



PRACTICAL
INDUSTRIAL
PROGRAMMING
using 61131-3
for PLCs





Technology Training that Works

Presents

Practical

Industrial Programming

*using **IEC 61131-3** for PLCs*

Web Site: www.idc-online.com

E-mail: idc@idc-online.com



Technical Information that Works

Copyright

All rights to this publication, associated software and workshop are reserved. No part of this publication or associated software may be copied, reproduced, transmitted or stored in any form or by any means (including electronic, mechanical, photocopying, recording or otherwise) without prior written permission of IDC Technologies.

Disclaimer

Whilst all reasonable care has been taken to ensure that the descriptions, opinions, programs, listings, software and diagrams are accurate and workable, IDC Technologies do not accept any legal responsibility or liability to any person, organization or other entity for any direct loss, consequential loss or damage, however caused, that may be suffered as a result of the use of this publication or the associated workshop and software.

In case of any uncertainty, we recommend that you contact IDC Technologies for clarification or assistance.

Trademarks

All terms noted in this publication that are believed to be registered trademarks or trademarks are listed below:

IBM, XT and AT are registered trademarks of International Business Machines Corporation. Microsoft, MS-DOS and Windows are registered trademarks of Microsoft Corporation.

Acknowledgements

IDC Technologies expresses its sincere thanks to all those engineers and technicians on our training workshops who freely made available their expertise in preparing this manual.



Technical Information that Works

Customized Training

In addition to standard on-site training, IDC specializes in customized courses to meet client training specifications. IDC has the necessary engineering and training expertise and resources to work closely with clients in preparing and presenting specialized courses.

These courses may comprise a combination of all IDC courses along with additional topics and subjects that are required. The benefits to companies in using training is reflected in the increased efficiency of their operations and equipment.

Training Contracts

IDC also specializes in establishing training contracts with companies who require ongoing training for their employees. These contracts can be established over a given period of time and special fees are negotiated with clients based on their requirements. Where possible IDC will also adapt courses to satisfy your training budget.

References from various international companies to whom IDC is contracted to provide on-going technical training are available on request.

Some of the thousands of Companies worldwide that have supported and benefited from IDC workshops are:

Alcoa, Allen-Bradley, Altona Petrochemical, Aluminum Company of America, AMC Mineral Sands, Amgen, Arco Oil and Gas, Argyle Diamond Mine, Associated Pulp and Paper Mill, Bailey Controls, Bechtel, BHP Engineering, Caltex Refining, Canon, Chevron, Coca-Cola, Colgate-Palmolive, Conoco Inc, Dow Chemical, ESKOM, Exxon, Ford, Gillette Company, Honda, Honeywell, Kodak, Lever Brothers, McDonnell Douglas, Mobil, Modicon, Monsanto, Motorola, Nabisco, NASA, National Instruments, National Semi-Conductor, Omron Electric, Pacific Power, Pirelli Cables, Proctor and Gamble, Robert Bosch Corp, Siemens, Smith Kline Beecham, Square D, Texaco, Varian, Warner Lambert, Woodside Offshore Petroleum, Zener Electric

Table of Contents

1	An introduction - IEC 1131-3 on PLC programming	1
1.1	Development & growth of Programmable Controllers	2
1.2	Need for standardization in programming approach	3
1.3	Drawbacks in conventional programming methodology	4
1.4	Features of IEC-1131-3 language definition	10
1.5	Summary	11
2	PLC software architecture	12
2.1	Software quality attributes	12
2.2	IEC software architecture	13
2.3	Component parts of IEC Software architecture	14
2.3.1	Configuration	14
2.3.2	Resource	15
2.3.3	Task	15
2.3.4	Program	15
2.4	Functions and function blocks	16
2.5	Local and global variables	17
2.5.1	Directly represented variable	17
2.5.2	Access paths	18
2.5.3	Execution control	18
2.6	Mapping software model to real life systems-Examples	18
2.7	Applications	20
2.7.1	Program Organization Unit (POU)	21
2.7.2	Hierarchical design	22
2.7.3	Communications	22
2.8	Conclusion	24
3	Common elements in IEC-1131-3	25
3.1	Common elements	25
3.1.1	Character set	26
3.1.2	Identifiers	26
3.1.3	Keywords	26
3.1.4	Comments	27
3.2	Elementary data types	27
3.2.1	Integer	27
3.2.2	Real or floating point	28
3.2.3	Time or duration	28
3.2.4	Date and time data	29
3.2.5	String	30

	3.2.6	Bit string	30
3.3		Generic data type	31
	3.3.1	Initial values	31
3.4		Derived data types	31
	3.4.1	Derived directly from elementary type	31
	3.4.2	Structured data type	32
	3.4.3	Enumerated data type	32
	3.4.4	Sub range data types	32
	3.4.5	Array data types	33
	3.4.6	Default initial values of derived data type	33
3.5		Variables	34
	3.5.1	Internal variable	34
	3.5.2	Input variables	35
	3.5.3	Output variables	35
	3.5.4	Input/output variables	35
	3.5.5	Global variable and external variable	36
	3.5.6	Temporary variables	36
	3.5.7	Directly represented variable	36
	3.5.8	Access variables	37
	3.5.9	Variable attributes	37
3.6		Variable initialization	38
3.7		Functions	38
	3.7.1	Execution control	41
	3.7.2	Function blocks	42
3.8		Programs	44
3.9		Resource	44
3.10		Tasks	44
	3.10.1	Non-preemptive scheduling	45
	3.10.2	Preemptive scheduling	45
	3.10.3	Task assignment	46
	3.10.4	Configuration	47
3.11		Summary	47
4		Structured text	48
4.1		Introduction	48
4.2		Statements used for assignments	49
4.3		Expressions	49
4.4		Evaluating an expression	50
4.5		Statements	51
	4.5.1	Function block calls	51
4.6		Conditional statements	52
	4.6.1	IF...THEN...ELSE	52
	4.6.2	CASE statement	53
4.7		Iteration statements	53

4.7.1	FOR ... DO	53
4.7.2	WHILE ... DO	54
4.7.3	REPEAT ... UNTIL	54
4.7.4	EXIT	54
4.7.5	RETURN	55
4.8	Implementation dependence	55
4.9	Summary	55
5	Function block diagram	57
5.1	Introduction	57
5.2	Basics	58
5.3	Methodology	58
5.4	Signal flow	60
5.5	Feedback path	61
5.6	Network layout	61
5.7	Function execution control	62
5.8	Jumps and labels	62
5.9	Network evaluation rules	63
5.10	Summary	64
6	Ladder diagrams	65
6.1	Introduction	65
6.2	Basic concept	66
6.3	Graphical symbols used in ladder diagram	66
6.4	Boolean expressions using ladder diagrams	69
6.5	Integrating functions and function blocks within ladder diagrams	70
6.6	Feedback paths	71
6.7	Jumps and labels	72
6.8	Network evaluation rules	72
6.9	Portability	73
6.10	Summary	74
7	Instruction list	75
7.1	Introduction	75
7.2	Structure of IL programming language	75
7.2.1	Basics	75
7.2.2	Instruction structure	76
7.2.3	Comparison with Structured Text (ST) language	77
7.2.4	General semantics of IL expressions	77
7.2.5	Modifiers for deferred execution	78

7.2.6	Other modifiers	79
7.3	Calling functions and function blocks	80
7.3.1	Function block - Formal call with an input list	80
7.3.2	Function block - Informal call	80
7.3.3	Function block - Call with an input list	81
7.3.4	Calling a function - Formal call	81
7.3.5	Calling a function - Informal call	81
7.4	Portability and other issues	82
7.5	Summary	83
8	Sequential Function Chart (SFC)	84
8.1	Introduction to the basic concept of SFC	84
8.1.1	Structure of SFC	85
8.2	Steps	89
8.2.1	Initial step	89
8.2.2	Normal step	89
8.3	Transitions	91
8.4	Actions	92
8.5	Action qualifiers	95
8.6	Action control function block	98
8.7	Execution rules	98
8.8	Design safety issues	99
8.9	Top down design	100
8.10	Summary	101

An introduction to IEC standard 1131 part-3 on PLC programming

This chapter contains information on the use and growth of programmable controllers in industry, the basic problems in the earlier approach adopted for programming these devices and the move towards development of standards for programming. It also discusses the contribution of the standard in improvement of software quality, productivity and maintainability.

Objectives

On completing the study of this chapter, you will learn:

- The basics of Programmable controllers and their role in modern industry
- The need for standardization of PLC languages
- A review of the programming approach prevailing before the evolution of the standard and its shortcomings
- The features of IEC 1131-3 and its contribution towards qualitative improvements to control software
- Move towards open vendor independent systems and software portability

Note

Before we go further, we will get a few basic aspects clear in our minds. The **International Electro-technical Commission** (IEC for short) is the Geneva based international standards making body, which formulates standards for electrical and electronic equipment. These are adopted both within Europe and in most other industrial nations of the world and integrated into their national standards (incorporating regional variations where required). IEC 1131 is the standard relating to programmable controllers. Part 3 of this standard deals with the languages used for programming these devices and is commonly referred as IEC-1131-3. Even though IEC has renumbered its standards since 1997 by prefixing the numeral 6, we will refer to it by the earlier designation of 1131-3, which is still widely used in the industry rather than 61131-3.

The standard IEC-1131 is organized as follows:

Part	Title
1	General information
2	Equipment requirements and tests
3	Programmable languages
4	User guidelines
5	Messaging service specification
6	Communication via fieldbus
7	Fuzzy control programming
8	Guidelines for implementation of languages for programmable controllers

The standard uses the acronym of PC while referring to Programmable Controllers, but in deference to the common use of this abbreviation for the Personal Computer, we will use the term **PLC** (Programmable Logic Controllers) in this manual instead of PC. This is in spite of the fact that the scope of present day programmable controllers extends beyond the conventional interlocking function and includes highly complex control requirements.

1.1 Development and growth of Programmable Controllers (PLC)-An introduction

The PLC is now an indispensable part of any industrial control system. Originally developed in the late 60's to serve the automation needs of the automobile industry in USA, PLCs have grown much beyond this sector and today it is difficult to name an industry segment that does not use a PLC. The initial purpose was to replace hardwired relay based interlocking circuits by a more flexible device. The flexibility came through the programmability of the device, which made it possible to use the same basic hardware for any application as well as the ability to quickly change the program and modify the behavior of a circuit. This obviously, is not possible with a hard-wired relay logic circuit.

Thus, the original PLC had:

- Inputs in the form of contacts (called as digital inputs)
- A processor
- A software to control the processor
- Outputs in the form of contacts, referred as a digital outputs (or sometimes as voltages to drive external relays)

The **Inputs** and **Outputs** (called as I/O) were grouped in printed circuit boards, usually plug-in type modules each containing circuits for several inputs or outputs. Such modules grouped together in rack formation along with the Processor module, the power supply module etc. form the hardware of the PLC. Large PLC configurations usually contain several additional racks containing I/O cards daisy chained with the main PLC. More complex systems have redundant power supply modules and additional processors to increase the processing power or to execute multiple tasks simultaneously.

The PLC market thus comprises various sizes of configurations:

- Micro PLC's of up to 100 I/O's
- Small PLC's of between 100 and 200 I/O's.
- Medium PLC's of up to a 1000 I/O's.
- Large PLC's of more than 1000 I/O's

PLC's are now extensively used in many industrial sectors including petrochemicals and food processing and are largely replacing conventional devices in almost all fields of activity.

As the use of PLCs grew, they became more versatile and started including the capabilities of 3 term PID controllers with analog inputs and outputs in addition to the combinational logic systems of hardwired circuits. The analog input and output signals usually follow the 4 to 20 mA signal standard, also developed in the 60's and which have become the de-facto standard of the instrumentation industry.

Also, as the equipment, which the PLCs served to control became complex, with several of them (each served by its own PLC) acting in tandem, it became imperative to connect them together and share the information between them. Communication links thus came to be a part of the modern PLC system. Figure 1.1 shows a typical PLC system incorporating several of the features cited above.

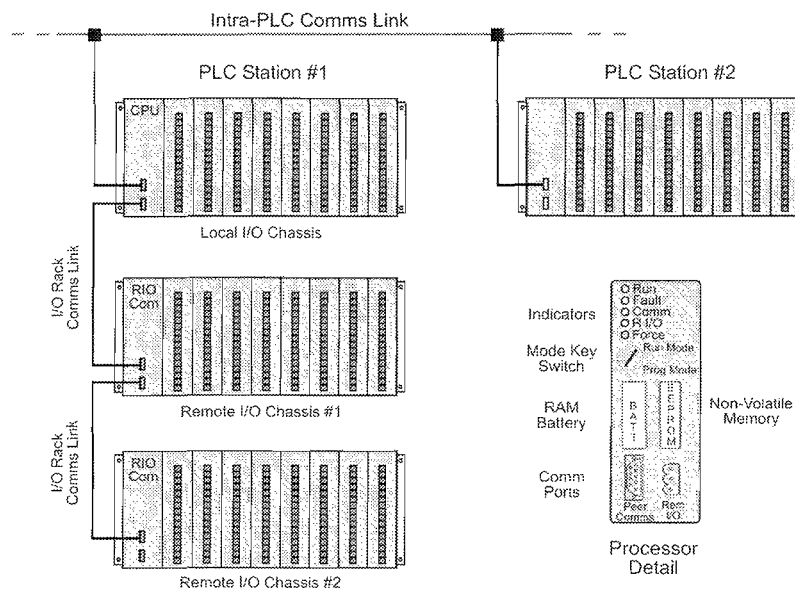


Figure 1.1
A typical PLC system

1.2 Need for standardization in programming approach

The software used in the early PLCs was of the ladder diagram type, which closely resembles the circuit diagram with which all electrical engineers are familiar and still remains one of the most popular PLC programming languages. (We will see more about this language later in this chapter). The modern PLC system has however grown far beyond its initial capabilities as a programmable logic controller and needed more versatile tools for programming. The simple ladder diagram method of programming was

unequal to this task and had to be supplemented. The resulting multiplicity of languages and sometimes dialects of a basic language became too complex for users to handle, with each vendor's product requiring use of their proprietary programming language/tools. Interoperability of PLC's of different vendors also caused problems of achieving integrated control.

The development of **MAP** (Manufacturing Automation Protocol) by General Motors was an initiative to enable communication between the PLC's of diverse manufacturers. More than the standardization of programming languages, the MAP initiative's main objective was communication between PLC's. The MAP standard could achieve this objective but at a very high hardware cost and still had performance limitations.

PLC manufacturers realized that the future growth of PLC and their widespread use in industry would not be possible unless the fundamental issue of program portability is addressed. Thus started a move for a uniform programming approach to be adopted by all the vendors through a basic programming standard. While certain additional capabilities or extensions could be built-in to their product by different vendors, the basic features were to be uniform thus ensuring portability of code and interoperability. IEC-1131-3 is a result of this effort and has been evolved on a consensual basis by largely adopting the prevalent programming practices of major manufacturers in the PLC industry.

Another standardization initiative is by the **Instrument Society of America (ISA)** whose Fieldbus is an attempt to facilitate interconnection of devices distributed in the field such as pressure transmitters, temperature controllers, valve positioning systems etc. Though some of the issues of the structure of internal software of these devices are addressed in the fieldbus standard, the standard does not cover languages used for their programming.

1.3 Drawbacks in conventional programming methodology

As discussed in the previous section, most PLCs use some form of ladder Diagram based programming a typical example of which is given in figure 1.2 below.

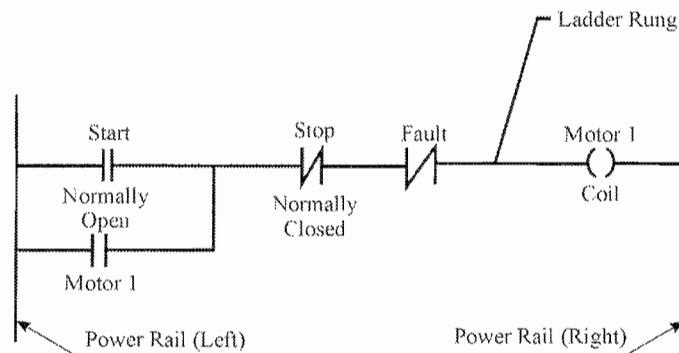


Figure 1.2
A typical ladder diagram

Note how similar this diagram is to the conventional circuit diagram and how easy it is to follow its action. The Diagram describes the logic used for starting a motor direct on line. START and STOP are command inputs received from a control station. FAULT is a signal from protection scheme of the motor. When START is ON it causes the output COIL to pick up. The START and STOP commands are of momentary (pulsed) type and the status of this output is used to lock the output in the status 'true' till STOP or FAULT inputs change status. In addition to being simple, many PLCs also give a dynamic display that shows the status of all these inputs and outputs in real time during the running of the

system. Any malfunction both in the external signals or in the program itself is thus easily identified and corrected.

However, this programming approach has a number of limitations in the context of the modern PLC. We will discuss the areas where conventional programming approach proves inadequate in some detail below.

The limitations are:

- Lack of software structure
- Problems in reusability
- Poor data structure definition
- Lack of support for describing sequential behavior
- Difficulty in handling complex arithmetical operations

Software structure

One of the main programming requirements when dealing with complex control systems is to be able to break down the task into a number of smaller, less complex problems and establish clear interconnections to one another. These individual pieces of code are called program blocks or program units. Since these program units may be coded by different programmers and used in different parts of a control system by others, care should be taken to ensure that internal registers and memory locations of a subroutine do not get changed inadvertently by another program block as a result of faulty code. This needs the data to be properly encapsulated or hidden, which is not possible with the ladder programming approach. This makes use of this program technique difficult for complex tasks as any errors can result in catastrophic behavior of the control system.

Software reuse

In many control problems, one finds that certain logic gets repeated in a number of places. With the ladder diagram programs it is necessary to duplicate the same circuit over and over again. This makes the memory usage and the program execution inefficient. For example, the basic motor starting scheme cited in figure 1.2 may get repeated for a number of motors in a system. Arranging the logic sequence of this control in a block, which can be invoked many times (with minor variations and changes in the input/output designations) would simplify the program greatly. Facility for such code reuse is usually limited in conventional ladder diagram programs.

Data structure

In the conventional approach to programming, digital data (both input and output) are represented as single bits. Analog data is kept in the form of registers, which are generally 16 bits long. In this approach, there is no facility to represent related data in a group in the form of a predetermined structure.

Modern programs approach control problems using object orientation. For example, pressure sensors may be represented as a class of objects with each different sensor being an instance of this class. A pressure sensor may have certain data associated with it. These can typically be: the current value of pressure, a set point for the pressure, a time value for setting an output flag when the set value of pressure is reached, a digital alarm output etc. It is possible to carry out a set of logical operations (such as generating an alarm in the event of a set value of pressure being exceeded beyond a set interval of time) by using the values in the associated data structure. It is possible to use this data block for different pressure sensors (each an instance of the class) by changing the data object's contents. To be able to do this, the data values associated with each sensor must have a unique name but all of them will have a common data definition. In PLC programming

terminology, this data class is called as a Data Structure with each instance of the sensor being represented by a variable.

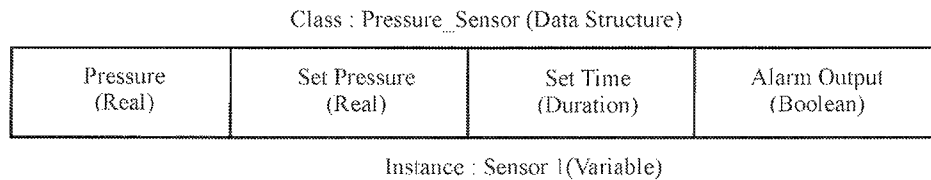


Figure 1.3

Data structure for pressure sensors

Normally, in the conventional ladder programming methodology, the different pieces of data described above are spread throughout program (and the PLC memory) and because of this, data violations can occur easily. The lack of facility for defining a data structure in the conventional ladder programming method will therefore impose constraints in implementing object oriented control strategies.

Support for sequential operations

Many industrial controls perform operations as per a set sequence, particularly those relating to automate operation of processes or machinery. Such operations involve:

- An initial step
- A transition with an associated condition; On fulfilling the condition, deactivate the previous step and go to the next step
- The subsequent step where a set of operations will be performed
- The next transition followed by the next step and so on till the end of the sequence

Representing a sequence of this nature logically in a ladder diagram is somewhat cumbersome. We will illustrate this by an example. A typical chemical batch process for a reactor vessel works in the fashion described below. (Refer to figure 1.4).

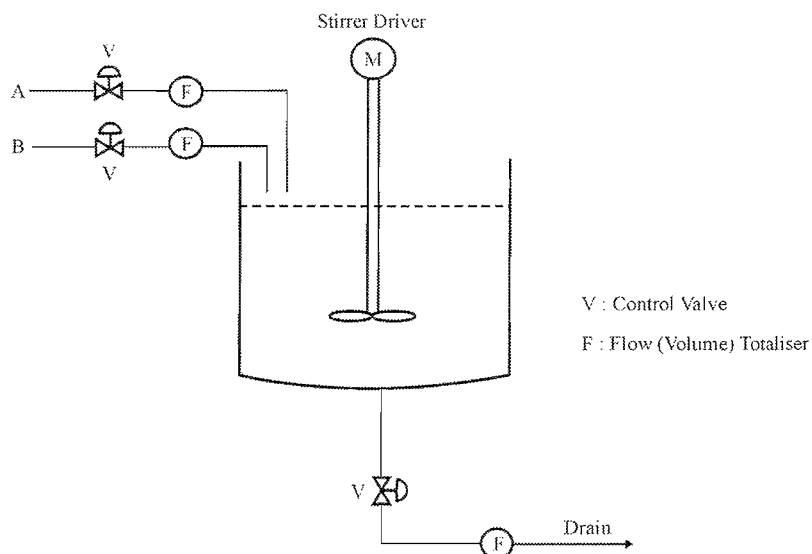


Figure 1.4

A typical chemical process

The process sequence goes like this:

1. Start of process
2. Check readiness of all systems. If ready, go forward
3. Open valve for reagent A
4. Measure volume of flow of A and compare against set value
5. Close A when the set value is reached and open valve for reagent B
6. Measure volume of flow of B and compare with set value
7. Close B when the set value is reached. Start stirring. Start timer
8. Check for time to reach a predetermined set value
9. Stop stirring when the set value is reached. Reset timer. Open Drain valve
10. Check flow and compare to sum of volumes of A and B
11. Close drain when total flow is equal to the sum. Reset all flow sensors. Go back to step 1

We may introduce further complexities in this process by incorporating additional parallel steps. For example, it may be required that a certain temperature needs to be maintained during step 7. To do this we may introduce a sequence for monitoring the temperature and switching an electrical heater on and off at certain temperature limits.

To represent the sequence of steps 1 to 11 logically in a ladder diagram, we may need to arrange the ladder rungs in different groups. One group consists of the transition checks (represented in s. no. 2, 4, 6, 8 and 10). The next group consists of the transition states to signify which step is currently active. (In the above example the steps are 1, 3, 5, 7, 9, and 11 with one of them being active at any point of time.) The third is the step actions, i.e., perform certain predetermined tasks at each step as dictated by the process. Based on this approach, the above example can be represented by the following ladder diagram in figure 1.5.

While this looks fairly straight forward, the complexities of the nature cited in the above example, with additional parallel action sequences, will make the process more difficult to represent and understand if we have to extend the above ladder diagram program. (We would invite the readers who may be familiar with creating ladder logic circuits to try doing this to have a ‘feel’ of the programming limitations this method imposes).

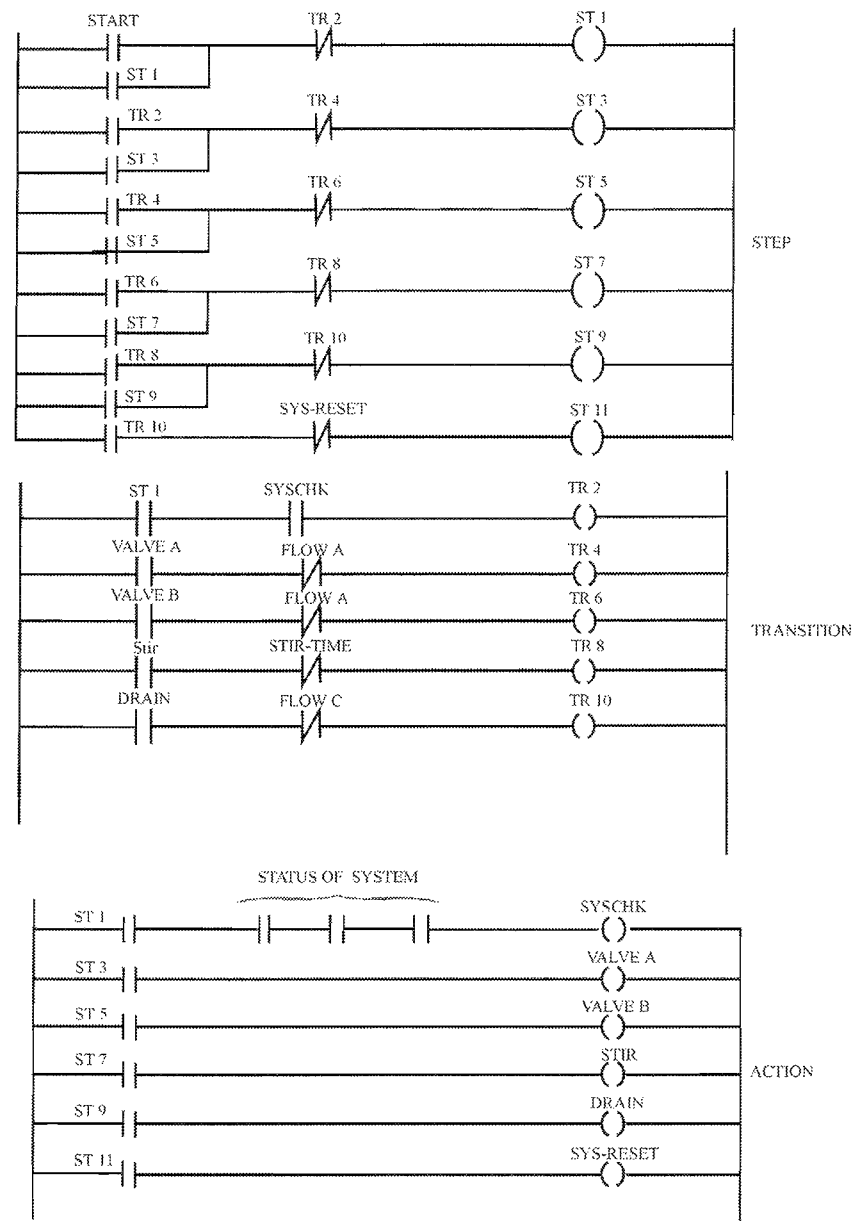


Figure 1.5
Ladder diagram of a typical batch process

Execution control

Execution control is another aspect of process control systems, which calls for more sophisticated programming methodologies. To understand this, we have to first look the sequence of operation of a PLC system. Refer figure 1.6 below.

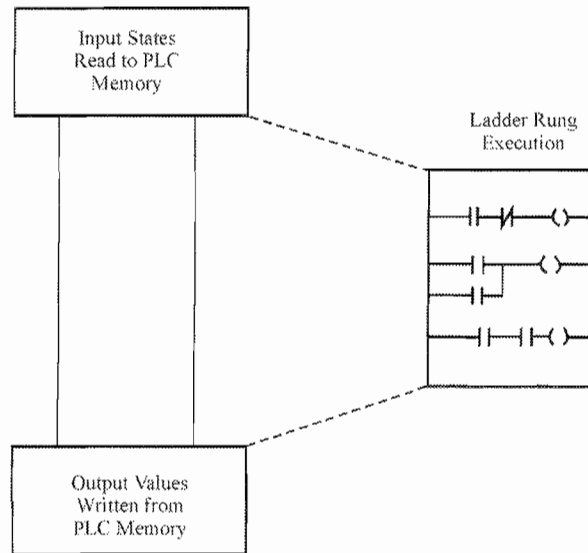


Figure 1.6
PLC operation sequence

PLCs execute a program in a cyclic order. To begin with, all input values are read via the I/O modules and the status stored in PLC memory. Next the ladder rungs are executed starting from the top and proceeding downwards till all rungs are covered. The output values are stored in the PLC memory as each rung is executed. Finally all the output values held in the memory are written to the physical outputs in a single operation. The time of executing the entire program thus depends on the length of the program and the complexity of the logic. Generally, longer the program, higher is the time for each pass of execution (scan cycle time). This makes the behavior of such a system non-deterministic. That is, there is no guarantee that when there is a change of status of a particular input, the actions to be taken as a result will be performed within a stipulated time. (Remember, we are talking of milliseconds here).

While the delay may be acceptable in many cases, there may be situations where a non-deterministic behavior can cause problems. For example, in a process, a particular condition may call for extremely fast sequence of corrective actions without which catastrophic failures may occur. To delay such corrective action for a whole scan cycle time (the maximum possible delay) may not be an acceptable option. If it is required that the action is initiated within 100 milliseconds and the PLC has a scan rate of 1000 milliseconds, the objective is NOT going to be achieved. The only way this can be done is to divide the program into different sections and arrange the execution in such a way that critical sections are scanned at faster intervals. This kind of control is generally difficult to implement in the ladder diagram approach.

Similarly, implementing PID control functions in PLCs for processes requiring fast control behavior will need more sophisticated programming approach. The non-deterministic nature of the simple ladder diagram program can give rise to problems of control and a change in the program due to, say the addition of a few rungs of interlock conditions, may influence the way the system performs its control function.

Arithmetic operations

It is possible that certain logic steps in a control system may require arithmetic operations to be performed. For example, in the control system described in the earlier section, step no. 10 involves an addition of two set values to decide when the transition to the next step should occur. Ladder diagram programming in most implementations can perform such

PLC software architecture

This chapter outlines the software architecture proposed in part 3 of IEC-1131 standard and the salient concepts of the software model. The strengths of the software approach adopted are also discussed in detail.

Objectives

On completing the study of this chapter, you will learn:

- The principal quality attributes of a software
- The software architecture proposed by the standard IEC-1131 Part 3
- Definitions of the IEC software architecture components and details of these Component parts
- Execution control
- Mapping software model to real life systems
- How applications and POU's ensure hierarchical approach
- Communication model as per IEC-1131

2.1 Software quality attributes

Software used in an industrial control PLC is like any other computer application. Its quality will depend on the following factors:

- Capability
- Availability
- Usability
- Adaptability

We will review these aspects in detail below.

Capability

The software should possess an adequate degree of responsiveness compatible with the control application for which it is used. It should be able to schedule tasks in the manner

required by the control application without compromising system safety aspects. The hardware that it runs on should be able to handle the program, variables used and data required to perform the control functions without memory overflow.

Availability

Like any hardware system, software too should perform its functions with a high degree of availability. It should be robust enough to continue operating without crashing and with a high **Mean Time Between Failures** (MTBF) value. It should be easy to troubleshoot in the event of a failure so that the **Mean Time To Repair** (MTTR) is low. This calls for structured program architecture, which is easy to understand with well-commented code. It should withstand intended and unintended actions by users without malfunctioning and should be secure from external threats.

Usability

Any software application should be user-friendly and should have an easy learning curve so that, users are able to acquire the necessary degree of familiarity to work with the software quickly. The human interfaces to the system must be clear, unambiguous and intuitive with graphical devices used for interaction.

Adaptability

Improvements during the life of the software must be easy to implement so that its functionality can be extended and the life of the software prolonged to the extent possible without major upgrades. The software should offer portability to another platform without major code changes and should be easily reusable in the new environment. Any software has a finite life and will need to be upgraded in tune with technology advancements at some point in time. When this happens, it should be possible to upgrade it with little effort and without any loss of functionality.

2.2 IEC software architecture

The standard IEC-1131-3 has been evolved keeping in view the attributes discussed in the section above. Adopting the software model proposed in the standard thus helps to develop software of good quality, which would benefit the software developers, software maintenance personnel and the system users alike.

We will see in this section the model architecture proposed in the standard. The developers of the IEC standard had the objective of resolving the differences in the programming approach of the then existing PLC control systems offered by major vendors. In addition, they have also attempted to foresee the impact that emerging technologies and future control requirements will have on the PLC software and language structures. The software architecture must be able to accommodate these future requirements without major structural changes. The software model may seem to be somewhat elaborate going by current hardware architecture but has to be looked at in the context of possible future demands.

A program requires the following types of interfaces in order to be executed by a PLC:

- I/O interface
- System interface
- Communication interface

I/O (or Input /Output) interfaces are the PLC's connection to the real or physical world. The inputs can be contacts from various external devices such as control switches, limit switches, push buttons, contacts associated with instruments and so on. These are the

‘digital’ inputs. Other inputs can be from measuring instruments and will be some kind of analog values such as 4-20 mA, 0-10V etc. These are the analog inputs to the PLC. Digital outputs are usually contacts for driving relays, contactors etc. and analog outputs will be 4-20mA or other signals that can be used as set points for devices to carry out position control, speed control etc.

System interfaces refer to the system services required to ensure the execution of a PLC program by the hardware. These are usually in the form of specific PLC hardware and embedded firmware, which enable invoking of programs, initialising the programs and executing them.

Communication interfaces facilitate exchange of data with other devices and PLCs as well as operator stations and other output devices.

The basic software architecture proposed by the standard is shown in figure 2.1 below.

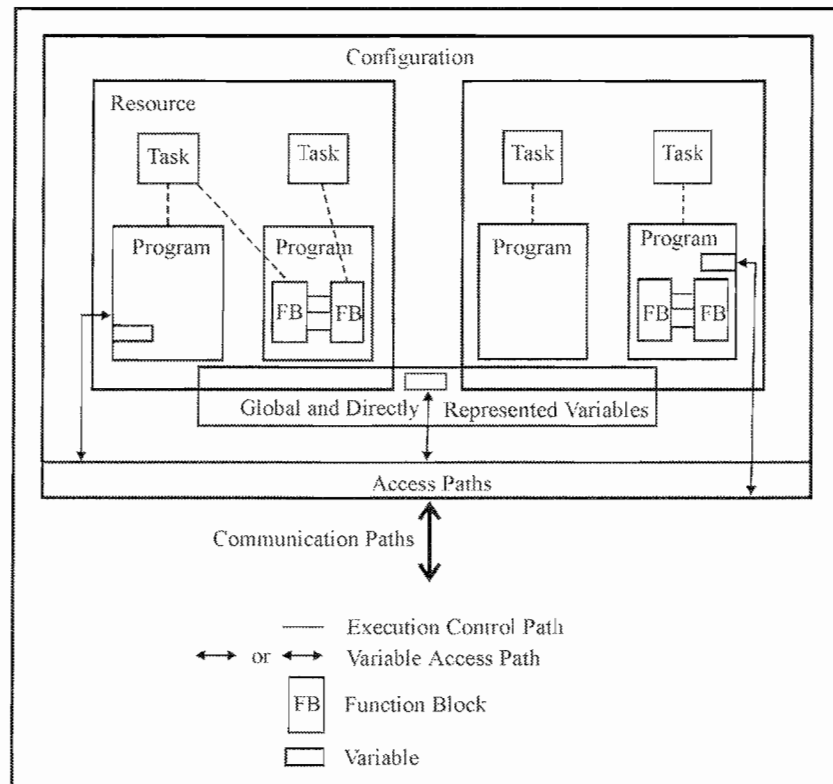


Figure 2.1
The IEC software architecture

2.3 Component parts of IEC Software architecture

As seen in the above diagram, the software model suggested in IEC-1131-3 contains a number of elements arranged in a hierarchical order. We shall review these components in the following paragraphs.

2.3.1 Configuration

Definition (1.3.17)

A language element corresponding to a programmable control system as defined in IEC-1131-1.

At the top is the **configuration**, which can be equated with the software for a complete PLC system. In a control system spanning across multiple machines, each with its own PLC, with all of them interacting with each other, there will be multiple configurations, with the software for each PLC considered as one configuration. These configurations will communicate with each other through special variables or via defined access paths using the communication function blocks specified in Part 5 of the IEC 1131 standard.

2.3.2 Resource

Definition (1.3.63)

A language element corresponding to 'signal processing function' and its 'man machine interface' and sensor and actuator interface functions', if any, as defined in IEC-1131-1.

Next in the hierarchy is the Resource. A configuration can have one or more resources. A resource is the engine that supports and directs the execution of one or more programs. In other words, a program can function only when it is loaded into a resource. A resource can exist in any device and that includes a PLC. For example, a resource can also function in a PC and can support a PLC program for simulation purpose. A resource provides the interface between the different I/O devices, the **Human-Machine Interface (HMI)** of the PLC and the PLC program.

Just as a configuration can be equated to a PLC, a resource can be considered as the software counterpart of a processor within a PLC. A multi-processor PLC can thus have multiple resources, each one of them associated with a given processor card. A resource in a PLC is an independent component and will not be under the control of any other resource. Thus it is possible that a PLC can start and execute a number of independent programs simultaneously.

2.3.3 Task

Definition (1.3.75)

An execution control element providing for periodic or triggered execution of a group of associated program organisation units.

We saw in the paragraph above that a resource executes multiple programs. It does so using software elements called tasks. A task can invoke one or more programs or function blocks to be executed at a certain periodicity or by triggering them when a set of conditions is fulfilled. The programs or function blocks are executed once when commanded to do so by the task.

2.3.4 Program

A program consists of a logical assembly of all programming elements and software constructs necessary for the intended signal processing actions required for the control of a machine or a process by a programmable control system. It contains instructions to be executed by the PLC hardware in a predetermined sequence. The instructions can be in the form of function blocks interconnected together to form a sequence of actions. As discussed in the previous chapter, a program obtains or 'reads' inputs from the PLC's I/O devices, uses these inputs while executing the instructions and stores or 'writes' the results to the PLC's outputs. Programs are initiated and controlled by tasks. A program has to be assigned to a specific task and the task must be configured to execute periodically or triggered upon certain conditions being fulfilled.

2.4 Functions and function blocks

Definition of FUNCTION (1.3.28)

A Program Organizational Unit which, when executed, yields exactly one data element (which may be multi valued e.g., an array or structure) and whose invocation can be used in textual languages as an operand in an expression.

Definition of FUNCTION BLOCK (1.3.29)

An instance of Function Block Type

Definition of FUNCTION BLOCK TYPE (1.3.30)

A programmable controller programming language element consisting:

- *Definition of a data structure partitioned into input, output and internal variables*
- *Set of operations to be performed upon the elements of a data structure when an instance of the function block type is invoked*

Function block is the most important concept of this standard, as this is the feature, which makes a hierarchical program structure possible. As we saw in the previous paragraph, a program consists of various function blocks joined together to form a sequence of activities. It is also possible for function blocks contained within a program to be directly controlled by a task in which case their execution can take place independent of the execution of the parent program. (Refer to figure 2.1.) Function blocks help to organize a complex problem into simple steps that can be managed easily. The program thus gets properly structured. The other main feature of a function block is that certain common control functions with identical logical sequence can be defined under a common function block type and instances of this function block type can be created and used for a specific control requirement. What is normally referred to, as a function block is actually a function block instance? For example, a function block type, which imitates the action of a PID controller, can be instantiated for various PID control loops just by changing the input and output designations and other control parameters.

A function block defines data as a set of inputs and outputs. The function block can store the results of its operation in a particular location within the data structure. The outputs can also be used to modify internal variables, which can be local or global. The values of all these inputs and outputs are maintained as a variable of a given data structure, unique to each function block instance. The inputs of the structure can be thus derived from the outputs of other function blocks and in turn its output can be used as inputs for other function blocks or in its own next evaluation cycle. This ability to remember the outputs and use them again as inputs by itself or by other function blocks is what makes them extremely versatile. By using global variables to share data, communication between different programs in a configuration becomes possible. Also, transfer of data between different configurations is enabled using designated variables.

Standard function block types have been specified in the IEC-1131-3 standard for commonly used requirements such as R-S bistables, timers, real time clocks etc. But the interesting feature is that a programmer can create new function blocks and these can be based on other existing function blocks and other software elements.

Functions are different from function blocks. Though the software architecture shown in Figure 2.1 does not specifically indicate this component, functions are a form of program organisation unit defined by the standard. A function can have many inputs but gives a single output by performing certain operations on or using the inputs.

For example, a mathematical function such as SIN () finds the value of say SIN (A) where A is an input and gives the result of the calculation as an output value. An AND function may have two or more logical inputs and will give an output after performing a logical AND of the inputs. Its output will be TRUE if all the inputs are TRUE, else FALSE.

We can note two things here. One is that, there is a single output, which is the result of a manipulation of the one or more inputs to the function. The second is that there is no storage mechanism such as a variable within the function and because of this, a given set of inputs always results in the same value of output. In this respect, functions behave differently from function blocks, which may give different outputs because of their ability to use stored values of their earlier evaluations in their processing.

2.5 Local and global variables

Definition of global variable (1.3.34)

A variable whose scope is global.

Definition of global scope (1.3.33)

Scope of a declaration applying to all program organization units within a resource or configuration.

Definition of local scope (1.3.48)

The scope of a declaration or label applying only to the program organization unit in which the declaration or label appears.

Variables can be declared within programs and function blocks as per the standard. Local variables can be named within a configuration, program, function block or function. A variable is local by default to the software element in which it is declared. But if a variable is declared as global in a program, it can be accessed from within all the software elements inside that program including nested function blocks. When global variables are declared in a resource or configuration level, they will be accessible to all the software elements within that resource or configuration. Global variables thus provide a method of sharing data between programs or between elements of a program.

2.5.1 Directly represented variable

Definition of direct representation (1.3.23)

A means of representing a variable in a programmable controller program from which a manufacturer specified correspondence to a physical or logical location may be determined directly.

Directly represented variables are used to address directly memory locations within a PLC. Input or output memory locations can be represented in the format:

% I W 75 (Input) or % Q 100 (Output).

These variables can only be declared and accessed within programs but not within function blocks, as such use would make the function blocks non-generic. Even in programs, use of many directly represented variables renders the program less portable as the memory locations can change between applications. These variables provide yet another way of exchanging information between different programs.

2.5.2 Access paths

Definition (1.3.2)

The association of a symbolic name with a variable for the purpose of open communication.

As the definition implies, the main purpose of an access path is communication. Transferring data and information between different IEC configurations is provided by the use of designated named variables, which can be accessed by remote configurations. These special variables which provide the access path for communication are declared using the name format: VAR_ACCESS and will formally declare the interface that a configuration presents to other remote configurations. The standard does not define the physical aspect of communication, which can be the Ethernet or Fieldbus or any other proprietary protocol. It just assumes that some means of communication is available.

2.5.3 Execution control

Configurations and resources control the execution of Program organisation units contained in them. Starting of a configuration results in the following actions:

- Global variables are initialized
- Resources started
- Variables within the resource initialized
- Tasks enabled
- Program elements controlled by the task are executed.

Shutdown of a configuration results in:

- Stopping of all resources
- Tasks in these resources are disabled
- Programs and function blocks stop executing

2.6 Mapping software model to real life systems-Examples

Simple systems with a single processor meant for controlling one machine or a system usually have one configuration, one resource and one program. However larger systems having multiple processors may be represented by a single configuration. Each processor may correspond to a single resource and may control more than one program. It is also possible to have a distributed control system spread across several PLCs connected through a high-speed communication network. In this case it is possible for each of these PLC's assigned to resources, which in turn will be combined under different configuration groups. These models are shown in figures 2.2 to 2.4.

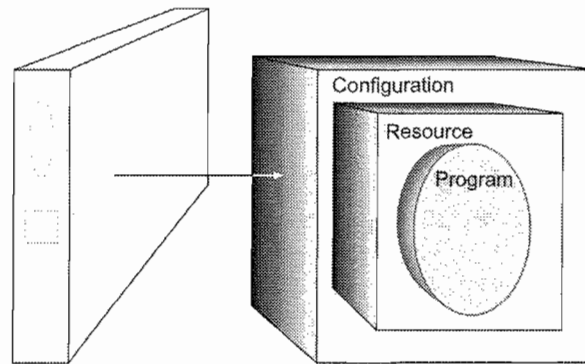


Figure 2.2
Single processor PLC

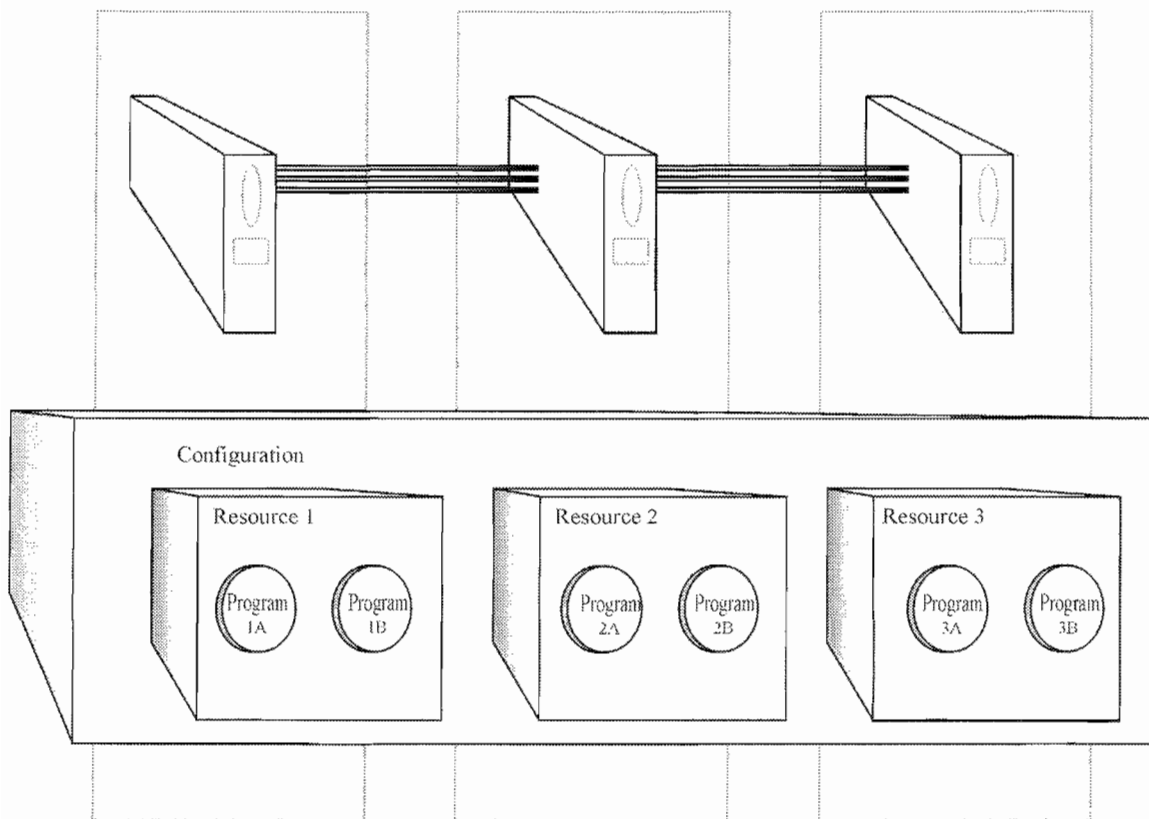


Figure 2.3
Multi processor PLC

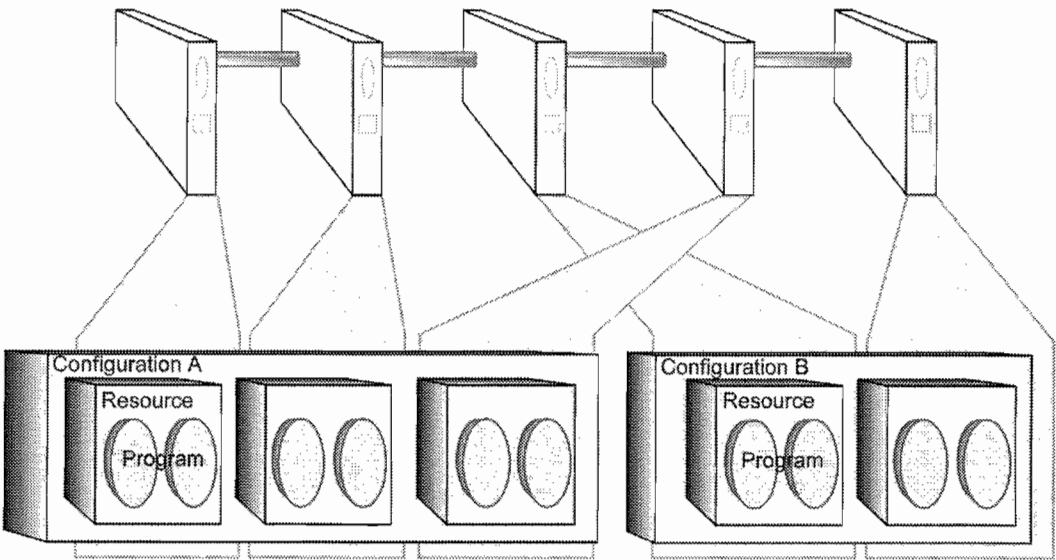


Figure 2.4
Multiple PLCs connected using a network

As can be seen in Figure 2.3, three processors are part of a single configuration and each of the processors represents one resource. Each resource contains two programs. The table (2.1) below illustrates the arrangement.

Processor 1	Processor 2	Processor 3
Configuration		
Resource 1	Resource 2	Resource 3
Programs 1A and 1B	Programs 2A and 2B	Programs 3A and 3B

Table 2.1
Multiple processors shared by a single configuration

In the multi PLC configuration shown in Figure 2.4, five PLC’s communicating over a network form two configurations. Each configuration has multiple resources and each resource multiple programs. Table 2.2 below shows the arrangement.

Processor 1	Processor 2	Processor 4	Processor 3	Processor 5
Configuration A			Configuration B	
Resource 1	Resource 2	Resource 3	Resource 4	Resource 5
Programs 1A and 1B	Programs 2A and 2B	Programs 3A and 3B	Programs 4A and 4B	Programs 5A and 5B

Table 2.2
Multiple processors shared by multiple configurations

2.7 Applications

An application is a solution for the control of a unit of plant. Though this is not formally defined in the standard, is still an essential feature of the software of a large PLC system. Figure 2.5 illustrates.

In this case applications A, B and C are part of a single configuration and include programs spanning across several resources. An application should contain at least one program. An application should include controls that start, manage and shutdown a machine or plant unit and should connect to all the sensors and actuators associated with the machinery or plant. It should perform all combinatory logical interlocking sequences and control loops required for the operation of the machine/plant. In addition to these, it should interact with the operator terminals and communicate with other remote devices.

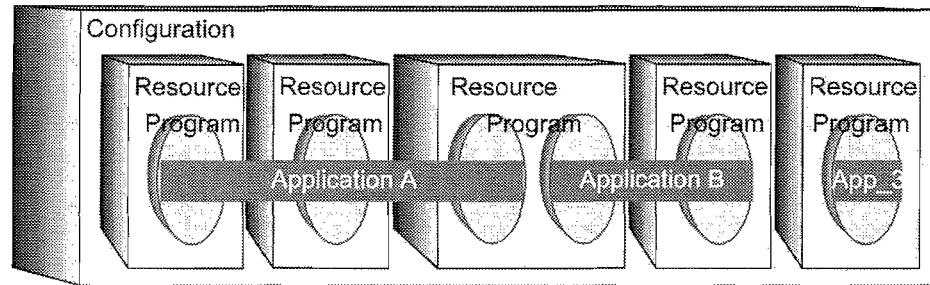


Figure. 2.5
Multiple application environments

2.7.1 Program Organisation Unit (POU)

Definition of POU (1.3.61)

A function, function block or a program.

Note: This term may refer to either a type or instance.

A program organisation unit includes programs, function blocks or functions. We have gone through the details of these elements of language earlier in the chapter. These elements are the basis by which IEC-1131-3 languages provide reusability. A program can be reused in the form of program instances. So can function blocks and functions. Since the basic type is already properly validated, the instances can be ensured to be error free. The only limitation is that these elements cannot be recursively used as in common high level programming languages such as Pascal. In other words, a function block cannot call the same block inside its code. This is so because the behaviour of recursive programs, function blocks or functions cannot be easily verified or validated in real time mode and can yield unpredictable results. Another feature is that instances of a program or function block can be copied in several resources and used for controls that are similar to each other and can be grouped under a common prototype. The table 2.3 below illustrates this concept.

POU type	Replicated as	Remarks
Program Type	Program instance	Macro level reuse for a family (Conveyor line control, Steam turbine control)
Function block type	Function block instance	Coding of reusable control algorithms such as PID control, ramp etc.
Function type	Function	Used for mathematical or logical functions such as SIN (), AND etc.

Table 2.3
POU elements and reusability

2.7.2 Hierarchical design

The design of POU structures using types and instances of program elements ensures that it is possible to adopt both top down and bottom up approach and arrive at a hierarchically clear program. The top down approach decides the overall structure and identifies repeatable module types. These modules are then developed and assembled in the form of instances to make a complete application. Figure 2.6 illustrates this hierarchical approach.

Program type A is the top of the hierarchy (not shown). The program instance A1 of program type A is what is actually stored in a resource (shown in the bottom left hand). This contains an instance R1 of function block type R. This function block type R contains an instance X1 of function block X and so on. This function block type may use one or more functions, which are the smallest program elements in this hierarchy (not shown in the figure).

Usually a PLC system vendor provides a library of functions as well as function blocks for commonly encountered control problems. Creation of custom function blocks using these libraries or using basic code structures is also permitted in most implementations.

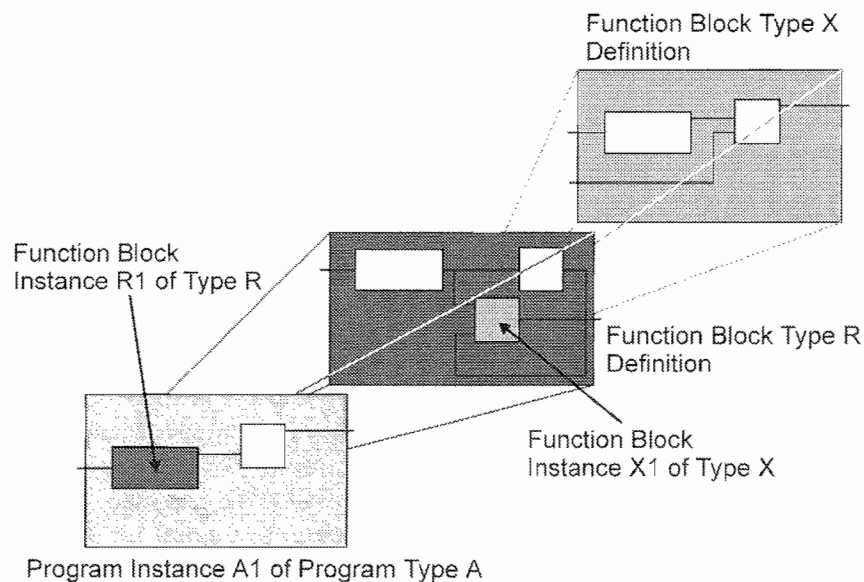


Figure 2.6
Hierarchical software decomposition

2.7.3 Communications

Ability of to communicate data between remote processors is the key to building large control systems and IEC-1131-3 provides ample scope for such communication. Generally, communication can be grouped under two categories, internal and external. Internal communications can either be communication between program elements or between different resources within a configuration. Internal communication takes place using variables within a program as in Figure 2.7 (a) or using global variables within a configuration as shown in Figure 2.7 (b).

External communications between configurations use communication function blocks SEND and RCV as shown in Figure 2.8 (a) or access paths with special VAR_ACCESS type of variables as shown in Figure 2.8 (b).

The function blocks used for external communication are defined in part 5 of IEC-1131 standard.

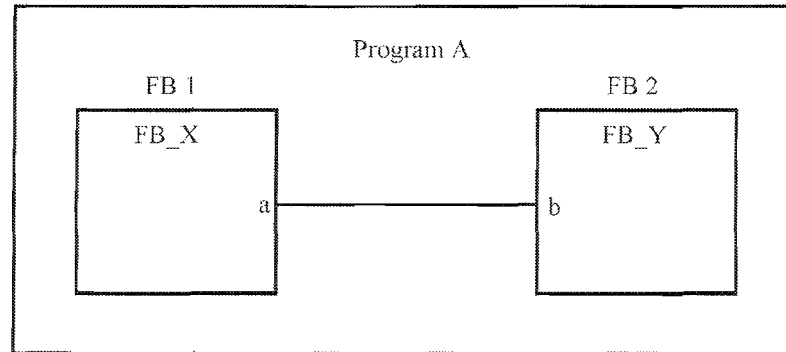


Figure 2.7 (a)
Example of internal communication (within a program)

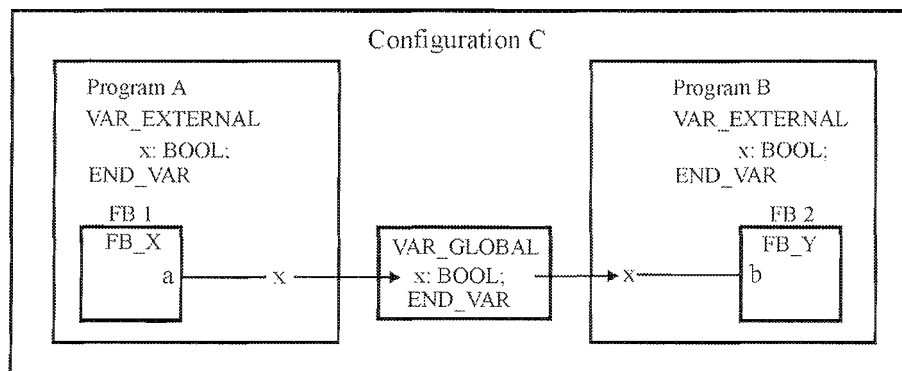


Figure 2.7 (b)
Example of internal communication (within a configuration)

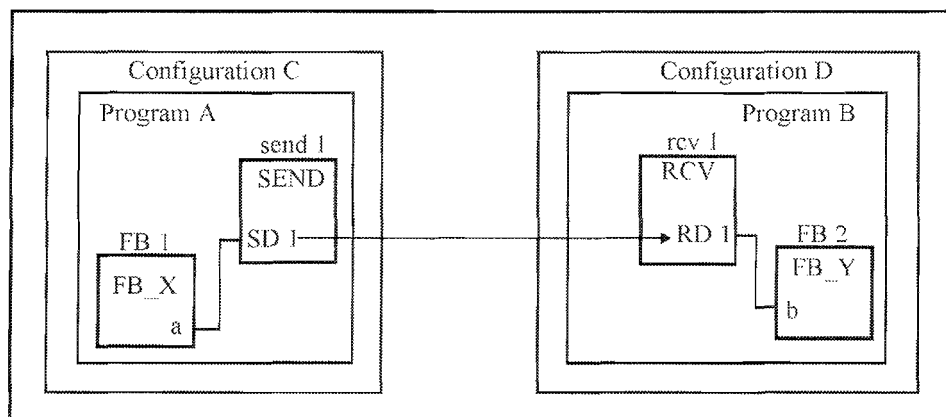
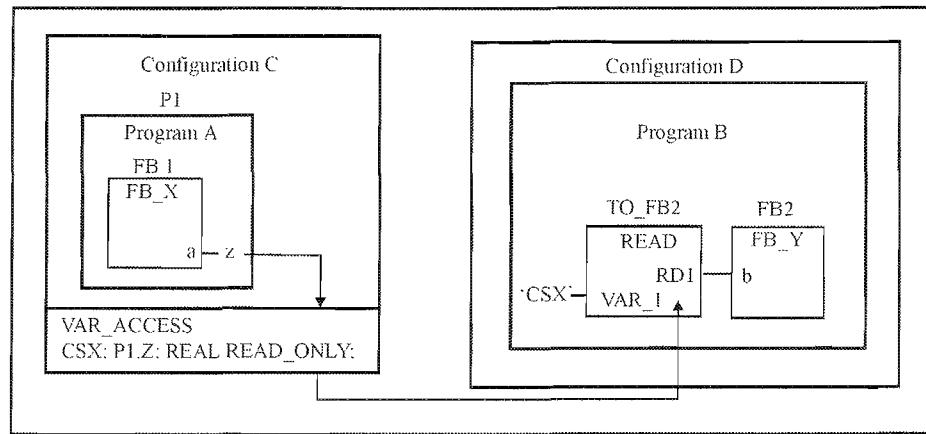


Figure 2.8 (a)
Example of external communication using function blocks



Communication Via Access Paths

Figure 2.8 (b)
Example of external communication using Access Paths

2.8 Conclusion

IEC-1131-3 thus adopts a well-structured, hierarchical software architecture, which corresponds to the hardware architectures commonly used. We saw examples of how real life control software can be mapped to various hardware configurations. We also discussed the various component parts of this architecture and how, using the concept of program organisation units, it is possible to design complex systems by deploying top down and bottom up approach. We also saw the manner in which this approach can ensure software reuse and error free design. The facilities for communication of data flow within a configuration and between a set of configurations provided in IEC-1131-3 makes the development of complex systems possible. The details of such communication were also illustrated.

Common Elements in IEC-1131-3

This chapter outlines the common programming elements, which are used in all the programming languages defined in IEC-1131-3 standard. Knowledge of these elements and the syntax of their use is a must for anyone who wishes to develop control software for PLC applications.

Objectives

On completing the study of this chapter, you will learn about:

- Character sets, identifiers, keywords and comments
- Various data types covered by IEC-1131-3 standard (elementary types and derived types) and their initialization
- Variables and how to declare them in program constructs
- Types of Functions covered by the standard and their execution control
- Usage of function blocks, defining of function block types and declaring function block instances
- Usage of programs, defining program types and declaring program instances
- Usage of resources
- Tasks and scheduling, task declaration and execution of function blocks and programs controlled by a task
- Configuration definition

3.1 Common elements

IEC-1131-3 defines both textual languages and graphical languages. **Structured Text (ST)** and **Instruction List (IL)** are the two textual languages. Graphical languages are **Function Block Diagram (FBD)** and **Ladder diagram (LD)**. Sequential Function charts represent the total flow of a control system and can use any of the above languages embedded in its transitions and action blocks. IEC-1131 provides several common programming elements for constructing PLC control systems based on any of these languages and defines their usage. For example, the variables and data types are common

to all the programming languages and will follow the same standard definitions irrespective of whether they are to be used in ST, IL LD or FBD language. We will learn about some of these common elements and the rules governing their usage here.

We will first describe the basic rules governing:

- Character set
- Identifiers
- Keywords and
- Comments

We will thereafter review in detail the Data types covered in the standard, the different types of variables and how to declare and initialize them. We will then go through the defining and declaration of program organisation units viz., functions, function blocks and programs. Finally, we will learn about resources, tasks and configurations. Complying with the guidelines contained in the standards is important in order to ensure complete portability between systems.

3.1.1 Character set

IEC 1131-3 restricts usage of characters to those of the Basic Code Table of ISO 646. This set contains digits, letters and other characters normally found in any PC keyboard. Alternatives are provided when the same key may be used for different characters in different countries. For examples the vertical bar character '|' is permitted to be replaced by '!'. Use of special alphabet variants specific to a country are not permitted except when allowed by the national extension of the standard. Language element names are case insensitive. However when used in comments in the code or in printable strings the characters are used with case sensitivity. Language keywords are case sensitive and should always be in uppercase.

3.1.2 Identifiers

Identifiers are used for naming different language elements such as variables, new data types, function blocks and programs.

The rules governing the naming of identifiers are as follows:

- A name can contain letters, digits and the underline character '_'
- Lower case alphabets are permitted
- The first character shall not be a digit
- Two underline characters cannot occur together
- There should be no embedded space within a name
- The first six characters should be unique i.e., no two names should have an identical string in the first six characters

3.1.3 Keywords

Keywords are special words used to define program constructs or the start/end of particular software elements (example: FUNCTION, END_FUNCTION). Identifier names should not preferably use keywords even though many compilers would distinguish the two based on context. IEC 1131-3 has standard functions and function blocks, which should be regarded as keywords and not be used as identifiers (e.g., RS, SIN, TON etc.). The set of keywords as defined by the standard are listed in table C-2 of Annex C of the standard.

3.1.4 Comments

As we have discussed earlier, comments in the code serve to explain the flow and purpose of the code so that its understanding and maintenance become easier. Comments are placed between the set (`*` and `*`). Instruction List language puts some restrictions on the placement of comments. Otherwise, comments can be placed wherever a space can be interposed. Nested comments are NOT permitted. Multiple lines of comments are permitted.

3.2 Elementary data types

Modern PLC systems have to handle a wide variety of data. IEC-1131-3 standard recognizes this need and provides for a comprehensive range of data types. Data types can either be Elementary or Derived. Elementary data types are:

- Integer to be used for counters and identities
- Real (Floating point) for arithmetical calculations
- Time (duration) and Time & Date for timers and time triggered functions
- String for text information processing
- Bit String for low level device operation
- Boolean for logic operation

Each elementary data type has various sub types defined in the standard. Also each data type has a set of formats for defining literal (constant) values. We will briefly describe these data types below.

3.2.1 Integer

Integer data type is generally used to denote parameters, which cannot assume a decimal value such as counts and identities (whole numbers). Unsigned integer representation can be used in situations where a negative value is not expected. Signed integers can assume both positive and negative values. For the same number of bits used an unsigned integer can cover larger range of values. The standard defines various types of integers (based on whether signed or unsigned and the number of bits used) and the user can select a particular type depending on the expected range of values likely to be encountered in a given application. The table below lists the integer types covered in the standard.

IEC Data type	Description	Bits	Range
SINT	Short Integer	8	-128 to +127
INT	Integer	16	-32768 to 32767
DINT	Double Integer	32	-2^{31} to $2^{32} - 1$
LINT	Long Integer	64	-2^{63} to $2^{63} - 1$
USINT	Unsigned short integer	8	0 to 255
UINT	Unsigned integer	16	0 to $2^{16} - 1$
UDINT	Unsigned double integer	32	0 to $2^{32} - 1$
ULINT	Unsigned long integer	64	0 to $2^{64} - 1$

Figure 3.1
Integer types as per IEC-1131-3

Examples of use are as follows. For a counter where the count cannot exceed 100, SINT will be suitable. Where large positive and negative values can be expected as in the case of a position encoder, long integer of LINT type can be used.

3.2.2 Real or floating point

There are two types of Real variables as shown in the table in Figure 3.2 below.

IEC Data type	Description	Bits	Range
REAL	Real	32	$\pm 10^{\pm 38}$
LREAL	Long Real	64	$\pm 10^{\pm 308}$

Figure 3.2

Types of real data

The format of these data types shall be as defined by IEC 559. These data types are useful in representing very small to very large numbers, which can take both positive and negative values. They are typically used for analog values from transducers, tachometers etc., for algorithms in closed loop controls and for analog outputs driving position controls. REAL values have a precision of one part in 2^{23} and LREAL values, one part in 2^{52} . REAL literals are used for representing constant values using decimal notations or exponential notation for using very large or small nos. A single underscore character can sometimes be added for clarity as a separator between digits.

Examples are:

12.2345

+11_233.234

-2.43 E -25

0.534 e 22

3.2.3 Time or duration

This data type is used for representing time duration, examples being process control timing and time delays used for generating error alarms. The data is stored using Days, hours, minutes, seconds and milliseconds. The maximum length of duration and precision are implementation dependent.

Time literals are used in Long form and Short forms. In both forms the following letters are used:

d days

h hours

m minutes

s seconds

ms milli-seconds

Examples of short form are:

T#1d4h34m43s22ms

T#35m23.5s

(Note the use of a decimal, which is permitted in the last field of the literal)

Long form is similar to the above but gives improved readability.

Examples are:

TIME#1d_4h_34m_43s_22ms

T#35m_23.5s

The use of underscore characters between the fields shown above ensures better readability.

3.2.4 Date and time data

The date and time data type is very useful for recording the date and time of specified events, for calculation of elapsed periods between specific events and for triggering specific actions at a predetermined time and date. The date and time data can be any one of the following types as shown in the table in Figure 3.3.

IEC Data type	Description	Bits	Usage
DATE	Calendar date	Implementation dependent	Storing calendar dates
TIME_OF_DAY or TOD	Time of day		As real time clock
DATE_AND_TIME or DT	Date and time of day		Storing date and time of day

Figure 3.3

Types of date and time data

When using these data types as literals two forms viz., long and short are used as given in the table in Figure 3.4.

IEC Data type	Short Form	Long Form
DATE	D#	DATE#
TIME OF DAY	TOD#	TIME_OF_DAY#
DATE AND TIME	DT#	DATE_AND_TIME#

Figure 3.4

Form of date and time data in literals

In date literals the format is Year, followed by month followed by date (yyyy-mm-dd) as shown below.

D#2002-09-21 or

DATE#2002-09-21 both of which stand for 21st of September 2002.

TIME_OF_DAY literals use the format hh:mm:ss using a 24 hour scale. Examples are:

TOD#13:10:22.23 or TIME_OF_DAY#13:10:22.23

Date and Time literals are a combination of the above and use the format yyyy-mm-dd-hh:mm:ss.

Examples are

DT#2002-09-21-13:10:22.23 and

DATE_AND_TIME#2002-09-21-13:10:22.23

Which combine the values shown in the two data type examples above.

3.2.5 String

Strings are used for the purpose of storing textual information for batch identities, operator displays and messages to other systems via communication interfaces. The length of information that can be stored is implementation dependent. Booth printable and non-printable characters can be used in a string. All string literals must be framed within single quote sign '. Non-printable characters can be inserted by prefixing the hex value of

character (as two hex digits) by \$ sign. Similarly commonly used control characters should be preceded by \$ sign when used within strings. Figure 3.5 illustrates such cases.

Code	Interpretation
\$\$	Dollar sign
\$'	Single quote character
\$L or \$l	Line feed character
\$N or \$n	New line character
\$P or \$p	Form feed (new page)
\$R or \$r	Carriage return
\$T or \$t	Tab character

Figure 3.5
Control characters within strings

3.2.6 Bit string

Bit string data types are provided for storing binary data which is commonly used for exchange of status information with remote devices and also for low level bit operations for interfacing with PLC hardware. Table in Figure 3.6 below shows the different bit string data types.

IEC Data type	Description	Bits	Range
BOOL	Bit string of 1 bit	1	Logical state
BYTE	Bit string	8	Binary data
WORD	Bit string	16	Binary data
DWORD	Bit string	32	Binary data
LWORD	Bit string	64	Binary data

Figure 3.6
Bit string data types

BOOL data type is used for status data, which can be FALSE (0) or TRUE (1). FALSE and TRUE are reserved keywords and cannot be used for other purposes in a program. Data types of more than 1 bit are used to define the contents of multiple bit data.

3.3 Generic data type

Generic data types are used for variables in functions and function blocks where overloaded I/O is supported. Overloaded I/O refers to the capability of a variable to be used for different data types but which have similar properties. It should be noted that only manufacturer-defined functions can be overloaded but this is not applicable to user-defined functions. PLC manufacturer should list functions that support overloading and ensure that all I/Os of a overloaded function are of the same generic data type. Generic data type will start with the prefix 'Any'. Figure 3.7 shows the hierarchy of elementary data types and how they can be combined under different generic data types.

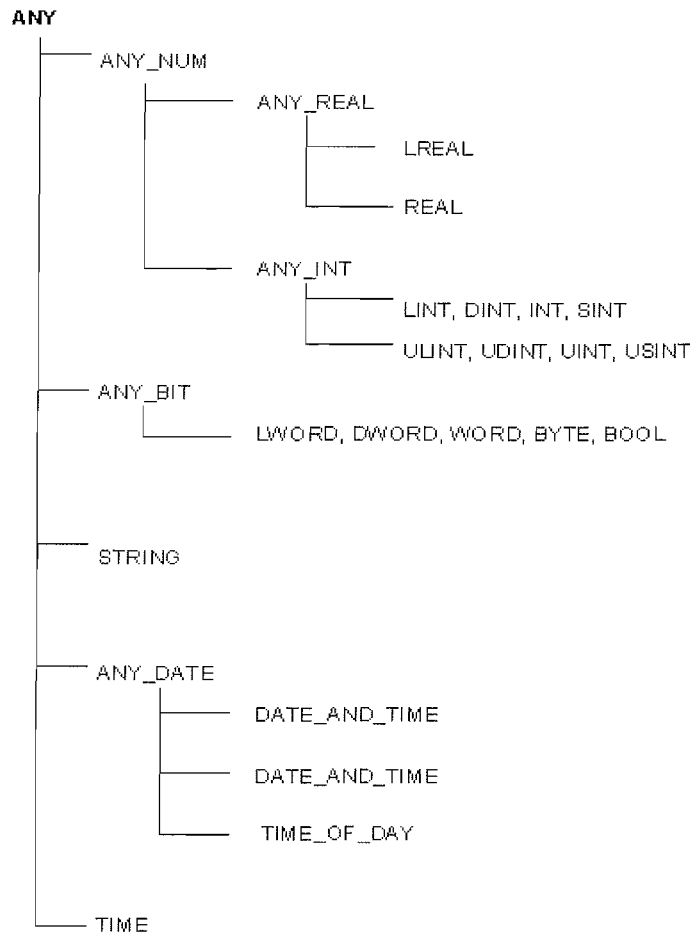


Figure 3.7
Hierarchy of data types

3.3.1 Initial values

The standard defines the default initial values applicable to different data types. For magnitude type of data and bit string type (refer Figure 3.7), the initial value is set to 0. For string data, it is null string. Date values take the initial value of 0001-01-01. All these initial values can be overridden when a variable of a particular data type is declared.

3.4 Derived data types

3.4.1 Derived directly from elementary type

New data types can be defined using the above elementary data types. They are declared as given in the example given below.

```

TYPE
  VOLUME : LREAL;
END_TYPE
  
```

The above statement declares a derived data type called volume, which corresponds to elementary data type LREAL.

3.4.2 Structured data type

This is a category of derived type where a composite data type can be defined using a structure in which each field will correspond to one of the elementary data types. We saw the example of such a data structure for a pressure sensor in chapter 1 (Figure 1.4).

This derived type (named `PRESSURE_SENSOR`) is defined as a combination of the following four different parameters:

- The pressure value as currently available from the sensor
- A set value of pressure
- The time for which the pressure can exceed the set value
- An alarm output

This can be declared as follows.

`TYPE PRESSURE_SENSOR:`

`STRUCT`

```

CURRENT_PRESSURE  :    PRESSURE;
SET_PRESSURE      :    PRESSURE;
SET_TIME          :    TIME;
ALARM             :    BOOL;
```

`END_STRUCT;`

`END_TYPE;`

Note the use of a derived data type called `PRESSURE` while declaring the new structured data type `PRESSURE_SENSOR`. This derived type must earlier be declared using an elementary data type.

3.4.3 Enumerated data type

It is possible that a particular variable may only take certain specific values, in which case these values can be explicitly specified using an enumerated list. Let us say, a Selector switch has two modes viz., auto and manual. The variable representing the status of the selector switch can be defined as an enumerated type having the two states specified above. The data type for this variable can be defined as follows.

`TYPE`

`SWITCH_MODE: (AUTO, MANUAL);`

`END_TYPE`

The above statement means that the data type `SWITCH_MODE` can take either of the values `AUTO` or `MANUAL`. It may be possible that these literals `AUTO` and `MANUAL` may occur for other data types also. As such, while referring to the enumerated literal, it should be prefixed with the enumerated data type followed by `#`, for e.g., `SWITCH_MODE#AUTO`.

3.4.4 Sub range data types

Sometimes it may be necessary to restrict the range of values that a variable may be assigned. Let us say the speed range of a motor can vary only between 600 RPM and 1000 RPM. The data type for the speed can be defined as:

`TYPE`

`MOTOR_SPEED : UINT (600 .. 1000) ;`

`END_TYPE`

Necessary range checks must be introduced in the compiler to ensure that values outside this range are not assigned.

3.4.5 Array data types

In many applications, it may be advantageous for a single variable to store different values of a particular type of parameter. For example, in a furnace it may be necessary to measure temperature over an 8 hour period and store them all to a common variable. This will require a data type, which should be a linear array, which can be represented as follows:

```
TYPE FURNACE_TEMPERATURE :  
  ARRAY (1 .. 8) OF UINT;  
END_TYPE
```

Let us further assume that in the same furnace the measurement is done at 4 points on the periphery and at 6 different heights and all these measurements have to be stored in one multi element matrix, the data type can be defined as follows.

```
TYPE FURNACE_TEMPERATURE_MATRIX:  
  ARRAY ( 1..4, 1..6) OF FURNACE_TEMPERATURE;  
END_TYPE
```

Note that this array data will hold the temperature recorded over the entire peripheral surface of the furnace for 8 hours measured hourly. Also note that the first array FURNACE_TEMPERATURE has been declared using the elementary type UINT (considering that the temperature values will always be positive and covered by the range of this data type). The second array data type FURNACE_TEMPERATURE_MATRIX has been declared using the data type FURNACE_TEMPERATURE, which is a single dimensional array defined earlier. The number of array dimensions and the depth of nesting (arrays within arrays) is implementation dependent.

3.4.6 Default initial values of derived data type

All derived data types can be assigned initial default values, which will override the default values of the corresponding elementary data type. These defaults are included in the type definition. Note the example below.

```
TYPE PRESSURE: REAL := 1.0;  
(*Default value defined as 1 bar*)  
END_TYPE  
TYPE PRESSURE_SENSOR:  
  STRUCT  
    CURRENT_PRESSURE      :      PRESSURE := 1.5;  
    SET_PRESSURE           :      PRESSURE := 3.0;  
    SET_TIME               :      TIME := T#10m;  
    ALARM                  :      BOOL := 0;  
  END_STRUCT;  
END_TYPE;  
TYPE  
  SWITCH_MODE: (AUTO, MANUAL) := AUTO;  
(*Enumerated value default set as AUTO*)  
END_TYPE
```

Similarly array data types can also be given default initial values for each array element. When a structured data type is used in other derived data types, each such

derived type can have separate initialization values for individual elements of the structure.

3.5 Variables

Variables are used to store values of various intermediate parameters in different program organisation units. Variables can be used for inputs, outputs or for internal values of parameters used in the processing of the POU. Input, output and input/output variables provide external interfaces between POU's or between a POU and the external environment. (Recall the discussions under the section *Communications* in the previous chapter). Global variables can be declared in programs, resources and configurations. A variable that is declared as External in a POU can access global variables declared outside the POU. In the case of function blocks, variables declared as external, can reference global variables defined in a configuration, resource or program that contains the function block.

Variables can be declared using the construct:

```
VAR
  <declaration>
END_VAR
```

The declaration will consist of the variable identifier followed by the data type of the variable, which can be elementary or derived. Multiple variables can be declared in a single line. For example:

```
X, Y, Z : REAL;
S1, S2  : SINT;
P1, P2  : PRESSURE;
```

The above statements declare variables X, Y and Z which are real, S1 and S2 which are short integers and P1 and P2 which are of derived data type PRESSURE.

Variables can be of the following types:

- Internal
- Input
- Output
- Input/output
- Global and External
- Temporary
- Directly represented

We will discuss the details of these types below.

3.5.1 Internal variable

These variables are local to a POU and are declared as follows.

```
VAR
  X, Y, Z : REAL;
  S1, S2  : SINT;
END_VAR
```

3.5.2 Input variables

Input variables act as parameters to a POU such as programs, functions and function blocks and the values for these variables are supplied from external sources. The declaration of these variables is done as follows:

```
VAR_INPUT
  Switch1, Switch2      : BOOL;
  Max_Value              : REAL
END_VAR
```

3.5.3 Output variables

Output variables act as output parameter to a POU which will be written to external variables. The declaration of this type of variables is done as follows.

```
VAR_OUTPUT
  PUMP1, PUMP2          : BOOL;
  Message1              : STRING (15);
END_VAR
```

3.5.4 Input/output variables

Some variables can act both as inputs and outputs and are declared as follows.

```
TYPE
  MODE
    (READY, ON, OFF);
END_TYPE
VAR_IN_OUT
  STATUS : MODE
END_VAR
```

The above instruction defines an enumerated data type MODE and a variable STATUS is declared as an Input /output variable of the data type MODE. Depending on input value of STATUS as obtained from preceding function blocks and other conditions (say, POSITION), the value of STATUS can be changed and the new value can be written to the variable for use by other external POUs. It may be more efficient to pass large multi element variables as input/output variables since in that case only the address of the variable is passed rather than the entire data string. The above example can be represented graphically in Figure 3.8.

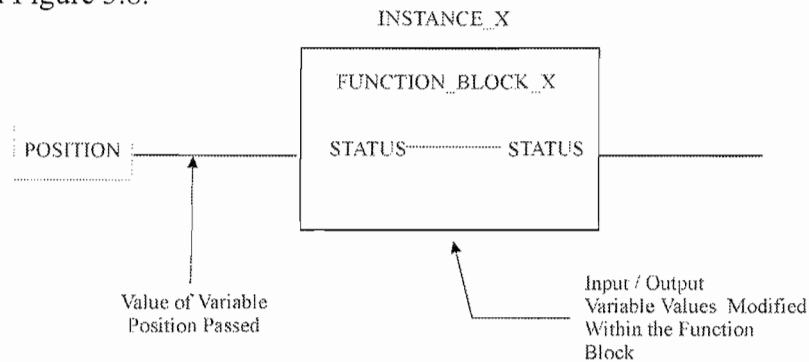


Figure 3.8
Graphical representation of the use of input/output variable

3.5.5 Global variable and external variable

The purpose of these variables is to facilitate communication between POU's within a configuration. Variables declared as global within a configuration are accessible to all POU's within the configuration. A POU, which has to access a global variable, has to declare it as an External variable within the POU. For example, let us say that a set point of a controller has to be communicated a number of POU's in a control system.

The declaration at the configuration level will be:

```
VAR_GLOBAL
    SetPoint1      : REAL;
END_VAR
```

The POU's within the configuration which are required to use the global variable SetPoint1 will have to declare it as an external variable as follows.

```
VAR_EXTERNAL
    SetPoint1      : REAL;
END_VAR
```

Any change in the value of the variable will be read by all POU's that declare it as an external variable.

3.5.6 Temporary variables

Variables that are used for intermediate values computed within a POU and have no need for access outside the POU can be declared as temporary variable. The values of temporary variables are stored in a separate area of memory, which is cleared as soon as the POU execution is completed.

```
VAR_INPUT
    P1, P2, P3 : REAL;
END_VAR
VAR_OUTPUT
    AV1 : REAL;
END_VAR
VAR_TEMP
    Total1 : REAL;
END_VAR
Total1 := P1+P2+P3
AV1 := Total1/3
```

In the above example, the variable Total1 is not needed to be retained outside the POU as it is used as an intermediate value in a calculation and hence is declared as a temporary variable. AV1 which is the average of the input variables P1, P2 and P3 is declared as an output variable being the result of the POU.

3.5.7 Directly represented variable

These are variables, which refer directly to a memory location of the PLC without using an identifier. These variables start with a % character followed by an alphabetic code of one or two letters. The first letter defines whether the memory location is used for input, output or for internal memory. Codes used are:

I	Input
Q	Output
M	Internal memory

The second letter shows the type of memory organisation (bits, bytes or word).

X	Bit
B	Byte (8 bits)
W	Word (16 bits)
D	Double word (32 bits)
L	Long word (64 bits)

The last part contains a numeral representing the memory location. Sometimes this field may have a set of numerals separated by periods to represent the physical location in a system. A few examples are given below.

%IX122	Input memory bit address 122
%IB100	Input memory byte 100
%IW20.1.13	Input memory word at rack 20, module 1, channel 13
%QL100	Output memory Long word at address 100
%Q140	Output memory Bit at 140
(Absence of second character denotes by default a Bit)	

3.5.8 Access variables

As we saw in the previous chapter, PLC systems can communicate over a network using access paths through access variables. Such variables can be input or output variables in a program, global variables or directly represented variables. They can also be specified with READ_ONLY or READ_WRITE attributes to restrict remote devices.

The syntax of usage is as follows.

```
VAR_ACCESS
  Motor_RPM : RPM : REAL READ_ONLY
END VAR
```

The above means that RPM is the global parameter through which the value of Motor_RPM can be accessed for 'read-only' purpose by external devices.

3.5.9 Variable attributes

The different types of variables enumerated above can be used with specific attributes. They are:

```
RETAIN
CONSTANT
AT
```

RETAIN

When a variable has this attribute, the values are retained in PLC memory in the event of a power failure. (In some implementations, all variables are retained by default.

The attribute is defined as shown below:

```
VAR_IN RETAIN
  SpeedSetting : UINT;
END_VAR
```

CONSTANT

These are variables which have been given specific values that cannot change.

For example

```
VAR_IN CONSTANT
  SpeedRamp : REAL:= 18.5; (*RPM per Sec *)
END_VAR
```

AT

Global variables and variables declared in a Program can be given the attribute AT to assign specific memory locations for these variables. Unless a variable has this attribute, it will be automatically assigned to a memory location by the compiler or programming station. An example is:

```
VAR
    Digital_Values AT %QX100: ARRAY (1..10) OF BOOL;
END_VAR
```

3.6 Variable initialization

We have seen earlier how a data type can be assigned a default initial value. A variable of a particular data type will be initialized with the default value for the applicable data type. This can however be modified and a new initial value assigned to it while declaring the variable.

Note:

This is not applicable to external variables, as they will automatically assume the value of the corresponding global variable).

An example is:

```
VAR
    MAX_Pressure: REAL:=2.5 (*Initialised at 2.5 ATG*)
END_VAR
```

We have seen how a derived data type PRESSURE_SENSOR can be declared with default initial values. (See italicized text below).

```
TYPE PRESSURE_SENSOR:
    STRUCT
        CURRENT_PRESSURE      :      PRESSURE := 1.5;
        SET_PRESSURE          :      PRESSURE := 3.0;
        SET_TIME               :      TIME := T#10m;
        ALARM                  :      BOOL := 0;
    END_STRUCT;
END_TYPE;
```

A variable Pressure1 of data type PRESSURE_SENSOR can be initialised thus.

```
VAR_INPUT
    Pressure1:PRESSURE_SENSOR
    (CURRENT_PRESSURE:=2.3,SET_PRESSURE:=2.8,
     SET_TIME := T#8m, ALARM :=0);
END_VAR
```

3.7 Functions

Functions are a type of Program Organisation Unit. They are small reusable programs that form the fundamental building blocks of complex industrial control programs. A PLC implementation contains several standard functions for performing various mathematical, string and Boolean operations. In addition, user defined functions can also be created for repetitive tasks. Functions can be nested within one another. A function contains one or more inputs and returns a single output as a result of the evaluation defined within the function. It is possible to represent a function in any programming language defined in the standard. For example:

$\text{SINE_OF_ANGLE} := \text{SIN}(\text{ANGLE}) (*\text{ANGLE in Radians}*)$

Represents the trigonometric sine function which returns the value of input value ANGLE of an angle in radians using Structured Text language. The same can also be represented using function block diagram shown on Figure 3.9.

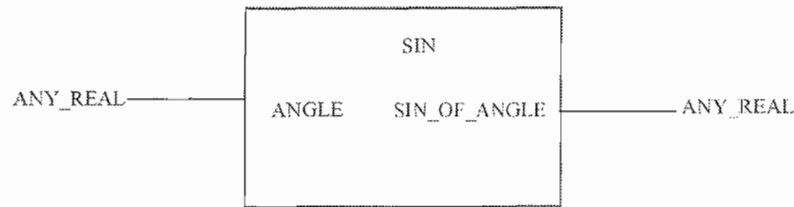


Figure 3.9

FBD for SIN function

Declaring a new function is done using the construct FUNCTION .. END_FUNCTION as follows.

```
FUNCTION SUM_OF_3:REAL
  VAR_INPUT
    A,B,C:REAL
  END_VAR
  SUM_OF_3 :=A+B+C
END_FUNCTION
```

This function adds three real number inputs and stores the result to SUM_OF_3.

There are functions which are capable of handling overloaded data types such as SQRT. The support to overloaded data types however depends on the implementation. This avoids the need to define multiple functions one for each data type for performing essentially the same operation. Many functions can handle values directly without the need for parameters to pass values.

Functions are available to convert data types, for example:

From Integer to REAL, From String to Integer type or Real type and so on.

It should be however remembered that such conversion may result in illegal values which are not appropriate to the data type to which the input is being converted. It is therefore necessary to ensure that the run time error detection and reporting is done. Various types of standard functions defined in the standard are reproduced in the tables below.

NAME	Description	DATA Type
ABS	Absolute value	ANY NUM
SQRT	Square root	ANY REAL
LN	Natural Logarithm	ANY REAL
LOG	Common Logarithm	ANY REAL
EXP	Exponential	ANY REAL
SIN	Sine of an angle in radian measure	ANY REAL
COS	Cosine of an angle in radian measure	ANY REAL
TAN	Tangent of an angle in radian measure	ANY REAL
ASIN	Arc sine of a number result in radians	ANY REAL
ACOS	Arc Cosine of a number result in radians	ANY REAL
ATAN	Arc Tangent of a number result in radians	ANY REAL

Figure 3.10

Numeric functions

Note that these functions support overloaded data types.

NAME	Description	ST	DATA Type
ADD	Addition I1+I2+..	+	ANY_NUM
MUL	Multiplication I1*I2*	*	ANY_NUM

Figure 3.11
Extensible arithmetic functions

In the above examples of extensible functions, the number of inputs can vary depending upon the requirements of the program.

NAME	Description	ST	DATA Type
SUB	Subtraction I1-I2	-	ANY_NUM
DIV	Division I1/I2	/	ANY_NUM
MOD	Modulus I1 MOD I2	MOD	ANY_INT
EXPT	Exponential I1 I2	**	ANY_REAL
MOVE	Assigns the value I1 to the result	:=	ANY

Figure 3.12
Non-extensible arithmetic functions

Note that these functions accept only two inputs.

NAME	Description	DATA Type
SHL	Shift bit string n positions to left, fill zero on right.	ANY_BIT
SHR	Shift bit string n positions to right, fill zero on left.	ANY_BIT
ROR	Shift bit strings n positions to right, rotate	ANY_BIT
ROL	Shift bit strings n positions to left, rotate bits	ANY_BIT

Figure 3.13
Bit string functions

NAME	Description	Symbol in Ladder diagram and FBD	DATA Type
AND	Result of I1 & I2 & I3..	&	ANY_BIT
OR	Result of I1 OR I2 OR..	>=1	ANY_BIT
XOR	I1 XOR I2 XOR ..	=2k+1	ANY_BIT
NOT	Result of NOT I1		BOOL

Figure 3.14
Boolean bit string functions

NAME	Description	DATA Type
SEL	Selection: If G is TRUE then result:=I1 ELSE it is I2	ANY
MAX	Maximum: Result :=Maximum of all inputs	ANY
MIN	Minimum: Result :=Minimum of all inputs	ANY
LIMIT	Gives the result as input I subject to limits set by the maximum and minimum values MN and MX	ANY
MUX	Gives the result as the value of input selected by the a given input K	ANY

Figure 3.15
Selection functions

NAME	Description	Symbol
GT	TRUE if input I1 is greater than I2	>
GE	TRUE if input I1 is greater than or equal to I2	>=
EQ	TRUE if input I1 is equal to I2	=
LE	TRUE if input I1 is less than or equal to I2	<=
LT	TRUE if input I1 is less than I2	<
NE	TRUE if input I1 is NOT equal to I2	<>

Figure 3.16
Comparison functions

NAME	Description	Symbol
SEL	Selection of one of two enumerated values depending on a boolean condition being TRUE	
MUX	To select one of a set of enumerated values based on the value of an integer selector	
EQ	Test the equality of two inputs of the same data type	=
NE	Test inequality of two inputs	<>

Figure 3.17
Functions for enumerated data type

Only a few of the standard functions as shown above can use enumerated data types since a variable of such data type does not have an intrinsic value which can be used in a numerical or string operation.

3.7.1 Execution control

The execution of a function used in a Ladder diagram or Function Block diagram can be controlled using a special input called EN. The function will execute only when EN becomes TRUE. When EN is false, the function remains inactive and does not evaluate its input and does not assign any value to its output. When a function executes successfully based on the value of EN, it sets the value of a special output ENO as TRUE. If the function encounters an error and is not able to complete its evaluation, the output ENO remains FALSE. This output can be connected to the input EN of another function. Sequential control of a set of functions can thus be controlled using the combination of EN and ENO and a chain of such functions can be arranged to produce an output only when all the functions execute in sequence without any error.

Figure 3.18 illustrates the above principle using a ladder-cum-function block diagram example.

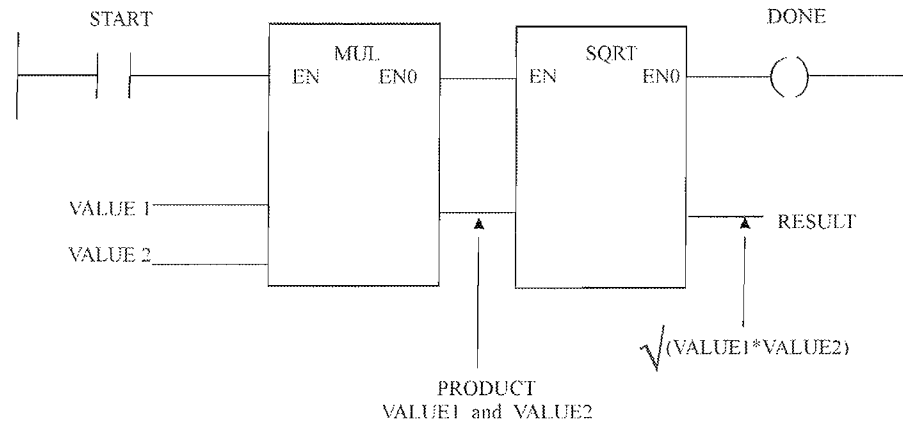


Figure 3.18
Execution control of functions

3.7.2 Function blocks

Function Block (FB) is the next type of POU. It has a pre-defined set of input variables and output variables associated with it. Each function block has an underlying algorithm, which enables evaluation of all its inputs and produces a set of output values. Unlike functions, function blocks can also have temporary variables, which store intermediate values during its evaluation. These are not accessible from outside the Function Block. The execution of a function block can also depend on its own output in a previous instance of execution. Thus the function block has data persistence since the previous output values can be remembered and reused. Also, the same set of inputs need not necessarily produce the same set of outputs because of data persistency.

Function Block type defines the data structure and the algorithm. Function block instance is a set of values held in this data structure. The algorithm modifies the data held in the structure when the FB instance executes. As we saw earlier, the input and output variables of a FB instance can be accessed externally but not the internal variables. An FB instance has to be invoked when explicitly requested as a part of a graphical network of connected blocks in a POU (known as the FBD) or by a call using Instruction List or Structured Text language.

FB instances are declared using variable definitions within a POU. Normally variable declarations are local by default, which means that FB instances can only be seen within the POU in which they exist. FB instances when declared as global can be accessed by any POU within the resource or configuration in which the global instance is declared. A function block instance can be passed as an input to another POU. The current input and output values of a FB instance can be accessed in textual language using the construct:

<Function Block instance> . <variable name>

FB types are declared using the definition:

FUNCTION_BLOCK

<Variable declarations>

<Algorithm>

END_FUNCTION_BLOCK

The algorithm or the body of the FB can be written in any language type (textual and graphical) supported by IEC-1131-3 standard.

Function block instances are declared just as variables. A function block instance is only a data structure which is processed as per algorithm defined in the function block type. For example:

```

VAR
  COUNTER1 : COUNTER
END_VAR

```

COUNTER is a function block already defined. COUNTER1 which is an instance of this function block is declared as a variable. Other possible ways are

```

VAR_GLOBAL
  COUNTER1 : COUNTER
END_VAR

```

The above is a construct in which an instance of a global function block COUNTER is declared.

```

VAR_INPUT
  COUNTER1 : COUNTER
END_VAR

```

Here an instance of function block COUNTER is used as input to another POU.

The example below shows the method of creating a custom function block COUNTER and how it is instantiated.

(*Declaring an enumerated data type for use in the FB COUNTER*)

```

TYPE
  POSITION : (RESET, ADD, HOLD)
END_TYPE

```

```

FUNCTION BLOCK COUNTER
(Define and initialise input, output*)

```

```

VAR_INPUT
  POSN : POSITION := RESET;
END_VAR
VAR_OUTPUT
  OUT : INT := 0;
END_VAR

```

```

(*Define procedure*)
IF POSN = RESET THEN
  OUT := 0
ELSIF POSN = ADD THEN
  OUT := OUT + 1
END_IF

```

```

END_FUNCTION BLOCK

```

Having defined the function block COUNTER, we can now proceed to declare an instance COUNTER1 of this function block in a Program COUNTADD as follows.

```

PROGRAM COUNTADD
VAR_INPUT
  INPUT_POSN : POSITION;
END_VAR
VAR_OUTPUT
  Maxadd : INT;
END_VAR
VAR
  COUNTER1 : COUNTER
END_VAR

```

(*In the above statements, input and output variables have *)

```

(*been declared and an instance of COUNTER has also been *)
(*declared. Now we can proceed to invoke instance COUNTER1 *)
COUNTER1 (POSN := INPUT_POSN);
Maxadd :=COUNTER1 . OUT;
END_PROGRAM

```

3.8 Programs

Programs are the largest type of POU's and are declared within a resource. In behaviour, Programs are similar to function blocks and are reusable using program instances. The main difference between Programs and Function blocks is that in programs, all types of variable such as directly represented variables, Global Variables and Access variables can be declared. Programs can contain instances of function blocks, which can be executed by different tasks. Program instances cannot be used within themselves (nested). As stated earlier, programs can only be declared within a resource. Program types are defined as follows.

```

PROGRAM <Name of Program type>
  <Variable declarations>
  <Algorithm, also called as program body>
END_PROGRAM

```

Programs are usually complex software constructs and are meant to control a complete system or equipment such as a boiler or a steam turbine. Program types are generic in nature and are applicable to a particular equipment type. Instances of a program type can be created and used to control a number of similar equipment, say a number of identical boiler units.

Instances of a program are declared thus:

```
PROGRAM BOILER_1 : BOILER
```

Where BOILER is a Program type whose instance is BOILER_1. Let us say that the program type covers input variables KCAL, SET_PR and SET_TEMP and output variable FUEL_RATE, these can be passed in the instance BOILER_1 as global variables for inputs and to directly represented output locations.

In this case, the statement will be:

```
PROGRAM BOILER_1 : BOILER (KCAL := A1, SET_PR := A2,
SET_TEMP = A3, FUEL_RATE => %QW33);
```

3.9 Resource

A Resource generally corresponds to a device that can execute IEC Programs and is defined within a configuration using an identifier and the processor on which the resource will be loaded. A resource contains Global variable declarations, Access variables that permit remote access to named variables, External variable declarations, Program declarations and task definitions.

3.10 Tasks

One of the main requirements in a large process control system is the ability to execute different parts of the control program at different rates depending on the process requirements. For example, a system may contain components with large inertia in its parameters, say a boiler furnace whose temperature can only vary slowly in view of the large thermal time constant and as a result, the function block which controls the furnace temperature may execute once in say 10 seconds. On the other hand, a turbine will have a

very fast speed response and the over speed monitoring function block will have to execute at a much faster rate. An interlocking logic of a fast moving process line may require even faster execution.

Tasks achieve this type of control by triggering programs and function blocks at specified time periods. The standard provides for allocation of specific programs and function blocks to specific tasks, which execute at predefined intervals and priority rates (0 for the highest priority and 1,2, 3 etc. in decreasing order). When multiple tasks are declared within a resource, they need to be scheduled. The scheduler decides the exact moment that a task has to execute. Since two tasks cannot run concurrently, some form of arbitration is needed to resolve the sequence in which two waiting tasks have to be taken up. There are two ways in which a PLC schedules tasks; by non-preemptive scheduling and by preemptive scheduling. We will briefly discuss them below.

3.10.1 Non-preemptive scheduling

The rules in this type of processing are:

- Tasks once started are allowed to be completed before the next in line is taken up. (Completion means that all programs and function blocks assigned to the task are executed in sequence once)
- If two tasks are waiting to be executed, the one with the higher priority is taken up first
- If two waiting tasks have equal priority, the one waiting for longer time is taken up first
- A task once completed is taken up for execution only on completion of the assigned task interval

The advantage is that designing a system based on non-preemptive method is more straightforward and easier. The disadvantage is that, it is difficult to predict the exact time interval with which a task will be executed and also, a desired execution sequence of tasks cannot be ensured. This results in unpredictable system behavior and is therefore non-deterministic in nature.

3.10.2 Preemptive scheduling

When a system with deterministic behavior is desirable, preemptive scheduling is to be used. This will ensure consistent timing between events. In this method, when a higher priority task is to be executed as per the specified interval, it is immediately scheduled and the currently active lower priority task is suspended and will be continued after the higher priority task terminates.

Figure 3.19 illustrates the distinction between the two methods of scheduling.

Tasks have to be declared as under:

```
TASK INTERLOCK    (INTERVAL := t#50ms, PRIORITY := 1);
TASK LOGRECORD (SINGLE := LOG_EVENT, PRIORITY := 2);
```

As can be seen the declaration starts with TASK followed by an identifier. The characteristics of the task are specified within the parentheses and consist of the following.

INTERVAL with time value indicates that the task has to execute with the periodicity defined by INTERVAL value.

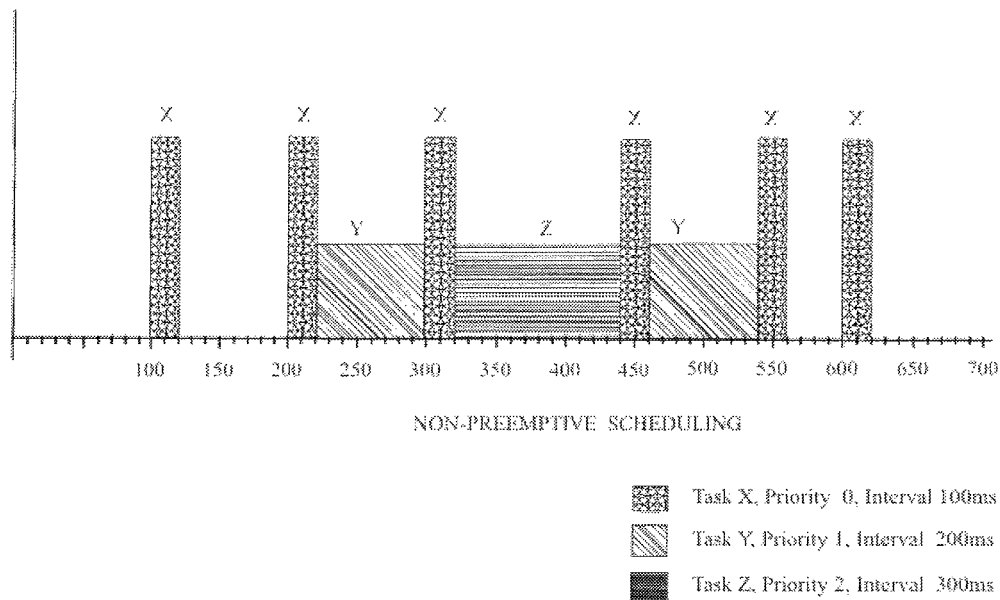


Figure 3.19A
Task scheduling- Non-preemptive approach

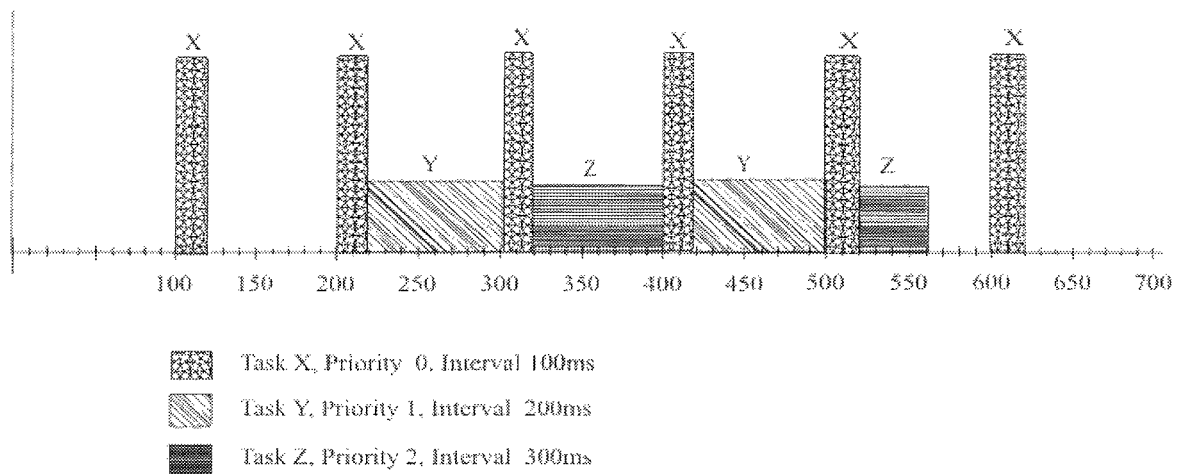


Figure 3.19B
Task scheduling- Pre-emptive approach

Alternatively, a task can be specified as SINGLE and assigned to a boolean variable. This indicates that the task will be executed ONCE only on the rising edge of the variable (when its value changes from 0 to 1).

Next is the PRIORITY which will specify the order of priority assigned to the task. As we saw in the earlier paragraphs, this value determines the sequence in which tasks are scheduled when more than one task are to be executed.

3.10.3 Task assignment

Programs and Function blocks are assigned to a task using the keyword WITH. It is not necessary to assign all programs and FB instances to a task. An FB instance not assigned to a task is executed in the same task as the program which contains the FB instance. Programs that are not assigned to a task execute continually in the background at the lowest priority level.

In order to ensure predictable outputs during execution of Programs and FB instances, IEC-1131-3 defines a set of rules, which are as follows:

- If more than one input value of FB named A are obtained from outputs of FB named B, all these outputs will be produced by the same execution of B
- When a number of FBs are assigned to the same task X and all of them obtain their inputs from the outputs of another FB named A assigned to a different task Y, all these out values of FB A will be produced by the same execution of task Y

The way these conditions are to be achieved is not mentioned in the standard. However, it can be inferred from the rules that:

- A temporary buffer will have to be used to store the value of outputs of FBs as they are generated during a task
- When the task is completed the buffer values are copied to the output locations in a single uninterrupted copy operation

3.10.4 Configuration

A configuration defines the software of a complete PLC. A configuration includes at least one and usually several, resources. A resource in turn defines several tasks and programs to be executed by these tasks. Global variables common to all resources are specified in the configuration. Access variables to be used for communication with other configurations are also specified at the configuration level. Global variables are also declared at resource level and are accessible to all the programs within the resource.

The configuration is introduced using the following software construct,

```
CONFIGURATION <Identifier>
GLOBAL VARIABLE Declaration
ACCESS VARIABLE Declaration
RESOURCE Declaration
< Global variables within resource>
<Task definitions>
<Programs>
END_CONFIGURATION
```

Variables used within a resource are identified hierarchically thus:

```
<Resource Identifier>.<Program Identifier>.<Variable name>
```

3.11 Summary

In this chapter, we had an over view of different data types and variables. We also learnt about the different types of POU's such as functions, function blocks and Programs. We reviewed the basic functions defined in the standard and how these are used to build a hierarchically arranged software structure. We also learned about resources and tasks and how tasks achieve a sequential execution of programs and function blocks using preemptive or non-preemptive scheduling approach. We saw how an entire PLC's software is organised within a configuration.

Most of the examples given in this chapter used the structured text method of programming. However, it is possible to represent them using the other languages such as Instruction List, Function Block Diagram, Ladder diagram and Sequential function charts, the last three being graphical languages. We will next go on to the details of each of these languages in the coming chapters.

Structured text

This chapter contains information on the programming language called Structured Text, which is one of the two textual languages defined in the standard IEC-1131-3.

Objectives

On completing the study of this chapter, you will learn about:

- The basics of Structured Text (ST) language
- Assigning values to variables using ST
- Creating expressions
- Use of operators
- Function/Function block calls using ST language
- Conditional and iteration statements

4.1 Introduction

Structured Text (abbreviated as ST) is a high level textual language similar to PASCAL. The similarity is superficial only as ST is primarily aimed at solving control problems in real time. Most of the examples about data type declarations, variable declarations and other constructs we saw in the last chapter were written using ST and the readers will therefore be somewhat familiar with the syntax and style of using ST. We had also discussed about how the programs are written with indenting to improve readability and comments to make the code understandable. The same conventions will be used in this chapter too.

In the strict sense, the syntax used for the above declarations does not form part of the ST language as defined in IEC-1131-3. The standard defines ST as a language that consists of statements that can be used to assign values to variables using various types of arithmetic and Boolean operators. We will go through the various constructs of ST language in this chapter.

4.2 Statements used for assignments

The general syntax of assignment statements takes the form:

`A := B`

A is a variable to which a value is assigned using B which can be an expression or a literal constant or another variable. The value obtained by evaluation of B is assigned to the variable A by this statement and replaces the previous value of A. Needless to say the data type of A should be the same as that of B.

Look at the following examples.

`Speed := 12.5;`

This statement assigns a literal constant value of 12.5 to a variable Speed. It is assumed that the variable Speed has been declared as a REAL data type variable in some preceding part of the POU.

`Countadd := Countadd + 1;`

The above statement assigns a value to an Integer variable Countadd, which is higher than its previous value by 1.

`Pressure [4] := Force/Area;`

This statement replaces the 4th term of the array variable Pressure by the value evaluated by the expression Force/Area where Force and Area are two other variables having some predetermined values.

`A := SIN (Angle)`

This is an example of the use of an arithmetic function (a sine function) for assigning a value to variable named A.

The statement

`Pressure_1 := Pressure`

Causes the value of an array variable Pressure to be assigned to another array variable Pressure_1 of the same data type. When assignments involve multi element variables as cited in the example above, all elements should be assigned before one or more of the elements are accessed by other operations.

4.3 Expressions

Expressions are part of statements where the values of one or several variables are manipulated using operators of arithmetic or Boolean types to produce a single value. This value is then assigned to another variable, which should be of the same data type as the result of the evaluation. An expression is placed on the right hand side of an assignment statement. The PLC implementation will return an error if the data type of the result of the evaluation of an expression does not match that of the variable to which the value is assigned.

The table in Figure 4.1 below lists the operators as defined in the standard. This list is in the inverse order of precedence (from highest to lowest) using which an expression will be evaluated.

OPERATOR	DETAIL
(...)	Expression within parentheses
Function (....)	Parameter list of a function, function evaluation
**	Raising to a power
-	Negative of
NOT	Boolean complement
*	Multiplication
/	Division
MOD	Modulus

OPERATOR	DETAIL
+	Addition
-	Subtraction
<, >, <=, >=	Comparison
=	Equality
<>	Inequality
AND, &	Logical (boolean) AND
XOR	Boolean Exclusive OR
OR	Boolean OR

Figure 4.1
Operators defined in IEC-1131-1

4.4 Evaluating an expression

An expression is evaluated using the order of precedence as shown above. Parts of expressions whose operators have higher precedence are evaluated first. In case operators in two parts have equal precedence, they are evaluated proceeding from left to right.

Consider for example, the expression:

$X := A + B/2.0 + \text{SQRT}(A+B) - 25.0/C$

With the values of $A=10.0$, $B=6.0$ and $C=5.0$

The evaluation is done in this order:

$B/2.0$	=	3.0	
$25.0/C$	=	5.0	
$A + 3.0$	=	13.0	$(*A + B/2.0*)$
$A + B$	=	16.0	
$\text{SQRT}(16.0)$	=	4.0	$(*\text{SQRT}(A+B)*)$
$13.0 + 4.0$	=	17.0	$(*A + B/2.0 + \text{SQRT } 16.0*)$
$17.0 - 5.0$	=	12.0	$(*17.0 - 25.0/C*)$
Thus X	=	12.0	

Adding a parenthesis to the above expression will change the result. Consider this:

$X := (A + B)/2.0 + (\text{SQRT}(A+B) - 25.0)/C$

Now the expression will be evaluated in this order:

$A+B$	=	16.0	
$16.0/2.0$	=	8.0	$*(A+B)/2.0*$
$A+B$	=	16.0	
$\text{SQRT}(A+B)$	=	4.0	
$4.0 - 25.0$	=	-21.0	$*(\text{SQRT}(A+B) - 25.0*)$
$-21.0/C$	=	-4.2	
$8.0 + (-4.2)$	=	3.8	

X is assigned the value 3.8

This example illustrates how the evaluation is affected by the change in precedence obtained by introducing a set of parentheses.

In the case of Boolean expressions, similar rules of precedence apply. However, a Boolean expression is evaluated only up to a point that is needed to determine the value.

Consider

$X := A \text{ AND } B \text{ OR } C$

If the value of A is FALSE, the expression is straightaway evaluated as FALSE as the evaluation of the lower precedence part $B \text{ OR } C$ will not affect the final result anyway.

4.5 Statements

Statements are software instructions for various purposes such as calling function blocks, iterative processing, conditional evaluation etc. These will be discussed in the following paragraphs.

4.5.1 Function block calls

A function block instance can be invoked in a program by calling the name of the FB instance and assigning values to each input parameter. On being thus invoked, the function block instance will execute its code using the input parameter values specified as a part of the calling statement and update the variables associated with the FB instance outputs. If parameter values are not supplied for one or more of the inputs, the values of the previous invocation will be used. If the FB instance is being invoked for the first time, the default values given while defining the function block type will be used. Consider the following example shown in Figure 4.2.

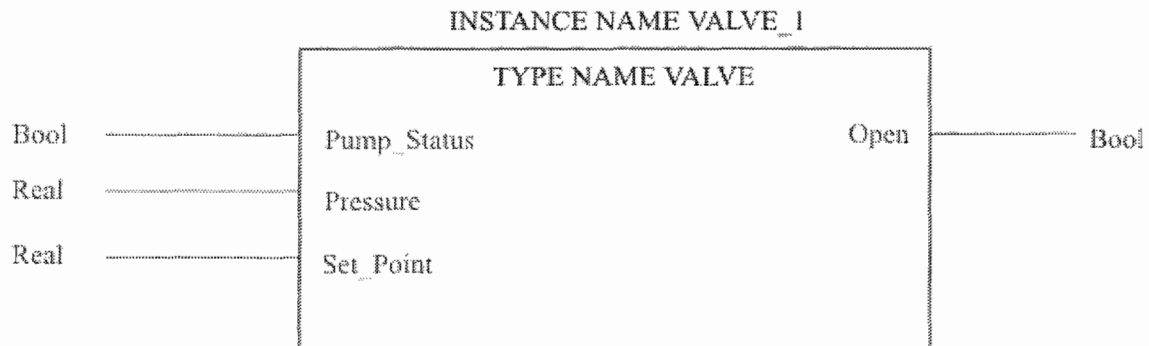


Figure 4.2

Example of a function block

Valve_1 is an instance of the function block type Valve shown in the above figure. The statement declares the instance:

```

VAR
  Valve_1 : Valve;
END_VAR
  
```

The initial invocation of the instance is done by the statement:

```
Valve_1(Pump_status:=Pump1,Pressure:=P1,Set_Point :=10.0);
```

A subsequent invocation can be:

```
Valve_1 (Pump_status := Pump2,Pressure :=P2);
```

In this case the value of Set_Point declared earlier is retained from the previous invocation.

The function block instance output can be assigned to other variables thus:

```
Valve1_Command := Valve_1.Open;
```

The value of Valve1_Command is the value of the output Open when the function block instance Valve_1 was last executed. Alternatively, the inputs of function block instance can be obtained directly from input memory locations and the output stored directly to variables as shown:

```
Valve_1 (Pump_status := IX80, Pressure :=IW100, Set_Point :=10.0,
Open:=Valve1_Command);
```

4.6 Conditional statements

Structured Text Language provides statements that decide a particular course of action depending on a set of conditions. We will see different types of such conditional statements in the following paragraphs,

4.6.1 IF...THEN...ELSE

The general format of this statement is as follows:

```
IF <Condition> THEN
  <Statement>
ELSE
  <Statement>
ENDIF;
```

The statement in the FB example Valve_1 in Figure 4-2 may typically be as follows.

```
IF Pump_Status AND (Pressure => Set_Point) THEN
  Open :=TRUE;
ELSE
  Open :=FALSE;
ENDIF;
```

If the conditional statement following IF is fulfilled then the value of variable Open will be set to TRUE. Otherwise it is set to FALSE.

These statements can be nested within each other to form more complex conditions. These will have the general form

```
IF <Condition 1> THEN
  <Statement1>
  IF <Condition 2> THEN
    <Statement 2>
  ELSE
    <Statement 3>
  ENDIF
ELSE
  <Statement 4>
ENDIF;
```

In the above construct, Statement 1 is executed if Condition 1 is fulfilled. After executing Statement 1, Condition 2 is checked and if true, Statement 2 is executed, otherwise, Statement 3 is executed. If Condition 1 is not fulfilled, then Statement 4 is only executed.

There is also the ELSIF construct, which operates thus.

```
IF <Condition 1> THEN
  <Statement1>
ELSIF <Condition 2> THEN
  <Statement 2>
ELSIF <Condition 3> THEN
  <Statement 3>
ELSE
  <Statement 4>
ENDIF;
```

Statement 1 is executed if condition 1 is fulfilled.

Statement 2 is executed if condition 2 is fulfilled.

Statement 3 is executed if condition 3 is fulfilled.

If none of the above conditions are fulfilled then Statement 4 is executed. It is assumed that conditions 1, 2 and 3 are mutually exclusive.

4.6.2 CASE statement

CASE is another type of conditional statement where certain actions are carried out depending on the value of an integer variable. For example:

```
CASE RPM_Setting OF
  1:      RPM   :=    100;
  2:      RPM   :=    200;
  3:      RPM   :=    300;
  Speed_Switch :=    TRUE;
ELSE
  RPM      :=    0;
  Speed_Switch :=    FALSE;
END_CASE;
```

CASE statement can also be used with enumerated variables. An example is shown below.

```
TYPE
  Setting : (SLOW, MEDIUM, FAST);
END_TYPE
VAR
  RPM_Setting : Setting;
END_VAR
CASE RPM_Setting OF
  SLOW:      RPM   :=    100;
  MEDIUM:   RPM   :=    200;
  FAST:      RPM   :=    300;
  Speed_Switch :=    TRUE;
ELSE
  RPM      :=    0;
  Speed_Switch :=    FALSE;
END_CASE;
```

4.7 Iteration statements

Iteration statements cause the execution of a set of statements repeatedly based on the value of a particular integer variable used as a counter to decide as to how many times the statements will be repeated. It is also possible to do the iteration based on a Boolean logic being fulfilled. Care must be taken to ensure that the execution does not cause an endless loop. These statements considerably increase the execution time of programs/function blocks. We will discuss below the various constructs used for iteration.

4.7.1 FOR ... DO

This construct causes an iteration to be performed based on the value of a integer variable of type INT, SINT or DINT. The syntax of FOR DO iteration construct is as follows.

FOR < Counter Variable initialization>

```

    TO <Final value of the variable>
    BY <Increment to be applied> DO
    <Statements to be executed>
END_FOR;

```

In this statement BY can be omitted, in which case the increment will be assumed to be 1. The iteration will be terminated once the value of the counter variable reaches the final value. (Since the check will be made prior to executing the statements, they will not be executed in the step when the counter value has gone beyond its final value).

Consider the following example:

```

For K    :=1    TO 11 BY 2 DO
    ALARM[K] := TRUE;
END_FOR;

```

This statement will store logical 1 to positions 1,3,5,7,9 and 11 of the array variable ALARM.

Note

It must be ensured that the value of the integer variable K is not modified in the statements inside the loop as it may give rise to unpredictable behavior.

4.7.2 WHILE ... DO

This construct will cause a set of statements to be repeated while a particular boolean expression is TRUE. Since the check is done before the statements are executed, the execution goes out of the loop once the expression becomes FALSE. The syntax of the construct is as follows:

```

WHILE <Boolean Expression> DO
    <Statements>
END_WHILE;

```

An example of this can be:

```

J:=1;
WHILE J<=100 & WORDS(J) <> 'KEY' DO
    J:=J+2;
END_WHILE;

```

4.7.3 REPEAT ... UNTIL

This is similar to the WHILE ... DO construct with the difference being that the check is made after the statements are executed. The syntax is:

```

REPEAT
    <Statements>
UNTIL <Boolean Expression>
END_REPEAT;

```

4.7.4 EXIT

It may sometimes become necessary to terminate the execution of an iteration routine on the occurrence of a certain event without completing the full sequence of the iteration. In such cases the EXIT statement is used to break out of the loop. It can only be used within an iteration, for obvious reasons.

See this example:

```

WHILE Volume_Tank_1 <=Set_Tank_Volume DO
    Valve_1 := OPEN;

```

```

IF ALARM [1] THEN
    VALVE_1 := CLOSED
    EXIT;
ENDIF;
END_WHILE;

```

When the tank fluid volume is less than the set value, valve no. 1 is kept open. At the same time, it is ensured that a certain alarm condition represented by the array position 1 of the variable ALARM is not TRUE. If it becomes TRUE, then valve 1 is closed and the iteration terminates.

4.7.5 RETURN

RETURN statement is similar to EXIT but is used inside functions and function blocks only. It causes the execution of the function block or function to cease based on the condition given in an expression being fulfilled and will continue the next part of the code. Consider this example.

```

FUNCTION_BLOCK Tank_Fill
TYPE
    Valve_Status : (OPEN, CLOSED);
END_TYPE
VAR_INPUT
    Volume_tank, Set_Tank_Volume : REAL;
    Alarm : BOOL;
END_VAR
VAR_OUTPUT
    Valve : Valve_Status := CLOSED;
    Msg : STRING;
END_VAR
WHILE Volume_Tank <= Set_Tank_Volume DO
    Valve := OPEN;
    IF ALARM THEN
        VALVE := CLOSED;
        Msg := 'Motor Overload Operated';
        EXIT;
    ENDIF;
END_WHILE;
END_FUNCTION_BLOCK;

```

In this example, function block Tank_Fill executes to fill a tank till the liquid volume reaches a preset value. However, if an alarm condition as indicated by the status of the Boolean variable ALARM becomes TRUE, it closes the filling valve and stops execution of the function block. It gives a message to alert the operator of an alarm condition in the system.

4.8 Implementation dependence

While IEC-1131-3 does not provide any limitations to ST language, it allows implementations to impose certain limits on some of the language features such as length of expression, length of statements, number of CASE selections etc. This is to ensure that the compiler is able to handle the code correctly and also to be certain that the target PLC capabilities are not exceeded.

4.9 Summary

- Structured Text (abbreviated as ST) is a high level textual language similar to PASCAL
- ST uses a strong data typing and so avoids errors due to type mismatch
- It is possible to represent complex process logic conditions using various features of the language
- Variables are declared and used to denote process conditions and to perform the required process operations
- Operators are defined in the language for arithmetical and boolean operations as well as for comparison
- Programs, Function blocks and functions can be written using Structured Text language
- Programs can use Function blocks and Functions by using the calls provided in the language
- Syntax of statements includes features such as conditional and iterative execution

Function block diagram

This chapter contains information on the graphical programming language Function Block Diagram (FBD) described in IEC-1131 part 3. Representation of control systems using function block diagram is explained using examples and rules for evaluation of FBD networks are also illustrated.

Objectives

On completing the study of this chapter, you will learn:

- The methodology of **Function Block Diagram** (FBD) method of programming industrial control systems
- Representation of signal flow between FBD elements
- Feedback paths
- Execution control method adopted
- Evaluation of FBD networks

5.1 Introduction

Function Block Diagram (FBD) is the second of the five programming languages covered in IEC-1131-3 viz., Structured Text, Function Block Diagram, Ladder Diagram, Instruction List and Sequential Function Chart. This is the first Graphical Language covered by the standard.

5.2 Basics

FBD is a graphical representation of an Industrial programmed control system and adopts a set of symbols and conventions defined in IEC-1131-3. It represents a control system in terms of signal flow between processing elements similar to the methodology adopted for signal flow in electronic circuits. By using interconnected graphical blocks, it expresses the behaviour of functions, function blocks and a composite program. An FBD can also be used within a Sequential Function Chart.

The standard accepts both semi-graphic and full-graphic modes of representations. Semi graphical mode uses normal characters such as – and | to be used for depicting graphical objects. Full graphic representation uses graphical symbols to represent program objects and is preferred by most vendors. (Figure 5.1 show some of the basic representations used both in semi graphic and full graphic modes).

The standard does not define the details of a full graphic system to a fine degree but leaves it to the implementation of the vendor. For example, the standard does not specify the format of line intersections, junctions and crossovers in full graphic implementation. These can be decided based on the properties of the graphic system and additional features such as angled/rounded corner of the FBD blocks, color/shaded boxes etc. can be used if the system permits them.

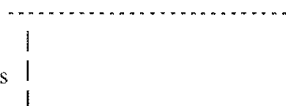
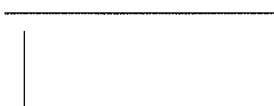
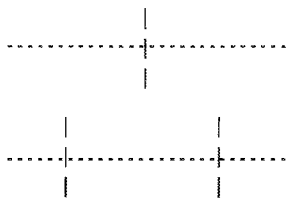
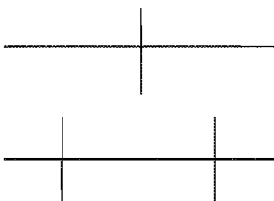

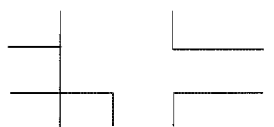

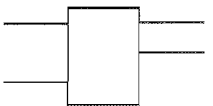
GRAPHICAL FEATURE	SEMI-GRAPHIC FORM	FULL GRAPHIC
Horizontal and Vertical Signal Flow Lines		
Interconnection of Horizontal and Vertical Signal Flows		
Signal Flow Corners		
Blocks With Connectors		

Figure 5.1
Commonly used graphic representation

5.3 Methodology

As we saw earlier, the Function Block Diagram language is used to express:

- The behaviour of a control program made up of functions and function blocks using a set of interconnected blocks
- The behaviour of steps, actions and transitions in sequential function charts
- The flow of signals between processing elements

An FBD network is similar in character to an electrical circuit diagram, which uses interconnecting lines between components to represent flow of signals. Typically, an FBD network is used to depict control loops and logic and is preferred by programmers who are familiar with electrical circuit diagrams.

FBD uses certain graphical conventions. For example:

- A function block is a rectangular block with inputs entering from the left and outputs exiting from the right
- The type name of the function block is always shown within the block
- The name of the function block instance is always shown above the block
- The formal names of function block inputs and outputs are shown within the block at the appropriate input and output points

(**Note:** In some implementations, input and output parameters are denoted using their 'pin' numbers and use other means such as a pull down menu, to show the full names)

Figure 5.2 shows a typical FBD network for a tank control program. TankControl is the name of the function block type (shown within the block) and the name of the instance is TankControl1. It receives inputs Weight, FullWeight and EmptyWeight besides a Command input from a thumb-wheel switch, represented by function block ThumbSwitch and instance Switch1. The outputs of the program are the values FILL, EMPTY and STIR.

The input Weight is itself derived from other function blocks shown in the diagram as instances Weigher1 and Weigher2 of the FB type Weigher. These compute a value using functions ADD and DIV, which represent the input Weight of the TankControl1 FB. Note here that this signal has been given a name AVE_WEIGHT and shown terminating in a >. This indicates a cross diagram connector used mainly to improve the readability of the program but has no other significance. The program behavior is in no way affected by the use of this representation. Also note the way the input signal Loaded Weight has been drawn. This is an example of a crossover representation of signal flow.

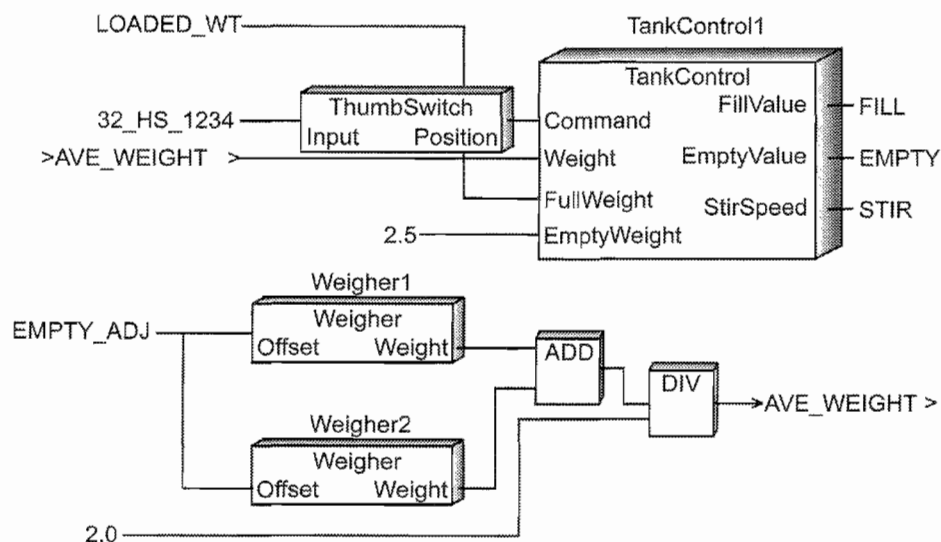


Figure 5.2
A typical FBD showing a tank control program

It is also possible to have direct or literal constants (such as the input EmptyWeight to FB instance TankControl1 having a fixed value of 2.5) as inputs to functions and function blocks.

5.4 Signal flow

As we discussed in the section on methodology, an FBD network is a representation based on signal flow between different program components.

Signals are considered to flow:

- From outputs of functions or function blocks
- To Inputs of other functions or function blocks

Outputs of function blocks are updated as a result of function block evaluations and thus changes of signal status are propagated from left to right across the FBD.

It is sometimes required to invert (or negate) the status of Boolean signals that is a TRUE signal has to be made as FALSE or vice versa when they flow from one function block to another. This is represented by a small circle at the input point in case it is a negated input; and at the output point in the case of a negated output. Figure 5.3 shows this feature.

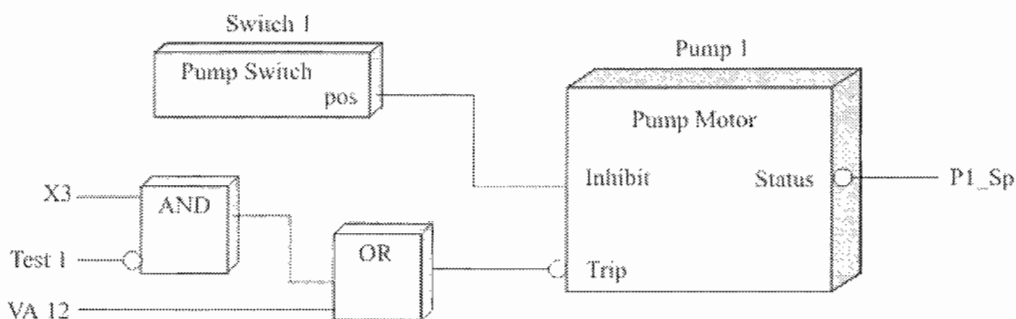


Figure 5.3
Negated boolean signals in an FBD

This representations is not supported in some implementations which use a NOT function to represent negation. Figure. 5.4 show such representation of the FBD network illustrated in Figure. 5.3. These two FBD networks are functionally the same.

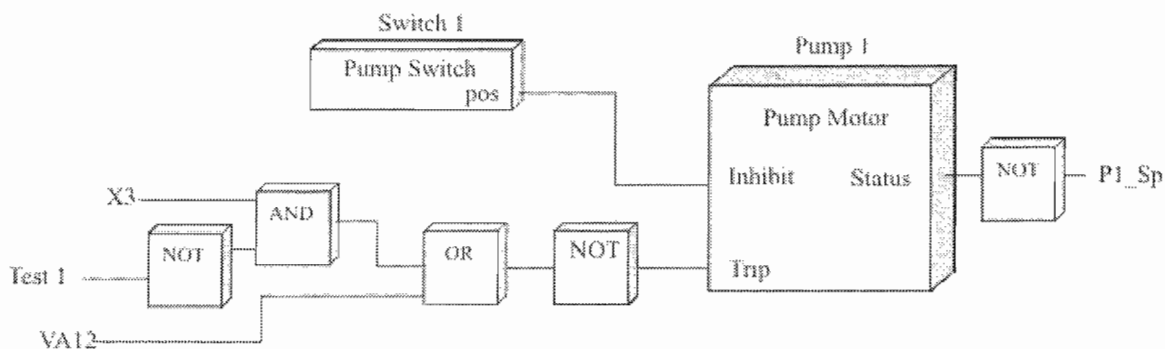


Figure 5.4
Negated boolean signals in an FBD represented by NOT function

5.5 Feedback path

The standard allows signal feedback, which is a special case of signal flow. In the preceding examples, we saw the signals proceeding from left to right that is the output of a function block forms the input to another function block to the right. Feedback is a case when the output of a function block forms the input of a preceding function block (i.e., one to the left in the FBD network).

In figure 5.5, which illustrates a feedback path where the output 'Level' from FB Load1 is fed back to FB Loop1 as the input ProcessValue. This representation is called explicit feedback path.

A feedback path can be Implicit in which case a connector will be used to link the output of a block on the right side of the FBD network to the input of a block on the left-hand side of the diagram.

A feedback path implies that a value within the feedback path is retained after the FBD network is evaluated and used as the starting value for the next evaluation. The system behaviour will however depend on the order in which the blocks are evaluated. In the example shown, if Loop1 is evaluated before Load1 then input ProcessValue will be set to the value of Load1.Level as obtained in the previous evaluation. But if Load1 is evaluated before Loop1 then the input FlowRate will be set to the value of Loop1.Output as retained from the previous evaluation. This means that different implementations may evaluate the same FBD network differently depending on the order of evaluation. In order to overcome this problem, the standard stipulates that an implementation may provide a facility to define explicitly the order of evaluation of the function blocks in an FBD. The exact method of defining this order is however left to the implementation design.

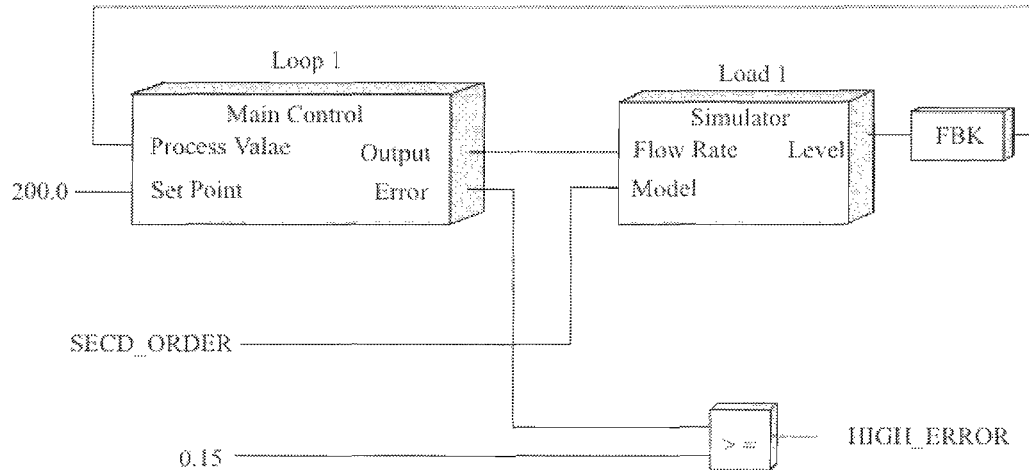


Figure 5.5
Example of explicit feedback path

5.6 Network layout

It is important to arrange the Function Block Diagram Networks in such a way that signal flow can be traced easily. The blocks should be placed so that signal path crossovers are avoided and no unnecessary direction changes are required in the signal path. The standard does not impose any specific limits on the size or complexity of a network.

However, it permits an implementation to limit the number of function block type specifications and the number of instances of a function block within one configuration. This will limit the size and complexity of the FBD networks that a particular implementation can support.

5.7 Function execution control

The execution control of functions in an FBD network depends, by default, on the position of the block. For example, in the FBD network shown in Figure. 5-5, the function \geq is evaluated after the function block Loop1. This type of control is called an implicit execution control. The control of the functions can however be made explicit by using the Function Enable input EN. This input EN is a Boolean variable. A function will be evaluated only if the input EN is TRUE. While the value of EN remains FALSE, the function will not be evaluated and its output will not be generated.

Similarly, the function output ENO is a Boolean variable which changes state from FALSE to TRUE when the function has been evaluated. By using a combination of EN input and ENO output, it is possible to trigger certain values to be generated when specific conditions are met. Figure 5.6 illustrates an example.

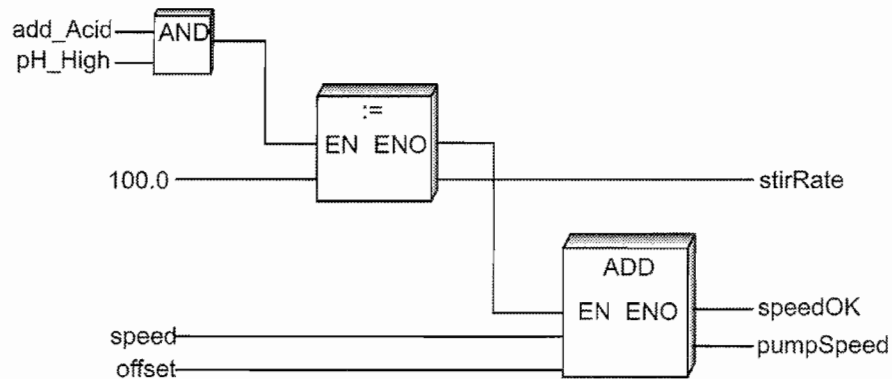


Figure. 5.6
Example of explicit function execution control

In this example, as long as either of the inputs `add_Acid` or `pH_High` is false, the input EN to the Assignment function `' := '` and ADD function are FALSE and these functions are not evaluated. When both these inputs `add_Acid` and `pH_High` are TRUE, the EN input to the Assignment function becomes TRUE. This function moves the value 100.0 to the output variable `stirRate`. When this operation is complete it sets the value of ENO as TRUE, which in turn causes the function ADD to be evaluated.

There are two issues to be noted. The standard does not clearly state about the use of the output value of a function whose EN input is FALSE. The other point is that a function in which the evaluation results in an internal error such as an overflow, will not set the output ENO.

5.8 Jumps and labels

In an FBD network, it is possible to transfer control from one part of a network to another using a 'jump' facility. For this purpose, a section of an FBD network can be represented by a 'label'. By assigning a Boolean value to a label identifier, control can be transferred

to the part of the network associated with that particular label. A double arrow, followed by the label identifier, represents these Boolean signals. See figure. 5.7 below.

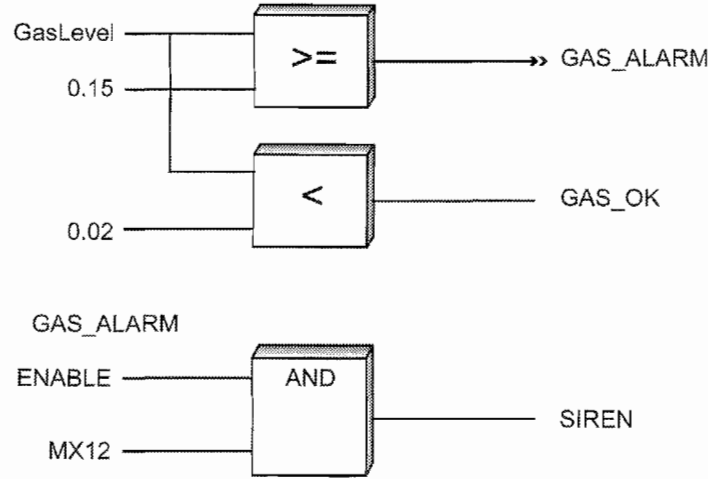


Figure 5.7

Example of transfer of control using 'jump' facility

In this example, when the value of the input GasLevel is equal to or greater than the value 0.15, the comparator function \geq is evaluated and sets the boolean output signal of the Label Identifier Gas_Alarm TRUE. This causes the control to be transferred to the section of the FBD network identified by the label Gas_Alarm.

It is to be noted that the standard is not clear on what happens to the rest of the network after a jump has been executed. One way is to complete the evaluation of all the linked blocks and then execute the jump. Another possibility is to stop evaluation and transfer control to the part of the FBD represented by the Label identifier. Because of this ambiguity, the IEC guidelines for using the standard 1131-3 recommends that jumps should not be used within an FBD network. For situations where such transfer of control is essential, it is recommended that Sequential Function Charts (the last of the five IEC 1131-3 languages) be used for programming.

5.9 Network evaluation rules

The sequence of evaluation of function blocks is implementation-specific, since the standard does not define any order for evaluation of the functions. Most implementations follow an evaluation sequence from left to right. Some implementations may do it starting from top and proceeding downwards.

However, in the interest of ensuring consistency of the data transferred between functions and function blocks, the following rules should be adopted:

- No element in a network shall be evaluated unless the states of all inputs have been evaluated. This means that before proceeding to evaluate the functions and function blocks in an FBD network, all input values coming from other elements must be available
- The evaluation of a network element shall not be complete until the states of all its outputs have been evaluated. In other words, the outputs of a function block cannot be considered as available until all its outputs have been evaluated

- The evaluation of a network is not complete until the outputs of all its elements have been evaluated. That is to say, that all outputs of functions and function blocks should be updated before the evaluation of an FBD network can be taken to be complete
- When data are transferred from one FBD network to another, all the elements coming from the first network should be produced by the same network evaluation. The evaluation of the second network shall not start till all the values from the first network are available. This rule is of particular interest in a network where parts of a network run under the control of different tasks at different scan cycle timings

5.10 Summary

Function Block Diagram expresses control behaviour of an industrial control system using a network of software blocks with signals transferred between them. It is similar in concept to an electrical circuit diagram, where interconnections between blocks represent signal flow. It is ideal for use in a wide range of situations including closed loop control and Boolean logic.

While the standard includes jump instructions for transfer of control, the ambiguities inherent in the standard make such use inadvisable. FBD networks are better suited to express continuous circuit behaviour and recourse should be taken to Sequential Function Charts when there is a need for sequencing. For ensuring proper execution of the program and correct control functions, the basic rules for network evaluation should be followed.

Ladder diagrams

This chapter contains information on the graphical programming language Ladder Diagram (LD) described in IEC-1131 part 3. Representation of control systems using ladder diagram is explained using examples and rules for evaluation of LD networks are also illustrated.

Objectives

On completing the study of this chapter, you will learn:

- The basic concept of Ladder Diagram based programming
- Graphical symbols applicable to Ladder diagrams
- Boolean Expressions using Ladder Diagrams
- Integrating Functions and Function Blocks within Ladder Diagrams
- Feedback paths
- Use of Jumps and labels in Ladder Diagrams
- Network evaluation rules
- Portability of LD programs to other languages

6.1 Introduction

Ladder Diagram is the second graphical language for programming industrial control systems described in IEC-1131-3. Ladder diagrams are derived from electrical circuit diagrams, which have been conventionally used to represent relay logic operations. Many of the symbols and terminology have also been adopted from the circuit diagrams. In other words, Ladder Diagrams are graphical representations of Boolean expressions, but they can also include other information normally not possible with Boolean expressions.

Ladder diagrams can be used to define the behaviour of:

- Functions
- Function Blocks
- Programs
- Transitions in Sequential Function Charts

6.2 Basic concept

Figure 6.1 show a typical example of ladder diagram. In this diagram, DrainClose, DoorClose etc. are Boolean variables represented as contacts. PumpOff is a variable whose state is decided by the variables DrainClose, DoorClose and Manual and is represented as a coil.

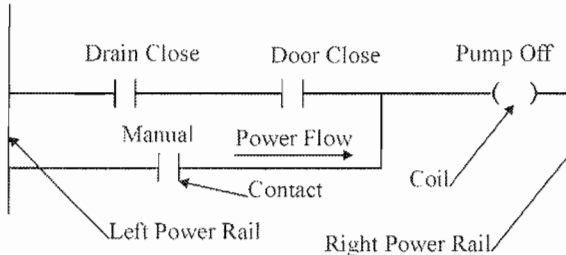


Figure 6.1
Example of a ladder diagram

As can be seen from this example, a ladder diagram has two vertical power rails, one on the left and another on the right side of the diagram. The left vertical rail carries the power to the coil (notionally) through the contacts. The contacts are arranged along horizontal ‘rungs’ and the power flows through these rungs to energise the coil, which is on the right hand side of the logic diagram, when the logic condition represented by the contacts is TRUE (AND logic). Alternative paths can be present with other contacts in them (in parallel), which can be used to build OR logic. In the above example Drain Close and Door Close represent an AND logic. Contact Manual which gives an alternative path is part of the OR logic. PumpOff will become TRUE when DrainClose AND DoorClose are both true OR Manual is true. In Structured Text language the following expression would describe this logic.

Pump Off := (Drain Close AND Door Close) OR Manual;

A contact can be considered as providing a ‘read only’ state of the variable represented by a coil since it cannot change the state of the variable itself. On the other hand, the coil can be considered as a ‘write only’ access to the same variable. The network that is associated with the logic of a coil is called as a ‘ladder rung’.

The standard permits the right side power rail not being shown and is therefore omitted in some implementations. Also note that the names of the variables are shown above the respective symbols.

6.3 Graphical symbols used in ladder diagram

As in the case of Function Block Diagrams, the standard permits the use of semi-graphic and full graphic representations in an implementation. Since the full graphic display depends on features and capabilities specific to the hardware used in the implementation, these are not defined in the standard in minute detail.

The standard defines various types of contacts in Figure 6.2. The first column describes contact behavior, the second the semi graphic symbol and the last column full graphic symbol.


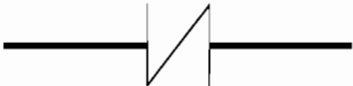

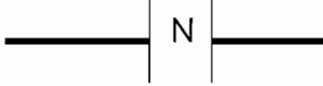
<u>Normally Open Contact</u> Contact where power flow occurs from left to right when the associated variable is 1	---- ----	
<u>Normally Closed Contact</u> Contact where power flow occurs from left to right when the associated variable is 0	---- /----	
<u>Positive transition sensing Contact</u> Contact where power flow occurs from left to right for one ladder diagram evaluation when the associated variable changes from 0 to 1	---- P ----	
<u>Negative transition sensing Contact</u> Contact where power flow occurs from left to right for one ladder diagram evaluation when the associated variable changes from 1 to 0	---- N ----	

Figure. 6.2

Representation of contacts in a ladder diagram

The contact symbols shown in the above diagram include contacts, which detect normal and inverse states, as well as rising and falling states of power flow. The latter group causes the associated coil to be set for one evaluation of the Ladder Diagram.

Coil

The Coil is set to the state according to the power flow coming from the left hand link. If power flow is ON, the coil state is set ON.

**Negated Coil**

The negated coil is set to the opposite state to the power flow coming from the left hand link. If the power flow is ON, the coil state is set OFF.

**SET Coil**

This coil is set on the ON state when there is power flow coming from the left hand link. The coil remains set until it is RESET.

**Figure. 6.3**

Representation of coils in a ladder diagram

Symbol for coils and special types of coils are shown in figures 6.3, 6.4 and 6.5.

The coil symbols SET and RESET allow a variable to be latched in ON condition by SET signal and cleared by the RESET signal. Figure 6.5 shows representation of special types of coils.

The Retentive type coils shown in the above diagram are useful in representing variables, which should be held in a particular state in the event of a PLC power failure and subsequent restoration. This may be applicable to position of interlocks, plant operation modes etc., which should remain in the correct state, unchanged when the PLC is powered up after a power failure. (This is similar in action to stay-put selector switches used in conventional control desks/panels).

Positive and negative transition coils detect the change of power flow and change the state of the associated variable for one evaluation of the ladder diagram.

RESET Coil

This coil is reset to the OFF state when there is power flow coming from the left hand link. The coil remains OFF until it is RESET.

**Retentive memory coil**

The same behaviour as the normal coil, except that the state of the associated variable is retained on PLC power fail.

**SET retentive memory coil**

The same behaviour as the SET coil except that the state of the associated variable is retained on PLC power fail.

**Figure. 6.4**

Representation of Special coils in a ladder diagram

RESET retentive memory Coil

The same behaviour as the RESET coil except that the state of the associated variable is retained on PLC power fail.

--(RM)-- —(RM)—

Positive transition-sensing coil

If the power flow on the left hand link changes from OFF to ON, the variable associated with the coil is set ON for one ladder rung evaluation.

---(P)--- —(P)—

Negative transition-sensing coil

If the power flow on the left hand link changes from ON to OFF, the variable associated with the coil is set ON for one ladder rung evaluation.

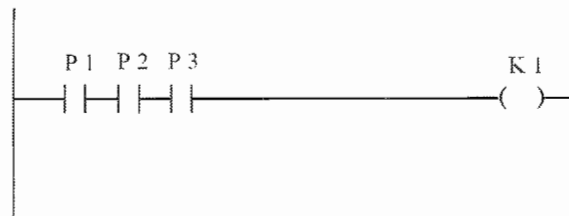
---(N)--- —(N)—

Figure 6.5

Representation of more special coil types in a ladder diagram

6.4 Boolean expressions using ladder diagrams

As seen in the previous section, the contacts in combination with a coil arranged in a ladder rung can be used to represent various combinational logic states. Figure. 6.6 represents AND logic of inputs A1, A2 and A3. When all these are TRUE, the variable X1 represented by the coil is set TRUE.

**Figure 6.6**

Representation of AND logic in a ladder diagram

The above diagram is equivalent to the following expression in a Structured Text program.

```
X1 := P1 AND P2 AND P3
```

Alternative parallel paths joined by vertical connectors are used to represent OR logic as shown in Figure 6.7.

In the above figure the variable X1 corresponds to the following Structured Text expression.

```
X1 := (P1 OR R1 AND P2 AND P3) OR (S1 AND S2)
```

Thus various combinatory logic conditions can be represented using the symbols given in the figures 6.2 to 6.5.

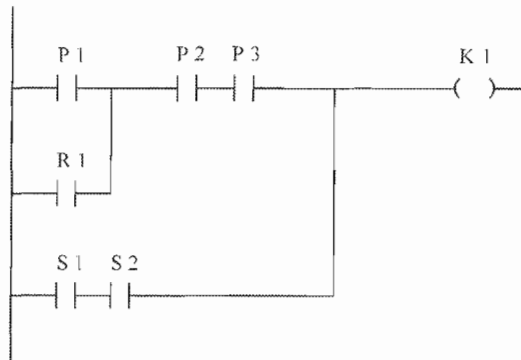


Figure. 6.7
Representation of combined AND OR logic in a ladder diagram

6.5 Integrating functions and function blocks within ladder diagrams

It is possible to include functions as well as function blocks within Ladder Diagram networks provided certain basic conditions are fulfilled. These are:

- The Functions/Function Blocks should have Boolean inputs and output
- Inputs can be directly driven from ladder rungs
- Outputs can drive coils

An example is shown in Figure 6.8.

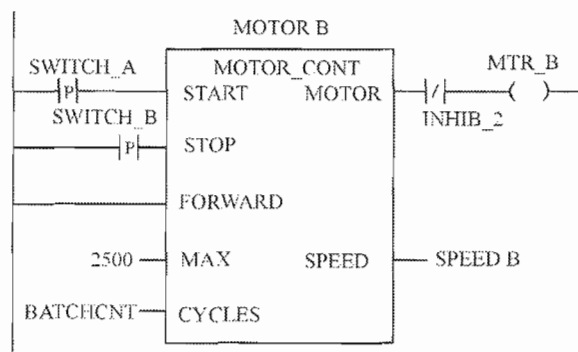


Figure. 6.8
Function block embedded within a ladder diagram

In this example, SWITCH_A and SWITCH_B represent contacts of a Pushbutton for starting and stopping a motor. These are Boolean inputs to the Function Block MOTOR_B. Output MOTOR is a Boolean variable which is used to set the variable MTR_B depending on the status of variable INHIB_2.

It may be noted that the Function Blocks represented within a Ladder diagram can have other non-Boolean inputs and outputs as well, as shown in the example. Value 2500.0 is a constant value assigned to input MAX. Similarly the function block has an input CYCLES whose value is from variable BATCHCNT.

Two other aspects need mention. The standard stipulates that there should be at least one Boolean input in a Function Block, which is to be connected to the left-hand power rail. Secondly, the technical report on the application of the standard provides for explicit

evaluation of functions and function blocks using EN input and ENO output. (Figure 6.9 shows an example of the latter).

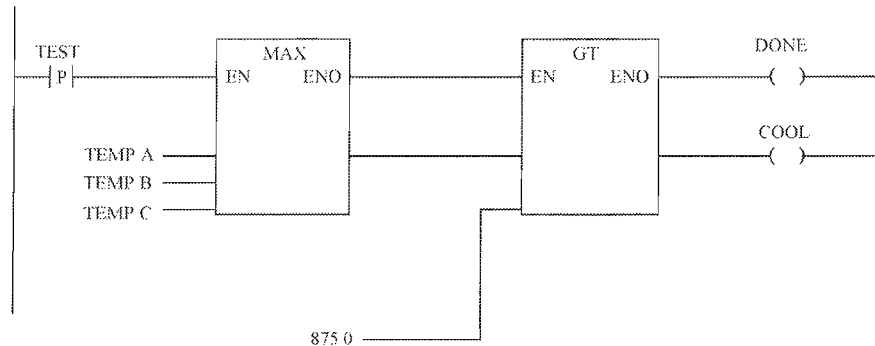


Figure 6.9

Use of EN and ENO parameters in a function block embedded within a ladder diagram

In figure 6.9, there are two functions. Function MAX evaluates the maximum value of three variables TEMPA, TEMPB and TEMPC. Function GT (Greater Than) evaluates whether the input received from MAX function is greater than 875. If TRUE, it sets the variable COOL to ON state. The above operation is controlled using the variable TEST, which drives the enabling (or control) input to function MAX. TEST is a positive transition contact and causes power to flow to MAX for one evaluation. When the evaluation is done the output ENO of MAX is set TRUE which sets input EN of function GT and enables its evaluation. The output DONE of GT indicates that the function GT has been evaluated.

6.6 Feedback paths

Ladder diagrams can incorporate feedback loops when a contact of the variable that is being evaluated occurs in the ladder rung, which powers this variable. Figure 6.10 illustrates this usage.

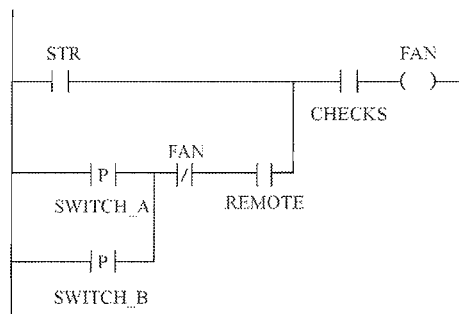


Figure 6.10

Feedback in a ladder diagram

In this example the contact FAN occurs in the rung which contains coil FAN. The value of FAN is taken from the previous evaluation of this variable FAN and the current state of the variable FAN is evaluated accordingly. However if any of the subsequent rungs contain this variable the result as obtained in the current evaluation will be used for evaluation.

6.7 Jumps and labels

It is possible to transfer control from one part of a Ladder diagram to another part using graphical Jump and a label identifier. This is illustrated in Figure. 6.11.

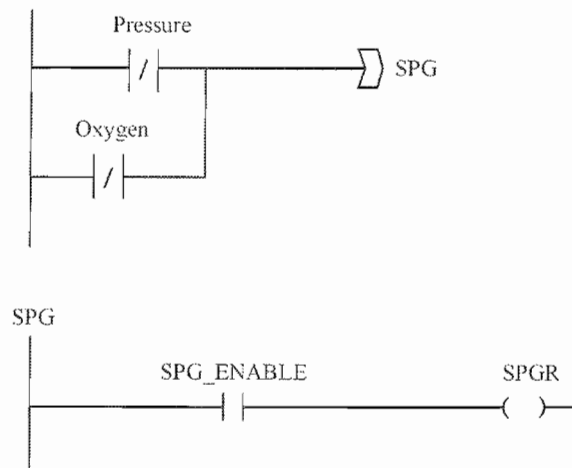


Figure 6.11
Graphical Jump in a ladder diagram

In the diagram shown, control is transferred to the part of the network identified by label SPG when the variables represented by contacts PRESSURE and OXYGEN are TRUE. The double arrow sign (>>) indicates that a jump has to be executed. However IEC 1131-3 does not clearly stipulate the behaviour of a ladder diagram program once a jump has been executed. Therefore, IEC guidelines on using this standard recommend that jumps may not be used within an LD network and that Sequential function charts be used to achieve such functionality.

6.8 Network evaluation rules

The sequence of evaluation of Ladder diagrams starts from the top rung and proceeds downwards.

The following rules are adopted to ensure that there is no inconsistency or ambiguity in evaluation:

- No element in a network shall be evaluated unless the states of all inputs have been evaluated. This means that before proceeding to evaluate the ladder rungs, functions and function blocks in a ladder network, all input values coming from other elements must be available

Note: This rule may not be applicable to networks that involve feedback, since the input value of feedback variable may not be available before evaluation of the network

- The evaluation of a network element shall not be complete until the states of all its outputs have been evaluated. In other words, the outputs of a ladder diagram cannot be considered as available until all its outputs have been evaluated.
- The evaluation of a network is not complete until the output of all its elements have been evaluated. That is to say, that all outputs of ladder rungs, functions and function blocks should be updated before the evaluation of a ladder network can be taken to be complete

- When data are transferred from one network to another, all the elements coming from the first network should be produced by the same network evaluation. The evaluation of the second network shall not start till all the values from the first network are available

6.9 Portability

Ladder diagram networks where simple logical conditions are involved can be expressed as function block diagrams or using Structured Text Programming language. For example, the Ladder Network shown in figure 6.7 can be represented by an equivalent Function Block Diagram shown in figure 6.12 as well by a Structured Text statement as shown in an earlier section.

However with Ladder diagram networks involving the use of enabling inputs and outputs EN and ENO, translation into Structured Text language will not be possible. However, such ladder Diagram networks can be translated into FBD networks since FBD construct permits use of EN and ENO variables. Figure 6.13 below shows a translation of the ladder network illustrated in Figure. 6.9.

In this case, the transition sensing contact has been represented using a function for detecting the rising edge of a signal (R_TRIG).

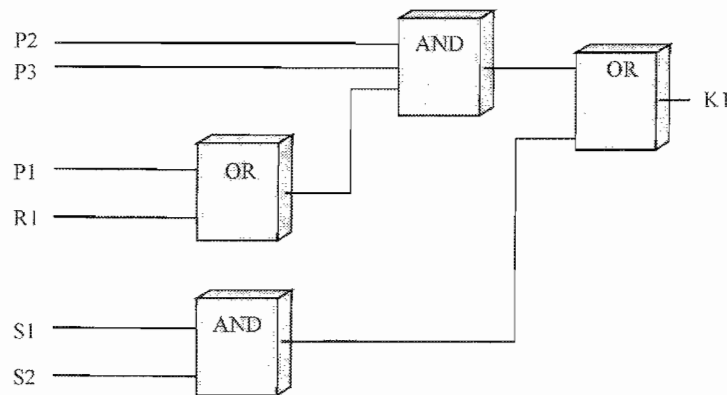


Figure 6.12
FBD equivalent of the ladder diagram shown in figure 6.7

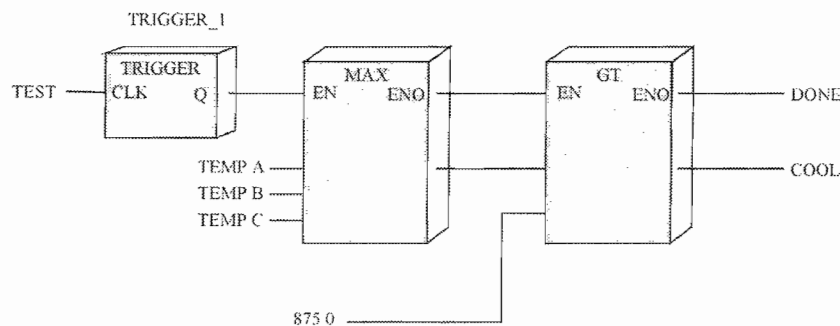


Figure 6.13
FBD equivalent of ladder diagram shown in figure 6.9

6.10 Summary

Ladder diagrams are derived from the electrical circuit diagrams, which have been conventionally used to represent relay logic operations. LD network is particularly suitable for programming combinatory logic encountered in process interlocking.

Functions and function blocks can be embedded in a logic diagram provided that:

- The Functions/Function Blocks have Boolean inputs and outputs
- Inputs can be directly driven from ladder rungs
- Outputs can drive coils

The Functions and function blocks embedded in a Ladder Diagram Network can have EN/ENO type inputs and outputs. Though the standard provides for Jump type constructs for transfer of control, their use is not recommended. Network evaluation should follow the rules contained in the standard. The translation of LD networks to FBD networks is usually straightforward but it may not be possible to translate LD networks to Structured text and vice versa for all types of program constructs.

Instruction List

This chapter contains information on the programming language Instruction List (IL) described in IEC-1131 part 3. Representation of control systems using IL is explained using examples.

Objectives

On completing the study of this chapter, you will learn:

- Basic structure of IL programming language
- Standard operators, Conditional operators, jumps and labels
- Calling functions and function blocks
- Portability and other issues

7.1 Introduction

Instruction List (IL) is the fourth of the five programming languages covered in IEC-1131-3 viz., Structured Text, Function Block Diagram, Ladder Diagram, Instruction List and Sequential Function Chart. Instruction List is a low-level language using which the behavior of functions, function blocks and programs can be expressed.

7.2 Structure of IL programming language

7.2.1 Basics

Instruction List (IL) is similar in structure to machine assembler language and is based on a review done by IEC of many proprietary low-level Instruction List languages used by PLC manufacturers. The standard includes a wide range of operators used in these languages. IL is simple, easy to learn and can be used to solve problems of limited complexity. It is similar to the instruction set used by microprocessors on which many PLCs are based. In comparison, Structured Text, which resembles higher-level languages

such as Pascal, needs to be compiled to the appropriate machine assembly language before it can be executed on a PLC.

While some of the implementations treat Instruction List as the basic language to which any other language will be translated, others treat Structured Text as their base language on the consideration that IL is better suited to small PLCs only. This is despite the fact that IEC 1131-3 regards all languages as equally acceptable and does not recommend that a particular language be considered as the base language. As far as readability and debugging is concerned, IL is more difficult compared to Structured Text. The best practice would be therefore, to write complex logical problems in ST and use IL where there is a need for highly optimized and efficient code.

7.2.2 Instruction structure

IL consists of a series of instructions, each being a line of code. An instruction contains an operator and one or more operands. Operands are variables or constants that are manipulated by the operator. Operators usually operate on a single operand but some operators can handle more than one operand (which will be separated by commas).

An instruction stores a value to a register (a defined location in the memory) or use the value already stored, modify it and then store the changed value back in the register. The register is thus the result of an instruction (as per the definition of the standard) and is therefore called as the 'result register'. The result register is also referred to as the accumulator (or just accu). There are also operators that compare the value of a variable with the value currently stored in the result register and use the comparison to store a new value into the register or to perform other operations. The operation may be to jump to a specific instruction (skipping a few lines of instruction in the middle) and then continue from that instruction onwards. A label identifies the instruction to which the jump should be performed. A label must be followed by a colon mark. The sequence in which an instruction is written is: Label (optional), Operator, Operand and comment (optional).

The comment, which is used to document the code so that it can be understood clearly, should be placed within parentheses followed by * and should be at the end of the line. The standard does not permit a comment to be written anywhere else, either in the beginning of the instruction or between operator and operand. Figure 7.1 below shows a typical Instruction list.

Label	Operator	Operand	Comment
	LD	RPM	(*Load the variable RPM to result register*)
	GT	500	(*Test if Value in result register >500*)
	JMPC	Volt_A	(*If NOT jump to Volt_A*)
	LD	Voltage	(*If TRUE Load value of variable Voltage*)
	SUB	5	(*Subtract from it a value of 5*)
	ST	Voltage	(*Store the new value to Variable Voltage*)
Volt_A:	LD	1	(*Load value 1 *)
	ST	%Q70	(*Store the value to output 70*)

Figure 7.1
Typical set of IL instructions

The above program tests whether the value of the variable RPM is greater than 500. For this purpose, the current value of variable RPM is loaded into the result register by the first LD instruction after which the register contains an Integer being the current value of RPM. Then by using the GT (Greater Than) operator it is verified whether the current value of the register is greater than the Constant 500. If TRUE, the value 1 is stored in the register and if FALSE the value 0 is stored. Note that at the end of this operation, the value of variable RPM is no longer available in the register and will be replaced by the Boolean value of 1 or 0.

If the register value is FALSE then the JMPNC instruction causes the sequence to jump to the line represented by the Label Volt_A. If the register value is TRUE, the execution continues without the jump and loads the variable of Voltage to the register and using the subtraction operator (SUB) deducts a constant of 5 from this value. Now the register will contain the result of the subtraction. The next instruction ST stores the new content of the register to the variable 'Voltage'. If the jump is performed, the above intervening steps are omitted and the value of variable Voltage is not reduced by 5 and the instruction starting with the line containing the label Volt_A is executed directly. The last two lines of the program load the Boolean value 1 to the result register and using the ST (store) instruction the value 1 is stored to the output represented by the number 70.

Note that in order to improve readability, the IL program is written using fixed Tab positions (or as a table) so that the component parts of each line (Label, Operator, Operand and Comment) are clearly identifiable by a gap.

7.2.3 Comparison with Structured Text (ST) language

The above example shows that an IL program has to break down a task into a number of small instruction steps to be executed in a sequence and the flow is controlled using jump instructions and corresponding labels. Comments written at the end of each line are useful in making the meaning of each instruction clear to those who read the code. In contrast, a high level language will contain fewer lines and does not require so many comments since the language itself resembles the natural language closely. The IL program in Figure 7.1 can be re-written in Structured Text (ST) language as:

```
IF RPM > 500
  Voltage := Voltage-5
END_IF
%Q70 := 1
```

Compare the simplicity and readability of the ST program and fewer lines of code compared to the IL program to perform the same set of operations.

7.2.4 General semantics of IL expressions

The IL language instructions thus generally follow the expression:

```
New Result      :=      Current Result Operator      Operand
```

For example, the subtraction instruction in the example in Figure 7.1 can be represented as :

```
New Result      :=      Current Result SUB      5
```

That is, the operator SUB subtracts the value 5 from the Current value held in the result register and the New Result obtained by this operation is written to the result register.

Similarly a comparison operator such as GT compares the Current Result with the operand and the New Result becomes 1 if the comparison is TRUE. This value is stored to the result register.

```
New Result      :=      Current Result GT      500
```

In case the above comparison is FALSE, the value 0 is stored in the result register.

There are some operators whose action does not fit into the above generalization. For example the store operator ST stores the Current Result to the variable name defined by the operand. For example the instruction can be expressed as:

Operand := Current Result

	ST	Voltage	
--	----	---------	--

7.2.5 Modifiers for deferred execution

Use of modifiers in the form of a bracket (parenthesis) is allowed by the standard to defer the execution of an instruction. These are useful when it is required to evaluate an arithmetic function containing nested expressions such as:

$A + (B * (C + D))$.

This operation can be performed using the deferred execution modifier pair '(' and ')' as follows.

```
LD      A      (*Load A          1*)
ADD (    B      (*Defer ADD, Load B  2*)
MUL (    C      (*Defer MUL, Load C  3*)
ADD      D      (*Add D          4*)
)        (*Do deferred MUL operation  5*)
)        (*Do Deferred ADD operation  6*)
```

Note: The numbers shown at the end of the comment line are the instruction numbers.

At the end of this operation, the result register will contain the value of the expression $A + (B * (C + D))$. However in this process a stack of result registers need to be created so that intermediate values arising out of deferred execution can be held in the memory. The above IL program will create a stack of 3 registers during its execution. Figure 7.2 below shows the values in result stacks at the end of each line of instruction. The stack of result registers is represented as Register0, Register1 and Register2.

Instruction No.	Register0	Register1	Register2
1	A		
2	A	B	
3	A	B	C
4	A	B	C + D
5	A	B * (C + D)	
6	A + B * (C + D)		

Figure 7.2

Illustration of result register stack

Jump instructions if used within parentheses would cause problems with correct evaluation and even though the standard does not specifically state that their use is illegal it is recommended that they be not used in program modules containing deferred execution steps.

7.2.6 Other modifiers

Other modifiers such as N (for negation) and C (conditional) are also used in IL language. N is used in combination with Boolean operators to invert the result of a Boolean operation. For example:

```
LD          Bool-1
ANDN        Bool-2
```

The above instructions will invert the result of an AND operation of Boolean variables Bool-1 and Bool-2 and store the result to the result register.

Modifier 'C' is only meant for use with a jump instruction. Use of it makes the operation of the instruction conditional. Use of both N and C with a jump instruction inverts the condition for the jump operation.

For example:

```
JMP          Lab-1    denotes an unconditional jump to the label Lab-1.
```

```
JMPC         Lab-1    denotes a jump, which is conditional on the current value of
the result register being TRUE.
```

```
JMPNC        Lab-1    denotes a jump, which is conditional on the current value of
the result register NOT being TRUE.
```

Figure 7.3 shows the various IL operators including Comparison and Jump operators and modifiers that can be used with these operators. The 'comments' column gives the explanation of use of each operator.

Operator	Modifiers	Operand	Comments
Main Operators			
LD	N	ANY	Load operand into results register
ST	N	ANY	Store results register into operand
S		BOOL	Set operand TRUE
R		BOOL	Set operand FALSE
AND	N , (ANY	Boolean AND
&	N , (ANY	Boolean AND (Alternative)
OR	N , (ANY	Boolean OR
XOR	N , (ANY	Boolean exclusive OR
NOT		ANY	Negation
ADD	(ANY	Addition
SUB	(ANY	Subtraction
MUL	(ANY	Multiplication
DIV	(ANY	Division
MOD	(ANY	Modulo division
Comparison and Jump operators			
GT	(ANY	Comparison Greater than
GE	(ANY	Comparison Greater than or equal to
EQ	(ANY	Comparison equal
NE	(ANY	Comparison NOT equal
LE	(ANY	Comparison less than or equal
LT	(ANY	Comparison less than
JMP	C, N	ABEL	Jump to Label
CAL	C, N	NAME	Call function block
RET	C, N		Return from function or function block
)			Execute last deferred operator

Figure 7.3
List of IL operators

Note:

- Operators with multiple modifiers can operate with one or more of them together. An example of this is given in Figure 7.4 below
- Operand ANY indicates that the operator can be used for more than one data type such as Integers, Floating point numbers, real, date/time etc
- S and R operators can be used for Boolean data type only

Operator	Action
AND	Boolean AND
AND (Deferred Boolean AND
ANDN	Inverted Boolean AND
ANDN (Deferred Boolean AND with result inverted

Figure 7.4*Multiple modifiers-example of usage*

7.3 Calling functions and function blocks

The standard provides different alternative methods for calling functions and function blocks from within IL programs. The method of calling function blocks differs from that for calling functions. There are three different ways in which a function block can be called.

7.3.1 Function block - Formal call with an input list

In this method the operator CAL is followed by function block parameters and their values. The parameters can either be given directly or calculated. Each parameter must have a specific name.

An example of such a call is given below.

```

CAL      LOOP-1 (
SP      :=    200
PV      :=    (
LD      %IW30
ADD     5
)
)

```

The function block instance called is Loop-1. The parameter SP takes a value of 200. The parameter PV has its value calculated by addition of 5 from the Input Word 30.

7.3.2 Function block - Informal call

In contrast to the earlier method, an informal call first defines the values of parameters SP and PV before calling the function block using the operator CAL. The same function block call cited earlier will be written in this method as follows. Loop1.SP indicates that parameter SP of function block instance Loop-1.

```

LD      200
ST      Loop-1.SP
LD      %IW30
ADD     5
ST      Loop-1.PV
CAL     Loop-1

```

7.3.3 Function block - call using input operators

This method can be used only with certain specific IEC Standard function blocks. Also a number of special operators are reserved for use with such function blocks. A list of these reserved operators for use with standard function blocks is shown in Figure 7.5.

Operator	Function block type	Comments
S1,R	SR Bistable	Set and Reset SR bistable
S,R1	RS Bistable	Set and Reset RS bistable
CLK	R_Trig, rising edge detector	Clock input to rising edge detector function block
CLK	F_Trig falling edge detector	Clock input to falling edge detector function block
CU, R, PV	CTU, up-counter	Control parameters for up-counter function block
CD, PV	CTD, down-counter	Control parameters for down-counter function block
CU, CD, R, PV	CTUD, up/down counter	Control parameters for up/down-counter function block
IN, PT	TP, Pulse timer	Control parameters for pulse timer function block
IN, PT	TON, ON delay timer	Control parameters for on delay timer function block
IN, PT	TOF, Off delay timer	Control parameters for off delay timer function block

Figure 7.5

IL operators reserved for function block operations

IL language can reference function block inputs and write values to outputs using variables specified in the function block declaration.

7.3.4 Calling a function - Formal call

In this type of invoking functions, the name of the function is given first, followed by values of input parameters.

For example:

```
SHR (
  IN := %IW20
  N := 5
)
```

The above code calls function SHR (shift right) giving values to input parameters IN and N.

7.3.5 Calling a function - Informal call

In this method, when a function is invoked the function name is first given followed by the input parameter values. The first parameter value is supplied by the current value of the result register.

For example:

```
LD      %IW20
SHR     5
```

This is how the call will be made for function SHR using the informal call method. The difference is thus only in the syntax for specifying the value of the first input parameter.

In both cases the result returned by the function is available in the result register and can be loaded to an output word by using an instruction such as:

```
ST      %QW20
```

Which loads the result to output word 20.

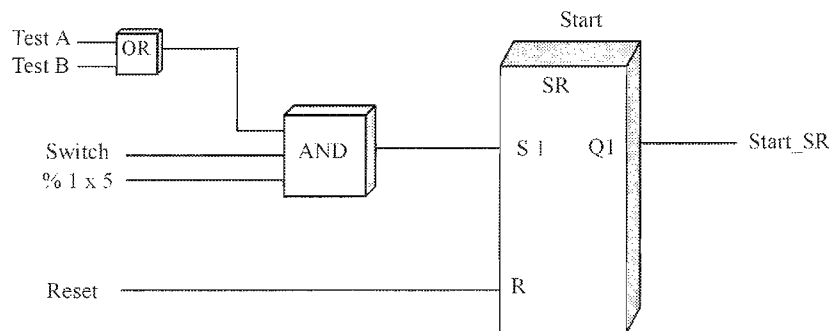
7.4 Portability and other issues

Conversion of a program from IL into other languages is not easy and can be done only when a restricted range of operators is used as per a strict format. Similarly, translating from other languages to IL, though relatively easier, is not straightforward and has its own limitations.

Figure 7.6 below shows an example of how a Function Block Diagram can be translated into an IL program.

Another issue is that IL language is not fully described in the standard and many issues are left open. The way instructions operate is not always defined in detail. Method of storing of array and string type variables (which are of multi-element type) in the result register is not clearly spelt out. Run time behavior under various error conditions such as use of an incorrect data type is also not defined clearly.

All this places a heavy responsibility on those who write or maintain IL programs. The programmer has to ensure that no incorrect data types, which do not match with the respective operator, are used. Also, the code must be properly documented to ensure that in the event of problems, it can be understood clearly by anyone who reads it. Jump instructions should generally be avoided since they interrupt the program flow and make understanding the code difficult.



This can be written in Instruction List as follows :

LD	Test A	(* Test A or *)
OR	Test B	(* Test B*)
AND	Switch	(* AND Switch*)
AND	% 1 x 5	(* AND Input 5*)
ST	Start.S1	(* Set input of Start *)
LD	Reset	(* Load value of Reset *)
ST	Start.R	(* Store in reset input *)
CAL	Start	(* Call fb. Start *)
LD	Start.Q1	(* Load output Q1*)
ST	Start_SR	(* and store in Start_SR *)

Figure. 7.6
Portability example

7.5 Summary

In this chapter we learnt the following:

- Instruction List (IL) is a text based programming language and closely resembles low-level machine language instructions used for programming microprocessors. It is simple to implement and is therefore adopted by many smaller PLC manufacturers
- Programming complex tasks is difficult with IL when compared to a higher level language such as Structured Text
- IL is generally used for simple problems requiring highly optimized code
- IL has well defined semantic rules, which are used in the instructions. An instruction contains operators with or without modifiers, operands and labels (where required). Comment, though optional is highly recommended to ensure that the code is readable and can be understood properly
- IL provides the facility of calling function blocks and functions using several alternatives formats. The standard defines special operators which can be used when calling IEC standard functions
- While jump instructions are allowed, their use makes the program flow discontinuous and difficult to follow. Incorrect use (say, within a program block containing deferred instructions) may give rise to errors in execution. Use of jump instruction is therefore not desirable except when unavoidable
- The standard does not provide full definitions for errors and their handling nor defines the program behavior when encountering errors such as inappropriate data types. Since only limited language validation checks are done, such errors can only be detected during run time
- Conversion from IL to other languages is difficult and can be done only under certain conditions. Conversion from other language to IL is relatively easier but by no means straightforward

Sequential Function Chart (SFC)

This chapter outlines the basics of Sequential Function Chart, which is one of the three graphical languages defined in IEC-1131-3.

Objectives

On completing the study of this chapter, you will learn about:

- The basic concepts of Sequential Function Chart language
- Structure of the language and its main features
- Steps and transitions
- Defining sequences using other IEC languages
- Execution of actions within sequential function charts
- Ensuring design safety while using sequential function charts to represent complex control systems
- Top down design of a PLC control system using sequential function charts

8.1 Introduction to the basic concept of Sequential Function Chart (SFC)

The **Sequential Function Chart** (SFC) language has its origin in Petri-net, a methodology used for representing the flow of computer applications. The states and transitions were represented in this system using circles and arcs. A graphical language called Grafnet for representation of industrial process control systems using a similar methodology was developed based on a French national standard. This language was supported by most of the European manufacturers of early PLC systems. Subsequently, IEC standard 848:1988 for preparation of function charts for control systems was evolved based on Grafnet language.

IEC-1131-3 has adopted many definitions relating to sequential function chart from IEC 848 and the enhancements to SFC methodology over the earlier standards are primarily aimed at integrating it with the other languages of IEC-1131-3.

The SFC representation indicates the following in respect of an industrial process or machinery:

- Main phases of the process
- In the case of machinery, the main states.
- Behavior of action blocks pertaining to each phase or state
- Conditions for change from one phase to the next phase

Any process has certain well-defined phases or stages occurring in a predetermined sequence. Each stage has certain associated functions. The representation of the sequence in which the functions are performed stage-wise is the essence sequential function chart. The SFC is represented as a sequence of **steps**, with each step representing a phase of the process being controlled.

The next important aspect is the progression from one phase to the next. This change is determined based on certain conditions. These conditions which decide when a process progresses from one step to the next can either be time based or triggered a combinatory logic being fulfilled. In SFC parlance, this is called a **transition**.

In some cases, it may be necessary to skip some steps of the process or branch into one or more of several alternative sequences based on certain process conditions or dynamic parameters of the process. SFC provides methods to handle such special situations using **divergent paths**. These divergent paths usually combine at some later stage of the process and this is known as **convergence** of sequences.

We will now go into the details of these aspects.

8.1.1 Structure of SFC

As we saw in the earlier paragraph, an SFC consists of a series of steps, with each step representing a stage of a process. The change over from one step to the next is decided by transitions, which follow a step. It is to be stressed here that a SFC need not necessarily show an entire process. A complex process can be split into several SFCs each representing a part of the process. A Function block can itself be defined using a SFC and can be integrated into a larger POU. As in the case of other graphical IEC languages such as the FBD or the LD, the Sequential Function Chart can also be represented in full graphic or semi-graphic form. The semi graphic representation is illustrated (using a part of a SFC) in figure 8.1.

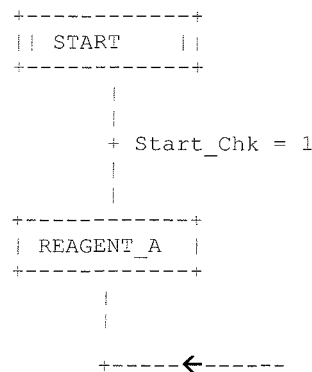


Figure 8.1

Example of SFC in semi graphic representation

Most current day implementations favor the full graphic format, which will be used in all the ensuing examples.

Refer to figure 8.2, which shows an example of the SFC for a simple process sequence already described in figure 1.4 (Chapter 1).

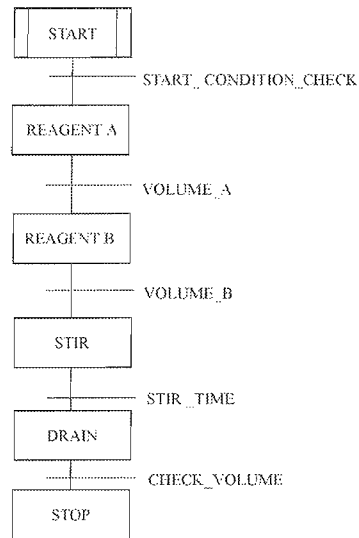


Figure 8.2

Example of SFC for a process sequence

In this SFC, it may be noted that there are several rectangular boxes, each joined to the next by a vertical line. There is a horizontal line across each of the vertical lines. The rectangular boxes are the process **steps**. Each step performs a certain set of actions. The vertical lines between the steps indicate the sequences in which the steps are to be executed for control of the process. These are the **sequence lines**. The SFC convention is that the execution sequence of steps is generally from top to bottom. This is the default sequence. When the sequence is in any other manner, an arrow will be placed in the sequence line indicating the progression sequence.

The horizontal lines marked across each sequence line are the **transitions**, which decide when exactly the process should move to the next step. The conditions of each transition are also shown in the SFC for clarity. The first step of any SFC should be the step designated as START, which as you may note, is represented by a rectangle two vertical lines on each side. This is the step that will be executed at the cold start of a PLC.

Another important convention of the SFC is that, there can normally be only one active step in a sequence. When a transition condition becomes TRUE, the previous step, which was active, is turned off and the next step after the transition is turned on. The flow of active steps is referred to as **sequence evolution**. A transition is usually described on the SFC itself using any of the IEC programming languages. In complex cases, the standard permits **named transitions**. Such named transitions can be represented separately in an associated diagram.

In the above example, the process is terminated by a STOP step, which means that the process terminates once the described sequence is completed. This may not however, always be the case. Depending on the requirements of a given process, the same sequence of steps may have to be repeated over and over again. Or the repetition will depend on the position of a certain selector switch, called, say, as REPEAT. If switch REPEAT is on, the sequence will be automatically repeated. Otherwise it will proceed to the step STOP and will get terminated.

Refer to figure 8.3 for a repeating sequence of the same process shown in the earlier case. Note the arrow in the branching sequence line, which indicates the direction in which the sequence to be followed.

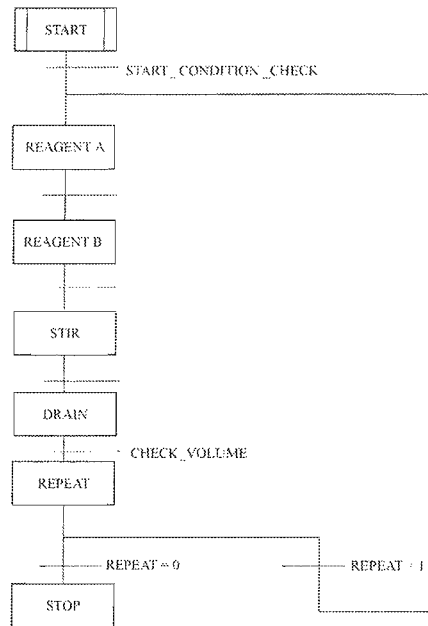


Figure 8.3

Example of SFC for a repetitive process sequence

We can consider other possible alternate sequences in this process. Let us say, that the stir interval will be extended, if necessary, by checking the pH value of the tank contents. If this value has not reached a set value, the stir step has to be repeated. Figure 8.4 represents this additional condition introduced using a divergent sequence path.

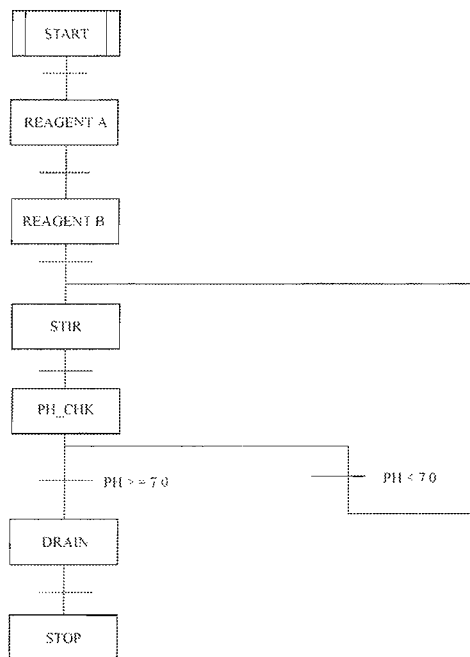


Figure 8.4

Example of SFC for a conditional branch in the sequence

In this example, the process loops back to an earlier step and repeats the sequence. In other cases, the alternative sequence may bypass a few steps and converge at a later step. An example is shown in figure 8.5, where a heating step has been added. Let us say that for a particular product mix, the contents of the tank need to be heated to temperature T whereas for another product, this can be skipped. Depending on the desired product, the required step will be activated.

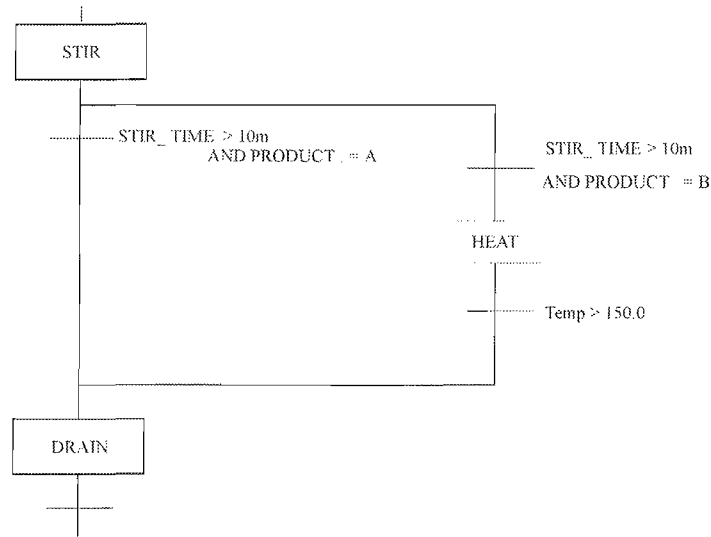
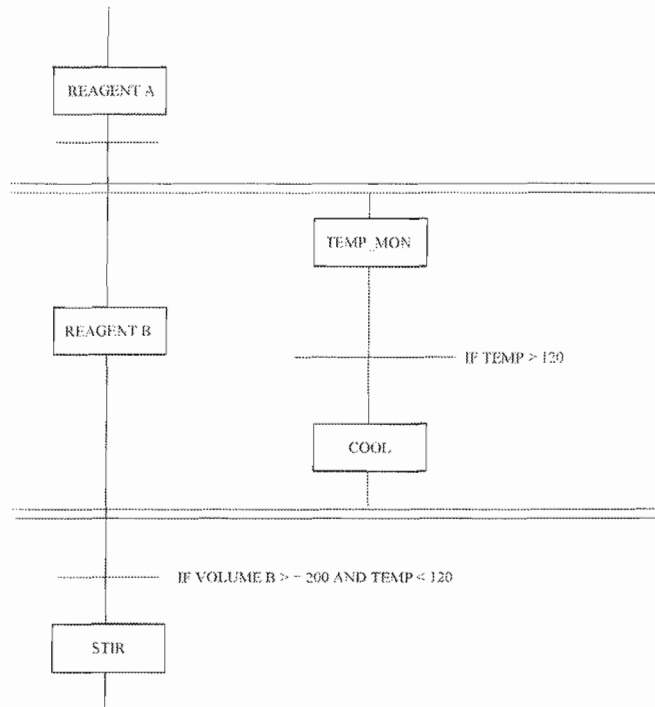


Figure 8.5

Another example of SFC for a conditional branch in the sequence

We may note here that each branch has a transition associated with it. This is usually the case with processes having divergent paths. In such cases, each transition is evaluated in sequence from left to right. The first transition in the sequence, which on evaluation permits the progression to the next step, becomes the controlling one. The standard also provides methods to change the evaluation sequence.

In the examples shown in figures 8.3, 8.4 and 8.5, only one of the branch sequences can be executed at a given instant. But in certain processes, it may be necessary that several parallel sequences have to be operated simultaneously. For example, when a process is in progress various parameters such as temperature may have to be monitored constantly and the process may have to be shut down if the values reach dangerous limits. Or an alarm has to be raised to enable the operator to take some manual tuning action. These actions have to go on simultaneously along with the main process. The standard provides a way of achieving this requirement using **simultaneous sequence divergence**. The SFC indicates simultaneous divergence using a double horizontal line. Refer to the example in figure 8.6.

**Figure 8.6**

An example of SFC with simultaneous sequence divergence

The sequences, which are operating simultaneously after divergence, may once again come back to a common main sequence later through a **simultaneous sequence convergence**. In the example shown in figure 8.6, such a convergence is also represented. It should be noted that the transition after the simultaneous convergence is evaluated only when all steps in the different paths preceding it are active. Then, on successful evaluation of the transition, the execution moves to the next step, before which the previous steps in all the different simultaneous sequence paths are made inactive.

8.2 Steps

We learnt about the use of steps in the previous section. There are essentially two types of steps.

8.2.1 Initial step

The START step in the SFC is an initial step represented by a rectangle with double vertical lines. An SFC can have only one such initial step and this is the step which will be activated first when a PLC is started,

8.2.2 Normal step

All the other steps in the SFC are normal steps and are represented by plain rectangles.

Each step should be given a unique name, which is written in the center of the rectangle. Within a given SFC, the name of a step cannot be repeated. The actions to be taken when a step becomes active are defined using one or more action blocks. The action blocks can be represented in any of the IEC languages, viz., ST, IL, LD or FBD. A complete description of action blocks will be given in a subsequent section.

A step has two associated variables, which can be accessed from any point in the SFC. The first is the **Step active flag**. This flag is set when the step is active. The variable associated with this flag is named as **<Step name>. X** and its status can be tested from any point. The variable will be set to TRUE when the step is active. This is useful to also set the status of a related variable. See the figure 8.7 below.

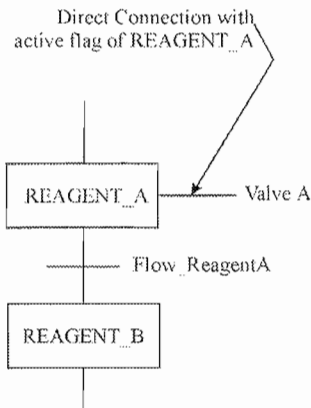


Figure. 8.7
Use of active flag of an SFC step

In this Figure, the step ReagentA denotes that valve A should be kept open till the transition condition to the next step is fulfilled. By associating the variable ValveA with the active flag ReagentA.X, it can be ensured that the valve A is held in open position till the step ReagentA is active.

The second variable associated with a step is the **elapsed time** (.T). This variable is designated by the name **<Step Name>. T** and is of data type TIME. When a step becomes active, the flag is set to 0 and then starts counting the elapsed time. This can be used directly in the evaluation of the subsequent transition, in case the step is meant to be active for a pre-determined duration. This variable value is retained even after the step has ceased to be active and is reset when the step becomes active once again in a later cycle. The example shown in figure 8.8 illustrates this principle.

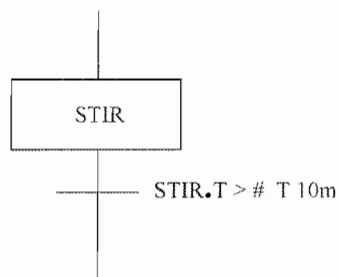


Figure 8.8
Use of elapsed time variable of an SFC step

In this example, the step STIR has to be performed for a set value of time. The variable STIR.T enables the next transition to check whether the required interval has elapsed. On completion of a certain elapsed period, the process moves to the next step Drain. The elapsed time variable can also be used for program diagnostic purposes to ensure that the steps are active for the periods they are meant to be.

8.3 Transitions

We have learnt earlier in this chapter that transitions are used to determine when the process should move from one active step to another in the sequence. Transition conditions can be expressed using any of the other IEC standard languages. In the examples, which we have seen so far, the transitions have been represented on the SFC, using structured text. But, the same can be represented using Ladder diagram or Function block diagram as well.

In the case of structured text, it takes the form of an expression. If the expression on evaluation returns the value TRUE, the transition takes place. (Refer figure 8.9A). The statement for transition from the step ReagentA is conditional on the total flow of the reagent A having reached a set value given by the expression:

Flow_ReagentA => 980

When the flow has totaled to 980 (Kilo Liters) the above expression becomes TRUE and the transition to the next step is initiated.

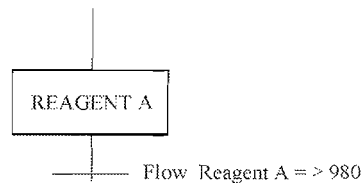


Figure 8.9A

Use of ST expression for transition

The same condition can also be used in the form of a Ladder diagram as shown in figure. 8.9B.

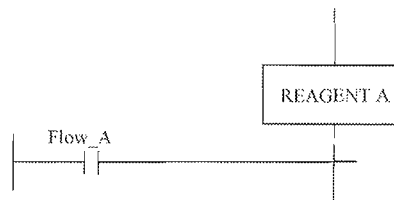


Figure. 8.9B

Use of LD rung for transition

In this case the transition replaces the coil ---()---. When the contacts given in the rung establish power flow from the left power rail to the transition line, transition is triggered.

A transition condition can also be in the form of an FBD as shown in figure 8.9C below.

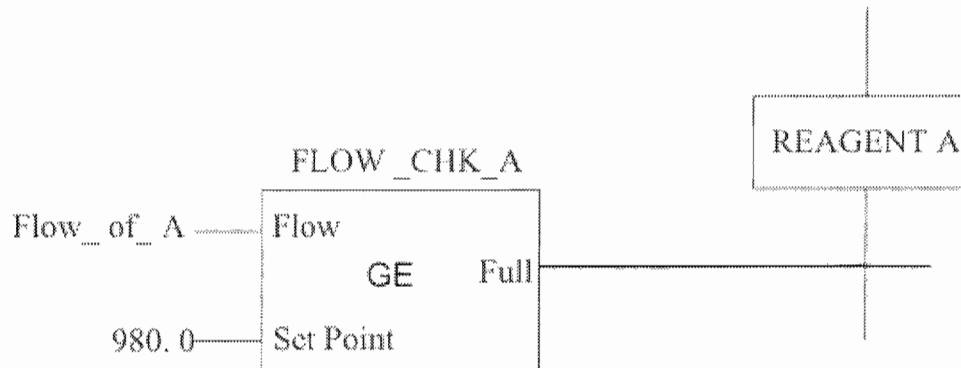


Figure 8.9C
Use of FBD for transition

In this example a function block instance called Flow_Chk_A compares the flow against a set value and once the set value is reached, it makes the Boolean output Flow_Limit TRUE. This triggers transition to the next step in the sequence.

A transition connector can be used if the transition logic cannot be accommodated in the same line. The example in figure 8.9D shows such a case.

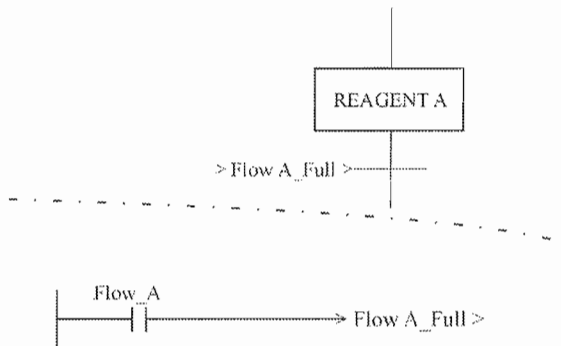
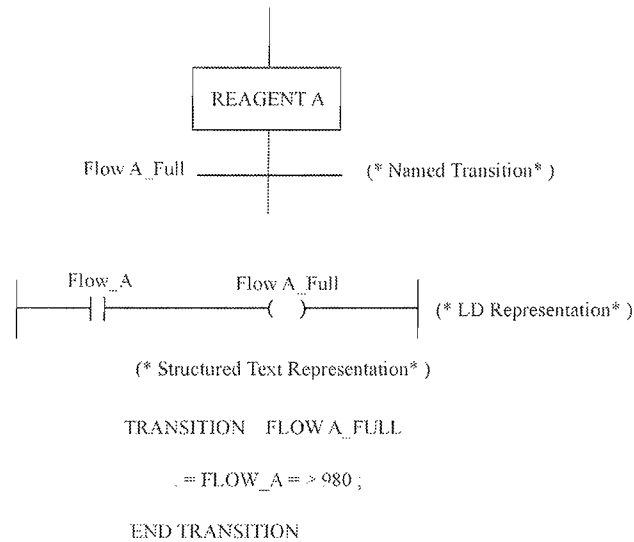


Figure 8.9D
Use of transition connector

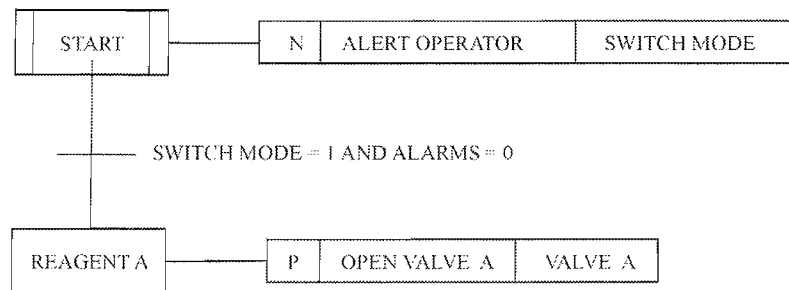
Transitions can also be identified by names and using this name, the transition can be defined in any page of the SFC using any of the IEC languages. Figure 8.10 shows examples of such named transitions.

**Figure 8.10**

Defining a named transition using IEC languages

8.4 Actions

As we saw earlier each step of an SFC represents a particular state of a machine or a process. When a set of conditions is fulfilled, a transition occurs and a sequence of actions is initiated to achieve the process condition defined for the concerned step. In the SFC graphic representation, this is shown as a rectangular box within which the actions corresponding to the step are indicated using any of the IEC standard languages. An example is shown in figure 8.11 below.

**Figure 8.11**

Representation of SFC actions

The rectangular box contains a qualifier, the name of the action, which should be unique within the SFC and an indicator variable where applicable. The qualifier defines when an action is to be initiated. The list of qualifiers normally used in SFCs is shown in the table in figure 8.12 below.

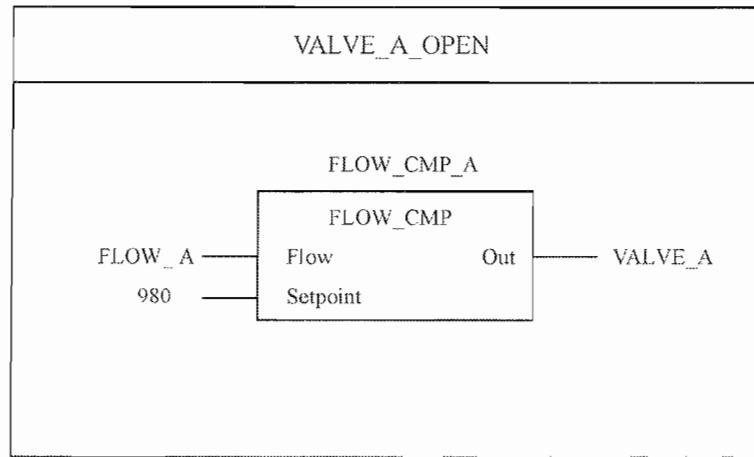
Qualifier	Description
None	Same as N
N	Executes when the step is active
S	Sets an action active (stored)
R	Resets a stored action
L	Terminates after a given period
D	Time delayed, starting after a given period
P	Pulse action, executes once when the step is activated
SD	Stored and time delayed, an action which is set active after a delay and executing even though the associated step may get deactivated before the time delay elapses
DS	Delayed and stored. In this case if the step is deactivated before the time delay elapses, the action is NOT stored.
SL	Stored and time limited. The action is started and executes for the set period.

Figure 8.12
Action qualifiers

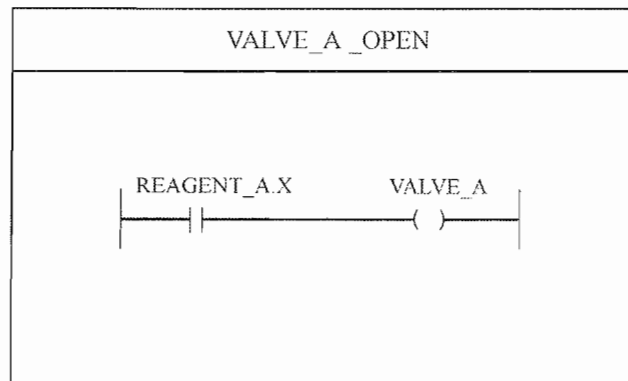
An action can be represented either within the SFC or if it is too complex, in another page or diagram. When an action is thus represented in another part of the diagram, its name is used to identify it. An action may be used in more than one step if it gets repeated in different parts of a process. Sometimes a step has no actions and simply waits for the next transition. In that case, it is said to be performing a 'wait' function.

The indicator variable is used for annotation purposes and its use is optional. It usually represents the key variable, which indicates that an action has been completed.

As in the case of transitions, actions too can be described using any of the IEC languages.



(a) Action described using FBD



(b) Action described using LD

Figure 8.13

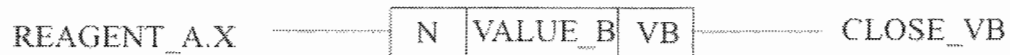
Representation of SFC actions using different IEC languages

An action in SFC format can also be used within Program organization units written using any of the other IEC languages. Figure. 8-14 shows an example of an action within a LD program. An action is initiated when there is a power flow into the action block. Action indicator variable is used to signal the completion of this action.


Figure 8.14

Use of action block within a LD rung

An action used within a function block diagram is shown in figure. 8.15.


Figure 8.15

Use of action block within a FBD

8.5 Action qualifiers

We saw in the last section the various action qualifiers used in action blocks. Qualifiers are used to define when exactly during a step an action should be initiated. Some of the actions are of stored type, which, once initiated continue till reset. Such actions may span across one or more steps and their operation requires rather careful scrutiny. Timing diagrams can be used to study the operation of actions with different qualifiers. With stored actions, a reset command is necessary, as otherwise the action will continue indefinitely. Figures 8.16 to 8.24 illustrate various action qualifiers using timing diagrams.

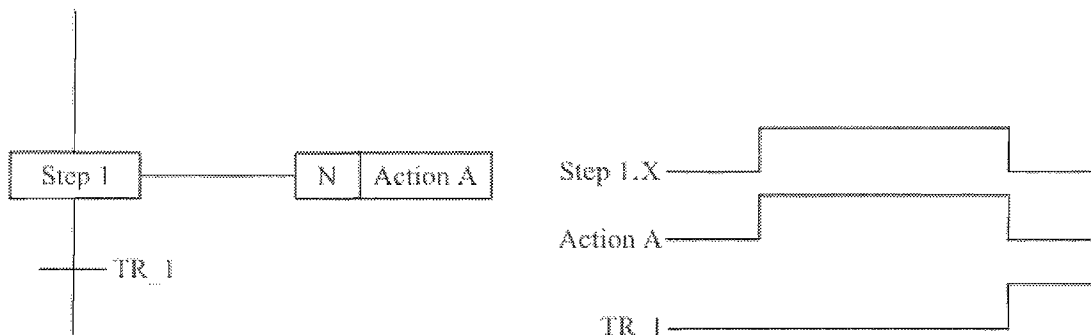


Figure 8.16
Use of action qualifier-N

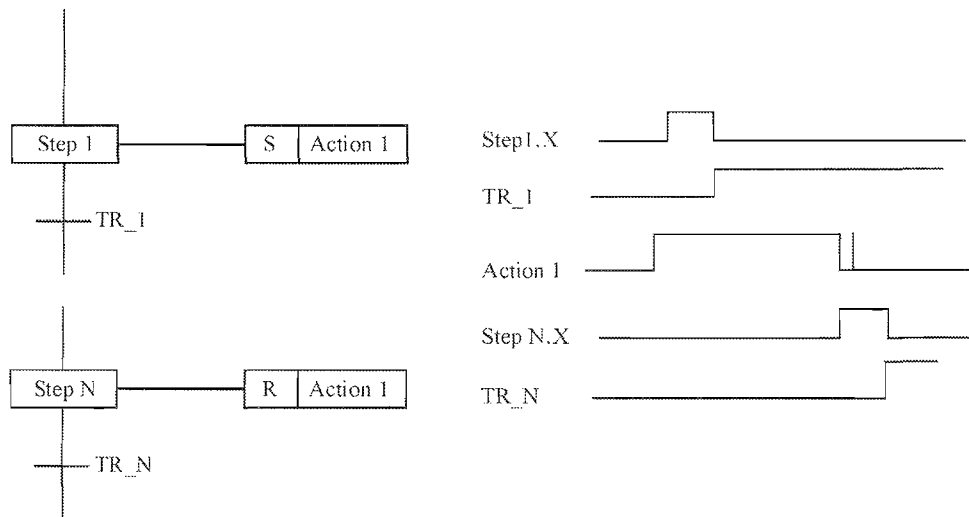


Figure 8.17
Use of action qualifiers-S and R

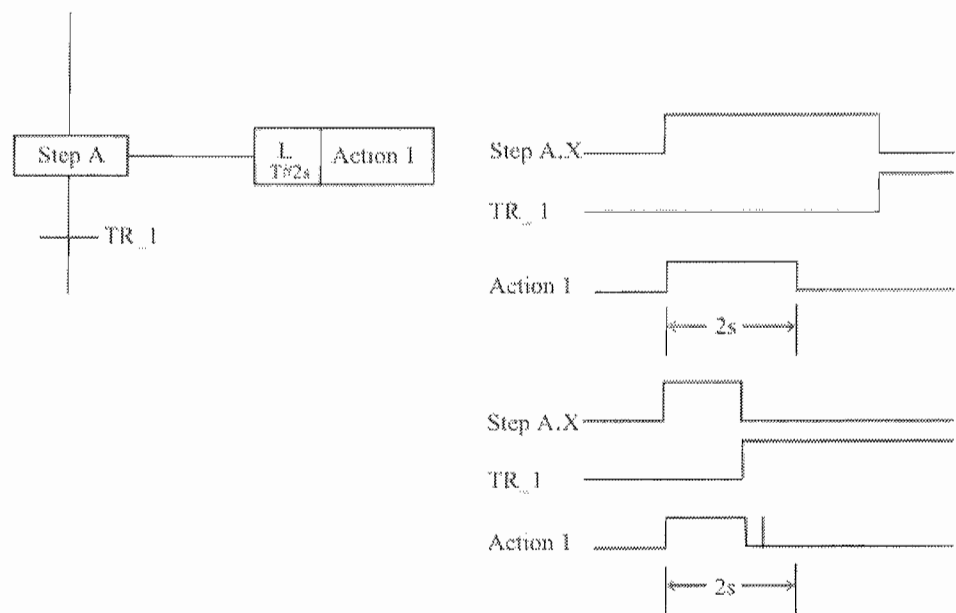


Figure 8.18
Use of action qualifier-L

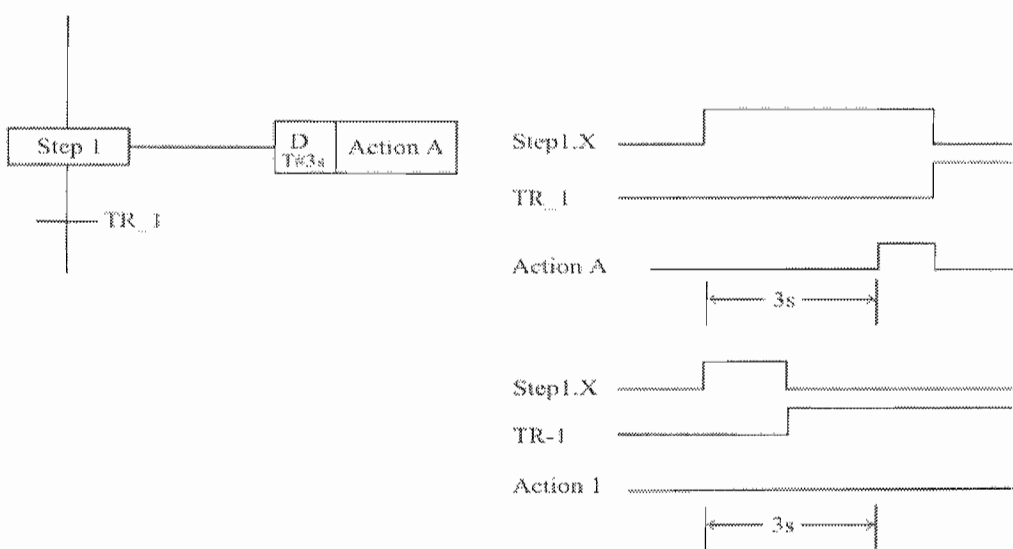


Figure 8.19
Use of action qualifier-D

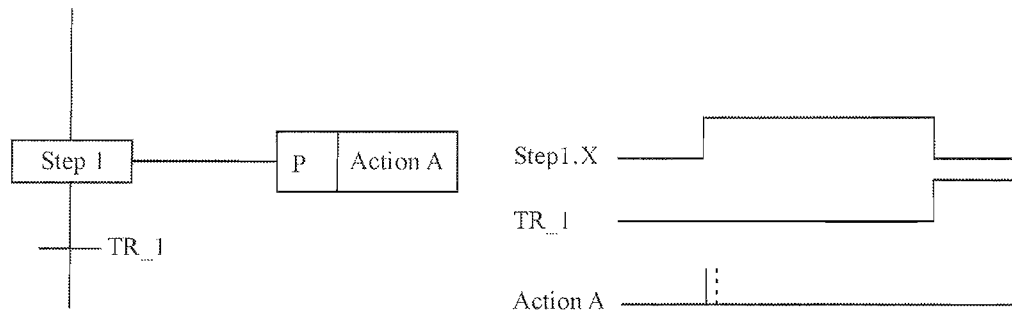


Figure 8.20
Use of action qualifier-P

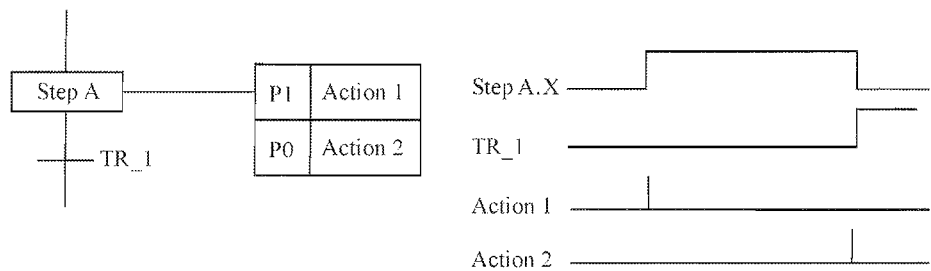


Figure 8.21
Use of action qualifiers-P0 and P1

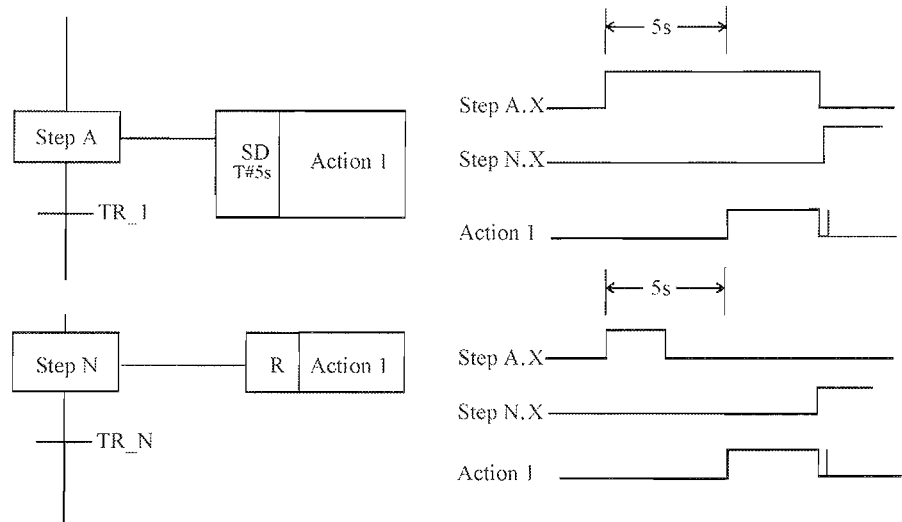


Figure 8.22
Use of action qualifiers-SD and R

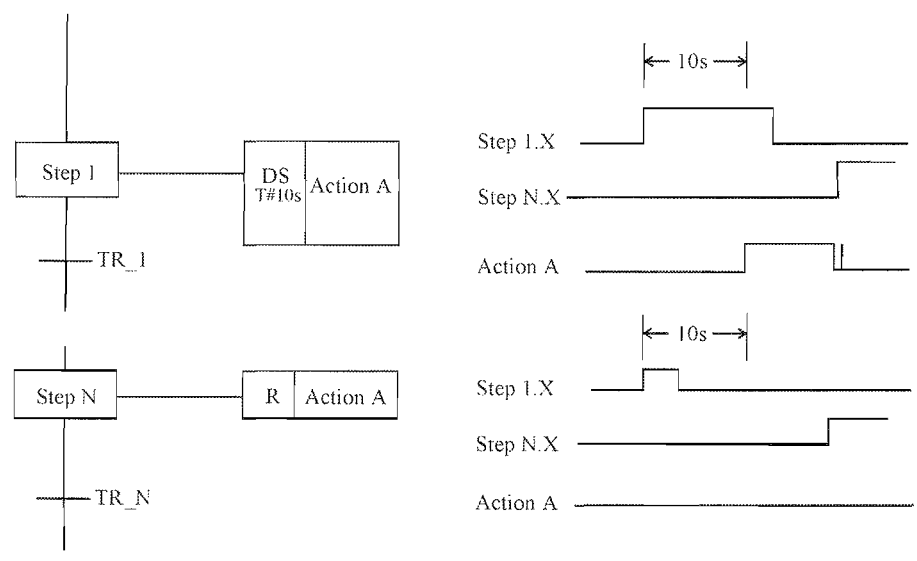


Figure 8.23
Use of action qualifiers-DS and R

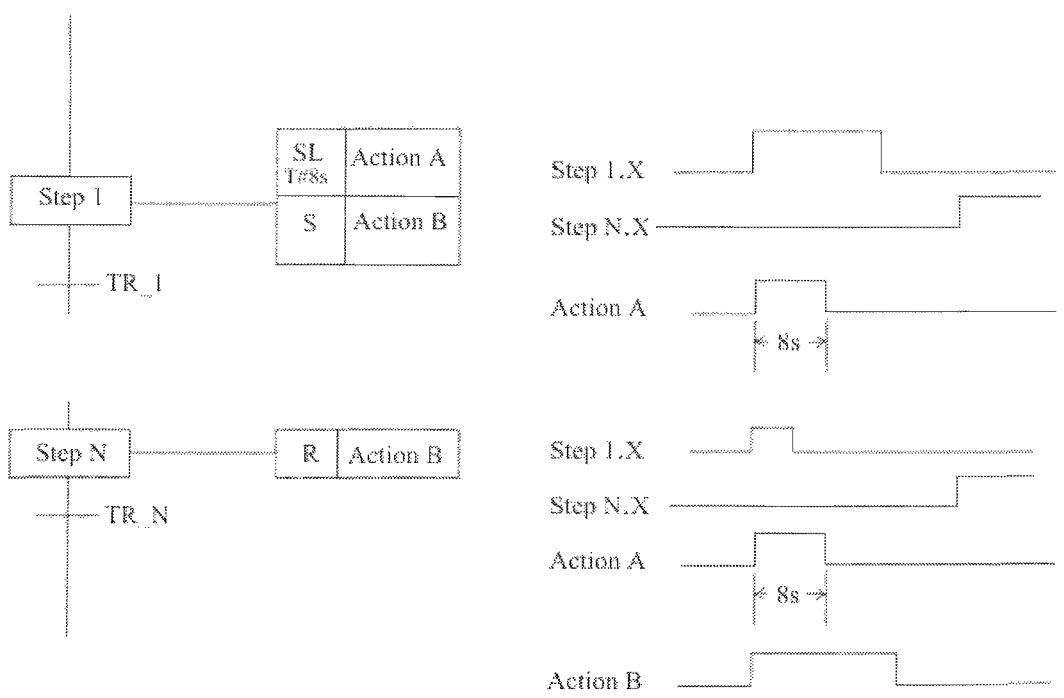


Figure 8.24
Use of action qualifiers-SL and R

It may be seen that in most of the above Figures, there is a small spike in the action status towards the end. The reason is that the actions associated with a step are executed for one last time after the step is deactivated.

8.6 Action control function block

The standard defines a hypothetical function block representing an action control function. Each action qualifier is represented by an instance of this function block. . The output Q of this function block results in the execution of an action when its value becomes TRUE.

8.7 Execution rules

SFC is used to describe the functioning of a POU and its execution is controlled by a task. Generally POU's are executed periodically, say, once in 500 milliseconds. An SFC is evaluated during each execution. The rules for evaluation are as follows:

- The initial step is the first step to be evaluated after system initialization. Actions associated with this step are executed first
- At each execution, the status of active steps is first evaluated and all transitions following these active steps are evaluated
- Actions, which ceased in the last evaluation, are executed one last time
- All actions associated with active steps are executed
- Steps that precede transitions, which are evaluated as true in the current cycle, are deactivated and succeeding steps are activated
- Actions associated with deactivated steps are marked for execution for one last time in the next evaluation

Some of the aspects to be noted in designing an SFC are:

- Two steps cannot be directly linked. There must be a transition between them
- Similarly, two transitions must be separated by a step
- A transition from one step may lead to two or more steps. These steps are executed simultaneously. The associated sequences execute independently thereafter
- The evaluation of a transition condition and (if it is TRUE) the deactivation of the preceding step and activation of the succeeding step are all actions that occur with no delay
- If several different transition conditions become TRUE in any given evaluation cycle, the deactivation and activation of associated steps happen with no delay
- If a step is activated, the actions associated with the step are executed first even though the transition after this step may be TRUE already

8.8 Design safety issues

As we saw in the earlier section on transitions, it is possible to have two or more branches from a step with each branch having its own transition. Even if more than one transition becomes TRUE at the same time, only one branch is selected for execution. By default, the leftmost transition, which is TRUE, is the one that is selected. However, the user can define the transition priority by numbering the transitions. The transition having the lowest number has the highest priority. An asterisk placed at the center of the branch line indicates user-defined priority. Figure 8.25 illustrates this action.

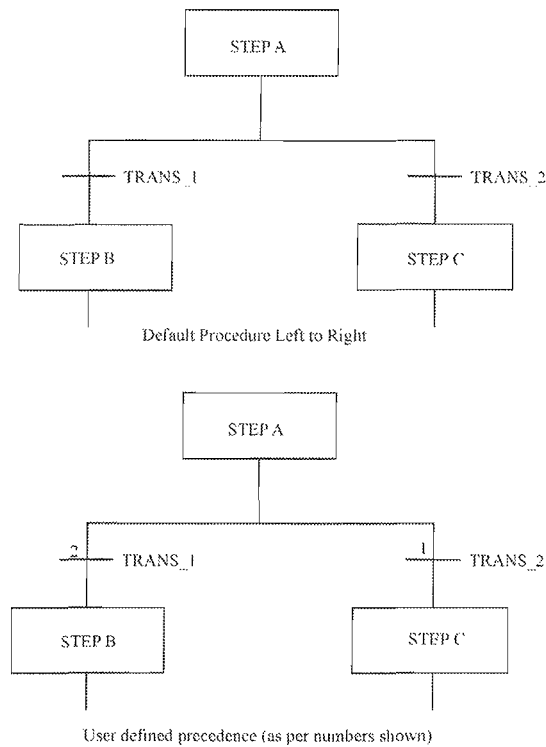


Figure 8.25
Transition priority

Wherever possible, the programmer must ensure that the divergent paths are mutually exclusive by introducing suitable transition logic condition.

Branching can be used to skip some of the steps as well as loop back to an earlier step. Examples of such divergence were discussed in earlier sections in this chapter.

Simultaneous convergence and divergence are used to ensure parallel operation of more than one sequence. In the case of a transition following a simultaneous convergence, all the steps that converge simultaneously must be active before the succeeding transition is evaluated. Figure 8.26 shows an example.

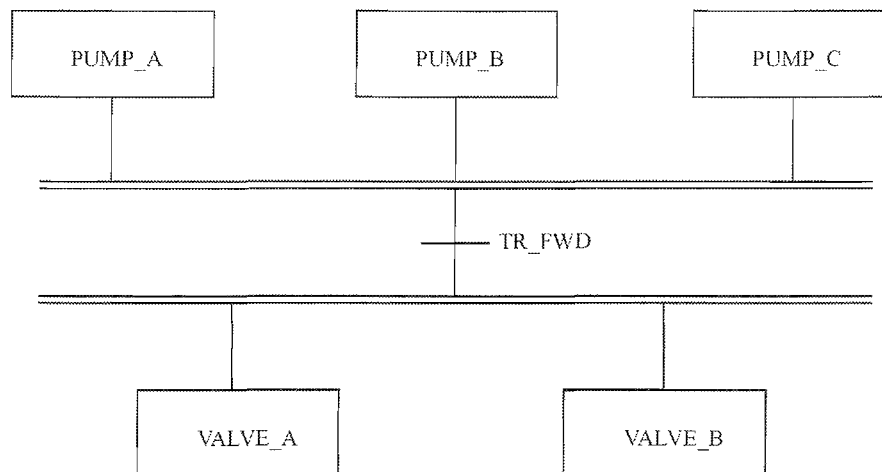


Figure 8.26
Simultaneous convergence and divergence

In this example, steps PUMP_A, PUMP_B and PUMP_C must be active before transition 1 is tested. Under this condition, if Transition TR_FWD becomes TRUE, steps PUMP_A, PUMP_B and PUMP_C are deactivated and steps VALVE_A and VALVE_B are activated. All these happen simultaneously. An SFC program must avoid conditions, which will lead to unsafe situations. For example, two converging steps may be mutually exclusive by virtue of the preceding transition conditions being so. Under such condition convergence will never occur and the sequence will therefore not proceed beyond this point. Many compilers can detect such unsafe SFC constructs and alert the programmer.

8.9 Top down design

An SFC is primarily used for top down design of a complex process control system. Any process consists of a number of states. The major steps are identified and represented in the form of an SFC. The conditions that cause a change of state are identified and represented as transitions. The actions to be taken in each step are represented as action blocks. Thus the entire process sequence can be shown in a SFC. In a complex process each step may itself consist of a number of minor states, which can be too cumbersome to be shown using action blocks alone. In such cases, each major step can be represented as another sequential function chart, with all minor steps represented as steps of this SFC with the sequence represented using transitions.

Thus an SFC can be arranged hierarchically into smaller sequences each with its SFC with all of them being combined in a overall SFC to simplify understanding.

Many implementations support the above concept and refer to this feature as macro step (which is not defined in the standard, though). The macro step is the name given to a step, which encapsulates an SFC within the step.

To avoid difficulties in interpreting such complex layered SFC systems, certain programming aspects should be noted.

- Meaningful names must be given to steps, transitions and actions so that the purpose of their use is clear from the name itself
- Names should be unique within a POU
- An SFC should be kept small. Hierarchical design should be used to represent complex systems by depicting action blocks in the form of lower level SFCs
- Simultaneous sequences should not interact with each other as far as possible. For example, steps in different sequences should not change the same variable
- The top level SFC must make sure that any halted state within a sub sequence is properly closed on completion of the sub sequence. Such problems can arise from plant trip condition and must be handled correctly by the main SFC

8.10 Summary

Sequential function charts provide a flexible and hierarchical method of representing complex control processes graphically. The main features of this method are:

- Alternative sequences in a process can be represented using divergent paths.
- Simultaneous divergence of different sequences is provided in SFCs
- The elements of SFC such as steps and transitions can be represented using any of the graphical languages defined in the standard

- Action qualifiers are used to define the manner in which actions are sequenced
- Though SFC is generally used for representing large control systems, they can also be used to represent low level actions in a part of a sequence such as a function block, which occur within a POU

PROGRAMMING OF INDUSTRIAL CONTROL

TYPICAL EXAMPLES PRESENTED AS PROBLEMS

PREAMBLE

- The examples given below represent a cross section of real life situations encountered in industrial processes. Participants will use IsaGRAF PLC programming software to model their solutions and validate them using the simulation tools provided.
- It is advised that the same problem be worked out using different languages applicable in each case so that their comparative merits can be understood. Overall framework of the solution may be preferably done using a Sequential Function Chart.
- Though the software includes Flow Chart as one of the techniques, this method may not be used as it is not a part of the IEC-1131-3 standard

PROBLEM-1: MOTOR START

A Motor is started by an ON push button (pulse type contact) and stopped by an OFF push button (pulse type contact) through an output coil (Coil M). Motor Trips if there is a fault (contact F is TRUE) and cannot be started till it is reset (becomes FALSE). Conditions to be taken care of are as follows:

1. When push buttons ON and OFF give their pulse inputs (assumed for a duration of 500 m. secs), the action required (START or STOP) should take place within 3 seconds. Otherwise an alarm should initiate (coil A1). The condition START and STOP is sensed by the contact of motor power contactor (K). When motor runs contact K is TRUE else it is FALSE.

Note: Such a situation can arise when there is some failure in the motor control circuit which is beyond the PLC's scope.

2. An alarm A2 should initiate when F becomes TRUE and the motor trips as a result.
3. An alarm A3 should initiate if F becomes TRUE and the motor continues to run beyond 5 Sec.
4. Reset button for releasing alarms.

PROBLEM-2: MAGNETIC SEPARATOR

A magnetic separator is used to attract scrap iron pieces that may be mixed up with bulk materials and cause damage to conveying or crushing equipment.

A metal detector detects a iron piece on a conveyor (represented by a push-button type momentary contact MD). A magnet (Coil M) suspended over the conveyor at a point downstream is energised. After a delay sufficient for the metal piece to reach the point where the magnet is located and be removed by it, the travelling monorail from which the magnet is suspended moves away (coil H1) from the conveyor. The trolley is stopped by the forward limit switch LSF when it is over a scrap chute. The magnet is disconnected from supply to allow the metal pieces to fall into the chute and the monorail takes it back (coil H2) to the original position. It is stopped by limit switch LSB when it reaches the position above the conveyor.

If at any point before the trolley starts moving away from the conveyor, another piece is detected, the magnet continues in its position till the piece is removed from the conveyor.

Two timers are to be used. One is for timing the detection of the metal piece to the start of magnet moving away (T1). Another is for the magnet to be de-energised over the chute (T2). Once a new piece is detected, the timers start from zero time again.

Assumption is that the time between detection of metal and its being attracted is more than the complete travel cycle time of the trolley (both ways).

PROBLEM 3: OVERHEAD TANK AND LOW LEVEL SUMP

An overhead tank is filled up by a pump from a low level sump. The conditions of operation are:

1. The pump starts when the tank level becomes low (TLL).
2. The pump stops when the tank gets filled up (TLH).
3. The pump stops if the sump level becomes too low (SLL).
4. The pump does not start if sump level is too low.
5. The condition of tank level being low and pump not running will give an alarm A1 after a wait time of 10 seconds.
6. Provide a Reset button to start the pump again.
7. Stop button for stopping the sequence.
8. Reset button for releasing alarms.

PROBLEM 4: PRESSURE SWITCH OPERATED COMPRESSORS

Two compressors feed compressed air into a air-receiver. The start and stop of the compressors is to be automatic through two LOW pressure switches L1 and L2 and one NORMAL pressure switch N1. L1 picks up (TRUE) at a higher pressure value compared to L2. N1 picks up at higher pressure than L1.

Low pressure switches go True when pressure < set value; while normal pressure switch goes True when pressure > set value.

1. Compressor C1 will start if switch L1 picks up (TRUE).
2. In addition Compressor C2 will start if L2 picks up (TRUE).
3. C2 will stop if L1 becomes FALSE.
4. C1 stops if switch N1 picks up (TRUE).
5. An alarm A1 should initiate if L1 picks up and C1 does not start within 5 seconds.
6. An alarm A2 should initiate if L2 picks up and C2 does not start within 5 seconds.
7. An alarm A3 should initiate if N1 remains FALSE for more than 15 Minutes.
8. Reset button for releasing alarms.

PROBLEM 5: PRESSURE SIGNAL OPERATED COMPRESSORS

This problem is similar to that of the previous one except that L1, L2 and N1 are replaced by a pressure transducer. The conditions of operation are as follows.

1. If pressure < 35 ATG for more than 5 secs will cause starting of C1.
2. If pressure < 25 ATG for more than 5 secs will cause starting of C2.
3. C2 will stop if pressure > 35 ATG.
4. C1 will stop if pressure > 40 ATG.
5. An alarm A1 should initiate if Pressure < 35 ATG and C1 does not start within 10 seconds.
6. An alarm A2 should initiate if Pressure < 25 ATG and C2 does not start within 10 seconds.
7. An alarm A3 should initiate if Pressure remains < 35 ATG for more than 15 Minutes.
8. Reset button for releasing alarms.

Note: ATG represents atmospheres gauge

PROBLEM 6: NEUTRALISING TANK OPERATION

This is a typical batch mode operation of an effluent neutralizing system. The simplified sequence of this process is as follows.

1. At the start command of the system, the effluent is transferred from a storage reservoir to a neutralizing tank using a transfer pump.
2. Pump will stop when tank level HIGH relay picks up.
3. A stirrer then starts and operates for a period of 5 minutes.
4. A pH measurement probe draws a sample and an output pH value is obtained.
5. Based on the value, the following are decided.
 - Neutralizing agent to be used. (If $\text{pH} < 7$ agent A else agent B). For agent A, valve A will be opened and for agent B, valve B will be opened.
6. Valve A or B to open.
7. Close valve A or B when the pH reaches value of 7.
8. Stirrer operates for a period of 10 minutes
9. Open Drain valve of neutralizing tank.
10. The pump will start when tank level LOW relay picks up.
11. Repeat the cycle from step 1.
12. Start button to start the sequence.
13. Stop button to stop the sequence.



IDC Technologies – Technology Training That Works



Technology Training that Works

Pre-Workshop Questionnaire

Industrial Programming using IEC 61131-3

Full Name _____

City/Country _____ Date _____

Would you kindly answer the questions below.

Please answer all questions to the best of your ability.

1. What are the two main reasons you attending this training workshop?

2. Briefly describe your main responsibilities in your current job?

3. Where did you hear about this workshop?

<input type="checkbox"/>	IDC Technologies Brochure	<input type="checkbox"/>	Web Site
<input type="checkbox"/>	Colleague	<input type="checkbox"/>	Other _____

4. Have you been on a previous IDC Technologies workshop? ☐ Yes ☐ No

5. In which area do you work?

<input type="checkbox"/>	Trades	<input type="checkbox"/>	Manager
<input type="checkbox"/>	Technician	<input type="checkbox"/>	IT
<input type="checkbox"/>	Technologist	<input type="checkbox"/>	Engineer
<input type="checkbox"/>	Other _____		

6. What size is your organisation?

<input type="checkbox"/>	less than 50 people	<input type="checkbox"/>	More than 100 People
<input type="checkbox"/>	Between 50 and 100 people		



Technology Training that Works

Technical Questions

1. What are the different "languages" that make up the IEC 61131-3 standard ?

2. What are the main (or key) features of the IEC 61131-3 standard ?

3. Why does the IEC 61131-3 standard have resources ? What does a resource mean ?

4. Will IEC 61131-3 languages result in applications that run slower and require more memory than using a simple ladderlogic program ?

5. Is it possible to transfer IEC 61131-3 programs from one vendor's PLC to another ?



Technology Training that Works

Post Course Questionnaire

Workshop Name:

Name: Company: Date:

*To help us improve the quality of future technical workshops your honest and frank comments will provide us with valuable feedback.
Please complete the following:*

How would you rate the following?

Please place a cross (x) in the appropriate column and make any comments below.		Poor			Average			Excellent				
		0	1	2	3	4	5	6	7	8	9	10
1.	Subject matter presented											
2.	Practical demonstrations											
3.	Materials provided (training manual, software)											
4.	Overhead slides											
5.	Venue											
6.	Instructor											
7.	How well did the workshop meet your expectations?											
8.	Other, please specify.....											

Miscellaneous

Which section/s of the workshop did you feel was the most valuable?

Was there a section of the workshop that you would like us to remove/modify?

Based on your experience today, would you attend another IDC Technologies workshop?

Yes ☐ No ☐

If no, please give your reason

Do you have any comments either about the workshop or the instructor that you would like to share with us?

Do we have your permission to use your comments in our marketing?

Yes ☐ No ☐

Would you like to receive updates on new IDC Technologies workshops and technical forums?

Yes ☐ No ☐

If yes, please supply us with your email address (**please print clearly**):

and postal address:

Do you think your place of work may be interested in IDC Technologies presenting a customised in-house training workshop? Yes ☐ No ☐

If yes, please fill in the following information:

Contact Person: Position:

E-mail: Phone Number:

New IDC Workshops

We need your help. We are constantly researching and producing new technology training workshops for Engineers and Technicians to help you in your work. We would appreciate it if you would indicate which workshops below are of interest to you.

Please cross (x) the appropriate boxes.

INSTRUMENTATION AUTOMATION & PROCESS CONTROL	INFORMATION TECHNOLOGY
Practical Automation and Process Control using PLCs	Practical Web-Site Development & E-Commerce Systems for Industry
Practical Data Acquisition using Personal Computers & Standalone Systems	Industrial Network Security for SCADA, Automation, Process Control and PLC Systems
Practical On-line Analytical Instrumentation for Engineers and Technicians	
Practical Flow Measurement for Engineers and Technicians	ELECTRICAL
Practical Intrinsic Safety for Engineers and Technicians	Safe Operation and Maintenance of Circuit Breakers and Switchgear
Practical Safety Instrumentation and Shut-down Systems for Industry	Practical Power Systems Protection for Engineers and Technicians
Practical Process Control for Engineers and Technicians	Practical High Voltage Safety Operating Procedures
Practical Industrial Programming using 61131-3 for PLCs	Practical Solutions to Power Quality Problems for Engineers and Technicians
Practical SCADA Systems for Industry	Wind & Solar Power – Renewable Energy Technologies
Fundamentals of OPC (OLE for Process Control)	Practical Power Distribution
Practical Instrumentation for Automation and Process Control	Practical Variable Speed Drives for Instrumentation and Control Systems
Practical Motion Control for Engineers and Technicians	
Practical HAZOPS, Trips and Alarms	ELECTRONICS
	Practical Digital Signal Processing Systems for Engineers and Technicians
DATA COMMUNICATIONS & NETWORKING	Shielding, EMC/EMI, Noise Reduction, Earthing and Circuit Board Layout
Practical Data Communications for Engineers and Technicians	Practical EMC and EMI Control for Engineers and Technicians
Practical DNP3, 60870.5 & Modern SCADA Communication Systems	
Practical FieldBus and Device Networks for Engineers and Technicians	MECHANICAL ENGINEERING
Troubleshooting & Problem Solving of Industrial Data Communications	Fundamentals of Heating, Ventilation & Airconditioning (HVAC) for Engineers and Technicians
Practical Fibre Optics for Engineers and Technicians	Practical Boiler Plant Operation and Management
Practical Industrial Networking for Engineers and Technicians	Practical Centrifugal Pumps – Efficient use for Safety & Reliability
Practical TCP/IP & Ethernet Networking for Industry	
Practical Telecommunications for Engineers and Technicians	PROJECT & FINANCIAL MANAGEMENT
Best Practice in Industrial Data Communications	Practical Project Management for Engineers and Technicians
Practical Routers & Switches (including TCP/IP and Ethernet) for Engineers and Technicians	Practical Financial Management and Project Investment Analysis
Troubleshooting & Problem Solving of Ethernet Networks	Practical Specification and Technical Writing for Engineers and Other Technical People

For our full list of titles please visit: www.idc-online.com/training

If you know anyone who would benefit from attending an IDC Technologies workshop or technical forum, please fill in their contact details below..

Name:

Position:

Company Name:

Address:

Email:

Name:

Position:

Company Name:

Address:

Email:

**Thank you for completing this questionnaire,
your opinion is important to us.**

IDC TECHNOLOGIES

Worldwide Offices

AUSTRALIA

West Coast Office

982 Wellington Street, West Perth, WA 6005

PO Box 1093, West Perth, WA 6872

Telephone: (08) 9321 1702 • Facsimile: (08) 9321 2891

East Coast Office

PO Box 1750, North Sydney, NSW 2059

Telephone: (02) 9957 2706 • Facsimile: (02) 9955 4468

CANADA

402-814 Richards Street, Vancouver, BC V6B 3A7

Telephone: 1-800-324-4244 • Facsimile: 1-800-434-4045

IRELAND

PO Box 8069, Shankill Co Dublin

Telephone: (01) 473 3190 • Facsimile: (01) 473 3191

NEW ZEALAND

Parkview Towers, 28 Davies Avenue, Manukau City

PO Box 76-142, Manukau City

Telephone: (09) 263 4759 • Facsimile: (09) 262 2304

SINGAPORE

100 Eu Tong Sen Street, #04-11 Pearl's Centre, Singapore 059812

Telephone: (65) 6224 6298 • Facsimile: (65) 6224 7922

SOUTH AFRICA

17 Allendale Road, New Brighton, Sandton, 2196

PO Box 785640, Sandton 2146

Telephone: (011) 883 2859 • Facsimile: (011) 883 0610

Toll Free: 0800 114 160

UNITED KINGDOM

46 Central Road, Worcester Park, Surrey KT4 8HY

Telephone: (020) 8335 4014 • Facsimile: (020) 8335 4120

UNITED STATES

7101 Highway 71 West #200, Austin TX 78735

Telephone: 1-800-324-4244 • Facsimile: 1-800-434-4045

Web Site: www.idc-online.com

