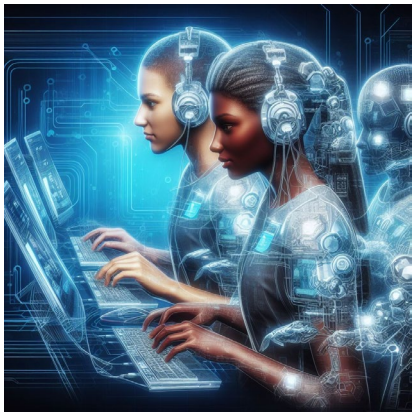
CHAPTER 4

HARDWARE AND SOFTWARE LOGIC

Lecture material for TTK 4175 Instrumentation Systems and Safety at the Department of Engineering Cybernetics, Norwegian University of Science and Technology (NTNU).

Author: Professor Mary Ann Lundteigen, Department of Engineering Cybernetics



The essence of making safe logic for hardware and software?
Illustration generated by Microsoft Copilot (powered by OpenAI), July 2025.

The content is updated regularly to improve precision and ensure relevance, which is reflected in the revision number. Please reach out to mary.a.lundteigen@ntnu.no if you have comments or suggestions for improvement.

Rev: 2.0/2026

Revision tracking (most recent)

Rev	Date	Modifications
2.0/26	01.07.2026	Updated after spring semester

Contents

4 Hardware and software logic

Industrial control and safety systems are implemented using a combination of hardwired devices and programmable controllers. Relays, contactors, and solenoid valves perform logic functions in hardware, while industrial controllers execute logic in software.

This chapter introduces the operation of relays and contactors. Regarding software, the emphasis is on IEC 61131-3, a standard that defines the programming languages relevant for industrial controllers. In practice, such functions are rarely realized solely in hardware or software; instead, industrial systems typically combine hardwired devices with application software executed by industrial controllers, as illustrated in Fig. 1.

As a concluding part, the Chapter provides insight into the factors that frame the development of software for safety-related systems, with particular emphasis on IEC 61508 and IEC 61511 (two standards covered in more detail in Chapter 9).

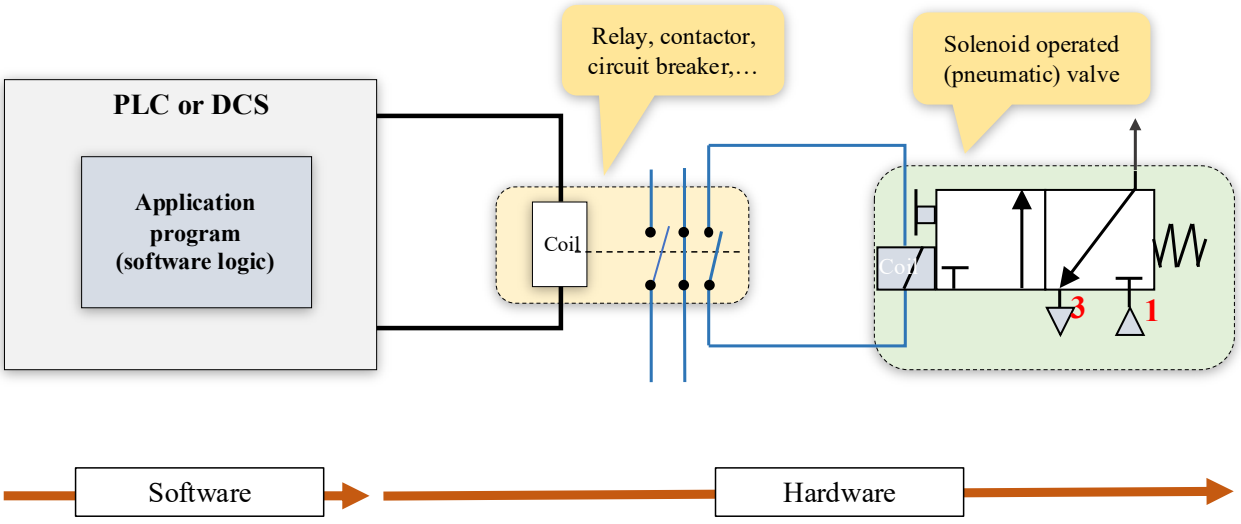


Fig. 1. Software and hardware logic, and how they are related to each other

The chapter builds on the topics in Chapter 3 (Technical documentation) on pneumatic and hydraulic control valves.

4.1 Abbreviations

DCS	Distributed control system
ESD	Emergency shutdown
FB	Function block
FC	Fail close
IL	Instruction list
LD	Ladder diagrams
NC	Normally closed
NDE	Normally de-energized
NE	Normally energized
NO	Normally open
PCS	Process control system
PLC	Programmable electronic controller
PSD	Process shutdown system
OT	Operational Technology
SFC	Sequential flow chart
SIS	Safety instrumented system

ST	Structured text

4.2 Hardwired-implemented logic

Hardware-implemented logic operations are still widely used in industry, usually in combination with industrial (programmable) controllers. It relies on utilizing relays and contactors to make or break electrical circuits, thereby controlling the power supply or signal flow to connected devices.

An advantage of hardwired logic is its high reliability. The functions it performs are typically simple enough to verify, and their failure modes are well understood. However, a key drawback is that changing hardwired logic is more labor-intensive than changing software-based solutions. Additionally, as the number of control functions increases, the wiring complexity grows rapidly, making the system harder to manage and modify.

4.2.1 Relays and contactors

Fig. 2 shows how a relay and a contactor may look, illustrating that they can differ in size and appearance. The most noticeable difference in physical appearance is their size. A relay is often thin and light, while a contactor can be a bit heavier and brick-like.

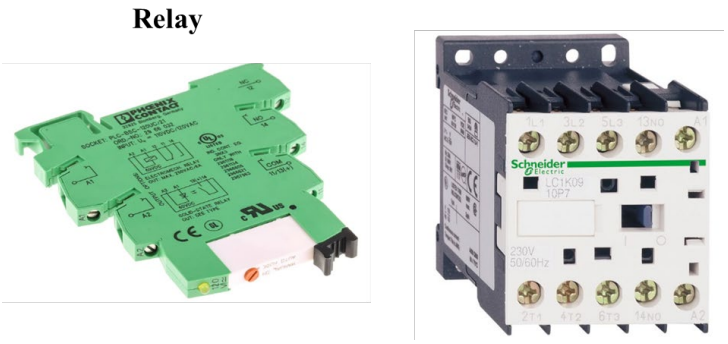


Fig. 2. Picture of a relay and a contact (there are also other variants and sizes)

The logical operations achieved by integrating the devices mentioned into electrical circuits and establishing the necessary interfaces with industrial controllers are documented in an electrical circuit diagram.

Contacts and relays provide similar types of functions, but their applications are slightly different:

- Relays are generally used for control signals or low-current switching, often in logic circuits and for galvanic isolation in on/off signal transmission.
- Contactors are designed to handle higher currents and voltages, such as those required to start motors or switch power to industrial equipment.

A related component, not defined as a relay or contactor but as a protective switching device, is the circuit breaker. Upon detecting an over- or undervoltage/current or a short circuit, the circuit breaker automatically opens its contacts to interrupt the current and prevent further damage. As such, circuit breakers may also be part of hardware-implemented logic.

There are several types of relays, and an overview is provided in Tab. 1. In this chapter, we focus on electromechanical relays and contactors, as they are the most widely used for the applications we consider. Both are considered electromechanical devices because they have electrical components (a coil) and moving mechanical parts.

Considering the electromagnetic design principle, both component types consist of two main parts:

- A coil that generates an electromagnetic field strong enough to actuate one or more sets of mechanical contacts.

- Contacts that open or close electrical circuits in response to the coil's activation.

Relays and contactors used for safety applications are designed to be force-guided, meaning they include two or more contact sets that are physically connected and cannot move independently. In this way, it is possible to obtain confirmation from multiple contacts that the switch was performed consistently or that it failed. This principle is specified in IEC 61810 (2019) a standard that has replaced EN 50205.

Logical operations in electrical circuits are performed by opening or closing contacts:

- Opening a contact corresponds to an "off" operation, as it breaks the electrical circuit and stops current flow.
- Closing a contact corresponds to an "on" operation, as it completes the circuit and allows current to flow.

Tab. 1. Relay types

Relay type	Switching function	Characteristics
Electro-magnetic relay	Switching by the combination of a coil and the electromechanical operation of switches.	• Application area logic and control operations and galvanic isolation in on/off signal transmission. • Widely used • Low cost • Also widely used for safety functions (may have safety «SIL» certification) • Response time: typically, from 5-15 ms • Can be ex-certified.
Electro-magnetic contactors	Same type of function as an electromagnetic relay, but designed for high-power/high-current. Control side: from low to high voltage/current. Load side: High voltages and/or currents	• Application area: switching electrical loads on and off, e.g., motors, heaters, and lighting. • Can be ex-certified • Response time: typically, from 10-100 ms
Reed relay	Switching is achieved by energizing two thin metal strips, «reeds,» inside a glass bulb, which causes them to attract each other and close the circuit.	Compared to electro-mechanical relays: • More costly • Response time is faster: typically 0.1-2 ms • Higher reliability • Fewer moving parts Other characteristics: • Applicable with high voltages, but low /moderate currents • Can be used for safety applications (may have safety «SIL» certification) • Smaller in size • Protected (sealed) parts – less exposed to degradation and «explosion-proof». • Considered more reliable (than electromagnetic ones) • Can be ex-certified.
Solid-state relays	Switching is achieved by operating semiconductors.	• No moving parts • Can be applied with low voltage/current as well as high voltage/currents, depending on design

		principle: e.g., MEMS, MOSFET, TRIAC. • Not considered as a contactor (as a contactor relates more to a specific application area) • Can be ex-certified.

Not covered in this table are specific overload protection devices. While traditional overload relays are mounted separately, some modern integrated motor starters combine both the contactor and overload protection into a single product.

4.2.2 Combinations of control signals (NE, NDE) and contact types (NO, NC)

Contacts can be designed as either:

- Normally closed (NC)
- Normally open (NO)

"Normal" refers to the contact position when no control signal is applied, i.e., its rest position. A relay may have one or more contact sets, each of which is either NC or NO.

The control signal can be applied in two different ways:

- Normally Energized (NE)
- Normally De-energized (NDE)

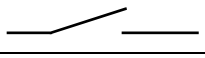
In this context, "energized" means that current is flowing through the coil of a relay or contactor. The term "normal" refers to the expected state of the control signal during regular, steady-state operation; i.e., the signal is continuously applied unless an abnormal condition occurs or the system intentionally removes it.

It is important to note that the term "normal" has been used with two different meanings depending on the context:

- For contacts, "normal" refers to their default physical state when no control signal is applied (e.g., normally open or normally closed).
- For control signals, "normal" refers to the expected operational condition, i.e., whether the signal is typically applied or not applied during regular system operation.

The resulting states of the contacts—determined by the type of contact and the nature of the control signal—are summarized in Tab. 2.

Tab. 2. Contacts and control signals

Contact type (rest/unpowered position)	**Control signal (coil) – during normal operation**		**Symbol**
	Normally Energized (NE)	Normally Deenergized (NDE)	
Normally closed (NC)	Open	Closed	
Normally open (NO)	Closed	Open	

The following two examples help illustrate typical behavior:

- A normally closed (NC) contact combined with a normally energized (NE) control signal results in the contact being open during regular operation. If the control signal is lost or removed, the contact returns to its closed state due to its mechanical force.

- A normally open (NO) contact combined with a normally energized (NE) control signal means the contact is closed during regular operation. If the control signal is lost, the contact returns to its open state.

Which combinations of contact and control signal types should be used? The choice depends on the specific application and safety requirements. Here are two common scenarios:

- Normally Energized (NE) control signals are often used to achieve fail-safe behavior. In this setup, if the control signal is lost (e.g., due to a power failure or broken wire), the contact returns to its default (rest) position. The contact type—normally closed (NC) or normally open (NO)—is selected based on the system's safe state.
    - Example: An emergency stop button typically uses an NE signal with NO contact. If the control signal is lost, the contact opens, removing power from the machine, causing it to stop automatically.
- Normally De-Energized (NDE) control signals may be appropriate in systems where there is no single, clearly defined safe state. In such cases, whether a function is safe or hazardous depends on the specific context.

Example: Consider an emergency stop button on a crane that disconnects the load during a lifting operation. This action may be safe if no one is in the crane lifting area, but it could be dangerous if people are present and do not have time to evacuate. In such scenarios, it may be safer to actively apply the control signal to ensure the function is only executed under controlled conditions. The choice between normally open (NO) or normally closed (NC) contacts in this setup depends on the overall system design and safety philosophy.

Unlike systems using Normally Energized (NE) control signals, where loss of power leads to a safe state, systems with NDE control signals require a reliable power supply and often a battery backup to ensure the control signal remains available when needed.

Until the 1990s, safety logic was typically implemented solely using hardware components. While systems like the railway signaling system at Oslo Central Station, installed in the 1960s, are still in operation today, most modern logic operations, including those related to safety, have since transitioned to software-based implementations. However, even with software-based control, relays and contactors remain essential, as software must still interact with physical devices to execute control actions. These components continue to play a critical role in bridging digital logic with real-world operations.

With the growing threat of cyberattacks targeting industrial control systems, there is increasing awareness of the need to combine hardware and software logic, especially for the most critical safety functions. This hybrid approach enhances system resilience by ensuring that essential operations can continue or be safely shut down, even in the event of a software failure or security breach.

4.2.3 Relay design features

A relay is generally split into two main parts, as illustrated in Fig. 3:

- Control side:
    - Coil: Wire wound around a magnetic core, generating a magnetic field when energized.
    - Control signal:
        - 5 V, 12 V, and 24 V DC or 230 V AC.
        - 10 mA to 150 mA.
- Load side:

- One or more contacts that open or close in the circuit with which they interact. In this example, there are two contacts: one normally closed (NC) and one normally open (NO), which change position depending on whether the coil is energized.
- Operates in circuit with:
  - 5V, 12 V, 24V or 230 VAC
  - 10mA -16 A.

- Armature: A movable component that shifts the position of the contacts in response to the magnetic field generated by the coil.

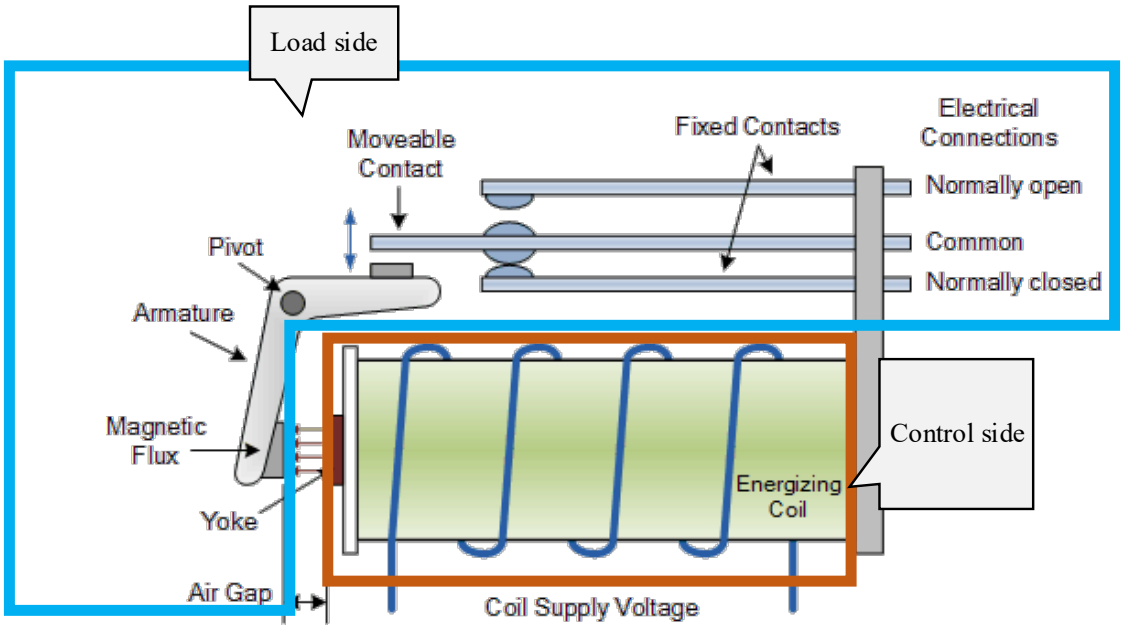


Fig. 3. Electromechanical relay (electronics-tutorials.ws)

When a relay has more than one contact, we refer to one of them as the primary and the others as auxiliary contact sets ("hjelpekontakter"). The primary contacts are those involved in the actual operation, while the auxiliary contacts are used for supporting functions, such as raising an alarm or reporting other status conditions.

4.2.4 Contactor design features

While a contactor performs similar functions to a relay, its internal design differs significantly due to the need to operate safely in high-current and high-voltage circuits, as illustrated in Fig. 4. Like an electromagnetic relay, it has a control side and a load side:

- Control side:
  - Coils (here, two): These generate the electromagnetic field required to actuate the contactor. Due to the higher power demands, contactors often use two coils instead of one to ensure reliable operation.
  - Control signal:
    - Typical voltage levels are: 24 V DC or 230 V AC
    - Typical current levels are: 50mA - 300 mA.
- Load side:

- One or more movable and stationary contacts: The movable contacts connect with or disconnect from the stationary contacts, depending on the force applied by the electromagnetic field, causing the contactor to close or open.
- Springs (here, two): These help return the contacts to their default position once the coil is de-energized.
  - Operates in circuit with:
    - 400 VAC – 690VAC
    - 9A-2000A

The control signal can be either NDE or NE, and contactors with contact sets can be either NO or NC. The contactor may, as a relay, have auxiliary switches, not shown in Fig. 4. The purpose of these additional contact sets is to perform functions such as providing feedback/status of the primary contact. The coil may first operate the auxiliary contacts. In contrast, the circuit containing the auxiliary contacts can generate a sufficient electromagnetic field (together with the electromagnets) to move the contacts in the power circuit.

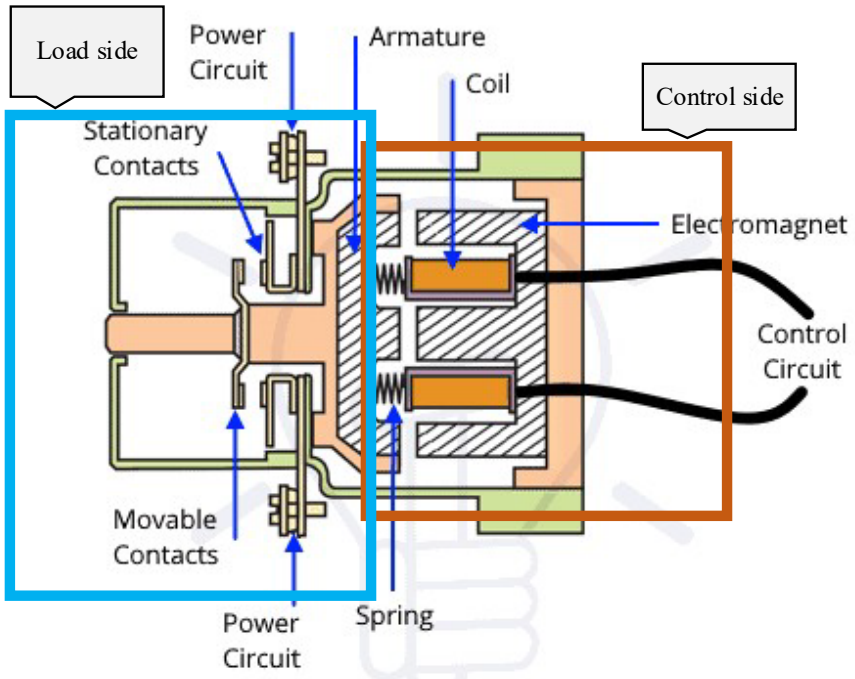


Fig. 4. Construction of a contactor (electricaltechnology.com)

4.2.5 Example: Activating a shutdown process

Fig. 5 illustrates how relays can be used to initiate a process shutdown (PSD). While the diagram focuses on a simplified example, it is important to note that activating a PSD may trigger multiple shutdown functions simultaneously, even if those are not shown here. The diagram can be read as follows:

The diagram can be interpreted as follows:

- The state of the TRIP coil determines whether the PSD is active. When the coil is energized, the PSD is not activated. If the coil is de-energized, the PSD is triggered, ensuring fail-safe behavior in the event of power loss.

- Under normal operating conditions, when there is no need to activate the PSD, pressure switch *ps* and level switch *ls* are assumed to be closed. If either pressure or level is above its setpoint, the corresponding contact opens.
- As long as both *ps* and *ls* remain closed, the PSD coil stays energized, indicating that no shutdown is required.
- However, if either *the ps* or the *ls* contact opens or the loop power is lost, the PSD coil is de-energized, initiating a shutdown.
- The PSD coil controls two sets of contacts*: psd1* (NO) and *psd2* (NC). When the PSD coil is energized, psd1 is closed and psd2 is open.
- If the PSD coil is de-energized, two things happen:
  - *psd1* opens, de-energizing the TRIP coil, which initiates the PSD shutdown.
  - *psd2* closes, activating an alarm that the operators may see on their screen. We may consider psd2 to be an auxiliary contact (as it reports status or alarm), while psd1 is the primary (as being involved in the trip itself).
- A bypass (BY) function can be used to temporarily prevent PSD activation, regardless of the state of *ps* or *ls*. To enable the bypass, the *by* contact (NO) must be closed—typically via a dedicated pushbutton. When the *pb* contact is closed, the bypass coil (BP) is energized, which closes the *bp* contact. As long as *bp* remains closed, the TRIP coil stays energized, and the PSD system is not activated—even if a shutdown condition occurs.

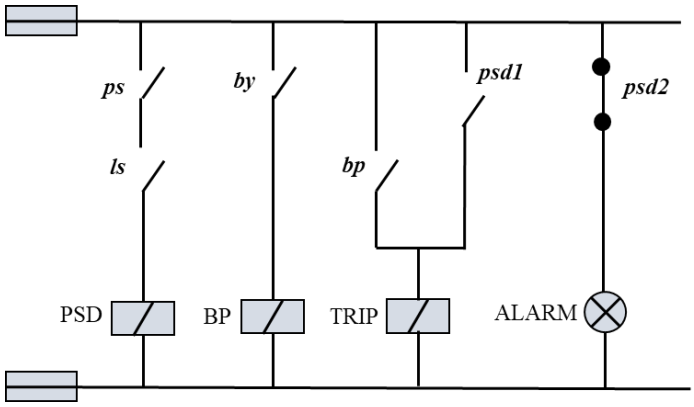


Fig. 5. Relay logic in electrical diagram

4.2.6 Example: Operating shutdown valves

Fig. 6 illustrates a simple example of relays interacting with solenoid-operated pneumatic valves that control the position of two shutdown valves.

We assume that the shutdown valves are fail-safe closed, meaning they will close upon loss of air pressure and open if pressure is applied.

- On the left side, a Normally Energized (NE) control signal causes a Normally Open (NO) contact to close. As a result, the solenoid valve is energized, and we assume this energized state keeps the shutdown valve open. If the control signal is lost, the NO contact opens, de-energizing the solenoid and causing the shutdown valve to close. This arrangement is therefore fail-safe, as the valve moves to a safe (closed) position in the event of power loss.
- On the right side, the control signal causes a Normally Closed (NC) contact to remain closed. This keeps the solenoid valve energized, maintaining the shutdown valve in the open position. If the control signal is lost, the NC contact remains closed, and the solenoid remains energized; therefore, the shutdown valve remains open. This arrangement is not fail-safe in the event of power loss, but it may be preferred in applications where accidental valve closure could pose a greater risk than leaving it open.

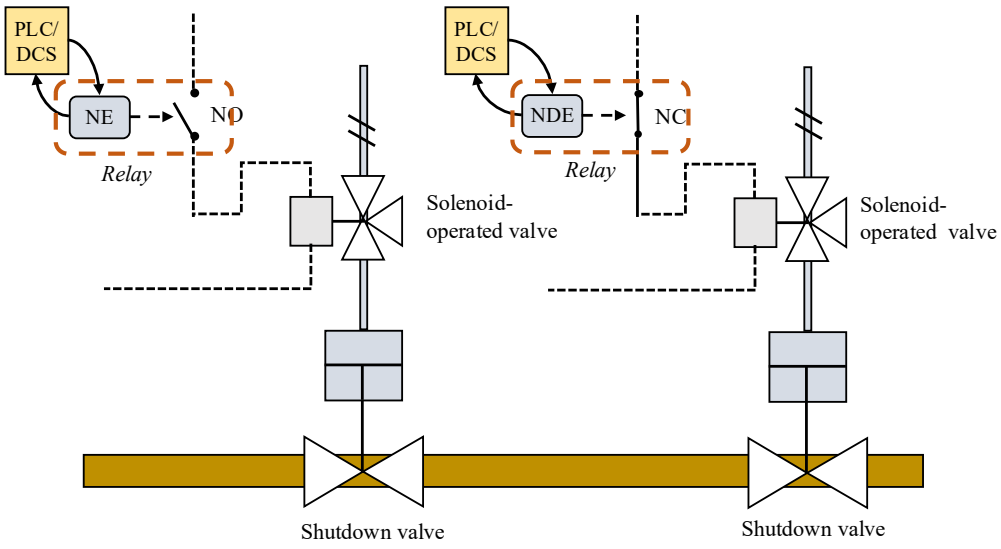


Fig. 6. Relay logic operating a solenoid valve

4.2.7 Example: Control power supply to a motor

A motor can be most effectively controlled using logical operations implemented in a Programmable Logic Controller (PLC) or a Distributed Control System (DCS). The programmed logic determines when the motor should start, stop, and at what speed it should operate. However, relay and contactor logic is still required to meet the safety and electrical isolation requirements defined in the Machinery Directive and relevant machinery safety standards. This includes ensuring the safe disconnection and reconnection of power under both normal and abnormal operating conditions, often at the local level, directly at the motor.

Fig. 7 shows an excerpt from an Electrical Circuit Diagram used in the Siemens motor lab in the TTK4175 course. The diagram includes most likely contactors, not relays, but the functionality is the same. Switches, meaning manually operated contacts, are also incorporated. A key point when interpreting such diagrams is that they are always presented in the de-energized state. From the diagram, we can interpret the following hardwired functions:

- The safety switch S0 has normally closed (NC) contacts that can be manually operated to disconnect power, for example, during maintenance.
- The main contractor (K1) with normally open (NO) contacts is a precondition for supplying power to the motor and is also able to remove power.
- Safety-certified contactors K2 and K3, whose control signals are set by the safety logic of a programmable safety controller, must both agree to give power to the motor.

Any loss of the control signal will remove power from the motor, which we can regard as a fail-safe feature.

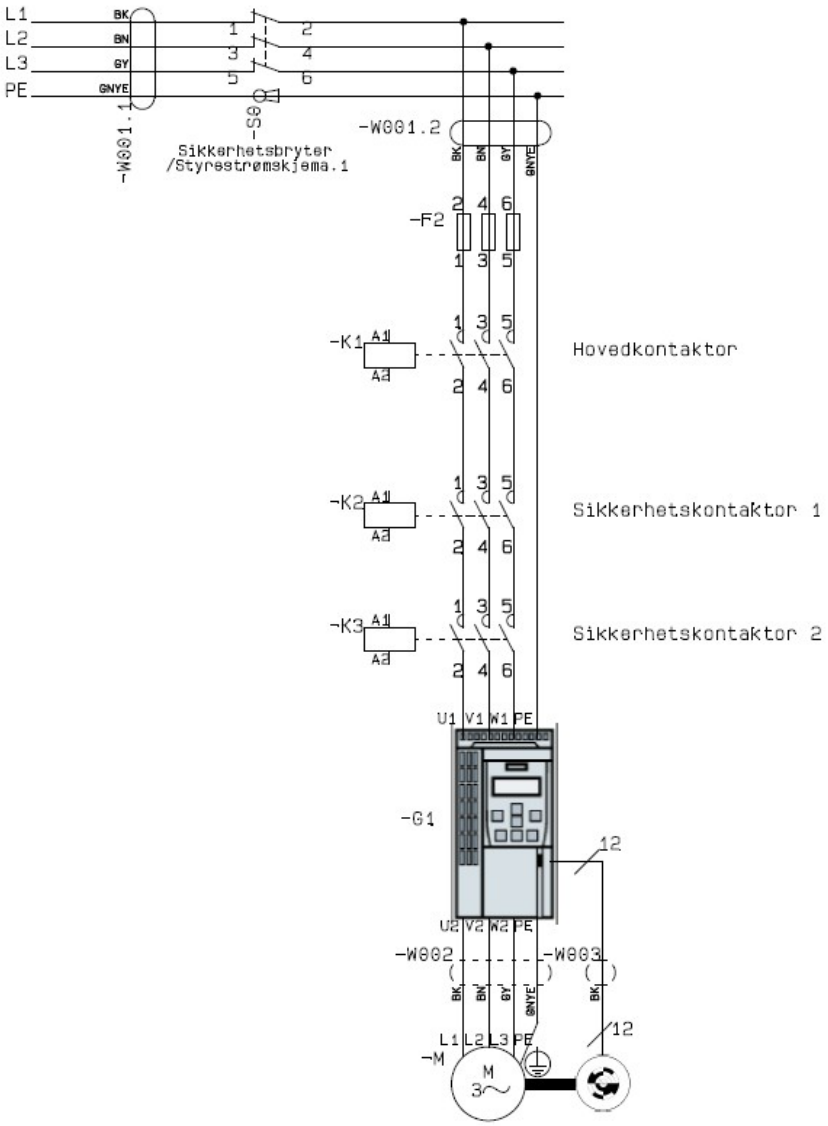


Fig. 7. Part of one of the electrical circuit diagrams for Siemens motor lab

4.3 Software logic for industrial controllers

A Programmable Logic Controller (PLC) or a Distributed Control System (DCS) consists of one or more central processing units (CPUs), input/output (I/O) modules, and communication interfaces that connect these components. The application software is loaded into the CPU, which uses the I/O modules to interact with the physical system it controls.

Programming a PLC or DCS is quite different from writing code in general-purpose languages like Python or C++. These systems are not programmed directly on the device. Instead, programming is done indirectly using a PLC or DCS programming editor, which runs on a standard computer known as an Engineering Workstation (EWS). The editors facilitate standardized languages for industrial controllers defined in IEC 61131-3 (2013). Due to cybersecurity requirements, the EWS is typically configured with restricted access—for example, it is often isolated from the internet to prevent unauthorized access or malware intrusion.

In practice, programming a PLC or DCS is less about writing code from scratch and more about configuring pre-built function blocks and flowcharts to meet the specific needs of an application, such as process control or

safety automation. This is one reason why PLC and DCS software is often referred to as an application program—it is tailored to a particular operational context using standardized components.

4.3.1 PLC and DCS programming languages

A PLC and a DCS rely on two types of software-implemented functions:

- The application program that contains the logical operations to be performed by the PLC or DCS
- The runtime, which is the execution environment that makes the application program run on the PLC or DCS

Application programs are programmed, or more precisely configured, in a DCS or PLC editor, such as Siemens TIA Portal or ABB Control Builder. The editor has two components: the tool for developing the application program, and the compiler that translates the program into an executable (binary) format suitable for the CPU.

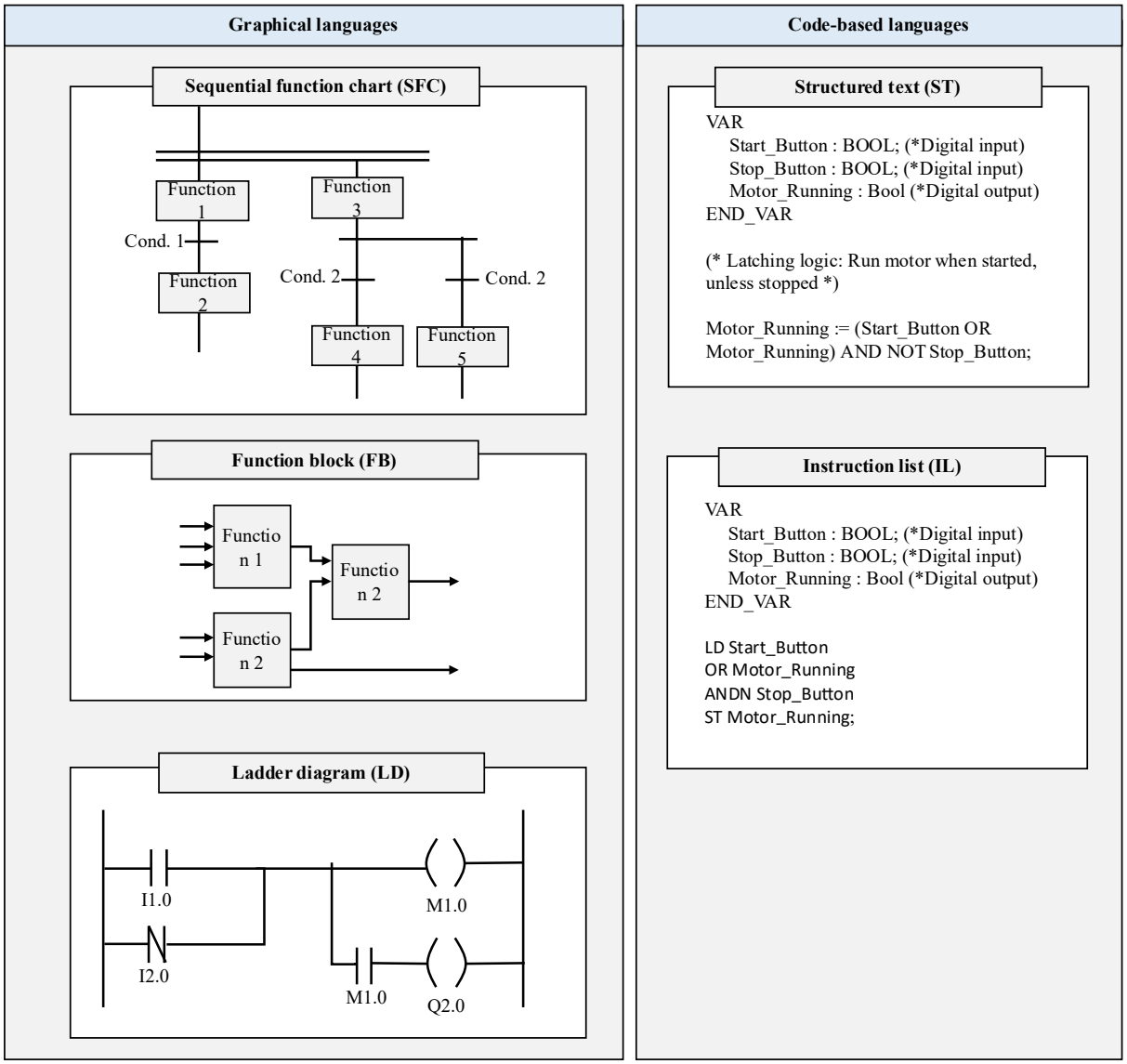


Fig. 8. PLC and DCS programming languages

The application program is developed using one or more of the five standardized programming languages specified in IEC 61131-3:

- **Ladder Diagram (LD).** A graphical language where logic is represented by symbols, like electrical diagrams. Applicable to PLCs.
- **Structured Text (ST).** A code language for writing simple instructions and more complex formulas, often based on Pascal syntax. Applicable to DCS and PLCs.
- **Function Blocks (FB) and FB diagrams.** A graphical language organized into functional blocks and their interfaces. Each functional block can be implemented in one of the other languages, for example, structured text. The FBs can be elementary function blocks, such as AND, OR, and XOR, or FBs housing several logical operations, including other FBs. Examples of function blocks include a PID regulator and a multiplication function. Functional blocks are well-suited for modular programming, specifically configuration, and the same functional blocks can be reused (and configured) within the program and in other (new) programs. Applicable to DCS and PLCs.
- **Sequential Function Chart (SFC).** A graphical language that describes the program sequence of functions. Sequences can be sequential or parallel. Each function is represented as a step, with conditions governing transitions to other steps. The language is well-suited to represent the overall structure of the program, whereas the realization of each function can be achieved, for example, using function blocks or structured text. Applicable to DCS and PLCs.
- **Instruction List (IL).** This is a low-level, assembler-like language in which only one logical operation is coded per line. This approach is best suited for shorter programs or those that require efficient code execution. Not so commonly applied anymore, and when used, it was for PLC, not DCS.

Fig. 8 illustrates some characteristics of programming languages, grouping them into two categories: graphical languages and coding languages. The choice of languages is typically guided by the manufacturer's programming guidelines or the programmer's personal preference. An application program often combines several approaches. For example, an SFC may realize the individual (sequential) functions and conditions through a combination of ST, FB, and, as appropriate, LD.

It seems that many control system vendors incorporate CODESYS, a hardware-independent software platform for programming and configuring industrial controllers that supports all languages specified in IEC 61131-3.

Many PLC and DCS editors provide a library of pre-developed functions encapsulated in FBs, such as PID controllers, valve operations, and motor control. Each of these FBs can be developed with a combination of ST, LDs, and SFC, depending on what is most suitable for the purpose. For safety PLCs or DCSs, the library functions have often been certified to IEC 61508-3, meaning that an independent body (like Exida or one of the TÜV organizations) has testified that the software architecture, development process, development environment (e.g., programming editors), and implemented functionality meet the standard's requirements.

4.3.2 Application program development

The main steps of application program development are illustrated in Fig. 9 and include:

1. **Identify and configure connected hardware:** Begin by configuring the hardware in the PLC editor. This includes adding all relevant devices such as the CPU, input/output (I/O) modules, power supply, and bus systems. Each device must be properly addressed. Operator stations with Human-Machine Interfaces (HMIs), including graphical representations of the controlled system and alarm systems, should also be included.

2. **Develop application program:** Develop the application using the PLC or DCS programming editor. These editors typically support a combination of programming languages, including Structured Text (ST), Function Blocks (FBs), and Sequential Function Charts (SFCs). Programming with FBs and SFCs is often referred to as *configuration* rather than *coding*, due to its graphical nature, which primarily involves setting parameters and connecting pre-programmed components. Each language is discussed in more detail later in the text.

    - As PLCs are generally more single-system oriented, all logic is programmed to be executed in each cycle.

- As DCS control is executed across multiple distributed controllers, the automation logic is structured into control modules. Instead of a single cyclic program, a DCS runs multiple scheduled control tasks in parallel, each responsible for its own process loop, equipment unit, or function block. This allows the DCS to coordinate large, multi-node continuous processes in a structured and scalable way. Control modules typically include built-in features such as mode handling, alarm management, sequencing, interlocks, and event handling.

3. **Compile the application program:** Compile the application within the programming editor. This step checks for errors, which must be resolved before proceeding. The compilation process translates the high-level code into machine-readable instructions that the PLC can execute. Some editors, such as Siemens TIA Portal and HIMA SILworX, also support simulation of the program before downloading it to the controller.

4. **Download the application program: Download** the compiled application and hardware configuration to the PLC or DCS while the controller is in *program mode*. Once the download is complete, switch the controller to *run mode* to begin execution.

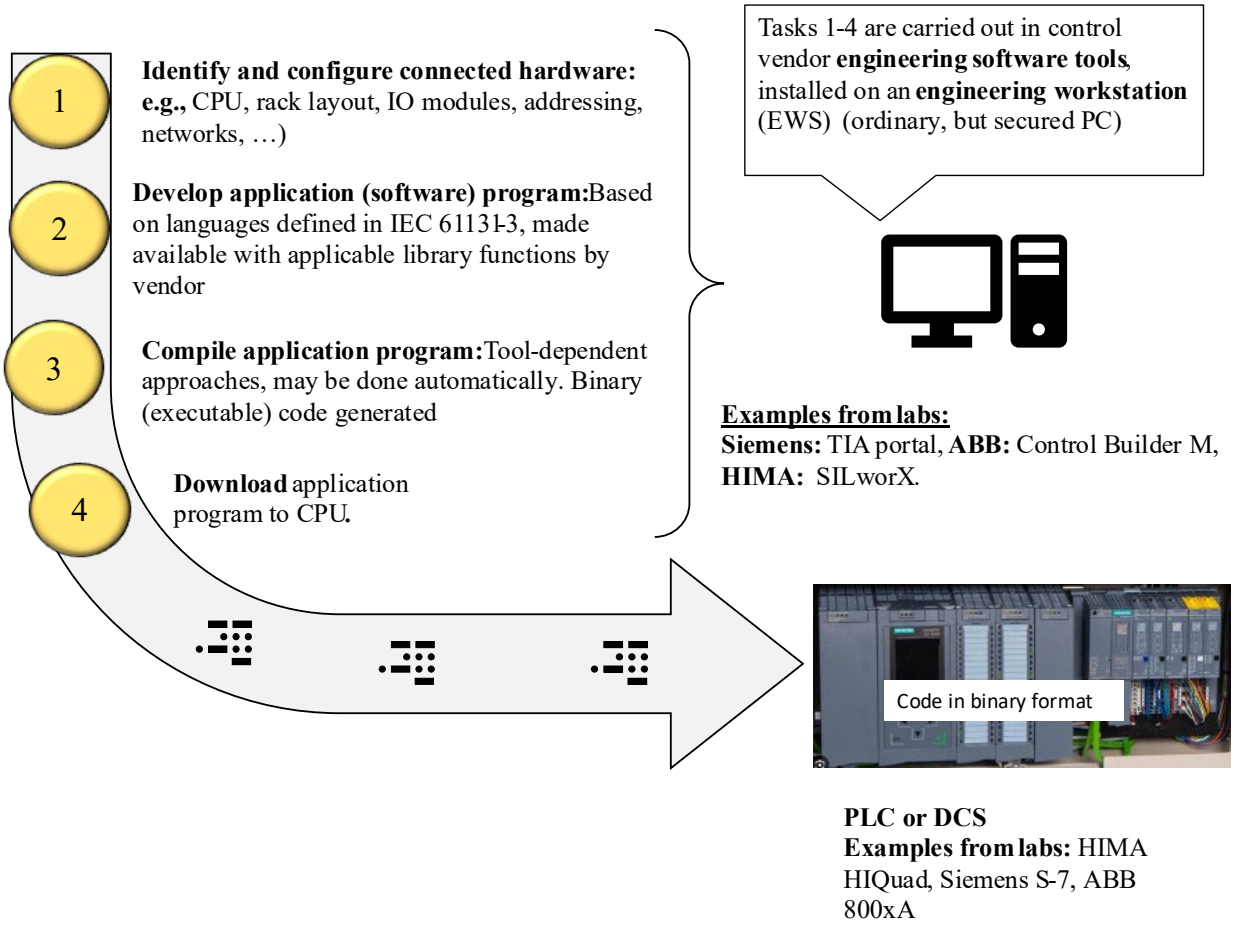


Fig. 9. Steps of programming a PLC or DCS

4.3.3 Runtime environment

The runtime environment is the software managing the execution of the application program, such as:

- Load the application program, meaning to initiate the application program execution in compiled binary executable form, upon power-up, after restart, cold or warm reseat, or after a mode change.
- Execute (run) the application program according to cyclic scan execution or event-driven execution models.

- Perform functions supporting application program execution, including management of variable (data) storage, memory allocation and updates, scheduling and prioritization of tasks, and control of scan cycles, execution deadlines, timers, and deterministic execution behavior.
- Handle interfaces with hardware, including I/O subsystems and digital communication (fieldbus or Industrial Ethernet).
- In the case of safety PLCs or DCSs, handle safety-related runtime mechanisms, such as fail-safe behavior or controlled (graceful) degradation.

Runtime executes the application program differently on a PLC compared to a DCS, illustrated in a simplified way with Fig. 6.

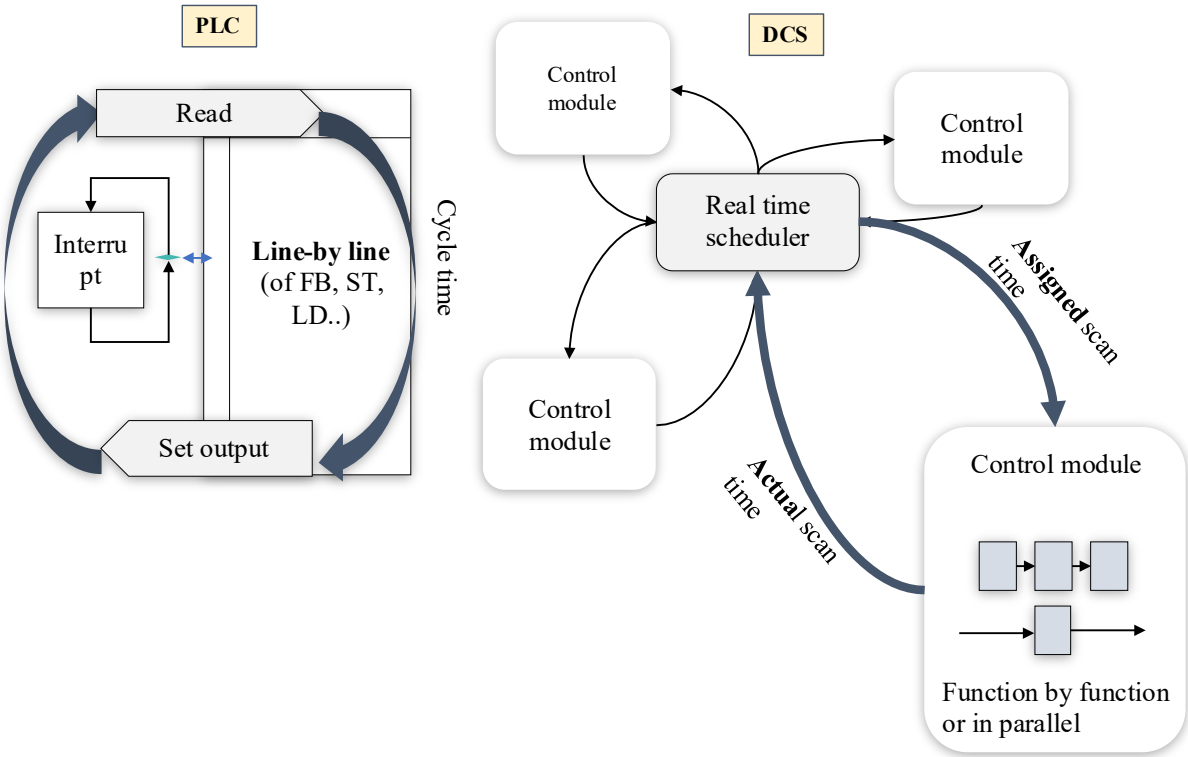


Fig. 10. Application program execution for PLC vs DCS

The PLC runtime typically executes the application logic using a **cyclic execution model**:

- **Update ("read") Inputs:** Input data from physical I/O modules or communication interfaces (e.g. fieldbus or Industrial Ethernet) are transferred into an input process image, isolating the application logic from direct hardware access.
- **Execute Application Logic:** Based on the stored input data, the application logic is executed deterministically according to the configured execution order of tasks and program units. Execution order matters, although execution is not strictly line-by-line but follows the language and task semantics.
- **Update ("write") Outputs:** The results of the logic execution are written to an output process image. Physical outputs or networked devices are updated from this image, and output values are held constant until the next update cycle.

The time required to complete one iteration of these steps is called the cycle time. The application and task configuration must ensure that this cycle meets the system's real-time requirements.

PLCs may additionally support high-priority or event-driven tasks, including hardware or communication-triggered interrupts, to handle fast-changing signals or time-critical functions. These tasks may

update shared data asynchronously relative to the cyclic task, and improper handling of these interactions may lead to inconsistent or unexpected behavior.

In a DCS architecture, the runtime executes the application program in a **distributed manner**.

- Rather than a single centralized scan, the runtime coordinates execution across multiple independent control controllers, each responsible for a defined set of control functions.
- Each controller executes its assigned control functions at configured, deterministic execution intervals under a real-time scheduling framework. Input acquisition, output updates, and data exchange are handled locally at the controller level, with explicit communication mechanisms used for inter-controller data sharing.
- Each controller therefore acts as an autonomous control unit, executing control functions implemented using function blocks, sequencing mechanisms, and textual logic (e.g., structured text or vendor-specific equivalents).
- Execution follows a data-dependency-driven model, in which function blocks are invoked when their inputs are available; independent functions may be scheduled and executed concurrently, subject to controller resource constraints.
- •The execution interval of each controller is deterministic, while actual execution time within that interval may vary but is bound and monitored to ensure completion within defined real-time constraints.
- Hardware interrupts and low-level scheduling are handled by the DCS firmware and are generally not exposed directly to the application programming model.

4.3.4 Example using Siemens PLC editor

The webpage Instrumentationtools.com provides an example of how to control a motor using the Siemens programming editor. The editor has a library of relevant FBs for this purpose, including some explicitly designed for motor control.

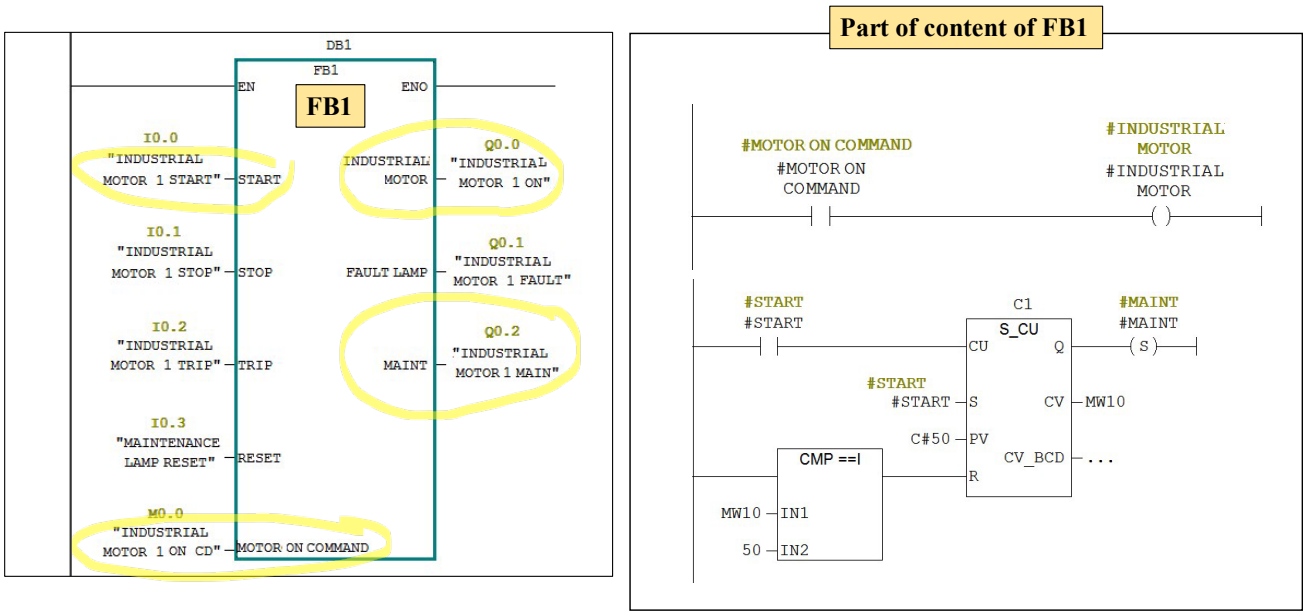


Fig. 11. Motor control FB (Instrumentationtools.com)

Graphically, the FB is shown as a rectangular block, with inputs on the left side and outputs that can connect to I/O or internal signals. Each FB that is used for the programming is instantiated from the library by adding it to a data block (DB), as shown in Fig. 11. The editor allows the user to view the content of each FB from the library to better understand its functionality. For FB1, we notice that the realization is a mixture of LD and FBs.

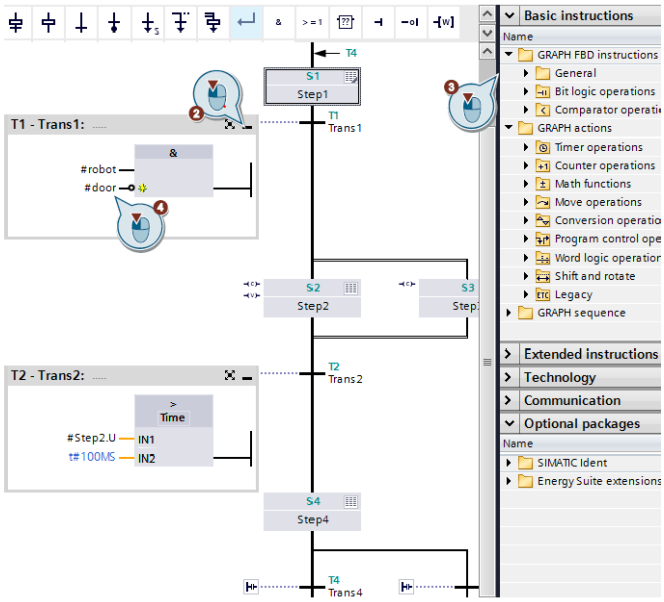


Fig. 12. Example of an SFC program (Instrumentationtools.com)

Control that involves sequential tasks can be programmed using a sequential flowchart (SFC). An example also from the Siemens TIA portal manual is shown in **Fig. 12**.

Here, each numbered box identifies the function to be performed and the completion required before proceeding. A horizontal line and a short description indicate the condition for moving forward (transition). The conditions that determine whether the program sequence can proceed can be implemented using functional blocks.

4.3.5 Ladder logic vs hardwire logic

Ladder Logic (LD) was developed to simplify the transition from traditional relay-based control systems to programmable logic controllers (PLCs). It mimics the appearance and logic flow of electrical relay circuits, making it easier for electricians and technicians familiar with relay diagrams to adopt PLC programming.

Fig. 14 illustrates two variants of what Ladder Diagrams may look like, with the same logical operations shown for a circuit diagram in Fig. 5. The symbols applied for LD are explained in Fig. 13.

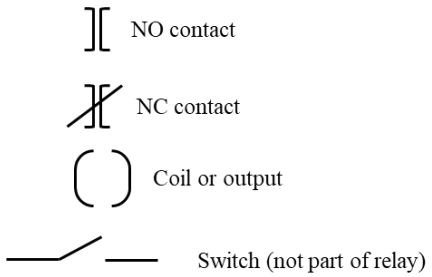


Fig. 13. LD symbols related to relays and operation

Compared to the layout of a traditional electrical circuit diagram, the Ladder Diagram (LD) has the following characteristics:

- Orientation: The LD is rotated 90 degrees relative to the electrical circuit diagram. In LD, logic flows horizontally from left to right, rather than vertically.

- Structure: Each branch (or "rung") represents a logical operation and is read from left to right. All branches together form one complete program execution cycle. It is always shown as inactive, like the rest state of a circuit diagram.
- Graphical symbols: Layout and naming borrowed from electrical circuit diagrams.

The logical operations in Fig. 14 can therefore be interpreted in the same way as we did for Fig. 5.

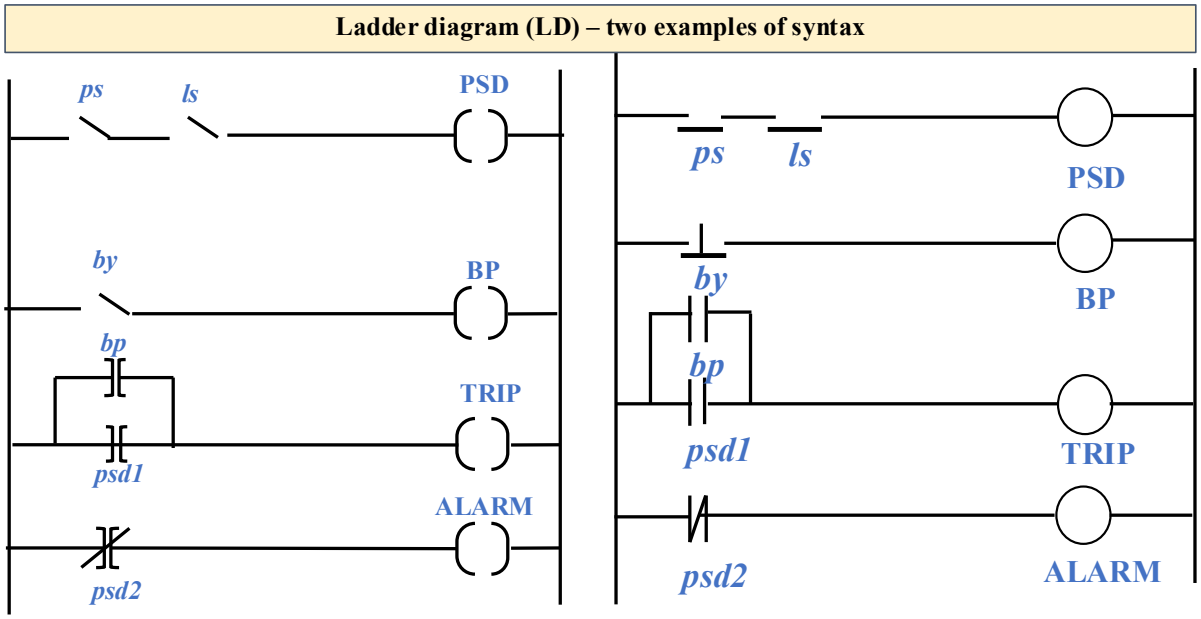


Fig. 14. From electrical circuit diagram to ladder diagram (LD)

4.4 Preparation of software coding

It can be wise to describe all the planned functions and their interactions before starting to make an application program, without considering any specific programming language. Graphical models are often well-suited for the purpose, and here we are suggesting a few alternatives:

- Functional block diagram, identifying some high-level functions and their inputs and outputs
- State transition diagrams
- Context diagrams

4.4.1 Functional block diagrams

Functional block diagrams, which identify functions and their relationships, are often helpful as an abstraction of the software to be implemented. The diagrams can include both software functions and hardware-implemented functions, ensuring that all interaction dependencies are identified, including controller actions and feedback. The diagram can be a helpful, technology-independent, intuitive way to discuss system implementation with other disciplines involved in the system's design or operation.

Fig. 15 illustrates a functional block diagram for valve operation, with the hardwired-implemented functions stippled. The purpose of the system is to set the correct position of the air-to-operate (ATO) valve.

The diagram identifies the following subfunctions to be implemented by the controller:

Controller actions:

- Send an open or close command to the solenoid-operated valve.
- Send a confirmation request to the external system according to the open/close request

The controller relies on the following input:

- Open/close request
- The position of the ATO valve (confirmed open or confirmed closed)
- Status of reset

Controller algorithms must include:

- Before the command is placed:
  - Monitor open/close requests
- After the command has been placed:
  - Maintain command so that the position of the solenoid-operated valve remains unchanged until the reset has been received.
  - Compare feedback of ATO valve position with open/close request

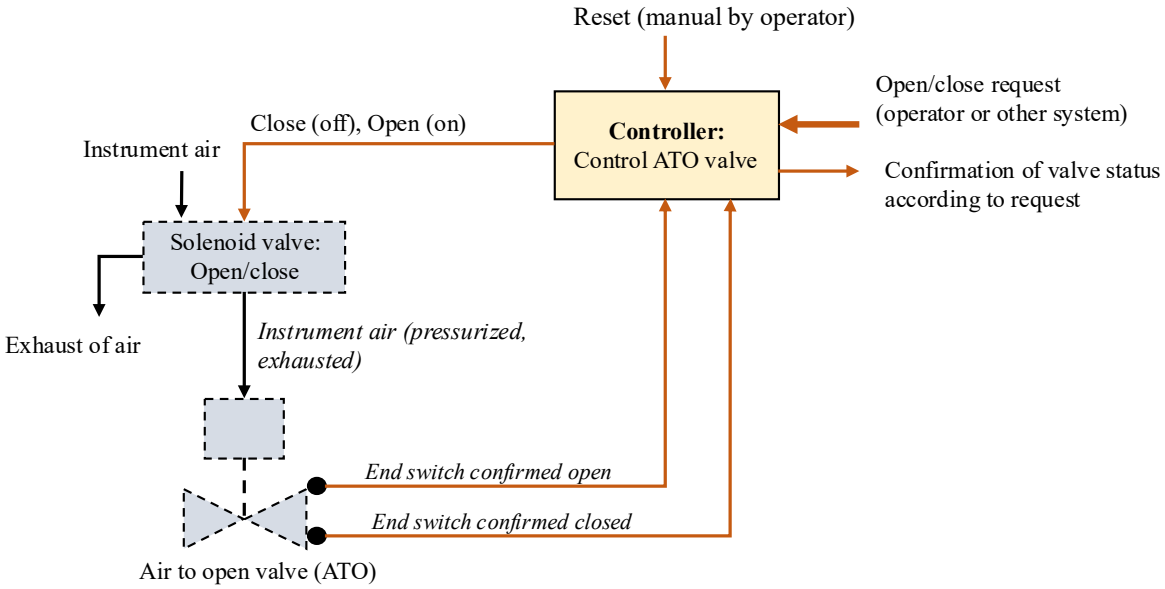


Fig. 15. Functional block diagram of valve control

Discussions with future users could identify the need to clarify whether the command should be placed if the valve is already in the requested position. If this condition is added, corrupted feedback may prevent the valve from being activated.

4.4.2 State transition diagram

State diagrams are another complementary approach to detailing the states of a system and how the values of parameters exchanged determine them. The diagrams may be drawn in different ways as long as they provide necessary clarity and consistency.

Fig. 16 represents a state diagram for the logic operations identified in Fig. 15. The rounded boxes represent states and transitions, marked by arrows. A state diagram for valve control is illustrated as in Fig. 16. In the illustration, the commands have a yellow background, while confirmations and fault detection have a green background.

The parameters used are:

States	Explanation
VALVE_OPEN	The valve is in the open position
VALVE_CLOSES	Valve is in the transition to close

VALVE_CLOSED	Valve is in the closed position
VALVE_OPENS	Valve is in transition to open
RESET_ACTIVATED	Reset has been activated
VALVE_FAULT	Valve is in the fault state due to detected faults
Commands, confirmations, and fault detection	Explanation
INIT	Command: Initialize system (remove fault states)
OPEN	Command: Open the valve
$\overline{OPEN}$	Command: Close valve
RESET	Command: Reset (condition for re-opening)
ES_OPEN + $\overline{ES\_CLOSED}$,	Fault detection: Valve reported open, while being closed
$\overline{ES\_OPEN} + ES\_CLOSED$	Fault detection: Valve reported closed, while being open
ES_OPEN	Confirmation: Valve is open
ES_CLOSED	Confirmation: Valve is closed
TIMER_10SEC	Fault detection: Timer 10 seconds (closing requirement)
TIMER_14SEC	Fault detection: Timer 14 seconds (closing requirement)

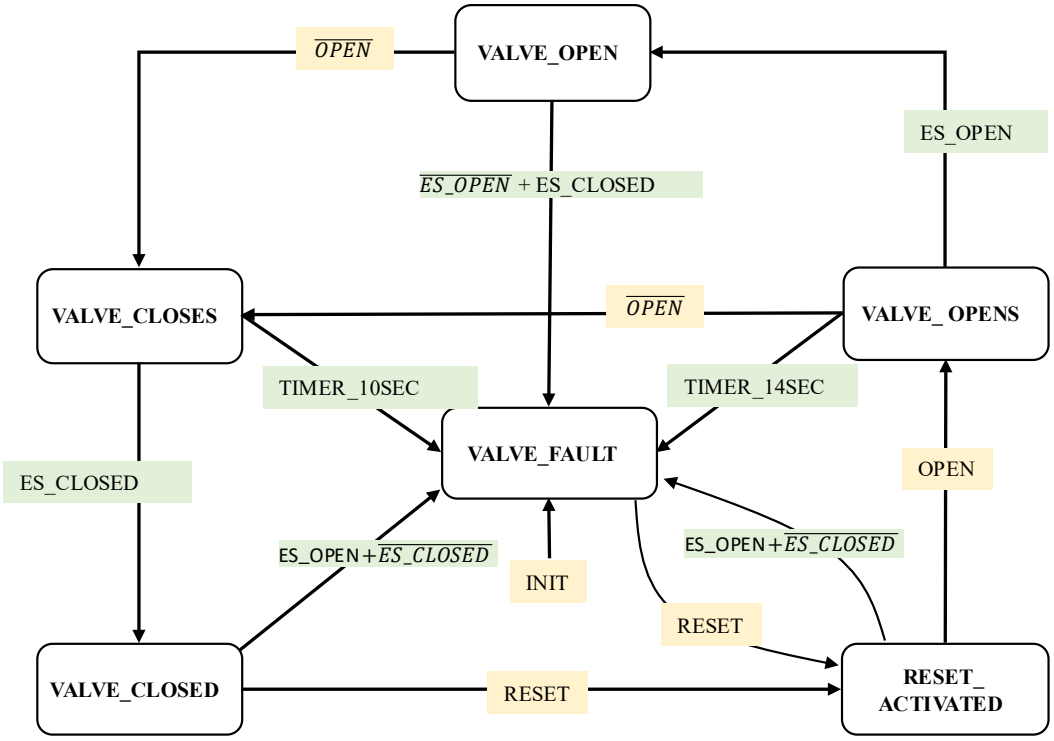


Fig. 16. State diagram for the valve operation

This diagram illustrates the following functionality of the system:

- The valve is kept open if OPEN is active, and otherwise, the valve starts to close
- The valve is confirmed closed when confirmed by the valve position indicator (“end switch”) ES_CLOSED

- The valve will not open again until reset manually (RESET).
- After the reset has been activated, it is possible to command the valve to open (OPEN).
- The valve is confirmed open when confirmed by the valve position indicator ES_OPEN.

The valve enters the fault state in case of the following detected fault states:

- The valve reports as closed, while it is in the open state (and vice versa).
- The valve reports as open, while it is not yet reset.
- The valve takes too long to close (more than 10 seconds) or open (more than 14 seconds).

4.4.3 Context diagram

A context diagram illustrates how the application program interacts with elements outside the system boundary. It focuses on external entities that exchange information or signals with the system. Fig. 17 presents an example of valve operation. In this example, the application program needs to interact with:

- An external reset switch
- The operator
- The actuated valve

These elements represent the environment in which the application program operates, highlighting the input it receives and the outputs it generates.

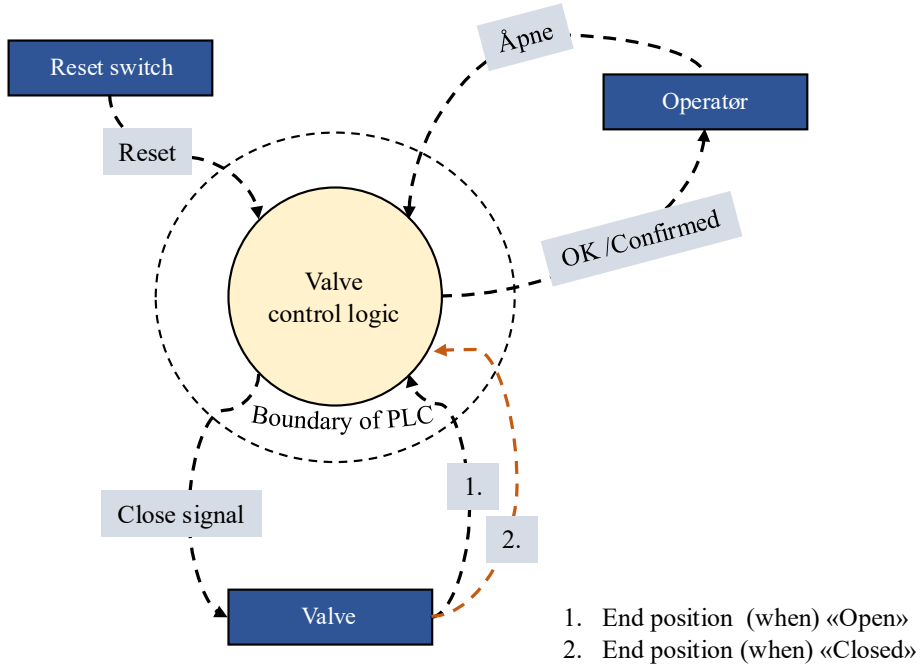


Fig. 17. Context diagram for the valve operation (Adopted from T. Onshus)

4.5 Standards for developing software for safety systems

Implementing software for safety-critical systems, such as safety-instrumented systems (SIS), imposes restrictions that are not imposed on regular control functions. Two key international standards are particularly important when developing or configuring software for PLCs and DCSs in the process industry:

- IEC 61508-3 (2010a):

This standard outlines the requirements for software development in safety-critical systems. It applies broadly across industries and includes:

- Restrictions on the use of programming languages (not limited to PLC/DCS environments)
- Requirements for compiler tools and development environments
- Guidelines for the software development process, including documentation and verification

IEC 61508-3 provides a generic framework for software implementation and is often used as the basis for certification of software libraries and programming editor tools. Major control system manufacturers, such as Siemens, ABB, Kongsberg Maritime, and Honeywell, adhere to this standard during product development, even for systems used outside the process industry.

- IEC 61511-1 (2016):

  This standard is specifically tailored to the process industry and focuses on the application and configuration of already certified software components. It provides guidance on:

  - The use of pre-certified software libraries
  - Configuration procedures and tools
  - Responsibilities of system integrators and end users (e.g., plant operators)

  IEC 61511-1 is typically applied during the engineering and deployment phase of safety systems, ensuring that certified tools and components are used correctly and safely.

4.5.1 Generic software development process per IEC 61508-3

Unlike general-purpose software development, programming for safety systems must be carried out in a way that allows a certain level of trust, referred to as safety integrity, expressed in terms of four levels: safety integrity level (SIL) 1, SIL 2, SIL 3, or SIL 4. Each SIL level corresponds to a maximum permitted probability that the safety system fails to perform as specified and when needed. However, estimating the probability of software failure is challenging, and in many cases, it is considered impractical or impossible due to the complexity and variability of software behavior. In safety system development, especially when the focus is on configuration rather than custom coding, the primary strategy to ensure sufficient safety integrity is to:

- Avoid introducing configuration errors, and
- Use development tools and processes that help detect and eliminate any errors that do occur.

To sufficiently reduce the probability of introducing or overlooking software errors, IEC 61508-3 (2010b) has specified several measures, such as:

1. Modular Development and Verification:

   The standard emphasizes modular software development, supported by documented verification and validation activities at each stage of the process.

2. Tool Qualification:

   There are specific requirements for the development tools used, including their functionality, reliability, and suitability for safety-critical applications.

3. Language and Coding Restrictions:

   The standard imposes restrictions on the choice of programming languages and limits the types of functionalities that can be implemented to reduce complexity and improve verifiability.

4. Competence Requirements:

   All personnel involved in software development must demonstrate sufficient competence in the principles and practices outlined in IEC 61508-3.

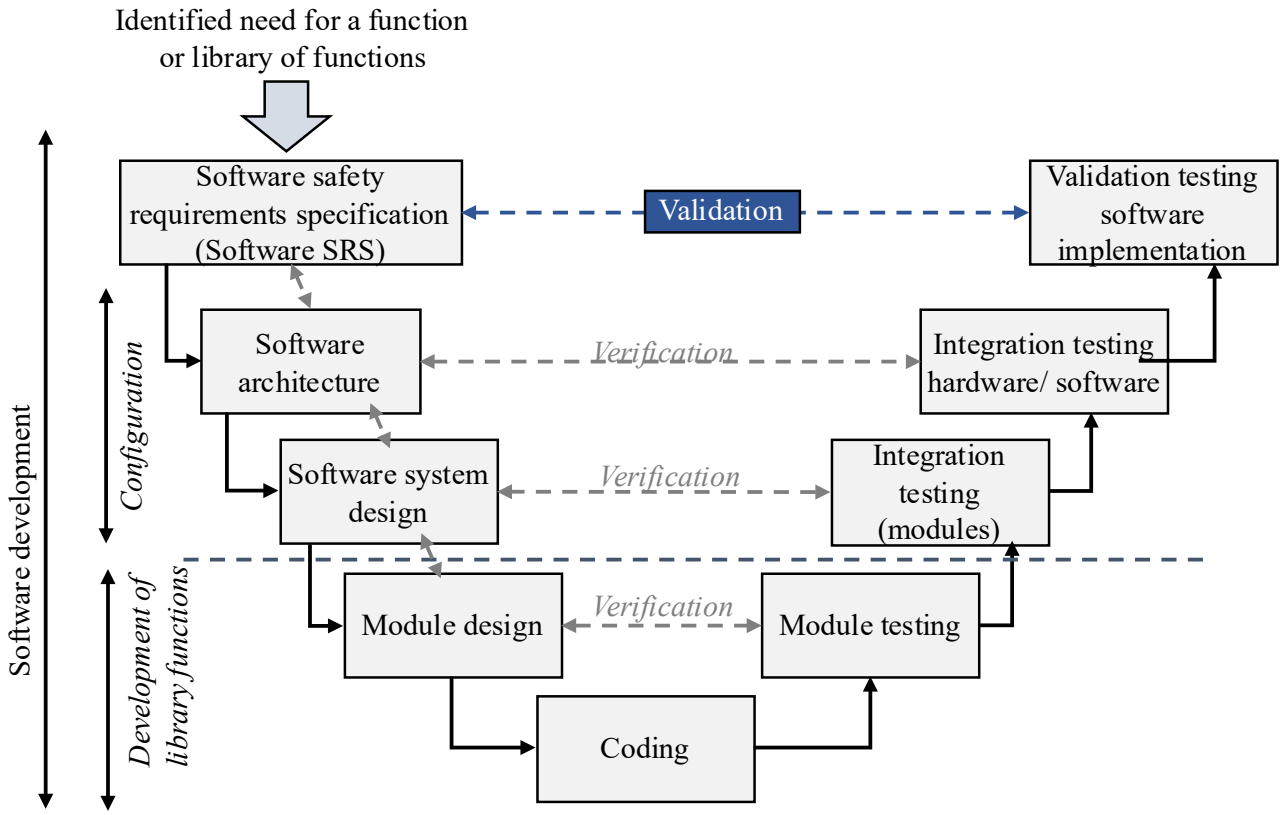


Fig. 18. V-model for software development (adapted from IEC 61508-3)

IEC 61508-3 recommends the use of the V-model (illustrated in Fig. 18) as a structured approach to modular development and verification. This model aligns development activities on the left side with corresponding verification and validation activities on the right. The key stages include:

- Software Safety Requirements Specification (SRS): This document defines the functional and safety requirements for the software. It includes:
    - Requirements to be implemented by the controllers, expressed in a format suitable for logical implementation.
    - Architectural considerations, such as grouping of related functions and their allocation across one or more controllers.
- Software Architecture Design: This stage defines the main software components, including:
    - Operating systems, databases, controller software, communication protocols, programming tools, and diagnostic utilities.
    - The influence of the SRS on the selection and configuration of these components is assessed.
- Software System Design: The architecture is decomposed into software modules, and responsibilities for implementation and verification are assigned.
- Module Design: Each module is further specified in accordance with the system design. This includes detailed requirements for functionality, timing, interfaces, and data formats, forming the basis for coding.
- Module Testing: Each coded module is tested to verify compliance with its design specifications. This includes checks for:
    - Functional correctness
    - Timing behavior

  - Data formats and interfaces
  - Code quality and structure

- Integration Testing: The software modules are integrated and tested on the target hardware (or a simulated environment) to ensure correct interaction between components.
- Validation Testing: The fully integrated system is validated to confirm that it performs as intended under all expected operating conditions. This includes interaction with external systems and processes. Validation may also involve document reviews, analyses, and inspections to ensure that the software meets all specified requirements.

Each step of the V-model in IEC 61508 is supported by detailed requirements that vary by Safety Integrity Level (SIL), ranging from SIL 1 to SIL 4. These requirements are summarized in tables in the standard, which help developers understand the measures expected at each SIL (SIL is explained in more detail in Chapter 9). Tab. 3 shows some few examples from these tables, focusing on requirements related to software architecture, development tools, and programming/coding practices.

The tables use the following notations to indicate the level of recommendation for each measure:

- Mandatory (M) – The measure must be applied.
- Highly Recommended (HR) – The measure should be applied, unless an apparent and justified reason is provided for omitting it.
- Recommended (R) – The measure should be applied, or an equivalent alternative should be used.
- Not Recommended (NR) – The measure is discouraged, and if used, its application must be justified.
- "---" (Dash) – No specific recommendation is made for this measure.

Tab. 3. Extracted examples from IEC 61508-7 (with a few minor text edits)

Application	Technique/measure	SIL 1	SIL 2	SIL 3	SIL 4
Software architecture (samples)	Fault detection	…	R	HR	HR
	Error detection codes	R	R	R	HR
	Fault assertion programming	R	R	R	HR
	Graceful degradation	R	R	HR	HR
	Dynamic reconfiguration	…	NR	NR	NR
	Artificial intelligence – fault correction	…	NR	NR	NR
	Modular approach	HR	HR	HR	HR
Development tools (samples)	Support strongly typed programming languages (e.g., Ada, C++)	HR	HR	HR	HR
	Support suitable programming language(s)	HR	HR	HR	HR
	Certified tools and translators	R	HR	HR	HR
Programming and coding (samples)	Computer-aided design tools	R	R	HR	HR
	Modular approach	HR	HR	HR	HR
	Structured programming	HR	HR	HR	HR
	Use of trusted/verified software elements (if available)	R	HR	HR	HR
	Forward traceability between the software SRS and its implementations	R	R	HR	HR

IEC 61508-3 does not mandate specific programming languages but provides guidance on language characteristics suitable for safety-related software, such as:

- The language must be adapted to the user and the problem domain, rather than to a specific processor or hardware platform. For this reason, assembly language is not recommended.
- Well-established and widely used languages are preferred over niche or proprietary languages with limited adoption.
- The languages to support:
  - Block structure for modular and readable code
  - Runtime checks, such as bounds checking for arrays and tables
  - Modular design, enabling the use of small, manageable software components
  - Encapsulation, allowing control over access to code and data within modules
  - Real-time capabilities, including support for exception handling and interrupt management
  - Pre-verified function blocks, where possible, to reduce the risk of introducing errors
  - Tool support for debugging, version control, and configuration management

Characteristics mentioned as not suitable, due to the possibility of unexpected or undesired behavior, are:

- Unconditional jumps except for subroutine calls.
- Recursion.
- Pointers, tree structure (heaps), or any dynamic variables or objects.
- Cancel handling at the source code level.
- Many jumps in and out of loops, blocks, or subprograms.
- Implicit initialization and/or declaration of variables.

IEC 61508-7 (2010) identifies examples of suitable programming languages, including Ada and C (and, under defined restrictions, C++), for developing safety-related application software from "scratch" and for the development of reusable software elements. The same programming languages are applicable for implementing the functionalities underlying the five IEC 61131-3 standardized programming languages.

4.5.2 Application program development with IEC 61511

IEC 61511-1 (2016) is the process sector standard for functional safety and the primary standard for application program development for safety-instrumented systems (SIS). The standard assumes that safety-certified PLCs or DCSs are used, with editors providing pre-certified software libraries, validated compilers, and software programming manuals (compliant with IEC 61508-3).

IEC 61511 outlines several general requirements for SIS application program development, where a central requirement is establishing a software safety requirements specification (software SRS) that addresses the following:

- Identification of all Safety Instrumented Functions (SIFs) and their corresponding logical operations
- Definition of program structure and modular organization
- Mapping of hardware and software architecture to SIFs, meaning describing what logical operations are implemented by hardware and what are implemented by software.
- Specification of which certified library modules are to be used
- Identification of how the configuration ensures fail-safe performance (in terms of input faults, internal faults, or loss of power)

- Interaction between SIS and other systems, such as those in the process control system. For example, regarding status information, alarms, and faults are reported to the process control system rather than to the SIS.
- Use of diagnostic features to detect faults and errors, including:
    - Data integrity checks
    - Sensor validation
    - External watchdog mechanisms
- Inclusion of additional functionality to support:
    - Function testing
    - Maintenance activities
    - System start-up and planned shutdowns

Additional requirements include:

- Traceability of implemented functionality against the original specifications is essential to ensure that no functionality has been overlooked.
- Version control to manage software changes is essential for both safety and security purposes.
- Security features to protect the integrity and reliability of the safety system, relating to measures identified in a cybersecurity risk analysis, following IEC 62443 (2009–2025)

IEC 61511 permits the use of non-safety controllers, i.e., controllers not certified or compliant with IEC 61508. However, such a need arises in very rare cases, and the end user must provide sufficient evidence to document "prior use". Because these cases are rare, the details of prior-use restrictions are not covered here.

4.6 Bibliography

IEC 61131-3. (2013). *Programmable controllers - Part 3: Programming languages*. International Electrotechnical Commission.

IEC 61508-3. (2010a). *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements* International Electrotechnical Committee.

IEC 61508-3. (2010b). *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*. International Electrotechnical Commission.

IEC 61508-7. (2010). *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures* International Electrotechnical Commission.

IEC 61511-1. (2016). *Functional safety - Safety instrumented systems for the process industry sector - Part 1: Framework, definitions, system, hardware and application programming requirements*. International Electrotechnical Commission.

IEC 61810. (2019). *Electromechanical elementary relays - Part 1: General and safety requirements (IEC 61810-1:2015+A1:2019 CSV)*. Electrotechnical Commission.

IEC 62443. (2009–2025). *Security for industrial automation and control systems. Multiple parts, each published separately.* International Electrotechnical Commission.