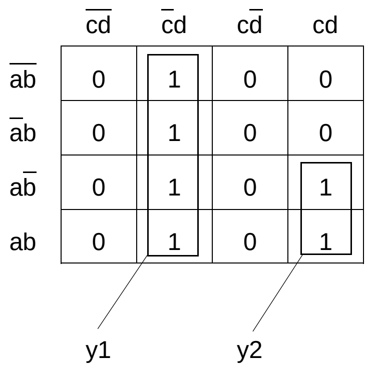
| | $\overline{c}\,\overline{d}$ | $\overline{c}d$ | $c\overline{d}$ | $cd$ |
|---|---|---|---|---|
| $\overline{a}\,\overline{b}$ | 0 | 1 | 0 | 0 |
| $\overline{a}b$ | 0 | 1 | 0 | 0 |
| $a\overline{b}$ | 0 | 1 | 0 | 1 |
| $ab$ | 0 | 1 | 0 | 1 |

*Fig. B3.11: Value table*

The next step consists of combining the statuses, for which "1" has been entered in the value table. This is done in blocks whilst observing the following rules:

- The combining statuses in the KV diagram must be in the form of a rectangle or square
- The number of combining statuses must be a result of function $2^x$.

This results in the following:



| | $\overline{c}\,\overline{d}$ | $\overline{c}d$ | $c\overline{d}$ | $cd$ |
|---|---|---|---|---|
| $\overline{a}\,\overline{b}$ | 0 | 1 | 0 | 0 |
| $\overline{a}b$ | 0 | 1 | 0 | 0 |
| $a\overline{b}$ | 0 | 1 | 0 | 1 |
| $ab$ | 0 | 1 | 0 | 1 |

*Fig. B3.12: Value table*

The variable values are selected for the established block and these in turn combined disjunctively.

$$
\begin{aligned}
y1 &= \bar{c}d \\
y2 &= acd \\
\mathbf{y} &= \bar{c}d \wedge acd \\
&= (\bar{c} \vee ac) \wedge d \\
&= (\bar{c} \vee a) \wedge d \\
&= \mathbf{\bar{c}d} \vee \mathbf{ad}
\end{aligned}
$$

Naturally, the KV diagram is not limited to 16 squares. 5 variables, for instance, would result in 32 squares ($2^5$), and 6 variables 64 fields ($2^6$).

# Chapter 4

# Design and mode of operation of a PLC

## 4.1 Structure of a PLC

With computer systems, differentiation is generally made between hardware, firmware and software. The same applies for a PLC, which is essentially based on a micro computer.

The **hardware** consists of the actual device technology, i.e. the printed circuit boards, integrated modules, wires, battery, housing, etc.

**firmware** is the software part, which is permanently installed and supplied by the PLC manufacturer. This includes fundamental system routines, used for starting the processor after the power has been switched on. Additionally, there is the operating system in the case of programmable logic controllers, which is generally stored in a ROM, a read-only memory, or in the EPROM.

Finally, there is the **software**, which is the user program written by the PLC user. User programs are usually installed in the RAM, a random access memory, where they can be easily modified.

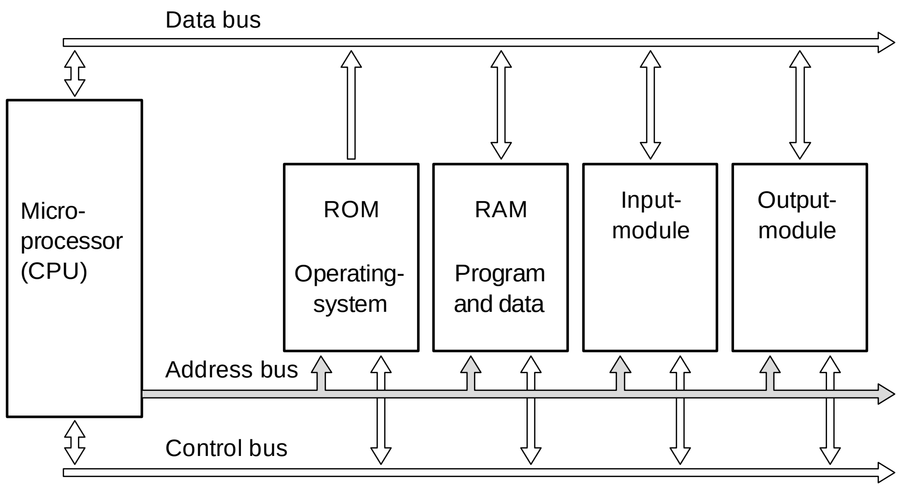


*Fig. B4.1: Fundamental design of a microcomputer*

Fig. B4.1 illustrates the fundamental design of a microcomputer. PLC hardware – as in the case of almost all of today's microcomputer systems – is based on a bus system. A bus system is a number of electrial lines divided into address, data and control lines. The address line is used to select the address of a connected bus station and the data line to transmit the required information. The control lines are necessary to activate the correct bus station either as a transmitter or sender.

The major bus stations connected to the bus system are the microprocessor and the memory. The memory can be divided into memory for the firmware and memory for the user program and data.

Depending on the structure of the PLC, the input and output modules are connected to a single common bus or – with the help of a bus interface – to an external I/O bus. Particularly in the case of larger modular PLC systems, an external I/O bus would be more usual.

Finally, a connection is required for a programming device or a PC, nowadays mostly in the form of a serial interface.

Fig. B4.2 illustrates the Festo FPC 101 as an example.

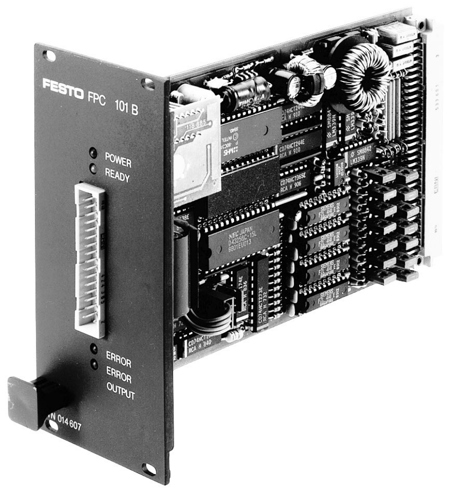


*Fig. B4.2:
Programmable logic controller Festo FPC 101*

## 4.2 Central control unit of a PLC

In essence, the central control unit of a PLC consists of a microcomputer. The operating system of the PLC manufacturer makes the universal computer into a PLC, optimised specifically for control technology tasks.

**Design of the central control unit**
Fig. B4.3 illustrates a simplified version of a microprocessor which represents the heart of a microcomputer.

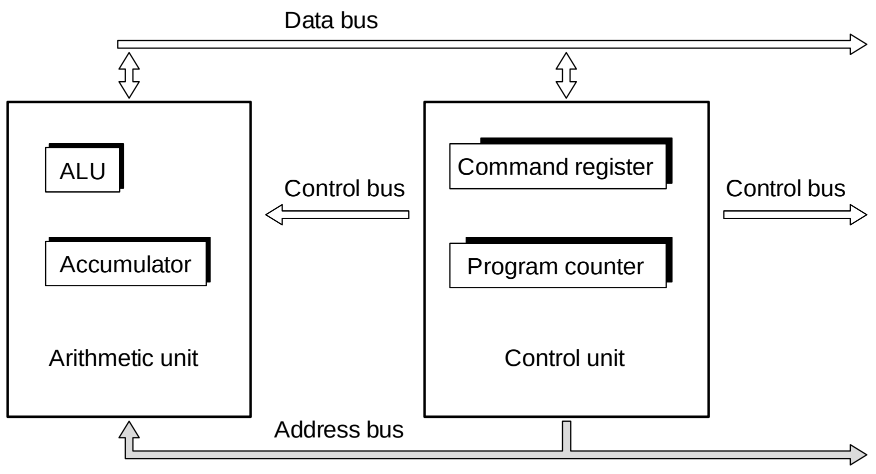


*Fig. B4.3: Design of a microprocessor*

A microprocessor consists in the main of an arithmetic unit, control unit and a small number of internal memory units, so-called registers.

The task of the **arithmetic unit** – the ALU (arithmetic logic unit) – is to execute arithmetic and logic operations with the data transmitted.

The **accumulator**, AC for short, is a special register assigned directly to the ALU. This stores both data to be processed as well as the result of an operation.

The **instruction register** stores a command called from the program memory until this is decoded and executed.

A **command** consists of an operation part and an address part. The operation part indicates which logic operation is to be carried out. The address part defines the operands (input signals, flags etc.), with which a logic operation is to be executed.

The **program counter** is a register, which contains the address of the next command to be processed. The following section will be dealing with this in greater detail.

The **control unit** regulates and controls the entire logic sequence of the operations required for the execution of a command.

**Instruction cycle within central control unit**
Today's conventional microcomputer systems operate according to the so-called "von-Neumann principle". According to this principle, the computer processes the program line by line. In simple terms, you could say that each program line of the PLC user program is processed in sequence.

This applies wholly irrespective of the programming language, in which the PLC program is written, be it in the form of a text program (statement list) or a graphic program (ladder diagram, sequential function chart). Since these various forms of representation always result in a series of program lines within the computer, they are subsequently processed one after the other.

In principle, a program line, i.e. generally a command, is processed in two steps:

- fetching the command from the program memory
- executing the command

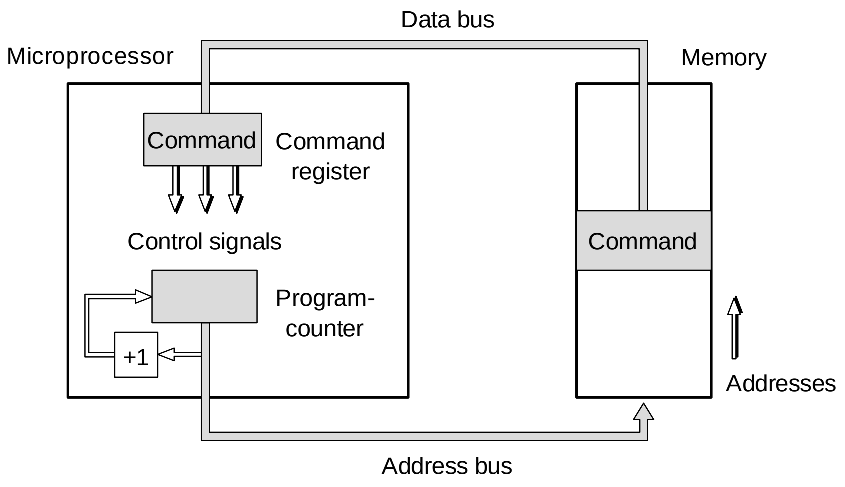


*Fig. B4.4:*
*Command sequence*

The contents of the program counter are transferred to the address bus. The control unit then causes the command at a specified address in the program memory, to be relayed to the data bus. From there, the command is read to the instruction register. Once the command has been decoded, the control unit generates a sequence of control signals for execution.

During the execution of a program, the commands are fetched in sequence. A mechanism which permits this sequence is therefore required. This task is performed by a simple incrementer, i.e. a step enabling facility in the program counter.

## 4.3 Function mode of a PLC

Programs for conventional data processing are processed once only from top to bottom and then terminated. In contrast with this, the program of a PLC is continually processed cyclically.

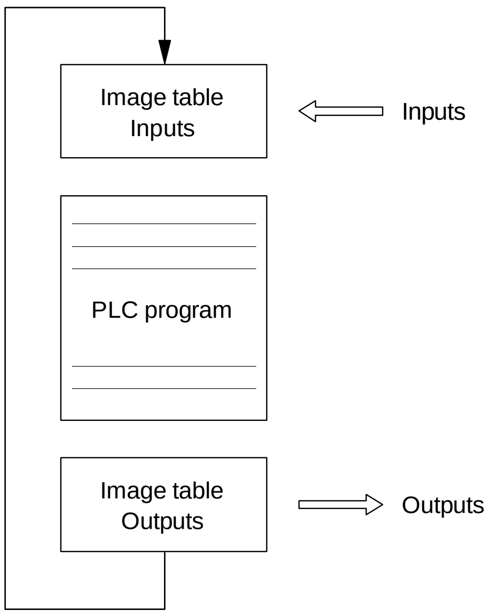


*Fig. B4.5: Cyclical processing of a PLC program*

The characteristics of cyclical processing are:

- As soon as the program has been executed once, it automatically jumps back to the beginning and processing is repeated.
- Prior to first program line being processed, i.e. at the beginning of the cycle, the status of the inputs is stored in the image table. The process image is a separate memory area accessed during a cycle. The status of an input thus remains constant during a cycle even if it has physically changed
- Similar to inputs, outputs are not immediately set or reset during a cycle, but the status stored intermediately in the process output image. Only at the end of a cycle are all the outputs physically swit-ched according to the logic status stored in the memory.

The processing of a program line via the central control unit of a PLC takes time which, depending on PLC and operation can vary between a few microseconds and a few milliseconds.

The time required by the PLC for a single execution of a program including the actualisation and output of the process image, is termed the cycle time. The longer the program is and the longer the respective PLC requires to process an individual program line, the longer the cycle. Realistic time periods for this are between approximately 1 and 100 milliseconds.

The consequences of cyclical processing of a PLC program using a process image are as follows:

- Input signals shorter than the cycle time may possibly not be recognised.
- In some cases, there may be a delay of two cycle times between the occurence of an input signal and the desired reaction of an output to this signal.
- Since the commands are processed sequentially, the specific behaviour sequence of a PLC program may be crucial.

With some applications it is essential for inputs or outputs to be accessed directly during a cycle. This type of program processing, bypassing of the process image, is therefore also supported by some PLC systems.

## 4.4 Application program memory

Programs specifically developed for particular applications require a program memory, from which these can be read cyclically by the central control unit. The requirements for such a program memory are relatively simple to formulate:

- It should be as simple as possible to modify or to newly create and store the program with the help of a programming device or a PC
- Safeguards should be in place to ensure that the program cannot be lost – either during power failure or through interference voltage
- The program memory should be cost effective
- The program memory should be sufficiently fast in order not to delay the operation of the central control unit.

Nowadays, three different types of memory are used in practice:

- RAM
- EPROM
- EEPROM

**RAM**

The RAM (random access memory) is a fast and highly cost effective memory. Since the main memory of computers (i.e. PLCs) consist of RAMs, they are produced in such high quantities that they are readily available at low cost without competition.

RAMs are read/write memories and can be easily programmed and modified.

The disadvantage of a RAM is that it is volatile, i.e. the program stored in the RAM is lost in the event of power failure. This is why RAMs are backed up by battery or accumulator. Since the service life and capacity of modern batteries are rated for several years, RAM back-up is relatively simple. Despite the fact that these are high performance batteries it is nevertheless essential to replace the batteries in good time.

**EPROM**
The EPROM (erasable programmable read-only memory) is also a fast and low cost memory which, in comparison with RAM, has the added advantage of being non-volatile, i.e. remanent. The memory contents therefore remain intact even in the event of power failure.

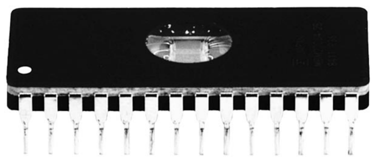


*Fig. B4.6:*
*Example of an EPROM*

For the purpose of a program modification, however, the entire memory must first be erased and, after a cooling period, completely reprogrammed. Erasing generally requires an erasing device, and a special programming unit is used for programming.

Despite this relatively complex process of erasing, – cooling – reprogramming EPROMs are very frequently used in PLCs, since these represent reliable and cost effective memories. In practice, a RAM is often used during the programming and commissioning phase of a machine. On completion of the commissioning, the program is then transferred to an EPROM.

**EEPROM**
The EEPROM (electrically erasable programmable ROM), EEROM (electrically erasable ROM) and EAROM (electrically alterable ROM) or also flash-EPROM have been available for some time. The EEPROM in particular, is used widely as an application memory in PLCs. The EEPROM is an electrically erasable memory, which can be subsequently written to.

## 4.5 Input module

The input module of a PLC is the module, which sensors are connected to. The sensor signals are to be passed on to the central control unit. The important functions of an input module (for the application) are as follows:

- Reliable signal detection
- Voltage adjustment of control voltage to logic voltage
- Protection of sensitive electronics from external voltages
- Screening of signals

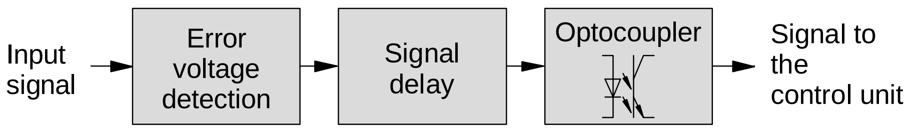


*Fig. B4.7: Block diagram of an input module*

The main component of today's input modules which meets these requirements is the optocoupler.

The optocoupler transmits the sensor information with the help of light, thereby creating an electrical isolation between the control and logic circuits, thereby **protecting** the sensitive electronics from spurious external voltages. Nowadays advanced optocouplers guarantee protection for up to approximately 5 kV, which is adequate for industrial applications.

The **adjustment of control and logic voltage**, in the straightforward case of a 24 V control voltage, can be effected with the help of a breakdown diode/resistor circuit. In the case of 220 V AC, a rectifier is connected in series.

Depending on PLC manufacturer **reliable signal detection** is ensured either by means of an additional downstream threshold detector or a corresponding range of breakdown diodes and optocouplers. Precise data regarding the signals to be detected is specified in DIN 19 240 .

The **screening** of the signal emitted by the sensor is critical in industrial automation. In industry, electrical lines are generally loaded heavily due to inductive interference voltages, which leads to a multitude of interference impulses on every signal line. Signal lines can be screened either via shielding, discrete cable ducts etc, or alternatively the input module of the PLC assumes the screening via an input signal delay.

This therefore requires the input signal to be applied for a sufficiently long period, before it is even recognised as an input signal. Since, due to their inductive nature, interference impulses are primarily transient signals, a relatively short input signal delay of a few milliseconds is sufficient to filter out most of the interference impulses.

Input signal delay is effected mainly via the hardware, i.e. via connection of the input to an RC module. In isolated cases, however, it is also possible to produce an adjustable signal delay via the software.

The duration of an input signal delay is approximately 1 to 20 milliseconds – depending on manufacturer and type. Most manufacturers offer especially fast inputs for tasks, where the input signal delay is then too long to recognise the required signal.

Differentiation is made between positive and negative switching connections when connecting sensors to PLC inputs. In other words, differentiation is made between inputs representing a current sink or a current source. In Germany for instance, in compliance with VDI 2880, positive switching connections are mainly used, since this permits the use of protective grounding. Positive switching means that the PLC input represents a current sink. The sensor supplies the operating voltage or control voltage to the input in the form of a 1-signal.

If protective grounding is employed, the output voltage of the sensor is short-circuited towards 0 volts or the fuse switched off in the event of a short-circuit in the signal line. This means that a logic 0 is applied at the input of the PLC.

In a number of countries, the use of negative switching sensors is commonplace, i.e. the PLC inputs operate as a power source. In these cases, a different protective measure must be used to prevent a 1-signal from being applied to the input of the PLC in the event of a short-circuit on the signal line. Possible methods are the earthing of the positive control voltage or insulation monitoring, i.e. protective grounding as a protective measure.

## 4.6 Output module

Output modules conduct the signals of the central control unit to final control elements, which are actuated according to the task. In the main, the function of an output – as seen from the application of the PLC – therefore includes the following:

- Voltage adjustment of logic voltage to control voltage
- Protection of sensitive electronics from spurious voltages from the controller
- Power amplification sufficient for the actuation of major final control elements
- Short-circuit and overload protection of output modules

In the case of output modules, two fundamentally different methods are available to achieve the above: Either the use of a relay or power electronics.

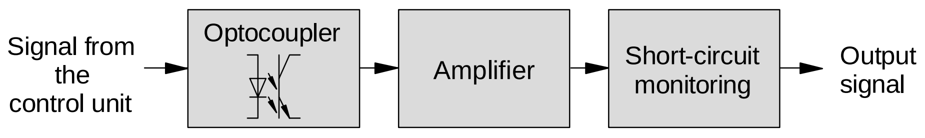


*Fig. B4.8: Block diagram of an output module*

The optocoupler once again forms the basis for power electronics and ensures the protection of the electronics and possibly also the voltage adjustment.

A protective circuit consisting of diodes must protect the integral power transistor from voltage surges.

Nowadays **short-circuit protection**, **overload protection** and **power amplification** are often ensured with fully integral modules. Standard short-circuit protection measures the current flow via a power resistor so as to switch off in the event of short-circuit; a temperature sensors provides overload protection; a Darlington stage or alternative power transistor stages provide the necessary power.

The permissible power of an output module is usually specified in a way which permits differentiation to be made between the permissible power of an output and the permissible cumulative power of an output module. The cumulative power of a module is almost always considerably lower than the total of individual permissible ratings, since power transistors transmit heat to one another.

If relays are used for the outputs, then the relay can assume practically all the functions of an output module: The relay contact and relay coil are electrically isolated from one another; the relay represents an excellent power amplifier and is particularly overload-proof, only short-circuit protection must be ensured via an additional fuse. In practice, however, optocouplers are nevertheless connected in series with relays, since this renders the actuation of relays easier and simpler relays can be used.

Relay outputs have the advantage that they can be used for different output voltages. By contrast, electronic outputs have considerably higher switching speeds and a longer service life than relays. In most cases, the power of the very small relays used in PLCs corresponds to that of the power stages of electronic outputs.

In Germany for example, outputs are also connected positive switching in accordance with VDI 2880, i.e. the output represents a power source and supplies the operating voltage to the consuming device.

In the case of a short circuit of the output signal line to earth, the output is short-circuited, if normal protective grounding measures are used. The electronics switch to short circuit protection or the fuse switches off, i.e. the consuming device cannot draw any current and is therefore unconnected and rendered safe. (In accordance with DIN 0113, the deenergised status must always be the safe status.)

If negative switching outputs are used, i.e. the output represents a current sink, the protective measure must be adapted in such a way, that the consuming device is rendered safe in the event of a short circuit on the signal line. Again, protective grounding with isolation monitoring or the neutralising of the positive control voltage are standard practice in this case.

## *4.7 Programming device / Personal computer*

Each PLC has a programming and diagnostic tool in support of the PLC application.

- Programming
- Testing
- Commissioning
- Fault finding
- Program documentation
- Program storage

These programming and diagnostic tools are either vendor specific programming devices or personal computers with corresponding software. Nowadays, the latter is almost exclusively the preferred variant, since the enormous capacity of modern PCs, combined with their comparatively low initial cost and high flexibility, represent crucial advantages.

Also available and being developed are so-called hand-held programmers for mini control systems and for maintenance purposes. With the increasing use of LapTop personal computers, i.e. portable, battery operated PCs, the importance of hand-held programmers is steadily decreasing.

**Essential software system functions forming part of the programming and diagnostic tool**

Any programming software conforming to IEC 1131-1 should provide the user with a series of functions. Hence the programming software comprises software modules for:

- Program input
  Creating and modifying programs in one of the programming languages via a PLC.
- Syntax test
  Checking the input program and the input data for syntax accuracy, thus minimizing the input of faulty programs.
- Translator
  Translating the input program into a program which can be read and processed by the PC, i.e. the generation of the machine code of the corresponding PC.
- Connection between PLC and PC
  This data circuit effects the loading of a program to the PLC and the execution of test functions.
- Test functions
  Supporting the user during writing and fault elimination and checking the user program via
  - a status check of inputs and outputs, timers, counters etc.
  - testing of program sequences by means of single-step operations, STOP commands etc.
  - simulation by means of manual setting of inputs/outputs, setting constants etc.
- Status display of control systems
  Output of information regarding machine, process and status of the PLC system
  - Status display of input and output signals
  - Display/recording of status changes in external signals and internal data
  - Monitoring of execution times
  - Real-time format of program execution

- Documentation
  Drawing up a description of the PLC system and the user program. This consists of
  - Description of the hardware configuration
  - Printout of the user program with corresponding data and identifiers for signals and comments
  - Cross-reference list for all processed data such as inputs, outputs, timers etc.
  - Description of modifications
- Archiving of user program
  Protection of the user program in non volatile memories such as EPROM etc.

# Chapter 5

# Programming of a PLC

## 5.1 Systematic solution finding

Control programs represent an important component of an automation system.

Control programs must be systematically designed, well structured and fully documented in order to be as

- error-free
- low-maintenance
- cost effective

as possible

**Phase model of PLC software generation**
The procedure for the development of a software program illustrated in fig. B5.1 has been tried and tested. The division into defined sections leads to targeted, systematic operation and provides clearly set out results, which can be checked against the task.

The phase model consisting of the following sections

- Specification: Description of the task
- Design: Description of the solution
- Realisation: Implementation of the solution
- Integration/commissioning: Incorporating into environment and testing the solution

can be applied to basically all technical projects. Differences occur in the methods and tools used in the individual phases.

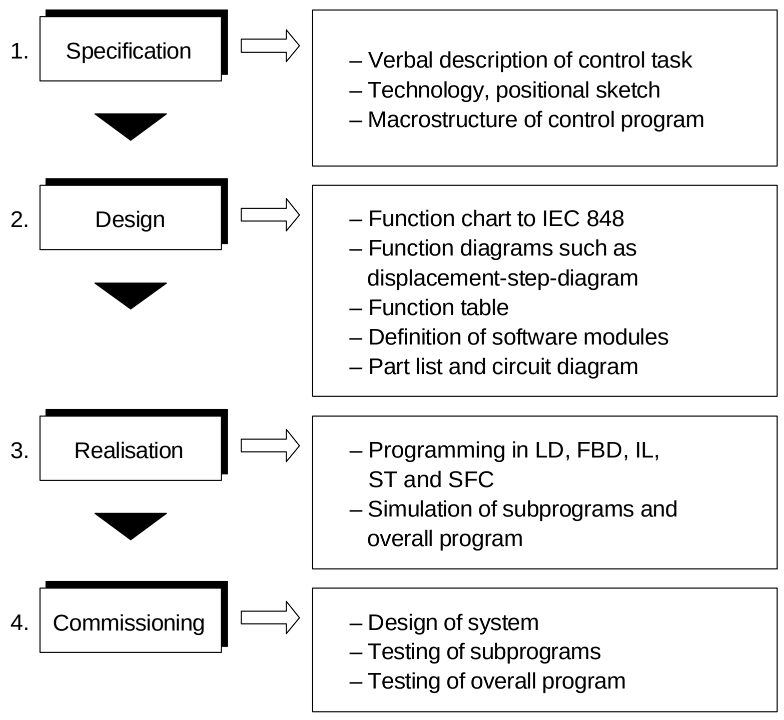

1. Specification
– Verbal description of control task
– Technology, positional sketch
– Macrostructure of control program
2. Design
– Function chart to IEC 848
– Function diagrams such as displacement-step-diagram
– Function table
– Definition of software modules
– Part list and circuit diagram
3. Realisation
– Programming in LD, FBD, IL, ST and SFC
– Simulation of subprograms and overall program
4. Commissioning
– Design of system
– Testing of subprograms
– Testing of overall program

*Fig. B5.1: Phase model for the generation of PLC software*

The phase model can be applied to control programs of varying complexity; for complex control tasks the use of such a model is absolutely essential.

The individual phases of the model are described below.

**Phase 1: Specification (Problem formulation)**
In this phase, a precise and detailed description of the control task is formulated. The specific description of the control system function, formalised as much as possible, reveals any conflicting requirements, misleading or incomplete specifications.

The following are available at the end of this phase:

- Verbal description of the control task
- Structure/layout
- Macrostructuring of the system or process and thus rough stucturing of the solution

**Phase 2: Design (Concrete form of solution concept)**
A solution concept is developed on the basis of the definitions established in phase 1. The method used to describe the solution must provide both a graphic and process oriented description of the function and behaviour of the control system and be independent of the technical realisation.

These requirements are fulfilled by the function chart (FCH) as defined in DIN 40 719, Part 6 or IEC 848. Starting with a representation of the overall view of the controller (rough structure of the solution), the solution can be refined step by step until a level of description is obtained, which contains all the details of the solution (refinement of rough structure).

In the case of complex control tasks, the solution is structured into individual software modules in parallel with this. These software modules implement the job steps of the control system. These can be special functions such as the realisation of an interface for visualisation or communications systems, or equally permanently recurring job steps.

The displacement-step diagram represents another standard form for the description of control systems apart from the function chart to DIN 40 719, Part 6.

**Phase 3: Realisation (Programming of solution concept)**
The translation of the solution concept into a control program is effected via the programming languages defined in IEC 1131-3. These are: sequential function chart, function block diagram, ladder diagram, statement list and structured text.

Control systems operating in a time/logic process and available in FCH to DIN 40 719, P.6, can be clearly and easily programmed in a sequential function chart. A sequential function chart, in as far as possible, uses the same components for programming as those used for the description in the function chart to DIN 40 719, T.6.

Ladder diagram, function block diagram and statement list are the programming languages suitable for the formulation of basic operations and for control systems which can be described by simple operations logic operations or boolean signals.

The high-level language structured text is mainly used to create software modules of mathematical content, such as modules for the description of control algorithms.

In so far as PLC programming systems support this, the control programs or parts of a program created should be simulated prior to commissioning. This permits the detection and elemination of errors right at the initial stage.

**Phase 4: Commissioning**
**(Construction and testing of the control task)**
This phase tests the interaction of the automation system and the connected plant. In the case of complex tasks, it is advisable to commission the system systematically, step by step. Faults, both in the system and in the control program, can be easily found and eliminated using this method.

**Documentation**
One important and crucial component of a system is documentation, which is an essential requirement for the maintenance and expansion of a system. Documentation, including the control programs, should be available both on paper and on a data storage medium. The documentation consists of the document of the individual phases, printouts of the control programs and of any possible additional descriptions concerning the control program. Individually these are:

- Problem description
- Positional sketch or technology pattern
- Circuit diagram
- Terminal diagram
- Printouts of control programs in SFC, FBD etc.
- Allocation list of inputs and outputs
  (this also forms part of the control program printouts)
- Additional documentation

## 5.2 IEC 1131-3 structuring resources

IEC 1131-3 is a standard for the programming of not just one individual PLC, but primarily also of complex automation systems. Control programs for extensive applications must be clearly structured in order to be intelligible, maintainable and possibly also portable, i.e. transferable to another PLC system.

Definitions are required not only for elementary language commands, but also for the language elements for structuring. Structuring resources (fig. B5.2) relate to the control programs and the configuration of the automation system.

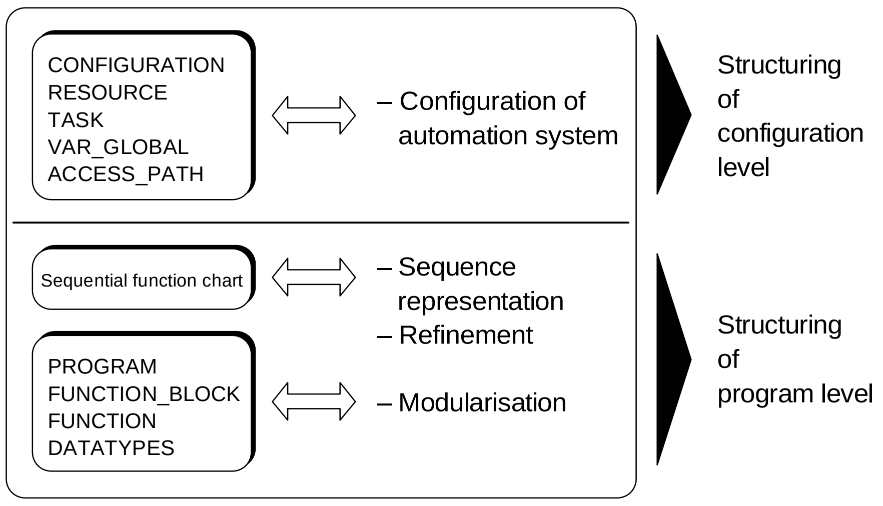

CONFIGURATION
RESOURCE
TASK
VAR_GLOBAL
ACCESS_PATH
– Configuration of automation system
Structuring of configuration level
Sequential function chart
– Sequence representation
– Refinement
PROGRAM
FUNCTION_BLOCK
FUNCTION
DATATYPES
– Modularisation
Structuring of program level

*Fig. B5.2: IEC 1131-3 structuring method*

**Structuring resources at program level**
The structuring resources – program, function block and function – contain the actual control logic (rules) of the control program. These are also known as program organisation units. These structuring resources are available for any programming language. They are used for the modularisation of control programs and the user program – this concerns primarily programs and function blocks – or also supplied by the manufacturer – as far as programs, function blocks are concerned.

IEC 1131-3 defines a comprehensive set of standardised functions and function block. These can be expanded by own user functions and function block for special or continually recurring tasks.

Software modules, which can be used in any way, are entered in libraries, where they are made available.

Programs represent the outer program organisation shell and can be differentiated from the function block mainly by the fact that they cannot be invoked by any other program organisation unit.

The sequential function chart represents another resource for structuring at program level. The contents of the actual programs and function block can again be clearly and intelligibly represented by means of a sequential function chart.

**Structuring resources at configuration level**
The language elements for configuration describe the incorporation of control programs in the automation system and their time-related control.

The automation system represents a configuration (CONFIGURATION language element). Within the configuration, there are global variables (VAR_GLOBAL language element).

A resource (RESOURCE language element) corresponds to the processor of a multiprocessor system, to which one or several programs are assigned. In addition, it comprises control elements, which include the time-related control of programs. This control element is a task (TASK language element). The control element Task defines whether a program is to be processed cyclically or once only, triggered by a specific event. Programs not specifically linked to a task are processed cyclically in the background and with the lowest priority.

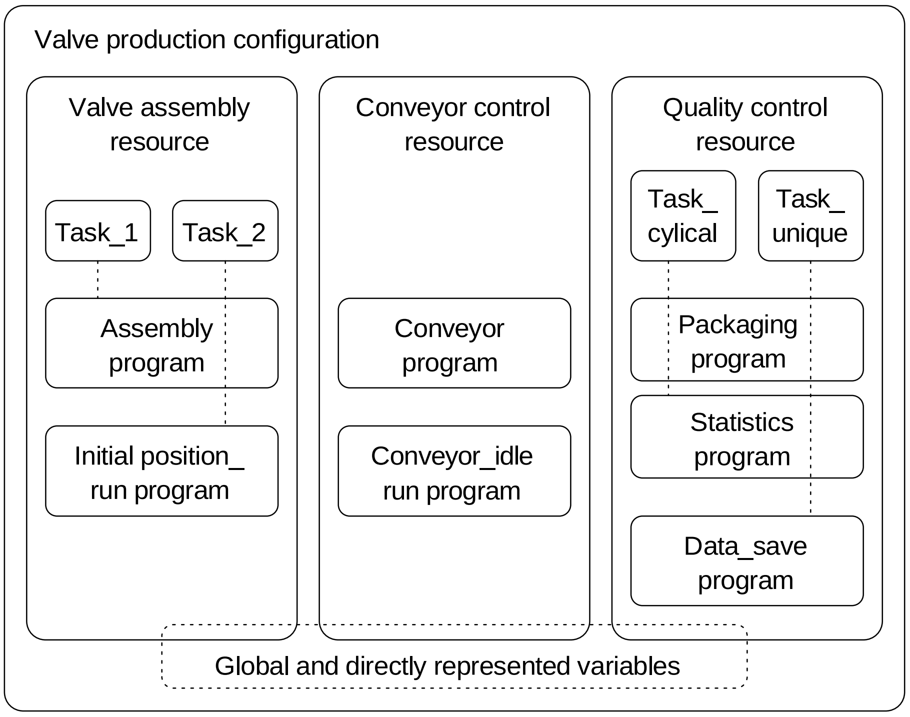


*Fig. B5.3:
Graphic example
of a configuration*

The structuring resources for configuration are shown in a combined overview in fig. B5.3. An example applying these to an automation task is given by way of an explanation.

The task in hand is to design and automate a production line for the assembly of pneumatic valves.

A PLC multiprocessor with three processor cards has been designated for the valve assembly. The processor cards are assigned to the valve assembly, the conveyor control and quality control.

The programs Statistics and Data_saving are associated with different tasks. As such they possess different execution characteristics. The program Statistics evaluates and compresses the quality data at regular intervals. The priority of this program is low. It is started regularly, e.g. every 20 minutes, by the task Task_cyclical. In the event of an EMERGENCY-STOP, the program Data_saving is to transmit all available data to a higher-order cell computer in order to prevent any potential data loss. The program is started event-driven of the highest priority via the EMERGENCY-STOP signal.

IEC 1131-3 provides defined and thus standardised interfaces for the exchange of data within a configuration. If specific information such as a read variable, is required in different program organisation units, this variable is designated as a global variable. Data can then be exchanged via a variable designated as such. Global variables can only be accessed in programs and function blocks.

What is of interest for networked systems is communication beyond a configuration. Special standard communication function blocks are available to the user for this. These are defined in IEC 1131-5 and are used in IEC 1131-3. Another possibility is the definition of access paths (language resource ACCESS_PATH) to specific variables. These can then also be read or written from other positions.

## *5.3 Programming languages*

IEC 1131-3 defines five programming languages. Although the functionality and structure of these languages is very different, these are treated as one language family by IEC 1131-3 with overlapping structure elements (variable declaration, organisation parts such as function and function block, etc.) and configuration elements.

The languages can be mixed in any way within a PLC project. The unification and standardisation of these five languages represent a compromise of historical, regional and branch-specific requirements. Provision has been made for future expansion, (such as the function block principle or the language Structured Text) plus necessary information technology details (data type etc.) have been incorporated.

The language elements are explained with the help of a machining process involved in valve production. Two sensors are used to establish whether a workpiece with correctly drilled holes is available at the machining position. If the valve to be machined is of type A or type B – this is set via two selector switches – the cylinder advances and presses the sleeve into the drilled hole.

**Ladder diagram (LD)**
Ladder diagram is a graphic programming language derived from the circuit diagram of directly wired relay controls. The ladder diagram contains contact rails to the left and the right of the diagram; these contact rails are connected to switching elements (normally open/normally closed contacts) via current paths and coil elements.

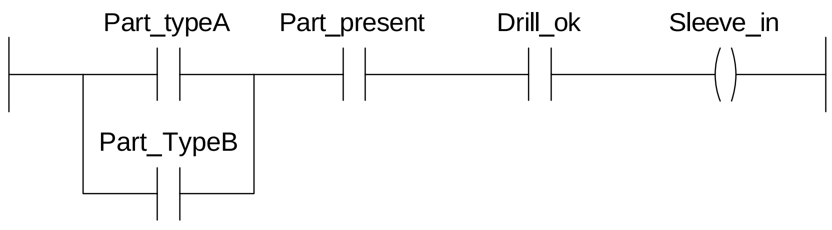


*Fig. B5.4:*
*Example of ladder diagram language*

**Function block diagram (FBD)**
In the function block diagram, the functions and function blocks are represented graphically and interconnected into networks. The function block diagram originates from the logic diagram for the design of electronic circuits.

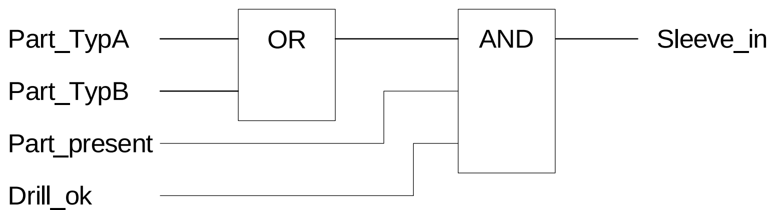


*Fig. B5.5:*
*Example of function block diagram language*

## Instruction list (IL)

Statement list is a textual assembler-type language characterised by a simple machine model (processor with only one register). Instruction list is formulated from control instructions consisting of an operator and an operand.

```
LD      Part_TypeA
OR      Part_TypeB
AND     Part_present
AND     Drill_ok
ST      Sleeve_in
```

*Fig. B5.6: Example of instruction list language*

With regard to language philosophy, the ladder diagram, the function block diagram and instruction list have been defined in the way they are used in today's PLC technology. They are however limited to basic functions as far as their elements are concerned. This separates them essentially from the company dialects used today. The competitiveness of these languages is maintained due to the use of functions and function blocks.

## Structured text (ST)

Structured text is high-level language based on Pascal, which consists of expressions and instructions. Instructions can be defined in the main as: Selection instructions such as IF...THEN...ELSE etc., repetition instructions such as FOR, WHILE etc. and function block invocations.

```
Sleeve_in := (Part_TypeA OR Part_TypeB) AND Part_present AND Drill_ok;
```

*Fig. B5.7: Example of structured text language*

Structured text enables the formulation of numerous applications, beyond pure function technology, such as algorithmic problems (high-order control algorithms etc.) and data handling (data analysis, processing of complex data structures etc.).

**Sequential function chart (SFC)**
The sequential function chart is a language resource for the structuring of sequence-oriented control programs.

The elements of the sequential function chart are steps, transitions, alternative and parallel branching.

Each step represents a processing status of a control program, which is active or inactive. A step consists of actions which, identical to the transitions, are formulated in the IEC 1131-3 languages. Actions themselves can again contain sequence structures. This  feature permits the hierarchical structure of a control program. The sequential function chart is therefore an excellent tool for the design and structuring of control programs.

# Chapter 6

# Common elements of programming languages

## 6.1 Resources of a PLC

According to IEC 1131-3, only inputs and outputs and the controller memory can be addressed directly by a control program. Direct addressing in this instance means that in the program an input, output or memory element of the controller is affected immediately and not indirectly via a defined symbolic variable. Naturally, IEC 1131-3 recognises numerous other resources, e.g. timers and counters. However, these are integrated into functions and function blocks in order to ensure the highest possible degree of control program portability between different control systems.

**Inputs, outputs and the memory**
The most important controller constituents include the inputs, outputs and the memory. Only via its inputs can a controller receive information from the connected processes. Similarly it can only influence these via its outputs or store information for subsequent continued processing.

The designations for the resources inputs, outputs and memory elements are defined by IEC 1131-3 and mandatory.

*Fig. B6.1: Designations for inputs, outputs and memory*

| | |
|---|---|
| Inputs | I |
| Outputs | Q |
| Memory | M |

Without further reference, these designate only binary inputs or outputs and single bit memory elements, designated as a flag.

The standard generally speaks of directly represented variables. These are variables, which are referred directly to the hardware-related available inputs, outputs and memory elements of the controller. The allocation of inputs, outputs and flags and their physical or logical position in the control system is defined by the respective controller manufacturer.

Insofar as the controller supports this, resources can be addressed, which are defined in excess of one bit. IEC 1131-3 employs a further letter to describe this, which follows the abbreviation I, Q or M and, for instance, designates bytes and words.

IEC 1131-3 designates the data types shown in fig. B6.2 in conjunction with inputs, outputs and flags.

| | |
|---|---|
| BOOL | Bit sequence of length 1 |
| BYTE | Bit sequence of length 8 |
| WORD | Bit sequence of length 16 |

*Fig. B6.2:
Data types*

1-Bit sizes, such as defined by the data type BOOL (boolean), may only assume the values 0 or 1. Consequently, the range of values for BOOL type data consists of the two values 0 and 1.

In contrast with this, one should observe that in the case of bit sequence data types consisting of more than one bit, there is no immediate connected range of values. All bit sequence data types such as for instance BYTE and WORD are merely a combination of several bits. Each of these bits has the value 0 or 1, but their combination does not have its own value.

The mandatory designation methods for inputs, outputs and flags of different bit length are represented in fig. B6.3.

| | | |
|---|---|---|
| I, Q, M<br>or<br>IX, QX, MX | Input bit, output bit, memory bit | 1 bit |
| IB, QB, MB | Input byte, output byte, memory byte | 8 bit |
| IW, QW, MW | Input word, output word, memory word | 16 bit |

*Fig. B6.3:
Designations for inputs, outputs and memory*

An individual bit of an input, output or flag may also be addressed without the additional abbreviation X for the data type.

Since a controller always has a relatively large number of inputs, outputs and flags available, these must be specially identified for the purpose of differentiation. Numbering is used in IEC 1131-3 to this end, such as in the following example:

| | |
|---|---|
| I1 | Input 1 |
| IX9 | Input 9 |
| I15 | Input 15 |
| QW3 | Output word 3 |
| MB5 | Memory byte 5 |
| MX2 | Memory 2 |

IEC 1131-3 does not specify the number range, which is permissible for this numbering and whether it should start with 0 or 1. This is specified by the controller manufacturer.

A hierarcical number of inputs, outputs and flags may also be used, if the controller in use has been suitably configured.

A point is used to separate the indiviudal levels of the hierarchy. The number of hierarchy levels has not been defined.

In the case of hierarchical numbering, the highest position in the number on the left must be coded, the numbers further to the right represent consecutive lower positions.

*Example*

- I3.8.5

The specified input I3.8.5 can therefore be be made up as follows:

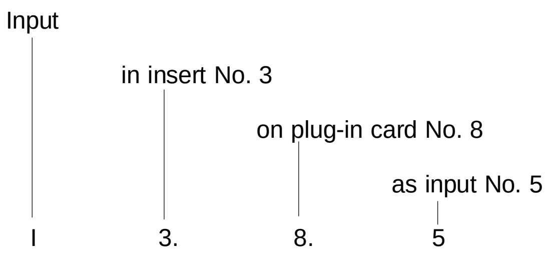


*Fig. B6.4:*
*Structure of*
*hierarchical designations*

IEC 1131-4 does not make any comment regarding the assignment of individual bits in a BYTE or WORD. Controller manufacturers frequently choose hierarchical designation methods to assign individual bits as parts of words. As such, F6.2 could for instance represent the bit number 2 of flag word number 6. However, this does not necessarily have to be so, since flag bit F6.2 and flag word FW6 need not be in any way connected. Moreover, no definition has been made as to whether the numbering of individual bits in one word is to start on the left or the right (bit number 0 on the far right has been the most frequently used so far).

**Directly addressed variables**
If resources in a control program are to be addressed directly, the resource designation must be prefixed with the sign %.

Examples of directly addressable variables:

| | |
|---|---|
| %IX12<br>or<br>%I12 | Input bit 12 |
| %IW5 | Input word 5 |
| %QB8 | Output byte 8 |
| %MW27 | Memory word 27 |

The use of directly addressed variables is permissible solely in programs, configurations and resources.

The program organisation units Function and Function block must operate exclusively with symbolic variables in order to keep these as controller-independent as possible and as such more widely usable.

## *6.2 Variables and data types*

The use of exclusively directly represented variables (resources, inputs, outputs and memory) is not enough to create control programs. Frequently, data is required, which contains specific information, also of a more complex nature. This data can be specified direct, e.g. time data or counter values or accessible via variables only – i.e. via a symbolic designation. The most important definitions for dealing with data or variables is shown below.

**Symbolic addressing**
A symbolic identifier always consists of capital or lower case letters, digits and an underline. An identifier must always begin with a letter or an underline. The underline can also be used to render an identifier more readable. It is however a significant character. The two identifiers Motor_on and Motoron are therefore different. Several underlines are impermissible. If the controller supports capital and lower case letters, then the use of these letters must not be of any significance. The two identifiers MOTORON and Motoron are interpreted identically and designate the same object.

The following identifiers are impermissible:

| | |
|---|---|
| 123 | Name does not start with a letter |
| Button_? | The last character is invalid, since it is neither a letter or a number |

Furthermore, symbolic identifiers must not be identical with key words. As a rule, key words are words reserved for specific tasks.

**Representation of data**
Within a control program, it must be possible for time values, counter values etc. to be specified.

Accordingly, IEC 1131-3 has laid down the definitions for the representation of data to specify

- Counter values
- Time values
- Strings

| ***Description*** | ***Examples*** | |
|---|---|---|
| Integers | 12, -8, 123_456*, +75 | |
| Floating point numbers | -12.0, -8.0, 0.123_4* | |
| Numbers to base 2<br>(Binary numbers) | 2#1111_1111<br>2#1101_0011 | (255 decimal)<br>(211 decimal) |
| Numbers to base 8<br>(Octal numbers) | 8#377<br>8#323 | (255 decimal)<br>(211 decimal) |
| Numbers to base 16<br>(Hexadecimal numbers) | 16#FF or 16#ff<br>16#D3 or 16#d3 | (255 decimal)<br>(211 decimal) |
| Boolean zero and one | 0, 1 | |

* The use of individual underlines between the digits is permissible to improve readability. However, the underline is not significant.

*Table B6.1: Representation of numerical data*

IEC 1131-3 provides for different types of time data:

- Duration, e. g. for measuring results
- Date
- Time of day, e. g. for synchronisation from start or end of an event (also in conjunction with date)

| ***Description*** | ***Examples*** |
|---|---|
| Time duration | T#18ms, t#3m4s, t#3.5s<br>t#6h_20m_8s<br>TIME#18ms |
| Date | D#1994-07-21<br>DATE#1994-07-21 |
| Time of day | TOD#13:18:42.55<br>TIME_OF_DAY#13:18:42.55 |
| Date and time | DT#1994-07-21-13:18:42.55<br>DATE_AND_TIME#1994-07-21-13:18:42.55 |

*Table B6.2: Representation of time data*

The specification of a time duration consists of an introductory part, the key word T# or t#, and a sequence of time-related sections – i.e. days, hours, minutes, seconds and milliseconds.

Abbreviations for time data:

| | |
|---|---|
| d | Days |
| h | Hours |
| m | Minutes |
| s | Seconds |
| ms | Milliseconds |

Capitals may also be used instead of lower case letters and individual underlines inserted for the purpose of better readability.

A fixed format has also been specified by IEC 1131-3 for the specification of a date, time of day or a combination of both. Each specification starts with a key word; the actual information is represented as shown in table B6.2.

Another important method of representation of data is the use of a sequence of characters also known as strings, which may be required for the exchange of information, e.g. between different controllers, with other components of an automation system or also for the programming of texts for display on control and display units.

A string consists of zero or several characters, introduced and ended by a simple inverted comma.

*Table B6.3: Representation of strings*

| ***Example*** | ***Description*** |
|---|---|
| ’B’ | String of length 1, containing the character B |
| ’Warning’ | String of length 7, containing the string Warning |
| ’’ | void string |

## Data types

IEC 1131-3 defines a large number of data types for different tasks. One such data type, BOOL, has already been mentioned. A BOOL type variable either assumes the value 0 or 1.

| *Keyword* | *Data type* | *Range of values* |
|---|---|---|
| BOOL | Boolean number | 0, 1 |
| SINT | Short integer | 0 to 255 |
| INT | Integer | -32 768 to +32 767 |
| DINT | Double integer | -2 147 483 648 to +2 147 483 647 |
| UINT | Unsigned integer | 0 to 65 535 |
| REAL | Floating point number | +/-2.9E-39 to +/-3.4E+38 |
| TIME | Time duration | implementation-dependent |
| STRING | Variable-long string | implementation-dependent |
| BYTE | Bit sequence 8 | no range of values declarable |
| WORD | Bit sequence 16 | no range of values declarable |

*Table B6.4:
A number of elementary data types*

Two other important data types, named INT and UINT define integer numbers. Variables of data type INT (integer) permit numeric values of -32 768 to +32 767. The range of values of data type INT therefore covers both negative and positive numbers. Type UINT variables (unsigned integer) permit positive values only. The range of values for UINT extends from 0 to 65 535. SINT (short integer) and DINT (double integer) are additional data types defining integer numbers. However, these have an even smaller or greater range of values than data type INT. The data type REAL contains floating point numbers. These are numbers, which can contain places after the point, such as for instance 3.24 or -1.5. Data type TIME is used to specify time, and may contain a time duration such as for instance 2 minutes and 30 seconds.

Apart from these elementary predefined data types, the user has the possibility of defining own data types. This is useful in cases where the problem definition goes beyond the realms of pure control technology.

Derived data types are declared within a TYPE...END_TYPE construct. The complete declaration is listed below for enumeration type Colour in table B6.5.:

```
TYPE
        Colour: (RED, BLUE, YELLOW, BLACK);
END_TYPE
```

Table B6.5:
Derived data types

| ***Derived data type*** | ***Declaration***<br>***TYPE … END_TYPE*** |
|---|---|
| Enumeration type | Colour: (RED, BLUE, YELLOW, BLACK); |
| Subrange type | Reference_range: INT(80..110); |
| Fields (array) | Position: ARRAY[1..10] OF REAL; |
| Structures | Coordinates:<br>STRUCT<br>X:REAL;<br>Y:REAL;<br>END_STRUCT; |

The significance of the individual data types in table B6.5 is explained briefly below:

- A data element of the type Colour may only assume one of the values RED, BLUE, YELLOW or BLACK.
- A data element of the data type Reference_range may only assume values between 80 and 110, including the lower and upper limit 80 or 110.
- A Position type data element represents a list with 10 entries. Each entry has the value of a REAL number. Individual entries can be indexed via the index.
- A Coordinates type data element contains two REAL numbers, which can be accessed via their names X and Y.

Not every controller needs to recognise all these data types. Each controller manufacturer puts together a set of data types, which may be used in the controller.

**Variable declaration**
With the use of data, the right of access to this data must be clearly defined. To this end, IEC 1131-3 uses a variable declaration.

In order to understand the function of a variable declaration, it is first of all necessary to establish that the controller program is constructed into individual organisation units.

These units are:

- Configuration
- Resource
- Programs
- Function blocks
- Functions

All variables have a specific position. In the case of programming languages in text form (IL and ST), variable declarations are roughly the same as those used in the programming language Pascal. For graphic forms of representation, a tabular form with equivalent contents would be feasible. These are however not specified in IEC 1131-3.

All variable declarations (fig. B6.5) always start with a keyword, which designates the position of the variable in the organisation unit of the controller, and end with the keyword END_VAR.

```
VAR
    Temp       : INT;           (*Temperature                     *)
    Manual     : BOOL;          (*Flag for manual operation       *)
    Full, Open: BOOL;           (*Flag for "full" and "open"      *)
END_VAR
```

*Fig. B6.5:
Variable declaration*

The variables and their assignment to a data type are entered between these keywords in that the symbolic identifier or identifiers of the variables are specified, the data type named after a colon and the declaration closed with a semicolon. If several variables are declared, they are repeated correspondingly. Normally, each declaration is written in a separate line in this case.

IEC 1131-3 differentiates between six different types of access to variables. Each type has a keyword, which introduces the variable declaration.

| | |
|---|---|
| Input variables | VAR_INPUT |
| Output variables | VAR_OUTPUT |
| Input/output variables | VAR_IN_OUT |
| Local variables | VAR |
| Global variables | VAR_GLOBAL |
| External variables | VAR_EXTERN |

*Table B6.6: Keywords for the declaration of variables*

Input variables are declared with the keywords VAR_INPUT and END_VAR.

```
VAR_INPUT
    Input   : INT;      (*Input value                                    *)
END_VAR
```

*Fig. B6.6: Declaration of an input variable*

Variables specified in this way represent input variables fed externally to an organisation unit, e.g. a function block. These can be read only within the organisation unit.

Modifications are not possible.

Analogous to this, output variables are defined with the keywords VAR_OUTPUT and END_VAR.

```
VAR_OUTPUT
    Result : INT;      (*Feedback value                                 *)
END_VAR
```

*Fig. B6.7: Declaration of an output variable*

The data, which computes an organisation unit and feeds this back externally is declared as above.

All organisation unit results are to be transferred beyond the organisation units via variables declared in this way. Within the organisation units, these can be read and written. Externally, read access only is permitted.

In cases where variables containing input and output values are permitted, these must be created with the keywords VAR_IN_OUT and END_VAR.

```
VAR_IN_OUT
    Value      : INT;
END_VAR
```

*Fig. B6.8: Declaration of an input/output variable*

This form represents a third option and permits the declaration of variables, which may be read and used within the organisation unit.

In the case of a variable declared as VAR_IN_OUT, it is assumed that values will be supplied both to and from the organisation unit.

Often, variables are required for intermediate results, which are to remain unknown externally. Locally named variables such as these are initiated with VAR and closed with END_VAR.

```
VAR
    Z          : INT;  (*Intermediate result                                 *)
END_VAR
```

*Fig. B6.9: Declaration of a local variable*

The variables specified here are local to an organisation unit and can only be used within this. They are unknown in all other organisation units and therefore inaccessible.

One typical application are memory locations for intermediate results, which are not of any interest in other areas of the program. In the case of these variables, it should be noted that they may also exist several times in different organisation units. In this way, it is for instance possible for several function blocks to declare the local variable Z. These local variables are totally unrelated and differ from one another.

A variable may also be globally declared, in which case it may be accessed universally. The necessary declaration is carried out in a similar way, whereby the keywords VAR_GLOBAL and VAR_EXTERNAL are used.

*Fig. B6.10: Declaration of a global variable*

```
VAR_GLOBAL
    Global_value: INT;
END_VAR
```

This is how all global data for a control program is declared. Global data is universally accessible. This declaration can only be found in the organisation units configuration and resource.

*Fig. B6.11: Declaration of access of a global variable*

```
VAR_EXTERNAL
    Global_value: INT;
END_VAR
```

In order to facilitate access of global data to an organisation unit, this declaration is to be recorded in the organisation unit.

Without the declaration shown above, access to global data would not be permissible.

This very strict declaration unit for all variables uniquely defines which variables are recognised by an organisation unit and how it may be used. A function block may for instance read but not change its input variables, and a program using a function block may read only but not change its output variables.

The keyword AT is used to assign variables to the inputs and outputs of the controller.

*Fig. B6.12: Declaration of variables with assignment to inputs of a controller*

```
VAR
    Stop_button AT %I2.3: BOOL;
    Temperature AT %IW3: INT;
END_VAR
```

Declarations in this form are the best means for defining the significance of all inputs and outputs of the controller. If a change occurs in the system and its connection to the controller, only these declarations need be changed. Any usage of the Stop_button, or the temperature within an existing program, remain unaffected by this.

According to IEC 1131-3 it is however nevertheless possible to use directly addressed variables without being assigned to a symbolic identifier. The declaration in that case is as follows:

```
VAR
    AT %I4.2      : BOOL;
    AT %MW1       : WORD;
END_VAR
```

## Initialisation

Very often it is essential for a variable to be given an initial value. This value may change several times during the processing of a program, even though it is defined at the start.

Initial statuses such as these are also important for other data. Such initial values are specified jointly with the declaration of the variables. A global variable of this type named Dozen is to be declared which, at the start of the program, assumes the value 12.

```
VAR_GLOBAL
    Dozen     : INT := 12;
END_VAR
```

*Fig. B6.13: Declaration of a global variable with initial value*

As shown by this example, the initialisation value is always inserted between the data type – in this instance INT – and the closing semicolon. The specification of the initialisation value always requires the prefixed symbol :=.

In this way, each variable can be assigned a special initial value. Fundamentally, variables always have a defined initial value at the start of a program. This is facilitated by the characteristic defined in IEC 1131-3, whereby data types already have a preset value. Each variable is preallocated the initial value of the corresponding data type – unless otherwise specified in the program. A list of initial values of a selection of elementary data types can be seen in table B6.7.

| ***Data type*** | ***Initial value*** |
|---|---|
| BOOL, SINT, INT, DINT | 0 |
| UINT | 0 |
| BYTE, WORD | 0 |
| REAL | 0.0 |
| TIME | T#0s |
| STRING | ” (void string) |

*Table B6.7: Preset initial values*

## 6.3 Program

The program for a controller is divided into individual organisation units, which are as follows at the programming level:

- Programs
- Function blocks
- Functions

These program organisation units are available in all programming languages.

IEC 1131-3 defines a wide range of standardised functions and function blocks for typical control tasks. Apart from these specified functions and function blocks, IEC 1131-3 permits the definition of own functions and function blocks. Manufacturers or users can thus produce tailor-made software modules for a particular application.

**Functions**
Functions are software modules which, when invoked provide exactly one result (data element). This is why in a text language the invocation of a function may be used as an operand in one expression.

Functions cannot contain status information. This means that the invocation of a function with the same arguments (input parameters) must provide the same result.

The addition of INT values or logic OR functions are examples for functions.

Functions and their invocation may be represented graphically or in text form.

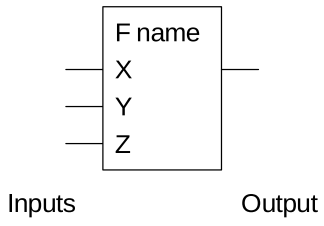


*Fig. B6.14: Graphic representation of a function*

Graphically, a function is represented in the form of a rectangle. All input parameters are listed on the lefthand side, the output parameters are shown on the righthand side. The function name is entered within the rectangle. Formal input parameters may be specified along the edges within the rectangle. This is necessary with some function groups, such as the bit shift functions for instance (fig. B6.15b). For functions with identical inputs, such as in the case of the ADD function (fig. B6.15a), no formal parameter names are required.

```
VAR
    AT %QW4 : INT;
    AT %IW9  : INT;
    AT %IW7  : INT;
    AT %MW1 : INT;
END_VAR
```

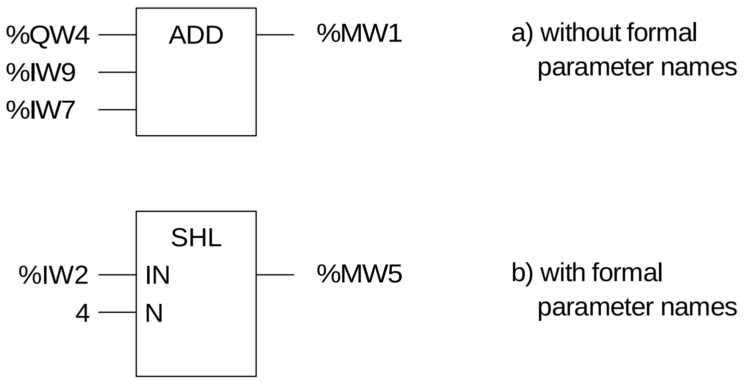

%QW4
%IW9
%IW7
ADD
%MW1
a) without formal parameter names
SHL
%IW2
4
IN
N
%MW5
b) with formal parameter names

*Fig. B6.15: Use of formal parameters with functions*

The boolean inputs or outputs of a function may be negated, i.e. by specifying a circle directly outside the rectangle (fig. B6.16)

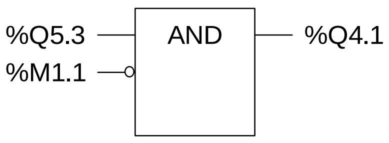


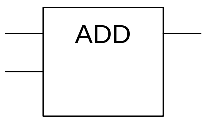
*Fig. B6.16: Representation of boolean negations*

If a function is invoked, its inputs and the function output must be connected.

The ADD function illustrated in fig. 6.15a processes INT values, for which the deployed directly addressed variables such as %QW4 etc. are declared as variables of data type INT. Equally, the ADD function could be applied to type SINT or REAL counter values.

Functions such as these, which operate to input parameters of a different data type are termed as overloaded, type-independent functions in IEC 1131-3. Fig. B6.17 illustrates the characteristics of an overloaded function using the example of an ADD function.

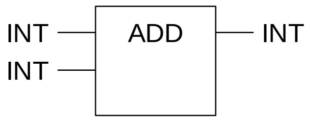
ADD function as example for an overloaded function



All data types defining numbers are permissible as input and result parameters

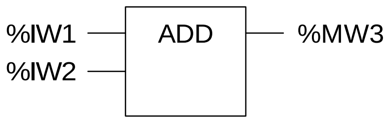
a) Input parameters of type INT

**general**



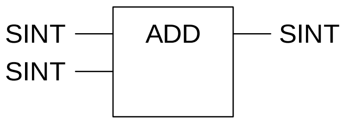
**example**

```
VAR
      AT %IW1   : INT;
      AT %IW2   : INT;
      AT %MW3 : INT;
END_VAR
```



b) Input parameters of type SINT

**general**



**example**

```
VAR
      AT %IB4    : SINT;
      AT %IB5    : SINT;
      AT %MB6  : SINT;
END_VAR
```

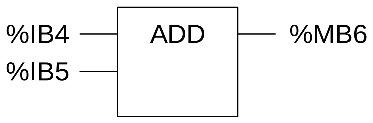
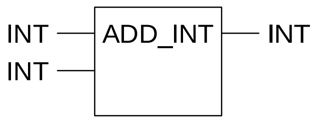


*Fig. B6.17: Overloaded, type-independent function*

If an overloaded function is limited to a particular data type by the controller – i.e. data type INT as shown in fig. B6.18 – this is referred to as a typed function. Typed functions are recognisable by their function name. Typing is indicated by the addition of the underline, followed by the desired type.

**general**



**example**

```
VAR
    AT %IW1   : INT;
    AT %IW2   : INT;
    AT %MW3   : INT;
END_VAR
```

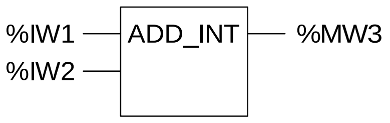


*Fig. B6.18:*
*A typed function*

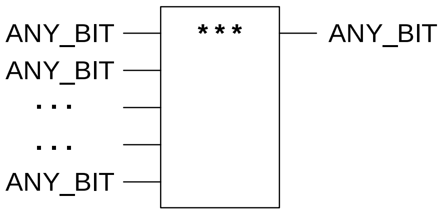
### Standard functions

The most important standard functions for the realisation of basic control technology tasks are listed below.

Since a wide variety of standard function are able to operate using input parameters of different data types, the data types have been combined into groups. Each group is given a generic data type. The most important generic data types are shown in table B6.8.

| | |
|---|---|
| ANY_NUM | all data types for floating point numbers such as REAL and for integer numbers such as INT, UINT etc., are contained in ANY_REAL and ANY_INT. |
| ANY_INT | all data types for integers such as INT, UINT etc. |
| ANY_REAL | all data types, defining floating-point numbers such as REAL or LREAL |
| ANY_BIT | all bit sequence data types such as BOOL, BYTE, WORD etc. |

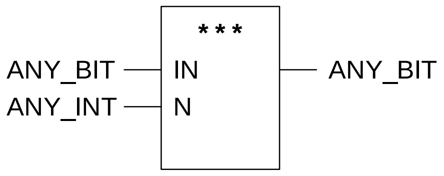
*Table B6.8:*
*Generic data types*



*** = name or symbol

Table B6.9: Bit-by-bit boolean functions

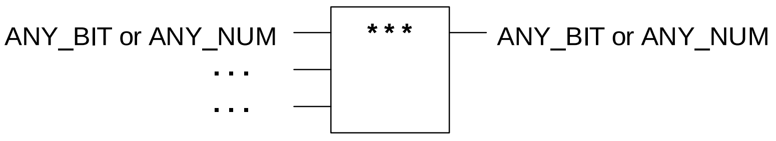
| ***Name*** | ***Symbol*** | ***Description*** |
|---|---|---|
| AND | & | AND operation of all inputs |
| OR | >=1 | OR operation of all inputs |
| XOR | =2k+1 | Exclusive OR operation of all inputs |
| NOT | | Negating input |



*** = name

Table B6.10: Bit shift functions

| ***Name*** | ***Description*** |
|---|---|
| SHL | Shift IN by N bits to the left, fill with zeros to the right |
| SHR | Shift IN by N bits to the right, fill with zeros to the left |
| ROR | Shift IN cyclically by N bits to the right |
| ROL | Shift IN cyclically by N bits to the left |

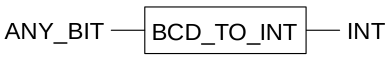


* * * = name or symbol

| ***Name*** | ***Symbol*** | ***Description*** |
|---|---|---|
| GT | > | Greater (falling sequence) |
| GE | >= | Greater or equal (monotonic sequence) |
| EQ | = | Equal |
| LE | <= | Less or equal (monotonic sequence) |
| LT | < | Less (rising sequence) |
| NE | <> | Not equal, non expandable |

*Table B6.11:
Comparison functions*

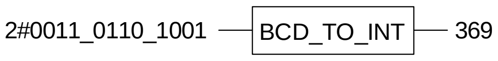
a) Graphic representation



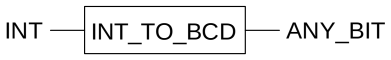
**Description:**

Converts variables of type BYTE, WORD etc. into
variables of type INT.
The bit sequence-variable contains data in BCD format.
(binary coded decimal number)

**Example:**



b) Graphic representation



**Description:**

Converts variables of type INT into variables of type BYTE,
WORD etc.
The bit sequence-variable contains data in BC format.

**Example:**

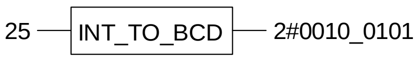
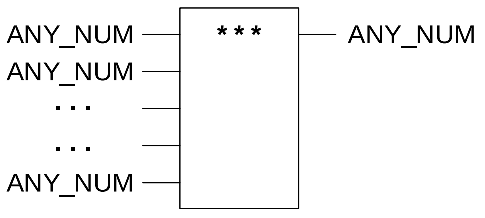


*Table B6.12:
Functions for
type conversion*



*** = name or symbol

Table B6.13: Arithmetic functions

| ***Name*** | ***Symbol*** | ***Description*** |
|---|---|---|
| ADD | | Add all inputs |
| MUL | * | Multiply all inputs |
| SUB | – | Subtract second input from first input |
| DIV | / | Divide first input by second input |
| MOVE | := | Assign input to output, non expandable |

**Function blocks**
Function blocks are software modules, which supply one or several result parameters.

One important characteristic is the possibility of instantiation of function blocks. If a function block in a control program is to be used, a copy or instance must be created. This is effected via the assignment of an instance name. Linked to this identifier is a data structure, which stores the statuses of this function block copy (values of the output parameters and internal variables). The status information of the function block copy remains intact from one processing to the next.

This can be demonstrated using the example of the standard function block for counting operations. The current counter value remains from one counting operation to the next and can thus be interrogated at any chosen time. This type of behaviour cannot be realised via the language resource, as described above.

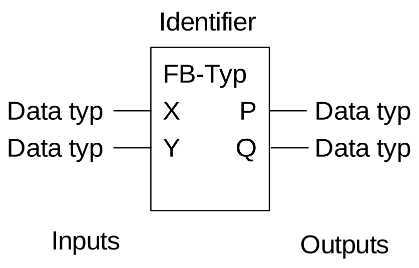


*Fig. B6.19:*
*Graphic representation of a function block copy*

Graphic representation of function blocks is also available (apart from representation in one of the text languages). These are represented by rectangles in the same way as functions (fig. B6.19). The input parameters are entered on the left and the output parameters on the right. The type of function block is specified within the rectangle. Then formal parameter names are entered on the left and right edges within the box. The identifier, under which the module is addressed is above the function block.

If a function block is used, then this must be given a identifier. If the inputs are allocated – i.e. current transfer parameters are available – then these are used for processing. If the inputs are not connected, then the stored values of previous invocations are accessed again or the corresponding initial values are used.

Fig. B6.20 shows the use (invocation) of the standard function block for a counter.

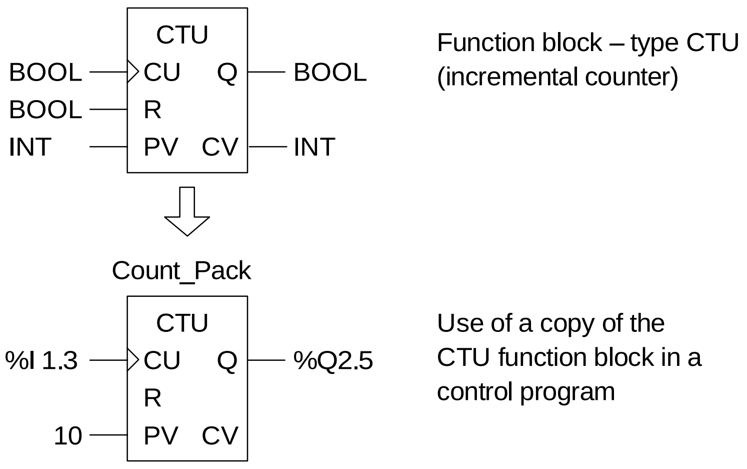

CTU
BOOL
BOOL
INT
CU
R
PV
Q
CV
BOOL
INT
Function block – type CTU (incremental counter)
Count_Pack
CTU
%I 1.3
10
CU
R
PV
Q
CV
%Q2.5
Use of a copy of the CTU function block in a control program

*Fig. B6.20:*
*Use (invocation) of the CTU function block (incremental counter)*

The copy used of the function block CTU bears the identifier Count_Pack. Each positive switching edge of input %I1.3 increases the current counter value by 1. When the set preselect value 10 has been reached, output Q of Count_pack and thus output %Q2.5 assumes a 1-signal; in all other cases a 0-signal is assumed.

It is also possible for several copies to be created of one and the same function block within a control program, as illustrated fig. B6.21.

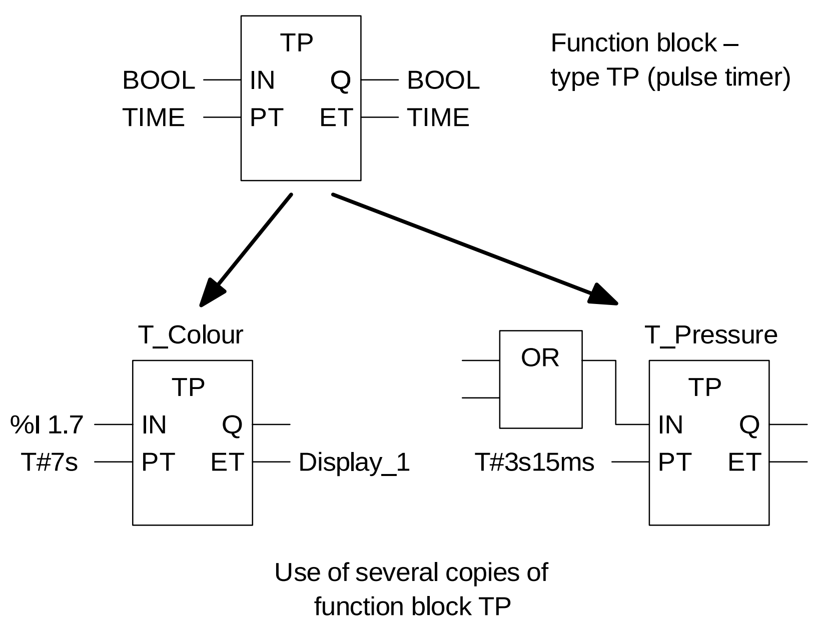


*Fig. B6.21: Use of several copies of a function block*

## Standard function blocks

Table B6.14 lists the most important function blocks standardised by IEC 1131-3.

| | |
|---|---|
| SR | Bistable function block (primarily setting) |
| RS | Bistable function block (primarily resetting) |
| CTU | Incremental counter |
| CTD | Decremental counter |
| TP | Pulse |
| TON | Switch-on signal delay |
| TOF | Switch-off signal delay |
| R_TRIG | Edge detection: rising edge |
| F_TRIG | Edge detection: falling edge |

*Table B6.14:*
*Standard function blocks*

## User-defined functions

Apart from the functions specified, IEC 1131-3 permits the definition of own functions.

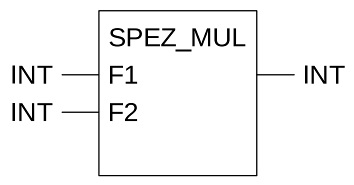
The following rules apply for graphic declaration:

- Declaration of the function within a FUNCTION... END_FUNCTION construct.
- Specification of the function name and the formal parameter names and data types of input and outputs of the function.
- Specification of the names and data types of internal variables used in the function; a VAR... END_VAR construct may be employed for this. No function block copies may be used as internal variables, since these would necessitate the storing of status information.
- Programming of the function body in one of the languages LD, FBD, IL or ST.

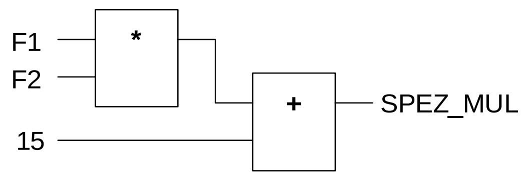
The sample function SPEC_MUL in fig. B6.22 is given two parameters of type INT. The two parameter values are multiplied, and the figure 15 added. The value calculated in this way feeds back the function as a result.

FUNCTION

(* Description of external interface *)



(* Functionbody: *)
(* Programmed in FBD language *)



END_FUNCTION

*Fig. B6.22:*
*Example function SPEZ_MUL*

The use of a function could be as demonstrated in fig. B6.23.

```
VAR
      AT %MW1  : INT;
      AT %MW2  : INT;
      AT %MW3  : INT;
      AT %IW4  : INT;
      AT %QW5  : INT;
END_VAR
```

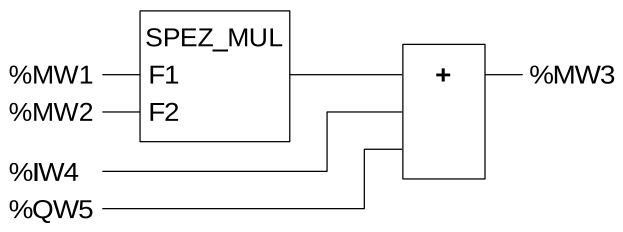


*Fig. B6.23:*
*Use of SPEZ_MUL function*

**User-defined function blocks**
The generation of own function blocks by the user is an important feature of IEC 1131-3.

The following rules listed apply for graphic declaration:

- Declaration of function blocks within a FUNCTION_BLOCK... END_FUNCTION_BLOCK construct.
- Specification of the function block name and the formal parameter names and data types of the inputs and outputs of the function block.
- Specification of the names and data types of internal variables; a VAR... END_VAR construct may be employed.
- Programming of the function block body in one of the languages LD, FBD, IL, ST or SFC .

Extended access of data, such as global variables are not taken into account here.

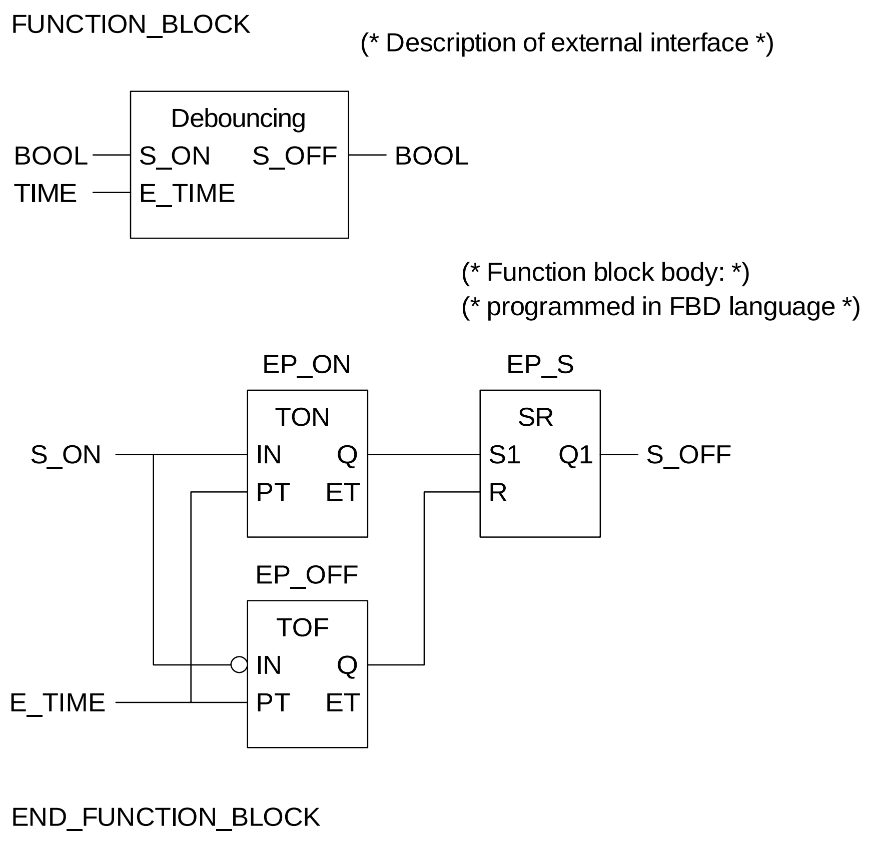


*Fig. B6.24 : Declaration of a function block*

The function block illustrated in fig. B6.24 represents a function block for the debouncing of signals, consisting of two input parameters, i.e. one boolean input for the signal and one time input for the adjustment of debounce time. The output parameter S_OFF supplies the debounced output signal.

**Programs**
A program consists of any language elements and constructs necessary to achieve the desired machine or process behaviour via the PLC.

Programs are therefore constructed in the main for functions, function blocks and the elements of the sequential function chart.

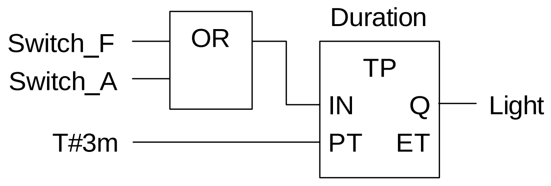
Program features are thus largely identical to those of function blocks. The only thing of interest at this stage are the differences:

- The delimiting keywords for program declarations are PROGRAM...END_PROGRAM.
- The use of directly addressable variables is permitted within programs only.

An example of this is illustrated in fig. B6.25.

```
PROGRAM staircase_light

VAR
  Switch_F    AT %IX0.0 : BOOL;  (* Light switch at front door        *)
  Switch_A    AT %IX0.1 : BOOL;  (* Light switch at appartment door   *)
  Lamp        AT %QX0.0 : BOOL;  (* Stairwell light                   *)
  Duration              : TP;    (* Time illuminated                  *)
END_VAR
```



```
END_PROGRAM
```

Fig. B6.25:
Example of a program

The name of the program is "staircase_light". Three boolean variables Switch_F, Switch_A and Lamp, allocated to two PLC inputs and one output, have been declared as internal variables. Added to this is the declaration of a function block copy of type Pulse Timer (TP).

The program realises the following small task:

The stairwell light is switched on for three minutes, if one of the two light switches on the apartment door or the front door is activated.

# *Chapter 7*

# *Function block diagram*

## 7.1 Elements of the function block diagram

The function block diagram is a graphic programming language, which is consistent, as far as possible, with the documentation standard IEC 617, P.12.

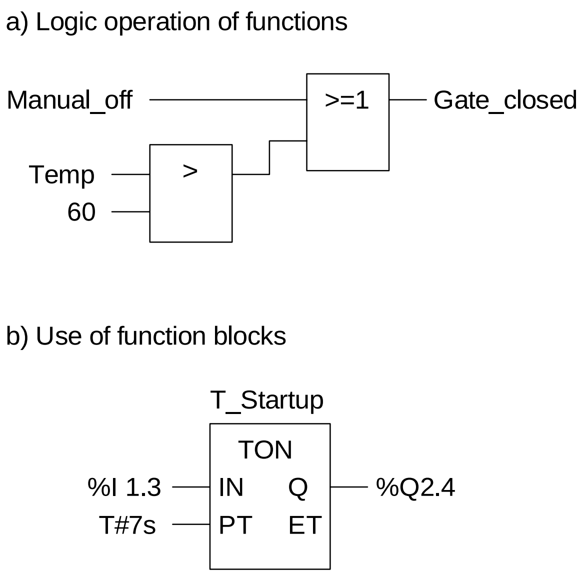


*Fig. B7.1: Function block diagram (FBD)*

The elements of the function block diagram are graphically represented functions and function blocks. These are interconnected by signal flow lines; directly linked elements form a network.

Fig. B7.1 illustrates two simple examples of the function block diagram. In fig. B7.1a, the variable Manual_closed and the result Greater-Comparison are OR'd. The result is mapped onto the variable Gate_closed. Fig. B7.1b represents the use of a function block. The signal delay T_startup is started via the input %I1.3 with the preset time of 7 seconds. The signal delay status, T_startup.Q, is assigned to the output %Q2.4.

## 7.2 Evaluation of networks

The direction of the signal flow in a network is from left to right. If a program organisation unit consists of several networks, these are processed in sequence from top to bottom.

The processing sequence within a program organisation unit may be influenced through the use of elements for execution control. This group of elements includes for instance the conditional and the unconditional jump. In fig. B7.2 a conditional jump is used to realise a program branch.

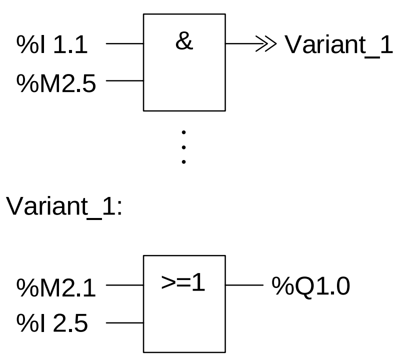


*Fig. B7.2: Use of a jump in FBD*

The conditional jump, represented via a double arrow, is executed, if the jump condition is met. Therefore, if input %I1.1 and flag %M2.5 both carry a 1-signal, then a jump will be executed to network with the identifier Variant_1 and processing continued at this point.

If a jump is to be executed to a network, the corresponding network must be prefixed with a symbolic name, the jump flag, ending with a colon. The jump flag must be identified in accordance with the rules for symbolic names.

## 7.3 Loop structures

When programming in the FBD language, it should be noted that no loop structures are permissible within networks. Structures of this type may be realised solely through the additional use of a feedback path. Fig. B7.3b illustrates an example of this.

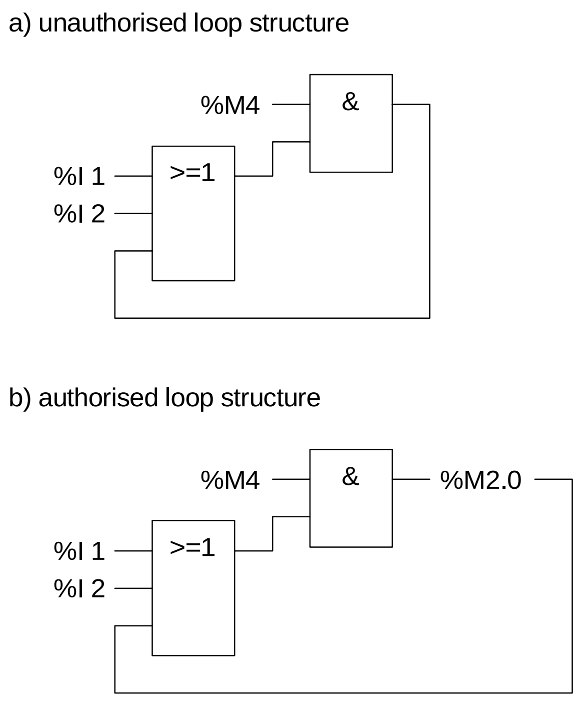


Fig. B7.3:
FBD with loop structures

By means of the utilisation of a feedback path, the third input of the OR-function assumes a defined value during its processing.

# Chapter 8

# Ladder diagram

## *8.1 Elements of the ladder diagram*

The ladder diagram language, like the function block diagram, represents a graphic programming language. The elements available in a ladder diagram are contacts and coils in different forms. These are laid out in rungs within the confines of power rails on the left and on the right.

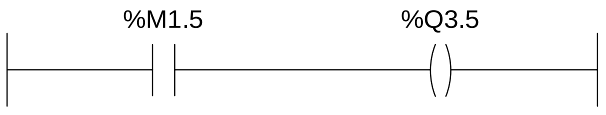
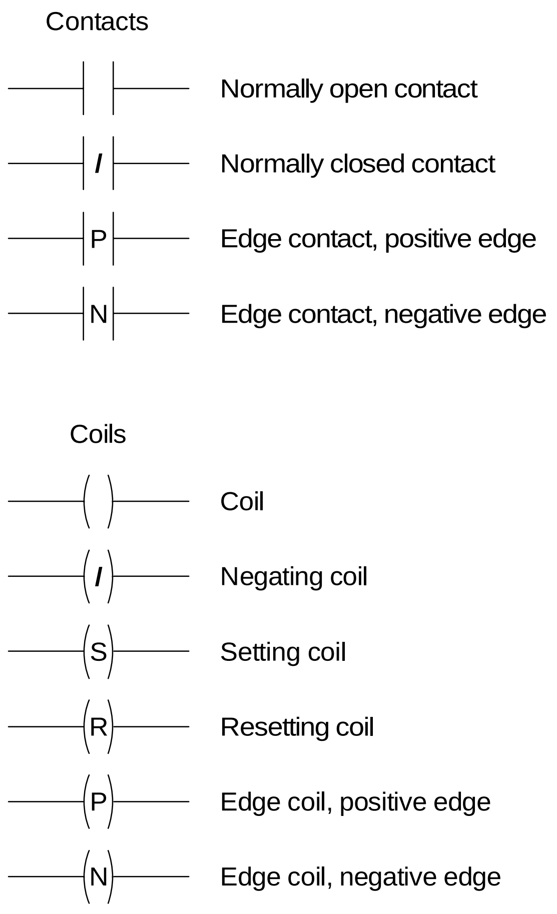


*Fig. B8.1: Basic structure of a rung*

Fig. B8.1 illustrates the basic structure of a current rung. In this example, the status of the flag %M1.5 is directly assigned to %Q3.5. Table B8.1 contains a list of the most important elements of a ladder diagram.



| | |
|---|---|
| **Contacts** | |
| —\| \|— | Normally open contact |
| —\|/\|— | Normally closed contact |
| —\|P\|— | Edge contact, positive edge |
| —\|N\|— | Edge contact, negative edge |
| **Coils** | |
| —( )— | Coil |
| —(/)— | Negating coil |
| —(S)— | Setting coil |
| —(R)— | Resetting coil |
| —(P)— | Edge coil, positive edge |
| —(N)— | Edge coil, negative edge |

*Table B8.1: Elements of the ladder diagram*

A normally open contact supplies the value 1, when the corresponding push button switch is closed. A normally closed contact reacts correspondingly with the value 1, when the switch or push button is opened.

Two edge contacts supply the value 1 for the transition from 0 to 1 (positive edge) or from 1 to 0 (negative edge).

With a normal coil, the result (resulting logic operation of the contacts) is copied to the specified variable; in the case of a negating coil, the negation of the result is transferred.

The setting coil assumes the value 1, if the result is 1, and remains unchanged even if the result is 0 in between. Similarly the resetting coil only assumes the value 0, if the result is 1. The 0 status of the coil is maintained.

The two edge coils are set, if the result changes from 0 to 1 (positive edge) or from 1 to 0 (negative edge).

The basic functions AND and OR may be realised by means of a corresponding configuration of contacts in the current rung.

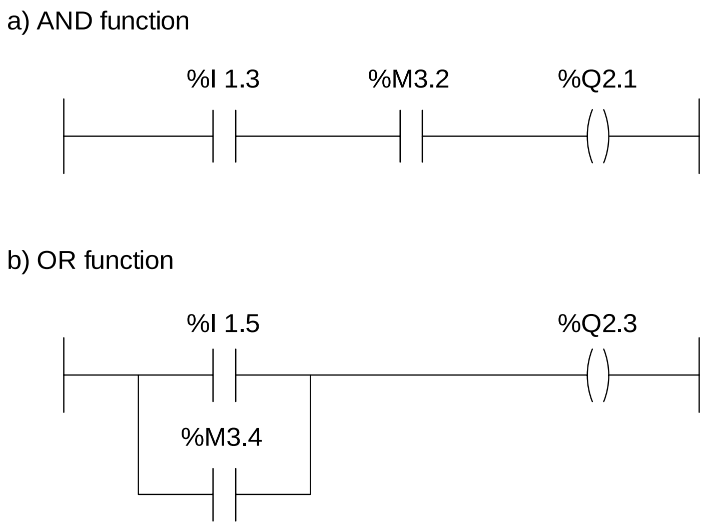


*Fig. B8.2:
Basic logic connections in ladder diagram*

The AND function is represented by means of the serial connection of the two contacts (fig. B8.2a). Output %Q2.1 is set only if both input %I1.3 and flag %M3.2 are set. In all other cases, output %Q2.1 is reset.

The OR function is obtained via parallel switching of contacts (fig. B8.2b). Output %Q2.3 assumes the value 1, if either input %I1.5 or flag %F3.4 assume the value 1, or if both conditions are met simultaneously.

## *8.2 Functions and function blocks*

Apart from the contact and coil elements, LD provides the unlimited use functions and function blocks in so far as this feature is supported by the controller used.

Prerequisite for the incorporation of so-called organisation units, is the availability of at least one boolean input and one boolean output of the block in question. If this is not the case, a boolean input with the formal parameter EN (enable) is added to the corresponding functions or function modules as well as a boolean output ENO (enable OK). The boolean inputs/outputs are required to permit the power flow through the block.

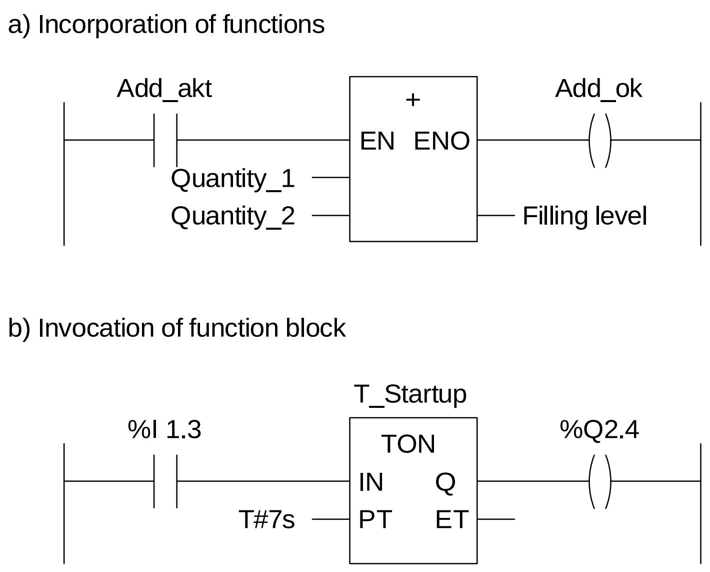


*Fig. B8.3: Functions and function blocks in ladder diagram*

The addition shown in fig. B8.3a is only undertaken, if a 1-signal is applied at the input EN. If this is the case, the variables Quantity_1 and Quantity_2 are added and the result of these variables assigned to the variable Filling Level. At the same time, the value of output ENO indicates, whether the addition has been executed, activated and correct (ENO=1). If the block has not been processed correctly, the output ENO assumes the value 0.

Function modules such as for instance the signal delay shown in fig. B8.3b can be incorporated in the ladder diagram without additional EN input and ENO output. The function block is connected with the elements of the current rung in the usual manner via the boolean input IN and the boolean output Q. If input %I1.3 in fig. B8.3b assumes the value 1, the function block copy T_Start is processed with the preset time duration of 7 seconds. The value of output Q of T_Start is assigned to output %Q2.4.

## 8.3 Evaluation of current rungs

Similar to the graphic programming language FBD, the power flow, and as such processing within a program organisation unit, is from left to right and from top to bottom. Equally, the processing sequence may also be changed in LD by using elements for execution control.

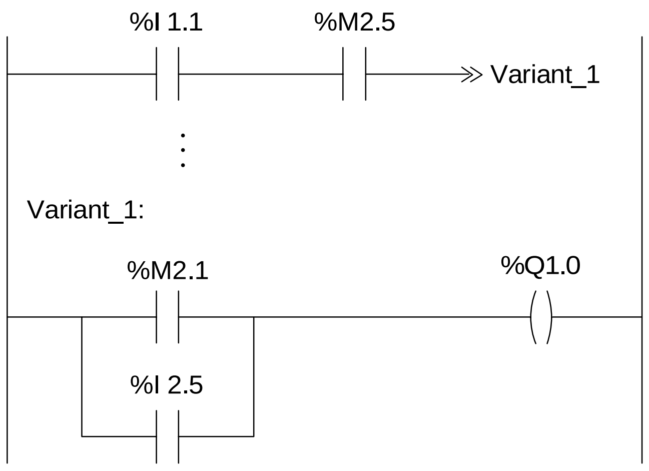


*Fig. B8.4:*
*Conditional jump in LD*

If the jump condition, in this case the AND operation of input %I1.1 and flag %M2.5, is met, a jump is executed to the current rung with the identifier Variant_1. Processing is then continued from this current rung onwards.

# Chapter 9

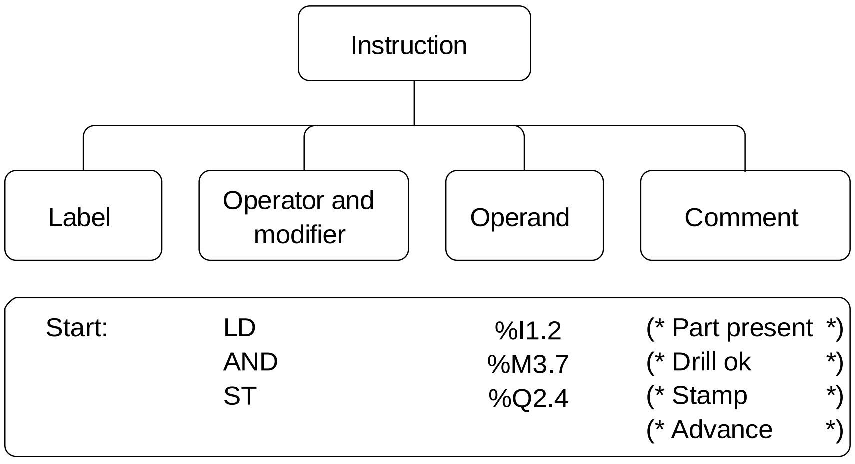
# Instruction list

## 9.1 Instructions

Instruction list is a textual, assembler-type programming language. Its instructions most closely ressemble the commands processed in a PLC.

A control program formulated in the Instruction List language consists of a series of instructions, whereby each instruction must begin with a new line.

A fixed format is specified for the formulation of an instruction. An instruction (fig. B9.1) starts with an operator with optional modifier and, if necessary for the particular operation, one or several Operands, separated by commas. Instructions may be preceded by a label followed by a colon. The label acts as a jump address. Labels are identifed in the same way as symbols. If a comment is used, this must represent the last element of the line. A comment is introduced via the string (*, and ended by the string *).



```
Start:   LD      %I1.2   (* Part present  *)
         AND     %M3.7   (* Drill ok      *)
         ST      %Q2.4   (* Stamp         *)
                         (* Advance       *)
```

*Fig. B9.1: Structure of an instruction*

The value of input %I1.2 is loaded to the accumulator and ANDed with value of flag %M3.7. The resultant actual result is assigned to output %Q2.4.

## 9.2 Operators

| *Operator* | *Modifier* | *Operand* | *Description/Significance* |
|---|---|---|---|
| LD | N | | Reads the specified operand to the accumulator and equates the current result to this operand |
| ST | N | | Stores the current results at specified operands |
| S | | BOOL | Sets boolean operand to the value 1, if the value of the current result is a boolean 1 |
| R | | BOOL | Resets the boolean operand to 0, if the value of the current result is a boolean 1 |
| AND | N, ( | BOOL | Boolean AND |
| & | N, ( | BOOL | Boolean AND |
| OR | N, ( | BOOL | Boolean OR |
| XOR | N, ( | BOOL | Boolean exclusive OR |
| ADD | ( | | Addition |
| SUB | ( | | Subtraction |
| MUL | ( | | Multiplication |
| DIV | ( | | Division |
| GT | ( | | Comparison: > |
| GE | ( | | Comparison: >= |
| EQ | ( | | Comparison: = |
| NE | ( | | Comparison: < > |
| LE | ( | | Comparison: <= |
| LT | ( | | Comparison: < |
| JMP | C, N | Label | Jump to label |
| CAL | C, N | Name | Invocation of function blocks |
| RET | C, N | | Return of function or function block |
| ) | | | Processing of a reset operation |

*Table B9.1:*
*Instruction list operators*

IEC 1131-3 defines the operators for the instruction list listed in table B9.1.

Operators are not linked with any priorities. Accordingly, operations are processed in the sequence, in which they are entered in the instruction list. If a different sequence is desired, this can be achieved through the use of brackets – so-called modifiers. Fig. B9.2 explains the use of some modifiers.

```
LDN     %I1.1     The value of input %I1.1 is loaded
                  to the accumulator negated
AND(    %I1.2     First the content in the parenthesis is
                  evaluated– in this example
OR      %I1.3     inputs %I1.2 and %I1.3
                  are OR connected – the result
)                 of the parenthesised expression
                  is AND connected with the
                  current result in the accumulator
JMPC    Start     The jump to the Start label is executed
                  only if the value of the result
                  just executed is a boolean 1.
```

*Fig. B9.2: Use and significance of modifiers*

## 9.3 Functions and function blocks

In instruction list too, the use of functions and function blocks is unlimited. Functions for typical control technology tasks, such as boolean logic or arithmetic operations (see fig. B9.3.a) are realised direct via the operands specified in table B9.1.

**a) Invocation via operator**

```
LD      Temp          (* Measured temperature*)
GT      60            (* Greater than 60 *)
OR      Manual_off    (* OR button Manual_off actuated *)
ST      Gate_off      (* Close gate *)
```

**b) Invocation via function name**

```
LD      %IW3          (* Load input word %IW3 *)
SHL     4             (* Shift %IW3 by 4 bits to the left *)
ST      %QW2          (* Store result in output word %QW2 *)
```

*Fig. B9.3: Invocation of functions*

More complex functions such as bit shift functions are invoked by the function name being placed in the operator field. The current result (in the Accumulator) must be used as the first argument of the function. If further arguments are required, these must be specified in the operand field separated by commas. The value returned by the function, represents the new current result.

Function blocks may be invoked according to different mechanisms (fig. B9.4).

**a) CAL with list of input parameters**

```
CAL     T_startup(IN := %I1.3, PT := T#7s )
```

**b) CAL with separate loading/storing of input parameters**

```
LD      T#7s             (* Load T#7s *)
ST      T_startup.PT     (* Store to T_startup.PT *)
                         (* -input parameter for preselected time *)
LD      %I1.3            (* Load %I1.3 *)
ST      T_startup.IN     (* Store to T_startup.IN *)
                         (* - transfer parameter for activation *)
                         (* of module *)
CAL     T_startup        (* Invocation of function block copy T_startup *)
```

*Fig. B9.4: Invocation of function blocks*

Prerequisite is for the variable T_Startup to be declared as a signal delay. The invocation of a function block may be clearly set out through the use of the operator CAL with a list of associated input parameters.

The variable T_Startup from fig. B9.4a, declared elsewhere as a signal delay, therefore represents a signal delay type block. Being a current argument, this is assigned the value of input %I1.3 for the activation input IN and the time duration T#7s for the input PT. As a result, function block T_Startup is invoked following the actualisation of parameters.

The transfer of parameters to a function block may also be effected separately from the actual function block invocation.

As shown in fig. B9.4b, the actual parameter values are loaded via elementary IL operations and assigned to the inputs of the function block. Only after this is function block T-Startup invoked and processed via a CAL instruction. The advantage of this method lies in the fact that the timing of the actualisation of arguments and the actual invocation of the function module may be separate.

# Chapter 10

# Structured text

## 10.1 Expressions

The Structured Text language is a Pascal-type high-level language, which incorporates the fundamental concepts of a modern high-level language, in particular the most important principles for the structuring of data and instructions. The structuring of data represents a common constituent of all five programming languages; the structuring of instructions, however, is an important feature of ST only.

An expression is an elementary constituent for the formulation of instructions. An expression consists of operators and operands. Frequently occurring operands are data, variables or function invocations. However, an operand may also be an expression itself. The evaluation of an expression supplies a value corresponding to one of the standard data types or to a user data type. For instance, if X is a number of the type REAL, then the expression SIN(X) also supplies a REAL type number. Table B10.1 contains an overview of operators.

| ***Operation*** | ***Symbol*** | ***Priority*** |
|---|---|---|
| Parenthesis | (expression) | highest |
| Function processing | Function name<br>(Transfer parameter list)<br>Example: LOG(X), SIN(Y) | |
| Exponentiation | ** | |
| Sign<br>Complement | –<br>NOT | |
| Multiplication<br>Division<br>Modulo | *<br>/<br>MOD | |
| Addition<br>Subtraction | +<br>– | |
| Comparison | <, >, ≤, ≥ | |
| Equality<br>Inequality | =<br><> | |
| Boolean AND | &, AND | |
| Boolean exclusive OR | XOR | |
| Boolean OR | OR | lowest |

*Table B10.1: Operators of structured text language*

The following are examples of expressions:

```
SIN(X)
4*COS(Y)
A ≤ B
A+B+C
```

The evaluation of an expression consists of applying the operators to the operands, whereby the operators are evaluated in a sequence defined by their precedence in table B10.1. An operator with higher precedence is evaluated prior to an operator with lower precedence.

*Example*

A, B and C are variables of data type INT; A assumes the value 1, B the value 2 and C the value 3. The evaluation of expression A+B*C supplies the value 7. If a sequence other than that specified by the precedence is desired, this is possible by using brackets. Using the above numeric values, the expression (A+B)*C supplies the value 9 9.

If an operator has two operands, the leftmost operand is to be evaluated first. The expression SIN(X)*COS(Y) is therefore evaluated in the sequence: Calculation of the expression SIN(X), calculation of the expression COS(Y), followed by the calculation of the product of SIN(X) and COS(Y).

As demonstrated in the previous paragraph, a function may only be invoked within an expression. The invocation is formulated by specifying the function name and the parenthesised list of arguments.

## 10.2 Statements

Table B10.2 contains a list of statements possible in the Structured Text language. A statement may extend beyond one line, whereby the line break will be treated in the same way as a blank space.

| ***Statement*** | ***Example*** |
|---|---|
| Assignment := | A := B;<br>CV := CV + 1;<br>Y := COS(X); |
| Invocation of function blocks | RS_Horn(S := Drill_faulty, R1 := push button);<br>Horn := RS_Horn.Q1; |
| Return from functions and function blocks RETURN | RETURN; |
| Selection statements | |
| IF | D:= B*B – 4*A*C;<br>IF D < 0.0 THEN Number_Sln := 0;<br>ELSIF D = 0.0 THEN<br>        Number_Sln := 1;<br>        X1 := –B / (2.0*A);<br>ELSE<br>        Number_Sln := 2;<br>        X1 := (–B + SQRT(D)) / (2.0*A);<br>        X2 := (–B – SQRT(D)) / (2.0*A);<br>END_IF; |
| CASE | CASE Voltage OF<br>        101 ... 200: Display := too_large;<br>        20 ... 100: Display := large;<br>        2 ... 29: Display := normal;<br>ELSE<br>        Display := too_small;<br>END_CASE; |
| Iteration statements | |
| FOR | Total := 0;<br>FOR I := 1 TO 5 DO<br>        Total := Total + I;<br>END_FOR; |
| REPEAT | Total := 0;<br>I := 0;<br>REPEAT<br>        I := I + 1;<br>        Total := Total + I;<br>UNTIL I = 5<br>END_REPEAT; |

*Table B10.2: Statements of structured text language*

| ***Instruction*** | ***Example*** |
|---|---|
| Iteration statements (continued) | |
| WHILE | Total := 0;<br>I := 0;<br>WHILE I < 5 DO<br>    I := I + 1;<br>    Total := Total + I;<br>END_WHILE; |
| Termination of loops EXIT | EXIT; |
| Void instruction | ;; |

*Table B10.2: Statements of structured text language (continued)*

**Assignments**
An assignment is the simplest form of an instruction. This replaces the actual value of the variable to the left of := with the value of the expression to the right of :=. Each assignment ends with a semicolon. One possible assignment (table B10.2) is A := B; whereby the value of the variable B is assigned to the variable A. As a result of the assignment CV := CV  + 1, the variable CV is increased by 1 as a result of the expression CV+1.

**Function blocks and functions**
A defined mechanism is set out in IEC 1131-3 for the invocation and also the early exit from a function or a function block.

As described, a function is invoked only as part of expression evaluation. The invocation itself consists of the specification of the function name, followed by the parenthesised list of input parameters.

The invocation of a function block is analogue through the specification of the instance name (copy) of the function block. This is followed by a parenthesised list consisting of value assignments to the input parameters. The specification of the name of the input parameter is mandatory, the individual input parameters may be listed in any sequence.

Moreover, it is not essential for all input parameters to be assigned a value in every invocation. If a particular input parameter is not assigned a value in an invocation, the previously assigned value or the initial value of the parameters applies.

Table B10.2 contains an example of a function block invocation. A horn is to sound, if a drill is defective. The status of the horn is stored by means of an RS function block.

The RETURN statement is to provide early exit from a function or a function block. The following is an example of the use of a RETURN command:

```
IF X <  0  THEN
        Value := -1;
        Error := 1;
        RETURN;
END_IF
Y := LOG(X);
```

If the value of X is less than 0, the block containing the sequence of statements is terminated immediately.

## *10.3 Selection statements*

Selection statements – also known as program branch statements – are available in the form of the IF and CASE statement. Different groups of statements may be selected and executed in relation to a defined condition. The program organisation unit may branch in different ways.

### IF statement

The general form of an IF statement is:

```
IF boolean expression1 THEN instruction(s)1;
[ ELSIF boolean expression2 THEN statement(s)2; ]
[ ELSE statement(s); ]
END_IF;
```

The parts in square brackets are optional, i.e. these may occur in an IF statement, but need not do so.

The simplest IF statement consists of an IF-THEN construct (simple branch).

This is demonstrated by the following example

```
IF X < 0 THEN X := –X;
END_IF;
Y := SQRT(X);
```

If the condition following the keyword IF is true, the statements following the keyword THEN are executed. If the condition is not met, the statements in the THEN part are not executed.

In the case of a concrete example this means: If the variable X is less than 0, i.e. negative, it is affixed a minus symbol and thus represents a positive value; if this is not the case, the statement with the square root function is executed immediately.

A simple branch may be achieved by means of an IF-THEN-ELSE construct:

```
Error := 0;
IF Part_ok THEN number := Number + 1;
ELSE error := 1;
END_IF;
```

The statements following the keyword THEN are executed, if the condition following the keyword IF is met; if the condition is not met, the statements formulated after the keyword ELSE are executed.

The example given deals with production parts. If the part is good (Part_ok = 1), the THEN part is executed, in this case the number of correctly produced parts is increased by 1; otherwise a bit is set for error detection.

If a branch is to be programmed for more than 2 branches, an IF-THEN-ELSIF construct may be employed. Table B10.2 illustrates this by way of an example, whereby the solutions of the quadratic equation $AX^2 + BX + C = 0$ are established. If the discriminant – in this case variable D – is less than 0, the subsequent THEN part is executed: there is no solution, i.e. Number_Sol := 0.

If the first condition is not met, i.e. D is greater or equal to 0, the condition following ELSIF will be evaluated: If it is met, i.e. D equals 0, the statements following the keyword THEN will be executed: The only existing solution is specified as X1.

Otherwise (D being greater than 0), the lines following the keyword ELSE will be executed: The two possible solutions X1 and X2 are specified.

**CASE statement**
If a selection of several possible statement groups is to be made, the CASE statement may be used.

The standard form for the multiple selection with CASE is:

```
CASE Selector OF
    Value1: statement(s)1;
    Value2: statement(s)2;
    ...
    Valuen: statement(s)n;
[ ELSE
    statement(s); ]
END_CASE;
```

The CASE statement consists of a selector, which supplies a variable of type INT during its execution, and a list of statement groups. Each group is assigned a value (label). The values are separated by commas if a statement group is dependent on several values. The values may also represent INT type variables.

With the evaluation of the CASE statement, the value of the selector is determined first, followed by the execution of the first group of statements, for which the computed value of the selector applies. If the value of the selector is not contained in any of the statement groups, the statements following the keyword ELSE are executed. If ELSE does not occur, no statements are executed.

In the example given in table B10.2, the text for a statement is selected in relation to the available measured value. The values for the selection of the statement are of the INT type.

## 10.4 *Iteration statements*

It is often necessary to execute statements repeatedly (program loops). The FOR loop is used, if the number of repetitions has been defined in advance, otherwise the REPEAT or the WHILE loop is used.

### FOR loop

The standard representation of the FOR loop is:

```
FOR Variable := Expression TO expression [ BY expression ] DO
    statement(s);
END_FOR;
```

A so-called control variable is set at a specific initial value and increased with every loop executed until the control variable reaches the end variable. A simple FOR loop is therefore executed according to the following mechanism:

```
FOR counter variable := Initial value TO final value DO
    Instructions;
END_FOR;
```

If no incremenets are specified, as formulated above, the control variable automatically increments by 1 with each loop completed. If a different increment is required, this may be specified via the keyword BY, followed by the desired value. The control variable may therefore not be changed within the loop – i.e. the statements being repeately executed. Furthermore, the control variable, initial value and final value must be expressions of the same integer data type (INT, SINT, DINT).

The test for the termination condition is made at the beginning of each iteration, so that the statements are not executed if the initial value exceeds the final value. A further characteristic of FOR loops is that these may be nested at any time.

An example of the application of a FOR loop is given in table B10.2. In this example, an addition of numbers 1 to 5 is realised via the loop. When the loop is executed for the first time, I has the initial value 1, the value of the variable Sum is also 1. For the second loop execution, I has the value 2, the variable Sum reaches the value 1+2=3 etc. After the fifth and last loop execution, the value for Sum is 15, the counter variable has reached the final value 5, and processing of the loop is thus completed.

## REPEAT loop

Unlike the FOR loop, with the REPEAT loop the number of iterations is not defined in advance via a specified final value. Instead, a condition – the so-called termination condition – is used.

The form of a REPEAT loop is as follows

```
REPEAT
    statement(s);
UNTIL Boolean expression
END_REPEAT;
```

The termination of the REPEAT loop is tested after the execution of the loop statements. The loop is therefore executed at least once. The termination condition must be changed in the loop, since the the loop will otherwise be executed indefinitely. It is therefore important to ensure that the loop has actually be completed. The following is to be checked:

- Does the termination condition actually include a variable, so that the condition can supply the value 1 (true)?
- Is the termination condition ever reached?

An example of the use of the REPEAT loop is demonstrated in table B10.2. Here too, the first five non-negative integers are added.

In the first loop execution, I has the value 1, the value of Sum is also 1. A check of the termination condition shows, that this is not met, whereby the loop is executed again. The loop is executed repeatedly until the termination condition is true. This will be the case after the fifth loop execution and the loop is ended. Here too, the result for the variable sum is 15.

**WHILE loop**
The WHILE loop represents a second option for the formulation of iterations by specifying a termination condition. The standard representation of a WHILE loop is:

```
WHILE Boolean expression DO
    statement(s);
END_WHILE;
```

If the boolean expression following the keyword WHILE is met, the statements following the keyword DO will be executed. The termination of the WHILE loop are therefore tested prior to the execution of the loop statements. The loop statements may therefore possibly not be executed at all. The termination condition is to be changed in the statements to be repeated.

It is important that the loop conditions are really met in order for the processing of the loop to be terminated.

The task of adding the numbers 1 to 5 can also be realised by using a WHILE loop (table B10.2). Unlike the REPEAT loop, the WHILE loop is executed repeatedly until the termination condition is true. In reality this means that the loop is executed for as long as I is less than 5. If I is equal or greater than 5, the loop is no longer processed.

In principle, a REPEAT loop can be replaced by a WHILE loop and vice versa.

**EXIT statement for the termination of a loop**
The EXIT statement must be used in order to terminate iterations before the end or termination condition is satisfied.

The following program illustrates the example of an EXIT statement:

```
S := 0;
FOR I := 1 TO 2 DO
    FOR J := 1 TO 3 DO
        IF error THEN EXIT;
        END_IF;
        S := S + J;
    END_FOR;
    (* If EXIT statement is executed then jump to this point is executed *)
    S := S + I;
END_FOR;
```

If the EXIT statement is within a nested loop, exit shall be from the innermost loop in which the EXIT is located. The next statement to be executed is the statement immediately after the loop end (END_FOR, END_WHILE, END_REPEAT). In the example given in fig. B10.1, this is the statement "S := S +I;".

The following applies in the case of the above example: If the value of the boolean variable Error is equal to 0, the algorithm for the variable S provides the value 15. If the variable Error has the value 1, the value computed for S is 3.

# Chapter 11

# Sequential function chart

## 11.1 Introduction

IEC 1131-3 defines the sequential function chart (SFC) as an important programming tool for control systems. Its clearly laid out structure provides a particularly clear program representation for a control system and as such is one of the most important parts of IEC 1131-3.

Each program of a sequence control system consists of steps and transitions (step enabling conditions). Apart from this, it also contains other important information concerning program execution and the type of program continuation.

If the program execution is not unique, but an individual path is to be selected from several possible paths, the representation of the sequential function chart illustrates this in a particularly graphic form.

## 11.2 Elements of the sequential function chart

The fundamental task of a sequential function chart is to structure a control program into individual steps and transitions (step enabling conditions) interconnected by directed links.

This requires a representation in graphic form, which makes the intention of the program clearly recognisable.

The IEC 1131-3 sequential function chart is structured in the form of a small defined set of simply constructed, graphic basic elements. These basic elements must be combined to create a control program. How this is achieved, is defined by a few simple rules in the standard.

The sequential function chart programming language is based, in as far as possible, on the function chart planning language to DIN 40 719 Part 6 or IEC 848. The only amendments made were those necessary to be able to generate executable commands for a PLC from a documentation element. An example of this is the action qualifier S. In the documentation standard, this qualifier is used to define the modes of action, i.e. the setting and resetting of an operand. A PLC requires unique commands. This is why the sequential function chart programming language employs two qualifiers to realise the two modes of action, the S and the R qualifier.

Since the sequential function charts requires the storing of status information (the active steps etc. at a given moment), the only program organisation units which can be structured using these elements are Programs and Function Blocks.

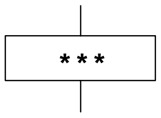
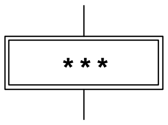
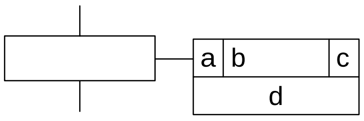
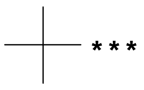
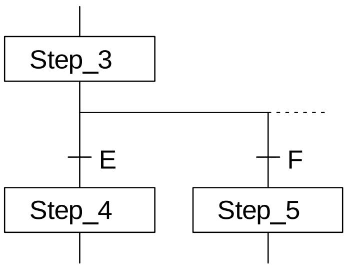
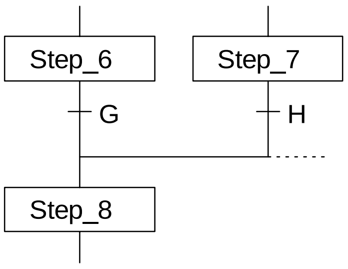
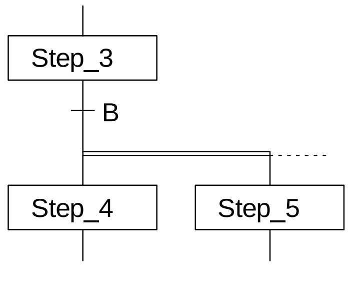
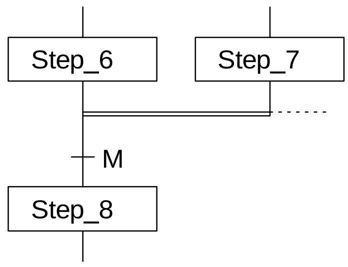
| | |
|---|---|
| a) Step with identifier *** |  |
| b) Initial step with identifier *** |  |
| c) Action block, containing the actions assigned to a step<br>Field a: Action qualifier<br>Field b: Name of action<br>Field c: Feedback variable<br>Field d: Action content |  |
| d) Transition with identifier *** or transition condition *** |  |
| e) Alternative branch |  |
| f) Junction of alternative paths |  |

*Table B11.1:
Elements of the sequential function chart
(graphic representation)*



g) Parallel (simultaneous) branch



h) Junction of parallel paths

*Table B11.1: Elements of the sequential function chart (graphic representation, continued)*

**The step**

A step contains a number of execution parts of the control program. Inputs and outputs may only be set or reset in a step. This also means that all correcting variables issued to the connected plant by a control program, can only be programmed in such steps.

The execution part assigned to a step, the so-called actions, are formulated within action blocks.

A step is either active, with the associated action being executed at the time, or inactive. In this way, the status of the connected system is defined at any given moment by the set of active steps in the control program.

As shown in table B11.1a, a step is represented graphically by a block. Each step has a symbolic name, which can be freely selected by the user. The same set of rules applies for the step name as those already mentioned for symbolic identifiers: a symbolic name may consist only of capital and lower case letters, digits and the underline and always starts with a letter or the underline.

Motor_3_on
Vacuum_off

*Fig. B11.1: Steps with step names*

All steps in a program or function block formulated in a sequential function chart must have different names. Even if two steps have the same execution parts, these are to be designated twice.

The reason for this is as follows:
Information is stored in the controller for each step. The unique assignment of this information to a step as well as the access to this data is effected via the step name.

The user can thus obtain information regarding

- the current status of a step (active, inactive),
- the time, for which a step has been active since initiation.

Table B11.2 illustrates the access to step data.

| | | |
|---|---|---|
| a) | Motor_3_on.X | boolean variable indicating whether the step Motor_3_on is active (Motor_3_on.X=1) or inactive (Motor_3_on.X=0) |
| b) | Motor_3_on.T | Variable of type TIME indicating how long the step Motor_3_on has been active since initiation. |

*Table B11.2: Information regarding a step*

The evaluation of the above data can be useful with regard to monitoring the connected system. Applications may also arise, which require the use of variables in the control program itself.

A special case within the step element is the so-called initial step (table B11.1b). This is drawn graphically by a double line.

Each network in a sequential function chart has precisely one initial step, which is executed as the first step within a network.

As already mentioned, the importance of a sequential function chart lies in its clearly structured graphic representation of a control program. It may however also be useful to represent sequence structures textually. The IEC 1131-3 standard provides an equivalent textual representation of SFC elements for this, which is as follows for the step element:

```
STEP Motor_3_on
   (* Contents of step *)
END_STEP


STEP Vacuum_off
   (*Contents of step*)
END_STEP
```

*Fig. B11.2: Textual representation of steps*

The textual representation of sequence structures may, depending on manufacturer, be part of the documentation of a control program (vendor-dependent); this type of layout of sequence structures does not require any restrictions with regard to format and character set for printing.

Moreover, it may be possible, for control programs in a standardised textual representation to be portable between PLCs by different manufacturers.

**The transition**
A transition or step enabling condition contains the logic condition permitting the transition, according to the program, from one step to the next.

As can be seen from table B11.1dh, the transition is represented by a horizontal line across the vertical directed link. Each transition has a transition condition, which is the result of the evaluation of a single boolean expression. The transition condition can be formulated in any of the IEC 1131 languages such as LD, FBD, IL or ST.

A transition condition is either true and then has the value 1, or false, when it has the value 0. Only if the condition is true, is the execution of the program or the function block continued at this point.

If a condition is always true, it can simply be identified by the number 1 at the transition. Transition conditions of this type which are always true may occur frequently in a program or function block in a sequential function chart.

| | |
|---|---|
| Always true transition condition | ┼ 1 |
| Transition condition never true | ┼ 0 |

*Table B11.3:*
*Special transitions*

**Interconnection of steps and step enabling conditions**
In practice, not much can be achieved with a single step or with a transition. A control program in the sequential function chart will therefore always be made up of a succession of numerous transitions and steps.

A sequence of transitions and steps is termed a step chain, sequence or also path.

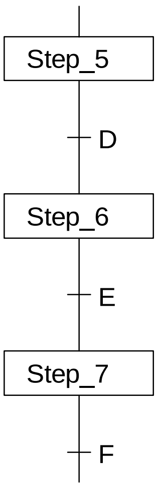


*Fig. B11.3:*
*Steps and*
*transitions in sequence*

Here, the transitions and steps must continually alternate. The logic path via this representation is always from top to bottom. The following behaviour can be seen in the example shown in fig. B11.3:

Assuming that step Step_5 is active, Step_5 remains active until transition D is true. Clearing of the transition results in the deactivation of the preceding step Step_5 and the activation of the successive step Step_6. As soon as step Step_6 is active, transition E of the controller is examined. If transition E is true, step Step_6 is ended and step Step_7 is processed, etc.

**The alternative branch**
It is frequently necessary for a branch to be programmed into a control program, whereby the program may be continued in different ways at this point.

The alternative branch to a different path is represented by a corresponding number of transitions after the horizontal line. In the example in table B11.1e, the path via the step Step_4 is evolved, if transition E is true, or the path via step Step_5 evolved, if transition F is true and E false.

The corresponding counterpart to the alternative branch is a junction of alternative paths. In the case of a junction of alternative paths, transitions must always be positioned above the horizontal line.

The program flow in table B11.1f passes from step Step_6 to step Step_8, if transition G is true or from step Step_7 to step Step_8, if transition H is true. The decisive factor here is the path through which the control program reaches this junction of alternative paths. If this took place via the path step Step_6, the step enabling condition H is meaningless. Conversely, transition G is not evaluated, if the control program reached the junction via the path with Step_7.

It should be noted that in the case of alternative branching only ever one path is followed by the control program. It is therefore not mandatory for the conditions to be mutually exclusive.

If no other specifications exist, the path furthest to the left is evolved. Priority for the calculation of transitions is therefore given from left to right.

This is probably the most commonly implemented variant used by controller manufacturers to achieve alternative branches.

In conjunction with three steps, a section of a program or function block with triple alternative branch could therefore be as follows.

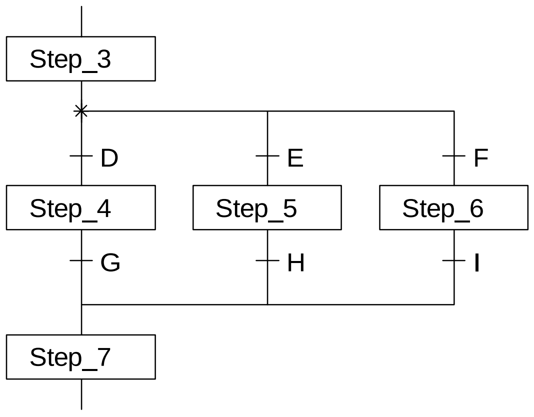


*Fig. B11.4:*
*Alternative branch:*
*Processing of transitions from left to right*

However, the IEC 1131-3 standard also offers the facility for the user to define the priority during the execution of the transitions. The definition of the functionality of alterntive branches in IEC 848, which requires a user programmed mutual exclusion of transition conditions, is also supported by IEC 1131-3 as a third method.

In contrast with the previous examples the numbers in the path in fig. B11.5 indicate a user-defined priority of the transition evaluation. The path with the lowest number has the highest priority.

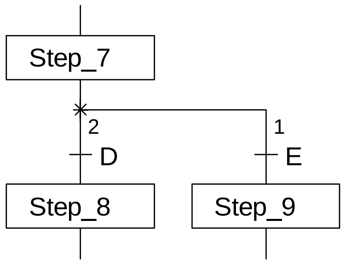


*Fig. B11.5: Alternative branch with user-defined priority*

As such, a transition is evolved from step Step_7 to step Step_9, if transition E is true, or a transition is evolved from step Step_7 to step Step_8, if transition D is true and transition E is false.

A loop structure may be regarded as a special case of alternative branching, whereby one or several paths return to a preceding step.

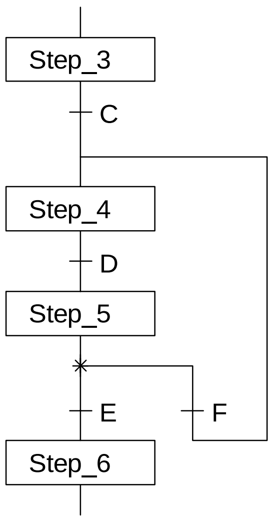


*Fig. B11.6: Representation of a loop*

In fig. B11.6 the program flow evolves from step Step_5 to step Step_4, if transition F is true and E is false. The evolution of step sequence Step_4, to Step_5 may be repeated in this way.

**Parallel branch**
A completely different functional element of the sequential function chart is parallel branching.

This is represented by a double line and a transition above this line (table B11.1g). As soon as transition B is fulfilled, an evolution from step Step_3 to step Step_4 and Step_5. These two steps are executed simultaneously.

A parallel branch determines that all connected paths are to be activated simultaneously and evolved independently of one another. In the case of the matching counterpart, and the joining of parallel paths, the transition is always represented underneath the horizontal double line.

Parallel joining contains a mechanism for synchronisation. Only when all the paths coming from above have been completely executed, is the subsequent transition evaluated. If it is true, the transition to the next step takes place. In table B11.1h this means: both steps Step_6 and Step_7 must be evolved prior to the evaluation of transition F.

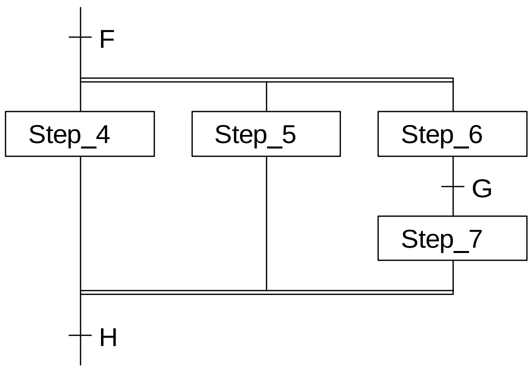


*Fig. B11.7: Representation of a triple parallel branch*

When step enabling condition F has been fulfilled, the three paths with steps Step_4, Step_5 and Step_6 and Step_7 must be evolved simultaneously.

Depending on the contents of transition G between the two steps Step_6 and Step_7, the control program may have to wait until transition G is fulfilled. The lower transition H is therefore only examined if the right path has been evolved completely. This can only be the case, if transition G in this path is true.

## 11.3 Transitions

Each transition is assigned a transition condition (step enabling condition). This provides the result of a boolean value.

In the simplest case, a step enabling condition can be specified by the interrogation of an input of the controller or another boolean variable. It is however also possible to program considerably more complex step enabling conditions.

**Formulation of Transition conditions**
Transition conditions can be programmed in the following languages

- Ladder diagram
- Function block diagram
- Instruction list
- Structured text

The contents of the transition condition are either specified directly at the transition (see fig. B11.8) or linked with the transition via a transition name (see fig. B11.9).

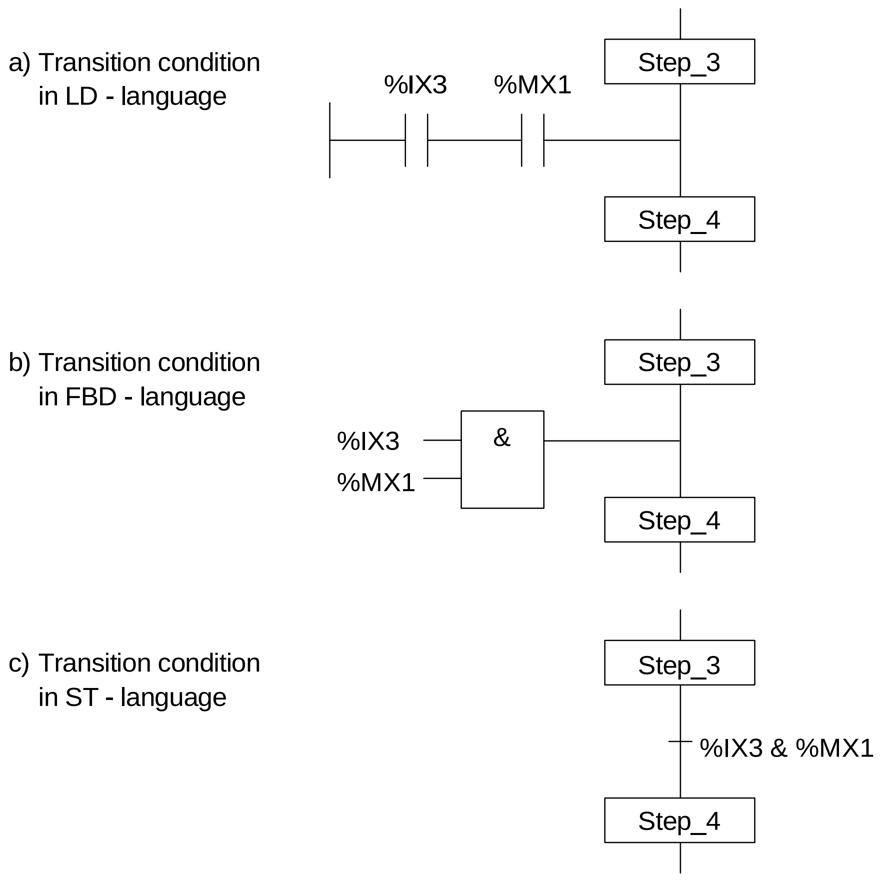

a) Transition condition in LD - language
%IX3
%MX1
Step_3
Step_4
b) Transition condition in FBD - language
%IX3
%MX1
&
Step_3
Step_4
c) Transition condition in ST - language
Step_3
%IX3 & %MX1
Step_4

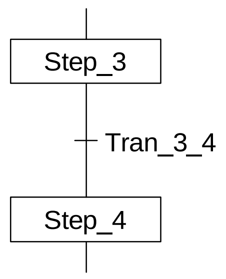
*Fig. B11.8: Direct specification of a transition condition*

Here, two results are connected via a logic AND function, whereby the transition condition will not be met until both input %IX3 and flag %MX1 assume the value 1.

The power or signal passes from left to right in the graphic languages LD and FBD, the LD or FBD network part is defined on the left, next to the transition symbol (horizontal line).

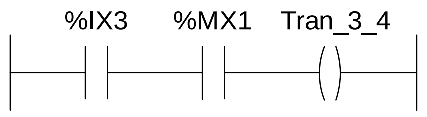
The boolean expression in ST languages is defined to the right of the transition symbol.

Transition name



a) Transition condition in LD - language

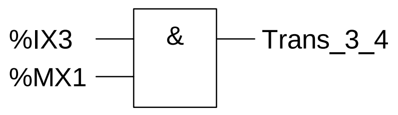
TRANSITION Tran_3_4:



END_TRANSITION

b) Transition condition in FBD - language

TRANSITION Tran_3_4:



END_TRANSITION

c) Transition condition in IL - language

```
TRANSITION Tran_3_4:
    LD    %IX3
    AND %MX1
END_TRANSITION
```

d) Transition condition in ST - language

```
TRANSITION Tran_3_4:
    : = %IX3 & %MX1;
END_TRANSITION
```

*Fig. B11.9:*
*Assignment of a transition condition to the transition by specifying a transition name*

If a transition name is used as an assignment mechanism from transition condition to transition, the transition name must refer to a TRANSITION...END_TRANSITION construct.

The transition condition is formulated within this construct and the boolean result mapped to the transition name.

The transition names within a program organisation unit, like the step names, must all differ from one another. A name is formulated according to IEC 1131-3 rules applicable to identifiers.

IEC 1131-3 also defines an equivalent textual representation for the graphic element Transition. The actual transition condition is programmed either in the IL or St language.

a) Transition condition formulated in ST language

```
STEP Step_3: END_STEP
TRANSITION FROM Step_3 TO Step_4
    := %IX3 & %MX1;
END_TRANSITION
STEP Step_4: END_STEP
```

b) Transition condition formulated in IL language

```
STEP Step_3: END_STEP
TRANSITION FROM Step_3 TO Step_4
    LD %IX3
    AND %MX1;
END_TRANSITION
STEP Step_4: END_STEP
```

*Fig. B11.10: Textual representation of transitions*

## 11.4 Steps

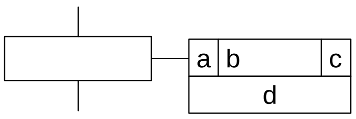
A step represents the execution part of a sequential function chart. Only within steps can a program or a function block within a controller influence the connected system via its outputs, by setting or resetting the outputs.

**Structure of a step within actions**
Each step may contain several actions. Each of these actions is to perform a job for the connected system. The structuring of a step into individual actions initially is merely an ordering function. This makes the step clearer, since it creates clearly defined limits between the individual job steps. However, since each action is assigned a qualifier, the structuring of a step into individual actions also define an additional functionality.

A step which does not contain any actions may be seen as a special case. Its sole purpose is to separate two step enabling conditions, which are to be evaluated consecutively. It thus permits a wait function, whereby the first step enabling condition has priority, irrespective of whether the second is already met or not, and the second step enabling condition must be met subsequently.

**Action blocks**
The graphic programming of steps is effected via individual action blocks. Each action is thereby connected with a particular characteristic.

An action block is represented in tabular format, which contains fixed positions for the specification of the action qualifier, the name of the action and the action content. In addition, a feedback variable may also be entered.



| | | |
|---|---|---|
| Field a: | Action qualifier: | |
| | N = non-stored | D = time delayed |
| | S = set, stored | DS = time delayed and stored |
| | R = reset | SD = stored and time delayed |
| | P = pulse (unique) | SL = stored and time limited |
| | L = time limited | |
| Field b: | Name of action | |
| Field c: | Feedback variable | |
| Field d: | Action content | |

*Fig. B11.11: Graphic representation of action block*

Again, the name b of an action represents a standard symbolic identifier, which acts purely as a means of differentiation and has no further significance.

Since a list of actions often forms part of a step, it may also be represented in conjuction with this.

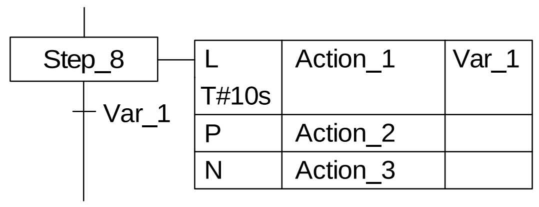


*Fig. B11.12:*
*List of action blocks*

The assignment of actions to a step in graphic form is effected by means of action blocks.

The assignment may however also be formulated textually. In the case of the example shown in fig. B11.12, this results in the following representation:

```
STEP Step_8
   Action_1( L, T#10s, Var_1 );
   Action_2( P );
   Action_3( N );
END_STEP
```

*Fig. B11.13:*
*Textual representation of a step with actions*

The contents of an action, i.e. the action itself, may be defined by means of several methods:

- Specification of a boolean variable
- Programming in instruction list
- Programming in structured text
- Ladder diagram
- Function block diagram
- Sequential function chart

The use of a boolean variable represents a simple and frequently used form of an action. In many cases, however, more complex actions containing a useful logic connection of different information will be required.

In the examples B11.14 to B11.16, the output %QX1.2 is set, if input %IX0.5 is set or if flags %MX1 and %MX3 are set. If neither is the case output %QX1.2 is reset.

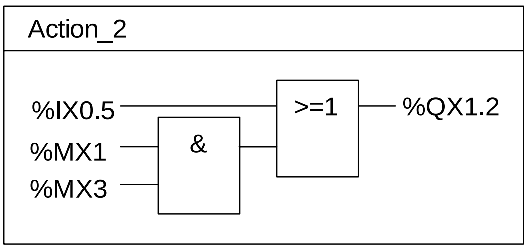


*Fig. B11.14:*
*Formulation of actions: graphic declaration in FBD*

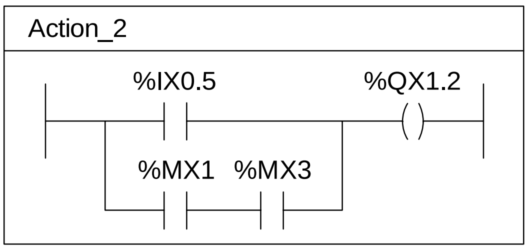


*Fig. B11.15:*
*Formulation of actions: graphic declaration in LD language*

| ***IL language*** | ***ST language*** |
|---|---|
| ACTION Action_2:<br>    LD %IX0.5<br>    OR( %MX1<br>    AND %MX3<br>    )<br>    ST %QX1.2<br>END_ACTION | ACTION Action_2:<br>    %QX1.2 := %IX0.5 OR (%MX1 AND %MX3);<br>END_ACTION |

*Fig. B11.16:*
*Formulation of actions: textual declaration*

Instead of an individual network of a statement sequence, several networks are also permissible within an action in textual languages. In this way, it is possible to incorporate a very wide range of actions in a step, and again a step itself may contain sequence structures (fig. B11.17).

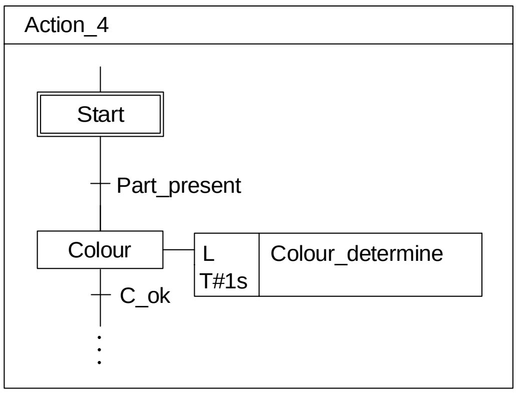


*Fig. B11.17: Formulation of actions: Inclusion of SFC elements in an action*

If individual fields of an action block are not required, such as for instance if a boolean variable is used as action content, a further simplification in the representation of an action block is permissible.

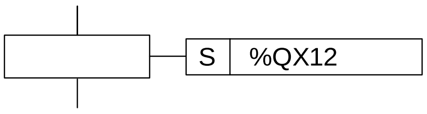


*Fig. B11.18: Short representation of an action block*

A feedback variable (c field) may be entered in each action block. Feedback variables are programmed within actions by the user and indicate their completion, timeout or error conditions. Fig. B11.19 indicates a frequently recurring application. Here the sequence of steps and transitions is structured in such a way that the action of a step sets the subsequent step enabling condition.

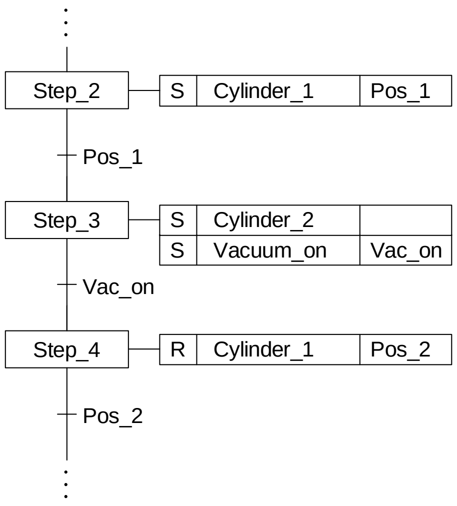


*Fig. B11.19: Use of feedback variables*

**Mode of action of action qualifiers**
The type of execution of the actions programmed by the user is defined by the associated action qualifier.

IEC 1131-3 defines the following action qualifiers

- **N** Non-stored
- **S** Set (stored)
- **R** Overriding reset
- **P** Pulse (unique)
- **L** Time limited
- **D** Time delayed
- **DS** Time delayed and stored
- **SD** Stored and time delayed
- **SL** Stored and time limited

Each action is the equivalent of exactly one of these qualifiers. In addition, the qualifiers L, D, DS, SD, SL have an associated duration of type Time, since these define a time behaviour of the action.

The qualifiers have a precisely defined significance. If a step is inactive, none of the action of the step is executed. With an active step, the following methods apply for the execution of an action qualifier.

**N** Non-stored
- the action is executed for as long as the step is active.

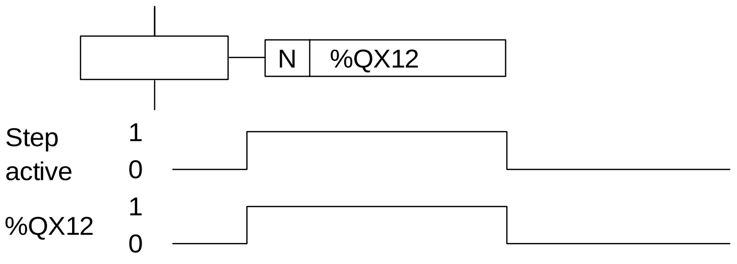


*Fig. B11.20: Non-stored action*

In the above example, the output %QX12 is set for as long as the step containing this action is active. On completion of the step, i.e. as soon as the subsequent enabling condition is met, the output is automatically reset.

**S** Set (stored)

- the execution of the action is executed permanently set (set stored).

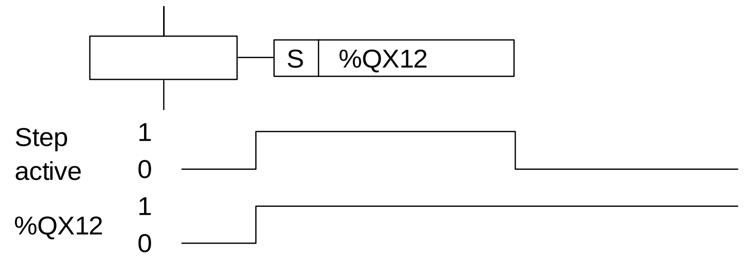


*Fig. B11.21: Set (stored) Action*

In this example, the output %QX12 is set for as long as the step containing this action is active. The output also remains set, when the subsequent step enabling condition is met and the step being considered is no longer active. The output can only be reset in another step, in another action, defined with qualifier R.

**R** Reset

- a previously set action (in another step) executed with the qualifier S, DS, SD, L or SL is cancelled.

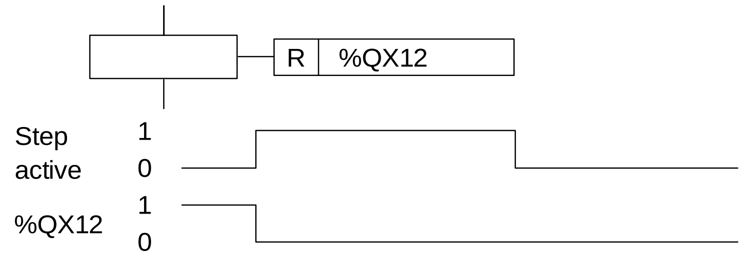


*Fig. B11.22: Reset action*

Output %QX12 has been set in another step in an action with one of the qualifiers S, DS, SD, L or SL and reset again via this action.

**P** Pulse (unique)

- unique execution of the action

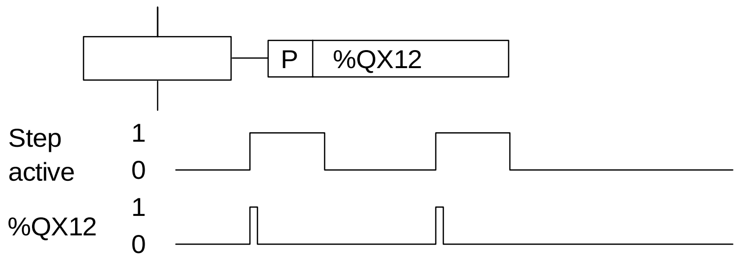


*Fig. B11.23:*
*Unique action*

During the initial execution of the action via the controller within the processing of the step, output %QX12 is set exactly once and then reset again. The output is reset uniquely again only after the exiting the step and a fresh re-entry into the step.

**L** Time limited

- The action is executed for a specific time.

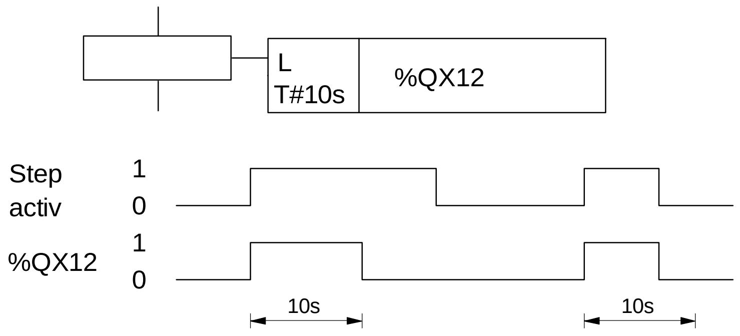


*Fig. B11.24:*
*Time limited action*

Output %QX12 is set for 10 seconds and subsequently reset again. This requires the step containing this action to be active for a period of at least 10 seconds. If the subsequent step enabling conditions are met prior to this time, the action time of the output is reduced also, since it is reset at the end of the step in any case.

**D** Time delayed

- The execution of the action is delayed until the end of the step.

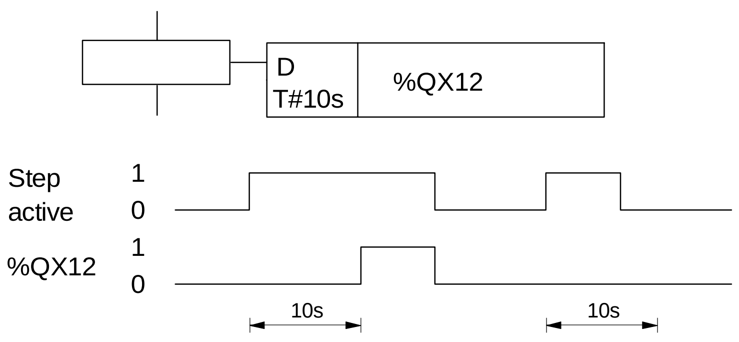


*Fig. B11.25:*
*Time delayed action*

Here, output %QX12 is not set until 10 seconds have expired and remains set until the step becomes inactive. If the duration during which the step is activated is less than 10 seconds, the output will not be set during the processing of this step.

**DS** Time delayed and stored

- The execution of the action is time-delayed and maintained beyond the end of the step.

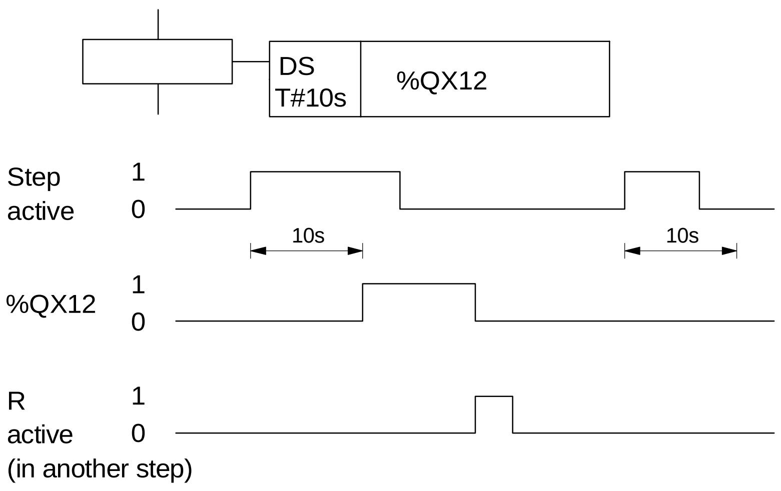


*Fig. B11.26:*
*Time delayed and stored action*

In this example too, output %QX12 is set after 10 seconds have expired. However, it remains set after completion of the step. It must be reset explicitely via another action with the qualifier R (in another step). If the duration of the step is not sufficiently long, in this case less than 10 seconds long, the output will never be set.

**SD** Stored and time-delayed

- the execution of the action is time delayed and is maintained beyond the end of the step

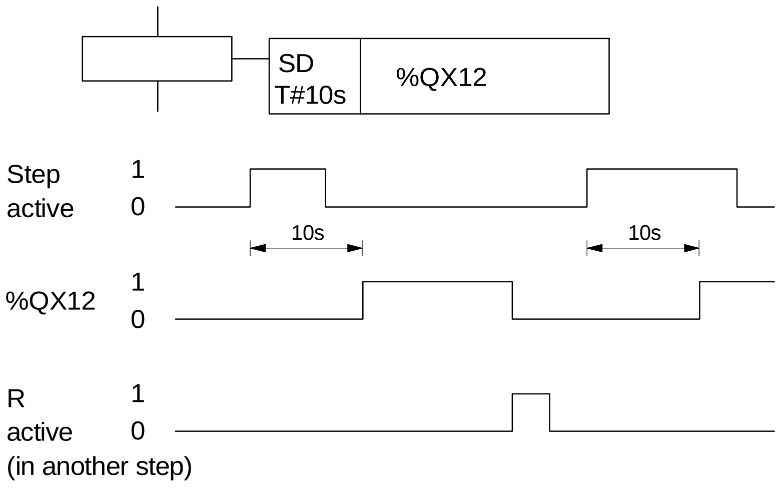


*Fig. B11.27: Stored and time delayed action*

Here too, output %QX12 is set after the expiry of 10 seconds. It remains set following the end of the step and can only be explicitly reset via another action with the qualifier R in another step. Unlike the mode of action of the DS qualifier, it is not necessary for the step to remain active beyond the duration of the delay for the output to be set.

**SL** Stored and time-limited

- the action is executed continuously for a specific period.

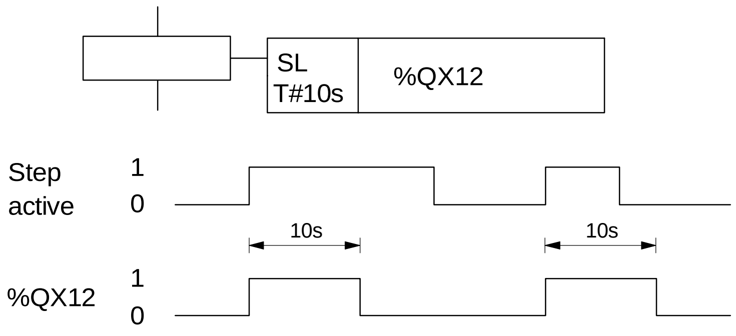


*Fig. B11.28: Stored and time delayed action*

The output is set for 10 seconds and then reset again. In contrast with the mode of action of the L qualifier, it is not necessary for the step to be active for a minimum of 10 seconds.

If the subsequent step enabling condition is met prior to this time expiring, i.e. if the step is active for less than 10 seconds, the active period of the output remains unaffected by this. The output can be reset at any time via another action with the qualifier R.

The mode of action of the individual qualifiers has been illustrated in the example of a boolean variable as an action. If more complex, i.e. non boolean actions are required, the type of execution is marginally different to the previously examined boolean variables. The networks are continually processed for as long as the step is active. As soon as the subsequent step enabling condition is met, however, the last, unique, execution of the networks is carried out once more.

This definition enables the targeted resetting of variables at the end of the processing of an action, when the N qualifier is used for more complex actions.

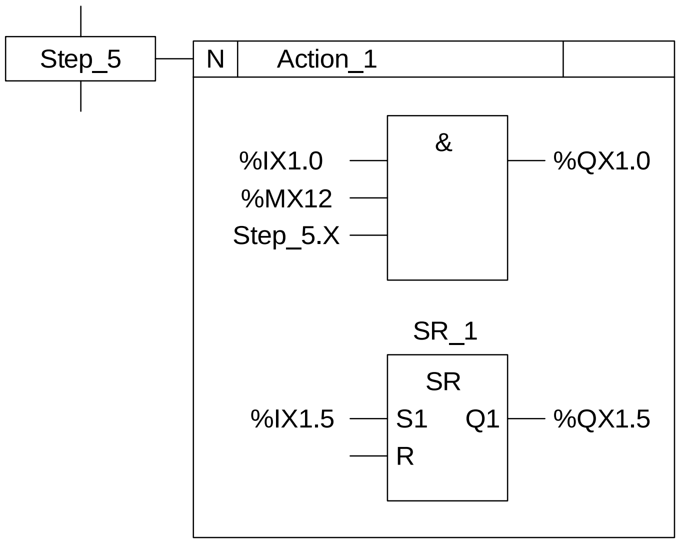


Fig. B11.29: Complex action in FBD language

If step Step_5 is deactivated, the last processing of the networks takes place with the value 0 for step flag Step_5.X. This results in output %QX1.0 being reset to 0 when the step is exited.

11.5 Example

**Problem description**
Components are transported together on a conveyor belt to a dual processing station. The drilling and countersinking units then move downwards as soon as a component is present. Two cylinders 1.0 and 2.0 are used to move the two machine tools. The conveying device is indexed by one working position via a third cylinder 3.0.

Two sensors B1 and B2 are provided to detect whether a workpiece is located underneath the drill or the countersink. The required drilling and countersink depths are sensed via two end position sensors B6 and B7. The initial positions of the transport cylinder, drill and countersink can be detected via the values of sensors B3, B4 and B5. Sensor B8 indicates an extended transport cylinder.

The system cannot always guarantee that a workpiece will be deposited underneath both the drill unit and the countersink after each transport movement. Processing should then be interrupted in the case of a missing workpiece. If both workpieces are missing at the same time, neither of the two tools should be lowered.

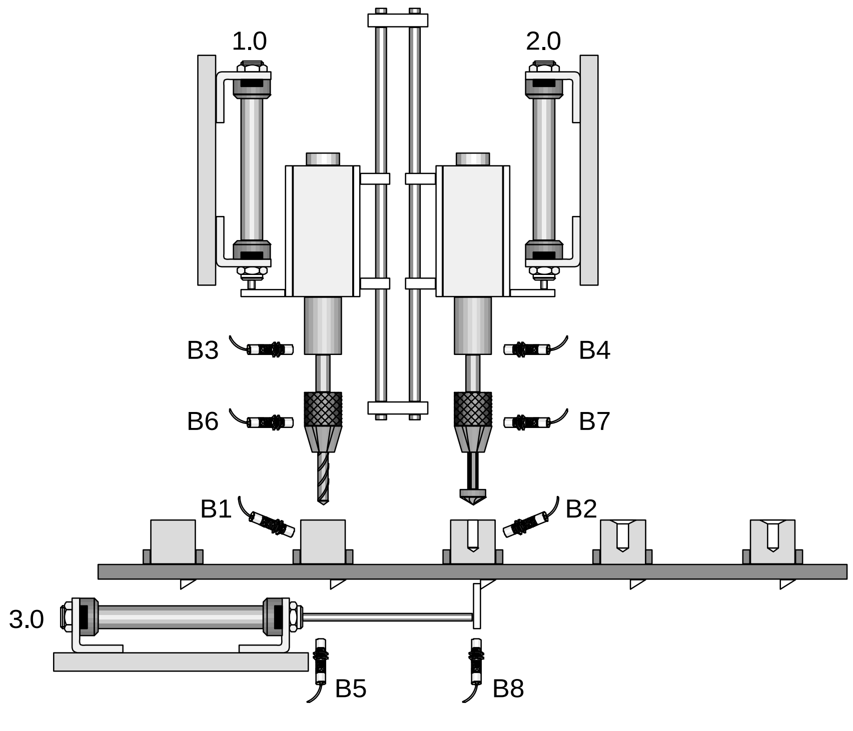


Fig. B11.30:
Positional sketch

## Allocation list

| *Equipment designation* | *PLC input/ PLC output* | *Task* |
|---|---|---|
| B1 | %IX0.1 | Detecting workpiece under the drill |
| B2 | %IX0.2 | Detecting workpiece under countersink |
| B3 | %IX0.3 | Initial position of drill unit (up) |
| B4 | %IX0.4 | Initial position of countersink (up) |
| B5 | %IX0.5 | Initial position of transport cylinder |
| B6 | %IX0.6 | Lower end position of drill unit reached |
| B7 | %IX0.7 | Lower end position of countersink reached |
| B8 | %IX0.8 | Transport cylinder extended |
| Y1 | %QX0.1 | Lower drill unit |
| Y2 | %QX0.2 | Lower countersink |
| Y3 | %QX0.3 | Transport feed |

*Table B11.4: Allocation list*

**Problem**

A control program is to be designed for this task. The solution is to achieve a configuration by means of a sequential function chart. The conditions and actions are then to be applied to the steps and transitions. The program is to be executed cyclically.

To simplify matters, you may assume that there is no need to use timers to compensate for positioning tolerances.

**Solution**

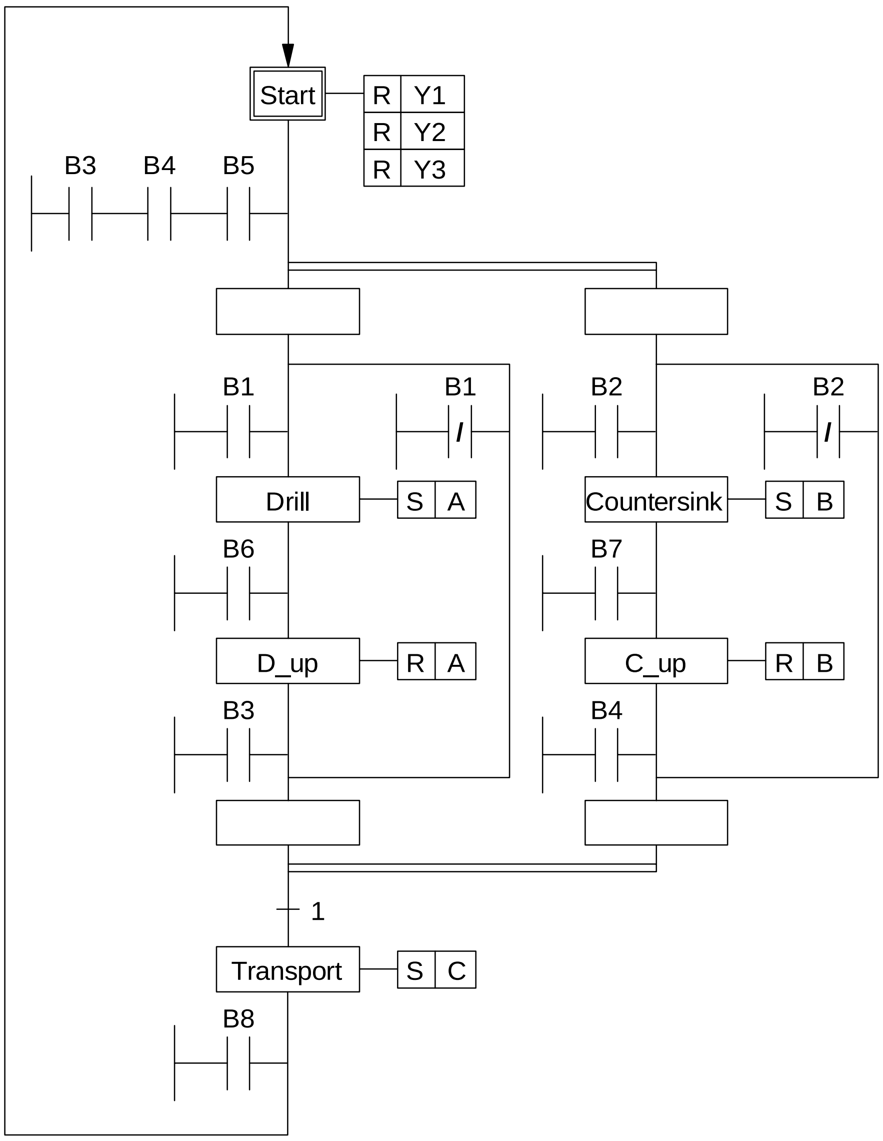


*Fig. B11.31: Program in sequence language*

All cylinders are brought into their initial position in one initial step. At the end of the program, this step is also used to retract the cylinder extended during the last step for the transport device.

When all the cylinders are in their initial position, a parallel branch with two sequences is started for drilling and countersinking. Both sequences in this example contain the same tasks, but use different tools. The lefthand sequence lowers the drill unit and lifts it again, and the righthand sequence evolves identically for the countersinking. The sequences merely differ in their use of sensors and actuators. A void step is incorporated at the top and bottom of both sequences to maintain the necessary sequence steps and transitions.

The program for the drilling unit evolves as follows. It detects whether a workpiece is available via the value of sensor B1. If this value is equal to 1, the workpiece is in the required position and the drilling process starts. Otherwise the entire drilling process is bypassed in an alternative path. Drilling of the hole starts with the lowering of the drill by setting Y1. When the lower end position is reached, i.e. drilling of the hole has been completed, sensor B6 signals the end of drilling. In the next step, the drill is returned to its normal position at the top. This part of alternative branching ends when the drill has reached the top. The program follows the same procedure for countersinking.

When both parallel sequences have been completed, a transition takes place in the program to the transport step. The necessary synchronisation – i.e. drilling and countersinking ready – is ensured by the sequential function chart and therefore does not require special treatment. A true step enabling condition is always inserted at this point in order for the steps and transitions to alternate.

In the last step Transport, the cylinder of the transport device is extended and the awaited completed action in the next transition condition. Thereafter, the whole process starts anew.

# Chapter 12

# Logic control systems

## *12.1 What is a logic control system?*

Logic control systems are controllers programmed through the use of boolean operations. All logic operations are processed and executed during a machining cycle.

Control tasks realised typically in the form of logic control, are characterised by the fact that no time duration is given within the process, but all or most of the conditions of the control program are examined simultaneously.

Examples of logic control systems can therefore be found in PLC applications, where safety aspects are of importance. The monitoring of certain tasks is often required to be permanent and independent of the time-related execution of the process. These requirements apply for instance in:

- Protective circuits: a device may only load, if all protective devices are switched on
- Safety interlocking

## *12.2 Logic control systems without latching properties*

Logic control systems without latching properties may be described by means of a combination of boolean operations, whereby the output signals of a controller are determined by a combination of input signals at a given time.

The basic logic operations AND, OR and NOT may be used to create any complex logic operations – and as such also control systems.

A number of boolean algebra methods such as function tables, boolean equations and disjunctive normal form (DNF) are used to describe the problem and to find a solution. The importance of these methods is demonstrated amongst other things in the more complex applications for logic control systems. The actual programming of a logic control system is preferably in the languages ladder diagram or function block diagram.

**Typical boolean operations**
The following represents basic control technology tasks such as boolean operations realised via PLC.

The solutions are represented in the languages LD, FBD, IL and ST. The solution sections are preceded by a declaration of the necessary PLC inputs and outputs. In addition, the description options of a function table and boolean equation are also listed.

**Negation:**
The output signal assumes the value 1, if the input signal has the value 0 and vice versa.

*Example*

Lamp H1 is illuminated as long as switch S1 is not actuated; it is extinguished, if the switch is closed. The purpose of S1 is therefore to switch off the lamp.

Function table

| S1 | H1 |
|---|---|
| 0 | 1 |
| 1 | 0 |

Boolean equation

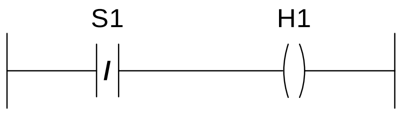
$$H1 = \overline{S1}$$

*Fig. B12.1: Description methods*

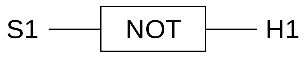
```
VAR
    S1 AT %I2.5      : BOOL;
    H1 AT %Q1.4      : BOOL;
END_VAR
```

*Fig. B12.2: Declaration of variables*

a) LD



b) FBD



c) IL

```
LDN   S1
ST    H1
```

d) ST

```
H1  : =  NOT S1;
```

*Fig. B12.3: Negation*

**AND-operation:**
The output signal only assumes the value 1, if all input signals have the value 1.

*Example* Lamp H1 is to be switched on only if the two switches S1 ad S2 are actuated.

Function table

| S1 | S2 | H1 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Boolean equation

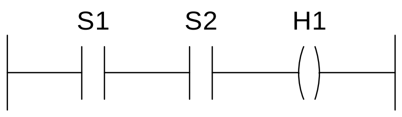
$H1 = S1 \wedge S2$

*Fig. B12.4: Description methods*

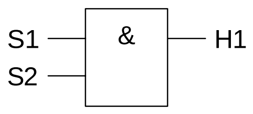
```
VAR
    S1 AT %I2.5       : BOOL;
    S2 AT %I2.6       : BOOL;
    H1 AT %Q1.4       : BOOL;
END_VAR
```

*Fig. B12.5: Declaration of variables*

a) LD



b) FBD



c) IL

```
LD     S1
AND    S2
ST     H1
```

d) ST

```
H1  : =  S1 AND S2;
```

*Fig. B12.6:*
*AND operation*

**OR-operation:**
The output signal assumes the value 1, if at least one input signal has the value 1.

*Example*

Lamp H1 is to be switched on, if at least one switch, S1 or S2 is actuated.

Function table

| S1 | S2 | H1 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Boolean equation

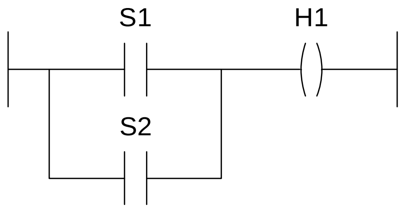
$H1 = S1 \vee S2$

*Fig. B12.7:*
*Description methods*

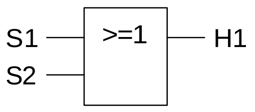
```
VAR
   S1 AT %I2.5      : BOOL;
   S2 AT %I2.6      : BOOL;
   H1 AT %Q1.4      : BOOL;
END_VAR
```

*Fig. B12.8:*
*Declaration of variables*

a) LD



b) FBD



c) IL

```
LD    S1
OR    S2
ST    H1
```

d) ST

```
H1  : =  S1 OR S2;
```

*Fig. B12.9: OR operation*

**Combined logic operations**

*Example* Lamp H1 is to be illuminated only, if precisely two of the three switches S1, S2, S3 are actuated.

The first to be created is the function table, whereby those combinations are selected, which provide the result 1. These are lines 4, 6 and 7. The boolean equation and thus the solution can be created on the basis of this combination. The conversion of the solution into the individual programming languages is contained in fig. B12.12.

Function table

| S1 | S2 | S3 | H1 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Boolean equation

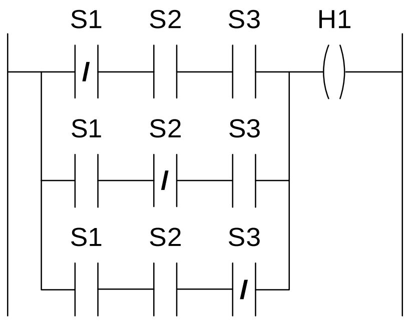
$$
\begin{aligned}
H1 = {} & (\overline{S1} \wedge S2 \wedge S3) \\
& \vee (S1 \wedge \overline{S2} \wedge S3) \\
& \vee (S1 \wedge S2 \wedge \overline{S3})
\end{aligned}
$$

*Fig. B12.10: Description methods*

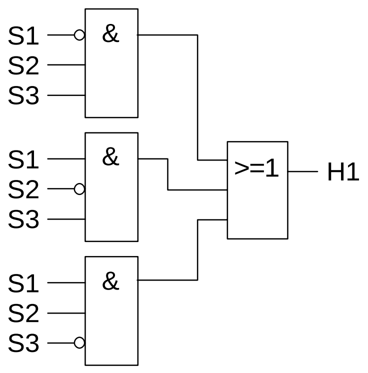
```
VAR
    S1 AT %I2.5      : BOOL;
    S2 AT %I2.6      : BOOL;
    S3 AT %I2.7      : BOOL;
    H1 AT %Q1.4      : BOOL;
END_VAR
```

*Fig. B12.11:*
*Declaration of variables*

a) LD



b) FBD



c) IL

```
LD (      S3
AND       S2
ANDN      S1
)
OR (      S1
ANDN      S2
AND       S3
)
OR (      S1
AND       S2
ANDN      S3
)
ST        H1
```

d) ST

```
H1  : =  (NOT S1 AND S2 AND S3)
         OR (S1 AND NOT S2 AND S3)
         OR (S1 AND S2 AND NOT S3);
```

*Fig. B12.12:*
*Combination of boolean operations*

## 12.3 Logic control systems with memory function

Many PLC applications require elementary memory operations. A memory function constitutes the retention, i.e. storage of a briefly occurring signal status. At a given instant, the output signals are dependent not only on the combination of input signals, but also on "previous statuses".. .

The example given here is that of a switch for switching on and off a lamp.

IEC 1131-3 defines two function blocks for the realisation of memory functions. These are function block SR (primarily setting) and RS (primarily resetting). A description of the blocks follows below.

### Function block SR

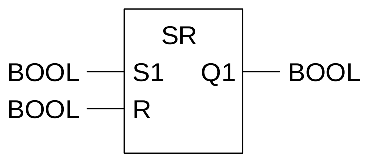


*Fig. B12.13: Function block SR, primarily setting*

The standard function block SR contains a dominant setting flipflop (bistable memory with preferred status 1). A 1-signal at the setting input S1 sets the flipflop, i. e. the value of Q1 becomes 1. The value which applies at reset input R is immaterial. A 1-signal at reset input R only brings output Q1 to value 0, if set input S1 is also 0. The set input with this flipflop is therefore dominant.

**Function block RS**

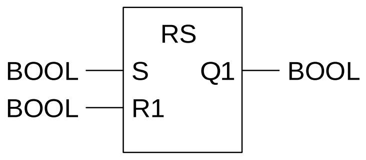


*Fig. B12.14: Function block RS, primarily resetting*

The standard function block RS contains a dominant resetting flipflop (bistable memory with preferred status 0). A 1-signal at the reset input R1 resets the flipflop, i.e. the value of Q1 becomes 0. The value which applies at the setting input S is immaterial.

The following example illustrates the use of elementary memory operations.

*Example*

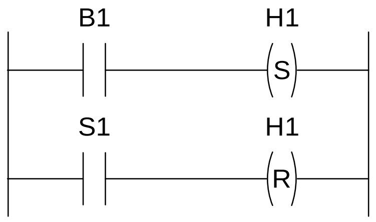
If sensor B1 has a 1-Signal, this indicates an error status in the system. A horn H1 is sounded. The horn can only be switched off by actuating push-button S1. It is possible to switch off the horn, even if the B1-signal continues to be applied.

```
VAR
    B1 AT %IX1      : BOOL;   (* Sensor detects error status              *)
    S1 AT %IX2      : BOOL;   (* Push button                              *)
    H1 AT %QX1      : BOOL;   (* Horn                                     *)
    RS_H1           : RS;     (* Flip-flop named RS_H1 for status         *)
                              (* of horn                                  *)
END_VAR
```

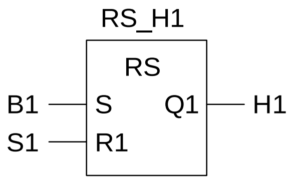
*Fig. B12.15: Declaration of variables*

In the languages FBD and ST, memory operations are realised by invoking a copy of the RS function block. The copy in this example has the name RS_H1. The invocation in FBD is effected by means of grafically linking the current transfer parameters with the inputs of the function block copy. Since the value of the function block copy is relevant, the output of the function block copy is connected correspondingly.

a) LD



b) FBD



c) IL

```
LD      B1
S       H1
LD      S1
R       H1

or

CAL     RS_H1 (S := B1, R1 := S1)
LD      RS_H1.Q1
ST      H1
```

d) ST

```
RS_H1 (S := B1, R1 := S1);
H1 := RS_H1.Q1;
```

*Fig. B12.16: Use of function block RS*

In the textual language ST, the invocation is effected by means of specifying the function block copy. The current parameters are also listed simultaneously. The value of the output of the function block copy RS_H1 can be accessed via the variable RS_H1.Q1; the name of the output variable is therefore defined via the names of the function block copy and the names of the desired output.

The languages LD and IL have their own operations for stored setting or resetting of variables, whereby the use of an RS function block can be omitted. It should be noted that the sequence of set and reset commands is crucial for the behaviour of the PLC. The command, which is to be dominant – in this instance the reset command – must only occur after the set command in the program, so that it is the last command to be executed and thereby determines the behaviour – in this case the output.

## *12.4 Edge evaluation*

Signals reaching the inputs via sensors are evaluated as 0 or 1 signals by the central control unit of the PLC, whereby the duration of signal statuses 0 and 1 is determined by the sensor.
For instance: A 1-signal applies for as long as a push button is pressed, otherwise a 0-signal is received.

In many cases, however, it is not the signal itself which matters, but the exact instant, during which the signal changes. This type of signal change is termed **Edge**.

*Example*

To elucidate this, imagine the switches (push buttons) of a lighting system, where the edge evaluation is mechanically implemented. By actuating the push button, the light comes on (irrespective of how long this push button is pressed). If the push button has been released in the meantime, the light may be switched of by pressing the push button again.

In exactly the same way, the moment in which the input signal changes from 0 to 1 must be registered in a PLC, since only ever one single reaction per push button actuation may be triggered (196 irrespective of how long the 1-signal applies. This prevents a process from being put in motion repeatedly by the controller, should a push button be actuated for too long. The edges of the input signal are evaluated for each program.

*Rising and falling edges*

In this context it is referred to as **edge detection**. Each binary signal has a rising and a falling edge:

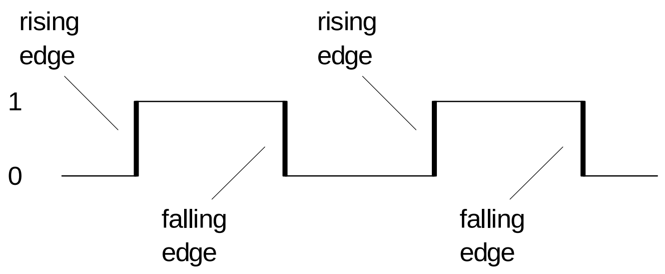


*Fig. B12.17:
Rising and falling edges*

**Rising edges** mark the instants, in which a signal level changes from 0 to 1.

**Falling edges** mark the exact instants, when a signal level changes from 1 to 0.

Whether rising or falling edges are evaluated within a program or function block depends on how the respective sensor is wired (normally closed/normally open contact) and how it is used.

A push button (normally open contact), for instance, creates a rising edge the moment it is pressed and a falling edge the moment it is released.

IEC 1131-3 provides two standard function blocks for the evaluation of edges.

**Function block R_TRIG, rising edge**
The standard function block R_TRIG (rising) is used for the detection of rising or positive edges. Its output Q has the value 1 from one execution of the function block to the next, if its input CLK (Clock for pulse) changes from 0 to 1.

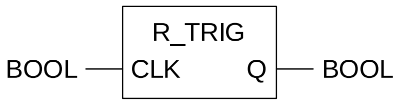


*Fig. B12.18: Function block R_TRIG, rising edge*

**Function block F_TRIG, falling edge**
A falling or negative switching edge is detected by means of the function block F_TRIG (falling). If a change has taken place at input CLK from 1 to 0, output Q assumes the value 1 for one processing cycle.

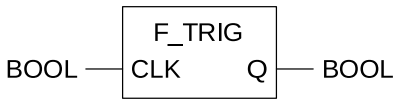


*Fig. B12.19: Function block F_TRIG, falling edge*

The following example shows the programming of the edge evaluation in the languages FBD, LD, IL and ST, whereby the rising edges are evaluated.

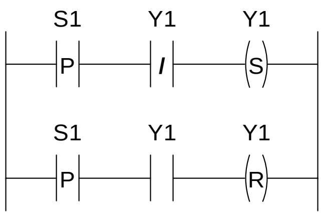
*Example*

Actuation of a push button S1 causes the door of a furnace to be opened. A repeat actuation of push button S1 causes the door to be closed.

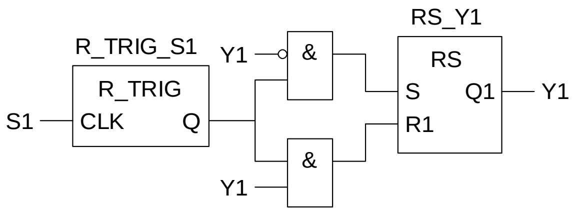
```
VAR
    S1 AT %IX1      : BOOL;  (* Switch for door                    *)
    H1 AT %QX1      : BOOL;  (* Coil for actuation of cylinder     *)
                             (* for door                           *)
    RS_Y1           : RS;    (* Flip-flop named RS_Y1 for status   *)
                             (* of coil                            *)
    R_TRIG_S1       : R_TRIG;(* Function block named R_TRIG_S1     *)
                             (* for detection of edge at S1        *)
END_VAR
```

*Fig. B12.20:*
*Declaration of variables*

a) LD



b) FBD



c) IL

```
CAL     R_TRIG_S1 (CLK := S1)
LD      R_TRIG_S1.Q
ANDN    Y1
S       Y1
LD      R_TRIG_S1.Q
AND     Y1
R       Y1
```

d) ST

```
R_TRIG_S1 (CLK := S1);
RS_Y1 ( S := R_TRIG_S1.Q  &  NOT Y1,
          R1 := R_TRIG_S1.Q  & Y1);
Y1 := RS_Y1.Q1;
```

*Fig. B12.21:*
*Use of function block R_TRIG*

In the languages FBD, IL and ST, edge detection is effected by means of invoking a R_TRIG function block. The name of the function block used in the example is R_TRIG_S1; R_TRIG_S1 represents a copy of the function block type R_TRIG.

The LD language has special contacts for the evaluation of edges, whereby the invocation of an R_TRIG function block is omitted.

# Chapter 13

# Timers

## *13.1 Introduction*

Many control tasks require the programming of time. For example, cylinder 2.0 is to extend, if cylinder 1.0 is retracted – but only after a delay of 5 seconds. This is known as a switch-on signal delay. Switch-on signal delays during the switching on of power sections is often also required for reasons of safety.

The timers of a PLC are realised in the form of software modules and are based on the generation of digital timing. The counted clock pulses are derived from the quartz generator of the microprocessor. The desired time duration is set in the control program.

IEC 1131-3 defines three types of timer function blocks:

- TP Pulse timing
- TON On-delay timing
- TOF Off-delay timing

Time duration is specified by means of a defined character format. A time specification is introduced by the characters T# or t#, followed by the time elements, i.e. days, hours, minutes, seconds, milliseconds.

The following represent examples of permissible time specifications:

| | |
|---|---|
| d | Day |
| h | Hour |
| m | Minute |
| s | Second |
| ms | Millisecond |

Details regarding time specifications may be found in chapter 6.2.

| |
|---|
| T#2h15m |
| t20s |
| T#10M25S |
| t#3h_40m_20s |

## 13.2 Pulse timer

The function block TP (timer pulse) is a  pulse timer, which is started by a shorter or longer 1-signal at input IN. A 1-signal now applies at output Q for the time specified at its input PT (preset time). The output signal Q therefore has a fixed duration, which can be specified by means of a time specification. It cannot be started again while the pulse timer is active. The current time value of the pulse timer is available at output ET (estimated time).

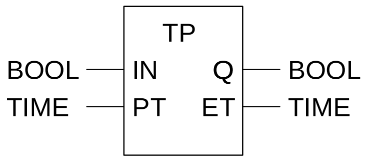


*Fig. B13.2: Timing diagram of pulse timer TP*

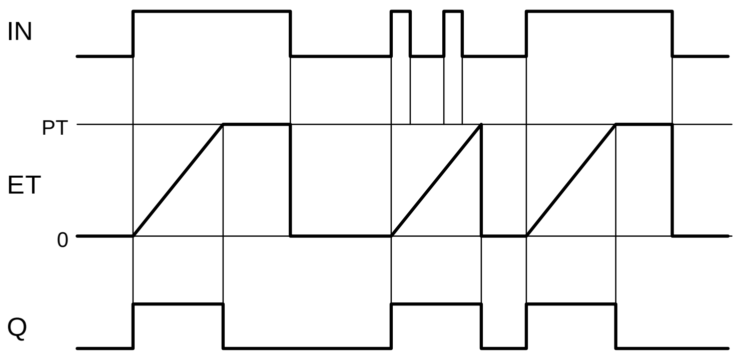


*Fig. B13.1: Function block TP, Pulse timer*

The use of a pulse timer is represented with the help of an example.

*Example*

Pressing of the start button S2 is to cause the piston of a cylinder to advance. This mechanism is to be used to clamp workpieces. When the piston has advanced fully, it is to remain in this position for 20 seconds. The cylinder then returns to its initial position.

```
VAR
    S2 AT %IX1      : BOOL;   (* Start button                        *)
    B1 AT %IX2      : BOOL;   (* Cylinder retracted                  *)
    B2 AT %IX3      : BOOL;   (* Cylinder advanced                   *)
    Y1 AT %QX1      : BOOL;   (* Advance cylinder                      *)
    SR_Y1           : SR;     (* Flip-flop named SR_Y1 for status    *)
                              (* of Y1                               *)
    TP_Y1           : TP;     (* TP function block named TP_Y1       *)
END_VAR
```

*Fig. B13.3: Declaration of variables*

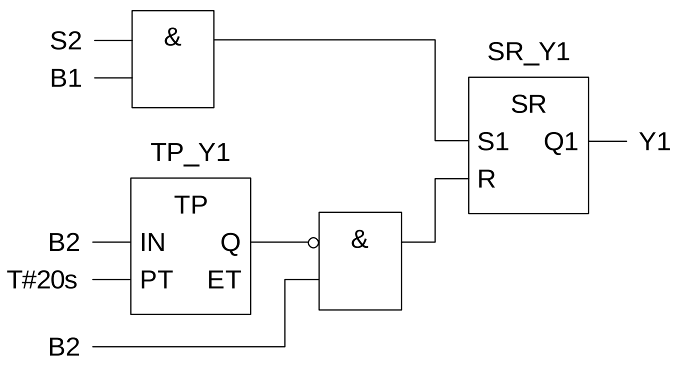


*Fig. B13.4: Use of pulse timer in FBD*

The control task has been programmed in the language FBD as an example. A timer function block may of course be used in any of the other languages. An example using a switch-off delay is given in chapter 13.4 to demonstrate this for the languages FBD, LD, IL and ST.

The valve Y1 for the actuation of the cylinder is switched via an SR flipflop SR_Y1. The set condition for SR_Y1 is met, if the start button for the retracted cylinder is actuated. As soon as the cylinder has advanced, the pulse timer TP_Y1 with the time of 20 seconds is started by the rising edge of sensor B2. Output Q of TP_Y1 now assumes a 1-signal. When the pulse timer has expired – the 20 seconds have passed – 0 applies  at output Q of TP_Y1. The reset condition for SR_Y1 is fulfilled: the cylinder retracts again.

**Note:** Formulations such as "pulse timer" with the name TP_Y1" mean: TP_Y1 is a copy of function block type TP, in this case a copy of the pulse timer.

## *13.3 Switch-on signal delay*

The function block TON (timer on-delay) is used to generate switch-on signal delays. After the start via a 1-signal at input IN, output Q does not assume value 1 until the time specified at input PT has expired, and retains this until input signal IN returns to 0. If the duration of the input signal IN is shorter than the specified time PT, the value of the output remains at 0.

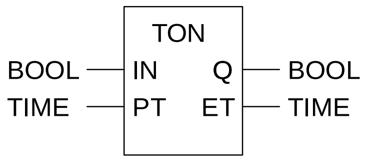


*Fig. B13.5:*
*Function block TON,*
*Switch-on signal delay*

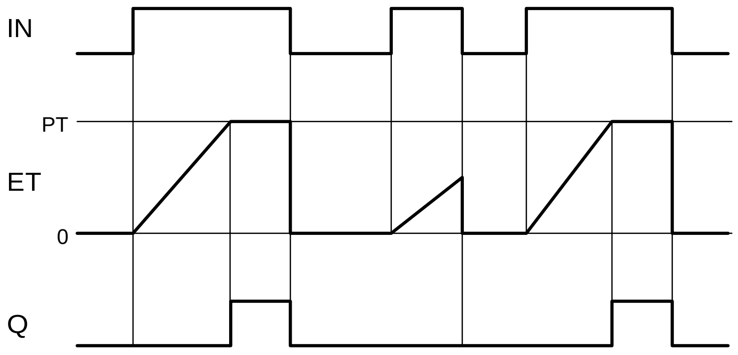


*Fig. B13.6:*
*Timing diagram of switch-on signal delay TON*

*Example*

Cylinder 1.0 extends, if start button S1 is actuated. Once this has been extended for 2 seconds, a second cylinder 2.0 moves to its forward end position. Sensors B1 and B2 indicate the retracted and the forward end positions of cylinder 1.0.

```
VAR
    S1 AT %IX1      : BOOL;   (* Start button                    *)
    B1 AT %IX2      : BOOL;   (* Cylinder 1.0 retracted          *)
    B2 AT %IX3      : BOOL;   (* Cylinder 1.0 advanced           *)
    Y1 AT %QX1      : BOOL;   (* Cylinder 1.0 advance            *)
    Y2 AT %QX2      : BOOL;   (* Cylinder 2.0 advance            *)
    RS_Y1           : RS;     (* Flip-flop named RS_Y1 for Y1    *)
    TON_Y2          : TON;    (* Switch-on signal delay named    *)
                              (* TON_Y2 for Y2                   *)
END_VAR
```

*Fig. B13.7: Declaration of variables*

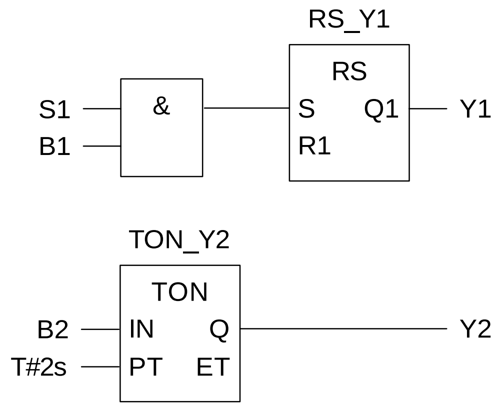


*Fig. B13.8: Use of switch-on signal delay in FBD*

Cylinder 1.0 is controlled via valve Y1. As soon as cylinder 1.0 has extended and sensor B2 has a 1-signal, the switch-on signal delay TON_Y2 is started. On expiry of 2 seconds, a 1-signal is applied at output Q of TON_Y2, and cylinder 2.0 extends. Cylinder 2.0 remains extended so long as the 1-signal is applied at input IN of TON_Y2, i. e. so long as cylinder 1.0 remains extended.

As illustrated by this example, not all inputs and outputs of a function block need be connected or supplied.

If an input of a function block is not connected – in this case the R1 input of RS_Y1 – the value of the input from the previous invocation is used. In this case, the initialisation value of the variable R1, which represents a boolean variable, is therefore preallocated with the value 0, i.e. function block RS_Y1 operates with the value 0 for parameter R1 during its invocation.

## 13.4 Switch-off signal delay

TOF (timer off-delay) is the name of the function block for a Switch-off signal delay. The timer is started via a 1-signal at input IN. At the same time, the output signal Q assumes the value 1. After the input signal IN has reverted to the value 0, the output remains at 1 for the duration PT and does not return to the value 0 until this has expired.

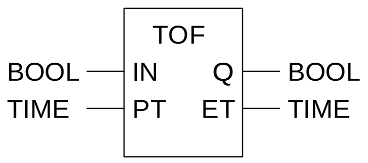


*Fig. B13.9: Function block TOF, switch-off signal delay*

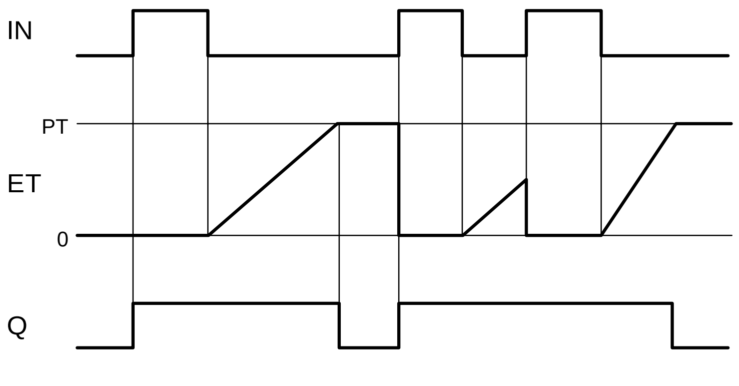


*Fig. B13.10: Timing diagram of switch-off signal delay TOF*

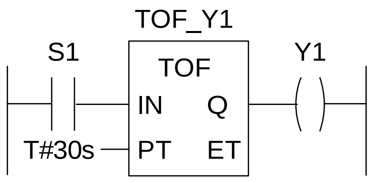
The following example illustrates the use of a switch-off signal delay in the languages FBD, LD, IL and ST.

*Example* Following the actuation of a push button, the cylinder of a stamping device is to extend momentarily. When the push button is released, the cylinder is to retract only after a stamping period of 30 seconds.

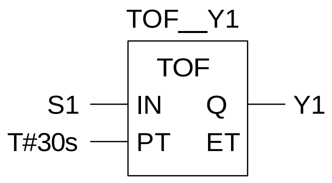
```
VAR
    S1 AT %IX1      : BOOL;   (* Switch                             *)
    Y1 AT %QX1      : BOOL;   (* Advance cylinder                   *)
    TOF_Y1          : TOF;    (* Switch-off signal delay named      *)
                              (* TOF_Y1 for Y1                      *)
END_VAR
```

*Fig. B13.11: Declaration of variables*

a) LD



b) FBD



c) IL

```
CAL    TOF_Y1 (IN := S1, PT := T#30s)
LD     TOF_Y1.Q
ST     Y1
```

d) ST

```
TOF_Y1 (IN :=S1, PT := T#30s);
Y1 := TOF_Y1.Q;
```

*Fig. B13.12: Use of switch-on signal delay in FBD*

In all the languages, a copy of the TOF function block TOF_Y1 is invoked to realise the switch-off signal delay of the stamping cylinder.

In the LD language, the function block is attached in a current rung via the boolean start input IN and the boolean output Q. If the normally open contact S1 supplies a 1-signal, a 1-signal also applies at output Q of TOF_Y1. The value Q is copied to the variable Y1. As soon as the 1-signal of S1 returns to 0, 1 still applies at output Q of TOF_Y1 for the period of 30 seconds, a 0-signal is supplied subsequently.

In the textual languages IL and ST, the switch-off signal delay is invoked by specifying the name TOF_Y1 of the declared copy and listing the relevant transfer parameters. The status of the switch-off signal delay can be obtained via output Q. In the example given here, the status of the switch-off signal delay TOF-Y1 is stored in the variable TOF_Y1.Q.

# Chapter 14

# Counters

## 14.1 Counter functions

Counters are used to detect piece numbers and events. Controllers frequently need to operate with counters in practice. A counter is for instance required, if exactly 10 identical parts are to be conveyed to a conveyor belt via a sorting device.

IEC 1131-3 differentiates between three different counter modules:

- CTU: Incremental counter
- CTD: Decremental counter
- CTUD: Incremental/Decremental counter

These standard function modules are used to detect standard, non time-critical counting.

With many control tasks it is however necessary to use so-called high-speed counters. "High-speed" in this case generally refers to a counter frequency in excess of 50 Hz, i. e. more than 50 events are counted per second. Tasks of this type cannot be solved with the standard counter function modules of a PLC.

The limitations of counter frequency in counter function blocks are due to the output signal delays. Each input signal – i.e. also the counter signal – is delayed by a certain time, before it is released for processing in the PLC. This prevents interference. A further limitation is the cycle time of the PLC.

This is why additional counter modules are generally available for PLCs for high-speed counting. High-speed counters are for instance used for the positioning of workpieces.

## 14.2 Incremental counter

The Incremental counter is known as a CTU (count up). The counter is set at the initial value 0 by a signal at reset input R.

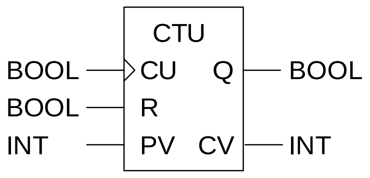


Fig. B14.1: Function block CTU, incremental counter

This current counter status is available at output CV (current value). The value in the counter is then increased by 1 with each positive edge at counter input CU (count up). At the same time the current value is compared in the function block with the preselect value PV. As soon as the current value CV is equal to or greater than the preselect value, the output signal assumes the value 1. Prior to reaching this value, output Q has a 0-signal.

The following example demonstrates the use of an incremental counter in the languages FBD, LD, IL and ST.

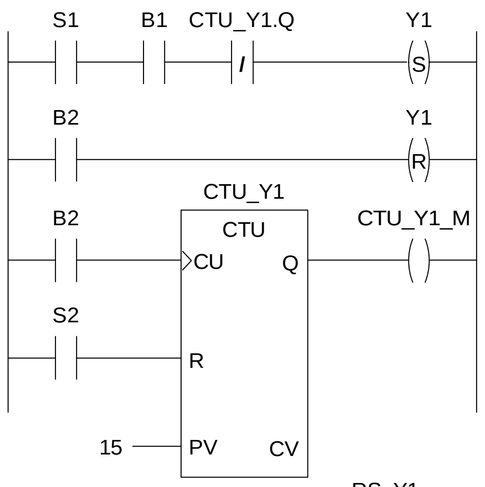
*Example*

Parts are to be ejected from a gravity-feed magazine via a cylinder. IF push button S1 is actuated, the cylinder is to advance, eject a work-piece and then retract again. 15 parts are to be ejected in this way. When 15 parts have been ejected, it should no longer be possible to trigger a cylinder movement via push button S1. First the counter must be reset by actuating push button S2.

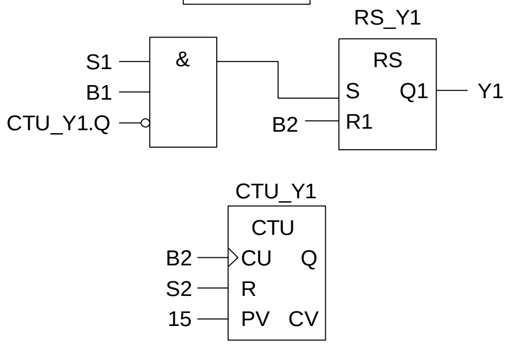
```
VAR
    S1 AT %IX1       : BOOL;   (* Push button for cylinder movement            *)
    S2 AT %IX2       : BOOL;   (* Push button for resetting of counter CTU_Y1 *)
    B1 AT %IX3       : BOOL;   (* Cylinder retracted                           *)
    B2 AT %IX4       : BOOL;   (* Cylinder advanced                            *)
    Y1 AT %QX1       : BOOL;   (* Advance cylinder                              *)
    Y1_advance
         AT %MX1     : BOOL;   (* stored condition: Advance cylinder           *)
    CTU_Y1_M
         AT %MX2     : BOOL;   (* stores the counter status CTU_Y1             *)
    RS_Y1            : RS;     (* Flip-flop named RS_Y1 for Y1                 *)
    CTU_Y1           : CTU;    (* Incremental counter named CTU_Y1 for the     *)
                               (* cylinder movements                           *)
END_VAR
```

*Fig. B14.2:
Declaration of variables*

a) LD



b) FBD



c) IL

```
LD      S1
AND     B1
ANDN    CTU_Y1.Q
ST      Y1_advance
CAL     RS_Y1 (S := Y1_advance, R1 := B2)
LD      RS_Y1.Q1
ST      Y1
CAL     CTU_Y1 (CU := B2, R := S2, PU := 15)
```

d) ST

```
RS_Y1 ( S := S1 & B1 & NOT CTU_Y1.Q, R1 := B2);
Y1 := RS_Y1.Q1;
CTU (CU := B2, R := S2, PU := 15);
```

*Fig. B14.3:
Use of
incremental counter*

A CTU function block (incremental counter) is used in all languages to realise the counting function; in the actual example the name of the declared copy is CTU_Y1.

The cylinder is actuated via a valve Y1. The valve itself is switched via an RS flipflop named RS_Y1. The cylinder advances only if push button S1 is actuated and the cylinder (B1=1) retracted and not with an expired counter (CTU_Y1.Q = 0). When the cylinder has reached its forward end position (B2=1), the value of Y1 reverts to 0 and the cylinder retracts again.

The cylinder strokes are counted via the counter named CTU_Y1. The counter has a defined status at the beginning of processing, since all variables are preallocated. This means that if the cylinder is in the initial position and none of the push buttons are actuated, whereby a 0-signal is applied at B2 and S2 and thus at the CU and R input; the preselect value PV is 15, the current counter value CV is 0. The counter has therefore not yet expired, output Q has the value 0.

Actuation of push button S1 causes the cylinder to extend, the rising edge of B2 leads to a counting pulse and the current value CV of CTU_Y1 is increased by 1. When 15 cylinder movements have been executed, the current counter value CV is equal to the preselect value PV; the counter has expired and this is indicated by the value 1 at output Q. The cylinder will not move until the counter resets, i.e. starts anew. This occurs by actuating push-button S2; the 1-signal at the R-input sets the actual counter value CV at 0, following this a 0-signal applies at output Q.

Mention is to be made at this point of a particular feature of the IL language. In STL, transfer parameters for a function block must be individual variables only. Expressions are not permissible. This is why the AND operation of variables S1, B1 and CTU_Y1.Q is copied to the boolean variable Y1_advance and these are then used as transfer parameters.

## 14.3 Decremental counter

Function block CTD (count down) is the decremental counter of IEC 1131-3 and represents the counterpart of the incremental counter.

The decremental counter with preselect value PV is loaded with a 1signal at input LD (load). During normal operation, each positive edge at input CD (count down) reduces the counter reading. The current counter reading is also available at output CV in this instance. Output Q of function block CTD is 0, until the current counter reading CV becomes less than or equal to 0.

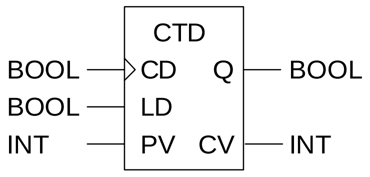


*Fig. B14.4: Function block CTD, decremental counter*

The use of a decremental counter is also demonstrated by a small example.

*Example*

A cylinder is moved via a valve Y1. The position of the cylinder is signalled via the sensors B1 (retracted) and B2 (extended). The cylinder is to advance, if push button S1 is pressed. When 10 strokes have been executed in this way, lamp H1 is illuminated and the counter has expired. The counter must be re-loaded with the preselect value, before any cylinder movements can be executed further. This is effected by means of actuating push button S2.

```
VAR
    S1 AT %IX1      : BOOL;   (* Push button for cylinder movement            *)
    S2 AT %IX2      : BOOL;   (* Push button for resetting of counter CTD_Y1 *)
    B1 AT %IX3      : BOOL;   (* Cylinder retracted                           *)
    B2 AT %IX4      : BOOL;   (* Cylinder advanced                            *)
    Y1 AT %QX1      : BOOL;   (* Advance cylinder                             *)
    H1 AT %QX2      : BOOL;   (* Lamp                                         *)
    RS_Y1           : RS;     (* Flip-flop named RS_Y1 for Y1                 *)
    CTD_Y1          : CTD;    (* Decremental counter named CTD_Y1 for the *)
                              (* cylinder movements                           *)
END_VAR
```

*Fig. B14.5: Declaration of variables*

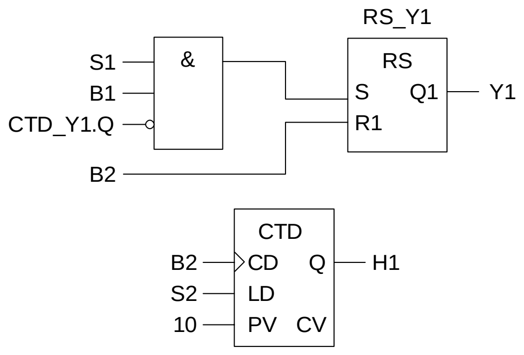


*Fig. B14.6:*
*Use of decremental counter in FBD language*

The valve Y1 is switched via an RS function block named RS_Y1. The set condition is met, when the cylinder is retracted, the counter has not yet expired and push button S1 is actuated. When the cylinder has extended completely, 0 applies again at output Q1 of RS_Y1.

The cylinder strokes are detected via a decremental counter named CTD_Y1. If the cylinder is in the initial position and none of the push buttons are actuated, the following values are applied at the inputs and outputs at the start of the decremental counter processing: the CD and the LD input have a 0-signal, the value 10 applies at input PV; the current counter value CV is 0, condition CV <= 0 is thus met and a 1-signal applies at output Q. The value 1 at output Q designates the decremental counter as expired. Lamp H1 is illuminated at the same time.

The preselect value 10 is not loaded as a current counter value until push button S2 is pressed. CV is now greater than 0, output Q is also O and the lamp is off. Cylinder movements may now be triggered by actuating push button S1. Each movement results in a counting pulse through the rising edge of B2, which reduces the current counter status by 1 each time. After 10 completed cylinder strokes the current counter reading is 0; the counter has expired. This is signalled by the value 1 at output Q.

Once the counter has been loaded with the start value 10, the counter operations may be repeated.

## 14.4 Incremental/decremental counter

Function block CTUD, Incremental/decremental counter, combines an incremental and a decremental counter.

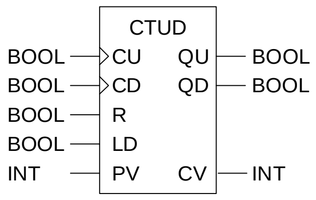


*Fig. B14.7: Function block CTUD, incremental/decremental counter*

The value of output QU is calculated in accordance with the equation: CV ≥ PV, the value of output QD in accordance with the equation CV ≤ 0.

Please note that the function of the decremental counter should be used only after the start value has been loaded to the counter via the command LD.

# Chapter 15

# Sequence control systems

## *15.1 What is a sequence control system*

Sequence control systems are processes in several, clearly separate steps. The progression from one step to the next depends on the step enabling conditions. One important characteristic is that only ever one step may be active or several steps only if these have been explicitely programmed as steps to be simultanously executed.

Compared with a logic control system, it offers the following advantages:

- the program is divided into steps and therefore more clearly arranged and easier to maintain and expand
- sequence control systems can easily be programmed graphically in a sequential function chart
- error detection in a process-related, graphically represented sequence control is more convenient and conclusive than that possible with logic control systems.

Typical examples for sequence controls are machine controls in the sphere of production technology or receptive controllers in process technology.

## *15.2 Function chart to IEC 848 or DIN 40 719, P.6*

The need for configuration is not immediately indicated in the case of small sequence-oriented controllers, but the need for improved functional descriptions increases with the growing complexity of tasks. Ladder diagrams and statement lists are poorly suited for structured description. Function charts (or also flow charts) were introduced as auxiliary means for top-down analysis and for the representation of processes function charts. The elements used for this type of description and their use have been standardised internationally by IEC 848. The IEC 848 standard with the addition of national definitions has been published in DIN 40 719, P.6.

Function charts describe in the main two aspects of a controller in accordance with defined rules:

- the actions to be executed (commands)
- the sequence of execution

A function chart is therefore divided into two parts (fig. B15.1). The sequence part represents the time-related execution of the process.

The sequence part does not describe the actions to be executed individually. These are contained in the action part of the function chart which, for the example in question, consists of blocks on the righthand side of the steps.

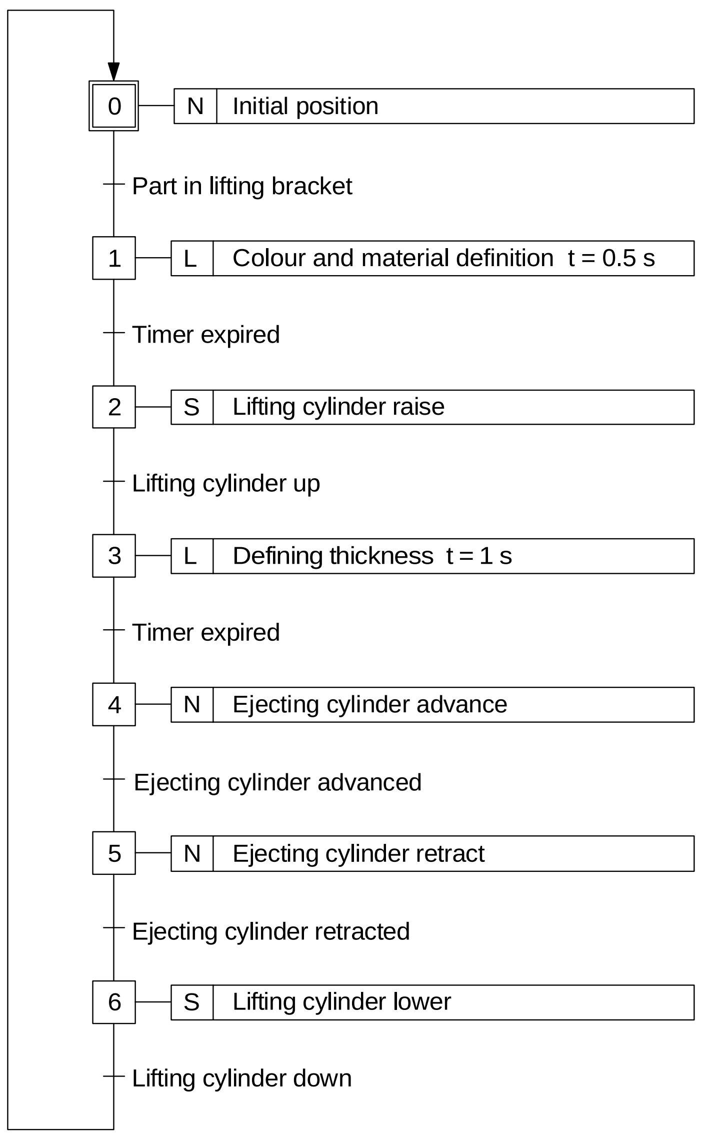

0
N
Initial position
Part in lifting bracket
1
L
Colour and material definition t = 0.5 s
Timer expired
2
S
Lifting cylinder raise
Lifting cylinder up
3
L
Defining thickness t = 1 s
Timer expired
4
N
Ejecting cylinder advance
Ejecting cylinder advanced
5
N
Ejecting cylinder retract
Ejecting cylinder retracted
6
S
Lifting cylinder lower
Lifting cylinder down

*Fig. B15.1: Function chart for a test process*

The following provides a brief explanation of the individual elements used to describe a function chart.

## Steps

A function chart is structured by means of steps. These are represented by blocks and identified with the respective step number.

The output status of the controller is identified by the initial step.

Each step is assigned actions (commands) containing the actual execution parts of the controller.

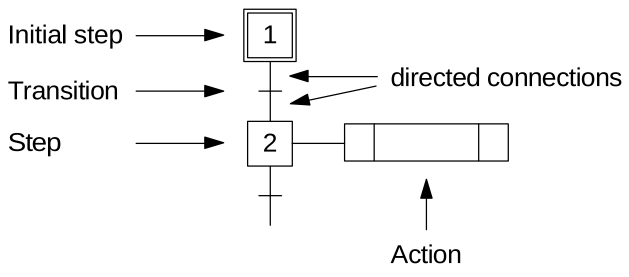


*Fig. B15.2:*
*Elements of function chart*

## Transitions

A transition is a link from one step to the next. The logic transition condition associated with the transition is represented next to the horizontal line across the transition. If the condition is met, the transition to the next step takes place and this is then processed by the controller.

**Sequence structures**
Three basic forms of sequence structure may be created by means of combining the step and transition elements:

- Linear sequence
- Sequence branch (alternative branch)
- Sequence splitting (parallel branch)

Steps and enabling conditions must always alternate irrespective of the form of the sequence structure. Sequence structures are processed from top to bottom.

In a linear sequence, only one transition follows a step and one step each transition. Fig. B15.1 illustrates an example of a linear sequence.

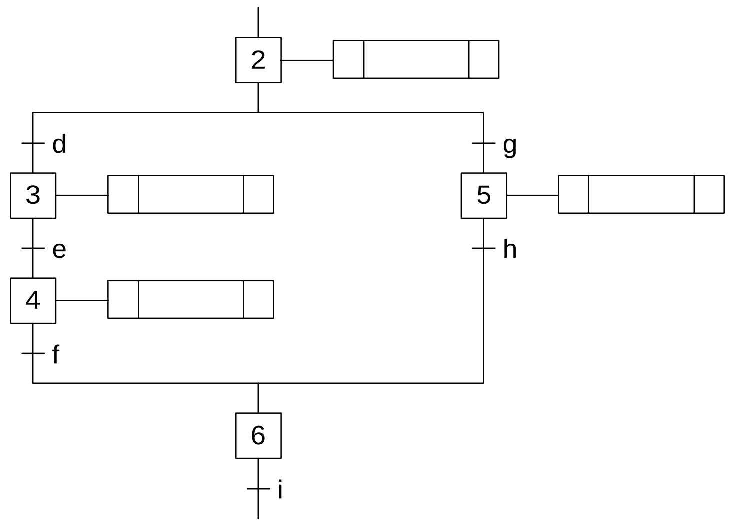


*Fig. B15.3: Alternative branch*

In the alternative branch shown in fig. B15.3, two or several transitions follow a step. The partial sequence, whose transition condition has been met first, is activated and processed. Since precisely one partial sequence may be selected with the alternative branch, the transition conditions – d and g in fig. B15.3 – must be mutually exclusive.

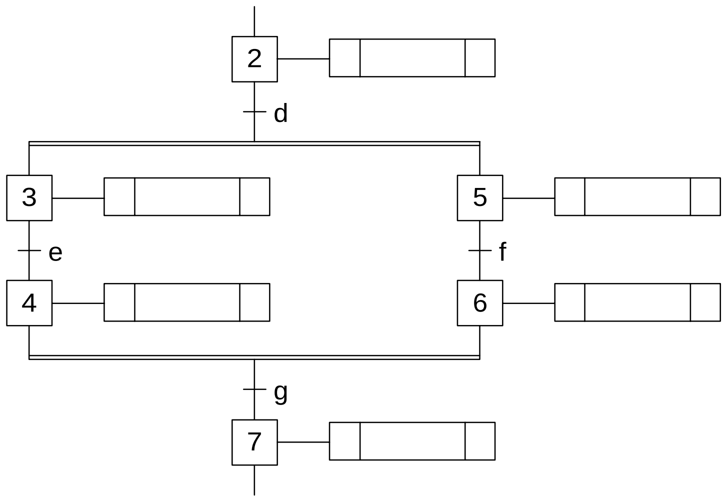
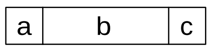


*Fig. B15.4: Parallel branch*

In the case of a parallel branch, of the transition condition is met, this leads to the simultaneous activation of several partial sequences. The partial sequences are evolved simultaneously, but completely independently of one another. The convergence of partial sequences is synchronised. Only when all parallel partial sequences have been evolved, may a transition to the next step underneath the double line – in this example to step 7 – take place.

**Action**
Each step contains actions, the actual execution parts of the controller. The action itself (fig. B15.5) is divided into three fields, whereby field a and c should only be represented if necessary.



a: Characterisation of actions to be executed

b: Description of action

c: Reference to all feedbacks associated with command

*Fig. B15.5: Action*

Table B15.1 contains the symbols defined in DIN 40 719, P.6 or IEC 848 used to describe the order of execution of the actions.

| | |
|---|---|
| S | stored |
| N | non-stored |
| D | delayed |
| F | enabling |
| L | limited |
| P | pulse |
| C | conditional |

*Table B15.1: Mode of actions*

If an action needs to be described in more detail, a combination of letter symbols should be selected in the order of this execution.

*Example*

**DCSF**
conditionally stored action after delay, subject to an additional enabling condition after storage.

**Step refinement**
As shown in fig. B15.6, each step may itself contain sequence structures. This facility is supported by the step-by-step refinement of a solution in the sense of a top-down design.

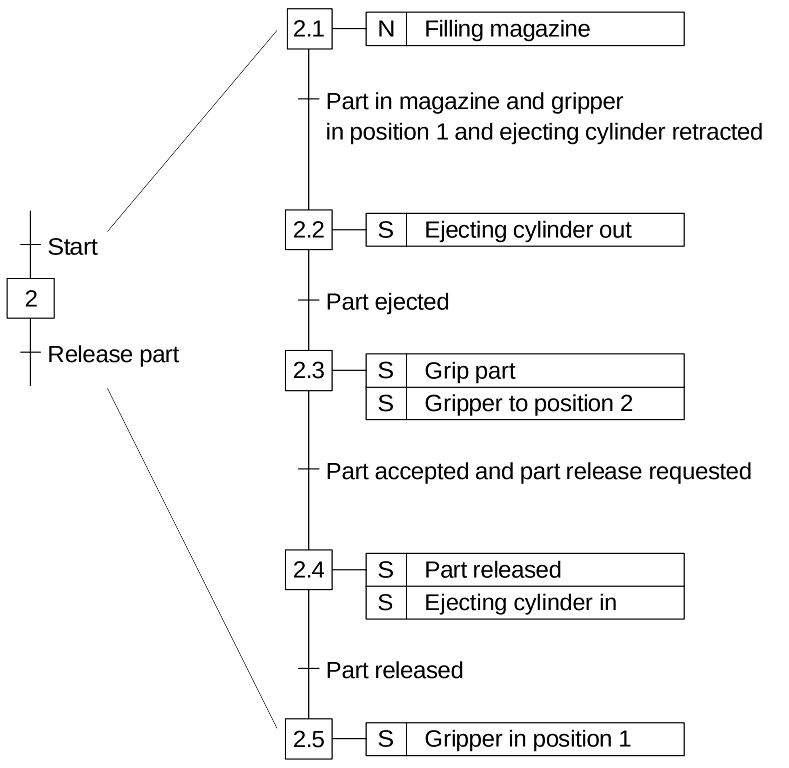


*Fig. B15.6: Step refinement*

## 15.3 Displacement-step diagram

The displacement-step diagram represents a sequence control graphically. The structure of a diagram of this type is described in VDI 3260.

The individual actuators and sensors are configured vertically in the diagram, and the individual steps of the controller horizontally. A function line indicates the signal status of the corresponding signalling element in each step. Signal lines link the individual function lines and indicate which signalling element in the process triggers which action. An arrow indicates the direction of action. The diagram is further clarified by symbols.

The displacement-step diagram is generally drawn up by the design engineer of a machine or system. When solving a control task, it is useful to design the displacement-step diagram prior to the programming.

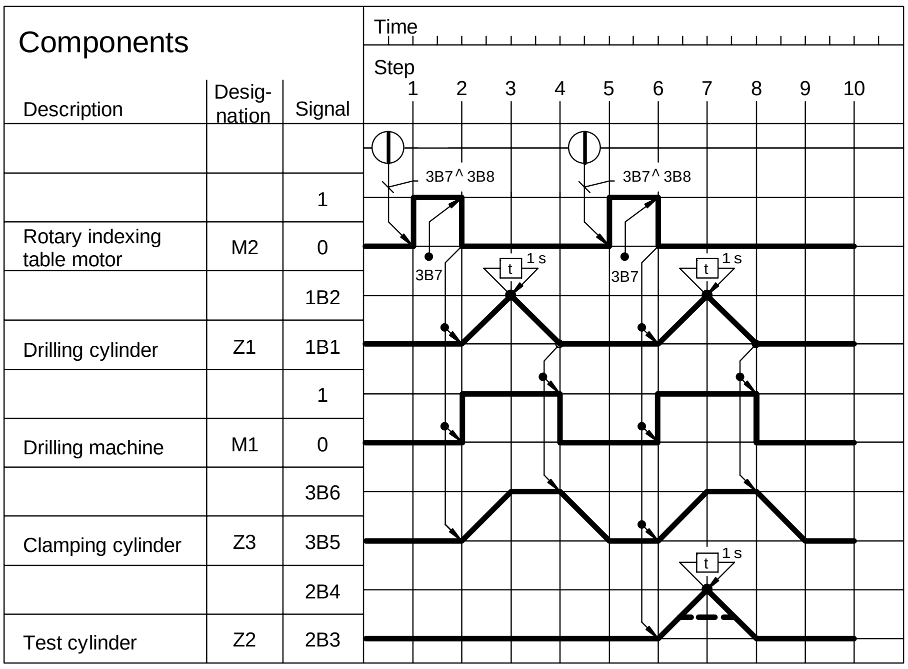


Fig. B15.7: Structure of a displacement-step diagram

# Chapter 16

# Commissioning and operational safety of a PLC

## 16.1 Commissioning

PLC programs are never final in that it is always possible to make corrections and subsequent adaptations to new system requirements.

Even during commissioning, program changes are often necessary. The commissioning of a system can be divided into basically four steps:

- Checking the hardware
- Transferring and testing the software
- Optimisation of software
- Commissioning of the system

**Checking the hardware**
Each sensor is connected to a specific input and each actuator to an output; addresses must not be mixed up.

This is why the first step in checking the hardware always comes after the allocation list. Are all the sensors and actuators allocated to the right input and output addresses? Is the function – for 0- and 1-signal – identified uniquely. The allocation list must be correct and completed in full as this forms part of the documentation prior to the commissioning of a program.

During checking, the outputs are set by way of a test. The actuators must then meet the functions specified.

**Transferring and testing of software**
Even prior to commissioning, all available off-line testing facilities of the program system should be used intensively. One such convenient test function is for instance, the simulation of the program.

Following this, the program is transferred to the central control unit of the PLC. A small number of PLCs now offer a facility for simulation: The entire program is executed without the inputs and outputs being connected. Similarly, only the connection of the outputs may be omitted. Processing of the PLC outputs thus only takes place in the image table, in that the image table is not switched through to the physically available outputs. This therefore eliminates the risk of damaging machines or system parts, which is of particular importance in the case of dangerous or critical processes.

After this, the individual program parts and system functions are tested: Manual operation, setting, individual monitoring programs etc., and finally the interaction of the program parts with the help of the overall program.

The system is therefore commissioned step-by-step. Important aspects of commissioning and error detection are test functions of the programming system such as single-step mode or the setting of stop points. Single-step mode  in particular is of importance, whereby the program in the PLC memory is executed line-by-line or step-by-step. In this way, any errors which may occur in the program can be immediately localised.

**Optimisation of software**
Larger programs can almost always be improved after the first test run. It is important that any corrections or modifications are made not just in the PLC program, but are also taken into account in the documentation. Apart from the documentation, the status of the software has to be saved.

**Commissioning of the system**
This already occurs in part during the testing and optimisation phase. Once the final status of PLC program and the documentation is established,  all the controller functions (in accordance with the task) need to be executed step for step again. The system is then ready to be accepted by the customer or the relevant department.

## *16.2 Operational safety of a PLC*

**PLC voltage supply**
Differentiation must be made between **Control voltage** (signals between controlled machine and the PLC) and the **Logic voltage** (for the internal voltage supply of the central control unit).

The level of the operating voltage of a PLC is specified in DIN IEC 1131/ Part 2. It is between 24VDC and 48VDC or 48VAC and 230VAC respectively. 120VAC may also apply for the American market.

**Control voltage**
The control voltage supplies the sensors and actuators. The user needs to connect a power supply unit to the controller for this. In Germany, the control voltage of a PLC is generally 24VDC or 230VAC. (In the main, DC voltage is used.) In other countries, different voltages are also used, e.g. 48VDC voltage or 120VAC voltage. How exactly the power supply unit is connected to the controller varies according to the PLC used.

The control voltage permits a certain amount of variation. The PLC modules are in the main protected against excessive voltages, depending on the module through which the central control unit is realised.

**Logic voltage**
In addition, a PLC requires a voltage supply for the internal logic: The logic voltage, which forms the signals in the central control unit. This is why the logic voltage must meet very high requirements, i.e., it needs to be stabilised. Either 5 V (TTL level) is used for this or approx, 10 V (CMOS level), depending on the module through which the central control unit is realised.

There are three possibilities of **Voltage supply**:

1. Control voltage and logic voltage are generated completely separately from the mains voltage.
2. Two power supply units combined in one housing for the generation of the two voltages.
3. The logic voltage is generated from the control voltage (not the mains voltage).

**Interference suppression**
All PLCs are extremely sensitive to voltage supply interference. Differentiation should be made between to different versions:

- Interferences reaching the logic voltage from the voltage supply via the power supply unit;
- Interferences, affecting the lines to and from the sensors and actuators.

1. **Interferences in the logic voltage**
   A main interference suppression filter and a capacitor protect against interferences of this type. The mains interference suppression filter protects against overvoltages and interference signals from voltage supply. A capacitor stores electrical energy, whereby the controller voltage supply is protected even in the event of brief voltage failures.

   If this type of voltage suppression has not been provided by the PLC manufacturer, a mains interference suppression filter and capacitor is to be fitted subsequently by the user.

2. **Line interferences to and from sensors and actuators**
   Interference pulses on electrical lines may cause a 1- or 0-signal to occur at PLC inputs, which has not be supplied by a sensor. The signal may be created as a result of effects from other cables.

   This kind of interference is dangerous: As a rule, input modules of a PLC are therefore protected by means of a series connected optocoupler and signal delay. The optocoupler protects against overvoltages of up to approx. 5000 V. The signal delay prevents spurious signals, since these are generally very brief. The length of time varies between 1 and 20 ms depending on the PLC. “High speed” input modules (without signal delay) must be shielded, for example, by screened cables.

Output modules also contain an optocoupler for protection against overvoltages. Moreover, the outputs are short-circuit protected, though normally not against sustained short-circuit.

## Mutual induction voltage

When inductive actuators (e.g. safety coils, solenoid coils) are switched, it creates a mutual induction voltage at the coil.

This mutual induction voltage must be eliminated to protect the output module. A suppressor diode is used for this. The output modules of a number of PLCs are already equipped with suppressor diodes of this type. The residual voltage in this case, however, remains an interference factor on the interconnecting cables. This is why the protective measures should be taken direct at the point of origin, i.e. on the coil: by means of a **suppressor diode** (for DC voltage only) or a **varistor** (voltage-dependent resistor). Two reverse polarity and **Zener diodes** switched parallel to the coil may also be used. With a voltage in excess of 150 V, however, several breakdown diodes must be switched in series.

## EMERGENCY-STOP

If the EMERGENCY-STOP device is actuated, it is essential for a condition to be achieved, which is harmless both for people and system. Final control elements and drives, which may produce dangerous situations, must be switched off immediately (e.g. spindle drives). Conversely, final control elements and drives, which may prove dangerous to people or the system when  switched off, must continue operating even in an emergency (e.g. clamping devices). The facility to operate an EMERGENCY-STOP must be available at any given time in a system.

This is why a standard electronic controller may not assume the EMERGENCY-STOP function. The EMERGENCY-STOP circuit must be established independently of the PLC by means of contactor technology. DIN 57116 also specifies this, since it would be impossible to switch an EMERGENCY-STOP with a malfunctioning controller.

Once the EMERGENCY-STOP device has been unlatched, it should be no longer possible for machinery to operate automatically.

*Example*

An output transistor of the PLC is burnt out. A voltage of 24 V applies permanently at the output (corresponding to a 1-signal). The solenoid value is actuated; the cylinder extends, even though the system has not been enabled. If the EMERGENCY-STOP command were to be executed by the PLC program, it would remain inactive, since the error only occurs "after" the program. The EMERGENCY-STOP command must therefore take effect downstream of the PLC as far as the cylinder.

One method is to connect the EMERGENCY-STOP function to the voltage supply of the output modules. The connection must be fail-safe. In the event of an EMERGENCY-STOP, all outputs assume the 0-signal. It is of no importance whether a particular output has been set or reset by the PLC.

If this method is employed, the connected actuators must move into a non hazardous position in the event of a 0-signal! The following actuators should be used if possible:

**Hydraulic/pneumatic valves:**
5/4- or 5/3-way valves with mid-position normally closed are used (poss. with additional clamping cylinder). These valves clamp the cylinder between fluid or air cushions. Correct connection: Short paths from cylinder to valve, restrictors in exhaust air of the valve.

**Electric motors:**
Brake motors are used. In the event of voltage failure, the brake comes into full effect as a result of spring force.

The EMERGENCY-STOP circuit in the **Hardware** carries out the actual safety function. In addition, the EMERGENCY-STOP command must also be stored in the **PLC program**. Whatever has been effected beyond the PLC by way of hardware must be retraced in each program: in this case, switching off the outputs. This is defined in a parallel program. Once the EMERGENCY-STOP has been reset, the system should not be able to start again on its own. A separate push button/switch is to be actuated to start the system.

The re-starting of the system may be controlled by means of the PLC program. There are two methods for starting:

- Continuing from the same point;
- Returning to the initial position and re-starting the machine.

In the second instance, it is necessary to switch over to manual or setting mode.

If **additional safety measures** are required for EMERGENCY-STOP, user's own **relays** or **pneumatic controllers** must be used. A special safety-oriented PLC may also be used, which operates by means of two separate central control units with two series connected output stages each.

**Fail-safe connection**
The majority of machines are switched on by means of one push button and switched off via another push button. The **OFF push button** additionally assumes a safety function: The working process may be interrupted at any time and the machine stopped. With EMERGENCY-STOP, however, the entire system is switched off. In contrast with EMERGENCY-STOP, the OFF function is controlled via the PLC.

It should be noted, however, that the OFF function must be maintained even if the wiring of the OFF button is defective. The connection must be **fail-safe**; i.e. the OFF button is to be connected and programmed in the form of a **normally closed contact**. (The allocation list provides information regarding the significance of the 1- and 0-signal!)

*Example* A signal generator monitors the oil temperature of a gear unit. For safety reasons, the connection of the signal generator is to be fail-safe: The 1-signal identifies the correct temperature, the 0-signal the incorrect temperature. If the connection is defective, the signal generator also assumes a 0-signal (even if in this case the cause is not the incorrect temperature).

This eliminates the situation, where a critical condition in the system is no longer signalled by the signal generator because of defective wiring.

# Chapter 17

# Communication

## 17.1 The need for communication

By communication, we understand the transfer of information i.e. data between the programmable logic controller and other data processing devices, whereby these devices are used as an auxiliary means for specific control tasks, e. g. input of data takes place via a computer, output of data via a printer controlling still remains the task of the PLC.

Automation increases the need for communication. Data needs to be continually passed on from production to other operational areas. This provides an overview of the production status and the individual tasks (production data acquisition).

Automated systems nowadays are equipped with complex error and fault detection systems. Fault indications and warnings must be generated, centralised and communicated automatically to the operator. To this end, a printer – for logging – or an electronic display is connected to the controller.

In some cases, data is to be transferred to the PLC by a computer in an active process, or several control devices are combined into one system network.

## 17.2 Data transmission

How can the PLC communicate with other data processing devices? The individual bits, which are combined into one data word, must be transmitted from one piece of data terminal equipment to another.

Basic differentiation is made here between two methods: parallel or serial data transmission.

Parallel data transmission means that a separate line must be available for each individual binary signal. When signal generators for example, are connected to a programmable logic controller, a separate wire is installed for each push button, limit switch, limiting value encoder and sensor to a terminal strip and from there to the input of the PLC. All information (“push button actuated”, “cylinder advanced”) can in this way be transmitted simultaneously (parallel) to the PLC. Since in the case of parallel transmission of input and output signals, a line is required for each signal generator, literally miles of cable bundles are installed overall for correspondingly complex machines.

For the parallel transmission of a data word, sufficient lines must therefore be available to transmit all bits of this data word simultaneously.

With **serial** data transmission only one binary signal is transmitted at a time. Again, using the example of the PLC: If several modules of a PLC are interconnected, it is not necessary for a individual line to be installed for each input or output, instead the information regarding inputs or outputs is transmitted consecutively (serial).

Accordingly only one data line is therefore required for the serial tansmission of data words, irrespective of the number of bits, to transmit the binary signals consecutively. In order to now be able to represent the various signals in the form of a related data word, it is necessary to agree the transmission speed, word length and specific start and end characters.

## *17.3 Interfaces*

Different coding procedures, transmission and operating methods as well as different methods of data protection make it essential to define electrical, functional and mechanical characteristics of interfaces in standards.

A parallel interface is also known as a Centronics interface. 8 data lines are available for data transmission, i.e. 8 bits may be transmitted simultaneously. The Centronics interface is very frequently used – over small distances – for the actuation of printers.

| | ***Voltage interfaces*** | | ***Current interface*** |
|---|---|---|---|
| Designation | V.24 | Centronics | 20 mA |
| Transmission mode | serial asynchronous | parallel | serial asynchronous |
| Mode of operation | full duplex | simplex | full duplex |
| Standard | V.24 RS-232-C | Centronics TTL | TTY |
| Transmission distance, transmission speed | up to 30 m 20 000 bit/s | up to 2 m $10^6$ bit/s | up to 1000 m 20 000 bit/s |
| Logic level Data line | 15 V ≥ ’0’ ≥ 3V -3 V ≥ ’1’ ≥ -15 V | ’1’ ≥ 2.4 V ’0’ ≥ -0.8 V | ’1’ = Current off ’0’ = Current on |

*Table B17.1: Interfaces*

The most frequently used interface for serial data transmission is the V.24 interface.

The Centronics and V.24 interface are both voltage interfaces. Bits are represented for ’0’ or ’1’ via a specified voltage level. In order to create this signal level, a joint ground line must be incorporated for the V.24 interface. In the case of a Centronics interface, each data line has its own ground line.

In the case of both interfaces, additional lines have been defined for data flow control apart from the data and ground lines.

Considerably more simple than via a V.24 interface is a connection configured via a serial 20 mA interface. All this current-loop interface needs is a transmitter and receiver loop for the transmission of data. A constant current of 20 mA signals the ’0’level (logic0), "current off" signals the ’1’level (logic1) on the data line. This interface is widely used in control technology due to its interference immunity.

## *17.4 Communication in the field area*

A multitude of information has to be transported within automated systems and machines. Simple binary sensor signals, analogue signals of measuring sensors or proportional valves, and also recorded data and parameters for the control of processes need to be exchanged reliably between the control technology components of an automated system.

The data exchange for this must take place within specified reaction times, since system parts could otherwise continue to operate uncontrolled.

A fieldbus is a serial, digital transmission system for these signals and data. All stations on a fieldbus must be in a position to receive the communication from other bus stations and to exchange data in accordance with the agreed protocol. A bus station, taking the initiative for the data exchange is known as a master. Bus stations receiving or supplying data purely on the instruction of the master are termed slaves.

Two-wire cables consisting of either twisted pairs or coaxial cables are used for the transmission of data in bus systems. The extent of wiring for bus coupled systems is therefore minimal.

A multitude of different bus systems is available in the market place, which can basically be divided into 2 groups: closed and open bus systems.

By **closed systems** we understand systems, which are

- vendor-specific,
- do not have any transmission protocol disclosure and
- are not compatible. Furthermore, they do not permit interfacing with devices of other manufacturers and adaptation associated with high expenditure.

Closed systems, for instance, are SINEC L1 by Siemens, SUCOnet K by Klöckner-Moeller, Data Highway by Allen Bradley, Festo Feldbus, Modnet by AEG/MODICON.

**Open systems**, in contrast have

- standardised interfaces and protocols,
- declared protocols and
- a multitude of devices by different manufacturers may be connected to the bus.

Open systems, for instance, are Profibus, Interbus-S, CAN, SINEC L2, ASI.

The advantages of networking with open bus systems are as follows:

- Decentralisation of control function
- Coordination of processes in separate areas
- Realisation of control and production data flow
  parallel to material flow
- Simplification of the installation and
  reduction of wiring costs (two-wire bus)
- Simplification of the commissioning of a system
  (greater clarity, pretested subsystems)
- Reduction in service costs
  (central system diagnostics)
- Use of equipment by different manufacturers
  in the same network
- Process data transmission right up to planning level

# Appendix

| | | |
|---|---|---|
| ***Bibliography of illustrations*** | Fig. B1.2: | Example of a PLC: AEG Modicon A120<br>AEG Schneider Automation GmbH,<br>Steinheimer Straße 117, 63500 Seligenstadt |
| | Fig. B1.4: | Compact PLC (Mitsubishi FX0)<br>Mitsubishi Electric Europe GmbH,<br>Gothaer Straße 8, 40880 Ratingen |
| | Fig. B1.4: | Modular PLC (Siemens S7-300)<br>Siemens AG,<br>AUT 111, Postfach 4848, 90475 Nürnberg |

| | | |
|---|---|---|
| ***Bibliography of literature*** | Kostka, Winfried | Dictionary of control technology<br>German-English/English-German<br>Festo Didactic KG, Esslingen, 1988 |
| | | Lexicon of control technology<br>Festo Didactic KG, Esslingen, 1988 |

| | | |
|---|---|---|
| ***Guidlines and standards*** | DIN VDE 0113/<br>EN 60204 | Electrical equipment of industrial machinery;<br>General definitions |
| | IEC 1131/<br>DIN EN 61131 | Programmable logic controllers;<br>Part 1: General information<br>Part 2: Equipment, requirements and tests<br>Part 3: Programming languages<br>Part 4: User guidelines (in preparation with ICE)<br>Part 5: Messaging service specification<br>(in preparation with ICE) |
| | DIN IEC 113 | Circuit documentation;<br>Part 7: Use of circuit symbols for binary elements<br>in circuit diagrams |
| | IEC 848 | Preparation of function charts for control systems |

# Index