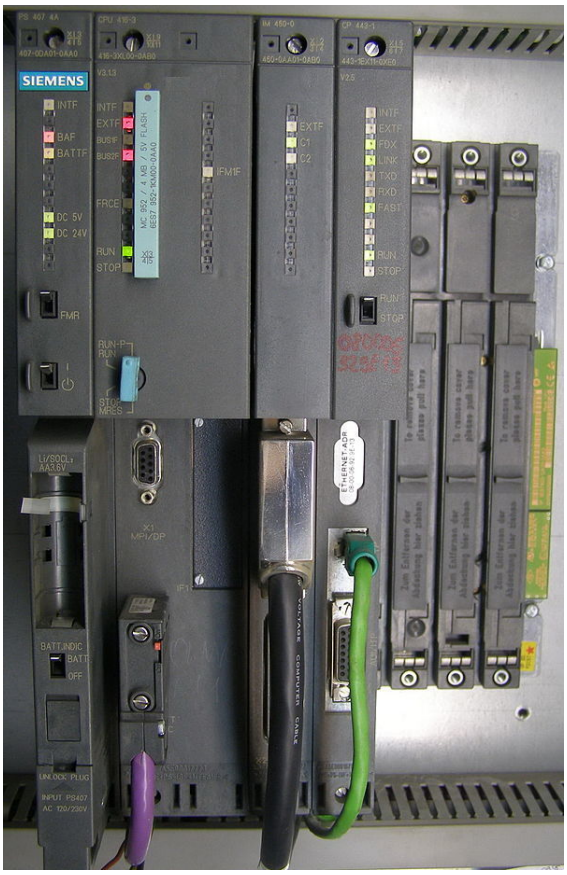
PLC

Contents

Chapter 1

Programmable logic controller



Siemens Simatic S7-400 system at rack, left-to-right: power supply unit PS407 4A, CPU 416-3, interface module IM 460-0 and communication processor CP 443-1.

A **programmable logic controller**, **PLC** or **programmable controller** is a digital computer used for automation of typically industrial electromechanical processes, such as control of machinery on factory assembly lines, amusement rides, or light fixtures. PLCs are used in many industries and machines. PLCs are designed for multiple analogue and digital inputs and output arrangements, extended temperature ranges, immunity to electrical noise, and resistance to vibration and impact. Programs to control machine operation are typically stored in battery-backed-up or non-volatile memory. A PLC is an example of a "hard" real-time system since output results must be produced in response to input conditions within a limited time, otherwise unintended operation will result.

1.1 History

Before the PLC, control, sequencing, and safety interlock logic for manufacturing automobiles was mainly composed of relays, cam timers, drum sequencers, and dedicated closed-loop controllers. Since these could number in the hundreds or even thousands, the process for updating such facilities for the yearly model change-over was very time consuming and expensive, as electricians needed to individually rewire the relays to change their operational characteristics.

Digital computers, being general-purpose programmable devices, were soon applied to control of industrial processes. Early computers required specialist programmers, and stringent operating environmental control for temperature, cleanliness, and power quality. Using a general-purpose computer for process control required protecting the computer from the plant floor conditions. An industrial control computer would have several attributes: it would tolerate the shop-floor environment, it would support discrete (bit-form) input and output in an easily extensible manner, it would not require years of training to use, and it would permit its operation to be monitored. The response time of any computer system must be fast enough to be useful for control; the required speed varying according to the nature of the process.[1] Since many industrial processes have timescales easily addressed by millisecond response times, modern (fast, small, reliable) electronics greatly facilitate building reliable controllers, especially because performance can be traded off for reliability.

In 1968 GM Hydra-Matic (the automatic transmission division of General Motors) issued a request for proposals for an electronic replacement for hard-wired relay systems based on a white paper written by engineer Edward R. Clark. The winning proposal came from Bedford Associates of Bedford, Massachusetts. The first PLC, designated the 084 because it was Bedford Associates' eighty-fourth project, was the result.[2] Bedford

Associates started a new company dedicated to developing, manufacturing, selling, and servicing this new product: Modicon, which stood for **MO**dular **DI**gital **CON**troller. One of the people who worked on that project was Dick Morley, who is considered to be the "father" of the PLC.[3] The Modicon brand was sold in 1977 to Gould Electronics, and later acquired by German Company AEG and then by French Schneider Electric, the current owner.

One of the very first 084 models built is now on display at Modicon's headquarters in North Andover, Massachusetts. It was presented to Modicon by GM, when the unit was retired after nearly twenty years of uninterrupted service. Modicon used the 84 moniker at the end of its product range until the 984 made its appearance.

The automotive industry is still one of the largest users of PLCs.

1.2 Development

Early PLCs were designed to replace relay logic systems. These PLCs were programmed in "ladder logic", which strongly resembles a schematic diagram of relay logic. This program notation was chosen to reduce training demands for the existing technicians. Other early PLCs used a form of instruction list programming, based on a stack-based logic solver.

Modern PLCs can be programmed in a variety of ways, from the relay-derived ladder logic to programming languages such as specially adapted dialects of BASIC and C. Another method is State Logic, a very high-level programming language designed to program PLCs based on state transition diagrams.

Many early PLCs did not have accompanying programming terminals that were capable of graphical representation of the logic, and so the logic was instead represented as a series of logic expressions in some version of Boolean format, similar to Boolean algebra. As programming terminals evolved, it became more common for ladder logic to be used, for the aforementioned reasons and because it was a familiar format used for electromechanical control panels. Newer formats such as State Logic and Function Block (which is similar to the way logic is depicted when using digital integrated logic circuits) exist, but they are still not as popular as ladder logic. A primary reason for this is that PLCs solve the logic in a predictable and repeating sequence, and ladder logic allows the programmer (the person writing the logic) to see any issues with the timing of the logic sequence more easily than would be possible in other formats.

1.2.1 Programming

Early PLCs, up to the mid-1990s, were programmed using proprietary programming panels or special-purpose programming terminals, which often had dedicated function keys representing the various logical elements of PLC programs.[2] Some proprietary programming terminals displayed the elements of PLC programs as graphic symbols, but plain ASCII character representations of contacts, coils, and wires were common. Programs were stored on cassette tape cartridges. Facilities for printing and documentation were minimal due to lack of memory capacity. The very oldest PLCs used non-volatile magnetic core memory.

More recently, PLCs are programmed using application software on personal computers, which now represent the logic in graphic form instead of character symbols. The computer is connected to the PLC through Ethernet, RS-232, RS-485 or RS-422 cabling. The programming software allows entry and editing of the ladder-style logic. Generally the software provides functions for debugging and troubleshooting the PLC software, for example, by highlighting portions of the logic to show current status during operation or via simulation. The software will upload and download the PLC program, for backup and restoration purposes. In some models of programmable controller, the program is transferred from a personal computer to the PLC through a programming board which writes the program into a removable chip such as an EEPROM or EPROM.

1.3 Functionality

The functionality of the PLC has evolved over the years to include sequential relay control, motion control, process control, distributed control systems and networking. The data handling, storage, processing power and communication capabilities of some modern PLCs are approximately equivalent to desktop computers. PLC-like programming combined with remote I/O hardware, allow a general-purpose desktop computer to overlap some PLCs in certain applications. Regarding the practicality of these desktop computer based logic controllers, it is important to note that they have not been generally accepted in heavy industry because the desktop computers run on less stable operating systems than do PLCs, and because the desktop computer hardware is typically not designed to the same levels of tolerance to temperature, humidity, vibration, and longevity as the processors used in PLCs. In addition to the hardware limitations of desktop based logic, operating systems such as Windows do not lend themselves to deterministic logic execution, with the result that the logic may not always respond to changes in logic state or input status with the extreme consistency in timing as is expected from PLCs. Still, such desktop logic applications find use in less critical situations, such as lab-

oratory automation and use in small facilities where the application is less demanding and critical, because they are generally much less expensive than PLCs.

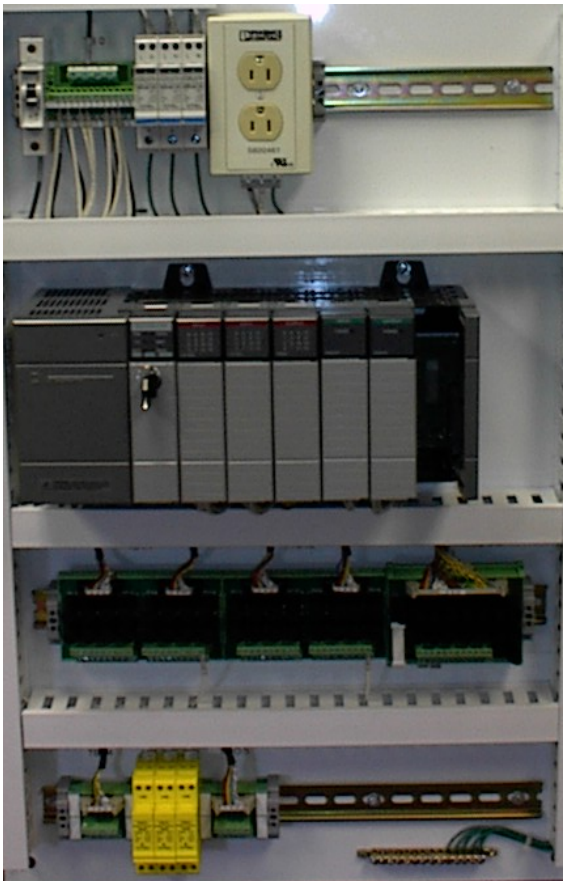
1.3.1 Programmable logic relay (PLR)

In more recent years, small products called PLRs (programmable logic relays), and also by similar names, have become more common and accepted. These are very much like PLCs, and are used in light industry where only a few points of I/O (i.e. a few signals coming in from the real world and a few going out) are involved, and low cost is desired. These small devices are typically made in a common physical size and shape by several manufacturers, and branded by the makers of larger PLCs to fill out their low end product range. Popular names include PICO Controller, NANO PLC, and other names implying very small controllers. Most of these have between 8 and 12 discrete inputs, 4 and 8 discrete outputs, and up to 2 analog inputs. Size is usually about 4" wide, 3" high, and 3" deep. Most such devices include a tiny postage stamp sized LCD screen for viewing simplified ladder logic (only a very small portion of the program being visible at a given time) and status of I/O points, and typically these screens are accompanied by a 4-way rocker push-button plus four more separate push-buttons, similar to the key buttons on a VCR remote control, and used to navigate and edit the logic. Most have a small plug for connecting via RS-232 or RS-485 to a personal computer so that programmers can use simple Windows applications for programming instead of being forced to use the tiny LCD and push-button set for this purpose. Unlike regular PLCs that are usually modular and greatly expandable, the PLRs are usually not modular or expandable, but their price can be two orders of magnitude less than a PLC and they still offer robust design and deterministic execution of the logic.

1.4 PLC topics

1.4.1 Features

The main difference from other computers is that PLCs are armored for severe conditions (such as dust, moisture, heat, cold) and have the facility for extensive input/output (I/O) arrangements. These connect the PLC to sensors and actuators. PLCs read limit switches, analog process variables (such as temperature and pressure), and the positions of complex positioning systems. Some use machine vision.[4] On the actuator side, PLCs operate electric motors, pneumatic or hydraulic cylinders, magnetic relays, solenoids, or analog outputs. The input/output arrangements may be built into a simple PLC, or the PLC may have external I/O modules attached to a computer network that plugs into the PLC.



Control panel with PLC (grey elements in the center). The unit consists of separate elements, from left to right; power supply, controller, relay units for in- and output

1.4.2 Scan time

A PLC program is generally executed repeatedly as long as the controlled system is running. The status of physical input points is copied to an area of memory accessible to the processor, sometimes called the "I/O Image Table". The program is then run from its first instruction rung down to the last rung. It takes some time for the processor of the PLC to evaluate all the rungs and update the I/O image table with the status of outputs.[5] This scan time may be a few milliseconds for a small program or on a fast processor, but older PLCs running very large programs could take much longer (say, up to 100 ms) to execute the program. If the scan time were too long, the response of the PLC to process conditions would be too slow to be useful.

As PLCs became more advanced, methods were developed to change the sequence of ladder execution, and subroutines were implemented.[6] This simplified programming could be used to save scan time for high-speed processes; for example, parts of the program used only for setting up the machine could be segregated from those parts required to operate at higher speed.

Special-purpose I/O modules, such as timer modules or

counter modules such as encoders, can be used where the scan time of the processor is too long to reliably pick up, for example, counting pulses and interpreting quadrature from a shaft encoder. The relatively slow PLC can still interpret the counted values to control a machine, but the accumulation of pulses is done by a dedicated module that is unaffected by the speed of the program execution...

1.4.3 System scale

A small PLC will have a fixed number of connections built in for inputs and outputs. Typically, expansions are available if the base model has insufficient I/O.

Modular PLCs have a chassis (also called a rack) into which are placed modules with different functions. The processor and selection of I/O modules are customized for the particular application. Several racks can be administered by a single processor, and may have thousands of inputs and outputs. A special high speed serial I/O link is used so that racks can be distributed away from the processor, reducing the wiring costs for large plants.

1.4.4 User interface

See also: User interface and List of human-computer interaction topics

PLCs may need to interact with people for the purpose of configuration, alarm reporting or everyday control. A human-machine interface (HMI) is employed for this purpose. HMIs are also referred to as man-machine interfaces (MMIs) and graphical user interfaces (GUIs). A simple system may use buttons and lights to interact with the user. Text displays are available as well as graphical touch screens. More complex systems use programming and monitoring software installed on a computer, with the PLC connected via a communication interface.

1.4.5 Communications

PLCs have built in communications ports, usually 9-pin RS-232, but optionally EIA-485 or Ethernet. Modbus, BACnet or DF1 is usually included as one of the communications protocols. Other options include various fieldbuses such as DeviceNet or Profibus. Other communications protocols that may be used are listed in the List of automation protocols.

Most modern PLCs can communicate over a network to some other system, such as a computer running a SCADA (Supervisory Control And Data Acquisition) system or web browser.

PLCs used in larger I/O systems may have peer-to-peer (P2P) communication between processors. This allows separate parts of a complex process to have individual control while allowing the subsystems to co-ordinate over the communication link. These communication links are also often used for HMI devices such as keypads or PC-type workstations.

1.4.6 Programming

PLC programs are typically written in a special application on a personal computer, then downloaded by a direct-connection cable or over a network to the PLC. The program is stored in the PLC either in battery-backed-up RAM or some other non-volatile flash memory. Often, a single PLC can be programmed to replace thousands of relays.[7]

Under the IEC 61131-3 standard, PLCs can be programmed using standards-based programming languages. A graphical programming notation called Sequential Function Charts is available on certain programmable controllers. Initially most PLCs utilized Ladder Logic Diagram Programming, a model which emulated electromechanical control panel devices (such as the contact and coils of relays) which PLCs replaced. This model remains common today.

IEC 61131-3 currently defines five programming languages for programmable control systems: function block diagram (FBD), ladder diagram (LD), structured text (ST; similar to the Pascal programming language), instruction list (IL; similar to assembly language) and sequential function chart (SFC).[8] These techniques emphasize logical organization of operations.[7]

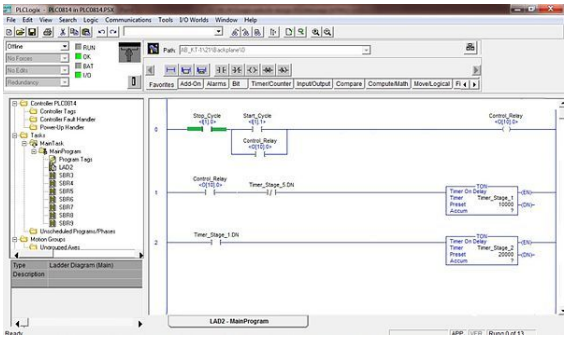
While the fundamental concepts of PLC programming are common to all manufacturers, differences in I/O addressing, memory organization and instruction sets mean that PLC programs are never perfectly interchangeable between different makers. Even within the same product line of a single manufacturer, different models may not be directly compatible.

1.4.7 Security

Prior to the discovery of the Stuxnet computer virus in June 2010, security of PLCs received little attention. PLCs generally contain a real-time operating system such as OS-9 or VxWorks and exploits for these systems exist much as they do for desktop computer operating systems such as Microsoft Windows. PLCs can also be attacked by gaining control of a computer they communicate with.[9]

1.4.8 Simulation

In order to properly understand the operation of a PLC, it is necessary to spend considerable time programming, testing, and debugging PLC programs. PLC systems are inherently expensive, and down-time is often very costly.



PLCLogix Simulation Software

In addition, if a PLC is programmed incorrectly it can result in lost productivity and dangerous conditions. PLC simulation software is a valuable tool in the understanding and learning of PLCs and to keep this knowledge refreshed and up to date. The advantages of using PLC simulation tools such as PLCLogix are that they save time in the design of automated control applications and they can also increase the level of safety associated with equipment since various "what if" scenarios can be tried and tested before the system is activated.[10]

1.4.9 Redundancy

Some special processes need to work permanently with minimum unwanted down time. Therefore, it is necessary to design a system which is fault tolerant and capable of handling the process with faulty modules. In such cases to increase the system availability in the event of hardware component failure, redundant CPU or I/O modules with the same functionality can be added to hardware configuration for preventing total or partial process shutdown due to hardware failure from any kind.

1.5 PLC compared with other control systems

Allen-Bradley PLC installed in a control panel

PLCs are well adapted to a range of automation tasks. These are typically industrial processes in manufacturing where the cost of developing and maintaining the automation system is high relative to the total cost of the automation, and where changes to the system would be expected during its operational life. PLCs contain input and output devices compatible with industrial pilot devices and controls; little electrical design is required, and the design problem centers on expressing the desired sequence of operations. PLC applications are typically highly customized systems, so the cost of a packaged PLC is low compared to the cost of a specific custom-built controller design. On the other hand, in the case of mass-produced goods, customized control systems are economical. This is due to the lower cost of the components, which can be optimally chosen instead of a "generic" solution, and where the non-recurring engineering charges are spread over thousands or millions of units.

For high volume or very simple fixed automation tasks, different techniques are used. For example, a consumer dishwasher would be controlled by an electromechanical cam timer costing only a few dollars in production quantities.

A microcontroller-based design would be appropriate where hundreds or thousands of units will be produced and so the development cost (design of power supplies, input/output hardware and necessary testing and certification) can be spread over many sales, and where the end-user would not need to alter the control. Automotive applications are an example; millions of units are built each year, and very few end-users alter the programming of these controllers. However, some specialty vehicles such as transit buses economically use PLCs instead of custom-designed controls, because the volumes are low and the development cost would be uneconomical.[11]

Very complex process control, such as used in the chemical industry, may require algorithms and performance beyond the capability of even high-performance PLCs. Very high-speed or precision controls may also require customized solutions; for example, aircraft flight controls. Single-board computers using semi-customized or fully proprietary hardware may be chosen for very demanding control applications where the high development and maintenance cost can be supported. "Soft PLCs" running on desktop-type computers can interface with industrial I/O hardware while executing programs within a version of commercial operating systems adapted for process control needs.[11]

Programmable controllers are widely used in motion control, positioning control and torque control. Some manufacturers produce motion control units to be integrated with PLC so that G-code (involving a CNC machine) can be used to instruct machine movements.

PLCs may include logic for single-variable feedback analog control loop, a proportional, integral, derivative (PID) controller. A PID loop could be used to control the temperature of a manufacturing process, for example. His-

torically PLCs were usually configured with only a few analog control loops; where processes required hundreds or thousands of loops, a distributed control system (DCS) would instead be used. As PLCs have become more powerful, the boundary between DCS and PLC applications has become less distinct.

PLCs have similar functionality as remote terminal units (RTU). An RTU, however, usually does not support control algorithms or control loops. As hardware rapidly becomes more powerful and cheaper, RTUs, PLCs and DCSs are increasingly beginning to overlap in responsibilities, and many vendors sell RTUs with PLC-like features and vice versa. The industry has standardized on the IEC 61131-3 functional block language for creating programs to run on RTUs and PLCs, although nearly all vendors also offer proprietary alternatives and associated development environments.

In recent years "safety" PLCs have started to become popular, either as standalone models or as functionality and safety-rated hardware added to existing controller architectures (Allen Bradley Guardlogix, Siemens F-series etc.). These differ from conventional PLC types as being suitable for use in safety-critical applications for which PLCs have traditionally been supplemented with hard-wired safety relays. For example, a safety PLC might be used to control access to a robot cell with trapped-key access, or perhaps to manage the shutdown response to an emergency stop on a conveyor production line. Such PLCs typically have a restricted regular instruction set augmented with safety-specific instructions designed to interface with emergency stops, light screens and so forth. The flexibility that such systems offer has resulted in rapid growth of demand for these controllers.

1.6 Discrete and analog signals

Discrete signals behave as binary switches, yielding simply an On or Off signal (1 or 0, True or False, respectively). Push buttons, Limit switches, and photoelectric sensors are examples of devices providing a discrete signal. Discrete signals are sent using either voltage or current, where a specific range is designated as *On* and another as *Off*. For example, a PLC might use 24 V DC I/O, with values above 22 V DC representing *On*, values below 2VDC representing *Off*, and intermediate values undefined. Initially, PLCs had only discrete I/O.

Analog signals are like volume controls, with a range of values between zero and full-scale. These are typically interpreted as integer values (counts) by the PLC, with various ranges of accuracy depending on the device and the number of bits available to store the data. As PLCs typically use 16-bit signed binary processors, the integer values are limited between −32,768 and +32,767. Pressure, temperature, flow, and weight are often represented by analog signals. Analog signals can use voltage or current with a magnitude proportional to the value of the process signal. For example, an analog 0 - 10 V input or 4-20 mA would be converted into an integer value of 0 - 32767.

Current inputs are less sensitive to electrical noise (i.e. from welders or electric motor starts) than voltage inputs.

1.6.1 Example

As an example, say a facility needs to store water in a tank. The water is drawn from the tank by another system, as needed, and our example system must manage the water level in the tank by controlling the valve that refills the tank. Shown is a "ladder diagram" which shows the control system. A ladder diagram is a method of drawing control circuits which pre-dates PLCs. The ladder diagram resembles the schematic diagram of a system built with electromechanical relays. Shown are:

- Two inputs (from the low and high level switches) represented by contacts of the float switches
- An output to the fill valve, labelled as the fill valve which it controls
- An "internal" contact, representing the output signal to the fill valve which is created in the program.
- A logical control scheme created by the interconnection of these items in software

In ladder diagram, the contact symbols represent the state of bits in processor memory, which corresponds to the state of physical inputs to the system. If a discrete input is energized, the memory bit is a 1, and a "normally open" contact controlled by that bit will pass a logic "true" signal on to the next element of the ladder. **Therefore, the contacts in the PLC program that "read" or look at the physical switch contacts in this case must be "opposite" or open in order to return a TRUE for the closed physical switches.** Internal status bits, corresponding to the state of discrete outputs, are also available to the program.

In the example, the physical state of the float switch contacts must be considered when choosing "normally open" or "normally closed" symbols in the ladder diagram. The PLC has two discrete inputs from float switches (Low Level and High Level). Both float switches (normally closed) open their contacts when the water level in the tank is above the physical location of the switch.

When the water level is below both switches, the float switch physical contacts are both closed, and a true (logic 1) value is passed to the Fill Valve output. Water begins to fill the tank. The internal "Fill Valve" contact latches the circuit so that even when the "Low Level" contact opens (as the water passes the lower switch), the fill valve remains on. Since the High Level is also normally closed, water continues to flow as the water level remains between

the two switch levels. Once the water level rises enough so that the "High Level" switch is off (opened), the PLC will shut the inlet to stop the water from overflowing; this is an example of seal-in (latching) logic. The output is sealed in until a high level condition breaks the circuit. After that the fill valve remains off until the level drops so low that the Low Level switch is activated, and the process repeats again.

| (N.C. physical (N.C. physical | | Switch) Switch) | | Low Level High Level Fill Valve | |------[ ]------|------[ ]--------------------(OUT)---------| | | | | | | | | Fill Valve | | |------[ ]------| | | | |

A complete program may contain thousands of rungs, evaluated in sequence. Typically the PLC processor will alternately scan all its inputs and update outputs, then evaluate the ladder logic; input changes during a program scan will not be effective until the next I/O update. A complete program scan may take only a few milliseconds, much faster than changes in the controlled process.

Programmable controllers vary in their capabilities for a "rung" of a ladder diagram. Some only allow a single output bit. There are typically limits to the number of series contacts in line, and the number of branches that can be used. Each element of the rung is evaluated sequentially. If elements change their state during evaluation of a rung, hard-to-diagnose faults can be generated, although sometimes (as above) the technique is useful. Some implementations forced evaluation from left-to-right as displayed and did not allow reverse flow of a logic signal (in multi-branched rungs) to affect the output.

1.7 See also

- Industrial control systems
- Industrial safety systems

1.8 References

[1] E. A. Parr, *Industrial Control Handbook*, Industrial Press Inc., 1999 ISBN 0-8311-3085-7

[2] M. A. Laughton, D. J. Warne (ed), *Electrical Engineer's Reference book, 16th edition*,Newnes, 2003 Chapter 16 *Programmable Controller*

[3] "The father of invention: Dick Morley looks back on the 40th anniversary of the PLC". *Manufacturing Automation*. 12 September 2008.

[4] Harms, Toni M. & Kinner, Russell H. P.E., *Enhancing PLC Performance with Vision Systems*. 18th Annual ESD/HMI International Programmable Controllers Conference Proceedings, 1989, p. 387-399.

[5] Maher, Michael J. *Real-Time Control and Communications*. 18th Annual ESD/SMI International Programmable Controllers Conference Proceedings, 1989, p. 431-436.

[6] Kinner, Russell H., P.E. *Designing Programable Controller Application Programs Using More than One Designer*. 14th Annual International Programmable Controllers Conference Proceedings, 1985, p. 97-110.

[7] W. Bolton, *Programmable Logic Controllers, Fifth Edition*, Newnes, 2009 ISBN 978-1-85617-751-1, Chapter 1

[8] Keller, William L Jr. *Grafcet, A Functional Chart for Sequential Processes*, 14th Annual International Programmable Controllers Conference Proceedings, 1984, p. 71-96.

[9]

[10] PLC simulation reference

[11] Gregory K. McMillan, Douglas M. Considine (ed), *Process/Industrial Instruments and Controls Handbook Fifth Edition*, McGraw-Hill, 1999 ISBN 0-07-012582-1 Section 3 *Controllers*

1.9 Further reading

- Daniel Kandray, *Programmable Automation Technologies*, Industrial Press, 2010 ISBN 978-0-8311-3346-7, Chapter 8 *Introduction to Programmable Logic Controllers*

1.10 External links

- PLC Complete Tutorial
- PLC FAQ's

Chapter 2

IEC 61131-3

IEC 61131-3 is the third part (of 8) of the open international standard IEC 61131 for programmable logic controllers, and was first published in December 1993 by the IEC. The current (third) edition was published in February 2013.

Part 3 of *IEC 61131* deals with programming languages and defines two graphical and two textual PLC programming language standards:

- Ladder diagram (LD), graphical
- Function block diagram (FBD), graphical
- Structured text (ST), textual
- Instruction list (IL), textual
- Sequential function chart (SFC), has elements to organize programs for sequential and parallel control processing.

2.1 Data types

- Bit Strings - groups of on/off values
  - BOOL - 1 bit
  - BYTE - 8 bit
  - WORD - 16 bit
  - DWORD - 32 bit
  - LWORD - 64 bit
- INTEGER - whole numbers

(Considering byte size 8 bits)

-
  - SINT - signed short (1 byte)
  - INT - signed integer (2 byte)
  - DINT - double integer (4 byte)
  - LINT - long integer (8 byte)
  - U - Unsigned - prepend a U to the type to make it unsigned integer.
- REAL - floating point IEC 60559 (same as IEEE 754-2008)
  - REAL - (4 byte)
  - LREAL - (8 byte)
- TIME - duration for timers, processes.
- Date and Time of day:
  - DATE - calendar date
  - TIME_OF_DAY - clock time
  - DATE_AND_TIME: time and date
- STRING - character strings surrounded by single quotes. Escaped characters are preceded by a dollar sign.
-
  - WSTRING - holds multi-byte strings.
- Arrays - multiple values stored in the same variable.
- Sub Ranges - puts limits on value i.e., (4-20) for current
- Derived - type derived from one of the above types.
  - TYPE - single type
  - STRUCT - composite of several variables and types.
- Generic - groups of the above types:
  - ANY
    - ANY_DERIVED
    - ANY_ELEMENTARY
    - ANY_MAGNITUDE
      - ANY_NUM - LREAL, REAL
      - ANY_INT - LINT, DINT, INT, SINT, ULINT, UDINT, UINT, USINT
    - ANY_BIT - LWORD, DWORD, WORD, BYTE, BOOL
    - ANY_STRING - STRING, WSTRING
    - ANY_DATE - DATE, TOD, DT

2.2 Variables

Variable attributes: RETAIN, CONSTANT, AT

- Global
- Direct (local)
- I/O Mapping - Input, Output, I/O
- External
- Temporary

2.3 Configuration

- Resource - Like a CPU
- Tasks - Can be multiple per CPU.
- Programs - Can be executed once, on a timer, on an event.

2.4 Program organization units (POU)

- Functions
  - Standard: ADD, SQRT, SIN, COS, GT, MIN, MAX, AND, OR, etc.
  - Custom
- Function Blocks
  - Standard:
  - Custom - Libraries of functions can be supplied by a vendor or third party.
- Programs

2.5 Configuration, resources, tasks

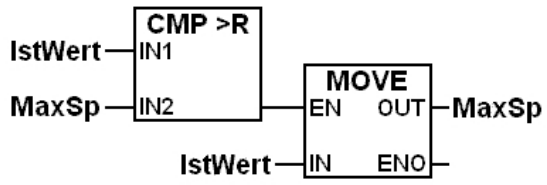
- Configuration - processing resources, memory for IO, execution rates, number of tasks.

2.6 External links

- Standard 61131-3 on IEC website

Chapter 3

Function block diagram



Simple function block diagram

The **Function Block Diagram** (**FBD**) is a graphical language for programmable logic controller design,[1] that can describe the function between input variables and output variables. A function is described as a set of elementary blocks. Input and output variables are connected to blocks by connection lines.

Inputs and outputs of the blocks are wired together with connection lines, or links. Single lines may be used to connect two logical points of the diagram:

- An input variable and an input of a block
- An output of a block and an input of another block
- An output of a block and an output variable

The connection is oriented, meaning that the line carries associated data from the left end to the right end. The left and right ends of the connection line must be of the same type.

Multiple right connection, also called divergence can be used to broadcast information from its left end to each of its right ends. All ends of the connection must be of the same type.

Function Block Diagram is one of five languages for logic or control configuration[2] supported by standard IEC 61131-3 for a control system such as a Programmable Logic Controller (PLC) or a Distributed Control System (DCS). The other supported languages are ladder logic, sequential function chart, structured text, and instruction list.

3.1 See also

- Functional block diagram
- Programmable logic controller

3.2 References

[1] R. W Lewis (2001) *Modelling Distributed Control Systems Using IEC 61499*. p. 9

[2] W. Bolton (2011) *Programmable Logic Controllers*. p. 14

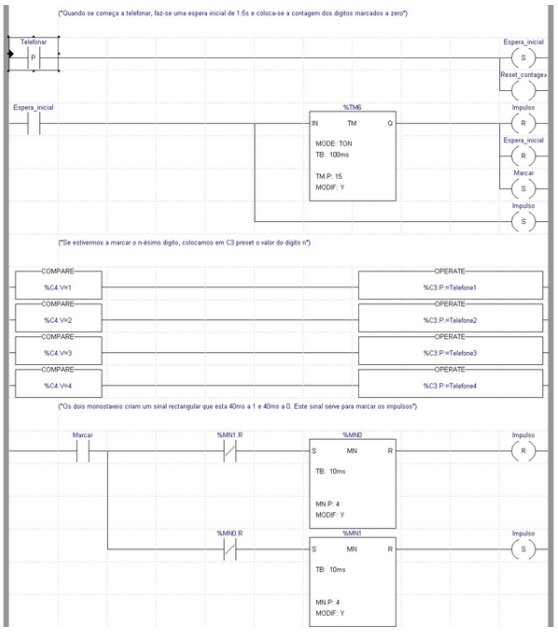
Chapter 4

Ladder logic

This article is about the programming language. For the FIRST competition, see Ladder Logic.

Ladder logic was originally a written method to document the design and construction of relay racks as used in manufacturing and process control. Each device in the relay rack would be represented by a symbol on the ladder diagram with connections between those devices shown. In addition, other items external to the relay rack such as pumps, heaters, and so forth would also be shown on the ladder diagram. See relay logic. Although the diagrams themselves have been used since the days when logic could only be implemented using switches and electromechanical relays, the term 'ladder logic' was only latterly adopted with the advent of solid state programmable logic.

Ladder logic has evolved into a programming language that represents a program by a graphical diagram based on the circuit diagrams of relay logic hardware. Ladder logic is used to develop software for programmable logic controllers (PLCs) used in industrial control applications. The name is based on the observation that programs in this language resemble ladders, with two vertical rails and a series of horizontal rungs between them. While ladder diagrams were once the only available notation for recording programmable controller programs, today other forms are standardized in IEC 61131-3.



Part of a ladder diagram, including contacts and coils, compares, timers and monostable multivibrators

4.1 Overview

Ladder logic is widely used to program PLCs, where sequential control of a process or manufacturing operation is required. Ladder logic is useful for simple but critical control systems or for reworking old hardwired relay circuits. As programmable logic controllers became more sophisticated it has also been used in very complex automation systems. Often the ladder logic program is used in conjunction with an HMI program operating on a computer workstation.

The motivation for representing sequential control logic in a ladder diagram was to allow factory engineers and technicians to develop software without additional training to learn a language such as FORTRAN or other general purpose computer language. Development, and maintenance, was simplified because of the resemblance to familiar relay hardware systems.[1] Implementations of ladder logic have characteristics, such as sequential execution and support for control flow features, that make the analogy to hardware somewhat inaccurate. This argument has become less relevant given that most ladder logic programmers have a software background in more conventional programming languages.

Manufacturers of programmable logic controllers generally also provide associated ladder logic programming systems. Typically the ladder logic languages from two manufacturers will not be completely compatible; ladder logic is better thought of as a set of closely related programming languages rather than one language. (The IEC 61131-3 standard has helped to reduce unnecessary differences, but translating programs between systems still requires significant work.) Even different models of programmable controllers within the same family may have different ladder notation such that programs cannot be

seamlessly interchanged between models.

Ladder logic can be thought of as a rule-based language rather than a procedural language. A "rung" in the ladder represents a rule. When implemented with relays and other electromechanical devices, the various rules "execute" simultaneously and immediately. When implemented in a programmable logic controller, the rules are typically executed sequentially by software, in a continuous loop (scan). By executing the loop fast enough, typically many times per second, the effect of simultaneous and immediate execution is achieved, if considering intervals greater than the "scan time" required to execute all the rungs of the program. Proper use of programmable controllers requires understanding the limitations of the execution order of rungs.

4.2 Example of a simple ladder logic program

The language itself can be seen as a set of connections between logical checkers (contacts) and actuators (coils). If a path can be traced between the left side of the rung and the output, through asserted (true or "closed") contacts, the rung is true and the output coil storage bit is asserted (1) or true. If no path can be traced, then the output is false (0) and the "coil" by analogy to electromechanical relays is considered "de-energized". The analogy between logical propositions and relay contact status is due to Claude Shannon.

Ladder logic has contacts that make or break circuits to control coils. Each coil or contact corresponds to the status of a single bit in the programmable controller's memory. Unlike electromechanical relays, a ladder program can refer any number of times to the status of a single bit, equivalent to a relay with an indefinitely large number of contacts.

So-called "contacts" may refer to physical ("hard") inputs to the programmable controller from physical devices such as pushbuttons and limit switches via an integrated or external input module, or may represent the status of internal storage bits which may be generated elsewhere in the program.

Each rung of ladder language typically has one coil at the far right. Some manufacturers may allow more than one output coil on a rung.

- —( )— A regular coil, energized whenever its rung is closed.
- —(\)— A "not" coil, energized whenever its rung is open.
- —[ ]— A regular contact, closed whenever its corresponding coil or an input which controls it is energized.
- —[\]— A "not" contact, closed whenever its corresponding coil or an input which controls it is not energized.

The "coil" (output of a rung) may represent a physical output which operates some device connected to the programmable controller, or may represent an internal storage bit for use elsewhere in the program.

4.2.1 Logical AND

------[ ]--------------[ ]----------------( ) Key Switch 1 Key Switch 2 Door Motor

The above realizes the function: Door Motor = Key Switch 1 AND Key Switch 2

This circuit shows two key switches that security guards might use to activate an electric motor on a bank vault door. When the normally open contacts of both switches close, electricity is able to flow to the motor which opens the door.

4.2.2 Logical AND with NOT

------[ ]--------------[\]----------------( ) Close Door Obstruction Door Motor

The above realizes the function: Door Motor = Close door AND NOT(Obstruction).

This circuit shows a pushbutton that closes a door, and an obstruction detector that senses if something is in the way of the closing door. When the normally open pushbutton contact closes and the normally closed obstruction detector is closed (no obstruction detected), electricity is able to flow to the motor which closes the door.

4.2.3 Logical OR

--+-------[ ]-------+----------------( ) | Exterior Unlock | Unlock | | +-------[ ]-------+ Interior Unlock

The above realizes the function: Unlock = Interior Unlock OR Exterior Unlock

This circuit shows the two things that can trigger a car's power door locks. The remote receiver is always powered. The lock solenoid gets power when either set of contacts is closed.

4.2.4 Industrial STOP/START

In common industrial latching start/stop logic we have a "start" button to turn on a motor contactor, and a "stop" button to turn off the contactor.

When the "start" button is pushed the input goes true, via the "stop" button NC contact. When the "run" in-

put becomes true the seal-in "run" NO contact in parallel with the "start" NO contact will close maintaining the input logic true (latched or sealed-in). After the circuit is latched the "stop" button may be pushed causing its NC contact to open and consequently the input to go false. The "run" NO contact then opens and the circuit logic returns to its quiescent state.

--+----[ ]--+----[\]----( ) | start | stop run | | +----[ ]--+ run -------[ ]--------------( ) run motor

The above realizes the function: run = (start OR run) AND (NOT stop)

> Note the use of parenthesis to group the logical OR function before evaluating the logical AND function (which has a higher order of operation priority). Also note the use of **NOT** to represent the "stop" NC contact logic.

This latch configuration is a common idiom in ladder logic. In ladder logic it is referred to as seal-in logic. The key to understanding the latch is in recognizing that "start" switch is a momentary switch (once the user releases the button, the switch is open again). As soon as the "run" solenoid engages, it closes the "run" NO contact, which latches the solenoid on. The "start" switch opening up then has no effect.

For safety reasons, an Emergency-Stop and/or Stop should be hardwired in series with the Start switch, and the relay logic should reflect this.

--[\]----[\]----+--[ ]--+---------( ) ES Stop | Start | Motor | | +--[ ]--+ Run

4.2.5 Complex logic

Here is an example of what two rungs in a ladder logic program might look like. In real world applications, there may be hundreds or thousands of rungs.

Typically, complex ladder logic is 'read' left to right and top to bottom. As each of the lines (or rungs) are evaluated the output coil of a rung may feed into the next stage of the ladder as an input. In a complex system there will be many "rungs" on a ladder, which are numbered in order of evaluation.

1. ----[ ]---------+----[ ]-----+----( ) Switch | HiTemp | A/C | | +----[ ]-----+ Humid 2. ----[ ]----[\]----------( ) A/C Heat Cooling

Line 1 realizes the function: A/C = Switch AND (HiTemp OR Humid)

Line 2 realizes the function: Cooling = A/C AND (NOT Heat)

This represents a slightly more complex system for rung 2. After the first line has been evaluated, the output coil "A/C" is fed into rung 2, which is then evaluated and the output coil "Cooling" could be fed into an output device "Compressor" or into rung 3 on the ladder. This system allows very complex logic designs to be broken down and evaluated.

4.2.6 Additional functionality

Additional functionality can be added to a ladder logic implementation by the PLC manufacturer as a special block. When the special block is powered, it executes code on predetermined arguments. These arguments may be displayed within the special block.

+-------+ -----[ ]--------------------+ A +---- Remote Unlock +-------+ Remote Counter +-------+ -----[ ]-------------------+ B +---- Interior Unlock +-------+ Interior Counter +--------+ --------------------+ A + B +---------- | into C | +--------+ Adder

In this example, the system will count the number of times that the interior and remote unlock buttons are pressed. This information will be stored in memory locations A and B. Memory location C will hold the total number of times that the door has been unlocked electronically.

PLCs have many types of special blocks. They include timers, arithmetic operators and comparisons, table lookups, text processing, PID control, and filtering functions. More powerful PLCs can operate on a group of internal memory locations and execute an operation on a range of addresses, for example,to simulate a physical sequential drum controller or a finite state machine. In some cases, users can define their own special blocks, which effectively are subroutines or macros. The large library of special blocks along with high speed execution has allowed use of PLCs to implement very complex automation systems.

4.3 Limitations and successor languages

Ladder notation is best suited to control problems where only binary variables are required and where interlocking and sequencing of binary is the primary control problem. Since execution of rungs is sequential within a program and may be undefined or obscure within a rung, some logic race conditions are possible which may produce unexpected results; complex rungs are best broken into several simpler steps to avoid this problem. Some manufacturers avoid this problem by explicitly and completely defining the execution order of a rung, however programmers may still have problems fully grasping the resulting complex semantics.

Analog quantities and arithmetical operations are clumsy to express in ladder logic and each manufacturer has different ways of extending the notation for these problems. There is usually limited support for arrays and loops, of-

ten resulting in duplication of code to express cases which in other languages would call for use of indexed variables.

As microprocessors have become more powerful, notations such as sequential function charts and function block diagrams can replace ladder logic for some limited applications. Very large programmable controllers may have all or part of the programming carried out in a dialect that resembles BASIC or C or other programming language with bindings appropriate for a real-time application environment.

4.4 See also

- Programmable logic controller
- Digital circuit
- IEC 61131

4.5 References

[1] Edward W. Kamen *Industrial Controls and Manufacturing*, (Academic Press, 1999) ISBN 0123948509, Chapter 8 *Ladder Logic Diagrams and PLC Implementations*

4.6 External links

- Beginners Ladder Logic
- Beginners Ladder Logic Primer
- Basic Ladder Logic
- "Chapter 6: ladder logic" by Tony R. Kuphaldt
- Ladder Logic Programming Examples

Chapter 5

Structured text

Structured text is one of the five languages supported by the IEC 61131-3 standard, designed for programmable logic controllers (PLCs). It is a high level language that is block structured and syntactically resembles Pascal, on which it is based. All of the languages share IEC61131 Common Elements. The variables and function calls are defined by the common elements so different languages within the IEC 61131-3 standard can be used in the same program.

Complex statements and nested instructions are supported:

- Iteration loops (REPEAT-UNTIL; WHILE-DO)
- Conditional execution (IF-THEN-ELSE; CASE)
- Functions (SQRT(), SIN())

5.1 Sample program

(* simple state machine *) TxtState := STATES[StateMachine]; CASE StateMachine OF 1: ClosingValve(); ELSE ;; BadCase(); END_CASE;

5.1.1 Additional ST programming examples

// PLC configuration CONFIGURATION DefaultCfg VAR_GLOBAL b_Start_Stop : BOOL; // Global variable to represent a boolean. b_ON_OFF : BOOL; // Global variable to represent a boolean. Start_Stop AT %IX0.0:BOOL; // Digital input of the PLC (Address 0.0) ON_OFF AT %QX0.0:BOOL; // Digital output of the PLC (Address 0.0). (Coil) END_VAR // Schedule the main program to be executed every 20 ms TASK Tick(INTERVAL := t#20ms); PROGRAM Main WITH Tick : Monitor_Start_Stop; END_CONFIGURATION PROGRAM Monitor_Start_Stop // Actual Program VAR_EXTERNAL Start_Stop : BOOL; ON_OFF : BOOL; END_VAR VAR // Temporary variables for logic handling ONS_Trig : BOOL; Rising_ONS : BOOL; END_VAR // Start of Logic // Catch the Rising Edge One Shot of the Start_Stop input ONS_Trig := Start_Stop AND NOT Rising_ONS; // Main Logic for Run_Contact -- Toggle ON / Toggle OFF --- ON_OFF := (ONS_Trig AND NOT ON_OFF) OR (ON_OFF AND NOT ONS_Trig); // Rising One Shot logic Rising_ONS := Start_Stop; END_PROGRAM

Function block example

//==
// Function Block Timed Counter : Incremental count of the timed interval
//==
FUNCTION_BLOCK FB_Timed_Counter VAR_INPUT Execute : BOOL := FALSE; // Trigger signal to begin Timed Counting Time_Increment : REAL := 1.25; // Enter Cycle Time (Seconds) between counts Count_Cycles : INT := 20; // Number of Desired Count Cycles END_VAR VAR_OUTPUT Timer_Done_Bit : BOOL := FALSE; // One Shot Bit indicating Timer Cycle Done Count_Complete : BOOL := FALSE; // Output Bit indicating the Count is complete Current_Count : INT := 0; // Accumulating Value of Counter END_VAR VAR CycleTimer : TON; // Timer FB from Command Library CycleCounter : CTU; // Counter FB from Command Library TimerPreset : TIME; // Converted Time_Increment in Seconds to MS END_VAR // Start of Function Block programming TimerPreset := REAL_TO_TIME(in := Time_Increment) * 1000; CycleTimer(in := Execute AND NOT CycleTimer.Q ,pt := TimerPreset); Timer_Done_Bit := CycleTimer.Q; CycleCounter(cu := CycleTimer.Q ,r := NOT Execute ,pv := Count_Cycles); Current_Count := CycleCounter.cv; Count_Complete := CycleCounter.q; END_FUNCTION_BLOCK

5.2 References

Chapter 6

Instruction list

Instruction List (**IL**) is one of the 5 languages supported by the IEC 61131-3 standard. It is designed for programmable logic controllers (PLCs). It is a low level language and resembles assembly. All of the languages share IEC61131 Common Elements. The variables and function call are defined by the common elements so different languages can be used in the same program.

Program control (control flow) is achieved by *jump* instructions and function calls (subroutines with optional parameters).

The file format has now been standardized to XML by PLCopen.

6.1 Example

LD Speed GT 1000 JMPCN VOLTS_OK LD Volts VOLTS_OK LD 1 ST %Q75

6.2 Variations from IEC61131

Many vendors whilst incorporating the full IEC61131 requirements have additional vendor specific calls/function blocks to suit their hardware such as reading or writing to I/O. Siemens PLC instruction list language is known as "Statement List" or "STL" in English, and "Anweisungs-Liste" or "AWL" in German, Italian and Spanish. The user of a Simatic development package may choose between German and International mnemonics to represent instructions. For example "A" for "AND" or "U" for "UND", "I" for "Input" or "E" for "Eingang".

6.3 See also

- Programmable Logic Controller

Chapter 7

Sequential function chart

Sequential function chart (**SFC**) is a graphical programming language used for programmable logic controllers (PLCs). It is one of the five languages defined by IEC 61131-3 standard. The SFC standard is defined as, *Preparation of function charts for control systems*, and was based on GRAFCET (itself based on binary petri nets [1]). [2]).

It can be used to program processes that can be split into steps.

Main components of SFC are:

- Steps with associated actions;
- Transitions with associated logic conditions;
- Directed links between steps and transitions.

Steps in an SFC diagram can be active or inactive. Actions are only executed for active steps. A step can be active for one of two motives:

- It is an initial step as specified by the programmer.
- It was activated during a scan cycle and not deactivated since.

Steps are activated when all steps above it are active and the connecting transition is superable (i.e. its associated condition is true). When a transition is passed, all steps above are deactivated at once and **after** all steps below are activated at once.

Actions associated with steps can be of several types, the most relevant ones being Continuous (N), Set (S) and Reset (R). Apart from the obvious meaning of Set and Reset, an N action ensures that its target variable is set to 1 as long as the step is active. An SFC rule states that if two steps have an N action on the same target, the variable must never be reset to 0. It is also possible to insert LD (Ladder Diagram) actions inside an SFC program (and this is the standard way, for instance, to work on integer variables).

SFC is an inherently parallel language in that multiple control flows (POUs in the standard's parlance) can be active at once.

Non-standard extensions to the language include macroactions: i.e. actions inside a program unit that influence the state of another program unit. The most relevant such macroaction is "forcing", in which a POU can decide the active steps of another POU.

7.1 References

[1] Fernandez, J. L.; Sanz, R.; Paz, E.; Alonso, C. (19–23 May 2008). "Using hierarchical binary Petri nets to build robust mobile robot applications: RoboGraph". "IEEE International Conference on Robotics and Automation, 2008.". Pasadena, CA, USA. pp. 1372–1377. doi:10.1109/ROBOT.2008.4543394.

[2] Lewis, R. W. *Programming industrial control systems using IEC 1131-3*.

7.2 Text and image sources, contributors, and licenses

7.2.1 Text

- **Programmable logic controller** *Source:* http://en.wikipedia.org/wiki/Programmable_logic_controller?oldid=628752621 *Contributors:* Amillar, Christian List, PierreAbbat, Deb, Ray Van De Walker, TomCerul, Michael Hardy, Modster, Kku, CesarB, Ahoerstemeier, Stan Shebs, Ronz, Yaronf, Darkwind, Ugen64, Glenn, GRAHAMUK, Mydogategodshat, Gepwiki, Patden, Mrand, Wernher, AnthonyQBachler, RedWolf, Texture, Pengo, Tobias Bergemann, Giftlite, Tom harrison, Ssd, Jfdwolff, Utcursch, Ot, Sonett72, Fuzlyssa, Andreas Kaufmann, Mike Rosoft, Diagonalfish, Discospinster, Rich Farmbrough, ArnoldReinhold, Vev, Harriv, Kakesson, CanisRufus, El C, Shanes, Spalding, John Vandenberg, (aeropagitica), Hooperbloob, Mdd, Alansohn, Gary, Wdfarmer, Hammertime, Wtmitchell, Velella, Wtshymanski, Suruena, Redvers, Woohookitty, WadeSimMiser, MFH, Kralizec!, Mandarax, Kbdank71, Jorunn, Rjwilmsi, Sarg, Utuado, FlaBot, Emarsee, Ldwolk, Lmatt, Srleffler, Vijayamurugan.p@gmail.com, Banaticus, YurikBot, Gerfriedc, Splash, Kimchi.sg, Bovineone, Wimt, Robertvan1, Dtrebbien, Aaron Brenneman, ArsénіureDeGallium, Jpbowen, TERdON, Bozoid, Syrthiss, DeadEyeArrow, Superluser, Nwk, Closedmouth, Back ache, Smurrayinchester, Benandorsqueaks, XAVeRY, CIreland, ChemGardener, Attilios, SmackBot, Mje112, Mitchan, Lds, Collieman, Shai-kun, Gilliam, JMiall, KD5TVI, Bluebot, Jprg1966, Tripledot, Baa, DHN-bot, Can't sleep, clown will eat me, Sevicke, VMS Mosaic, Pax85, Jwy, Kntrabssi, Vina-iwbot, Rahul.mishra, Gstortz, Skyscrap27, Kuru, JethroElfman, Nharipra, ML5, Robofish, Minna Sora no Shita, DabMachine, Iridescent, JoeBot, Shoeofdeath, IvanLanin, Lakee911, HaPi, LessHeard vanU, VGarner, CmdrObot, Ale jrb, Drinibot, PurpleChez, Netbymatt, Requestion, A876, Yaamboo, Meno25, DumbBOT, Zalgo, Thijs!bot, Epbr123, D4g0thur, N5iln, Oliver202, Marek69, Wildthing61476, JustAGal, AntiVandalBot, JimScott, Seaphoto, Prolog, Credema, Jcipc2004, Dougher, V3co, Steelpillow, Golgofrinchian, Radwell International, PhilKnight, SiobhanHansa, Elmschrat, Ljudina, Bongwarrior, VoABot II, Loyt, Ishi Gustaedr, Think outside the box, Doug Coldwell, Tedickey, Morphaeous, Email4mobile, Crunchy Numbers, LorenzoB, User A1, Martynas Patasius, Brian Radwell, Ripogenus77, DerHexer, Dailynetworks, Conquerist, Monkan, Microsp, Jim.henderson, Tholly, J.delanoy, Bin95, Maurice Carbonaro, Frapacino, Ctdmrod06010, Jerry, Tasnai, Katalaveno, SJP, Cmichael, RB972, Mike V, Pdcook, MkClark, VolkovBot, Jeff G., Supersteve04038, Ipso2, Philip Trueman, MrRK, TXiKiBoT, Burpen, Xhantar, Rei-bot, Sean D Martin, Broadbot, CanOfWorms, Jackfork, Ilyushka88, Isis4563, Krushia, Billinghurst, Andy Dingley, CrackleBot, Finalreminder, Cjmulvey, Ponyo, SieBot, Dlmackey, Nubiatech, Poliyal, Euryalus, KFullerton, Tiptoety, Oxymoron83, Lightmouse, OKBot, Fbolanos, Pointbonita, Denisarona, Lcwk86, Chris.rickey, ClueBot, Limebite, Rodhullandemu, Uncle Milty, CounterVandalismBot, Puchiko, Excirial, Razorflame, SchreiberBike, BOTarate, Skipperalfie, GrimmReaperSound, Versus22, Lambtron, BlueDevil, SoxBot III, XLinkBot, Stickee, Gerhardvalentin, Avoided, Charles Sturm, Skarebo, SilvonenBot, NellieBly, Control.optimization, Addbot, Mortense, Some jerk on the Internet, Mabdul, Tech Jockey, TutterMouse, Fieldday-sunday, CanadianLinuxUser, Cst17, MrOllie, Chamal N, Smhaatif, Debresser, Gary2001, OlEnglish, Teles, Gail, Loupeter, Yaseen805, Namoson, Luckas-bot, Yobot, Derectus, DJ LoPaTa, MrBlueSky, SteffiKl, Peter Flass, AnomieBOT, Ciphers, Pratik rathore, Jim1138, K4rmix, Materialscientist, Straight-shooter(42), ArthurBot, Xqbot, Athabaska-Clearwater, YakiraLight, MahdiEynian, Turk oğlan, Tatau1234, Shirik, RibotBOT, Gg automation, Shadowjams, Dougofborg, FrescoBot, Joe.dolivo, Injeek6, Cartiman, Alexandermorgantop, Mr.TidePubli, Novaseminary, SUL, Pelcer, Rkinner, Calmer Waters, RedBot, Yosyk07, ВикиКорректор, Newt Winkler, North8000, Theo10011, YouAndMeBabyAintNothingButCamels, Hughjack, Alperkaradas, DARTH SIDIOUS 2, Jfmantis, The Utahraptor, Tho2468, Midhart90, Ericmortenson, EmausBot, John of Reading, Tuankiet65, WikitanvirBot, Mixabest, MartyRobar, Solarra, Ladybetty, Wikipelli, Neilyboy4306, Fæ, Josve05a, Prdejong, TcomptonMA, Shible.isteak, Passion of Knowledge, MCorley, RoyKok, Bircwiki, 28bot, Rocketrod1960, Goatboy22, Petrb, ClueBot NG, Satellizer, Bped1985, DieSwartzPunkt, O.Koslowski, Anaveenraj31, Ryan Vesey, Scantime, Helpful Pixie Bot, Pdfsupply, Toffanin, Gurt Posh, Lindsey Sh, Wiki13, Maxhitchens, Allecher, Canoe1967, Raldrich123, MaartenMJR, Plc training, PLClogistics, Jama555, Sanjivee, Snow Blizzard, Afshan5zeba, Brent s plc, Otus scops, Rwgambill, Anbu121, Baka610, Justincheng12345-bot, Moaz786, Maddensue, Mahmdmuhy, ChrisGualtieri, Tech77, Toronto30, Rolfds, Sidelight12, Frosty, Aotewell, Maryeputnam, Safety-Hero, Shariat ipe, Cmariella, BIN95, Napy65, Leroy843, Jianhui67, Jsczurek, Lancie01, Natrajprasanna, Plctrainingin, Mal.hetherington and Anonymous: 780
- **IEC 61131-3** *Source:* http://en.wikipedia.org/wiki/IEC_61131-3?oldid=621208060 *Contributors:* Pengo, Andreas Kaufmann, Pgquiles, Diego Moya, Daira Hopwood, Lmatt, Raggha, ArsénіureDeGallium, Moe Epsilon, Radagast83, Neelix, AndrewHowse, Loyt, Windymilla, STBot, Pekaje, VolkovBot, Krushia, WRK, ClueBot, Wlydon, Kolyma, Mitch Ames, Dekart, MystBot, Addbot, MrOllie, SteffiKl, Rubinbot, Mopza, SassoBot, Dinhxuanduyet, Louperibot, Csciec, MD.KW, Christian Ettinger, Josve05a, KineticaRT, Mentibot, ChuispastonBot, Hmainsbot1 and Anonymous: 40
- **Function block diagram** *Source:* http://en.wikipedia.org/wiki/Function_block_diagram?oldid=580425168 *Contributors:* Gadfium, Mdd, Yobot and Anonymous: 1
- **Ladder logic** *Source:* http://en.wikipedia.org/wiki/Ladder_logic?oldid=624142699 *Contributors:* The Anome, TomCerul, Altenmann, Pengo, Tobias Bergemann, Brouhaha, DavidCary, Zigger, Gadfium, Sam Hocevar, Hugh Mason, Discospinster, Scampiandchips, Mjager, Mdd, Frank101, Cgmusselman, Wtshymanski, Emvee, Suruena, Bookandcoffee, Bushytails, Daira Hopwood, Cbdorsett, Lazynitwit, Boteman, FlaBot, Lmatt, JiVE, DVdm, Wavelength, Lexi Marie, Wilfried Elmenreich, Gaius Cornelius, DragonHawk, Thalter, Mugunth Kumar, Imaninjapirate, TBadger, Bruyninc, SmackBot, Eskimbot, Ordinant, Commander Keane bot, Russvdw, JoelWhy, Nmnogueira, Dudecon, Ninjayeti, Rajkosto, VdSV9, BananaFiend, Wm Seán Glen, CRGreathouse, Requestion, P.N, Malleus Fatuorum, Guy Macon, Rehnn83, Tarkaan, I B Wright, Ken g6, Rei-bot, One half 3544, Andy Dingley, Synthebot, Dj Adz, Phmoreno, Jooplefers, Yngvarr, SieBot, Jharris693, Carl142, Miremare, Masgatotkaca, ClueBot, Lambtron, Addbot, MrOllie, MrBlueSky, AnomieBOT, CBMalloch, Kdefoor, Nameless23, Rackmount-guy, RedBot, Toolnut, Jfmantis, AndyHe829, EmausBot, Racerx11, Plcancircuitdia, PhanuelB, Chris857, Tszhangto, Adamjgreen, ClueBot NG, Jeff Song, DieSwartzPunkt, Antiqueight, Pentagonpie, ChrisGualtieri, Maryeputnam, Langefj, SamizdataPrime, Leroy843 and Anonymous: 131
- **Structured text** *Source:* http://en.wikipedia.org/wiki/Structured_text?oldid=618591038 *Contributors:* Karol Langner, Discospinster, Toussaint, FlaBot, YurikBot, Mahahahaneapneap, Zwobot, Cedar101, Schmiteye, Bluebot, Jimwelch, JoaquinFerrero, Dougher, PhilKnight, Magioladitis, Andy Dingley, Addbot, Mortense, Rubinbot, Sae1962, DrilBot and Anonymous: 16
- **Instruction list** *Source:* http://en.wikipedia.org/wiki/Instruction_list?oldid=619212404 *Contributors:* Pengo, Karol Langner, Lmatt, Schmiteye, Bluebot, Jimwelch, STBot, Ziounclesi, SheffGruff, Krushia, Addbot, TechBot, Nameless23, ClueBot NG, Makecat-bot, Lugia2453 and Anonymous: 14
- **Sequential function chart** *Source:* http://en.wikipedia.org/wiki/Sequential_function_chart?oldid=630271923 *Contributors:* Pnm, Mbschenkel, Bearcat, Beland, Karol Langner, Pgquiles, Pearle, Ruud Koot, Lmatt, Romanc19s, BirgitteSB, ArsénіureDeGallium, Petertheking, SmackBot, Saihtam, CmdrObot, Suzannejb, E-s-B, Dougher, Magioladitis, Dbaechtel, One half 3544, VVVBot, ImageRemovalBot, Dekart, Kbkane, Addbot, David dawkins, Denispir, LucienBOT, Saehrimnir, Teuxe, RjwilmsiBot and Anonymous: 21

7.2.2 Images

- **File:BMA_Automation_Allen_Bradley_PLC_3.JPG** *Source:* http://upload.wikimedia.org/wikipedia/commons/2/25/BMA_Automation_Allen_Bradley_PLC_3.JPG *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* Elmschrat Coaching-Blog
- **File:Commons-logo.svg** *Source:* http://upload.wikimedia.org/wikipedia/en/4/4a/Commons-logo.svg *License:* ? *Contributors:* ? *Original artist:* ?
- **File:FBS_Maximum.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/d0/FBS_Maximum.jpg *License:* CC0 *Contributors:* Own work *Original artist:* JLes
- **File:HelloWorld.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/2/28/HelloWorld.svg *License:* Public domain *Contributors:* Own work *Original artist:* Wooptoo
- **File:Ladder_diagram.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/8/83/Ladder_diagram.png *License:* CC-BY-SA-2.5 *Contributors:* Own work *Original artist:* Nuno Nogueira (w:User:Nmnogueira)
- **File:PLCLogix_sample.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/48/PLCLogix_sample.jpg *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* 38chad
- **File:PLC_Control_Panel.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/42/PLC_Control_Panel.png *License:* Public domain *Contributors:* Originally from en.wikipedia; description page is/was here. *Original artist:* Original uploader was Dailynetworks at en.wikipedia
- **File:Question_book-new.svg** *Source:* http://upload.wikimedia.org/wikipedia/en/9/99/Question_book-new.svg *License:* ? *Contributors:* Created from scratch in Adobe Illustrator. Based on Image:Question book.png created by User:Equazcion *Original artist:* Tkgd2007
- **File:Siemens_Simatic_S7-416-3.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/f/f3/Siemens_Simatic_S7-416-3.jpg *License:* Public domain *Contributors:* Own work *Original artist:* Mixabest
- **File:Wikiversity-logo.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/9/91/Wikiversity-logo.svg *License:* ? *Contributors:* Snorky (optimized and cleaned up by verdy_p) *Original artist:* Snorky (optimized and cleaned up by verdy_p)

7.2.3 Content license

- Creative Commons Attribution-Share Alike 3.0