Karl-Heinz John · Michael Tiegelkamp

IEC 61131-3: Programming Industrial Automation Systems

Concepts and Programming Languages,
Requirements for Programming Systems,
Aids to Decision-Making Tools

With 139 Figures

Springer

Dipl.-Inform. KARL-HEINZ JOHN
Irrlrinnig 13
D-91301 Forchheim
e-mail: karlheinz.john@gmx.de

Dipl.-Inform. MICHAEL TIEGELKAMP
Kurpfalzstr. 34
D-90602 Pyrbaum
e-mail: Michael.Tiegelkamp@gmx.de

Contents

1 Introduction

The rapid advances in performance and miniaturisation in microtechnology are constantly opening up new markets for the programmable logic controller (PLC). Specially designed controller hardware or PC-based controllers, extended by hardware and software with real-time capability, now control highly complex automation processes.

The different types of PLC cover a wide task spectrum - ranging from small network node computers and distributed compact units right up to modular, fault-tolerant, high-performance PLCs. They differ in performance characteristics such as processing speed, networking ability or the selection of I/O modules they support.

Throughout this book, the term PLC is used to refer to the technology as a whole, both hardware and software, and not merely to the hardware architecture.

The broad spectrum of capability of the hardware requires corresponding support from suitable programming tools, to allow low-cost, quality-conscious creation of both simple and complex software solutions. Desirable features of programming tools include:

- Simultaneous use of several PLC programming languages
- "Online" modification of programs in the PLC
- Reverse documentation of the programs from the PLC
- Reusability of PLC program blocks
- "Offline" testing and simulation of user programs
- Integrated configuring and commissioning tools
- Quality assurance, project documentation
- Use of systems with open interfaces.

Modern PCs have enabled increasingly efficient PLC programming tools to be developed in the last 10 years.

The classical PLC programming methods, such as the instruction list, ladder logic or control system function chart, which have been employed until now, have reached their limits. Users want uniform, manufacturer-independent language

concepts, high-level programming languages and development tools similar to those that have already been in existence in the PC world for many years.

With the introduction of the international standard IEC 61131 a basis has now been created for uniform PLC programming taking advantage of the modern concepts of software technology.

1.1 Subject of the Book

The aim of this book is to give the reader an understandable introduction to the concepts and languages of standard IEC 61131. Simple examples are given to explain the ideas and application of the new PLC programming languages. An extensive example program summarises the results of each section.

The book serves as a helpful guide and introduction for people in training and at work who want to become acquainted with the possibilities of the new standard.

Some experience with personal computers and basic knowledge in the field of PLC technology are required. Experienced PLC programmers will also find information here which will ease the changeover to the programming systems of the new generation. For this purpose, the concepts and terminology of previous systems are compared and contrasted with those used in the world of IEC programming and the advantages of programming according to the IEC standard are explained.

This book is a useful reference work for students and facilitates the systematic learning of the new programming standard.

Readers can also use the enclosed "Buyer's Guide" to evaluate individual PLC programming systems for themselves. See the enclosed CD-ROM.

The formal contents and structure of the IEC standard are presented in a practice-oriented way. Difficult topics are clearly explained within their context, and the interpretation scope as well as extension possibilities of the standard are demonstrated.

This book is intended to give the reader concrete answers to the following questions :

- How do you program in accordance with IEC 61131? What are the essential ideas of the standard and how can they be applied in practice?
- What are the advantages of the new international standard IEC 61131 compared with previous (national) PLC programming standards? What innovations and opportunities does the new standard offer?

- What do users have to be aware of if they want to change to a programming system of the new generation?
- What features must contemporary programming systems have in order to be consistent with IEC 61131 and to fulfil this new standard?
- What do users need to look for when selecting a PLC programming system: what criteria are decisive for the performance of programming systems?

Chapter 2 presents the three basic building blocks of the standard: *program, function and function block*. An introductory example which includes the most important language elements of the standard and provides an overview of its programming methods gives an initial introduction to the concepts of IEC 61131.

Chapter 3 describes the *common language elements* of the five programming languages as well as the possibilities of data description with the aid of declarations.

The *five programming languages* of IEC 61131 are explained at length and illustrated by an extensive example in *Chapter 4*.

The strength of IEC 61131 is partly due to the uniform description of frequently used elements, the *standard functions* and *standard function blocks*. Their definition and application are described in *Chapter 5*.

After programming, the programs and the data have to be assigned to the features and hardware of the relevant PLC by means of *configuration*. This is to be found in *Chapter 6*.

The PLC market is developing into a technology with very specific requirements. These *special features of programming for a PLC* as well as their implementation using the new facilities of IEC 61131 are the subject of *Chapter 7*.

Chapter 8 summarises the most important qualities of the standard from Chapters 2 to 7. The essential advantages of the standard and of consistent programming systems are outlined here for reference.

Chapter 9 introduces the future standard *IEC 61499* for distributed automation processes. It is based on IEC 61131-3, but adopts a wider approach to cater for the demands for parallelism and decentralisation imposed by modern automation tasks.

Chapter 10 explains the use of the enclosed CD-ROM. It includes all the programming examples in this book, a buyer's guide in tabular form, and executable demo versions of two IEC programming systems.

The *Appendices* supply further detailed information.

The glossary in *Appendix J* gives a brief explanation of the most important terms used in this book in alphabetical order.

Appendix K contains the bibliography, which gives references not only to books but also to specialised papers on the subject of IEC 61131-3.

Appendix L is a general index which can be very helpful for the location of keywords.

1.2 The IEC 61131 standard

The five parts of the standard IEC 61131 summarise the requirements of modern PLC systems. These requirements concern the PLC hardware and the programming system.

The standard includes both the common concepts already in use in PLC programming and additional new programming methods.

IEC 61131-3 sees itself as a *guideline for PLC programming*, not as a rigid set of rules. The enormous number of details defined means that programming systems can only be expected to implement part but not all of the standard. PLC manufacturers have to document this amount: if they want to conform to the standard, they have to prove in which parts they do or do not fulfil the standard.

For this purpose, the standard includes 62 feature tables with requirements, which the manufacturer has to fill in with comments (e.g. "fulfilled; not implemented; the following parts are fulfilled:...").

The standard provides a *benchmark* which allows both manufacturers and customers to assess how closely each programming system keeps to the standard, i.e. complies with IEC 61131-3.

For further proof of compliance, PLCopen (see Section 1.3) defines further tests for compliance levels which can be carried out by independent institutions.

The standard was established by working group SC65B WG7 (originally: SC65A WG6) of the international standardisation organisation IEC (International Electrotechnical Commission) which consists of representatives of different PLC manufacturers, software houses and users. This has the advantage that it is accepted as a guideline by most PLC manufacturers.

1.2.1 Goals and benefits of the standard

Because of the constantly increasing complexity of PLC systems there is a steady rise in costs for:

- Training of applications programmers
- The creation of increasingly larger programs
- The implementation of more and more complex programming systems.

PLC programming systems are gradually following the mass software market trend of the PC world. Here too, the pressure of costs can above all be reduced by standardisation and synergy. Because the standard brings previously manufacturer-specific systems closer together, both manufacturers and customers stand to gain from IEC 61131-3.

Manufacturers (PLC hardware and software)
Several manufacturers can invest *together* in the multi-million dollar software required to fulfil the functionality necessary in today's market.

The basic form of a programming system is determined to a large extent by the standard. Basic software such as editors, with the exception of particular parts like code generators or "online"-modules, can be shared. Market differentiation results from supplementary elements to the basic package which are required in specific market segments, as well as from the PLC hardware.

Through the introduction of the standard a lively exchange of experience and products is currently taking place between hardware and software manufacturers. Development costs can be substantially reduced by buying ready-made products. The error-proneness of newly developed software can be greatly reduced by the use of previously tested software.

The risk of an inappropriate development (the system does not satisfy the market needs) is smaller. The standard sets the rules which the customer already knows from other IEC 61131-3 products.

The development costs of contemporary programming tools have increased significantly as a result of the required functionality. By buying ready-made software components or complete systems the "time to market" can be significantly shortened, which is essential in order to keep pace with the rapid hardware evolution.

Customers
Customers often work simultaneously with PLC systems from different manufacturers. Up to now this has meant that employees have needed to take several different training courses in programming, whereas with IEC 61131-3-compliant systems training is limited to the finer points of using the individual programming systems and additional special features of the PLCs. This cuts down on the need for system specialists and training personnel, and PLC programmers are more flexible.

The requirements of the standard ease the selection of suitable programming systems because systems that conform to the standard are easily comparable.

Though it is not expected that complete application programs will be able to be exchanged between different PLC systems in the foreseeable future, language elements and program structure are nevertheless similar among the different IEC systems. This facilitates porting onto other systems.

1.2.2 History and components

The standard IEC 61131 represents a combination and continuation of different standards. It refers to 10 other international standards (IEC 50, IEC 559, IEC 617-12, IEC 617-13, IEC 848, ISO/AFNOR, ISO/IEC 646, ISO 8601, ISO 7185, ISO 7498). These include rules about the employed character code, the definition of the nomenclature used or the structure of graphical representations.

Several efforts have been made in the past to establish a standard for PLC programming technology. Standard IEC 61131 is the first standard which has received the necessary international (and industrial) acceptance. The most important precursor documents to IEC 61131 are listed in Table 1.1.

Year	German	international
1977	DIN 40 719-6 (function block diagrams)	IEC 848
1979		Start of the working group for the first IEC 61131 draft
1982	VDI guideline 2880, sheet 4 PLC programming languages	Completion of the first IEC 61131 draft; Splitting into 5 sub-workgroups
1983	DIN 19239 PLC programming	Christensen Report (Allen Bradley) PLC programming languages
1985		First results of the IEC 65 A WG6 TF3
1990		IEC 61131 Parts 1 and 2 are made standard
1992		International standard IEC 61131-1, 2
1993	DIN EN 661131 Part 3	International standard IEC 61131-3
1994	DIN EN 661131 Parts 1 and 2	
1995		International standard IEC 61131-4
1996	Additional guide to DIN EN 661131 (user's guide, IEC 61131-4)	
1994 – 2001		Corrigendum to IEC 61131-3
1995, 1996		Technical Reports type 2 and 3
1996 - 2001		Amendments

Table 1.1. Important precursors and milestones of IEC 61131-3

The standard contains six parts as well as a Corrigendum, which includes error corrections in the standard (as of August 1999). Two Technical Reports and an Amendment complete the documentation. These are not (yet) however a direct part of the standard:

Part 1: *General information:*

Part 1 contains general definitions and typical functional features which distinguish a PLC from other systems. These include standard PLC properties, for example, the cyclic processing of the application program with a stored image of the input and output values or the division of labour between programming device, PLC and human-machine interface.

Part 2: *Equipment requirements and tests:*
This part defines the electrical, mechanical and functional demands on the devices as well as corresponding qualification tests. The environmental conditions (temperature, air humidity etc.) and stress classes of the controllers and of the programming devices are listed. A revision is at present under development.

Part 3: *Programming languages:*
Here the PLC programming languages widely used throughout the world have been co-ordinated into a harmonised and future-oriented version.

  The basic software model and programming languages are defined by means of formal definitions, lexical, syntactical and (partially) semantic descriptions, as well as examples.

Part 4: *User guidelines*
The fourth part is intended as a guide to help the PLC customer in all project phases of automation. Practice-oriented information is given on topics ranging from systems analysis and the choice of equipment right through to maintenance.

Part 5: *Messaging service specification:* (in preparation)
This last part is concerned with communication between PLCs from different manufacturers with each other and with other devices.

  In co-operation with ISO 9506 (Manufacturing Message Specification; MMS) conformity classes will be defined to allow PLCs to communicate, for example, via networks. These cover the functions of device selection, data exchange, alarm processing, access control and network administration. Part 5 has not yet been released (Committee Draft CD).

Technical Report 2 "Proposed Extensions to IEC 61131-3":
A list of proposals describes alternatives, extensions or changes to IEC 61131-3. This will be used for preparation of the next revision of the standard and exists at present in the form of a Committee Draft.

Technical Report 3 "Guidelines for the application and implementation of programming languages for programmable controllers":
This report supplies interpretations of points left open by the standard. It contains guidelines for implementation as well as application tips for the end user and programming advice.
It is currently at the voting stage (Committee Draft for Final Voting CDV) and will probably be published in IEC 61131-3.

Corrigendum "Proposed Technical Corrigendum to IEC 61131-3":
The Corrigendum corrects errors in the standard which were found after its publication.

Amendments "Proposed Amendments to IEC 61131-3":
The Amendments present a collection of improvements and above all extensions to the standard which have not as yet been published. This document was introduced in order to be able to add important improvements without having to wait for a complete new revision of the standard.

Part 8: *Fuzzy Control Language*:
The draft currently been debated at the voting stage extends the programming languages to include Fuzzy Logic.

The standard describes a modern technology and is therefore subject to strong innovation pressure. This explains why further development of the findings of the standard is being carried out at both national and international level.

This book is concerned with Part 3 "Programming Languages", in short IEC 61131-3. It also incorporates the findings and interpretations of the two Technical Reports ([IEC TR2-94] and [IEC TR3-94]) and the improvements described in the Corrigendum ([IEC CORR-94]).

IEC 61131-3 has been adopted in Germany as German standard "DIN EN 661131-3" ([DIN EN 661131-3-94]). It thus replaces the standards DIN 19239, DIN 40719T6 as well as the VDI guideline VDI 2880 Sheet 4.

1.3 The Organisation PLCopen

PLCopen, founded in 1992, is a manufacturer- and product-independent international organisation. Many PLC manufacturers, software houses and independent institutions in Europe and overseas are members of the organisation.

1.3.1 Aims

The aim of PLCopen is the promotion of the development and use of compatible software for PLCs ([PLCopen-99]).

The means for reaching this target are based on:

- The application of the international standard IEC 61131-3,
- The commitment of the members to produce or employ PLC products which conform to IEC 61131-3,
- Common marketing strategies such as fairs or workshops,
- Support for the international standardisation committee IEC WG 65B,
- Support for national standardisation committees like DIN-DKE UK 962.2 PLC,
- Establishment of compliance classes to allow better evaluation of programming systems, and commissioning of independent institutions to carry out the required checks.

PLCopen is not another standardisation committee, but rather a group with a common interest wanting to help existing standards to gain international acceptance. Detailed information can be found on the Internet (http://www.plcopen.org).

1.3.2 Committees and fields of activity

PLCopen is divided up into several committees, each of which handles a specific field of interest, as shown in Figure 1.1.:
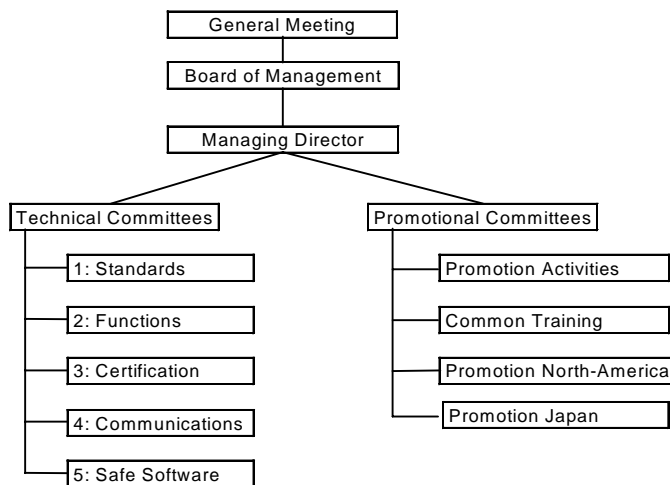
Figure 1.1. Committees of PLCopen

The technical committees work out guidelines for common policy; the promotional committees are responsible for marketing measures.

The work in the committees is carried out exclusively by representatives of individual companies and institutions. This ensures that the resulting papers will be accepted in industry.

1.3.3 Results

As a result of the preparatory work of the promotional committees, PLCopen is represented at several fairs in Europe and the USA. Workshops and advanced training seminars have brought the desired international recognition for PLCopen.

As a discussion forum for customers, manufacturers and software houses some impressive technical results have been achieved:

- Certification for manufacturers of PLC programming systems,
- Exchange format for user programs.

Certification

Certification tests demonstrate the standard compliance of programming systems designed and implemented in accordance with IEC 61131-3.

For this purpose a list of requirements which a PLC programming system has to fulfil in order to receive the PLCopen certificate of compliance with IEC 61131-3 was drawn up. The test is carried out by independent institutions with the aid of test programs and inspections.

For every programming language of IEC 61131-3 a grading system consisting of three levels exists with each level imposing stricter requirements. The requirements of the standard (feature table) mentioned in Section 1.2 serve as the basis of the classification. The feature table is used to determine which of the requirements must be available at the respective level.

1) **Base level**. The basic structure of programs developed with the programming system must be compatible with IEC 61131-3. The essential language elements of a programming language must be available.
2) **Portability level**. The selection of compulsory features is extended so that it is possible to exchange real software components between certified programming systems.
3) **Full level**. Further extension of the Portability level: inclusion of configuration information in the exchange process.

This enables all PLC manufacturers to document the degree of conformity of each of their languages to other IEC 61131-3-compliant systems.

By specifying a defined functionality, PLCopen guarantees a minimum compliance. This makes it possible to compare systems with each other.

Several of these compliance classes have already been established (IL Base Level, ST Base Level, AS Base Level, IL Portability level, FBD Base Level). The first IL certifications for PLC programming systems were awarded in August 1994. The remaining compliance classes are still in preparation.

While the Base Level test is purely offline, i.e. the certification test programs check the syntactical behaviour of the programming system, the Portability Level includes additional online tests, which test the semantic behaviour of the programming system in connection with a PLC. This ensures that the programming system interprets IEC 61131-3 in the correct manner.

To what extent portability can be implemented in actual systems is a constant subject of discussion and still remains to be seen. Limiting factors here are hardware features of the manufacturer-specific PLCs, which often shape the program architecture. Furthermore, the high functionality of IEC 61131-3 makes it extremely difficult for programming systems and PLC operating systems to implement all the functions. Small PLC manufacturers, on account of their target market or the capabilities of their PLC families, will hardly require all the properties of the standard — the implementation costs for parts of IEC 61131-3 are simply too great.

Thus the need to be able to exchange and/or port all IEC elements between all programming systems is not the highest priority. It is more important that a certain basic functionality should be available and that functionalities defined by the standard should be implemented correctly.
In the final analysis the wishes and demands of the customers will give the ultimate answer to all questions.

Exchange format for user programs
In order to be able to exchange user programs between two PLC programming systems, both must be able to understand the same file format, read it in and convert it into their own format.
   This exchange can be done by reading an ASCII file into the target system (as long as an import/export interface is available) or by means of the copy/paste function. In this case, the type definitions, data declarations and the code section are read in.

For a secured exchange of a program source, information such as place of origin, version, date, programmer's name etc. is important. As there are no rules or regulations in the IEC 61131-3 standard concerning file format, PLCopen has defined an ASCII-based format for textual blocks (exchange format FxF).

Not only programs can be imported into other IEC 61131-3 systems. In large applications, it is desirable to exchange data while a program is running. A special data format is necessary here, too. Information such as place of origin and the expiration date must be supplied with the data. PLCopen proposes data structures which can, for example, be transmitted with the aid of the blocks described in IEC 61131-5.

2 Building Blocks of IEC 61131-3

This chapter explains the meaning and usage of the main language elements of the IEC 61131-3 standard. These are illustrated by several examples from real life, with each example building upon the previous one.

The reader is introduced to the terms and ways of thinking of IEC 61131-3. The basic ideas and concepts are explained clearly and comprehensively without discussing the formal language definitions of the standard itself [IEC 61131-3-94].

The first section of this chapter gives a compact introduction to the conceptual range of the standard by means of an example containing the most important language elements and providing an overview of the methodology of PLC programming with IEC 61131-3.

The term "POU" (Program Organisation Unit) is explained in detail because it is fundamental for a complete understanding of the new language concepts.

As the programming language Instruction List (IL, see Chapter 4.1) is already well known to most PLC programmers, it has been chosen as the basis for the examples in this chapter .

2.1 Introduction to the New Standard

IEC 61131-3 not only describes the PLC programming languages themselves, but also offers comprehensive concepts and guidelines for creating PLC projects.

The purpose of this section is to give a short summary of the important terms of the standard without going into details. These terms are illustrated by a simple example. More detailed information will be found in the subsequent sections and chapters.

2.1.1 Structure of the building blocks

POUs correspond to the *Blocks* in previous (conventional) programming systems. POUs can call each other with or without parameters. As the name implies, POUs are the smallest independent software units of a user program.

There are three types of POUs: *Function (FUN)*, *Function block (FB)* and *Program (PROG)*, in ascending order of functionality. The main difference between functions and function blocks is that functions always produce the same result (function value) when called with the same input parameters, i.e. they have no "memory". Function blocks have their own data record and can therefore "remember" status information (instantiation). Programs (PROG) represent the "top" of a PLC user program and have the ability to access the I/Os of the PLC and to make them accessible to other POUs.

IEC 61131-3 predefines the calling interface and the behaviour of frequently needed s*tandard functions (std. FUN)* such as arithmetic or comparison functions, as well as *standard function blocks (std. FB),* such as timers or counters.

Declaration of variables

The IEC 61131-3 standard uses *variables* to store and process information. Variables correspond to (global) flags or bit memories in conventional PLC systems. However, their storage locations no longer need to be defined manually by the user (as absolute or global addresses), but they are managed automatically by the programming system and each possess a fixed *data type*.

IEC 61131-3 specifies several data types (Bool, Byte, Integer, ...). These differ, for example, in the number of bits or the use of signs. It is also possible for the user to define new data types: user-defined data types such as structures and arrays.

Variables can also be assigned to a certain I/O address and can be battery-backed against power failure.

Variables have different forms. They can be defined (declared) outside a POU and used program-wide, they can be declared as interface parameters of a POU, or they can have a local meaning for a POU. For declaration purposes they are therefore divided into different *variable types*. All variables used by a POU have to be declared in the declaration part of the POU.

The declaration part of a POU can be written in textual form **independently of the programming language used**. Parts of the declaration (input and output parameters of the POU) can also be represented graphically.

```
VAR_INPUT                        (* Input variable *)
  ValidFlag     : BOOL;          (* Binary value *)
END_VAR
VAR_OUTPUT                       (* Output variable *)
  RevPM         : REAL;          (* Floating-point value *)
END_VAR
VAR RETAIN                       (* Local variable, battery-backed *)
  MotorNr       : INT;           (* Signed integer *)
  MotorName   : STRING [10];     (* String of length 10 *)
  EmStop AT %IX2.0 : BOOL;       (* Input bit 2.0 of I/O *)
END_VAR
```

Example 2.1. Example of typical variable declarations of a POU

Example 2.1 shows the variable declaration part of a POU. A signed integer variable (16 bits incl. sign) with name MotorNr and a text of length 10 with name MotorName are declared. The binary variable EmStop (emergency stop) is assigned to the I/O signal input 2.0 (using the keyword "AT"). These three variables are known only within the corresponding POU, i.e. they are "local". They can only be read and altered by this POU. During a power failure they retain their value, as is indicated by the qualifier "RETAIN". The value for input variable ValidFlag will be set by the calling POU and have the Boolean values TRUE or FALSE. The output parameter returned by the POU in this example is the floating-point value RevPM.

The Boolean values TRUE and FALSE can be also be indicated by "1" and "0".

Code part of a POU
The code part, or instruction part, follows the declaration part and contains the instructions to be processed by the PLC.

A POU is programmed using either the textual programming languages Instruction List (IL) and Structured Text (ST) or the graphical languages Ladder Diagram (LD) and Function Block Diagram (FBD). IL is a programming language closer to machine code, whereas ST is a high-level language. LD is suitable for Boolean (binary) logic operations. FBD can be used for programming both Boolean (binary) and arithmetic operations in graphical representation.
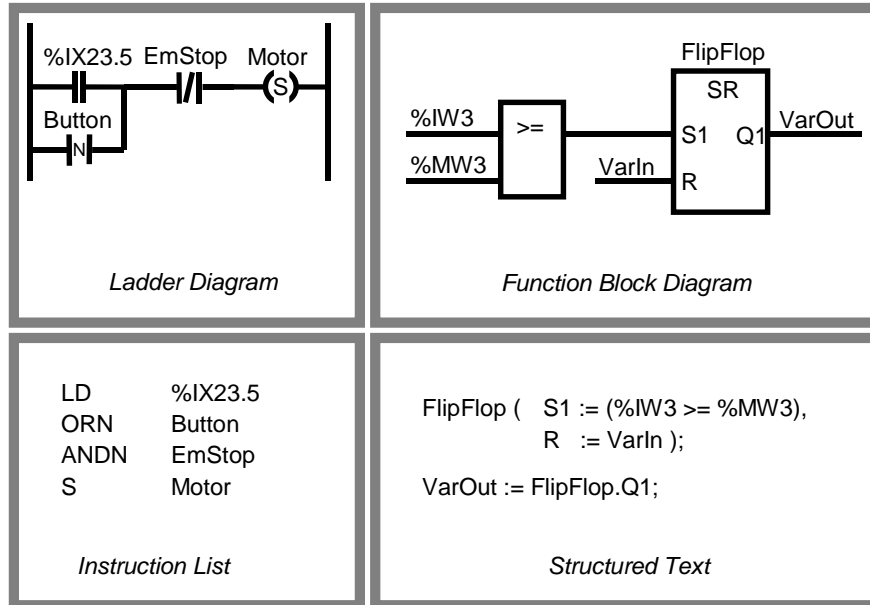
Figure 2.1. Simple examples of the programming languages LD, FBD, IL and ST.
The examples in LD and IL are equivalent to one another, as are those in FBD and ST.

Additionally, the description language Sequential Function Chart (SFC) can be used to describe the *structure* of a PLC program by displaying its sequential and parallel execution. The various subdivisions of the SFC program (steps and transitions) can be programmed independently using any of the IEC 61131-3 programming languages.

Sequential Function Chart
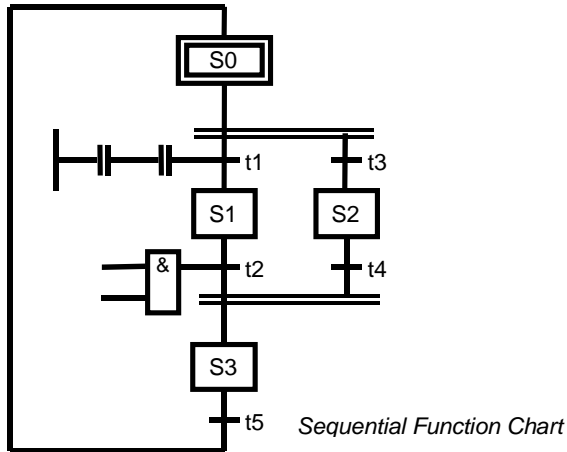
Figure 2.2. Schematic example of structuring using SFC. The execution parts of the steps (S0 to S3) and the transitions (t1 to t5) can be programmed using any other programming language.

Figure 2.2 shows an SFC example: Steps S0, S1 and S3 are processed sequentially. S2 can be executed alternatively to S1. Transitions t1 to t5 are the conditions which must be fulfilled before proceeding from one step to the next.

2.1.2 Introductory example written in IL

An example of an IEC 61131-3 program is presented in this section. Figure 2.3 shows its POU calling hierarchy in tree form.

This example is not formulated as an executable program, but simply serves to demonstrate POU structuring.
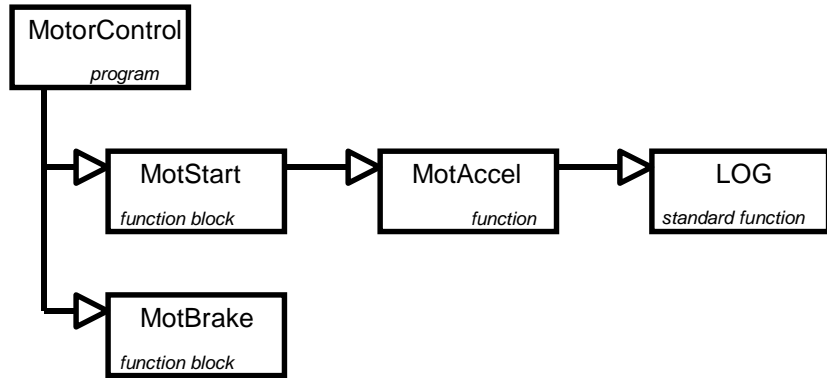
Figure 2.3. Calling hierarchy of POUs in the example

The equivalent IL representation is shown in Example 2.2.
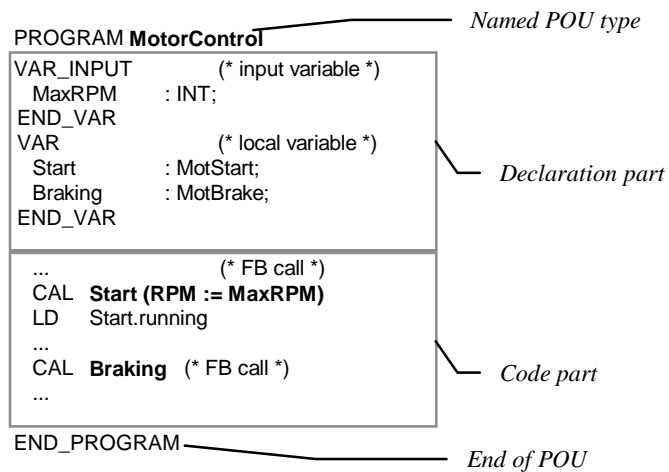
Example 2.2. Declaration of the program MotorControl from Figure 2.3 together with corresponding code parts in IL. Comments are represented by the use of brackets: "(* ... *)".

```
FUNCTION_BLOCK MotStart                    (* function block *)
VAR_INPUT     RPM:    INT;    END_VAR      (* declaration of RPM*)
VAR_OUTPUT   running: BOOL;   END_VAR      (* declaration of running*)
  ...
  LD        RPM
  MotAccel  100                            (* function call *)
  ...
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK MotBrake                    (* function block *)
...
END_FUNCTION_BLOCK
```

```
FUNCTION MotAccel : REAL                   (* function *)
VAR_INPUT Param1, Param2: INT; END_VAR     (* declaration of variables*)
  LD    Param2
  LOG                                      (* invoc. of Std. FUN LOG *)
  ...
  ST    MotAccel
END_FUNCTION
```

Example 2.3. The three subprograms of Fig. 2.3 in IL. LOG (logarithm) is a predefined standard function of IEC 61131-3.

MotorControl is the main program. When this program is started, the variable RPM is assigned an initial value passed with the call (as will be seen later). This POU then calls the block Start (MotStart). This POU in turn calls the REAL function MotAccel with two input parameters (RPM and 100). This then calls LOG – the IEC 61131 *standard function* "Logarithm". After processing Start (MotStart), MotorControl is activated again, evaluates the result running and then calls Braking, (MotBrake).

As shown in Example 2.2, the function blocks MotStart and MotBrake are not called directly using these names, but with the so-called "*instance names*" Start and Braking respectively.

2.1.3 PLC assignment

Each PLC can consist of multiple processing units, such as CPUs or special processors. These are known as *resources* in IEC 61131-3. Several programs can run on one resource. The programs differ in priority or execution type (periodic/cyclic or by interrupt). Each program is associated with a *task*, which makes it into a *run-time program*. Programs may also have multiple associations (*instantiation*).

Before the program described in Examples 2.2 and 2.3 can be loaded into the PLC, more information is required to ensure that the associated task has the desired properties:

- On which PLC type and which resource is the program to run?
- How is the program to be executed and what priority should it have?
- Do variables need to be assigned to physical PLC addresses?
- Are there global or external variable references to other programs to be declared?

This information is stored as the *configuration*, as illustrated textually in Example 2.4 and graphically in Figure 2.4.

```
CONFIGURATION MotorCon
  VAR_GLOBAL Trigger AT %IX2.3 : BOOL;  END_VAR
  RESOURCE Res_1 ON CPU001
    TASK T_1         (INTERVAL := t#80ms,   PRIORITY := 4);
    PROGRAM MotR  WITH T_1 : MotorControl (MaxRPM := 12000);
  END_RESOURCE
  RESOURCE Res_2 ON CPU002
    TASK T_2         (SINGLE := Trigger, PRIORITY := 1);
    PROGRAM MotP  WITH T_2 : MotorProg (...);
  END_RESOURCE
END_CONFIGURATION
```

Example 2.4. Assignment of the programs in Example 2.3 to tasks and resources in a "configuration"
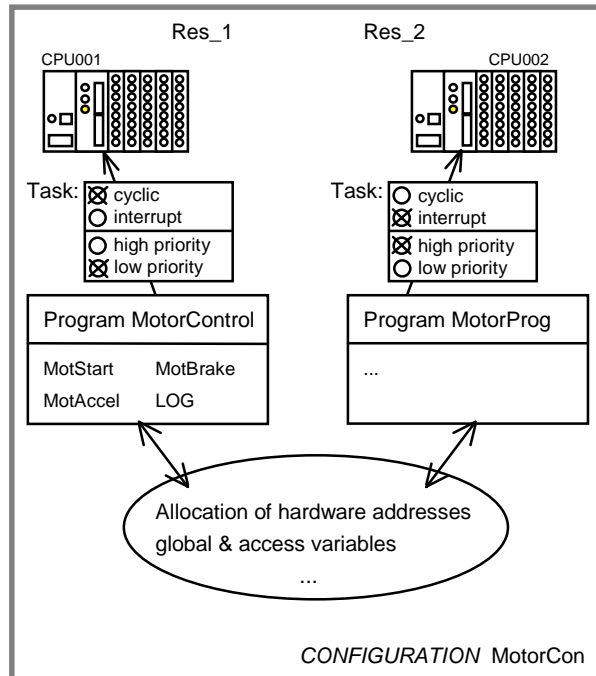
Figure 2.4. Assignment of the programs of a motor control system MotorCon to tasks in the PLC "configuration". The resources (processors) of the PLC system execute the resulting run-time programs.

Figure 2.4 continues Example 2.3. Program MotorControl runs together with its FUNs and FBs on resource CPU001. The associated task specifies that MotorControl should execute cyclically with low priority. Program MotorProg runs here on CPU002, but it could also be executed on CPU001 if this CPU supports multitasking.

The configuration is also used for assigning variables to I/Os and for managing global and communication variables. This is also possible within a PROGRAM.

A PLC project consists of POUs that are either shipped by the PLC manufacturer or created by the user. User programs can be used to build up libraries of tested POUs that can be used again in new projects. IEC 61131-3 supports this aspect of software re-use by stipulating that functions and function blocks have to remain "universal", i.e. hardware-independent, as far as possible.

After this short summary, the properties and special features of POUs will now be explained in greater detail in the following sections.

2.2 The Program Organisation Unit (POU)

IEC 61131-3 calls the blocks from which programs and projects are built Program Organisation Units (POUs). POUs correspond to the program blocks, organisation blocks, sequence blocks and function blocks of the conventional PLC programming world.

One very important goal of the standard is to restrict the variety and often implicit meanings of block types and to unify and simplify their usage.

Block types used in DIN 19239   POUs in IEC 61131-3

Figure 2.5. Evolution from previous block types (e.g. German DIN 19239) to the POUs of IEC 61131-3

As Figure 2.5 shows, IEC 61131-3 reduces the different block types of PLC manufacturers to three unified basic types. Data blocks are replaced by FB data memories ("instances", see below) or global multi-element variables (see also Chapter 3).

The following three *POU types* or "block types" are defined by the new standard:

POU type	keyword	meaning
Program	PROGRAM	Main program including assignment to I/O, global variables and access paths
Function block	FUNCTION_BLOCK	Block with input and output variables; this is the most frequently used POU type
Function	FUNCTION	Block with function value for extension of the basic PLC operation set

Table 2.1. The three POU types of IEC 61131-3 with their meanings

These three POU types differ from each other in certain features:
- **Function (FUN).** POU that can be assigned parameters, but has no static variables (without memory), which, when invoked with the same input parameters, always yields the same result as its function value (output).
- **Function block (FB)**. POU that can be assigned parameters and has static variables (with memory). An FB (for example a counter or timer block), when invoked with the same input parameters, will yield values which also depend on the state of its internal (VAR) and external (VAR_EXTERNAL) variables, which are retained from one execution of the function block to the next.
- **Program (PROG).** This type of POU represents the "main program". All variables of the whole program, that are assigned to physical addresses (for example PLC inputs and outputs) must be declared in this POU or above it (Resource, Configuration). In all other respects it behaves like an FB.

PROG and FB can have both input and output parameters. Functions, on the other hand, have input parameters and a function value as return value. These properties were previously confined to "function blocks".

   The IEC 61131-3 FUNCTION_BLOCK with input *and* output parameters roughly corresponds to the conventional function block. The POU types PROGRAM and FUNCTION do not have direct counterparts in blocks as defined in previous standards, e.g. DIN 19239.

A POU is an encapsulated unit, which can be compiled independently of other program parts. However, the compiler needs information about the calling interfaces of the other POUs that are called in the POU ("prototypes"). Compiled POUs can be linked together later in order to create a complete program.

The name of a POU is known throughout the whole project and may therefore only be used once. Local subroutines as in some other (high-level) languages are not permitted in IEC 61131-3. Thus, after programming a POU (*declaration*), its name and its calling interface will be known to all other POUs in the project, i.e. the POU name is always global.

This independence of POUs facilitates extensive modularization of automation tasks as well as the *re-use* of already implemented and tested software units.

In the following sections the common properties of the different types of POUs will first be discussed. The POU types will then be characterised, the calling relationships and other properties will be described, and finally the different types will be summarised and compared.

2.3 Elements of a POU

A POU consists of the elements illustrated in Figure 2.6:

-   POU type and name (and data type in the case of functions)
-   Declaration part with variable declarations
-   POU body with instructions.
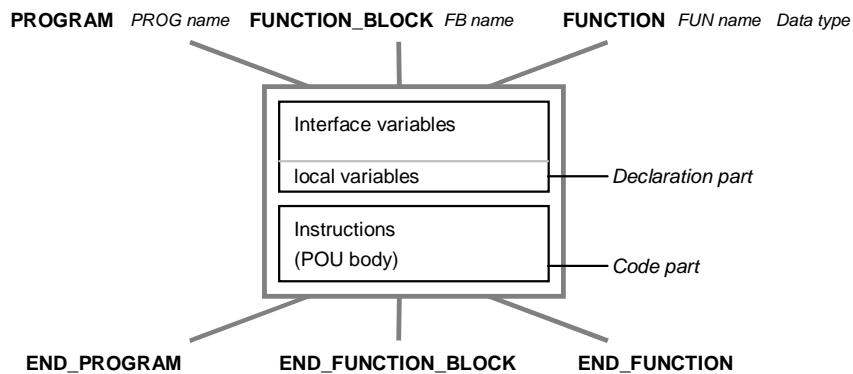
Figure 2.6. The common structure of the three POU types Program (left), Function Block (centre) and Function (right). The declaration part contains interface and local variables.

Declarations define all the variables that are to be used within a POU. Here a distinction is made between variables visible from outside the POU (POU interface) and the local variables of the POU. These possibilities will be explained in the next section and in more detail in Chapter 3.

Within the *code part* (body) of a POU the logical circuit or algorithm is programmed using the desired programming language. The languages of IEC 61131-3 are presented and explained in Chapter 4.

Declarations and instructions can be programmed in graphical or textual form.

2.3.1 Example

The elements of a POU are illustrated in Example 2.5.


```
FUNCTION_BLOCK FB name  ———  FUNCTION_BLOCK   NextState
                              ┌──────────────────────────────────────────────────┐
                              │ VAR_INPUT   VarIn    : BOOL;  END_VAR   (* input *) │
 ┌──────────────────────┐     │ VAR_OUTPUTVarOut1   : BYTE;            (* outputs *)│
 │ Interface variables  │     │             VarOut2 : BOOL;  END_VAR              │
 ├──────────────────────┤     │ VAR            VarLocal  : BYTE;  END_VAR (* state value *)│
 │ Local variables      │     └──────────────────────────────────────────────────┘
 └──────────────────────┘     ┌──────────────────────────────────────────────────┐
 ┌──────────────────────┐     │ LD  VarIn                                          │
 │ Instructions         │     │ ...                                                │
 │ (POU body)           │     │ LD  VarLocal                                       │
 └──────────────────────┘     │ ST  VarOut1                                        │
                              │ ...                                                │
                              │ ST  VarOut2                                        │
                              └──────────────────────────────────────────────────┘
END_FUNCTION_BLOCK  ———  END_FUNCTION_BLOCK
```
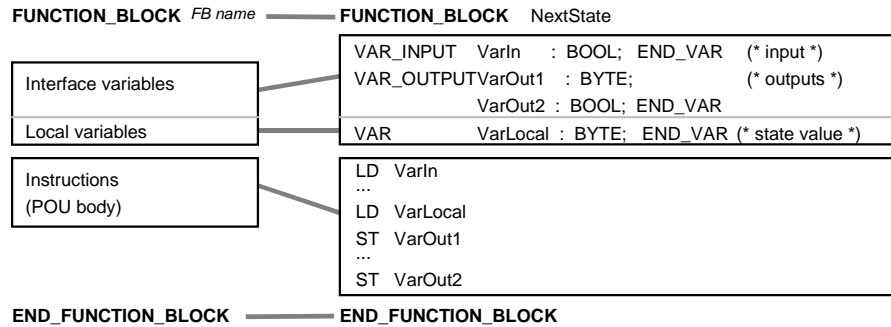
Example 2.5. Elements of a POU (left) and example of a function block in IL (right). The FB contains the input parameter VarIn as well as the two return values VarOut1 and VarOut2. VarLocal is a local variable.

The function block NextState written in IL contains the input parameter VarIn, the two return values VarOut1 and VarOut2 and the local variable VarLocal. In the FB body the IL operators LD (Load) and ST (Store) are used.

FUNCTION_BLOCK NextState


```
         ┌──────────────────┐
BOOL ────│ VarIn   VarOut1   │──── BYTE
         │         VarOut2   │──── BOOL
         └──────────────────┘
```

END_FUNCTION_BLOCK
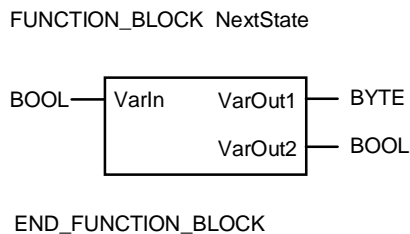
Example 2.6. Graphical representation of the calling interface of FB NextState in Example 2.5.

When using the graphical representation of the calling interface, local FB variables such as VarLocal are not visible.

2.3.2 Declaration part

In IEC 61131-3 variables are used for initialising, processing and storing user data. The variables have to be *declared* at the beginning of each POU, i.e. their assignment to a specific *data type* (such as BYTE or REAL) is made known.

Other *attributes* of the variables, such as battery backup, initial values or assignment to physical addresses, can also be defined during declaration.

As shown by Example 2.7, the declaration of POU variables is divided into separate sections for the different *variable types*. Each *declaration block* (VAR_*...END_VAR) corresponds to one variable type and can contain one or more variables. As Example 2.8 shows, the order and number of blocks of the same variable type can be freely determined or can depend on how the variables are used in a particular programming system.

```
(* Local variable *)
VAR              VarLocal : BOOL;  END_VAR    (* local Boolean variable *)
(* Calling interface: input parameters *)
VAR_INPUT        VarIn : REAL;      END_VAR    (* input variable *)
VAR_IN_OUT       VarInOut : UINT;   END_VAR    (* input and output variable *)
(* Return values: output parameters *)
VAR_OUTPUT       VarOut : INT;      END_VAR    (* output variable *)
(* Global interface: global/external variables and access paths *)
VAR_EXTERNAL VarGlob : WORD;  END_VAR    (* external from other POU *)
VAR_GLOBAL       VarGlob : WORD;  END_VAR    (* global for other POUs *)
VAR_ACCESS       VarPath : WORD;  END_VAR    (* access path of configuration *)
```

Example 2.7. Examples of the declarations of different variable types.

```
(* Declaration block 1 *)
VAR             VarLocal1, VarLocal2, VarLocal3: BOOL; END_VAR
 (* Declaration block 2 *)
VAR_INPUT   VarIn1 : REAL;                        END_VAR
 (* Declaration block 3 *)
VAR_OUTPUT VarOut : INT;                          END_VAR
 (* Declaration block 4 *)
VAR             VarLocal4, VarLocal5 : BOOL;      END_VAR
 (* Declaration block 5 *)
VAR_INPUT   VarIn2, VarIn3 : REAL;                END_VAR
 (* Declaration block 6 *)
VAR_INPUT   VarIn4 : REAL;                        END_VAR
```

Example 2.8. Examples of declaration blocks: the order and number of the blocks is not specified in IEC 61131-3.

Types of variables in POUs.
As shown by Table 2.2, different types of variables may be used depending on the POU type:

Variable type	Permitted in:		
	PROGRAM	*FUNCTION_BLOCK*	*FUNCTION*
VAR	yes	yes	yes
VAR_INPUT	yes	yes	yes
VAR_OUTPUT	yes	yes	no
VAR_IN_OUT	yes	yes	no
VAR_EXTERNAL	yes	yes	no
VAR_GLOBAL	yes	no	no
VAR_ACCESS	yes	no	no

Table 2.2. Variable types used in the three types of POU

As Table 2.2 shows, all variable types can be used in programs. Function blocks cannot make global variables available to other POUs. This is only permitted in programs, resources and configurations. FBs access global data using the variable type VAR_EXTERNAL.

Functions have the most restrictions because only local and input variables are permitted in them. They return their calculation result using the function return value.

Except for local variables, all variable types can be used to import data into and export data from a POU. This makes data exchange between POUs possible. The features of this *POU interface* will be considered in more detail in the next section.

Characteristics of the POU interface

The POU interfaces, as well as the local data area used in the POU, are defined by means of assigning POU variables to variable types in the declaration blocks. The POU interface can be divided into the following sections:

- Calling or invocation interface: formal parameters (input and input/output parameters)
- Return values: output parameters or function return values
- Global interface with global/external variables and access paths.

The calling interface and the return values of a POU can also be represented graphically in the languages LD and FBD.

The variables of the calling interface are also called *formal parameters*. When calling a POU the formal parameters are replaced with *actual parameters*, i.e. assigned actual (variable) values or constants.

In Example 2.3 FB MotStart has only one formal parameter RPM, which is given the value of the actual parameter MaxRPM in Example 2.2, and it also has the output parameter running. The function MotAccel has two formal parameters (one of which is assigned the constant 100) and returns its result as the function return value

This is summarised by Table 2.3.

	Variable types	Remarks
Calling interface (formal parameters)	VAR_INPUT, VAR_IN_OUT	Input parameters, can also be graphically displayed
Return values	VAR_OUTPUT	Output parameters, can also be graphically displayed
Global interface	VAR_GLOBAL, VAR_EXTERNAL, VAR_ACCESS	Global data
Local values	VAR	POU internal data

Table 2.3. Variable types for interface and local data of a POU. See the comments in Example 2.7.

The formal parameters and return values of a POU
The calling interface and return value differ in their method of access and in their access rights (see also [IEC TR3-94]).

Formal input parameter (VAR_INPUT): Actual parameters are passed to the POU as *values*, i.e. the variable itself is not used, but only a copy of it. This

ensures that this input variable cannot be changed within the called POU. This concept is also known as *call-by-value*.

Formal input/output parameter (VAR_IN_OUT): Actual parameters are passed to the called POU in the form of a *pointer to their storage location*, i.e. the variable itself is used. It can thus be read and changed by the called POU. Such changes have an automatic effect on the variables declared outside the called POU. This concept is also known as *call-by-reference*.

By working with references to storage locations this variable type provides pointers like those used in high-level languages like C for return values from subroutines.

Formal output parameters, r**eturn values** (VAR_OUTPUT) are not passed to the called POU, but are provided by that POU as values. They are therefore not part of the calling interface. They appear together with VAR_INPUT and VAR_IN_OUT in graphical representation, but in textual languages such as IL or ST their *values* are read *after* calling the POU.

The method of passing to the calling POU is also *return-by-value*, allowing the values to be read by the calling instance (FB or PROG). This ensures that the output parameters of a POU are protected against changes by a calling POU. When a POU of type PROGRAM is called, the output parameters are provided together with the actual parameters by the resource and assigned to appropriate variables for further processing (see examples in Chapter 6).

If a POU call uses complex arrays or data structures as variables, the use of VAR_IN_OUT results in more efficient programs, as it is not the variables themselves (VAR_INPUT and VAR_OUTPUT) that have to be copied at run time, but only their respective pointers. However such variables are not protected against (unwelcome) manipulation by the called POU.

External and internal access to POU variables
Formal parameters and return values have the special property of being *visible* outside their POU: the calling POU can (but need not) use their names explicitly for setting input values.

This makes it easier to document the POU calling interface and parameters may be omitted and/or their sequence may be altered. In this context input and output variables are also protected against unauthorised reading and writing.

Table 2.4 summarises all variable types and their meaning. Access rights are given for each variable type, indicating whether the variable:

- is visible to the calling POU ("external") and can be read or written to there
- can be read or written to within the POU ("internal") in which it is defined.

Variable type	Access rights[a]		Explanation
	external	internal	
VAR *Local Variables*	-	RW	A local variable is only visible within its POU and can be processed only there.
VAR_INPUT *Input Variables*	W	R	An input variable is visible to the calling POU and may be written to (changed) there. It may not be changed within its own POU.
VAR_OUTPUT *Output Variables*	R	RW	An output variable is visible to the calling POU and may only be read there. It can be read and written to within its own POU.
VAR_IN_OUT *Input and Output Variables*	RW	RW	An input/output variable possesses the combined features of VAR_INPUT and VAR_OUTPUT: it is visible and may be read or changed within or outside its POU.
VAR_EXTERNAL *External Variables*	RW	RW	An external variable must be declared by another POU as *global* and is visible, and can be read and written to by all POUs. It may be changed within a POU just like a local variable and its changed value will be effective immediately for all POUs using it.
VAR_GLOBAL *Global Variables*	RW	RW	A global variable is declared within a POU and can be read and written to there by all other POUs (as *external* variable). It may be changed within a POU as a local variable and its new value will be effective immediately for all POUs using it.
VAR_ACCESS *Access Paths*	RW	RW	Global variable of configurations as communication channel between components (resources) of configurations (see also Chapter 6). It can be used like a global variable within a POU.

a   W=Write, R=Read, RW=Read and Write

Table 2.4. The meaning of the variable types. The left-hand column contains the keyword of each variable type in bold letters. In the "Access rights" column the read/write rights are indicated for the calling POU (external) and within the POU (internal) respectively.

IEC 61131-3 provides extensive access protection for input and output variables, as shown in Table 2.4 for VAR_INPUT and VAR_OUTPUT: input variables may not be changed within their POU, and output parameters may not be changed outside.

The information in this section about the declaration part of a POU is particularly relevant for function blocks. Section 2.4.1 will discuss this again when explaining FB instantiation.

The following examples show both external (calling the POU) and internal (within the POU) access to formal parameters and return values of POUs:

FUNCTION_BLOCK **FBTwo**   FUNCTION_BLOCK **FBOne**

```
VAR_INPUT                                VAR ExampleFB : FBTwo; END_VAR
  VarIn    : BYTE;
END_VAR
VAR_OUTPUT
  VarOut   : BYTE;
END_VAR
VAR VarLocal : BYTE; END_VAR



                                         ...
...                                      LD      44
LD      VarIn                            ST      ExampleFB.VarIn
AND     VarLocal                         CAL     ExampleFB       (* FB call *)
ST      VarOut                           LD      ExampleFB.VarOut
...                                      ...
END_FUNCTION_BLOCK                       END_FUNCTION_BLOCK
```

Example 2.9. Internal access (on the left) and external access (on the right) to the formal parameters VarIn and VarOut.

In Example 2.9 FBOne calls block ExampleFB (described by FBTwo). The input variable VarIn is assigned the constant 44 as actual parameter, i.e. this input variable is visible and initialised in FBOne. VarOut is also visible here and can be read by FBOne. Within FBTwo VarIn can be read (e.g. by LD) and VarOut can be written to (e.g. using the instruction ST).

Further features and specialities of variables and variable types will be explained in Section 3.4**.**

2.3.3 Code part

The instruction or code part (body) of a POU immediately follows the declaration part and contains the instructions to be executed by the PLC. IEC 61131-3 provides five programming languages (three of which have graphical representation) for application-oriented formulation of the control task.

As the method of programming differs strongly between these languages, they are suitable for different control tasks and application areas. Here is a general guide to the languages:

SFC	*Sequential Function Chart*: For breaking down the control task into parts which can be executed sequentially and in parallel, as well as for controlling their overall execution. SFC very clearly describes the program flow by defining which actions of the controlled process will be enabled, disabled or terminated at any one time. IEC 61131-3 emphasises the importance of SFC as an *Aid for Structuring* PLC programs.
LD	*Ladder Diagram*: Graphical connection ("circuit diagram") of Boolean

	variables (contacts and coils), geometrical view of a circuit similar to earlier relay controls. POUs written in LD are divided into sections known as *networks*.
FBD	*Function Block Diagram*: Graphical connection of arithmetic, Boolean or other functional elements and function blocks. POUs written in FBD are divided into *networks* like those in LD. *Boolean* networks can often be represented in LD and vice versa.
IL	*Instruction List*: Low-level machine-oriented language offered by most of the programming systems
ST	*Structured Text*: High-level language (similar to PASCAL) for control tasks as well as complex (mathematical) calculations.

Table 2.5. Features of the programming languages of IEC 61131-3

In addition, the standard explicitly allows the use of other programming languages (e.g. C or BASIC), which fulfil the following basic requirements of PLC programming:

- The use of variables must be implemented in the same way as in the other programming languages of IEC 61131-3, i.e. compliant with the declaration part of a POU.
- Calls of functions and function blocks must adhere to the standard, especially calls of standard functions and standard function blocks.
- There must be no inconsistencies with the other programming languages or with the structuring aid SFC.

Details of these standard programming languages, their individual usage and their representation are given in Chapter 4.

2.4 The Function Block

Function blocks are the main building blocks for structuring PLC programs. They are called by programs and FBs and can themselves call functions as well as other FBs.

   In this section the basic features of function blocks will be explained. A detailed example of an FB can be found in Appendix C.

The concept of the "instantiation of FBs" is of great importance in IEC 61131-3 and is an essential distinguishing criterion between the three POU types. This concept will therefore be introduced before explaining the other features of POUs.

2.4.1 Instances of function blocks

What is an "instance"?

The creation of variables by the programmer by specifying the variable's name and data type in the declaration is called *instantiation*.

In the following Example 2.10 the variable Valve is an *instance* of data type BOOL:

Valve   :   BOOL;   (* Boolean variable *)

          Name of variable   *data type*

Motor1   :   MotorType;   (* FB instance *)

          Name of FB instance   *FB type (user-defined)*

Example 2.10. Declaration of a variable as an "instance of a data type" (top). Declaration of an FB "variable" as an "instance of a user-defined FB type" (bottom).

Function blocks also are instantiated like variables: In Example 2.10 the FB instance Motor1 is declared as an instance of the user-defined function block (FB type) MotorType in the declaration part of a POU. After instantiation an FB can be used (as an instance) and called within the POU in which it is declared.

This principle of instantiation may appear unusual at first sight but, in fact, it is nothing new.

Up until now, for example, function blocks for counting or timing, known for short as counters and timers respectively, were mostly defined by their type (such as direction of counting or timing behaviour) and by a number given by the user, e.g. Counter "C19".

Instead of this absolute number the standard IEC 61131-3 requires a (symbolic) variable name combined with the specification of the desired timer or counter type. This has to be declared in the declaration part of the POU. The programming system can automatically generate internal, absolute numbers for these FB variables when compiling the POU into machine code for the PLC.

With the aid of these variable names the PLC programmer can use different timers or counters of the same type in a transparent manner and without the need to check name conflicts.

By means of instantiation IEC 61131-3 unifies the usage of manufacturer-dependent FBs (typically timers and counters) and user-defined FBs. Instance names correspond to the symbolic names or so-called *symbols* used by many PLC programming systems. Similarly, an FB type corresponds to its calling interface. In fact, FB instances provide much more than this: "Structure" and "Memory" for FBs will be explained in the next two subsections.

The term "function block" is often used with two slightly different meanings: it serves as a synonym for the FB instance name as well as for the FB type (= name of the FB itself). In this book "function block" means *FB type,* while an FB instance will always be explicitly indicated as an instance name.

Example 2.11 shows a comparison between the declarations of function blocks (here only standard FBs) and variables:

```
VAR
  FillLevel     : UINT;      (* unsigned integer variable *)
  EmStop        : BOOL;      (* Boolean variable *)
  Time9         : TON;       (* timer of type on-delay *)
  Time13        : TON;       (* timer of type on-delay *)
  CountDown     : CTD;       (* down-counter *)
  GenCounter    : CTUD;      (* up-down counter *)
END_VAR
```

Example 2.11. Examples of variable declaration and instantiation of standard function blocks (bold).

Although in this example `Time9` and `Time13` are based on the same FB type (TON) of a timer FB (on-delay), they are independent timer blocks which can be separately called as instances and are treated independently of each other, i.e. they represent two different "timers".

FB instances are visible and can be used within the POU in which they are declared. If they are declared as global, all other POUs can used them as well (with VAR_EXTERNAL).

Functions, on the other hand, are always visible project-wide and can be called from any POU without any further need of declaration. Similarly FB **types** are known project-wide and can be used in any POU for the declaration of instances.

The declaration and calling of standard FBs will be described in detail in Chapter 5. Their usage in the different programming languages is explained in Chapter 4.

Instance means "structure".
The concept of instantiation, as applied in the examples of timer or counter FBs, results in *structured variables*, which:

-   describe the FB calling interface like a data structure,
-   contain the actual status of a timer or counter,
-   represent a method for calling FBs.

This allows flexible parameter assignment when calling an FB, as can be seen below in the example of an up/down counter:

```
VAR
  Counter  :    CTUD;      (* up/down counter *)
END_VAR
```

Example 2.12. Declaration of an up/down counter with IEC 61131-3

After this declaration the inputs and outputs of this counter can be accessed using a data structure defined implicitly by IEC 61131-3. In order to clarify this structure Example 2.13 shows it in an alternative representation.

```
TYPE CTUD :    (* data structure of an FB instance of FB type CTUD *)
  STRUCT
    (* inputs *)
    CU :        BOOL;    (* count up *)
    CD :        BOOL;    (* count down *)
    R  :        BOOL;    (* reset *)
    LD :        BOOL;    (* load *)
    PV :        INT;     (* preset value *)
    (* outputs *)
    QU :        BOOL;    (* output up *)
    QD :        BOOL;    (* output down *)
    CV :        INT;     (* current value *)
  END_STRUCT;
END_TYPE
```

Example 2.13. Alternative representation of the data structure of the up/down counter (standard FB) in Example 2.12

The data structure in Example 2.13 shows the formal parameters (calling interface) and return values of the standard FB CTUD. It represents the caller's view of the FB. Local or external variables of the POU are kept hidden.

   This data structure is managed automatically by the programming or run-time system and is easy to use for assigning parameters to FBs, as shown in Example 2.14 in the programming language IL:

```
LD    34
ST    Counter.PV    (* preset count value *)
LD    %IX7.1
ST    Counter.CU    (* count up *)
LD    %M3.4
ST    Counter.R     (* reset counter *)
CAL   Counter       (* invocation of FB with actual parameters *)
LD    Counter.CV    (* get current count value *)
```

Example 2.14. Parameterisation and invocation of the up/down counter in Example 2.12

In this example the instance Counter is assigned the parameters 34, %IX7.1 and %M3.4, before Counter is called by means of the instruction CAL (shown here in bold type). The current counter value can then be read.

As seen in Example 2.14, the inputs and outputs of the FB are accessed using the FB instance name and a separating period. This procedure is also used for structure elements (see Section 3.5.2).

Unused input or output parameters are given initial values that can be defined within the FB itself.

In Section. 4.1.4 further methods of calling FBs in IL by means of their instance names are shown.

Instance means "memory".
When several variable names are declared for the same FB type a sort of "FB data copy" is created for each instance in the PLC memory. These copies contain the values of the local (VAR) and the input or output variables (VAR_INPUT, VAR_OUTPUT), but not the values for VAR_IN_OUT (these are only pointers to variables, not the values themselves) or VAR_EXTERNAL (these are global variables).

This means that the instance can store local data values and input and output parameters over several invocations, i.e. it has a kind of "memory". Such a memory is important for FBs such as flip-flops or counters, as their behaviour is dependent on the current *status* of their flags and counter values respectively.

All variables of this memory are stored in a memory area which is *firmly* assigned to this one FB instance (by declaration). This memory area must therefore be *static*. This also means that the stack cannot be used in the usual way to manage local *temporary* variables!

Particularly in the case of function blocks which handle large data areas such as tables or arrays, this can lead to (unnecessarily) large static memory requirements for FB instances.

Therefore, in addition to the static data area provided by "VAR ... END_VAR", programming systems sometimes use dynamic data areas declared with "VAR_DYN ... END_VAR". These can be assigned to the stack mechanism on the PLC in order to save memory. By using such dynamic data PLC programmers may optimise the memory needs of their temporary data.

Furthermore, large numbers of input and output parameters can lead to memory-consuming FB instances. The use of VAR_IN_OUT instead of VAR_INPUT and VAR_OUTPUT respectively can help reduce memory requirements.

In Section 2.3.2 the read/write restrictions on the input and output variables of POUs were detailed. This is of particular importance for FB instances:

- Input parameters (formal parameters) of an FB instance maintain their values until the next invocation. If the called FB could change its own input variables, these values would be incorrect at the next call of the FB instance, and this would not be detected by the calling POU.

- Similarly, output parameters (return values) of an FB instance maintain their values between calls. Allowing the calling POU to alter these values would result in the called FB making incorrect assumptions about the status of its own outputs.

Like normal variables, FB instances can also be made retentive by using the keyword RETAIN, i.e. they maintain their local status information and the values of their calling interface during power failure.

Finally, the relationship between FB instances and conventional data blocks (DB) will be explained.

Relationship between FB instances and data blocks.

Before calling a conventional FB, which has no local data memory (besides formal parameters), it is common practice to activate a data block containing, for example, recipe or FB-specific data. Within the FB the data block can also serve as a local data memory area. This means that programmers can use a conventional FB with individual "instance data", but have to ensure the unambiguous assignment of the data to the FB themselves. This data is also retained between FB calls, because the data block is a global "shared memory area", as shown in Example 2.15:

```
                                    VAR_GLOBAL
   JU     DB 14    (* global DB *)    FB_14 : FB_Ex;  (* global instance *)
                                    END_VAR
   JU     FB 14    (* FB call *)     CAL  FB_14        (* invocation of FB instance*)
   ...                               ...
   a) Conventional DB/FB pair        b) FB instance in IEC 61131-3
```

Example 2.15. The use of a conventional DB/FB pair is similar to an FB instance as defined in IEC 61131-3.

This topic will be discussed in more detail in Section 7.8.

This type of instantiation is restricted to function blocks and is not applicable to functions (FUNCTION).

Programs are similarly instantiated and called as instances in the Configuration as the highest level of the POU hierarchy. But this (more powerful) kind of instance differs from that for FBs, in that it leads to the creation of run-time programs by association with different tasks. This will be described in Chapter 6.

2.4.2 Re-usable and object-oriented FBs

Function blocks are subject to certain restrictions, which make them re-usable in PLC programs:

- The declaration of variables with fixed assignment to PLC hardware addresses (see also Chapter 3: "directly represented variables": %Q, %I, %M) as "local" variables is not permitted in function blocks. This ensures that FBs are independent of specific hardware. The usage of PLC addresses as global variables in VAR_EXTERNAL is, however, not affected.
- The declaration of access paths of variable type VAR_ACCESS (see also Chapter 3) or global variables with VAR_GLOBAL is also not permitted within FBs. Global data, and thus indirectly access paths, can be accessed by means of VAR_EXTERNAL.
- External data can only be passed to the FB by means of the POU interface using parameters and external variables. There is no "inheritance", as in some other programming languages.

As a result of these features, function blocks are also referred to as *encapsulated*, which indicates that they can be used universally and are free from unwelcome side effects during execution - an important property for parts of PLC programs. Local FB data and therefore the FB function do not directly rely on global variables, I/O or system-wide communication paths. FBs can manipulate such data areas only indirectly via their (well-documented) interface.

The FB instance model with the properties of "structure" and "memory" was introduced in the previous section. Together with the property of encapsulation for re-usability a very new view of function blocks appears. This can be summarised as follows:

> "A function block is an independent, encapsulated data structure
> with a defined algorithm working on this data."

The algorithm is represented by the code part of the FB. The data structure corresponds to the FB instance and can be "called", something which is not possible with normal data structures. From each FB type any number of instances can be derived, each independent of the other. Each instance has a unique name with its own data area.

Because of this, IEC 61131-3 considers function blocks to be *object-oriented*. These features should not, however, be confused with those of today's modern "object-oriented programming languages (→OOP)" such as, for example, C++ with its class hierarchy!

To summarise, FBs work on their *own data area* containing input, output and local variables. In previous PLC programming systems FBs usually worked on *global data areas* such as flags, shared memory, I/O and data blocks.

2.4.3 Types of variables in FBs

A function block can have any number of input and output parameters, or even none at all, and can use local as well as external variables.

In addition or as an alternative to making a whole FB instance retentive, local or output variables can also be declared as retentive *within* the declaration part of the FB.

The values of input or input/output parameters cannot be declared retentive in the FB declaration part (RETAIN) as these are passed by the calling POU and have to be declared retentive there.

For VAR_IN_OUT it should be noted that the **pointers** to variables can be declared retentive in an instance using the qualifier RETAIN. The corresponding **values** themselves can, however, be lost if they are not also declared as retentive in the calling POU.

Due to the necessary hardware-independence, directly represented variables (I/Os) may not be declared as local variables in FBs, such variables may only be "imported" as global variables using VAR_EXTERNAL.

One special feature of variable declaration in IEC 61131-3 are the so-called edge-triggered parameters. The standard provides the standard FBs R_TRIG and F_TRIG for rising and falling edge detection (see also Chapter 5).

The use of edge detection as an attribute of variable types is only possible for input variables (see Section 3.5.4).

FBs are required for the implementation of some typical basic PLC functions, such as timers and counters, as these must maintain their status information (instance data). IEC 61131-3 defines several standard FBs that will be described in more detail and with examples in Chapter 5

2.5 The Function

The basic idea of a function (FUN) as defined by IEC 61131-3 is that the instructions in the body of a function that are performed on the values of the input variables result in an *unambiguous* function value (free from side effects). In this sense functions can be seen as manufacturer- or application-specific extensions of the PLC's set of operations.

The following simple rule is valid for functions: the same input values *always* result in the same function (return) value. This is independent of how often or at what time the function is called. Unlike FBs, functions do not have a memory.

Functions can be used as IL operators (instructions) as well as operands in ST expressions. Like FB types, but unlike FB instances, functions are also accessible project-wide, i.e. known to all POUs of a PLC project.

For the purpose of simplifying and unifying the basic functionality of a PLC system, IEC 61131-3 predefines a set of frequently used standard functions, whose features, run-time behaviour and calling interface are standardised (see also Chapter 5).

With the help of user-defined functions this collection can be extended to include device-specific extensions or application-specific libraries.

A detailed example of a function can be found in Appendix C. Functions have several restrictions in comparison to other POU types. These restrictions are necessary to ensure that the functions are truly independent (free of any side effects) and to allow for the use of functions within expressions, e.g. in ST. This will be dealt with in detail in the following section.

2.5.1 Types of variables in functions and the function value

Functions have one or any number of input parameters. As opposed to FBs, they do not have output parameters but return exactly *one element* as the function (return) value.

The function value can be of any data type, including derived data types. Thus a simple Boolean value or a floating-point double word is just as valid as an array or a complex data structure consisting of several data elements (multi-element variable), as described in Chapter 3.

Each programming language of IEC 61131-3 uses the function name as a special variable within the function body in order to explicitly assign a function value.

As functions always return the same result when provided with the same input parameters, they may not store temporary results, status information or internal data between their invocations, i.e. they operate "without memory".

Functions can use local variables for intermediate results, but these will be lost when terminating the function. Local variables can therefore not be declared as retentive.

Functions may not call function blocks such as timers, counters or edge detectors. Furthermore, the use of global variables within functions is not permitted.

The standard does not stipulate how a PLC system should treat functions and the current values of their variables after a power failure. The POU that calls the function is therefore responsible for backing up variables where necessary. In any case it makes sense to use FBs instead of functions if important data is being processed.

Furthermore, in functions (as in FBs) the declaration of directly represented variables (I/O addresses) is not permitted.

2.5.2 Execution control with EN and ENO

In LD and FBD, functions have a special feature not used in the other programming languages of IEC 61131-3: here the functions possess an additional input and output. These are the Boolean input EN (Enable In) and the Boolean output ENO (Enable Out).
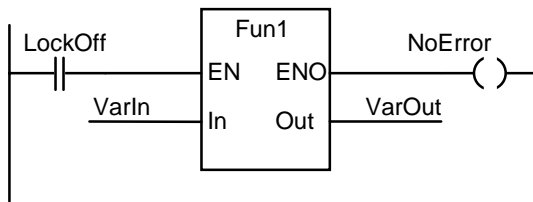
Example 2.16. Graphical invocation of a function with EN/ENO in LD

Example 2.16 shows the graphical representation for calling function Fun1 with EN and ENO in LD. In this example Fun1 will only be executed if input EN has the value logical "1" (TRUE), i.e. contact Lockoff is closed. After error-free execution of the function the output ENO is similarly "1" (TRUE) and the variable NoError remains set.

With the aid of the EN/ENO pair it is possible to at least partially integrate any function, even those whose inputs or function value are not Boolean, like Fun1 in Example 2.16, into the "power flow". The meaning of EN/ENO based on this concept is summarised in Table 2.6:

EN	Explanation[a]	ENO
EN = FALSE	If EN is FALSE when calling the function, the code-part of the function may not be executed. In this case output ENO will be set to FALSE upon exiting in order to indicate that the function has not been executed.	ENO = FALSE
EN = TRUE	If EN is TRUE when calling the function, the code-part of the function can be executed normally. In this case ENO will initially be set to TRUE **before** starting the execution.	ENO = TRUE
	ENO can afterwards be set to TRUE or FALSE by instructions executed within the function body.	ENO = individual value
	If a program or system error (as described in Appendix E) occurs while executing the function ENO will be reset to FALSE by the PLC.	ENO = FALSE (error occurred)

a   TRUE = logical "1", FALSE = logical "0"

Table 2.6. Meaning of EN and ENO within functions

As can been seen from Table 2.6, EN and ENO determine the *control flow* in a graphical network by means of conditional function execution and error handling in case of abnormal termination. EN can be connected not only to a single contact as in Example 2.16, but also with a sub-network of several contacts, thus setting a complex precondition. ENO can be similarly be evaluated by a more complex sub-network (e.g. contacts, coils and functions). These control flow operations should however be logically distinguished from other LD/FBD operations that represent the *data flow* of the PLC program.

These special inputs/outputs EN and ENO are not treated as normal function inputs and outputs by IEC 61131-3, but are reserved only for the tasks described above. At present IEC 61131-3 does not use or refer to EN/ENO in LD or FBD in function blocks, but this will probably be changed in the near future

   The use of these additional inputs and outputs is not included in the other IEC 61131-3 programming languages. In FBD the representation of EN/ENO is allowed as an additional feature.

   The function call in Example 2.16 can be represented in IL if the programming system supports EN/ENO as implicit system variables.

If a programming system supports the usage of EN and ENO, it is difficult to convert POUs programmed with these into textual form. In order to make this possible, EN/ENO would also have to be keywords in IL or ST and would need to be automatically generated there, as they are in LD/FBD. Then a function called in LD could be written in IL and could, for example, set the ENO flag in case of an error. Otherwise only functions written in LD/FBD can be used in LD/FBD programs. The standard, however, does not make any statement about how to use

EN and ENO as keywords and graphical elements in LD/FBD in order to set and reset them.

On the other hand, it is questionable whether the usage of EN and ENO is advantageous in comparison functions (std. FUN, see also Appendix A). A comparison function then has two Boolean outputs, each of which can be connected with a coil. If this comparison is used within a parallel branch of an LD network, ENO and the output Q have to be connected separately: ENO continues the parallel branch while Q, in a sense, opens a new sub-network.

Because of this complexity only some of today's IEC programming systems use EN/ENO. Instead of dictating the Boolean pair EN/ENO in LD/FBD there are other conceivable alternatives:

- EN and ENO can be used both implicitly and explicitly in all programming languages,
- Each function which can be called in LD/FBD must have at least one binary input and output respectively,
- Only standard functions have an EN/ENO pair (for error handling within the PLC system). This pair may not be used for user-defined functions.

The third alternative is the nearest to the definition in IEC 61131-3. This would, however, mean that EN and ENO are PLC system variables, which cannot be manipulated by the PLC programmer.

2.6 The Program

Functions and function blocks constitute "subroutines", whereas POUs of type PROGRAM build the PLC's "main program". On multitasking-capable controller hardware several main programs can be executed simultaneously. Therefore PROGRAMs have special features compared to FBs. These features will be explained in this section.

In addition to the features of FBs, a PLC programmer can use the following features in a PROGRAM:

- Declaration of directly represented variables to access the physical I/O addresses of the PLC (%Q, %I, %M) is allowed,
- Usage of VAR_ACCESS or VAR_GLOBAL is possible,
- A PROGRAM is associated with a task within the configuration, in order to form a run-time program, i.e. programs are not called explicitly by other POUs.

Variables can be assigned to the PLC I/Os in a PROGRAM by using directly represented or symbolic variables as global or POU parameters.

Furthermore programs describe the mechanisms by which communication and global data exchange to other programs take place (inside and outside the configuration). The variable type VAR_ACCESS is used for this purpose.

These features can also be used at the resource and configuration levels. This is, in fact, to be recommended for complex PLC projects.

Because of the wide functionality of the POU PROGRAM it is possible, in smaller projects, to work without a configuration definition: the PROGRAM takes over the task of assigning the program to PLC hardware.

Such possibilities depend on the functionality of a programming system and will not be dealt with any further here.

A detailed example of a PROGRAM can be found in Appendix C.

The run-time properties and special treatment of a PROGRAM in the CPU are expressed by associating the PROGRAM with TASKs. The program is instantiated, allowing it to be assigned to more than one task and to be executed several times simultaneously within the PLC. This instantiation differs, however from that for FB instances.

The assignment of programs to tasks is done in the CONFIGURATION and is explained in Chapter 6.

2.7 Calling Functions and Function Blocks

In this section we will deal with the special features which have to be considered when calling functions and function blocks. These features apply to standard as well as user-defined functions and function blocks.

The following examples will be given in IL. The use of ST and graphical representation in LD and FBD are topics of Chapter 4.

2.7.1 Mutual calls of POUs

The following rules, visualised in Figure 2.7, can be applied to the mutual calling of POU types:

- PROGRAM may call FUNCTION_BLOCK and FUNCTION, but not the other way round,
- FUNCTION_BLOCK may call FUNCTION_BLOCK,
- FUNCTION_BLOCK may call FUNCTION, but not the other way round,
- Calls of POUs may not be recursive, i.e. a POU may not call (an instance of) itself either directly or indirectly.
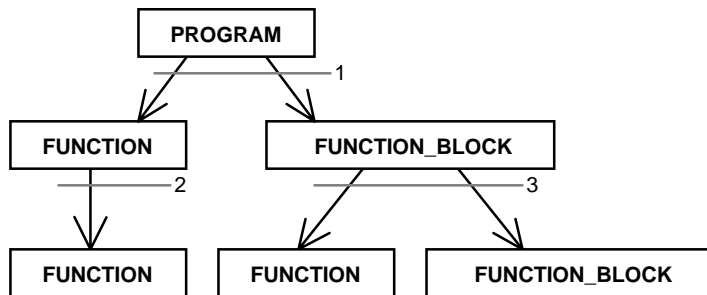
1   Program calls function or function block

2   Function calls function

3   Function block calls function or function block

Figure 2.7. The three possible ways of invocation among the POU types

Programs and FB instances may call FB instances. Functions, on the other hand, may not call FB instances, as otherwise the independence (freedom from side effects) of functions could not be guaranteed.

Programs (PROGRAM) are instantiated to form run-time programs within the Configuration by association with a TASK. They are then called by the Resource.

2.7.2 Recursive calls are forbidden

IEC 1131-3 clearly defines that POUs may not call themselves (*recursion*) either directly or indirectly, i.e. a POU may not call a POU instance of the same type and/or name. This would mean that a POU could "define itself" by using its own name in its declaration or calling itself within its own body. Recursion is, however, usually permitted in other programming languages in the PC world.

If recursion were allowed, it would not be possible for the programming system to calculate the maximum memory space needed by a recursive PLC program at run time.

Recursion can always be replaced by corresponding iterative constructs, i.e. by building program loops.

Both the following figures show examples of invalid calls:

```
FUNCTION  Fun1 : BOOL                    FUNCTION  Fun1 : BOOL

                                         VAR
INT ──┤Par1                                Par1, Par2 : INT;
INT ──┤Par2   ├── BOOL                   END_VAR


        Fun1
                                         LD    Par1
Par1──┤Par1                              Fun1  Par2
Par2──┤Par2   ├── Fun1                   ST    Fun1


END_FUNCTION                             END_FUNCTION
```

Example 2.17. Invalid recursive call of a function in graphical and IL representation: nested invocation.

In Example 2.17 the same function is called again within function Fun1.

   The top half of this example shows the declaration part of the function (input variables Par1 and Par2 of data type INT and function value of type BOOL).

   In the bottom part, this function is called with the same input variables so that there would be an endless (recursive) chain of calls at run time.

```
FUNCTION_BLOCK  FunBst
VAR_INPUT
 In1   :   DINT;                         (* input variable *)
END_VAR
VAR
 InstFunBst   : FunBst;            (* improper instance of the same type *)
 Var1  :   DINT;                         (* local variable *)
END_VAR
...
  CALC   InstFunBst (In1 := Var1);    (* illegal recursive invocation! *)
...
END_FUNCTION_BLOCK
```

Example 2.18. Forbidden recursive call of an FB in IL: nesting already in declaration part.

Example 2.18 shows function block FunBst, in whose local variable declaration (VAR) an instance of its own *type* (FunBst) is declared. This instance is called in the function body. This would result in endlessly deep nesting when instantiating the FB in the declaration, and the memory space required for the instance at run time would be impossible to determine.

Programmers themselves or the programming/PLC system must check whether unintentional recursive calling exists in the PLC program.

This checking can be carried out when creating the program by means of a POU calling tree, as the illegal use of recursion applies to FB types and not to their instance names. This is even possible if FB instance names are used as input parameters (see also Section 2.7.5).

The following example shows how recursive calls can occur even if a function or FB instance does not directly call itself. It suffices if they mutually call each other.
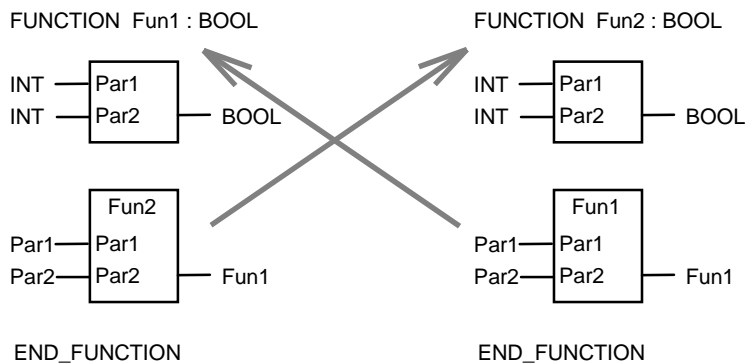
Example 2.19. Recursion by mutual invocation in graphical representation

Such types of recursion are, on principle, not permitted in IEC 61131-3. The calling condition may be defined as follows: if a POU is called by POU A, that POU and all the POUs below it in the calling hierarchy may not use the name of POU A (FB instance or function name).

Unlike most of the modern high-level languages (such as C), recursion is therefore prohibited by IEC 61131-3. This helps protect PLC programs against program errors caused by unintentional recursion.

2.7.3 Calling with formal parameters

When a FUN/FB is called, the input parameters are passed to the POU's input variables. These input variables are also called *formal parameters*, i.e. they are placeholders. The input parameters are known as *actual parameters* in order to express that they contain actual input values.

When calling a POU, the formal parameters may or may not be explicitly specified. This depends on the POU type (FUN or FB) and on the programming language used for the POU call (see also Chapter 4).

Table 2.7 gives a summary of which POU types can be called, in textual and graphical representation, with or without giving the formal parameter names.

Language	Function	Function block	Program
IL	without	with [a]	with
ST	with or without [b]	with	with
LD and FBD	with [b]	with	with

a   possible in three different ways, see Section 4.1.4
b   with std. FUN: if a parameter name exists

Table 2.7. Possible explicit specification of formal parameters ("with" or "without") in POU calls

In FBs and PROGs the formal parameters must always be specified explicitly, independently of the programming language. In IL there are different ways of doing this (see Section 4.1.4).

In ST functions can be called with or without specifying the names of the formal parameters.

Many formal parameters of standard functions do not have a name (see Appendix A). Therefore these cannot be displayed in graphical representation and cannot be explicitly specified in textual languages.

IEC 61131-3 does not state whether the names of formal parameters can be specified when calling user-defined functions in IL. However, in order to keep such function calls consistent with those of standard functions, it is assumed that the names of formal parameters may **not** be used with function calls in IL.

The same rules are valid for the calling of standard functions and standard function blocks. Example 2.20 shows examples for each POU type.

FB declaration:	FUN declaration:	PROG declaration:
FUNCTION_BLOCK **FBlk**	FUNCTION **Fctn** : INT	PROGRAM **Prgrm**
VAR_INPUT	VAR_INPUT	VAR_GLOBAL
Par1 : TIME;	Par1 : TIME;	**FunBlk** : FBlk;
Par2 : WORD;	Par2 : WORD;	VarGlob: INT;
Par3 : INT;	Par3 : INT;	
END_VAR	END_VAR	END_VAR
... (*instructions *)	... (*instructions *)	... (* instructions *)
END_FUNCTION_BLOCK	END_FUNCTION	END_PROGRAM

```
(* 1. Invocations in IL *)
LD            t#20:12
Fctn          %IW4, VarGlob                                              (* function call *)
CAL FunBlk    (Par1 := t#20:12, Par2 := %IW4, Par3 := VarGlob)   (* FB call *)
```

```
(* 2. Invocations in ST *)
Fctn          (t#20:12, %IW4, VarGlob)                                   (* function call *)
Fctn          (Par1 := t#20:12, Par2 := %IW4, Par3 := VarGlob); (* function call *)
FunBlk        (Par1 := t#20:12, Par2 := %IW4, Par3 := VarGlob); (* FB call *)
```

Example 2.20. Equivalent function and function block calls with and without explicit formal parameters in the textual languages IL and ST. In both cases the invocation (calling) is done in the program Prgrm.

In IL the first actual parameter is loaded as the current result (CR) before the invocation instruction (CAL) is given, as can be seen from the call of function Fctn in Example 2.20. When calling the function the other two parameters are specified separated by commas, the names of these formal parameters may not be included.

The two equivalent calls in ST can be written with or without the names of formal parameters. The input parameters are enclosed in brackets each time.

In the call of FB instance FunBlk in this example all three formal parameters are specified in full in both IL and ST.

The usage of formal and actual parameters in graphical representation is shown in Example 3.18.

2.7.4 Calls with input parameters omitted or in a different order

Functions and function blocks can be called even if the input parameter list is incomplete or not every parameter is assigned a value.

If input parameters are **omitted**, the names of the formal parameters that are used must be specified explicitly. This ensures that the programming system can assign the actual parameters to the correct formal parameters.

If the **order** of the parameters in a FUN/FB call is to be changed, it is also necessary to specify the formal parameter names explicitly. These situations are shown, as an example in IL, in Example 2.21.

```
(* 1.    complete FB call *)
CAL    FunBlk (Par1 := t#20:12, Par2 := %IW4, Par3 := VarGlob);

(* 2.    complete FB call with parameters in a changed order *)
CAL    FunBlk (Par2 := %IW4, Par1 := t#20:12, Par3 := VarGlob);

(* 3.    incomplete FB call *)
CAL    FunBlk (Par2 := %IW4);

(* 4.    incomplete FB call with parameters in a changed order *)
CAL    FunBlk (Par3 := VarGlob, Par1 := t#20:12);
```

Example 2.21. Examples of the FB call from Example 2.20 with parameters omitted and in a different order, written in IL

This means that either all formal parameters must be specified, and the parameter order is not relevant, or no formal parameters are used and the entries must appear in the correct order. The formal parameters always have to be specified when calling FBs, for functions this is, however, language-dependent (see Table 2.7).

Assignments to input variables can be omitted if the input variables are "initialised" in the declaration part of the POU. Instead of the actual parameter that is missing the initial value will then be used. If there is no user-defined initial value the default value for the standard data types of IEC 61131-3 will be used. This ensures that input variables **always** have values.

For FBs initialisation is performed only for the first call of an instance. After this the values from the last call still exist, because instance data (including input variables) is retained.

2.7.5 FB instances as actual FB parameters

This section describes the use of FB instance names as well as their inputs and outputs as actual parameters in the calling of other function blocks.

Using Example 2.22 this section explains what facilities IEC 61131-3 offers for indirect calling or indirect parameter assignment of FB instances.

```
FUNCTION_BLOCK  MainFB
  VAR_IN_OUT

    Time1 : TON;                        (* 1st instance of TON *)
    Time2 : TON;                        (* 2nd instance of TON *)

     InstFB : SubFB;                    (* Instance of SubFB *)

  END_VAR
  ...                                   (* FB instance call *)
  CAL  InstFB (  Timer :=  Time1,       (* FB instance name *)
                 TimeQ :=Time2.Q,       (* FB output *)
                 TimeIN :=TRUE)         (* FB input *)
  ...
  LD    InstFB.Time3.Q                  (* Loading the FB output *)
  ...
  CAL  Time1                            (* Invocation of on-delay timer*)
```

Example 2.22. Using the FB instance name Time1 and the output parameter Time2.Q as actual parameters of another FB. The timers Time1 and Time2 are instances of the standard FB TON (on-delay, see Chapter 5). SubFB will be declared in Example 2.23.

Instance names and the inputs and outputs of instances can be used as the actual parameters for input or input/output variables. Table 2.8 shows the cases where this is allowed (in table: "yes") and not allowed ("no").

		As actual parameter for **SubFB**		Return value, External variable
Instance	Example	VAR_INPUT	VAR_IN_OUT (pointer)	VAR_EXTERNAL VAR_OUTPUT
Instance name	Time1	yes [a]	yes [b]	yes [c]
-input	Time2.IN	-	-	-
-output	Time2.Q	yes	no [d]	-

a   Instance may not be called within SubFB (indirect FB call not possible)
b   Indirect FB call is possible, output of instance may not be changed within SubFB
c   Direct FB call, output of instance may not be changed within MainFB
d   The function (return) value of a function cannot be used either

Table 2.8. Possible cases for using or indirectly calling FB instances as actual parameters of FBs. The "Example" column refers to Example 2.22. The last column shows that FB instances may also be used as external variables or as return values. Time2.IN may not be used for read access and cannot therefore be passed as a parameter.

As this summary shows, only certain combinations of function block instance names and their inputs and outputs can be passed as actual parameters to function blocks for each of the variable types.

VAR_INPUT: FB instances and their outputs cannot be called or altered within SubFB if they are passed as VAR_INPUT. They may, however, be read.

VAR_IN_OUT: The output of an FB instance, whose pointer would be used here, is not allowed as a parameter for this variable type. An erroneous manipulation of this output can thus be prevented. Similarly, a pointer to the function value of a function is **not** allowed as a parameter for a VAR_IN_OUT variable.

The instance passed as a parameter can then be called, thereby implementing an *indirect FB call*.

The outputs of the FB instance that has been passed may not be written to. FB instance inputs may, however, be freely accessed.

VAR_EXTERNAL, VAR_OUTPUT: FB instances are called directly, their inputs and outputs may only be read by the calling POU.

Example of an indirect FB call.

Example 2.23 shows (together with Example 2.22) the use of some cases permitted in Table 2.8 within function block SubFB.

```
FUNCTION_BLOCK  SubFB
  VAR_INPUT
    TimeIN :  BOOL;        (* Boolean input variable *)
    TimeQ  :  BOOL;        (* Boolean input variable *)
  END_VAR
  VAR_IN_OUT
    Timer  :  TON;         (* pointer to instance Time1 of TON – input/output variable *)
  END_VAR
  VAR_OUTPUT
    Time3  :  TON;         (* 3rd instance of TON *)
  END_VAR
  VAR
    Start : BOOL := TRUE; (* local Boolean variable *)
  END_VAR
  ...
```

```
(* Indirect call of Time1 setting/checking the actual parameter values using Timer *)
LD     Start
ST     Timer.IN        (* starting of Timer Time1 *)
CAL    Timer           (* calling the on-delay timer Time 1 indirectly *)
LD     Timer.Q         (* checking the output of Time1 *)
```

```
...
```

```
(* Direct call of Time3; indirect access to Time2 *)
LD     TimeIN          (*indirect checking of the input of Time2 is not possible *)
ST     Time3.IN        (* starting the timer using Time3.IN *)
CAL    Time3           (* calling the on-delay timer Time3 directly *)
LD     Time3.Q         (*checking the output using Time3.Q *)
...
LD     TimeQ           (*indirectly checking the output of Time 2 *)
```

```
...
END_FUNCTION_BLOCK
```

Example 2.23. Alternative ways of calling the on-delay FB Time1 from Example 2.22 indirectly and usage of its inputs and outputs

This example shows the *indirect call* of FB Time1, whose instance name was passed to FB SubFB as an input/output variable in Example 2.22. The function block SubFB is only assigned the FB instance name Time1 at the run time of MainFB. In SubFB Time1 (as input variable Timer) is provided with the parameter Timer.IN and then called.

As shown with Example 2.23, it is also possible to access the inputs and outputs of an FB passed as an instance name. Here the instance inputs (Timer.IN) can be read and written to, whereas the outputs (as Timer.Q) can only be read.

The FB instance Time3 in this example serves as a comparison between the treatment of input parameters and of the return values of an FB as output variables.

FB instance names as actual parameters of functions.
Instance names (such as Time1) and components of an FB instance (such as Time2.Q) can also be used as actual parameters for functions. Initially this appears to be inconsistent with the requirement that functions have to produce the same result when supplied with the same inputs and that they may not call FBs.

This is, however, not as contradictory as it seems: the FB instance passed as a parameter is not **called**, but its input and output variables are treated like elements of an normal data structure, see also Section 2.4.1.

Function values as actual parameters.
Functions and function values may also be used as actual parameters for functions and function blocks. The input variables have the same data type as the function and are assigned the function value when called.

IEC 61131-3 does not give any explicit instructions about this possibility, thus making it implementation-dependent.

2.8 Summary of POU Features

The following table summarises all the essential POU features that have been presented and discussed in this chapter.

Feature	Function	Function Block	Program
Input parameter	yes	yes	yes
Output parameter	no	yes	yes
Input/output parameter	no	yes	yes
Function value	yes	no	no
Invocation of functions	yes	yes	yes
Invocation of function blocks	no	yes	yes
Invocation of programs	no	no	no
Declaration of global variables	no	no	yes
Access to external variables	no	yes	yes
Declaration of directly represented variables [a]	no	no	yes
Declaration of local variables	yes	yes	yes
Declaration of FB instances	no	yes	yes
Overloading and extension [b]	yes	no	no
Edge detection possible	no	yes	yes
Usage of EN/ENO [c]	yes	no	no
Retention of local and output variables	no	yes	yes
Indirect FB call	no	yes	yes
Usage of function values as input parameters [d]	yes	yes	yes
Usage of FB instances as input parameters	yes	yes	yes
Recursive invocation	no	no	no

a   for function blocks only with VAR_EXTERNAL
b   for standard functions
c   also planned for function blocks
d   not in IL, otherwise: implementation-dependent

Table 2.9. An overview of the POU features summarising the important topics of this chapter. The entries "yes" or "no" mean "permitted" and "not permitted" for the corresponding POU type respectively.

3 Variables, Data Types and Common Elements

Contents: see paper version.

4 The New Programming Languages of IEC 61131-3

Contents: see paper version.

5 Standardised PLC Functionality

The IEC not only standardises the syntax of programming languages, but even goes a step further to unify the implementation of typical PLC functions, such as timers, counters or special arithmetic operations.

The standard does this by defining typical PLC functions and function blocks and describing their behaviour exactly. These elements are known as *standard functions* and *standard function blocks* respectively. Their names are reserved keywords.

If the functions and function blocks in the programming systems and block libraries of different manufacturers are given the names specified in the standard, they must comply with the rules set out in the standard. Manufacturers can also offer additional PLC functions which, for example, support particular hardware properties or other characteristics of a PLC system.

The definition of an unambiguous standard for PLC functions is an essential requirement for uniform, manufacturer- and project-independent training, programming and documentation.

This chapter gives an overview of the most important standard functions and function blocks as well as the concepts used within them:

1) **Standard functions (std. FUN)**
   - Calling interface
   - Extensibility
   - Overloading
   - Examples
2) **Standard function blocks (std. FB)**
   - Calling interface
   - Examples

The standard functions correspond to the basic logical operators used in conventional PLC systems (addition, bit-shifting, comparison etc.), whereas the standard function blocks are responsible for PLC functions with status information, such as timers, counters, R/S flipflops and edge detectors.

In the following sections, the calling interfaces (input and output variables and the function [return] value) for standard functions and function blocks are described in detail. Practical examples accompany the various descriptions.

The graphical declarations of all the standard functions and function blocks, together with a short functional description, are given in Appendices A and B.

The general usage of functions and function blocks has already been discussed in Chapter 2 and the properties of their formal parameters in Chapter 3.

5.1 Standard Functions

IEC 61131-3 defines the following eight groups of standard functions:

1) Data type conversion functions,
2) Numerical functions,
3) Arithmetic functions,
4) Bit-string functions (bit-shift and bitwise Boolean functions),
5) Selection and comparison functions,
6) Character string functions,
7) Functions for time data types,
8) Functions for enumerated data types.

Table 5.1 summarises all the standard functions of the standard. The special functions for time data types (ADD, SUB, MUL, DIV, CONCAT) and neumerated data types (SEL, MUX, EQ, NE) are grouped together with the other functions in the categories Arithmetic, Comparison, Selection and Character string.

The table gives the function name, the data type of the function values and a short description of the function. Together with Table 5.2, it also gives the names and data types of the input variables.

With the exception of the generic data types, the abbreviations for the data types of the input variables and function values in Table 5.1 are listed in Table 5.2. These abbreviations correspond to the names of the input variables used by IEC 61131-3 for the respective standard functions. ENUM is an additional abbreviation used to make Table 5.1 clearer.

Standard functions (with data types of input variables)	Data type of function value	Short description	over-loa-ded	exten-sible
Type conversion				
*_TO_** (ANY)	ANY	Data type conversion	yes	no
TRUNC (ANY_REAL)	ANY_INT	Rounding up/down	yes	no
BCD_TO_** (ANY_BIT)	ANY	Conversion from BCD	yes	no
*_TO_BCD (ANY_INT)	ANY_BIT	Conversion to BCD	yes	no
DATE_AND_TIME_TO_- **TIME_OF_DAY** (DT)	TOD	Conversion to time-of-day	no	no
DATE_AND_TIME_TO_- **DATE** (DT)	DATE	Conversion to date	no	no
Numerical				
ABS (ANY_NUM)	ANY_NUM	Absolute number	yes	no
SQRT (ANY_REAL)	ANY_REAL	Square root (base 2)	yes	no
LN (ANY_REAL)	ANY_REAL	Natural logarithm	yes	no
LOG (ANY_REAL)	ANY_REAL	Logarithm to base 10	yes	no
EXP (ANY_REAL)	ANY_REAL	Exponentiation	yes	no
SIN (ANY_REAL)	ANY_REAL	Sine	yes	no
COS (ANY_REAL)	ANY_REAL	Cosine	yes	no
TAN (ANY_REAL)	ANY_REAL	Tangent	yes	no
ASIN (ANY_REAL)	ANY_REAL	Arc sine	yes	no
ACOS (ANY_REAL)	ANY_REAL	Arc cosine	yes	no
ATAN (ANY_REAL)	ANY_REAL	Arc tangent	yes	no
Arithmetic *(IN1, IN2)*				
ADD {+} (ANY_NUM, ANY_NUM)	ANY_NUM	Addition	yes	yes
ADD {+} [a] (TIME, TIME)	TIME	Time addition	yes	no
ADD {+} [a] (TOD, TIME)	TOD	Time-of-day addition	yes	no
ADD {+} [a] (DT, TIME)	DT	Date addition	yes	no
MUL {*} (ANY_NUM, ANY_NUM)	ANY_NUM	Multiplication	yes	yes
MUL {*} [a] (TIME, ANY_NUM)	TIME	Time multiplication	yes	no
SUB {-} (ANY_NUM, ANY_NUM)	ANY_NUM	Subtraction	yes	no
SUB {-} [a] (TIME, TIME)	TIME	Time subtraction	yes	no
SUB {-} [a] (DATE, DATE)	TIME	Date subtraction	yes	no
SUB {-} [a] (TOD, TIME)	TOD	Time-of-day subtraction	yes	no
SUB {-} [a] (TOD, TOD)	TIME	Time-of-day subtraction	yes	no
SUB {-} [a] (DT, TIME)	DT	Date and time subtraction	yes	no
SUB {-} [a] (DT, DT)	TIME	Date and time subtraction	yes	no
DIV {/} (ANY_NUM, ANY_NUM)	ANY_NUM	Division	yes	no
DIV {/} [a] (TIME, ANY_NUM)	TIME	Time division	yes	no
MOD (ANY_NUM, ANY_NUM)	ANY_NUM	Remainder (modulo)	yes	no
EXPT {**} (ANY_NUM, ANY_NUM)	ANY_NUM	Exponent	yes	no
MOVE {:=}(ANY_NUM, ANY_NUM)	ANY_NUM	Assignment	yes	no

a   Special function for time data type

Table 5.1. An overview of the standard functions (continued on next page)

Standard functions (with data types of input variables)	Data type of function value	Short description	over-loa-ded	exten-sible
Bit-shift *(IN1, N)*				
SHL (ANY_BIT, N)	ANY_BIT	Shift left	yes	no
SHR (ANY_BIT, N)	ANY_BIT	Shift right	yes	no
ROR (ANY_BIT, N)	ANY_BIT	Rotate right	yes	no
ROL (ANY_BIT, N)	ANY_BIT	Rotate left	yes	no
Bitwise *(IN1, IN2)*				
AND {&} (ANY_BIT, ANY_BIT)	ANY_BIT	Bitwise AND	yes	yes
OR {>=1} (ANY_BIT, ANY_BIT)	ANY_BIT	Bitwise OR	yes	yes
XOR {=2k+1} (ANY_BIT, ANY_BIT)	ANY_BIT	Bitwise EXOR	yes	yes
NOT (ANY_BIT, ANY_BIT)	ANY_BIT	Bitwise inverting	yes	no
Selection *(IN1, IN2)*				
SEL (G, ANY, ANY)	ANY	Binary selection (1 of 2)	yes	no
SEL [b] (G, ENUM, ENUM)	ENUM	Binary selection (1 of 2)	no	no
MAX (ANY, ANY)	ANY	Maximum	yes	yes
MIN (ANY, ANY)	ANY	Minimum	yes	yes
LIMIT (MN, ANY, MX)	ANY	Limitation	yes	no
MUX (K, ANY, ..., ANY)	ANY	Multiplexer (select 1 of N)	yes	yes
MUX [b] (K, ENUM, ..., ENUM)	ENUM	Multiplexer (select 1 of N)	no	no
Comparison *(IN1, IN2)*				
GT {>} (ANY, ANY)	BOOL	Greater than	yes	yes
GE {>=} (ANY, ANY)	BOOL	Greater than or equal to	yes	yes
EQ {=} (ANY, ANY)	BOOL	Equal to	yes	yes
EQ {=} [b] (ENUM, ENUM)	BOOL	Equal to	no	no
LT {<} (ANY, ANY)	BOOL	Less than	yes	yes
LE {<=} (ANY, ANY)	BOOL	Less than or equal to	yes	yes
NE {<>} (ANY, ANY)	BOOL	Not equal to	yes	no
NE {<>} [b] (ENUM, ENUM)	BOOL	Not equal to	no	no
Character string *(IN1, IN2)*				
LEN (STRING)	INT	Length of string	no	no
LEFT (STRING, L)	STRING	string "left of"	yes	no
RIGHT (STRING, L)	STRING	string "right of"	yes	no
MID (STRING, L, P)	STRING	string "from the middle"	yes	no
CONCAT (STRING, STRING)	STRING	Concatenation	no	yes
CONCAT [a] (DATE, TOD)	DT	Time concatenation	no	no
INSERT (STRING, STRING, P)	STRING	Insertion (into)	yes	yes
DELETE (STRING, L, P)	STRING	Deletion (within)	yes	yes
REPLACE (STRING, STRING, L, P)	STRING	Replacement (within)	Yes	yes
FIND (STRING, STRING)	INT	Find position	Yes	yes

a   Special function for time data type
b   Special function for enumeration data type

Table 5.1. (continued)

Input	Meaning	Data type
N	Number of bits to be shifted	UINT
L	Left position within character string	UINT
P	Position within character string	UINT
G	Selection out of 2 inputs (gate)	BOOL
K	Selection out of n inputs	ANY_INT
MN	Minimum value for limitation	ANY
MX	Maximum value for limitation	ANY
ENUM	Data type of enumeration	

Table 5.2. Abbreviations and meanings of the input variables in Table 5.1

The function names are listed on the left-hand side of column 1 in Table 5.1. The meanings of the asterisks (*) in the function names of the "Type conversion" group are as follows (see also Appendix A):

*   Data type of the input variable (right hand side of column 1)
** Data type of the function value (column 2)

The names of the input variables of a function, if any, are given in the italic heading of the group, if they apply to the group as a whole. For example, the input variables for arithmetic functions are named IN1, IN2 and, when extended, IN3, IN4, ... If a function only has a single input variable, this does not have a name.

   If a function with several inputs has only one input of data type ANY (overloaded), its variable name is "IN" (without number). This applies to LIMIT, LEFT, RIGHT, MID and DELETE.

   Within IEC 61131-3, SEL is an exception to this uniformity. The inputs of this function are called G, IN0 and IN1 (instead of IN1, IN2).

Some standard functions have an alternative function name consisting of symbols in their graphical representation, as shown in curved brackets directly after the function name in Table 5.1. For example, the addition function can be called as ADD (operator in IL) or as "+" (graphical symbol in LD/FBD or within ST expressions).

5.1.1 Overloaded and extensible functions

The data types of the input variables are given in round brackets next to the function names in Table 5.1. Here generic data types, already introduced in Table 3.9, are also given for reasons of clarity. Each function whose input variable is described using a generic data type is called *overloaded* and has a "yes" in the corresponding column in Table 5.1. This simply means that the function is not restricted to a single data type for its input variables, but can be applied to different data types.

The data type of the function value (2nd column) is normally the same as the data type of its inputs. Exceptions are functions such as LEN, which expects a character string as its input but returns INT as its function value.

If a standard function can have a variable number of inputs (2, 3, 4,...), it is called *extensible*. Such functions have a "yes" in the corresponding column in Table 5.1.

No formal parameters have to be entered when calling extensible functions. In textual languages they are called simply by using actual parameters separated by commas – in graphical representation the parameter names inside the boxes are omitted.

In IEC 61131-3 these properties are **not** applied to user-defined functions, but can also be extended to these functions (and other POU types) as a supplement to the standard, depending on the programming system.

Overloading and extensibility of standard functions is explained with the use of examples in the next two sections.

Overloaded functions

Overloaded functions can be applied for processing several data types using only one function name.

An overloaded function does **not** always support **every** data type of a generic data type, as explained in Chapter 3.

For example, if a PLC programming system recognises the integer data types INT, DINT and SINT, only these three data types will be accepted for an overloaded function ADD which supports the generic data type ANY_INT.

If a standard function is not overloaded, but restricted to a certain elementary data type, an underline character and the relevant data type must be added to its name: e.g. ADD_SINT is an addition function restricted to data type SINT. Such functions are called *typed*. Overloaded functions can also be referred to as *type-independent*.

This is illustrated in Example 5.1 using integer addition:

*Typed standard
functions ADD_**

Overloaded standard function ADD
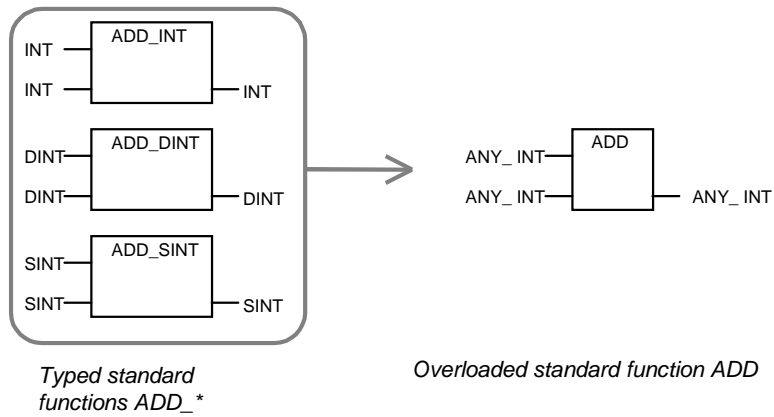
Example 5.1. Typed standard functions ADD_INT, ADD_DINT and ADD_SINT for integer addition and the overloaded standard function ADD

When overloaded functions are used, the programming system automatically chooses the appropriate typed function. For example, if the ADD function shown in Example 5.1 is called with actual parameters of data type DINT, the ADD_DINT function will automatically be selected and called (invisibly to the user).

When calling standard functions, each overloaded input and, in some cases, the function return value, must be of the same data type, i.e. it is not permissible to use variables of different types as actual parameters at the same time.

 If the inputs are of different data types, the PLC programmer must use explicit type conversion functions for the corresponding inputs and function return value respectively, as shown in Example 5.2 for ADD_DINT and ADD_INT. In such cases, instead of the overloaded function ADD its typed variant (e.g. ADD_DINT) should be used.

```
VAR
  Var_Integer        :      INT;
  Var_ShortInteger   :      SINT;
  Var_FloatingPoint  :      REAL;
  Var_DoubleWord     :      DWORD;
END_VAR
```
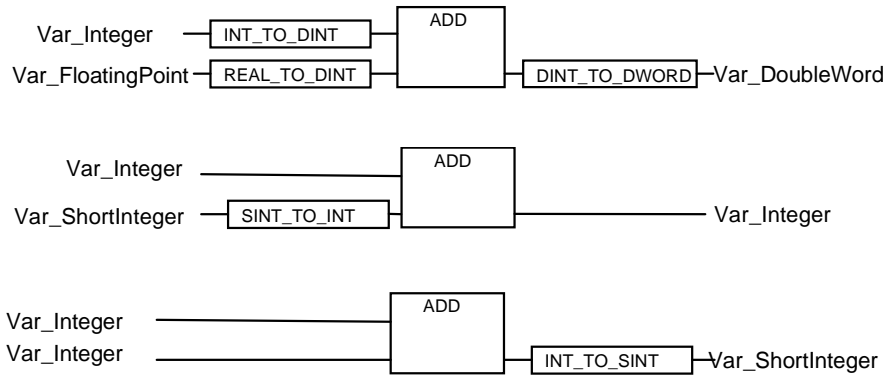

Example 5.2. Calls of the overloaded standard function ADD with type conversion functions to ensure correct input data types. In the top case the programming system replaces ADD with the typed standard function ADD_DINT. In the other two cases the function ADD_INT is used.

Extensible functions

Extensible standard functions can have a variable number of inputs, between two and an upper limit imposed by the PLC system. In graphical representation, the height of their boxes depends on the number of inputs.

Extending the number of inputs of a standard function serves the same purpose as using cascaded calls to the same function, in both the textual and the graphical programming languages of IEC 61131-3. Especially in the graphical languages LD and FBD, the amount of space required to write the function can be greatly reduced.

Cascaded adders   *Extended adder*
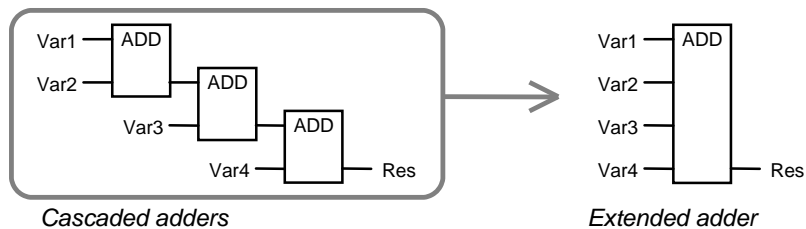
Instruction List (IL)	Structured Text (ST)
LD Var1	
ADD Var2	Res := Var1 + Var2;
ADD Var3	Res := Res + Var3;
ADD Var4	Res := Res + Var4;
ST Res	
Can be replaced by:	*Can be replaced by:*
LD Var1	
ADD Var2, Var3, Var4	Res := Var1 + Var2 + Var3 + Var4;
ST Res	

Example 5.3. Cascaded functions as an alternative representation for an extensible function showing addition in graphical and textual (IL and ST) representation

In Example 5.3 the triple call of the standard function ADD is replaced by a single call with extended inputs. Simplifications also result for the textual versions in IL and ST.

5.1.1 Examples

In this section the calling interfaces of the standard functions are shown in examples. The subject of calling functions has already been discussed in detail in Chapter 2.

   At least one example has been selected from each function group in Table 5.1. The examples are given in the textual languages IL and ST and graphical representations LD and FBD. In IL and ST the names of the formal parameters are not specified explicitly in the function calls.

For these examples the PROGRAM ProgFrameFUN in Example 5.4 is used as the basis for the common declaration part for the required variables.

```
TYPE                                    (* enumeration type for colours *)
  COLOURS :    ( lRed, lYellow, lGreen,        (* light *)
                 Red, Yellow, Green,           (* normal *)
                 dRed, dYellow, dGreen);       (* dark *)
END_TYPE

PROGRAM ProgFrameFUN        (* common declaration part for std. FUNs *)
  VAR                                   (* local data *)
    RPM        : REAL:= 10.5;           (* revs *)
    RPM1       : REAL;                  (* revs 1 *)
    RPM2       : REAL := 46,8895504;    (* revs 2 *)
    Level      : UINT  := 1;            (* revs level *)
    Status     : BYTE := 2#10101111;    (* status *)
    Result     : BYTE;                  (* intermediate result *)
    Mask       : BYTE := 2#11110000;    (* bit mask *)
    PLCstand   : STRING [11]:= 'IEC 61131-5';  (* character string *)
    AT %IB2    : SINT;                  (* for MUX selection *)
    AT %QX3.0  : BOOL;                  (* output bit *)
    DateTime   : DT := dt#1994-12-23-01:02:03;  (* date and time *)
    Time       : TIME := t#04h57m57s;   (* time *)
    TraffLight : COLOURS;               (* traffic light *)
    ColScale1  : COLOURS:= lYellow;     (* scale of colours 1 *)
    ColScale2  : COLOURS:= Yellow;      (* scale of colours 2 *)
    ColScale3  : COLOURS:= dYellow;     (* scale of colours 3 *)
    Scale      : INT   := 2;            (* selection scale of colours *)
  END_VAR
  ...           (* program body with examples *)
END_PROGRAM
```

Example 5.4. The common declarations for the examples explaining the usage of the standard functions

Type conversion functions

RPM ———[REAL_TO_UINT]——— Level

Instruction List (IL)	*Structured Text (ST)*
LD RPM **REAL_TO_UINT** ST Level	Level := **REAL_TO_UINT** (RPM);
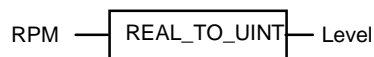
Example 5.5. Converting from REAL to UINT

This example shows type conversion of the REAL value RPM (floating point) to the unsigned integer (UINT) value Level.

The variable Level has the value 10 after executing the function, as it is rounded down from 10.5.

Numerical functions

```
RPM ──┤ LN ├── RPM1
```
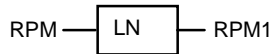
	Instruction List (IL)	*Structured Text (ST)*
LD	RPM	
LN		RPM1 := **LN** (RPM);
ST	RPM1	

Example 5.6. Natural logarithm

This example shows the calculation of a natural logarithm. Variable RPM1 has the value 2.3513 after execution.

Arithmetic functions

```
RPM ──┤ MUL
RPM1─┤
RPM2─┤     ├── RPM
```
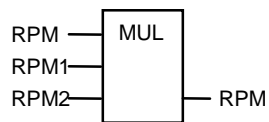
	Instruction List (IL)	*Structured Text (ST)*
LD	RPM	
MUL	RPM1	RPM := RPM * RPM1 * RPM2;
MUL	RPM2	
ST	RPM	

Example 5.7. Multiplication. Instead of using MUL twice in IL, the shortened form: MUL RPM1, RPM2 can also be used.

This example uses the overloaded multiplication function. Because its inputs are of type REAL it is mapped to the typed function MUL_REAL. The variable RPM has the value 1157,625 after execution of the function.

In graphical representation, the multiplication sign " * " may also be used instead of the keyword MUL. This is shown here only for ST.

Bit-shift functions

```
        ┌─────────┐
        │  SHL    │
Status ─┤ IN      │
Level ──┤ N       ├── Result
        └─────────┘
```
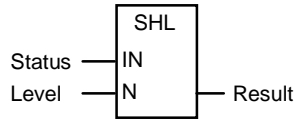
	Instruction List (IL)		*Structured Text (ST)*

```
LD      Status                      Result := SHL ( IN := Status,
SHL     Level                                      N  := Level );
ST      Result
```

Example 5.8. Bit-shift left

In Example 5.8 the shift function SHL is used to shift the value of the variable Status to the left by the number of bit positions specified by the value Level.

After executing the shifting function, Result has the value 2#01011110, i.e. when shifting left, a "0" is inserted from the right.

Bitwise Boolean functions

```
        ┌─────────┐
        │  AND    │
Status ─○┤        │
Mask ───┤         ├── Result
        └─────────┘
```
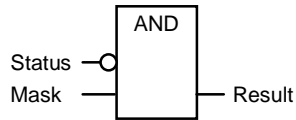
	Instruction List (IL)		*Structured Text (ST)*

```
LD      Status
NOT
AND     Mask                        Result := NOT Status & Mask;
ST      Result
```

Example 5.9. AND operation

As the logical AND is an extensible function, the input parameter names need not be given (similar to MUL). AND is also an overloaded function, its inputs and function value here are of type BYTE. The programming system therefore automatically uses the typed function AND_BYTE.

Instead of the IL instructions LD and NOT in Example 5.9 the Boolean operator LDN (load negated) could be used. This inversion is graphically represented by a function input with a small circle.

In graphical representation, the normal AND symbol "&" may also be used instead of the keyword AND. This is shown here only for ST.

In Example 5.9 AND is used to extract certain bits from the value Status with the aid of a bit mask.

The output Result has the value 2#01010000 after executing the function, i.e. the lower four bits have been reset by the mask.

Selection functions
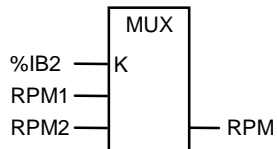
```
        ┌─────────┐
        │  MUX    │
%IB2 ───┤K        │
RPM1 ───┤         │
RPM2 ───┤         ├─── RPM
        └─────────┘
```

Instruction List (IL)	*Structured Text (ST)*

```
LD      %IB2                    RPM := MUX (  K    := %IB2,
MUX     RPM1, RPM2                            RPM1,
ST      RPM                                   RPM2 );
```

Example 5.10. Multiplexer

The multiplexer MUX has an integer input K and overloaded inputs of the same data type as the function value. The input parameter names, with the exception of K, can therefore be left out. In Example 5.10 K is of data type SINT (integer Byte with sign).

If the input byte %IB2 has the value "1", RPM is assigned the value RPM2 after execution, if it is "0", it is assigned the value RPM1.

If the value of input K is less than 0 or greater than the number of the remaining inputs, the programming system or the run-time system will report an error (see also Appendix E).

Comparison functions

Extended comparison   *Equivalent representation*
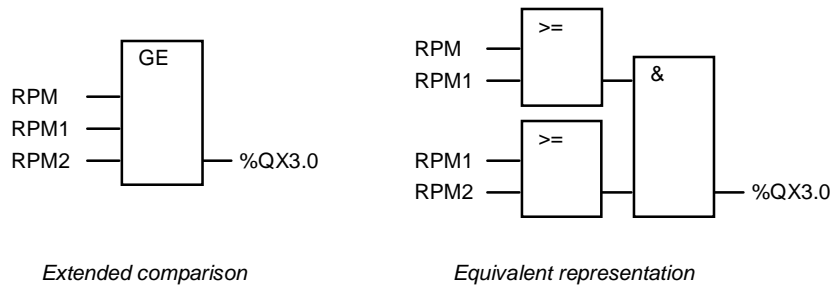
Instruction List (IL)	*Structured Text (ST)*

```
LD     RPM
GE     RPM1                    %QX3.0 := GE ( RPM,
GE     RPM2                                  RPM1,
ST     %QX3.0                                RPM2 );
```

Example 5.11. An extended comparison. An equivalent solution is shown in graphical representation on the right-hand side.

Comparison functions use overloaded inputs, their output Q is Boolean. They represent a kind of "connecting link" between numerical/arithmetic calculations and logical/Boolean operations.

In the graphical representation of Example 5.11 the comparison function extended by one additional input is also shown as an equivalent call of three functions. Here the key words GE and AND are replaced by the symbols ">=" and "&".

Character string functions

```
        ┌─────────┐
        │ REPLACE │
PLCstand─┤IN1      │
     '-3'─┤IN2      │
       2─┤L        │
      10─┤P        ├─ PLCstand
        └─────────┘
```

	Instruction List (IL)
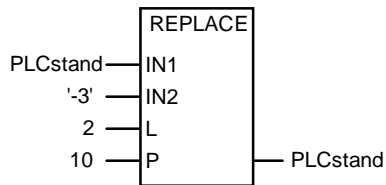
Instruction List (IL)

```
LD        PLCstand
REPLACE   '-3', 2, 10
ST        PLCstand
```

Structured Text (ST)

```
PLCstand:= REPLACE (
           IN2 := '-3',
           IN1 := PLCstand,
           P   := 10,
           L   := 2 );
```

Example 5.12. Example of **"REPLACE"** in IL and ST

The function REPLACE has no overloaded inputs. When calling this function, both graphically and in ST, each input parameter must be entered with its name.

Example 5.12 shows that these inputs can then be entered in any order (see ST example). On the other hand, the order is fixed if there are no input parameter names (see IL example).

The character string PLCstand has the STRING value 'IEC 61131-3' after execution.

Functions for time data types.

```
        ┌───────┐
        │   +   │
DateTime─┤IN1    │
   Time─┤IN2    ├─ DateTime
        └───────┘
```
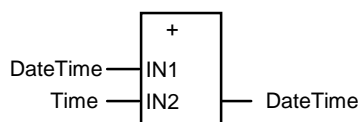
Instruction List (IL)

```
LD    DateTime
ADD   Time
ST    DateTime
```

Structured Text (ST)

```
DateTime:= DateTime + Time;
```

Example 5.13. An example of **"ADD Time"** in IL and ST

This time addition function (and also the corresponding subtraction) can be regarded as a continuation of overloaded addition — referring to mixed arguments: TIME, TIME_OF_DAY (TOD) and DATE_AND_TIME (DT).

The variable DateTime has the value DT#1994-12-23-06:00:00 after executing the function.

The addition and subtraction of time is not symmetrical. For subtraction, as opposed to addition, there are three additional operations for input data types DATE, TOD and DT. These operations are not available for addition, as it does not make much sense to add, for example, 10th October to 12th September.

In addition, it is not possible to add a TIME to a DATE, whereas this is possible for TIME, TOD and DT. In order to make this possible with DATE, the input must first be converted to DT and then added. Possible programming errors in time calculations can thus be avoided.

Functions for enumerated data types
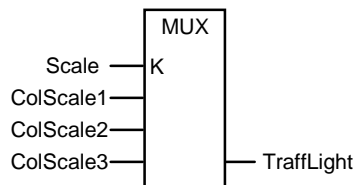
Instruction List (IL)	*Structured Text (ST)*
LD Scale	
MUX (ColScale1,	TraffLight := **MUX** (K := Scale,
ColScale2,	ColScale1,
ColScale3)	ColScale2,
ST TraffLight	ColScale3);

Example 5.14. An example of MUX with enumeration

IEC 61131-3 defines functions for the data type enumeration, one of which, the selection function MUX, is shown in Example 5.14.

A variable of data type enumeration (type declaration COLOURS) is selected using the INT variable Scale.

After executing MUX the variable TraffLight will have the values "lYellow", "Yellow" and "dYellow" from the colour scale (light, normal and dark) when the variable Scale has the values 0, 1 and 2 respectively.

5.2 Standard Function Blocks

IEC 61131-3 defines several standard function blocks covering the most important PLC functions (with retentive behaviour).

IEC 61131-3 defines the following five groups of standard FBs:

1) Bistable elements (= flipflops)
2) Edge detection
3) Counters
4) Timers
5) Communication function blocks.

Table 5.3 gives a concise list of all the standard FBs available in these groups. The table structure is very similar to the one for standard functions in Table 5.1. The communication FBs are defined in part 5 of IEC 61131 and not dealt with in this book.

Instead of the data types of the input and output variables, their names are listed here. These names, together with their corresponding elementary data types, can be found in Table 5.4.

Name of std. FB with input parameter names	Names of output parameters	Short description
Bistable elements		
SR (S1, R,	Q1)	Set dominant
RS (S, R1,	Q1)	Reset dominant
Edge detection		
R_TRIG {->} (CLK,	Q)	Rising edge detection
F_TRIG {-<} (CLK,	Q)	Falling edge detection
Counters		
CTU (CU, R, PV,	Q, CV)	Up counter
CTD (CD, LD, PV,	Q, CV)	Down counter
CTUD (CU, CD, R, LD, PV,	QU, QD, CV)	Up/down counter
Timers		
TP (IN, PT,	Q, ET)	Pulse
TON {T---0} (IN, PT,	Q, ET)	On-delay
TOF {0---T} (IN, PT,	Q, ET)	Off-delay
RTC (EN, PDT,	Q, CDT)	Real-time clock
Communication		***See IEC 61131-5***

Table 5.3. List of standard function blocks

Inputs / Outputs	Meaning	Data type

R	Reset input	BOOL
S	Set input	BOOL
R1	Reset dominant	BOOL
S1	Set dominant	BOOL
Q	Output (standard)	BOOL
Q1	Output (flipflops only)	BOOL
CLK	Clock	BOOL
CU	Input for counting up	R_EDGE
CD	Input for counting down	R_EDGE
LD	Load (counter) value	INT
PV	Pre-set (counter) value	INT
QD	Output (down counter)	BOOL
QU	Output (up counter)	BOOL
CV	Current (counter) value	INT
IN	Input (timer)	BOOL
PT	Pre-set time value	TIME
ET	End time output	TIME
PDT	Pre-set date and time value	DT
CDT	Current date and time	DT

Table 5.4. Abbreviations and meanings of the input and output variables in Table 5.3

The counter inputs CU and CD are of data type BOOL and have an additional attribute R_EDGE, i.e. a rising edge has to recognised in order to count up or down.

The return value of each standard FB is zero when the FB is called for the first time. Only the real-time clock displays the current date and time immediately at its CDT output (Current Date and Time).

The input parameter names of the standard FBs are keywords. In IL they can be applied as operators to FB instances, as described in Section 4.1.4.

The input parameters R and S have a second meaning in IL. There they are also the operators used to set and reset Boolean variables. This can cause difficulties that need to be solved when implementing programming systems.

5.2.2 Examples

In this section, examples are given to illustrate the calling interfaces of standard function blocks in the same way as for the standard functions. The subject of FB calls has already been discussed in detail in Chapter 2.

At least one example is given for each function group in Table 5.3. Both the textual languages IL and ST and the graphical representations LD and FBD are used.

In IL and ST the FB input parameter names are given explicitly in order to make the use of FB instances as clear as possible.

In the case of IL, the version of the function block call (see Section 4.1.4) that treats input parameters and return values as structure elements of the FB instance is used.

For the following examples, the PROGRAM ProgFrameFB in Example 5.15 is used as the basis for the common declaration part for the required variables and FB instances.

```
PROGRAM ProgFrameFB                  (* common declaration part for std. FBs *)

  VAR_GLOBAL RETAIN                  (* global, battery-backed data *)
    RealTime        :  RTC;          (* real-time clock *)
    TimePeriod      :  TIME := t#63ms;   (* 63 milliseconds as initial value *)
  END_VAR

  VAR                                (* local data *)
    FlipFlop        :  RS;           (* flag *)
    Button          :  R_TRIG;       (* edge detection button *)
    Counter_UD      :  CTUD;         (* counter up/down *)
    V_pulse         :  TP;           (* extended pulse *)
    Pulse           :  BOOL;         (* pulse flag*)
    EmOff           :  BOOL;         (* emergency off flag*)
    AT %IX1.4       :  BOOL;         (* emergency off *)
    AT %IX2.0       :  BOOL;         (* count up *)
    AT %IX2.1       :  BOOL;         (* load counter *)
    AT %IX2.2       :  BOOL;         (* start time *)
    AT %IX3.0       :  BOOL;         (* count down *)
    AT %IW5         :  INT;          (* count limit *)
    AT %MX3.2       :  BOOL;         (* flag *)
    AT %QX3.2       :  BOOL;         (* output *)
    MaxReached      :  BOOL;         (* counter at max. limit *)
    MinReached      :  BOOL;         (* counter at min. limit *)
    CounterValue  AT %MW2 : INT;     (* current counter value *)
    TimerValue      :  TIME;         (* current timer value *)
    DateAndTime     :  DT;           (* current date and time *)
  END_VAR

    ...                (* program body for following examples *)

  END_PROGRAM
```

Example 5.15. The common declarations for the examples on the usage of the standard function blocks

These declarations contain:

- FB instances (from FlipFlop to V_pulse)
- Directly represented variables (from %IX1.4 to %QX3.2)

- Symbolic variable (CounterValue)
- General variables (others).

The variables declared in the VAR section are declared as local variables and those declared in the VAR_GLOBAL RETAIN section are declared as battery-backed global variables.

Bistable element (flipflop)

```
          FlipFlop
         ┌─────────┐
         │   RS    │
         │         │
%MX3.2 ──┤S        │
         │         │
EmOff  ──┤R1    Q1 ├── %QX3.2
         └─────────┘
```
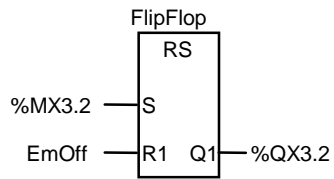
Instruction List (IL)		*Structured Text (ST)*

LD	%MX3.2	
ST	FlipFlop.S	
LD	EmOff	
ST	FlipFlop.R1	
CAL	**FlipFlop**	**FlipFlop** (S := %MX3.2,
LD	FlipFlop.Q1	R1 := EmOff);
ST	%QX3.2	

Example 5.16. Bistable element (flipflop)

Example 5.16 shows how to use a flipflop to store binary status information, in this case the value of flag %MX3.2.

The input R1 "dominantly" resets the output Q1, i.e. if both inputs are set to "1" the output remains "0".

Edge detection

```
          Button
         ┌─────────┐
         │ R_TRIG  │
         │         │
%IX1.4 ──┤CLK    Q ├── EmOff
         └─────────┘
```
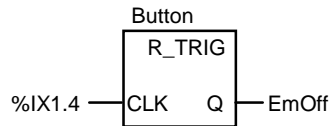
Instruction List (IL)		*Structured Text (ST)*

LD	%IX1.4
ST	Button.CLK

```
CAL   Button                        Button        ( CLK := %IX1.4);
LD    Button.Q
ST    EmOff                         EmOff := Button.Q;
```

Example 5.17. Rising edge detection with R_TRIG

FB instance Button of FB type R_TRIG in Example 5.17 evaluates the signal of an I/O bit and produces a "1" at Q when there is a rising edge (0→1 transition). To do this FB Button uses an internal edge detection flag that stores the "old" value of CLK in order to compare it with the current value.

This information is stored for one program cycle (until the next call) and can be processed by other program parts even if %IX1.4 has already returned to "0" again. At the next call in the following cycle, the Button flag will again be reset.
This means that for directly represented variables FB Button can only detect edges that occur at intervals of at least one program cycle.
IEC 61131-3 provides FBs R_TRIG and F_TRIG not only for immediate usage as shown in Example 5.17. These FBs are also implicitly used for edge detection to implement the variable attributes R_EDGE and F_EDGE (see Chapter 3).

Example 5.18 shows variable declaration using an edge-triggered input (bold text) within the declaration part of FB ExEdge.

```
FUNCTION_BLOCK  ExEdge
VAR_INPUT
  Edge        :   BOOL  R_EDGE;          (* edge-triggered *)
END_VAR
VAR_OUTPUT
  Flag        :   BOOL;
END_VAR
...
LD    Edge;                               (* access to edge flag *)
ST    Flag;
...
END_FUNCTION_BLOCK
```

Example 5.18. A declaration with R_EDGE for edge detection and usage in IL

To make the use of edge-triggered variables clearer, Example 5.19 shows how additional instructions which implement edge detection are added to Example 5.18. This is done — invisibly to the user — by the programming system.

```
FUNCTION_BLOCK  ExEdge
VAR_INPUT
  Edge        :   BOOL;          (* edge-triggered *)
END_VAR
VAR_OUTPUT
  Flag        :   BOOL;
END_VAR
```

```
VAR
   EdgeDetect:   R_TRIG;          (* FB instance "rising edge" *)
END_VAR
...
CAL   EdgeDetect (CLK := Edge);  (* FB call for edge detection *)
LD    EdgeDetect.Q;              (* load detection result from FB instance *)
ST    Flag;
...
END_FUNCTION_BLOCK
```

Example 5.19. Automatic extension of Example 5.18 using R_TRIG

The declaration of FB EdgeDetect in Example 5.19 is inserted implicitly and invisibly by the programming system. This FB is called with input variable Edge. Its output value EdgeDetect.Q is then used wherever the value Edge is originally accessed.

This example shows why IEC 61131-3 does not allow this kind of edge detection for output variables: these variables could be overwritten at any point within the POU. This would, however, violate the rule that FBs are not allowed to change the outputs of other called FBs! See also Section 2.3.2.

Counter

```
                       Counter_UD
                      ┌──────────┐
                      │   CTUD   │
                      │          │
%IX2.0 ───────────────┤>CU    QU ├── MaxReached
%IX3.0 ──┤ NOT ├──────┤>CD    QD ├── MinReached
%IX1.4 ───────────────┤R         │
%IX2.1 ───────────────┤LD        │
%IW5 ─────────────────┤PV     CV ├── CounterValue
                      └──────────┘
```
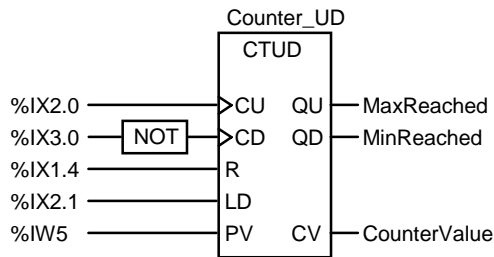
Instruction List (IL)	*Structured Text (ST)*
LD %IX2.0	
ST Counter_UD.CU	
LDN %IX3.0	
ST Counter_DU.CD	
LD %IX1.4	
ST Counter_DU.R	
LD %IX2.1	
ST Counter_DU.LD	
LD %IW5	
ST Counter_UD.PV	
CAL **Counter_DU**	**Counter_UD** (CU := %IX2.0
	CD := NOT(%IX3.0),

LD	Counter_UD.QU
ST	MaxReached
LD	Counter_DU.QD
ST	MinReached
LD	Counter_UD.CV
ST	CounterValue

```
              R   := %IX1.4,
              LD  := %IX2.1,
              PV  := %IW5);

MaxReached := Counter_UD.QU;
MinReached := Counter_UD.QD;
CounterValue := Counter_UD.CV;
```

Example. 5.20. The up/down counter CTUD

In this example, each input of the up/down counter Counter_UD is used. This is, however, not always necessary.

The inputs CU and CD can be activated simultaneously by a rising edge. In this case the current counter value would not change if the minimum or maximum had not already been reached.

Counter_UD in Example 5.20 counts up with each rising edge at %IX2.0 and counts down with each falling edge at %IX3.0. The pre-set counter value at PV is loaded from %IW5 if the load input LD is active when the FB is called. No rising edge is needed in this case.

Timer

Example 5.21 is an example of the usage of timer FBs. It demonstrates clearly how instances of timers maintain their values, especially those of the input parameters, between calls.

In principle, each input variable of a timer (or any FB) can be set immediately before calling. Such run-time parameter changes could be used to allow the same timer to be used to control several process times simultaneously. Such programming is, however, seldom used in practice as it makes the program difficult to read and can easily lead to errors.

It is sufficient to set the pre-set timer value PT for each instance only once, with the first call, and then to re-use it for later invocations. This means that calling the timer primarily serves to *start* the timer with input IN.

The output variables of a timer can be checked at any point in the program, i.e. they need not be evaluated immediately after calling the timer.

The output parameters are set at each call of the timer FB, i.e. they are updated with the current values of the physical timer running in the background. The timer value may therefore become obsolete between two timer calls. Therefore, in order to avoid distorting the desired time control, it must be ensured that the timer FB is called sufficiently frequently in a periodic task, not too long before Q or ET are evaluated.
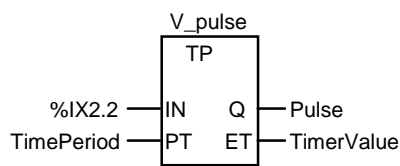
Output Q shows whether the time has elapsed or not, and output ET shows the time still remaining.

Timers are thus usually called in the following steps:

1) Setting of the timer value
2) Periodic starting with updating
3) Checking of the timer values.

In a PLC program executing periodically, these three steps are often combined in a single call. This simplifies the program and makes the graphical representation easier.

The behaviour of the different timers is shown in more detail in Appendix B.

```
                V_pulse
                ┌──────┐
                │  TP  │
                │      │
  %IX2.2 ──────IN    Q├─── Pulse
  TimePeriod ───PT   ET├─── TimerValue
                └──────┘
```

Instruction List (IL)	*Structured Text (ST)*
(* 1. set pulse length *)	(* 1. set pulse length *)
LD TimePeriod	V_pulse.PT := TimePeriod;
ST V_pulse.PT	
...	...
(* 2. start timer *)	
LD %IX2.2	
ST V_pulse.IN	(* 2. start timer *)
CAL **V_pulse**	**V_pulse** (IN := %IX2.2);
...	...
(* 3. get current timer value *)	(* 3. get current timer value *)
LD V_pulse.Q	
ST Pulse	Pulse := V_pulse.Q;
LD V_pulse.ET	
ST TimerValue	TimerValue := V_pulse.ET;

Example 5.21. Creating pulses using the timer TP

Example 5.21 shows the three steps required when using the instance V_pulse:

1) The timer value for V_pulse is pre-set to 63 milliseconds.
2) V_pulse is started by input bit 2.2.
3) V_pulse is evaluated by checking Q and ET.

Example 5.22 shows an example of the use of the timer FB RTC (real-time clock).

```
                        RealTime
                        ┌─────────┐
                        │   RTC   │
                        │         │
         %IX2.2 ────────┤IN     Q ├──┤NOT├──≫TimerError
dt#1994-01-11-12:00:00.00 ┤PDT  CDT ├──────── DateAndTime
                        └─────────┘
```

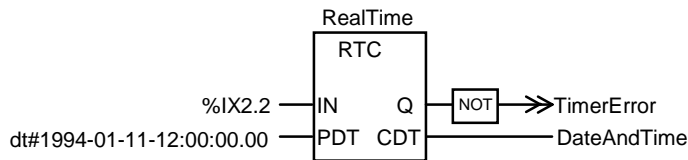| *Instruction List (IL)*	*Structured Text (ST)*

<table>
<tr><td colspan="2"><i>Instruction List (IL)</i></td><td colspan="2"><i>Structured Text (ST)</i></td></tr>
<tr><td></td><td>(* 1. set date *)</td><td colspan="2">(* 1. set date *)</td></tr>
<tr><td>LD</td><td>dt#1994-01-11-12:00:00.00</td><td colspan="2">RealTime.PDT := dt#1994-01-11-12:00;</td></tr>
<tr><td>ST</td><td>RealTime.PDT</td><td colspan="2"></td></tr>
<tr><td>...</td><td></td><td colspan="2">...</td></tr>
<tr><td></td><td>(* 2. start clock *)</td><td colspan="2"></td></tr>
<tr><td>LD</td><td>%IX2.2</td><td colspan="2"></td></tr>
<tr><td>ST</td><td>RealTime.IN</td><td colspan="2">(* 2. start clock *)</td></tr>
<tr><td>CAL</td><td>RealTime</td><td colspan="2">RealTime   (IN := %IX2.2);</td></tr>
<tr><td>...</td><td></td><td colspan="2">...</td></tr>
<tr><td></td><td>(* 3. get time *)</td><td colspan="2">(* 3. get time *)</td></tr>
<tr><td>LD</td><td>RealTime.Q</td><td colspan="2">IF RealTime.Q THEN</td></tr>
<tr><td>JMPCN</td><td>TimerError</td><td colspan="2"> DateAndTime := RealTime.CDT;</td></tr>
<tr><td>LD</td><td>RealTime.CDT</td><td colspan="2"> ELSE</td></tr>
<tr><td>ST</td><td>DateAndTime</td><td colspan="2"> ...   (* TimerError *)</td></tr>
<tr><td>JMP</td><td>Anywhere</td><td colspan="2"> ENDIF</td></tr>
<tr><td colspan="2">TimerError:</td><td colspan="2"></td></tr>
<tr><td>...</td><td>...</td><td colspan="2"></td></tr>
</table>

Example 5.22. Timer: real-time clock RTC

On a rising edge at %IX2.2 the date constant NewDate will pre-set the timer at the input PDT. As long as IN remains "1", the current date and time can be read at the output CDT.

The output Q is used to check whether the current date is still valid (Q is a copy of IN). If the date is invalid, the program jumps to the error routine TimerError.

6 State-of-the-Art PLC Configuration

IEC 61131-3 takes advantage of recent advances in technology by incorporating modern concepts that allow the modelling of PLC projects consisting of more than just single-processor applications.

The software model of IEC 61131-3 allows for practice-oriented structuring (modularization) of applications into units (POUs). This eases maintenance and documentation, and improves the diagnostics facilities for PLC programs.

An uniform software architecture is essential for portability of applications. The resources of PLCs are given explicit run-time properties, thus building a platform for hardware-independent programs.

In the traditional method of structuring PLC projects (see Figure 2.5), applications are modularised into blocks, and certain types of blocks (e.g. organisation blocks) have implicit run-time properties. IEC 61131-3 provides more sophisticated and standardised means of accomplishing this.

This chapter explains the *configuration elements* of IEC 61131-3, which are an important means of structuring applications and defining the interaction of POUs. Configuration elements describe the run-time properties of programs, communication paths and the assignment to PLC hardware.

IEC 61131-3 configuration elements support the use of today's sophisticated operating systems for PLCs. A typical PLC today can run multiple programs at the same time (multitasking).

6.1 Structuring Projects with Configuration Elements

The preceding chapters have discussed the programming and usage of POUs. This section gives an overview of the modelling and structuring of PLC applications at a higher level.

To do this, Figure 2.7 (POU calling hierarchy) needs to be seen within the context of the PLC program as a whole:
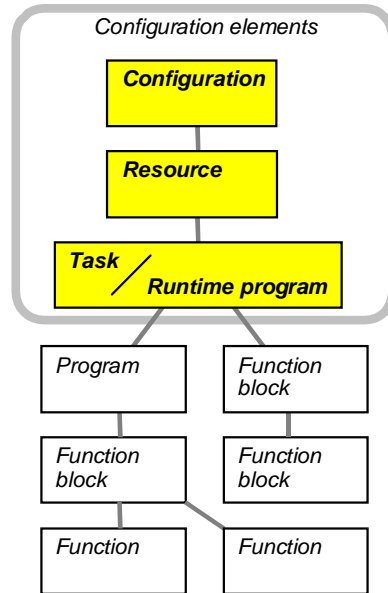
Figure 6.1. Overall structure of PLC programs according to IEC 61131-3, including POUs and configuration elements

As shown in Figure 6.1, the *configuration elements* configuration, resource and task are hierarchically located above the POU level.

While POUs make up the calling hierarchy, configuration elements assign properties to POUs:

- PROGRAMs and FBs are assigned run-time properties
- Communication relationships are defined between configurations
- Program variables are assigned to PLC hardware addresses.

First, we will describe the structure and meaning of the configuration elements themselves.

6.2 Elements of a Real-World PLC Configuration

Configuration elements match the elements found in real-world PLC systems:

- *Configuration*:   A PLC system, e.g. a controller in a rack with multiple CPUs, controlling a cell of machines
- *Resource*:   One CPU in a PLC, possibly capable of multitasking

- *Task*:   Run-time properties for programs and function blocks ("type" of PLC program)
- *Run-time program:* Unit consisting of a PROGRAM or FUNCTION_BLOCK and the TASK associated with it

The main programs of a CPU are made up of POUs of type PROGRAM. Larger applications tend to be structured in Sequential Function Chart, controlling the execution of the other POUs.

Main programs and function blocks are assigned run-time properties, like "cyclic execution" or "priority level", as indicated in Figure 6.2.

The term "run-time program" denotes the unit consisting of all the necessary POUs *and* the TASK, i.e. a program together with its run-time properties. A run-time program is therefore a self-contained unit capable of running independently in a CPU.

Figure 6.2 shows the relation between configuration elements and the components of real-world PLC systems (see also Figure 2.4):
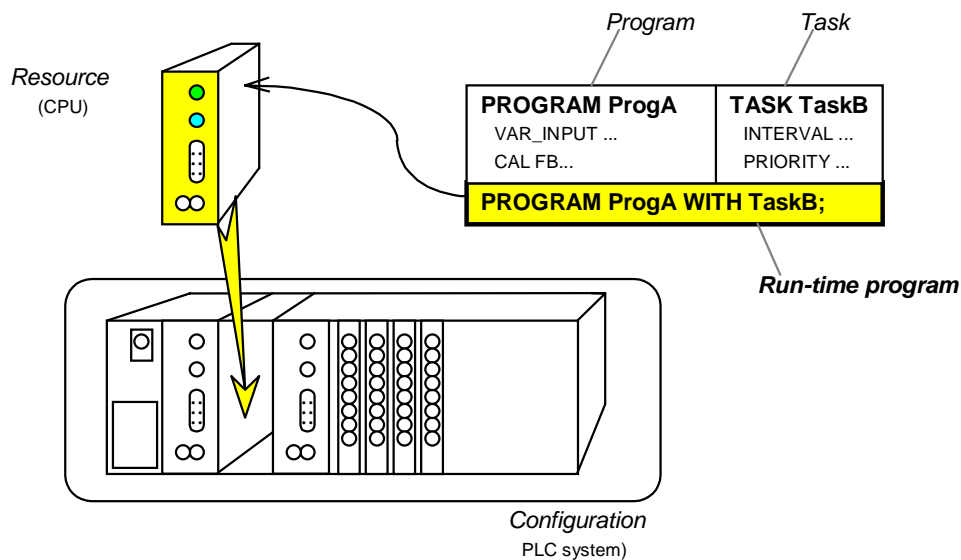
Figure 6.2. A real-world configuration. ProgA and TaskB are linked to form a run-time program and assigned to a CPU resource in a PLC system

The actual assignment of configuration elements to the elements of a PLC system will depend on the hardware architecture.

Using configuration elements, all tasks can be assigned to *one* CPU which will execute them simultaneously, or they can be assigned to different CPUs.

Whether a RESOURCE is to be regarded as one CPU or a group of CPUs contained in one rack therefore depends on the concrete PLC hardware architecture.

For small PLC systems, all configuration can be done in one POU of type PROGRAM: programs can declare global variables and access paths, and directly represented variables. The definition of run-time properties, CPU assignment, etc. can be performed implicitly by features of the programming system or PLC. This corresponds to the traditional approach for programming PLCs.

This capability of PROGRAM POUs facilitates gradual migration from existing applications to IEC 61131-3-compliant programs.

6.3 Configuration Elements

We will first give an overview of the configuration elements, and then explain them in more detail. We will refer to the example in Section 6.4.

6.3.1 Definitions

The functions of the configuration elements are as follows:

Configuration Element	Description
Configuration	- Definition of global variables (valid within this configuration) - Combination of all resources of a PLC system - Definition of access paths between configurations - Declaration of directly represented variables
Resource	- Definition of global variables (valid within this resource) - Assignment of tasks and programs to a resource - Invocation of run-time programs with input and output parameters - Declaration of directly represented variables
Task	- Definition of run-time properties
Run-time program	- Assignment of run-time properties to a PROGRAM or FUNCTION_BLOCK

Table 6.1. Definition of configuration elements. Directly represented global variables and access paths can also be defined within a PROGRAM.

Declaration of directly represented variables maps the entire configuration to the hardware addresses of the PLC. These declarations can be made at the configuration, resource or PROGRAM level. POUs can access these via VAR_EXTERNAL declarations.

When put together, the declarations of directly represented variables for all the POUs make up the allocation table of a PLC application. Rewiring, i.e. re-assigning symbolic addresses to absolute PLC addresses, can be carried out by simply modifying this list.

Configuration elements are typically declared in textual form. The standard provides a definition for a graphical representation of a TASK, but the graphical representation of all other configuration elements is left to the programming system and is therefore implementation-dependent.

6.3.2 The CONFIGURATION

IEC 61131-3 uses the Configuration (CONFIGURATION) to group together all the resources (RESOURCE) of a PLC system and provide them with means for data exchange. A configuration consists of the elements shown in Figure 6.3.

CONFIGURATION *Configuration name*

Type definitions Global declarations

RESOURCE declarations

ACCESS declarations
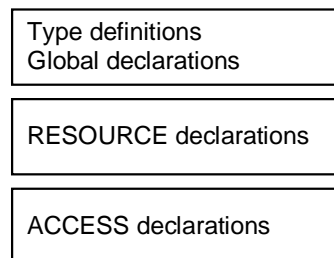
END_CONFIGURATION

Figure 6.3. Structure of a CONFIGURATION declaration

Within a configuration, type definitions with global validity for the entire PLC project can be made. This is not possible in other configuration elements.

Communication between configurations takes place via access paths defined with VAR_ACCESS. Variables defined with VAR_GLOBAL are valid only within one configuration, and are accessible to all resources, programs and function blocks of that configuration. VAR_EXTERNAL cannot be used at the configuration level.

Communication blocks for communication between configurations are defined in part 5 of IEC 61131 (see also Section 6.5).

```
CONFIGURATION PLC_Cell1
  VAR_GLOBAL ...  END_VAR
   RESOURCE CPU_Conveyor ON CPU_001 ... END_RESOURCE
   RESOURCE CPU_Hammer ON CPU_002... END_RESOURCE
  VAR_ACCESS ... END_VAR
END_CONFIGURATION
```

Example 6.1. Elements of the CONFIGURATION in Example 6.6

Example 6.1 shows part of the declaration of a configuration named PLC_Cell1.

It contains a section with global variables, which are not visible to other configurations, but only accessible from resources CPU_Conveyor and CPU_Hammer and all POUs executing on them.

VAR_ACCESS is used for exchanging data between the resources of the same configuration or different configurations (access paths)

Configurations and resources do not contain instructions like POUs, but solely define the relations between their elements.

6.3.3 The RESOURCE

A Resource is defined in order to assign TASKs to the physical resources of a PLC system. A resource consists of the elements shown in Figure 6.4.

RESOURCE *Resource name*   **ON**   *Resource*

Global declarations

TASK declarations

END_RESOURCE

Figure 6.4. Structure of a resource declaration

The resource name assigns a symbolic name to a CPU in a PLC. The types and numbers of the resources in a PLC system (individual CPU designations) are

provided by the programming system and checked to ensure that they are used correctly.

Global variables, which are permissible at resource level, can be used for managing the data that are restricted to one CPU.

Example 6.2 shows part of the declaration of two resources. The global data declared for resource CPU_002 cannot be accessed from resource CPU_001.

The keyword PROGRAM has a different meaning within a resource definition than it has at the beginning of a POU of type PROGRAM!

Within a resource declaration, the keywords PROGRAM ... WITH are used to link a task to a POU of type PROGRAM.

```
RESOURCE CPU_Conveyor ON CPU_001
  TASK ...
  PROGRAM ... WITH ...
END_RESOURCE
RESOURCE CPU_Hammer ON CPU_002
  VAR_GLOBAL ... END_VAR
  TASK ...
  PROGRAM ... WITH ...
END_RESOURCE
```

Example 6.2. Elements of the resources in Example 6.6

6.3.4 The TASK with run-time program

The purpose of a TASK definition is to specify the run-time properties of programs and their FBs.

The normal practice with PLC systems hitherto has been to use special types of blocks (e.g. organisation blocks, OBs), with implicit, pre-defined run-time properties. For example, they can be used to implement cyclic execution, or to make use of properties of the PLC system for interrupt handling or error responses.

A TASK definition according to IEC 61131-3 enables these program features to be formulated explicitly and vendor-independently. This makes program documentation and maintenance easier.

Figure 6.5 shows the structure of a textual TASK declaration:

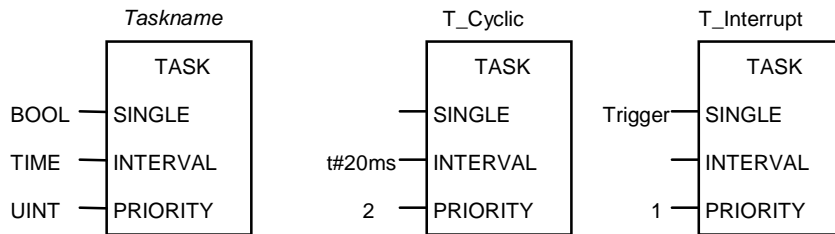**TASK** Task name   (Task properties)

PROGRAM Program name   **WITH**   Task name : (PROGRAM interface)

Figure 6.5. Structure of a textual declaration of a run-time program, defining a task and associating the task with a PROGRAM. The task properties give the parameter values of the task, the PROGRAM interface gives the actual parameters for the formal parameters.

The association of a TASK with a PROGRAM defines a run-time program with the name Program name. This is the instance name of the program of which the calling interface is given in the declaration. This interface includes input *and* output parameters of the POU of type PROGRAM and is initialised when starting the resource.

One PROGRAM can be executed in multiple instances (run-time programs) using such declarations.

A task can be defined textually, as shown above, or graphically, as shown in Figure 6.6. The task properties are shown as inputs to a box, but the graphical representation of the association with a program (PROGRAM...WITH...) is not defined by IEC 61131-3.

Figure 6.6. Graphical declaration of a task. Left: General form. Centre and right: Two tasks from Example 6.6

The input parameters for tasks shown in Table 6.2 are used for specifying the task properties.

TASK Parameter	Meaning
SINGLE	On a rising edge at this input programs associated with the TASK will be called and executed once.
INTERVAL	If a time value different from zero is supplied, all programs associated with the TASK will be executed cyclically (periodically). The value supplied is the interval between two invocations. This value can thus be used to set and monitor cycle time. If the input value is zero, the programs will not be called.
PRIORITY	This input defines the priority of the associated programs compared to other programs running concurrently (multitasking). The meaning is implementation-dependent (see text).

Table 6.2. TASK properties as input parameters

The meaning of the PRIORITY input will depend on how concurrency of multiple programs is implemented in the PLC system, and is therefore implementation-dependent. If a task with a priority higher than that of the task currently executing is activated, there are in principle two ways of resolving this conflict between tasks on the same CPU. It depends on the ability of the PLC system to *interrupt* a running task:

1) The task currently executing is *interrupted* immediately, to start execution of the task with higher priority. This is called pre-emptive scheduling.
2) The task currently executing is *not* interrupted, but continues normally until termination. Only then will the task with the highest priority of all *waiting* tasks be executed. This is called non-pre-emptive scheduling.

Both methods give the task with the highest priority control of the requested resource. If a task with the same priority as the task currently executing is scheduled to execute, it has to wait. If tasks with the same priority are waiting to execute, the one which has been waiting longest will be executed first.

```
TASK T_Quick   (INTERVAL := t#8ms,   PRIORITY := 1);
PROGRAM Motion WITH T_Quick  : ProgA (   RegPar := %MW3,
                                         R_Val => ErrorCode);

TASK T_Interrupt    (SINGLE := Trigger, PRIORITY := 1);
PROGRAM Lube WITH T_Interrupt  : ProgC;
```

Example 6.3. Elements TASK and PROGRAM...WITH... from Example 6.6

In Example 6.3, two tasks T_Quick (cyclic with short cycle time) and T_Interrupt (interrupt task with high priority) are defined.

T_Quick is started every 8 milliseconds. If execution of the program Motion associated with it takes longer than this time (because of interruptions by tasks with higher priority, for example), the PLC system will report a run-time error.

Program Lube has the same priority as Motion, so it has to wait if input Trigger changes to TRUE.

Within the assignment of a program to a task, actual parameters can be specified, as shown here for RegPar of ProgA with a directly represented variable. At run time, these parameters are set on each invocation of the run-time program. In contrast to FBs, output parameters can be specified as well as input parameters, and these are also updated at the end of the program. In Example 6.3, this will be done for R_Val of ProgA. Assignments of *output* parameters to variables are specified with "=>" instead of ":=" to distinguish them from *input* parameters.

If a PROGRAM is declared with no TASK association, this program will have lowest priority compared to all other programs and will be executed cyclically.

6.3.5 ACCESS declarations

The VAR_ACCESS ... END_VAR language construct can be used to define *access paths* to serve as transparent communication links between configurations.

Access paths are an extension of global variables, which are valid within one configuration only. For access paths, read and write attributes can be specified.

Variables of one configuration are made known to other configurations under symbolic names.

```
VAR_ACCESS

  ConvEmpty    :   CPU_Conveyor.%IX1.0        :      BOOL   READ_ONLY;
  ...
  ...
  ...
```

Name of access path — Variable to be accessed from outside — Data type and read/write permission for access path

```
END_VAR
```

Example 6.4. Declaration of an access path

Example 6.4 shows the structure of a declaration of an access path using variable ConvEmpty from Example 6.6.

Access paths can be defined for the following types of variables:

- Input and output variables of a PROGRAM
- Global variables
- Directly represented variables.

Access paths publish these variables under a new name beyond the scope of a configuration, so that they can be accessed using communication blocks, for example.
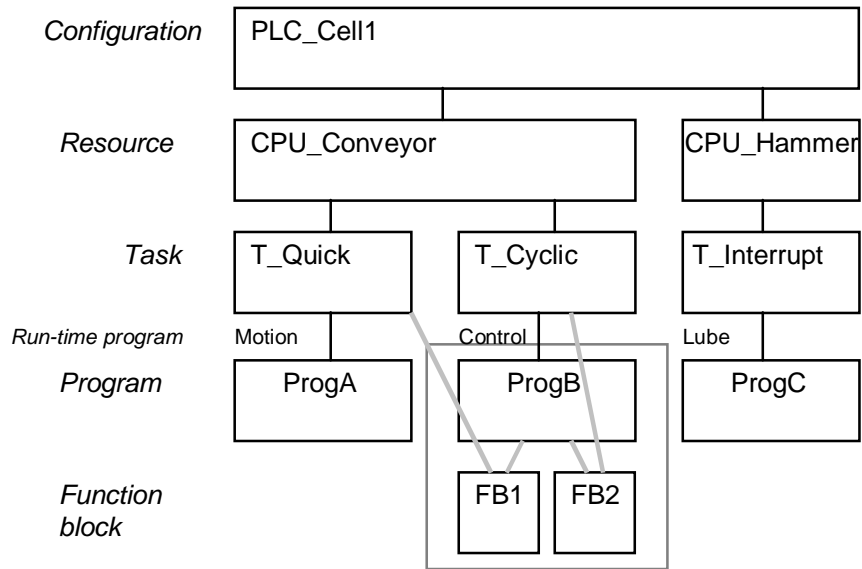
If a variable is a structure or an array, an access path can access only one single member or array element.

By default, access paths allow only read access (READ_ONLY permission). By specifying READ_WRITE, write operations on this variable can be explicitly allowed.

This permission must be specified immediately after the data type of the access variable. The data type of the access variable must be the same as that of the associated variable.

6.4 Configuration Example

Example 6.5 shows an overview of a configuration. This example is declared textually in Examples 6.7 and 6.6. The configuration consists of a PLC system with two CPUs, which are assigned several programs and function blocks. Parts of this example have already been discussed in previous sections of this chapter.

Configuration	PLC_Cell1
Resource	CPU_Conveyor	CPU_Hammer
Task	T_Quick	T_Cyclic	T_Interrupt
Run-time program	Motion	Control	Lube
Program	ProgA	ProgB	ProgC
Function block		FB1 FB2	

Example 6.5. Example of configuration elements with POUs (overview)

In Example 6.6, PLC_Cell1 physically consists of 2 CPUs. The first CPU can execute two tasks: one fast cyclic task with a short cycle time, and one slower cyclic task. The second CPU executes one task with interrupt property.

```
CONFIGURATION PLC_Cell1

VAR_GLOBAL
  ErrorCode    :   DUINT;
  AT %MW3      :   WORD;
  Start        :   INT;
END_VAR

RESOURCE CPU_Conveyor ON CPU_001
  TASK T_Quick      (INTERVAL := t#8ms,   PRIORITY := 1);
  TASK T_Cyclic     (INTERVAL := t#20ms, PRIORITY := 3);
  PROGRAM Motion WITH T_Quick  : ProgA (RegPar := %MW3);
  PROGRAM Control  WITH T_Cyclic : ProgB (InOut := Start,
                                          R_Val => ErrorCode,
                                          FB1 WITH T_Quick,
                                          FB2 WITH T_Cyclic);
END_RESOURCE

RESOURCE CPU_Hammer ON CPU_002
  VAR_GLOBAL
    Trigger  AT %IX2.5 : BOOL;
  END_VAR
  TASK T_Interrupt     (SINGLE := Trigger, PRIORITY := 1);
  PROGRAM Lube WITH T_Cyclic  : ProgC;
END_RESOURCE

VAR_ACCESS
  RegP          : CPU_Conveyor.Motion.RegPar:     WORD READ_WRITE;
  CONV_EMPTY : CPU_Hammer.%IX1.0         :     BOOL   READ_ONLY;
END_VAR

END_CONFIGURATION
```

Example 6.6. Textual declaration of Example 6.5. Names of variables, programs and FBs are printed in bold type.

```
PROGRAM ProgA          PROGRAM ProgB          PROGRAM ProgC
 VAR_INPUT              VAR_IN_OUT
   RegPar : WORD;        InOut : INT;
 END_VAR                END_VAR
                        VAR_OUTPUT
                         R_Value : DUINT;
                        END_VAR
                        ...                    ...
                        CAL Inst_FB2           CAL Inst_FB3
                        ...                    ...
END_PROGRAM            END_PROGRAM            END_PROGRAM
```

Example 6.7. Programs for Example 6.6; FB3 is not shown there, FB1 could implement error handling, for example.

In this example, run-time programs Motion, Control and Lube are created by associating programs ProgA, ProgB and ProgC respectively with a task definition.

Program Motion and FB instance FB1 (independent of program Control) are executed on CPU_Conveyor as quick tasks (T_Quick). FB2 of program Control is executed on the same CPU (CPU_001) as a cyclic task (T_Cyclic). Program Control is used here to define the run-time properties of the FB tasks involved. Function block instances associated with tasks in this way are executed **independently** of the program.

With every cyclic invocation of run-time program Control, the input parameter InOut is set to the value of variable Start. After termination of Control, the value of output parameter R_Val is assigned the variable ErrorCode.

On CPU_Hammer (the second CPU), program Lube is executed as an interrupt-driven task (T_Interrupt). FB3, being part of ProgC, automatically inherits the same run-time behaviour.

In this example, CPU_001 and CPU_002 are not variables, but manufacturer-defined names for the CPUs of PLC_Cell1.

6.5 Communication between Configurations and POUs

This section describes the means of exchanging data between different configurations and within one configuration, using shared data areas.

Such (structured) data areas are used for communication between different program parts, for exchange of data, for synchronisation and to support diagnostics.

It is the aim of IEC 61131-3 to provide a standardised communication model and thus enable the creation of well structured PLC programs, which facilitate commissioning and diagnostics and provide better documentation.

Modularization of applications eases re-use, which helps to reduce the time taken to develop new applications.

IEC 61131-3 defines several ways of exchanging data between different parts of a program:

- Directly represented variables,
- Input and output variables, and the return value, in POU calls,
- Global variables (VAR_GLOBAL, VAR_EXTERNAL),
- Access paths (VAR_ACCESS),
- Communication blocks (IEC 61131-5),
- Call parameters.

The first three methods are for communication *within one* configuration, while access paths and communication blocks are intended for communication *between different* configurations, or with the outside world.

Directly represented variables are not really intended to be used for communication between different parts of an application, but they are included in this list because their use is theoretically possible. Writing to PLC inputs (%I...) is not a suitable method. Outputs (%Q...) should be used to control the process, and not for temporary storage of internal information.

As Table 6.3 shows, these methods can be used at several levels, and the different configuration elements and POUs have different rights of access.

Communication method	CONF	RES	PROG	FB	FUN
Access path	x		x		
Directly represented variable	x	x	x		
Global variable	x	x	x		
External variable			x	x	
Communication block			x	x	
Call parameter		x	x	x	x

Key: CON: CONFIGURATION
        RES: RESOURCE
        PROG: PROGRAM
        FB: FUNCTION_BLOCK
        FUN: FUNCTION

Table 6.3. Communication methods available to configuration elements and POUs.

Access paths are used for exchanging data between configurations, i.e. across the boundaries of one PLC system, and can be used at configuration and program level.

Directly represented variables of a PLC (i.e. %I, %Q and %M) allow limited communication between different parts of an application, as they can be accessed globally on one system. Flags (%M) can be used for synchronising events, for example.

These variables may only be declared at program level or higher (i.e. globally), and function blocks may only access them with an external declaration. This is one important difference between IEC 61131-3 and previous PLC programming practice.

Global variables can be declared for configurations, resources and programs, and can be used at these levels.

Function blocks can access these variables (read and write) with an external declaration, but they cannot declare them themselves. Functions have no access to global or external variables.

External variables can be imported by programs and function blocks if they have been declared globally elsewhere.

Communication blocks are special function blocks used to transfer packets of data from the sender to the recipient. As these FBs are linked to one program, they are local to one configuration and not visible outside.

The definition of such standard communication blocks is contained in Part 5 of IEC 61131 (Communication Services).

Call parameters are used as input and output parameters when calling POUs. They can be used for transferring data into and out of a POU.

As explained in Chapter 2, parameter assignment to input variables and the checking of output variables of a function block can take place independently of its invocation, thereby resulting in characteristics of a communication mechanism which was previously beyond the capabilities of PLC programming.

Resources can pass values to programs when they are associated with tasks, as shown for Motion in Example 6.6. With every invocation, the values are passed as actual parameters or read as output parameters.
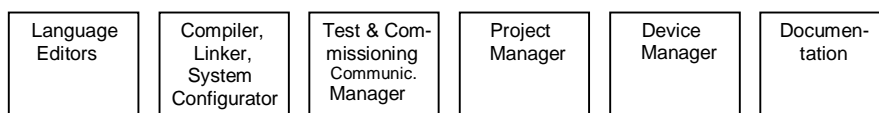
7 Innovative PLC Programming Systems

This chapter goes beyond the specifications of IEC 61131-3 and outlines the general requirements placed on the new generation of PLC programming systems in the marketplace. These mainly stem from the special conditions to be met in the PLC environment using the new standard-compliant technology.

7.1 Requirements of Innovative Programming Tools

The performance of a PLC programming system can be judged by three criteria:

- Technological innovation,
- Fulfilment of PLC-specific requirements,
- Cost/benefit ratio.

This chapter discusses these features and outlines the most important components of a PLC programming system.

Language Editors	Compiler, Linker, System Configurator	Test & Commissioning Communic. Manager	Project Manager	Device Manager	Documentation

Abb. 7.1. Important components of modular PLC programming systems

7.2 Technological Change

The rapid advances in PC technology have influenced PLC programming systems as well as other areas of computing technology. Important new developments are:

1) Increased processor performance,
2) Full-graphics display and printout,
3) High-performance operating systems,
4) Uniform user interfaces.

7.2.1 Processor performance

The increased performance of today's PC processors makes it possible to use advanced graphical editors that can display complex structures graphically on screen.

In addition to the programming languages that have been in use for years in PLC programming – low-level languages like Instruction List and graphical languages like SFC, Ladder or Function Block Diagram – high-level languages can now be employed to create efficient code, making optimum use of special features of PLC processors and PLC operating systems.

7.2.2 Full-graphics display and printout

The screen display or printout can be enlarged or reduced at will using zoom functions. The enhanced resolution compared to character-graphics allows more information to be displayed at once on screen or paper.

The visible screen window shows only a small part of the virtually unlimited working area, which can be scrolled horizontally and vertically. Especially when using graphical programming languages, this enables much more complex logic to be developed.

7.2.3 Operating systems

Systems based on MS-DOS were always hampered by the 640 KB restriction, which could only be evaded by cumbersome custom memory management. Windows, the current standard, dispenses with this limitation.

All the new operating systems support multi-tasking, which allows the programming tools to have multiple tasks executing concurrently, synchronised at defined program points.

7.2.4 Uniform user interfaces

The SAA/CUA industry standard introduced by IBM defined a uniform user interface for software, regardless of whether it was being executed on a personal computer or a mainframe.

Similar functions are invoked by the same keys (e.g. help function via function key F1). A universal screen layout (menus, toolbars...) makes it easier for users to familiarise themselves with different software packages, and they are supported by extensive help systems. Uniform dialog boxes have simplified interaction between users and software.

As Windows (95, 98, 2000, NT) provides the basis for almost all new IEC 61131-3 programming systems, they are becoming increasingly similar for the user to handle. Identical menu items or keyboard shortcuts are used for copying, pasting or finding data. The menu bars have the structure familiar from many standard software packages (Word, Excel for Windows,...).

7.3 Decompilation (Reverse Documentation)

Reverse documentation is a traditional requirement of the PLC market. Ideally, it should be possible to read out a PLC program directly from the controller, in order to modify it on-site at the machine, for example, far away from the office where it was developed. Maintenance personnel want to be able to read, print and modify the PLC program without having the original sources available on a PC.

Decompilation is the ability to retrieve all the information necessary to display and edit a POU directly from the PLC.

IL Code	INTEL 8086 Mnemo-Code (source)

VAR
 V1, V2, Vst: SINT;
 END_VAR
 	
LD   V1
AND   V2
ST   Vst
	
	MOV AL, V1
	AND AL, V2
	MOV Vst, AL

Example 7.1. Example of decompilable source code. Additional information about the variables is necessary (name, address, type)

Decompilation services can be graded according to the facilities offered:

- No decompilation,
- Decompilation with symbols and comments,
- Decompilation including graphics,
- Sources stored in the PLC.

7.3.1 No decompilation

Most new IEC 61131-3 programming systems do not support decompilation. The number of symbolic names required to decompile a program has steadily grown and they cannot be stored in the limited memory available on controllers.

It is rare to find PLCs with processor chips specially developed by the manufacturer (e.g. ASIC or bit slice) and using their own specially written machine code. For cost reasons it is more common to use standard processors. It is much more difficult to decompile the machine code from these standard processors back to Instruction List or Structured Text than it is with custom processors.

It is essential to be able to modify programs after commissioning. It is currently state of the art to keep all information related to a project (sources, libraries, interface and configuration information) on a hard disk drive. Ideally, the sources should be in a language-independent form so that they can be displayed in any programming language. Precautions must be taken in the software to ensure that the program on the controller and the program saved on the hard disk are identical before allowing modification.

7.3.2 Decompilation with symbols and comments

The binary code of a program as read out from the PLC will not suffice to create a compilable source. The PLC should provide lists specifying the current wiring (CONFIGURATION). This includes the assignment of symbolic variables to physical addresses, global variables, the mapping of programs to tasks and resources, etc.

Symbolic information (like variable names and jump labels) is typically not contained in the executable code. A symbol table, created during program development and sometimes including comments on declarations and instructions, must be stored in the PLC to enable decompilation of the program direct from the PLC.

7.3.3 Decompilation including graphics

Tghe PLC contains executable code. To be able to display this code graphically (in Ladder, Function Block Diagram or SFC), the code must either conform to certain syntax rules or it must be augmented by additional information. Using the first

method results in shorter programs, but restricts the facilities for graphical representation.

7.3.4 Sources stored in the PLC

The complex architecture of today's IEC 61131-3 programming systems makes it more and more difficult to pack all the information into the binary code. To have all the information needed for decompilation available on the PLC, a simple solution is to store the entire project information in compressed format in a separate slow, low-cost memory within the PLC. From there, this information can easily be transferred to the PC and edited using the programming system in the same way as during program development.

7.4 Language Compatibility

The five languages of IEC 61131-3 have a special relationship with one another.

Sequential Function Chart with its two methods of representation (textual and graphical) is different from the other languages because it is not used for formulating calculation algorithms, but for structuring programs and controlling their execution.

The logic operations and calculations themselves are defined in one of the other languages and invoked from SFC via action blocks, see Examples 4.51 and 4.52 in Section 4.6.6.

Each program consists of blocks (POUs), which invoke each other by means of calls. These blocks are independent of each other and can be written in different languages, even languages not defined by IEC 61131-3, provided the calling conventions of IEC 61131-3 are observed.

IEC 61131-3 does not go beyond defining common calling interfaces between blocks written in different languages. The question is:

Is it necessary to be able to display code written in one IEC 61131-3 language in another IEC 61131-3 language?

The use of different languages within the same program and the ability to display, print and edit POUs in any IEC 61131-3 language is discussed in the next two sections under the headings *Cross-compilation* and *Language independence*.

7.4.1 Cross-compilation

IEC 61131-3 does not require that a POU developed in one language should be able to be displayed in another language. The argument about the need for this has been going on as long as PLCs have been in existence.

What are the reasons for asking for this feature?
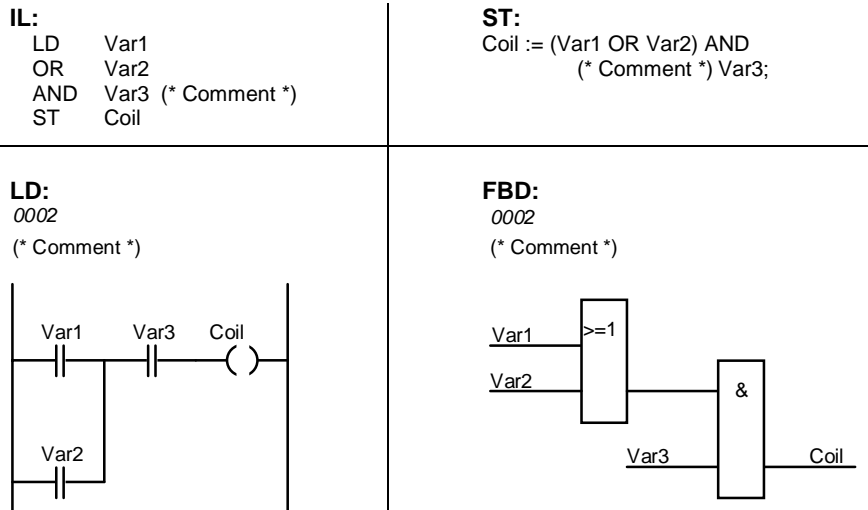
The motivation for cross-compilation
One important reason for wanting to be able to cross-compile parts of a program are the different levels of education and areas of activity of technicians and engineers. Depending on their field, they tend to be trained in different programming languages, making it difficult for them to work together.

In the automotive industry in the US, Ladder Diagram is the preferred language, while the same industry in Europe prefers Instruction List or Structured Text. In the plant construction industry, a functional language like FBD will be preferred. A computer scientist should have no difficulties using Structured Text. So, do we need a different language for every taste, but with editing facilities for all?

Some languages are better suited for certain problems than others. For example, memory management routines are obviously easier to read and write in IL or ST than in Ladder Diagram. A control program for a conveyor is clearer in Ladder Diagram than in ST. SFC is the best choice for a sequential control system.

In many cases, it is not so easy to select the right language. The same section of a program is frequently even needed by different users.

For example, a PLC manufacturer may provide POUs written in IL to support users in handling I/O modules. The user may be a conveyor belt manufacturer, using the PLC to monitor and control limit switches and motors, and preferring to work with Ladder Diagram. So the programmer modifies the code provided in IL to his needs, using Ladder Diagram to do so. The conveyor may then be supplied to a plant construction company where all programs are written in FBD, and the I/O control programs will be required for complete and uniform documentation.

```
IL:                                        ST:
   LD      Var1                            Coil := (Var1 OR Var2) AND
   OR      Var2                                      (* Comment *) Var3;
   AND     Var3  (* Comment *)
   ST      Coil
```

LD:
0002

(* Comment *)

FBD:
0002

(* Comment *)

Example 7.2. Example of cross-compilation between four different languages of IEC 61131-3

Different approaches in graphical and textual languages.
One difficulty with cross-compilation lies in the different ways of looking at a calculation. LD and FBD have their roots in Boolean or analogue value processing: there is "power flow", or not; values are propagated and calculated in parallel. Textual languages, like IL and ST, are procedural, i.e. instructions are executed one after the other.

This becomes obvious when looking at the network evaluation in Section 4.4.4. Example 7.3 gives an example in Ladder Diagram (FBD would be similar).

```
LD      Var1
JMPC Label1
ST      Var2
AND   Var3
JMPC Label2
```
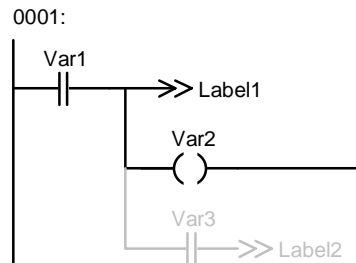

a) IL   b) LD

Example 7.3. Sequential execution of a program section in IL compared to parallel execution of a network in Ladder Diagram. This makes cross-compilation difficult.

According to the evaluation rules given for graphical languages (see Section 4.4.4), Var2 will always be assigned a value. If this network is converted from IL as it stands, Var2 will be assigned a value only if Var1 equals FALSE. Otherwise it would be necessary to rearrange the elements before cross-compiling (all ST instructions before a conditional JMP or CAL). This would change the graphical appearance of the network when cross-compiled to Ladder Diagram.

Looking at the procedural (IL) sequence and the simultaneous evaluation in Example 7.3, the IL sequence converted to Ladder is misleading. When Var1 := Var3 := TRUE, Label1 and Label2 are TRUE. The IL sequence jumps to Label1; in the Ladder Diagram version both labels are addressed in accordance with the evaluation rules of Ladder, and the next network to be activated is unclear.

This problem of cross-compilation is solved in many of the programming systems that possess this functionality by:

- not allowing further logic operations after an assignment in graphical languages,
- evaluating the code part of a graphical network from top to bottom (Example 4.38) and stopping evaluation when a control flow instruction is being executed.

Differences in languages affect cross-compilation.
Not all of the languages can be cross-compiled to each other. SFC is a language for structuring applications, making use of the other languages, but having a completely different design. We shall therefore only discuss cross-compilation between IL, ST, LD and FBD.

Restrictions in LD/ FBD.
Jumps can be made using labels, but are somewhat contradictory to the concept of "parallel" networks. Some functions, like management of system resources (stack operations), can only be expressed in very complicated, unreadable programs.

Constructs like CASE, FOR, WHILE, or REPEAT are not available in these languages and can only be implemented by using standard functions like EQ and complex network arrangements.

Unlike ST, these two languages allow only simple expressions to be used to index arrays.

LD is designed to process Boolean signals (TRUE and FALSE). Other data types, like integer, can be processed with functions and function blocks, but at least one input and one output must be of type BOOL and be connected to the power rail. This can make programs hard to read. For non-Boolean value processing FBD is better suited than LD.

Not all textual elements have a matching representation in the graphical languages. For example, some of the negation modifiers of IL are missing in LD and FBD: JMP, JMPC, RET and RETC are available, but JMPCN and RETCN are not. These can be formulated by the user with additional logic operations or supplementary (non-standard) graphical symbols can be included in the programming system.

Restrictions in IL/ ST.
The notion of a network, as used in the graphical languages LD and FBD, is not known in IL or ST.

In Ladder Diagram, attributes like buffering of variables (M, SM, RM) or edge detection (P, N) can be expressed by graphical symbols. This representation of attributes in the code part of a program does not comply with the strict concept of expressing the attributes of variables in the declaration part. There are no matching elements in the textual languages for these graphical symbols.

The use of EN and ENO poses another problem, as no matching element is available in IL or ST. Each user-defined function must evaluate EN and assign ENO a value to be useable in graphical languages. If EN and ENO are used in graphical languages and not used in textual languages, two versions of standard functions are needed (with and without EN/ENO processing), see Section 2.5.2.

Cross-compilation IL / ST.
A high-level language (ST) can be converted more easily and efficiently into a low-level, assembler-like language (IL) than vice versa. In fact, both languages

have some features that make cross-compilation into the other difficult. For example:

- ST does not support jumps, which have to be used in IL to implement loops. This makes cross-compilation of some IL programs difficult.
- IL supports only simple variables for array indices and actual parameters for function and function block calls, whereas in ST complex expressions may also be used.

Full cross-compilation only with additional information.
If a system in Ladder Diagram allows multiple coils and/ or conditional instructions with **different** values (corresponding to a one-output to many-input situation in FBD), see Example 7.4, auxiliary variables have to be used to cross-compile to IL. When cross-compiling this code part from IL or ST to ladder diagram, care has to be taken to avoid multiple networks being generated in the Ladder Diagram version.

Small modifications of an IL program can, therefore, result in major changes in the Ladder cross-compiled form.

```
LD     Var1
ST     Helper1
AND    Var2
ST     Var3
LD     Helper1
AND    Var4
ST     Var5
```
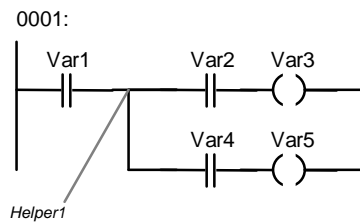

a) IL   b) LD

Example 7.4. If coils or conditional instructions are used in an LD network with different value assignments, direct cross-compilation is not possible. To cross-compile to IL, an auxiliary variable like Helper1 is necessary.

```
LD      Var1                                      0001:
OR (    Var3
ST      Helper      (* typically not allowed *)
AND     Var4
        )
AND     Var2
OR (    Helper
AND     Var5
        )
ST      Var7
```

a) IL   b) LD
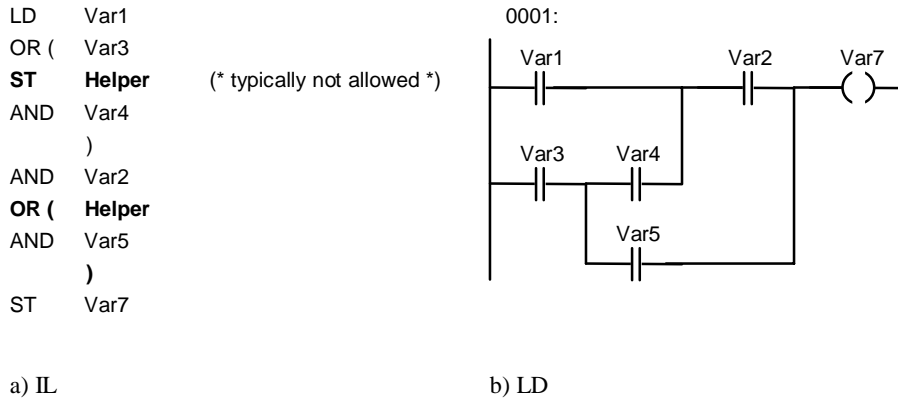
Example 7.5. Cross-compilation is not possible for graphical Ladder constructs which, when cross-compiled to IL, would result in improper nesting of expressions.

The Ladder network in Example 7.5, which looks perfectly clear in this graphical language, is not directly cross-compilable to IL. An auxiliary variable, like Helper, is necessary to temporarily store the intermediate result between Var3 and Var4, and several parentheses are needed.

Some programming systems automatically place graphical elements in the correct (optimum) position, depending on their logical relation. Other systems allow and require users to position the elements themselves. This topographical information needs to be stored in the form of comments in IL or ST, if a graphical representation is to be regenerated from the textual representation later. IL or ST programmers cannot reasonably be expected to insert this topographical information. In most programming systems, the information about the position of the graphical elements is therefore only kept in internal data storage and is lost when the program is cross-compiled to IL or ST. In most cases, there is no way back from the textual languages to the original graphical layout.

Quality criteria for cross-compilation.
It has been explained that full cross-compilation in the theoretical sense cannot be achieved. The quality of a programming system with respect to cross-compilation depends rather on how well it meets the following conditions:

1) The rules of cross-compilation should be so easy to understand that the programmer can always determine the result.
2) Cross-compilation must not change the semantics of the POU (only local modifications, entities must stay together).
3) Cross-compilation should not affect the run time of the program.
4) Cross-compilation must not introduce side-effects (i.e. not affect other parts of the program).
5) Ambiguities must be resolved (see Example 7.3).

As cross-compilation is not easy to achieve, many programming systems do not implement it.

7.4.2 Language independence

The POU, as an independent entity, is an important item of IEC 61131-3. To invoke a POU in a program, only the external interface of the POU needs to be known, but no knowledge about the code contained within it is required. To design a project top-down, it should therefore be possible to define all the interfaces of the POUs first and fill in the code later.

Once the external interface of a POU has been defined, it is typically made available to the whole project by the programming system. It consists of:

- the function name and function type, plus the name and data type of all VAR_INPUT parameters for a FUNCTION,
- the function block name, the name and data type of all VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT parameters and EXTERNAL references for a FUNCTION BLOCK,
- the program name, the name and data type of all VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT parameters and GLOBAL variables for a program.

One POU can invoke another function or function block instance without knowing which language the other POU has been programmed in. This means that the programming system does not have to provide a separate set of standard functions and function blocks for each programming language.

This principle can even be extended to languages outside the scope of IEC 61131-3 as the caller needs no knowledge of the block apart from its external interface. If the external interface and the calling conventions of an IEC 61131-3 programming system and a C compiler, for example, are compatible, it is equally possible to invoke a C subprogram.

IEC 61131-3 expressly allows the use of other programming languages.

7.5 Documentation

Different types of documentation are required to allow for efficient maintenance of applications and to support modern quality standards like ISO 9000:

1) *Cross-Reference List*. A table listing which symbols (variable name, jump label, network title, POU type or instance name etc.) are being used in which POUs.
2) *Program Structure*, giving an overview over the calling hierarchy of POUs. Each POU has a list giving the names of all POUs invoked from it. The Program Structure can be visualised graphically, or textually, with a nesting depth depending on the system.
3) *Allocation List* (*Wiring List*). A table giving the physical addresses of I/Os and the names of the variables assigned to these addresses.
4) *I/O Map.* A table of all I/O addresses used by the application, sorted by address. The I/O map is helpful in finding free I/O addresses when extending an application and for having the relation between PLC software and PLC hardware documented in a hardware-oriented manner.
5) *Plant Documentation*. A description of the entire plant, typically graphical. The entire plant contains multiple PLCs, machines, output devices etc. Each individual PLC will be only one "black box" in this documentation. The plant documentation is often generated with standard CAD programs, giving the topological grouping and connections between PLCs and other devices.
6) *Program Documentation*. Sources of POUs created with the programming system. When printed, these should closely match in structure and contents the representation on-screen while editing.
7) *Configuration*. The Configuration – as understood by IEC 61131-3 – describes which programs are to be executed on which PLC resources and with what run-time properties.

These types of documentation are neither required nor standardised by IEC 61131-3, but have become popular over the years for documenting PLC programs.

7.5.1 Cross-reference list

The cross-reference list consists of:

-   all *symbolic names* used in a program (or occasionally in an entire project),
-   the *data type* of all variables, which may be a *function block type* if a variable is a function block instance, plus declaration attributes (like RETAIN, CONSTANT or READ_ONLY) and the variable type (VAR_INPUT, ...),
-   the name and type of the POU, and the line number, for each usage of a variable,

- the kind of access to this variable at this program location,
- the location of the declaration of this variable.

Some systems support a cross-reference list only for individual POUs, or provide it only for global and external data.

The cross-reference list is helpful in finding program locations referencing variables during debugging.

Different sorting criteria are usually supported. The entries are usually sorted alphabetically by symbolic name. They can also be sorted by symbol type (input, output, function name, instance name,...) or data type (BYTE, INT,...).

Symbolic name	POU name	POU type	Line No.	Access	Data type	Attribute	Var type
Temp_Sensor	SOND.POE	Prog	10	Decl	INT	Retain	GLOBAL
	SOND.POE	Prog	62	Read			GLOBAL
	CONTR.POE	FB	58	Write			EXTERN.
. . .							

Example 7.6. Example of a cross-reference list sorted by symbolic name

7.5.2 Allocation list (wiring list)

The allocation list lists all variables which are assigned to physical I/O addresses of a configuration, plus the access path, if supported.

Sometimes, tools are provided for changing the assignment of symbols to physical addresses (rewiring). This often needs to be done when porting an application to another environment (e.g. another PLC with different I/O connections).

Symbolic name	I/O address
Temp_Sensor_1	%IB0
Temp_Sensor_2	%IB2
Temp_Control	%QB4
Temp_Save	%MB1
. . .	

Example 7.7. Example of an allocation list (wiring list)

7.5.3 Comments

Comments are an important part of program documentation. Application sources can be enhanced by descriptive comments at many locations (see Example 7.2):

- In ST, comments can be inserted wherever space characters are allowed,
- In IL, comments can be added at the end of every line,
- IEC 61131-3 does not include any guidelines on comments in the graphical languages. However, network comments, preceding and describing a network, are a valuable aid in documentation.

Some programming systems have menu settings to prevent (accidental) overwriting of the program and only allow changes to be made in the comments.

7.6 Project Manager

The task of the Project Manager is to consistently manage all information related to the implementation of a project. This includes:

- *Source information*
    - The sources of all POUs created, with
        - type definitions, declarations of global data, definition of access paths with VAR_ACCESS,...
        - descriptions of the call interfaces of all POUs in order to check their usage,
    - Version control of all sources,
    - Access restrictions, sometimes with different access levels authenticated by passwords, for:
        - modifying POUs,
        - printing programs,
        - editing libraries.
- *Object information*
    - Compiled sources (intermediate code, object code, executable files),
    - Project creation procedures (call-dependence, creation and modification information for controlling time-dependent compiling operations, for example in MAKE or batch processes),
    - Libraries (standard functions, standard function blocks, manufacturer defined blocks, communication function blocks, user libraries).

- *Online information*
    - Data for assigning parameters to the project (recipes),
    - Device and configuration information (PLC hardware, I/O modules,...),
    - Additional information for online testing (symbol information, breakpoints,...),
    - Communication information (protocols, interfaces).
- *Documentation* (e.g. cross-reference list, allocation list, program structure,...).

The Project Manager administers and archives all this data. Why is a standard file manager (e.g. the Windows Explorer) not sufficient?

POUs are interdependent. Multiple programs within one project can use the same POU, although it only exists as a single file. Function names and function block types have global scope throughout a project (whereas the scope of function block instances is limited to the POU they are defined in, unless they are explicitly declared GLOBAL).

For every invocation of a POU (instance or function) the compiler and editors need interface information about the POU being invoked (types and names of parameters). In order to reduce the overhead of gathering the same information again and again every time it is needed, the Project Manager can store such information and supply it to other parts of the programming system when requested.

Figure 7.2 shows the directory of a hard disk drive (D:), containing sub-directories. In order to visualise the structure of a project, it is necessary to evaluate interdependences between calls and environmental factors to enable the relations to be displayed, see Figure 7.3.

The programming system should assist the programmer in understanding the structure of the project. One project can contain several configurations (PLCs), and each configuration can contain several resources (CPUs). See Chapter 6 for the description of a configuration. Each resource will have a specification of the hardware associated with it (Resource1 File). Each resource can execute multiple programs (Program1, ...), which are implemented in the form of POUs invoking other POUs. Two distinct instances of one POU, as shown in Figure 7.3, contained in different programs, can be described by the same POU stored in only one file on disk.
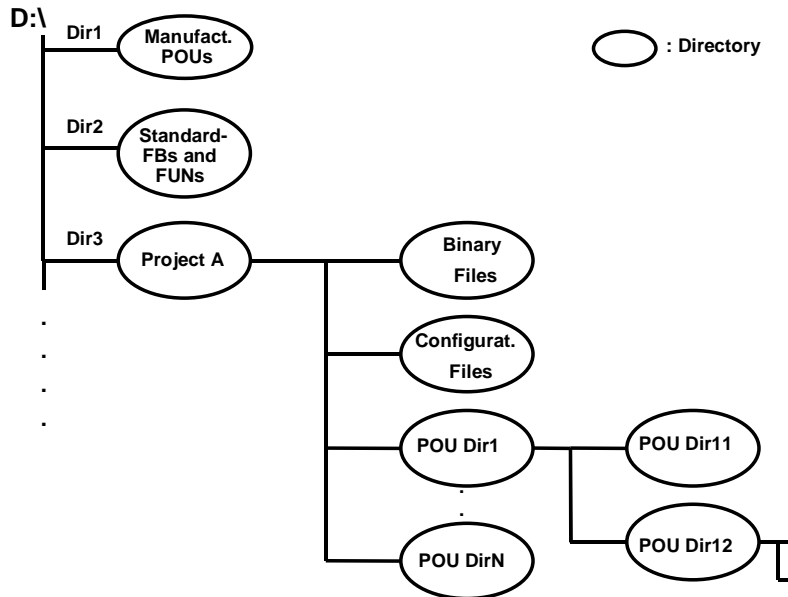
Figure **7.2.** Data for sources and libraries stored on disk (shown as drive D:\). The programming system should allow the user to create directories and sub-directories to resemble the project structure as far as supported by the operating system (e.g. Windows).

Rules can be established for finding information. For example, to compile Program1 for Resource1, all user, system and manufacturer POUs used in the program have to be collected. If one of the POU names called appears more than once on the disk, the system has to take appropriate action (e.g. apply a rule to choose one, or issue an error message); IEC 61131-3 does not discuss this problem.

For efficiency reasons, it is desirable to recompile only a minimum set of sources to create a new version of a resource, configuration or program after a change. For example, if, only Program1 has been modified, other programs, or even other resources, that do not need the modified POU need not be re-compiled.
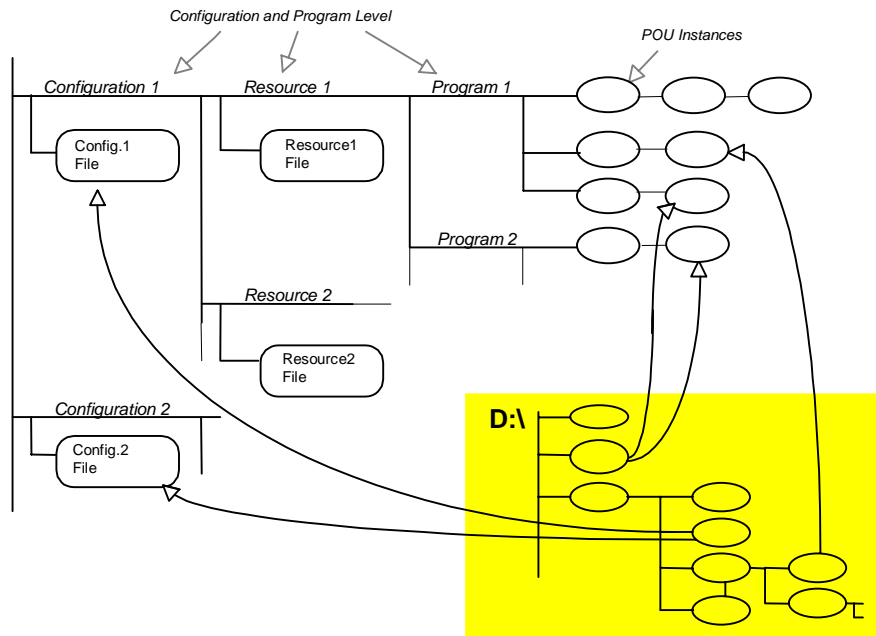
Figure 7.3. The logical structure of a project consists of configurations, resources and a POU calling hierarchy. As this structure in most cases is not identical with the physical file and directory structure, e.g. Figure 7.2, additional visualisation aids are helpful.

A Project Manager should therefore support

- registration of newly created files (sources, error logs, interface descriptions),
- import of files from different projects,
- display of all existing POUs,
- renaming and deletion of POUs,
- an information structure which makes the structure of the project (calling hierarchy, dependencies) understandable to the user,
- maintenance of all information (POUs, interfaces, calling hierarchy, binary files, etc.) that the rest of the programming system needs for creating a project.

Some IEC 61131-3 programming systems store data in individual files (e.g. one file per POU) on the hard disk, while others use a database. Either way, project files should only be edited via the Project Manager. To increase efficiency when supplying data in response to requests (e.g. from the linker), some information can be pre-processed, so that it is not necessary to scan the entire file system for each request.

7.7 Test & Commissioning Functions

The first stage of program development is typically performed on a PC, without a PLC. When the most important parts of the program have been completed, work is continued with the PLC in the target environment. The following are typical tasks performed at this commissioning stage:

- Download of the entire project or individual POUs (after modifications) to the PLC,
- Upload of the project from the PLC to the PC,
- Modification of the program in the PLC (either in "RUN" or "STOP" mode),
- Starting and stopping the PLC,
- Display of variable values (status),
- Direct setting of I/Os or variables (forcing),
- Deactivation of the physical outputs of the PLC, to prevent unsafe plant conditions during tests. Programs are executed and values are assigned to direct variables just as they would be in normal operation. Additional software or hardware ensure that physical outputs are not influenced by values written to the output variables.
- Retrieving PLC system data, communication and network information from the PLC,
- Program execution control (breakpoint, single step,...).

These functions are implemented in different ways and to different degrees by individual programming systems. IEC 61131-3 does not stipulate any requirements with respect to these features (yet).

7.7.1 Program transfer

After a POU has been created with the editor and been checked for syntax, PLC-specific code is created. This can take the form of machine code for direct execution by the PLC's processor, or it could consist of instructions to be interpreted by the PLC. The programming system puts this object code together with the object code of other POUs to make a program (link procedure).

All that remains to be done is:

- Connect the executable code with a defined *CONFIGURATION* (task association),
- Map the logical hardware addresses to the actual physical addresses of the PLC,
- Assign parameters to the PLC's operating system.

These tasks can be performed by a tool called a *System Configurator*, or in some implementations the PLC's operating system can do some of the work.

The entire project now needs to be transferred to the PLC. A tool known as the *Communication Manager* establishes a physical and logical connection to the PLC (e.g. a proprietary protocol on port COM1, or a fieldbus protocol). This tool performs some checks, which are partly invisible to the user. For example:

- Has contact been successfully established with the PLC?
- Does the PLC's current state allow transfer of new programs? Or can the PLC be put into the correct state?
- Is the PLC's hardware configuration compatible with the requirements of the program?
- Is the current user authorised to access this PLC?
- Visualisation of the current state of communication.

The program is then downloaded to the PLC, together with all the information needed to run it. The PLC program may now be started (cold restart).

7.7.2 Online modification of a program

If it is necessary to modify blocks whilst the program is running (PLC in "RUN" mode), this can be done in various ways:

- By changing the POU on the PC and compiling the whole program again with the programming system. Everything is then downloaded to the PLC. If this is to be done in "RUN" mode, a second memory area must be available and activated after the download is completed, as the download generally takes too long to suspend execution of the cyclic PLC program.
- By modifying the POU on the PC and downloading only the modified POU to the PLC. This requires a block management function to be available on the PLC, which will accept the new block and replace the old block with the new one after the download has been completed. As the old and new blocks usually differ in size, a "garbage collection" is required from time to time in order to be able to re-use memory space which would otherwise be wasted.
- By replacing only individual networks (SFC, LD, or FBD). This is only possible if other parts of the POU are not affected. E.g., jump labels in other networks must not move if the PLC operating system does not include jump label management.

7.7.3 Remote control: Starting and stopping the PLC

The PLC hardware typically features a switch to start and stop the PLC. This can be remote-controlled from the programming system.

IEC 61131-3 defines different start modes for a PLC (see Section 3.5.3):

1) *Cold Restart*. The PLC starts the program without memorising any variable values. This is the case, for example, after downloading the program to the PLC.
2) *Warm Restart*. Following a power outage, program execution is resumed at the point where it was interrupted (e.g. in the middle of an FB). All variables carrying the "RETAIN" attribute retain the value they had before the interruption, all other variables are reset to their initial value.
3) *Warm restart at beginning of program.* The values of all RETAIN variables are also retained and all other variables are re-initialised, but the program is restarted at the beginning. This takes place automatically if the interruption time exceeds a parameterised time limit.

Additional commissioning features are:

- Stopping of the PLC, either with the current output values or with "safe" output values,
- Deletion of memory areas to prevent uncontrolled restarts,
- Selection of special operating system modes, e.g. test mode, maximum cycle time, etc.
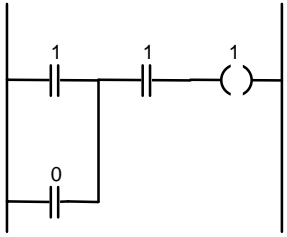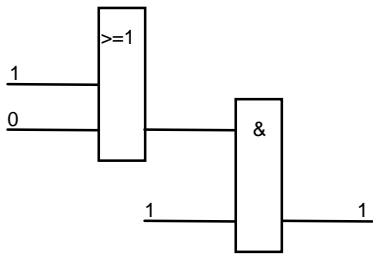
7.7.4 Variable and program status

The most important test function for debugging and commissioning a PLC program is the monitoring of the status of variables ("flags") and external I/Os. Ideally, values should be displayed in a user-selectable form, as shown in Example 7.8.

Variable	Type	Value
Coil	BOOL	1
Var1	BOOL	1
Var2	BOOL	0
Var3	BOOL	1

a) Variable List

LD	Var1	1
OR	Var2	**0**
AND	Var3	1
ST	Coil	1

b) IL (Variable Status)

Coil := Var1 OR Var2 AND Var3
     1   1   **0**   1

c) ST (Variable Status)

0002:
(* comment *)

d) LD (Variable Status)

0002:
(* comment *)

e) FBD (Variable Status)

LD	Var1	1
OR	Var2	**1**
AND	Var3	1
ST	Coil	1

f) IL (Current Result)

Coil := Var1 OR Var2 AND Var3
    1
       1
        1
           1

g) ST (Expression)

Example 7.8. Status view of the sample program from Example 7.2 during commissioning, shown in the different programming languages. The different display modes can be combined. (Example continued on next page).

0002:
(* comment *)

0002:
(*comment *)

h) LD (Power Flow Display)   i) FBD (Program Status)
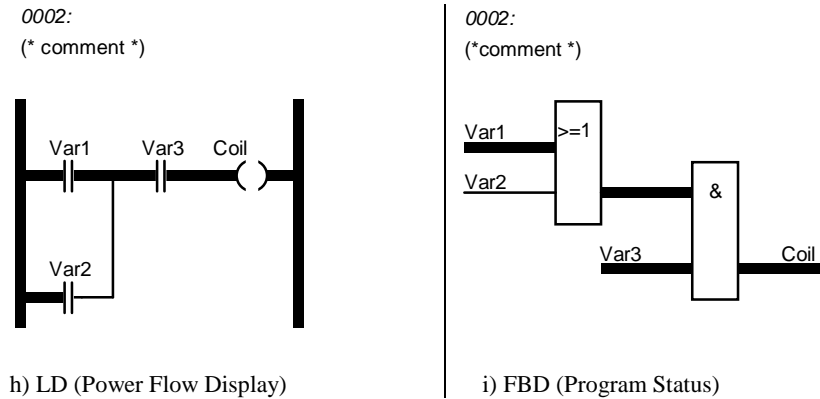
Example 7.8. (Continued)

Depending on the implementation in the PLC and/ or the programming system, there are different methods of viewing the current data and execution flow:

1) *Variable List*: Display of a list of variables (a). The variables contained in the list are scanned in the PLC and their values continuously updated on the screen. This method is frequently used for monitoring values from different parts of the program. Structured variables can also be displayed showing the values of their individual members.

2) *Variable Status*: All variables of a specific code portion are displayed. In Example 7.8, the values of Var1, Var2, Var3 and Coil are displayed in a separate window (a) or directly within a graphic (b-e).

3) *Program Status* (also *Power Flow*). Instead of displaying the values of individual variables, the result of each operation is displayed (the Current Result, see Section 4.1.2) (f-i). In the graphical languages, this is done by drawing thick or thin lines to represent TRUE and FALSE for Boolean expressions or displaying numerical values beside each line.

The quality of values provided will depend on the functionality, the speed and the synchronisation between the programming system and the PLC. Depending on the operating system and the hardware features available, the values on the PLC can be "collected" at different times:

- Synchronously,
- Asynchronously,
- On change.

Synchronous Status: Values are collected at the end of a program cycle. The values are all generated at the same point in the execution of the program, and are therefore consistent. This is also the time when variables mapped to I/O addresses are written to the outputs (update of the process image). A cyclic program will in most cases execute much faster than values can be retrieved by the programming

system, so the values viewed with the programming system are only updated every *n* cycles.

Asynchronous Status: The programming system constantly requests a new set of values for the specified variables (asynchronously to the cycle of the PLC program). The values are collected at the moment when the request is received. Each value is a snapshot of the respective memory location.

Status on change: Values are only be collected when they change (e.g. a signal edge). This requires special hardware (address monitoring) within the PLC.

Advanced systems supply additional analysis tools. These allow the behaviour of variable values to be visualised over time (logic analysis) as shown in Figure 7.4.
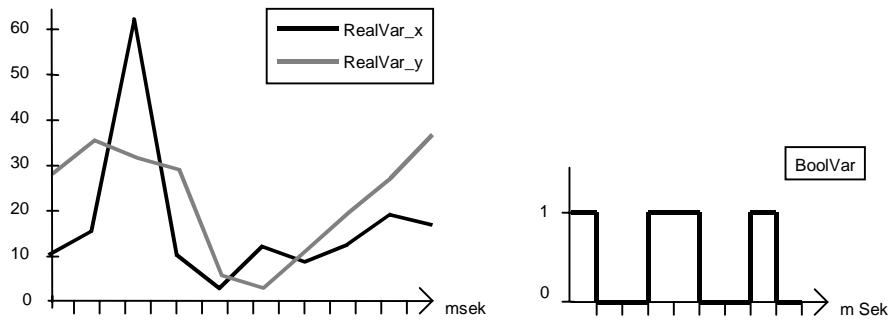
Figure 7.4. Example of data analysis of two REAL variables and one BOOL variable

A facility that is not often implemented as yet is the visualisation of the data flow between individual POUs, as shown in Figure 7.5.
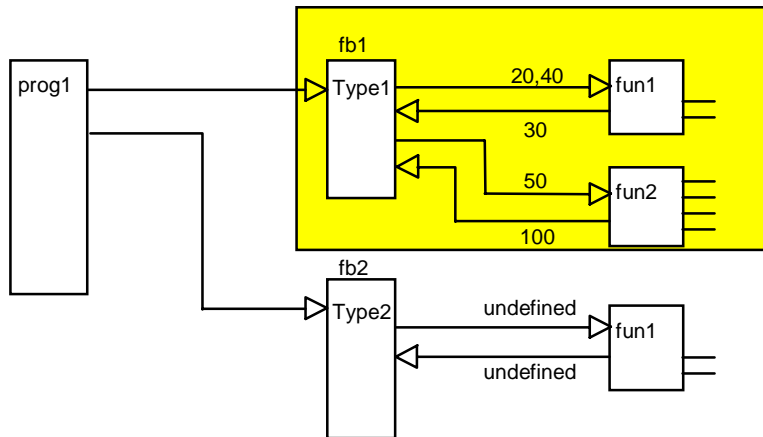
Figure 7.5. Display of the calling hierarchy of the POUs together with the actual parameters during execution of the PLC program.

When requesting variables to be displayed, it is not sufficient to specify the name of the POU in which they occur. In the example shown in Figure 7.5, the name "fun1" would not be sufficient to identify which variables to display, as this POU is called twice (*Type1* and *Type2*) and will return different values depending on the call parameters. For unambiguous identification it is therefore necessary to specify the call path and, in the case of functions, even the call location.

Additional difficulties arise from the fact that the local variables of functions only exist during execution of the function, and are otherwise undefined.

7.7.5 Forcing

To test the behaviour of programs and plant it is helpful to be able to force variables to specific values.

To simulate a specific plant condition for the program in order to test certain program parts, some variables are set to fixed values using the programming system, and the PLC is made to use these values instead of the actual values.

Local Variables

Variable	Data Type	POU Instance	Assigned Value
Var_Loc	BOOL	FB1	1
Var3	INT	FB1	20
...			

Directly represented variables (WORD)

%IW0: **0 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0**   *(Bits 0 to 15)*

%IW1: **0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0**   *(Bits 16 to 31)*

...

Symbolic Variables (Integer)

Start Address	Variable Name	Assigned Value
%QW20:	LimitValue	11.233
...		

Example 7.9. Example of the forcing of variables and I/O addresses (Boolean and integer values). When specifying the name of the "POU instance", it may be necessary to specify the calling hierarchy, or even the program location in the case of functions, see Section 7.7.4.

As long as the PLC is kept in "forcing" mode by the programming system and with the parameters set as shown in Example 7.9, the program will always find the value Boolean 1 when reading %IX0.2 or %IX0.6.

Depending on the implementation in the PLC, the variables being forced are either set to the forced value only once at the beginning of every cycle and the program itself can overwrite them during the cycle, or they are kept to the forced value throughout the cycle (overwriting is prevented).

7.7.6 Program test

Breakpoints and Single Step, functions well known from PC development environments, can also be used for debugging PLC programs:

- *Breakpoint Set/ Reset*. The user specifies a location in the program where execution is to be interrupted and further instructions from the user awaited. Advanced systems support conditional breakpoints, e.g. "Break at line 20 of block FB4 if Function_1 has been called, Variable_X has been set and Function_2 has been reached.". As mentioned before, specifying the line number and POU or instance name is not always sufficient, but the location in the calling hierarchy should be specified, see Figure 7.5.
- *Single Step*. After stopping at a breakpoint, the user can execute the following instructions or graphical elements one at a time.

7.7.7 Testing Sequential Function Chart programs

Special features are required to test programs written in SFC. Depending on the system, the following features are available:

- Setting/ Resetting transitions, to force or prevent a transition to the next step.
- Activation of steps or transitions, to begin execution of a POU at a specific location.
- Permanent blocking of steps or transitions, to prevent them from being activated.
- Reading and modifying system data that is relevant to SFC, e.g. step flags, step timers, etc.

This is the equivalent of forcing at SFC level.

7.8 Data Blocks and Recipes

A *recipe*, in PLC parlance, is a set of variable values, which can be replaced by a different one, to make the same PLC program behave differently. For example, in an automated process, only parameters like length, weight, or temperature need to be changed to produce a different product. Using recipes, such process data can be modified during operation.

The replacement of one set of data by another can be performed by the program itself. In this case, all the possible data sets have to be stored in PLC memory. Another possibility is to download a new data set to the PLC at a convenient point during operation, replacing an existing data set.

Some PLC systems use *Data Blocks* (DB) for this purpose. A data block is accessible throughout the project (global) and consists of data items of specific data types.

The data block number gives the program the base address of the data block that is currently active. The data items are accessed by an index (data word number). To switch to a different data block, only the base address of the new block has to be selected in order to activate it.
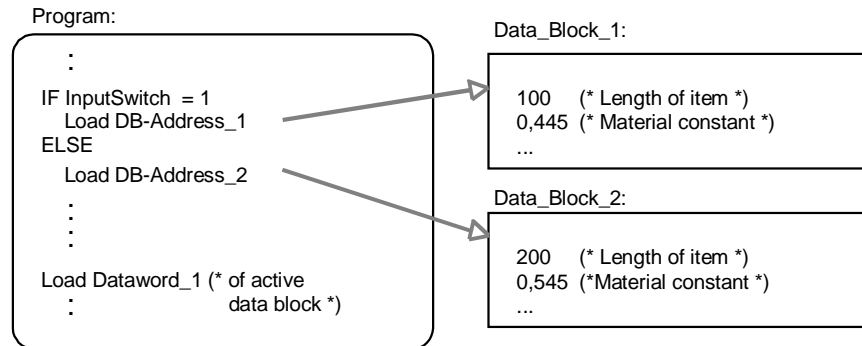
Figure 7.6. Using data blocks. Exactly one data block can be active at any one time.

In Figure 7.6, the instruction "Load Dataword_1" will load 100 or 200, depending on which data block is active.

One of the main uses of data blocks hitherto has been as data areas for function blocks: before calling the FB, the relevant data block (recipe) is selected and is then used by the FB for fetching parameters and holding temporary or internal values.

Common features of data blocks can be summarised as follows:

- DBs can be downloaded and replaced separately like any other block,
- DBs are globally accessible by any other block,
- Exactly one DB is active at any one time,
- DBs contain data type information about their data items.

As the subject of data blocks has not been raised until this late stage of our book, it is only natural at this point to ask: "Where are the DBs in this new standard?"

IEC 61131-3 does not discuss data blocks. Instead of activating one global data block to hold parameters and local information for a function block, instantiation of function blocks is used (see Chapter 2). Each instance of a FB is automatically assigned its own "local" data record. FBs can also access global data. IEC 61131-3 therefore fully covers the functionality of data blocks as a means of assigning parameters to function blocks and storing local data.

However, unlike POUs, the instance data areas of FBs **cannot** be replaced or initialised separately. The standardisation committee is aware of this deficiency and will cover it in Technical Report 2.

Ways of implementing most of the features of DBs using the methods of IEC 61131-3 are outlined here.

Data that belongs together is declared as a structure or an array. Like data blocks, these consist of a number of values, each having a data type. Such compound variables can be used as input or output parameters, local or global data. Data structures like this can also conveniently be used to implement recipes.

Switching between different sets of data can be done in several ways:

1) The PLC is stopped and a complete new program with different data is downloaded to the PLC. This method requires the PC to be permanently connected to the PLC and the process under control must be able to tolerate such interruptions.
2) Individual blocks (POUs) are replaced. The POUs downloaded from the PC replace POUs with identical names in the PLC, but have different initial values for the variables contained or call different POUs. The PLC's operating system must have a block management facility that allows replacement of individual blocks during operation.
3) Remote SCADA software is used to dynamically modify the (global) set of data.
4) All the sets of data required are contained within the PLC. The program invokes POUs with different sets of data or assigns its function blocks different parameter sets depending on the situation.

Replacing POUs seems appropriate for POUs which mainly:

- provide global data,
- copy or initialise data.

Global and external variables are a common way of supplying parameters to parts of a program, e.g. function blocks:

The disadvantage of using GLOBAL declarations is that all the data sets have to be stored in the PLC all the time. Systems that support the configuration of global variables can avoid this problem by allowing a resource to be supplied with values for its global data by another resource or via access paths.

```
Type Definition:              Main Program                    Using POU:

TYPE                          PROGRAM main                    ...
   rec_struct:                                                FUNCTION_BLOCK Work
   STRUCT                     VAR_GLOBAL                        VAR EXTERNAL
      par1:int;                 recipe:rec_struct;                Recipe: rec_struct;
      par2:int;               END_VAR                          END_VAR
      ...                     VAR
   END_STRUCT;                  rec1: rec_struct := ...;           ...
END_TYPE                       rec2: rec_struct := ...;           LD recipe.par1
                               Work1: Work;                       ...
                             END_VAR

                             IF Condition THEN
                               recipe := rec1;
                             ELSE
                               Recipe := rec2;
                             END_IF;

                                ...
                             Work1 ();
                                ...
```
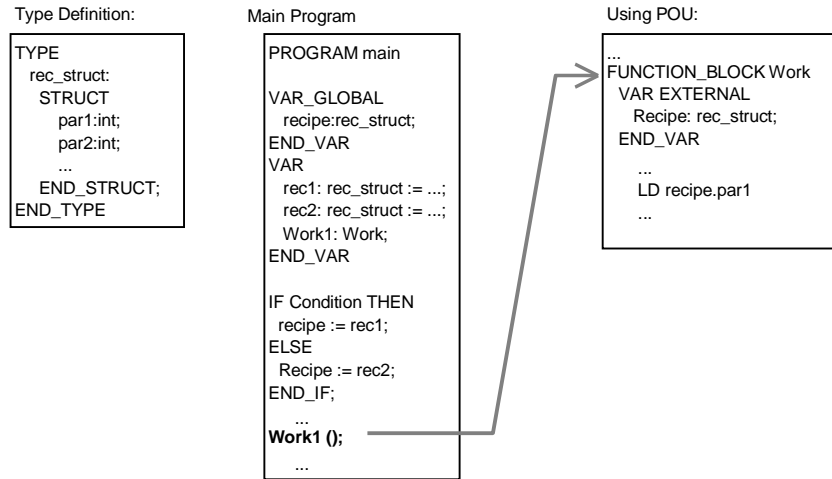
Example 7.10. The main program contains different sets of data (rec1, rec2). The global structure recipe is assigned one of these depending on a specified condition. Other POUs simply access the global variable with a corresponding EXTERNAL declaration.

Global data is inherently error-prone when programs have to be modified, as side effects could occur in all locations where the data is accessed. The IEC 1131-3 principle of object-oriented data is also violated by using global variables. To prevent this, call parameters could be used instead of the global variables in Example 7.10:

```
Main Program                          Using POU:

PROGRAM main                          ...
VAR                                   FUNCTION_BLOCK Work
  rec1: rec_struct := ...;             VAR_IN_OUT
  rec2: rec_struct := ...;               recipe: rec_struct;
  Work1: Work;                         END_VAR
END_VAR                                   ...
IF Condition THEN                         LD recipe.par1
  Work1 (rec1);                           ...
ELSE
  Work1 (rec2);
END_IF;
```
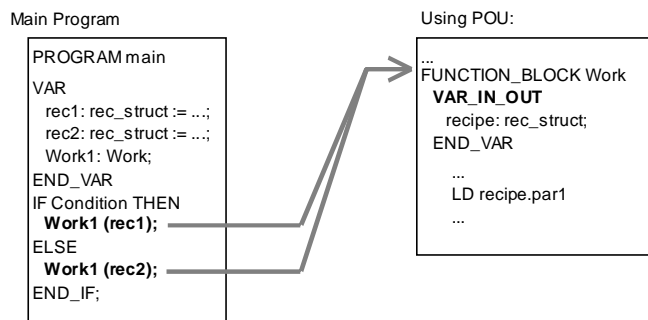
Example 7.11. To avoid the side effects that are possible when using the method in Example 7.10, data sets rec1 and rec2 are passed as input/ output parameters (this only passes a pointer and avoids copying huge data structures).

7.9 FB Interconnection

7.9.1 Data exchange and co-ordination of blocks in distributed systems

IEC 61131-3 defines PROGRAM blocks, which hierarchically call function blocks and functions, passing them parameters. Each PROGRAM (or certain function block instances) is assigned tasks. Communication between the tasks takes place using global variables or ACCESS variables.
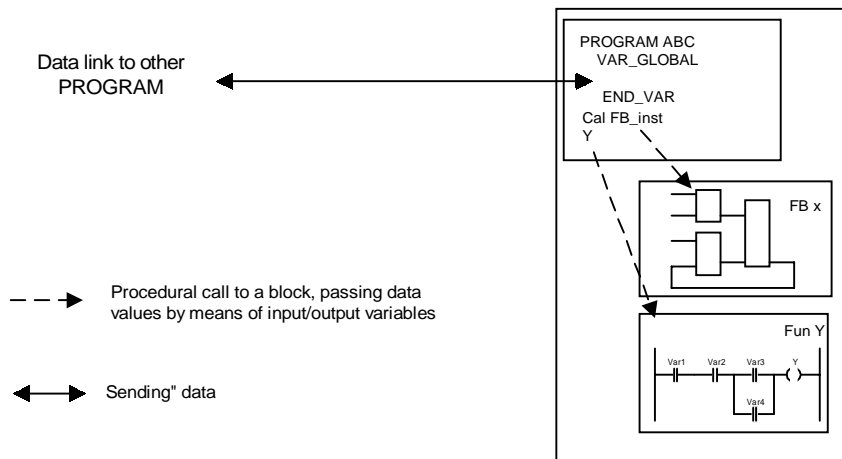
Figure 7.7. One program calling other blocks and passing them information through parameters. With other programs, only data is exchanged.

Distributed PLC systems, as used in industrial plants, power supply systems or building automation, require:

- parallel, autonomous execution of individual algorithms,
- geographically separate computing nodes,
- asynchronous data exchange.

Today, distributed applications are typically implemented as follows. Pre-fabricated blocks are copied from a library to create a new project. Missing functionality is implemented in new, specially written blocks. One PROGRAM, together with the function blocks and functions called, constitutes an executable unit. Several programs are written.

   Each PROGRAM is now assigned a node in the network, and the inputs and outputs of all programs are interconnected. Unconnected inputs of program instances are assigned individual parameter values where necessary. This is shown in Figure 7.8

   Libraries are usually implemented by experts from the hardware manufacturer or are part of the firmware (EPROM) of the PLC or network node.

This is an extreme case of the IEC 61131-3 programming model. The application is "configured" from pre-fabricated blocks. The programs run mostly autonomously without being "called" by other blocks. Functions and function blocks in the sense of IEC 61131-3 are provided locally to be called by the PROGRAM. A direct call to a block in another node or CPU is not possible (a PROGRAM may not invoke a PROGRAM).
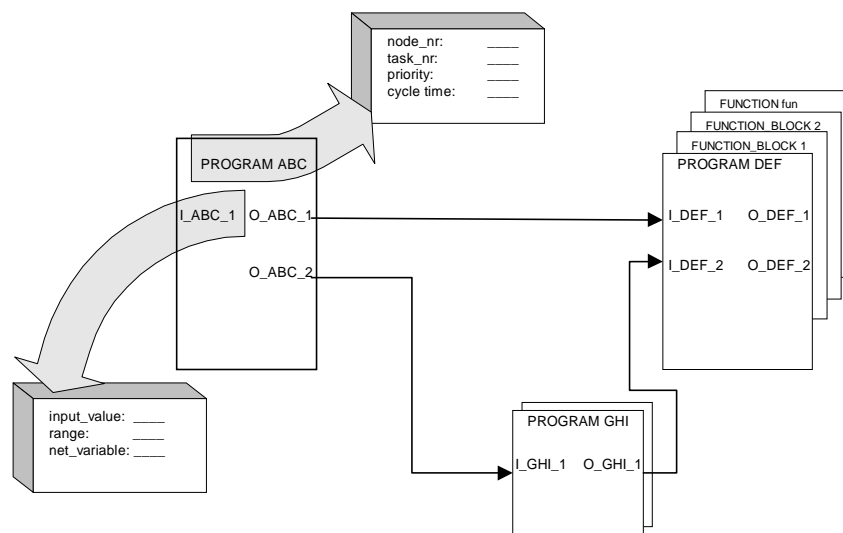
Figure 7.8. Assignment of blocks to network nodes and PLC tasks. All PROGRAMs run independently and are connected only via ACCESS variables.

It is possible to deviate from the definition in IEC 61131-3, which stipulates that only programs may be connected to objects in different tasks. Function blocks can then be distributed amongst computing nodes (tasks) and interconnected at will. This allows a much closer mapping of algorithms to network nodes. The per-

formance of a distributed automation system can be increased without changing the program by simply adding computing nodes and re-configuring appropriately.

This involves two problems, as shown in Figure 7.8:

1) The run-time behaviour of all blocks must be properly co-ordinated because control information sent together with other data information is confusing.
2) A mechanism is needed to ensure consistent flow of data between blocks (network nodes) by checking the validity of data items or groups of data items.

As long as all blocks interconnected are executed on the same network node, execution control can be implemented implicitly (execution control for blocks is defined in IEC 61131-3) or explicitly configured (execution number for every block). Additional control is needed if the tasks involved have no common time base, e.g. if they are executed on different nodes in the network.

This topic is currently being investigated by a working group of the IEC, see Chapter 9 and VDI [VDI 3696/1-93] [VDI 3696/3-93].

These interconnection techniques result in even greater separation between programming and configuration. The programmer writes programs using FBs and functions and interconnects these. Afterwards, the function blocks are assigned to computing nodes. One and the same application can execute locally in one task, or be distributed between many tasks, without having to modify the program structure.

7.9.2 Macro techniques in FB interconnection

A project with interconnections as described in Section 7.9.1 consists of a large number of blocks. To make the structure easier to understand, groups of blocks can be visually combined and shown as one block. Related functionality can be grouped onto separate working sheets, called *Function Charts*. This "*macro*" technique is explained below. Example 7.12 gives an example from the plant industry.
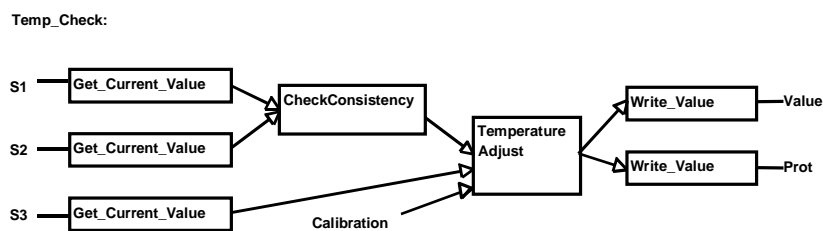
Example 7.12. Interconnection of simple basic elements in plant engineering

Placing and connection of blocks (Get_Current_Value, CheckConsistency, ...) is performed graphically by the user. All blocks have a clearly defined functionality. Data declarations are made implicitly by block instantiation. Only when connecting inputs and outputs does the user (or programming system) have to check for compatibility of data types.

Blocks provided by the manufacturer can be used to build more sophisticated blocks using a kind of macro technique. This corresponds to building complex data structures out of elementary data types. For this reason these blocks are sometimes called *Derived Function Blocks*.

Temp_Control:
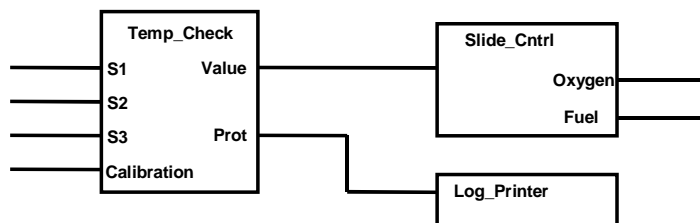
Example 7.13. The blocks from Example 7.12 (called Temp_Check) attain a higher degree of specialisation and sophistication when grouped together using macro techniques.

After definition as shown in Example 7.13, block Temp_Control can be used in applications.

If the blocks used are elementary blocks defined by the manufacturer (well-known behaviour), good simulation results can be achieved (e.g. run-time, effects of different hardware assignments or modifications in communication infrastructure).

7.10 Diagnostics, Error Detection and Error Handling

Diagnostics is basically the "detection of error conditions during operation, and localisation of the source of error". There are four areas where errors can occur:

1) PLC hardware, including connections to other devices,
2) PLC software (operating system),
3) User software,
4) Process behaviour: the process under control may enter an unforeseen state.

A general distinction is made between system errors and programming errors.

Vendors offer various tools for diagnostics. These can be either additional hardware that checks for error conditions and provides information about them, or software functions to be included in the application. SFC is a good language for detecting errors in a running installation (e.g. "Transition XY not firing") or for continuing with a defined response after detecting an error.

Error concept of IEC 61131-3.

IEC 61131-3 has only a very general approach to error handling, giving the user a certain amount of support in handling cases 2) and 3) above. The standard requires an error list to be provided by PLC manufacturers, indicating the system response to a variety of specified error conditions (see Appendix E):

1) The error is not reported. There must be a statement to this effect in the user documentation.
2) The possibility that the error might occur is detected when preparing (editing, compiling or loading) the program, and the user is warned accordingly.
3) The error is reported during execution of the program (at run time). Manufacturer-dependent procedures for handling the error are provided.

Quality assurance plays an important role in the automation business. The quality of today's compilers effectively prevents some typical program errors from slipping through at the compilation stage. Concepts of IEC 61131-3, like strict data type checking, even prevent some errors from occurring in the first place, during programming. However, some errors can only be detected at run time.

Some error situations, like division by zero (see Appendix E) should be checked by the PLC system. IEC 61131-3 [IEC TR3-94] recommends the definition of a uniform global (manufacturer-dependent) data structure for errors, which should contain the status of an operation (Error Yes/ No), the type of error (Division by zero) and the location of the error (POU name). This information could then be scanned by the application, or connected to the SINGLE input of a task (see Section 6.3.4). This task would be connected to a system routine or error routine.

In the event of an error, the PLC system would set the error status to TRUE and set other members of the data structure accordingly, thus starting the error task.

Extended error handling model (beyond IEC).

To improve software quality, it is desirable to provide the users themselves with a means of defining error conditions in a standardised form. A language construct like "asserted conditions" could be used for this. In this case, the programmer would implement the application with checks.

For example:

-   Is the value of a variable within the limits that apply at this program location?
-   Do two variables match?

FBD:
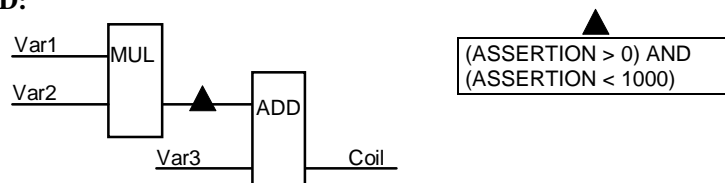
IL:

```
LD    Var1
MUL   Var2
ASSERTION((CR > 0) AND (CR < 1000))
ADD   Var3
ST    Coil
```

Example 7.14. For detection of run-time errors, it should be possible to implement checks that are calculated by the system itself. In the graphical languages, connections could be secured with assertion symbols. Expressions should be written in one of the languages of IEC 61131-3 (e.g. ST).

Such assertions can be simple expressions for the Current Result in IL, as shown in Example 7.14. Assertions can be used to check "critical" values at run time. Complex expressions can also be employed to compare input and output values, check for consistency or check important process parameters for logical relations.

Some systems provide automatic error checking facilities, e.g. for array indices, i.e. the index in an array must not be above the upper or below the lower limit of the array. This is supported by IEC 61131-3, see Chapter 3.

The response in the event of a violation of an assertion must be configurable, e.g.: Stop program; issue error message to visualisation system. For more sophisticated "exception handling", multiple error routines should be assignable to different assertions. These routines should have special privileges, such as the right to stop or restart the PLC system.

At present, assertion conditions and the associated error responses (without special privileges) still have to be written by users themselves, which is not always an ideal solution from the point of view of program readability.

The architecture of error handling is evident throughout a program, and is at present dependent on the manufacturer. The lack of standardised error detection and error handling routines makes porting of applications between different systems difficult, requiring specially trained system experts.

7.11 Hardware-Dependence

Studies have shown that even sophisticated cross-compilers can rarely automatically cross-compile more than 60% of a non-IEC 61131-3 PLC application to an IEC 61131-3 programming system. The reason is that the programs are heavily hardware-dependent. Custom routines are used to control special hardware, or specialised hardware addresses (status registers, system memory,...) are accessed.

IEC 61131-3 does not set out to eliminate the individuality of manufacturers. After all, a wide variety of software and hardware ensures high functionality. To ease portability, IEC 61131-3 provides the following mechanisms:

- All external information a program needs should be provided by IEC 61131-3-conformant global variables, access paths or communication FBs (see IEC 61131-5).
- Hardware I/O addresses used have to be declared in the PROGRAM or configuration scope.
- Hardware-dependent features have to be listed in a special table that manufacturers have to provide with their software.

The list of implementation-dependent parameters is given in Appendix F.

7.12 Readiness for New Functionality

Implementing the functionality of the new generation of fully graphical programming systems initially entails far greater overheads than that of existing systems. Close co-operation between hardware and software manufacturers is therefore essential to keep costs down. PLCopen, an independent group of PLC software and hardware manufacturers, represents one step in this direction.

   The European Community has conducted several projects to increase co-operation within the industry.

The aim is to:

- enable users to re-use applications on different platforms, thereby saving on development and testing,
- standardise interfaces (like OLE or OPC) to allow common development of tools (like logic analysers or simulation).

7.12.1 Exchange of programs and data

If a user is using different PLC systems, it is often a problem to replace program blocks during development (static program exchange) or exchange data between the different systems during operation (dynamic data exchange). This is shown in Figure 7.9.

To exchange data, communication function blocks are used during execution of a PLC program to send data information to another resource or task. As this resource might have a different processor or operating system, data must be exchanged in a general format, independent of individual PLC systems, which can be understood by all systems involved.

IEC 61131-5 defines the interfaces of standard function blocks for communication, but the structure and contents (semantics) of the information transferred is left open.

Program exchange is performed by copying a POU from one project to another, to re-use its functionality without re-writing it. Most programming systems support this. At present, this can often only be done one POU at a time (by copying the POU into the project). PLCopen has defined a data format which can carry additional information like author, verification information, etc. The next step in this process will be the definition of a source library to keep sources.
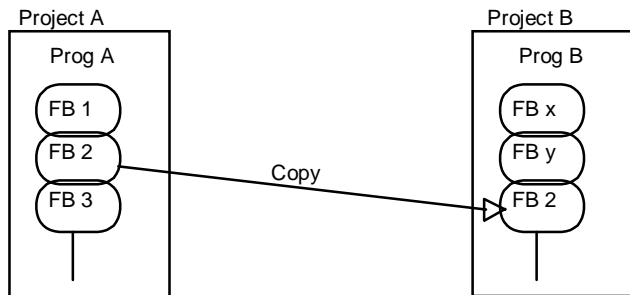
a) Programming system supports copying of an FB
(static program exchange)
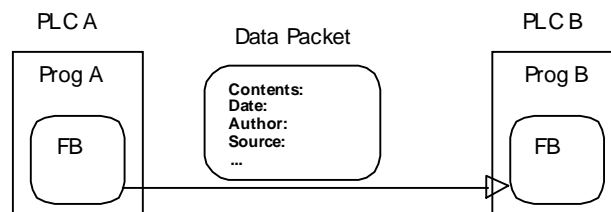
b) PLC A sends data to PLC B (dynamic data exchange)

Figure 7.9. Static program exchange (a) and dynamic data exchange (b) between two different projects and PLC families.

7.12.2 Extension with additional software packages

Some of today's IEC 61131-3 programming systems use the same operating system, but they are implemented completely differently and it is hard to imagine being able to use components of one programming system in another. Most systems even lack the modularity required for this purpose, although today's operating systems provide the necessary support.

A possible modular structure of a programming system is shown in Figure 7.10.
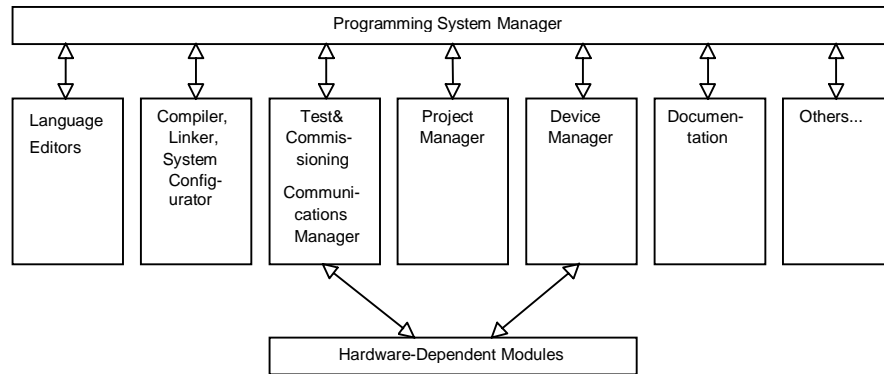
Figure 7.10. Modular structure of a PLC programming system is a requirement for future extensions to the system

At present, no standard exists for the architecture, nor for the method of communication (e.g. using files, OLE, OPC, class libraries, etc.).

Possible additions to the components shown above include:

- Plant design tools,
- Simulation tools,
- General-purpose data management systems,
- Specialised editors for parameters,
- Logic analysers,
- Plant diagnostics,
- Interfaces to PDA systems, SCADA systems, logging devices,
- Interfaces to CAD systems,
- Network administration.

8 Main Advantages of IEC 61131-3

Chapter 1 outlines goals and benefits of IEC 61131-3 for manufacturers and users. How well does this programming standard live up to expectations?

Many features and concepts of this way of programming PLCs have been described and explained in previous chapters. The core concepts are summarised again here.

The following outstanding features of PLC programming with IEC 61131-3 deserve special notice:

- Convenience and security with variables and data types,
- Blocks with extended capabilities,
- PLC configuration with run-time behaviour,
- Uniform programming languages,
- Structured PLC programs,
- Trend towards open programming systems.

8.1 Convenience and Security with Variables and Data Types

Local and global variables instead of hardware addresses
Formerly, all data memory of a PLC was accessed using global addresses, and the programmer had to take care that one part of a program did not overwrite the data of another part. This applied particularly to I/O addresses, flags and data blocks.

IEC 61131-3 replaces all global hardware addresses by named variables with a defined scope: the programming system *automatically* distinguishes between **global** variables and variables **local** to a POU. Global addresses can be accessed by assigning the address to a named variable in the declaration part and using this variable in the program.

Type-oriented access to PLC data

PLC programmers used to have to be careful to use the same data type when reading or writing to individual PLC addresses. It was possible to interpret the same memory location as an integer at one place in a program, and as a floating-point number at another.

IEC 61131-3 prevents such programming errors from occurring, as each variable (including direct hardware addresses) must be assigned a data type. The programming system can then check that all accesses use the proper data type.

Defined initial values for user data

All data is explicitly declared in the form of a variable, and assigned a data type in the declaration. Each data type has, either by default or as specified by the user, a defined initial value, so each and every variable used in a program is always correctly initialised in accordance with its properties.

Variables can be declared to be retentive (with a RETAIN qualifier). They are then automatically assigned to a battery-backed area of memory by the programming system.

Arrays and data structures for every application

Building on the predefined data types, the PLC programmer can design arrays and other complex data structures to match the application, as is the practice with high-level languages.

Limits of array indices and ranges of variable values are checked by the programming system as well as by the PLC system at run time.

Unified declaration of variables

The extensive facilities for using variables are generally identical in all the languages defined by IEC 61131.

8.2 Blocks with Extended Capabilities

Reuse of blocks

Blocks (POUs), such as functions and function blocks, can be designed to be independent of the target system used. This makes it possible to have libraries of reusable blocks, available for multiple platforms.

Parameters of a function block, input as well as output, and local data of each function block instance, keep their values between calls. Each instance of a function block has its own data area in memory, where it can perform its calculations independently of external data. It is not necessary to call a data block for the FB to work on.

Programs can also be used in several instances and be assigned to different tasks of one CPU.

Efficient assignment of block parameters

The standard provides a variety of mechanisms for passing data to and from blocks:

- VAR_INPUT: Value of a variable
- VAR_IN_OUT: Pointer to a variable
- VAR_OUTPUT: Return value
- VAR_EXTERNAL: Global variable of another POU
- VAR_ACCESS: Access path within a configuration.

Until now, the only items in this list that have been provided by most PLC systems have been global variables and the capability for passing values to a called block (but not for returning a value).

Standardised PLC functionality

To standardise typical PLC functionality, IEC 61131-3 defines a set of standard functions and function blocks. The calling interface, the graphical layout and the run-time behaviour of these is strictly defined by the standard.

This standard "library" for PLC systems is an important foundation for uniform and manufacturer-independent training, programming and documentation.

8.3 PLC Configuration with Run-Time Behaviour

Configurations structure PLC projects

Tasks and programs are assigned to the controller hardware at the highest level of a PLC project (the configuration). This is where the run-time properties, interfaces to the outside, PLC addresses and I/Os are defined for the various program parts.

Run-time features for PLC programs

Until now, the methods of specifying run-time properties, like cycle time and priority of programs, have often been system-specific. With IEC 61131-3, such parameters can be specified and documented individually by defining tasks.

PLC programs must not be recursive. The amount of memory required to hold the program at run time can therefore be determined off-line, and the programs are protected from unintentional recursion.

8.4 Uniform Programming Languages

IEC 61131-3 defines five programming languages, which can cover a wide range of applications.

As a result of this international standard, PLC specialists will in future receive more uniform training, and will "speak the same language" wherever they are employed.

The cost of training will be reduced, as only specific features of each new controller system have to be learned.

Documentation will be more uniform, even if hardware from more than one vendor is being used.

8.5 Structured PLC Programs

The various language elements of IEC 61131-3 allow clear structuring of applications, from definitions of blocks and data up to the hardware configuration.

This supports structured programming (top-down and bottom-up) and facilitates service and maintenance of applications.

The "structuring language", Sequential Function Chart, also enables users to formulate complex automation tasks clearly and in an application-oriented way.

8.6 Trend towards Open PLC Programming Systems

Standardisation of programming languages leads to standardisation of software, which makes vendor-independent, portable programs feasible, as is, for example, already the case in the personal computer domain with programming languages like assembler, COBOL and, most notably, C. The "feature tables" of IEC 61131-3 provide a basis for comparing programming systems with the same basic functionality from different vendors. There will still be differences between the systems of different manufacturers, but these will mainly be found in additional tools like logic analysers or off-line simulation, rather than in the programming languages themselves.

The common look and feel in PLC programming will become international and bring the separate markets in Europe, the US or Asia closer together.
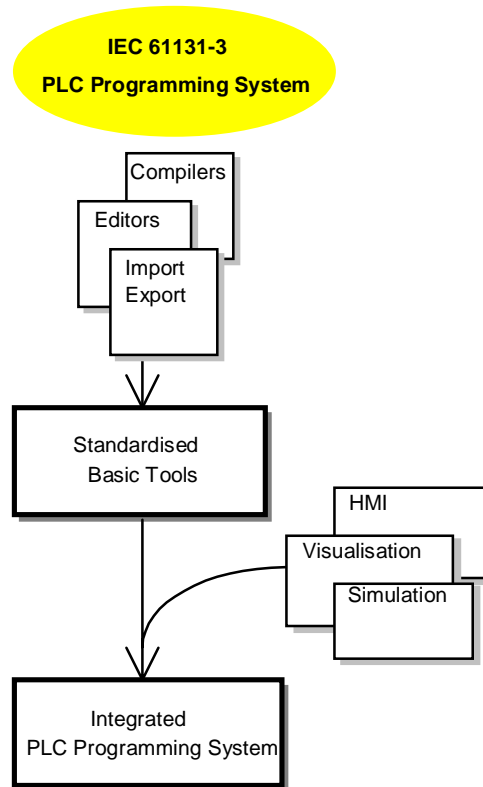
Figure 8.1. Trend towards open, standardised components built on IEC 61131-3-compliant programming systems

The new generation of PLC programming systems will have a standardised basic functionality, plus highly sophisticated additional tools to cover a wide range of applications.

As shown in Figure 8.1, standardisation by IEC 61131-3 promotes integrated systems, built from standardised components like editors, compilers, export and import-utilities, with open, re-usable interfaces.

Tools which are traditionally sold as separate packages, like HMI, simulation or visualisation, will cause de facto interface standards to be established for these components.

8.7 Conclusion

IEC 61131-3 will cause PLC manufacturers and users to give up old habits for a new, state-of-the-art programming technology. A comprehensive standard like IEC 61131 was necessary to achieve a uniform environment for innovating the configuration and programming of PLC systems.

This manufacturer-independent standard will reduce the training and familiarisation time for PLC programmers, programs written will be more reliable, and the functionality of PLC systems will catch up with the powerful software development environments available for PCs today.

Compliance with IEC 61131-3, migration paths from legacy systems towards the new architectures, and a powerful, ergonomic user interface will be the most important criteria for users for a wide acceptance of the new generation of PLC programming systems.

Today's complex requirements and economic constraints will lead to flexible, open, and therefore manufacturer-independent PLC programming systems.

9 Programming by Configuring with IEC 61499

Programming using graphical elements taken from the "real world" of the application to be programmed is becoming more and more important.

With the graphical languages LD, FBD or SFC of IEC 61131-3 previously discussed, data flow and logical execution sequence can be programmed and documented using symbols and names. However, it is also desirable to be able to display the topological distribution of programs, their general configuration and interconnections to other parts of a distributed automation project in a graphical manner. This takes place at a higher, more abstract level than the programming of POUs described so far.

The tools for configuring complex and distributed applications are called *configuration editors*. Program parts, such as function blocks, are combined to form larger units. This is done by *interconnection of function blocks*.

In order to standardise unified language elements for this purpose, work is currently being carried out on international standard IEC 61499, as a supplement to the existing IEC 61131. This chapter gives a brief summary of the basic concepts and ideas of this additional standard and explains its relationship with IEC 61131. The subject of this new standard would need to be discussed in greater detail in another book.

9.1 Programming by FB Interconnection with IEC 61131-3

In order to clarify the differences between IEC 61499 and IEC 61131-3, we shall first look at some special features of distributed programming.

The programming languages described in Chapter 4 are used to define algorithms for blocks. Function blocks and functions call each other, exchange information by means of their parameters and form a program in conjunction with a POU of type PROGRAM. A program runs as a task on a resource (CPU of a PLC). IEC 61131-3 essentially concentrates on describing single programs together with

their execution conditions. Information exchange between programs takes place using ACCESS variables or global data areas. This topic is discussed in Section 7.9 and illustrated by Figure 7.8.

Complex, distributed automation tasks have an extensive communication and execution structure. Intensive data exchange takes place between geographically separate control units. The semantic and temporal dependencies and conditions have to be specified.

To do this, programs are assigned to tasks of network nodes, execution conditions are defined as described in Section 6.1, and the inputs and outputs of programs (such as network addresses or parameter values) are interconnected.

Creating distributed automation solutions, i.e. configuring function blocks for physically different and geographically separate hardware and synchronising their execution, is the subject of future standard IEC 61499.

9.2 IEC 61499 – The Programming Standard for Distributed PLC Systems

The sequential invocation of blocks defined in IEC 61131-3 is not a suitable method for program structuring in distributed systems. This is already apparent in Figure 7.8. The goal of a distributed, decentralised system is to distribute programs between several control units and to execute them in parallel (in contrast to sequential execution with invocation by CAL). Here it is essential to ensure data consistency between nodes of the networked system, i.e. to define exact times for mutual data exchange.

Two kinds of information exchange play an essential part in IEC 61499:

1) Data flow of user data,
2) Control flow, which controls the validity of user data as event information.

The interaction of data and control flow could also be programmed by means of IEC 61131-3 using global variables and access paths. But the resulting overall program can easily become hard to read and slower to execute.

In order to describe the interactions between program parts and elements of control hardware within a distributed, networked automation system easily and exactly, IEC 61499 uses a model ("top-down" approach) with several hierarchical levels:

- System   - Application
- Device   - Function block
- Resource

The definitions of the terms Resource and Function Block are, however, wider than those of IEC 61131-3, as will be explained in this chapter. The IEC standard committees will have to co-ordinate the terms in both standards.

Instead of assigning PROGRAM and TASK to a resource, function blocks in IEC 61499 can be assigned run-time properties *directly* via the resource .

9.2.1 System model

In a real automation environment several control units, referred to here as *devices*, execute the same or different applications in parallel. This is outlined in Figure 9.1.
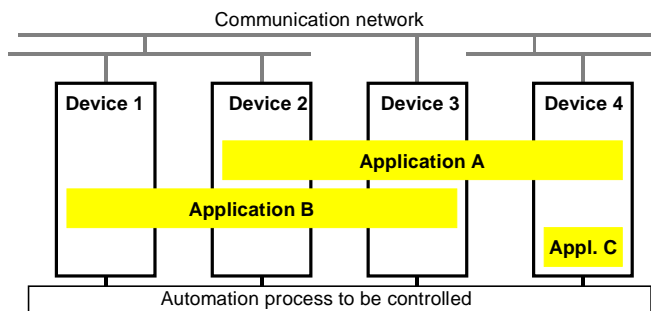
Figure 9.1. Control of a real process can be distributed between several devices. As in IEC 61131-3, several programs can also be configured for one device. The program parts interchange information via communication networks.

9.2.2 Device model

Closer examination of a device, as in Figure 9.2, shows that it consists of:

-    its application programs,
-    an interface to the communication network,
-    an interface to the automation process,
-    the device hardware, on which the resources run.

A resource represents an independent executable unit with parameters (a task in the general sense). Several resources can run on each device, and they can perform the same or different applications.

IEC 61499 uses two views of a distributed program, which are explained in this chapter. On the one hand, this standard looks at the hierarchy of System—Device—Resource, in order to describe system structure and the corresponding run-time properties. On the other hand, it also defines the user-oriented view of a distributed program. This user view is summarised by application and function block models that are discussed later in this chapter.
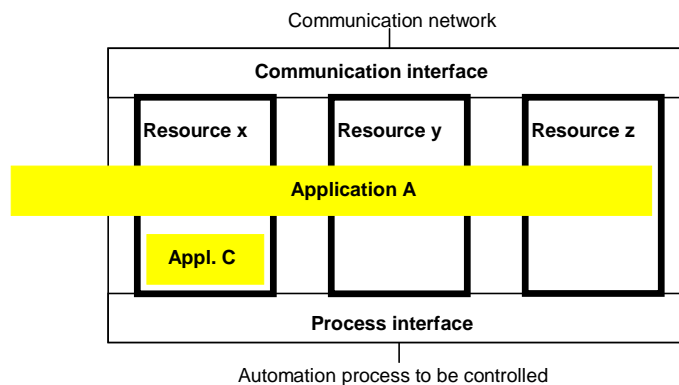
Figure 9.2. A device can contain several resources, which use common interfaces to exchange information with other control units and the automation process.

9.2.3 Resource model

A resource consists of function blocks, which exchange event as well as data information using special interfaces. There are two kinds of function blocks:

1) *Service interface function blocks*, which are standard FBs and form the interfaces to the automation process and the communication network.
2) User-defined function blocks, which make up the actual application program (algorithm).

As in IEC 61131-3, there is a distinction between FB type and FB instance.

   Run-time properties, such as the maximum number of instances, execution time, number of connections etc., can be assigned to each function block within the resource.
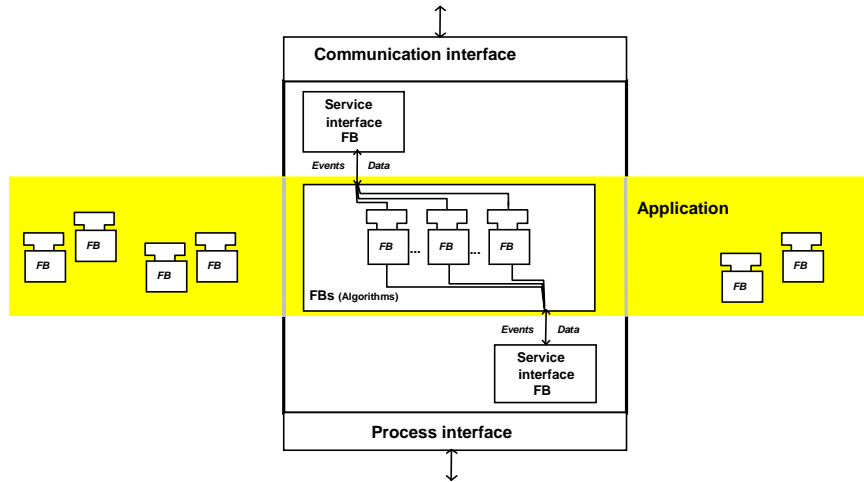
Figure 9.3. A resource consists of function blocks for controlling and data processing (algorithms) together with interface blocks (communication/ process).

The interconnection of FBs by the user is not carried out at resource level, but at application level (see next section). The real information exchange between the FBs of the application program takes place "invisibly" for the user via the communication and process interfaces.

An application can be implemented on one or more resources.

9.2.4 Application model

This section deals with the user-oriented view of a program. This view corresponds to the horizontal, grey "Application" bar in Figure 9.3, which can extend over several devices or resources.

The application level forms the real programming level because it is here that the FBs are interconnected with one another, independently of the resources on which they run. It describes the application – which may subsequently be distributed amongst several resources.

After the application program, consisting of several FBs, has been assigned to the resources and the program has been started, communication takes place transparently via the service interfaces with the connections specified by the user.

Figure 9.4 shows how an application is made up of both controlling parts (with events) and data processing parts (algorithms). Here the different graphical representation to that of IEC 61131-3 can be seen.
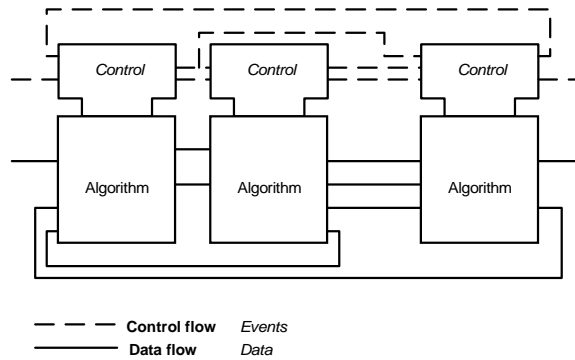
Figure 9.4. The application consists of interconnected function blocks; each of which has both controlling (control flow) and data processing (data flow) functions.

Control and data information always flows into a function block from the left and is passed on after processing from the outputs on the right.

9.2.5 Function block model

The function blocks are the smallest program units (like POUs). Unlike the FBs of IEC 61131-3, a function block in IEC 61499 generally consists of two parts:

1) Execution control: Creation and processing of events with control inputs and outputs (control flow),
2) Algorithm with data inputs and outputs and internal data (data flow and processing).

These function blocks can be specified in textual or graphical form. Function blocks are instantiated for programming, as in IEC 61131-3. The language elements for FB interface description are therefore very similar, see also Chapter 2.

Figure 9.5 shows the graphical representation of a function block in accordance with IEC 61499.

The algorithm part is programmed in IEC 61131-3 (like a POU body).

The execution control part is programmed using a state diagram or sequential function chart (SFC in IEC 61131-3). The events are input values for *state diagrams*, or *execution control charts (ECC)* . These ECCs control the execution times of the algorithm or parts of it depending on the actual state and incoming events.
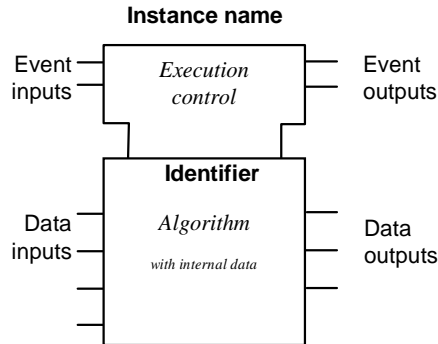
Figure 9.5. Graphical representation of a function block. Details of execution control, the internal algorithm and internal data are not shown at this level.
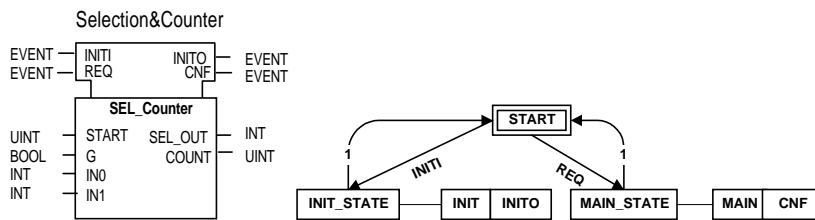
Example 9.1. Function block with typed formal parameters and state diagram (ECC)
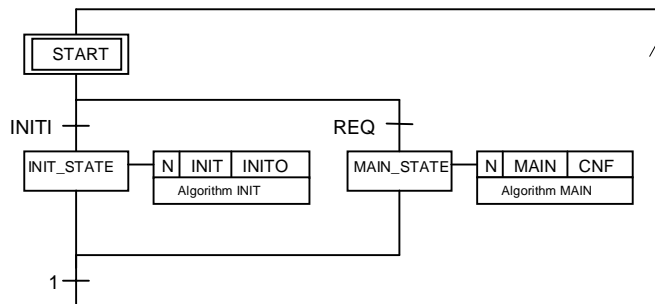
Example 9.2. Execution control of Example 9.1 using Sequential Function Chart (SFC) as defined in IEC 61131-3. The output events CNF and INITO are set by the application program and controlled by calling standard function blocks.

Example 9.1 contains function block Sel_Counter (instance name Selection&Counter), which consists of an ECC control part and the algorithm part, which itself consists of the two algorithms INIT and MAIN. The execution control determines which algorithm part will be active at what time.

In Example 9.1, when the event INITI occurs, the FB control will change from initial state START to state INIT_STATE and algorithm INIT will be executed. Afterwards event output variable INITO is set (action "N"), followed by a RESET (i.e. a signal pulse). Now execution control evaluates the next transition. This has the constant parameter "1" in this example, which means the condition is always true, and leads back to state START. Incoming event REQ is processed analogously.

This behaviour is equivalent to the actions of Sequential Function Chart (SFC) in IEC 61131-3 and is illustrated by Example 9.2. IEC 61499 assumes that it is more favourable to specify the execution control using state diagrams, as with this method only **one** state can be active at a time. In SFC this can be achieved by prohibiting simultaneous branches.

Example 9.3 shows the textual definition of the FB type in Example 9.1.

The keyword WITH connects an event input/ output with a data input/ output. If an event parameter is set, it indicates the validity of the corresponding data line (assigned by WITH).

Composite function blocks
For the purposes of clear, object-oriented representation, several *basic function blocks* can be combined to form a new *composite function block*, which looks just like a "normal" function block on the outside, as shown in Figure 9.6.

Composite function blocks do not have their own execution control part, as this is the sum of the controls of all the basic FBs of which it is composed. In the graphical representation in Figure 9.6 a) the FB header is therefore "empty".

```
FUNCTION_BLOCK Sel_Counter
  EVENT_INPUT
        INITI  WITH START;                    ALGORITHM INIT:
        REQ  WITH G, IN0, IN1;                  INTERNAL_COUNT := START;
  END_EVENT                                     COUNT := INTERNAL_COUNT;
                                              END_ALGORITHM
  EVENT_OUTPUT
        INITO  WITH COUNT;                    ALGORITHM MAIN:
        CNF  WITH SEL_OUT, COUNT;              IF G = 0 THEN SEL_OUT := IN0;
  END_EVENT                                     ELSE
                                                    SEL_OUT := IN1;
  VAR_INPUT                                     END_IF;
        START:          UINT;                   INTERNAL_COUNT :=
        G:              BOOL;                     INTERNAL_COUNT +1;
        IN0, IN1:       INT;                     COUNT := INTERNAL_COUNT;
  END_VAR                                      END_ALGORITHM

  VAR_OUTPUT
        SEL_OUT:        INT;
        COUNT:          UINT;
  END_VAR

  VAR
        INTERNAL_COUNT: UINT;
  END_VAR
  ...
  END_FUNCTION_BLOCK
```

Example. 9.3. Example 9.1 in textual representation (Structured Text ST of IEC 61131-3).
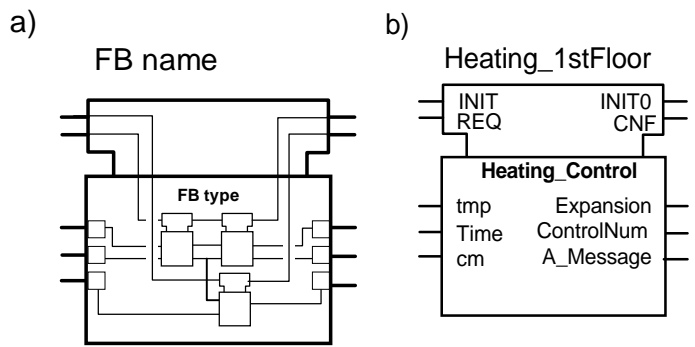
Figure 9.6. A composite function block consists of several interconnected function blocks with a common interface.
a) Example: Internal structure of a composite FB,
b) Example: External appearance of this FB.

9.2.6 Creating an application

Future IEC 61499 application programmers will write programs by configuring and assigning parameters to ready-made function blocks.

FB	Explanation
Standard FBs	• FBs with functionality as in IEC 61131-3 • Service interface FBs (standardised communication services) • Event FBs (standardised event generation and processing)
User-defined FBs	Algorithms and ECC execution control e.g. programmed with IEC 61131-3

Table 9.1. Different types of function blocks in IEC 61499

The function blocks in Table 9.1 can be implemented as basic or composite FBs.
   For example, Event FBs provide functions for merging and splitting events or creation of single or cyclic events.

A configuration editor is used for allocating blocks to resources (devices) and for interconnecting FBs.

9.3 Overview of the Parts of IEC 61499

The future standard IEC 61499 will consist of two parts, whose main contents are summarised in Table 9.2 ([IEC 61499-97]):

Parts	Contents
1. *Architecture*	Introduction and modelling, describes the validity, defines common terms, specification of function blocks, service interfaces, configuration and syntax.
2. *Model development cycle*	Contains descriptions supporting the life cycle of distributed programs. This is still in the conceptual phase (in 12/2000).

Table 9.2. Structure and contents of future standard IEC 61499

10 Contents of CD-ROM

10.1 IEC Programming Systems STEP 7 and OpenPCS

The CD-ROM enclosed in this book contains the following information, examples and programs:

1) **STEP 7 Demo Software** as a demo version for PLC programming with IEC 61131-3 using the languages: STL, LAD, FBD, S7-GRAPH, S7-SCL, CFC and S7-HiGraph[1]; running under Windows 95/98 and Windows NT
2) **Open PCS** as a demo version for PLC programming with IEC 61131-3 using the languages: IL, Ladder, FBD, ST, SFC, a US-conformant Ladder editor (including EN/ENO), as well as Smart PLC; running under Windows 3.x, Windows 95/98 and Windows NT
3) **IL examples** of this book
4) **Buyer's Guide** for IEC 61131-3-compliant programming systems.

File README.TXT on the CD contains important information about the installation and use of the files and programs. It shows how to copy the files onto hard disk and gives tips on how to use the examples and the buyer's guide.

README.TXT is an ASCII file and can be read using any editor (e.g. in DOS and Windows).

The files of the two programming systems are either self-extracting or in the form of an installation package, i.e. they cannot be read immediately but must first be decompressed or installed. No additional software is needed.

Demo versions of STEP 7 (Siemens) and OpenPCS (infoteam).
With the aid of the demo versions of two selected programming systems, readers can program, modify, extend and test all the examples in this book or create programs of their own in order to practise PLC programming with IEC 61131-3.

STEP 7 uses CFC and S7-HiGraph as tools for interconnecting FBs and for programming with state diagrams. OpenPCS contains a run-time package Smart PLC, which additionally allows execution of a PLC program on PC (offline

---

[1] STL corresponds to IL, LAD to LD, S7-GRAPH to SFC, and S7-SCL to ST

simulation). A CFC editor is included, as well as a Ladder editor, which combines the advantages of IEC 61131-3 programming with those of customary American-style Ladder programming.

Hints on using and purchasing full software versions of both programming systems (as well as hardware) can also be found in the relevant folders on the CD.

The authors are not responsible for the contents and correct functioning of these demo versions. These software packages have only a restricted scope compared with the full functionality of the corresponding products. Their use is only allowed in conjunction with this book for learning and training purposes.

IL examples

To save the reader having to re-type the programming examples in this book, the most important IL examples are provided on the CD. Further information about these can also be found in README.TXT.

10.2 Buyer's Guide for IEC 61131-3 PLC Programming Systems

The CD-ROM also contains a buyer's guide as a file in the format "Word for Windows Version 6.0 and higher".
Contents of the buyer's guide (file **BuyGuide.doc**):

    Buyer's Guide for IEC 61131-3 PLC Programming Systems
        Checklists for evaluation of PLC programming systems
        Using the checklists
        Checklists for PLC programming systems
            Compliance with IEC 61131-3
            Language scope, decompilation and cross-compilation
            Tools
            Working environment, openness, documentation
            General, costs

This buyer's guide essentially consists of tables, or "checklists", which permit objective evaluation of PLC programming systems compliant with the standard IEC 61131-3. The use of these lists is described in detail before explaining each criterion for PLC programming systems.

The file can be copied for multiple product evaluation and individual editing of the checklists. These tables are stored on the CD in four different file formats:

1) Microsoft Word for Windows, Version 6.0 and higher (file TABLES.DOC),
2) Microsoft Excel for Windows, Version 4.0 and higher (file TABLES.XLS),
3) ANSI text (file TABLES.ANS),
4) ASCII text (file TABLES.ASC).

The Excel version is advantageous, as all calculations can be done automatically.

A Standard Functions

This appendix contains a complete overview of all the standard PLC functions described by means of examples in Chapter 2. For every standard function of IEC 61131-3, the following information is given:

- Graphical declaration
- (Semantic) description
- Specification of some functions in Structured Text (ST).

Standard functions have input variables (formal parameters) as well as a function value (the returned value of the function). Some input variables are not named. In order to describe their functional behaviour the following conventions apply:

- An individual input variable without name is designated as "IN"
- Several input variables without names are numbered "IN1, IN2, ..., INn"
- The function value is designated as "F".

General data types (such as ANY or ANY_BIT) are used in the description. Their meaning is explained in Section 3.4.3 and they are summarised in Table 3.9. The designator ANY here stands for one of the data types: ANY_BIT, ANY_NUM, STRING, ANY_DATE or TIME.

   Many standard functions have a textual name as well as an alternative representation with a symbol (e.g. ADD and "+"). In the figures and tables, both versions are given.

A.1 Type Conversion Functions

*	Input data type, e.g. REAL
**	Output data type, e.g. INT
*_TO_**	Function name, e.g. REAL_TO_INT
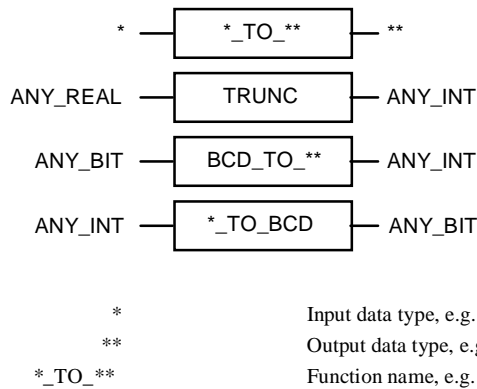
Figure A.1. Graphical declarations of the type conversion functions

These standard functions convert the input variable into the data type of their function value (type conversion).

Name	Function / Description
*_TO_**	When REAL values are converted to INT values, they are rounded up or down to the next whole number. Halves, e.g. 0.5 or 0.05, are rounded up.
TRUNC	This function cuts off the places of a REAL value after the decimal point to form an integer value.
BCD	The input and/or output values of type ANY_BIT represent BCD-coded bit strings for the data types BYTE, WORD, DWORD and LWORD. BCD coding is not defined by IEC 61131-3, it is implementation-dependent.

Table A.1. Description of the type conversion functions

A.2 Numerical Functions

```
ANY_NUM ─────┌──────────┐───── ANY_NUM
             │   ABS    │
             └──────────┘

ANY_REAL ────┌──────────┐───── ANY_REAL
             │   ***    │
             └──────────┘
```
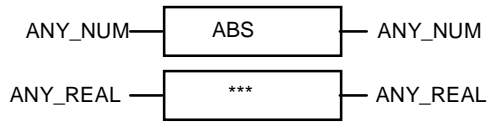
    ***   stands for: SQRT, LN, LOG, EXP,
                          SIN, COS, TAN,
                          ASIN, ACOS, ATAN

Figure A.2. Graphical declarations of the numerical functions

Name	Function	Description
ABS	Absolute value	$F := |\,IN\,|$
SQRT	Square root	$F := \sqrt{IN}$
LN	Natural logarithm	$F := \log_e (IN)$
LOG	Logarithm base 10	$F := \log_{10} (IN)$
EXP	Exponent base e	$F := e^{IN}$
SIN	Sine, IN in radians	$F := SIN\ (IN)$
COS	Cosine, IN in radians	$F := COS\ (IN)$
TAN	Tangent, IN in radians	$F := TAN\ (IN)$
ASIN	Principal arc sine	$F := ARCSIN\ (IN)$
ACOS	Principal arc cosine	$F := ARCCOS\ (IN)$
ATAN	Principal arc tangent	$F := ARCTAN\ (IN)$

Table A.2. Description of the numerical functions

A.3 Arithmetic Functions

***   stands for: ADD, +   ***   stands for: SUB, -
                         MULT, *   DIV, /
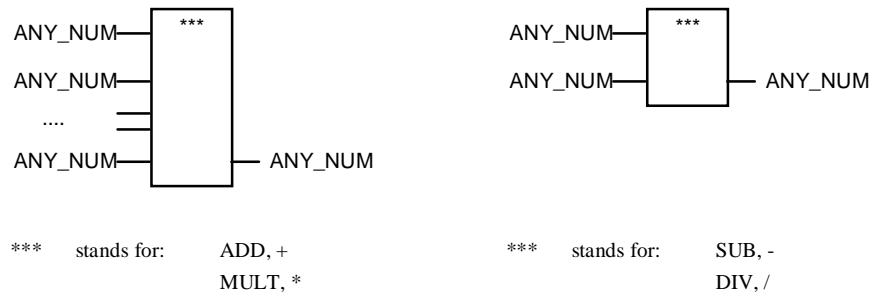
Figure A.3. Graphical declarations of the arithmetic functions ADD, MUL, SUB and DIV

*** stands for: EXPT, **

*** stands for: MOVE, :=
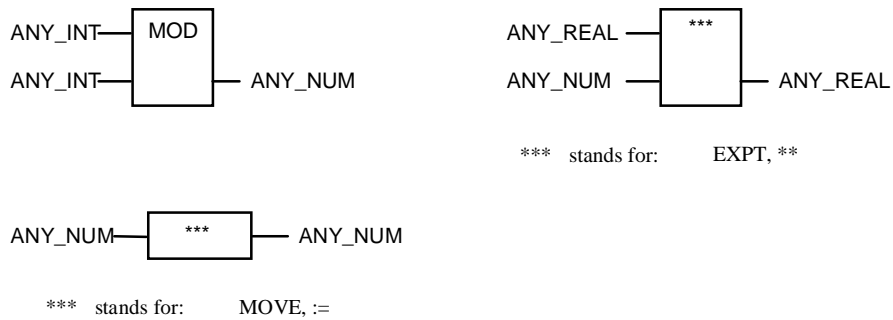
Figure A.4. Graphical declarations of the standard arithmetic functions MOD, EXPT and MOVE

Name	Symbol	Function	Description
ADD	+	Addition	F := IN1 + IN2 + ... + INn
MUL	*	Multiplication	F := IN1 * IN2 * ... * INn
SUB	-	Subtraction	F := IN1 - IN2
DIV	/	Division	F := IN1 / IN2
MOD		Remainder formation	F := IN1 - (IN1/ IN2)*IN2
EXPT	**	Exponentiation	$F := IN1^{IN2}$
MOVE	:=	Assignment	F := IN

Table A.3. Description of the arithmetic functions

In the case of the division of integers, the result must also be an integer. If necessary, the result is truncated in the direction of zero.

If the input parameter IN2 is zero, an error is reported at run time with error cause "division by zero", see also Appendix E.

A.4 Bit-Shift Functions

```
              ┌───────┐
              │  ***  │
   ANY_BIT ───┤IN     │
              │       │
      UINT ───┤N      ├─── ANY_BIT
              └───────┘
```
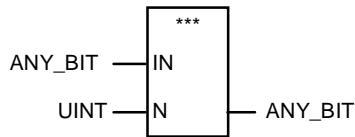
   ***   stands for: SHL, SHR, ROL, ROR

Figure A.5. Graphical declarations of the bit-shift functions SHL, SHR, ROR and ROL

Name	Function	Description
SHL	Shift to the left	Shift IN to the left by N bits, fill with zeros from the right
SHR	Shift to the right	Shift IN to the right by N bits, fill with zeros from the left
ROR	Rotate to the right	Shift IN to the right by N bits, in a circle
ROL	Rotate to the left	Shift IN to the left by N bits, in a circle

Table A.4. Description of the bit-shift functions

A.5 Bitwise Boolean Functions

*** stands for: AND, &, OR, >=1, XOR, =2k+1
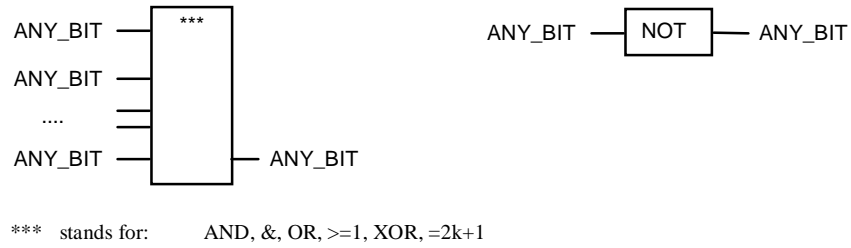
Figure A.6. Graphical declarations of the bitwise Boolean functions AND, OR, XOR and NOT

Name	Symbol	Function	Description
AND	&	Bit-by-bit AND	F := IN1 & IN2 & ... & INn
OR	>=1	Bit-by-bit OR	F := IN1 v IN2 v ... v INn
XOR	=2k+1	Bit-by-bit XOR	F := IN1 XOR IN2 XOR ... XOR INn
NOT		Negation	F := ¬ IN

Table A.5. Description of the bitwise Boolean functions

The logic operation is performed on the input parameters bit-by-bit. That is, every bit position of one input is gated with the corresponding bit position of the other input and the result is stored in the same bit position of the function value.

An inversion can also be represented graphically by a circle "o" at the Boolean input or output of a function.

A.6 Selection Functions for Max., Min. and Limit

*** stands for: MIN, MAX

Figure A.7. Graphical declarations of the selection functions MAX, MIN and LIMIT

Name	Function	Description
MAX	Maximum formation	F := MAX (IN1, IN2, ... , INn)
MIN	Minimum formation	F := MIN (IN1, IN2, ... , INn)
LIMIT	Limit	F := MIN (MAX (IN, MN), MX)

Table A.6. Description of the selection functions MAX, MIN and LIMIT

These three standard functions are specified by declarations in ST in Example A.1 and Example A.2.

```
FUNCTION        MAX : ANY          (* maximum formation; ANY stands for INT, ... *)
  VAR_INPUT    IN1, IN2, ... INn  :  ANY;      END_VAR
  VAR          Elem               :  ANY;      END_VAR
  IF IN1 > IN2 THEN        (* first comparison *)
      Elem    := IN1;
  ELSE
      Elem    := IN2;
  END_IF;
  IF IN3 > Elem THEN       (* next comparison *)
      Elem    := IN3;
  END_IF;
  ...
  IF INn > Elem THEN       (* last comparison *)
      Elem    := INn;
  END_IF;
  MAX     :=  Elem;        (* writing the function value *)
END_FUNCTION
```

Example A.1. Specification of the selection function MAX in ST; for MIN replace all „>" with „<".

The specification of the MIN function can be obtained by replacing all occurrences of ">" by "<" in the MAX specification in Example A.1.

```
FUNCTION        LIMIT : ANY         (* limit formation *)
  VAR_INPUT
    MN   :   ANY;
    IN   :   ANY;
    MX   :   ANY;
  END_VAR
  MAX    :=   MIN ( MAX ( IN, MN), MX);        (* call of MIN of MAX *)
END_FUNCTION
```

Example A.2. Specification of the selection function LIMIT in ST

A.7 Selection Functions for Binary Selection and Multiplexers
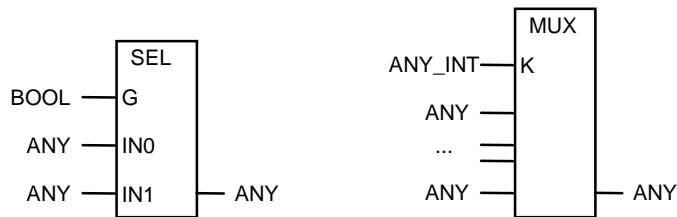
Figure A.8. Graphical declarations of the selection functions SEL and MUX

Name	Function	Description
SEL	Binary selection	F := IN0, if G = 0, otherwise IN1
MUX	Multiplexer	F := INi, if K = i

Table A.7. Description of the selection functions SEL and MUX

These two standard functions are specified in the following examples by declaration in ST:

```
FUNCTION       SEL : ANY         (* binary selection *)
  VAR_INPUT
    G    :    BOOL;
    IN0   :    ANY;
    IN1   :    ANY;
  END_VAR
  IF G = 0 THEN
      SEL    :=   IN0;       (* selection of upper input *)
  ELSE
      SEL    :=   IN1;       (*selection of lower input *)
  END_IF;
END_FUNCTION
```

Example A.3. Specification of the selection function SEL in ST

```
FUNCTION       MUX : ANY         (* multiplexer *)
  VAR_INPUT
    K    :    ANY_INT;
    IN0   :    ANY;
    IN1   :    ANY;
    ...
    INn   :    ANY;
  END_VAR
  IF (K < 0) OR (K>n) THEN
      ... error message .... ;        (* K negative or too large *)
  END_IF;
  CASE  K  OF
      0: MUX := IN0;        (* selection of upper input *)
      1: MUX := IN1;        (* selection of upper input *)
      ...
      n: MUX := INn;        (* selection of lowest input *)
  END_CASE;
END_FUNCTION
```

Example A.4. Specification of the selection function MUX in ST

A.8 Comparison Functions

*** stands for: GT, >, GE, >=,   *** stands for: NE, <>
                    LT, <, LE, <=,
                    EQ, =
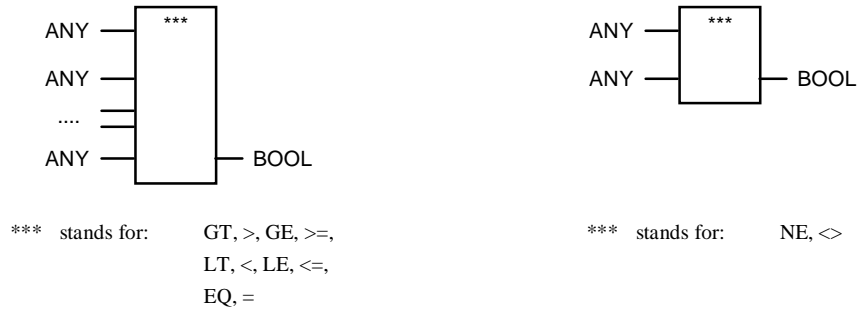
Figure A.9. Graphical declarations of the comparison functions GT, GE, LT, LE, EQ, NE

Name	Function	Description
GT	Comparison for „> "	F := 1, if INi > IN(i+1), otherwise 0
GE	Comparison for„> ="	F := 1, if INi >= IN(i+1), otherwise 0
LT	Comparison for„< "	F := 1, if INi < IN(i+1), otherwise 0
LE	Comparison for„< ="	F := 1, if INi <= IN(i+1), otherwise 0
EQ	Comparison for„= "	F := 1, if INi = IN(i+1), otherwise 0
NE	Comparison for„<>"	F := 1, if INi <> IN(i+1), otherwise 0

Table A.8. Description of the comparison functions

The specification of these standard functions is illustrated in Example A.5 by the declaration of GT in ST, from which the others can easily be derived.

```
FUNCTION      GT : BOOL          (* comparison for 'greater than' *)
  VAR_INPUT    IN1, IN2, ... INn  : ANY;     END_VAR
  IF      (IN1 > IN2)
  AND     (IN2 > IN3)
  ...
  AND     (IN(n-1) > INn) THEN
      GT :=  TRUE;        (* inputs are "sorted": in increasing monotonic sequence *)
  ELSE
      GT :=  FALSE;       (* condition not fulfilled *)
  END_IF;
END_FUNCTION
```

Example A.5. Specification of the comparison function GT in ST

A.9 Character String Functions
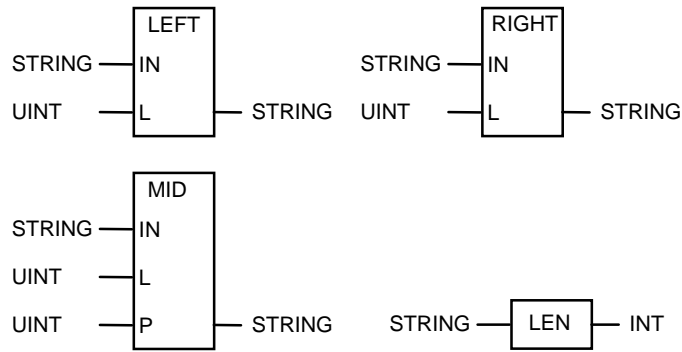
Figure A.10. Graphical declarations of the character string functions LEFT, RIGHT, MID and LEN
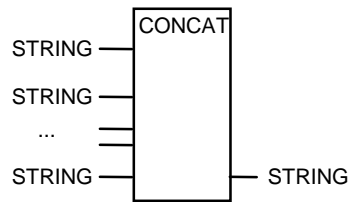
Figure A.11. Graphical declaration of the character string function CONCAT
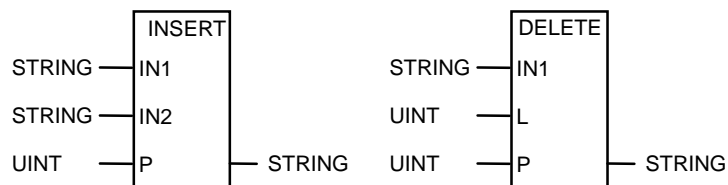
Figure A.12. Graphical declarations of the character string functions INSERT and DELETE
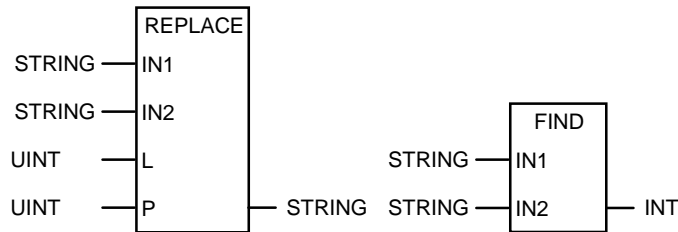
Figure A.13. Graphical declarations of the character string functions REPLACE and FIND

Name	Function	Description
LEN	Determines the length of a character string	F := number of the characters in IN
LEFT	Starting section of a character string	F := starting section with L characters
RIGHT	Final section of a character string	F := final section with L characters
MID	Central section of a character string	F := middle section from position P with L characters
CONCAT	Sequence of character strings	F := total character string
INSERT	Inserts one character string into another one	F := total character string with new part from position P
DELETE	Deletes section in a character string	F := remaining character string with deleted part (L characters) from position P
REPLACE	Replaces a section of a character string with another one	F := total character string with replaced part (L characters) from position P
FIND	Determines the position of a section in a character string	F := index of the found position, otherwise 0

Table A.9. Description of the character string functions

The positions of the characters within a character string of type STRING are numbered starting with position "1".

A.10 Functions for Time Data Types


```
TIME ──┤IN1                    TOD ──┤IN1                    DT ──┤IN1
        │    ***                      │    ***                     │    ***
TIME ──┤IN2 ├── TIME          TIME ──┤IN2 ├── TOD          TIME ──┤IN2 ├── DT
```

*** stands for: ADD, +, SUB, -
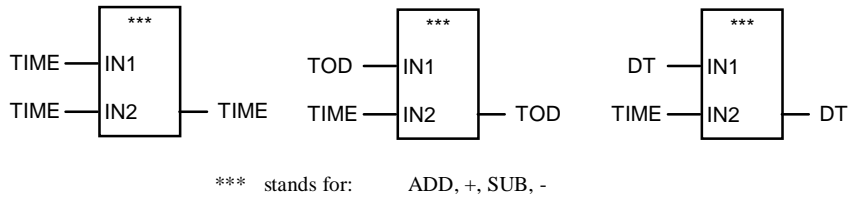
Figure A.14. . Graphical declarations of the common functions for addition and subtraction of time


```
DATE ──┤IN1                    TOD ──┤IN1                    DT ──┤IN1
        │    ***                      │    ***                     │    ***
DATE ──┤IN2 ├── TIME           TOD ──┤IN2 ├── TIME           DT ──┤IN2 ├── TIME
```

*** stands for: SUB, -
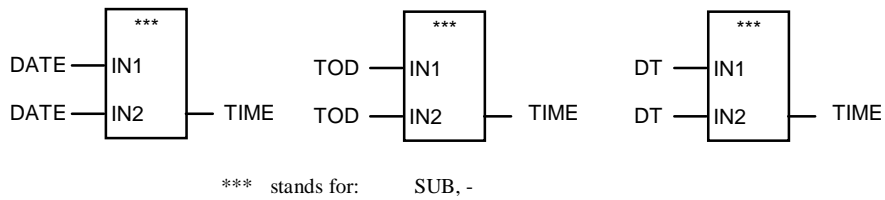
Figure A.15. Graphical declarations of the additional functions for subtraction of time


```
TIME    ──┤IN1                           DT ──┤IN1
           │    ***                            │  CONCAT
ANY_NUM ──┤IN2 ├── TIME                 TOD ──┤IN2 ├── DT
```

*** stands for: MUL, *, DIV, /
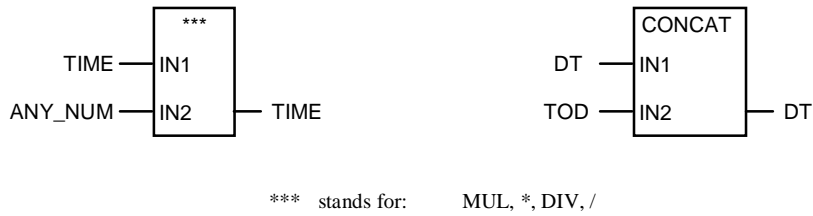
Figure A.16 Graphical declarations of the functions MUL, DIV and CONCAT for time
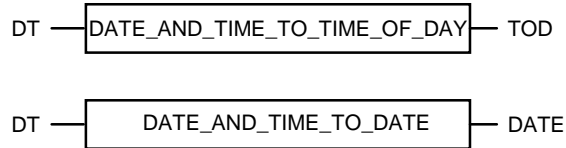
Figure A.17. Graphical declarations of the type conversion functions for time

The abbreviations TOD and DT can be employed as equivalents for the longer keywords TIME_OF_DAY and/or DATE_AND_TIME.

A.11 Functions for Enumerated Data Types

The standard functions SEL, MUX, EQ and NE can also be employed in some cases for enumerated data types. They are then applied as for integers (values of enumerations correspond to constants "coded" by the programming system).

B Standard Function Blocks

This appendix contains a complete overview of all the standard PLC function blocks described by means of examples in Chapter 2. For every standard function block of IEC 61131-3, the following information is given:

- Graphical declaration
- (Semantic) description
- Specification of some function blocks in Structured Text (ST).

In this book, the inputs and outputs of the standard FBs are given the names prescribed by the current version of IEC 61131-3 ([IEC 61131-3-94]). No allowance has been made for the fact that the possibility of calling FBs as "IL operators" can result in conflicts. These will necessitate either a change in the standard itself or a change in nomenclature when implementing programming systems in accordance with IEC 61131-3.

   These conflicts occur with the IL operators (see also Section 4.1) LD, R and S, which need to be distinguished from the FB inputs LD, R and S when checking for correct program syntax. These three formal operands could, for example, be called LOAD, RESET and SET in future.

B.1 Bistable Elements (Flip-Flops)
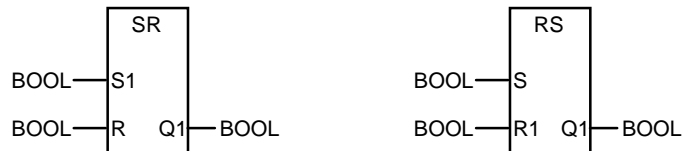
Figure B.1. Graphical declarations of the bistable function blocks SR and RS

```
FUNCTION_BLOCK      SR          (* flip flop set dominant *)
  VAR_INPUT
    S1   :    BOOL;
    R    :    BOOL;
  END_VAR
  VAR_OUTPUT
    Q1   :    BOOL;
  END_VAR
  Q1     :=  S1 OR ( NOT R AND Q1);
END_FUNCTION_BLOCK

FUNCTION_BLOCK      RS          (* flip flop reset dominant *)
  VAR_INPUT
    S    :    BOOL;
    R1   :    BOOL;
  END_VAR
  VAR_OUTPUT
    Q1   :    BOOL;
  END_VAR
  Q1     :=  NOT R1 AND ( S OR Q1);
END_FUNCTION_BLOCK
```

Example B.1 Specification of the bistable function blocks SR and RS in ST

These two flip-flops implement dominant setting and resetting.

B.2 Edge Detection
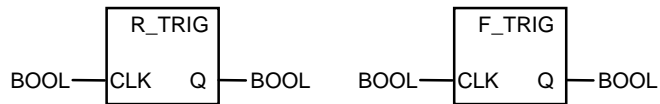
Figure B.2. Graphical declarations of the function blocks R_TRIG and F_TRIG

```
FUNCTION_BLOCK      R_TRIG     (* rising edge *)
 VAR_INPUT
   CLK :    BOOL;
 END_VAR
 VAR_OUTPUT
   Q    :    BOOL;
 END_VAR
 VAR RETAIN
   MEM :    BOOL := 0;                 (* initialise edge flag *)
 END_VAR
 Q       := CLK AND NOT MEM;           (* recognise rising edge *)
 MEM    := CLK;                        (* reset edge flag *)
END_FUNCTION_BLOCK

FUNCTION_BLOCK      F_TRIG     (* falling edge *)
 VAR_INPUT
   CLK :    BOOL;
 END_VAR
 VAR_OUTPUT
   Q    :    BOOL;
 END_VAR
 VAR RETAIN
   MEM :    BOOL := 1;                 (* initialise edge flag *)
 END_VAR
 Q       := NOT CLK AND NOT MEM;       (* recognise falling edge *)
 MEM    := NOT CLK;                    (* reset edge flag *)
END_FUNCTION_BLOCK
```

Example B.2. Specification of the function blocks R_TRIG and F_TRIG in ST

In the case of the function blocks R_TRIG and F_TRIG, it should be noted that they detect an "edge" on the **first** call if the input of R_TRIG is TRUE or the input of F_TRIG is FALSE.

B.3 Counters
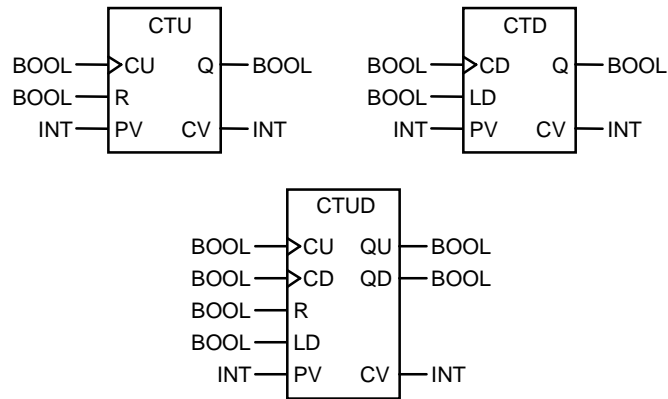
Figure B.3. Graphical declarations of the function blocks CTU, CTD and CTUD

```
FUNCTION_BLOCK        CTU       (* up counter *)
  VAR_INPUT
    CU  :   BOOL    R_EDGE;    (* CU with rising edge *)
    R   :   BOOL;
    PV  :   INT;
  END_VAR
  VAR_OUTPUT
    Q   :   BOOL;
    CV  :   INT;
  END_VAR
  IF  R  THEN                  (* reset counter *)
       CV     :=  0;
  ELSIF  CU AND ( CV < PV)  THEN
       CV      :=  CV + 1;     (* count up *)
  ENDIF;
  Q    :=   (CV >= PV);        (* limit reached *)
END_FUNCTION_BLOCK
```

Example B.3. Specification of the function blocks CTU and CTD in ST (continued on next page)

```
FUNCTION_BLOCK      CTD      (* down counter *)
 VAR_INPUT
   CD   :   BOOL    R_EDGE;   (* CD with falling edge *)
   LD   :   BOOL;
   PV   :   INT;
 END_VAR
 VAR_OUTPUT
   Q    :   BOOL;
   CV   :   INT;
 END_VAR
 IF  LD  THEN                 (* reset counter *)
      CV    := PV;
 ELSIF  CD AND ( CV > PV)  THEN
      CV    := CV - 1;        (* count down *)
 ENDIF;
 Q    :=   (CV <= 0);         (* zero reached *)
END_FUNCTION_BLOCK
```

Example B.3. (Continued)

```
FUNCTION_BLOCK      CTUD     (* up-down counter *)
 VAR_INPUT
   CU   :   BOOL    R_EDGE;   (* CU with rising edge *)
   CD   :   BOOL    R_EDGE;   (* CD with falling edge *)
   R    :   BOOL;
   LD   :   BOOL;
   PV   :   INT;
 END_VAR
 VAR_OUTPUT
   QU   :   BOOL;
   QD   :   BOOL;
   CV   :   INT;
 END_VAR
 IF  R  THEN                  (* reset counter (reset dominant) *)
      CV   :=   0;
 ELSIF  LD  THEN
      CV   :=   PV;                     (* set to count value *)
 ELSIF  CU AND ( CV < PV)  THEN
      CV   :=   CV + 1;       (* count up *)
 ELSIF  CD AND ( CV > PV)  THEN
      CV   :=   CV - 1;       (* count down *)
 ENDIF;
 QU   :=   (CV >= PV);        (* limit reached *)
 QD   :=   (CV <= 0);         (* zero reached *)
END_FUNCTION_BLOCK
```

Example B.4. Specification of the function block CTUD in ST

B.4 Timers

***   stands for: TON, T---0,   TOF, 0---T,   TP
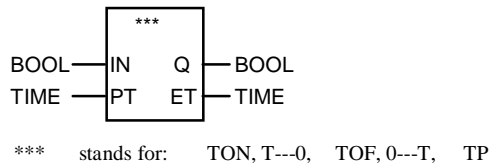
Figure B.4. Graphical declarations of the function blocks TON, TOF and TP

The timers TP, TON and TOF are specified here using timing diagrams.

   This time behaviour is only possible if the cycle time of the cyclic PLC program in which the timer is used is negligibly small in comparison with the duration PT if the timer is called only once in the cycle.

   The diagrams show the behaviour of outputs Q and ET depending on input IN. The time axis runs from left to right and is labelled "t". The Boolean variables IN and Q change between "0" and "1" and the time value ET increases as shown.
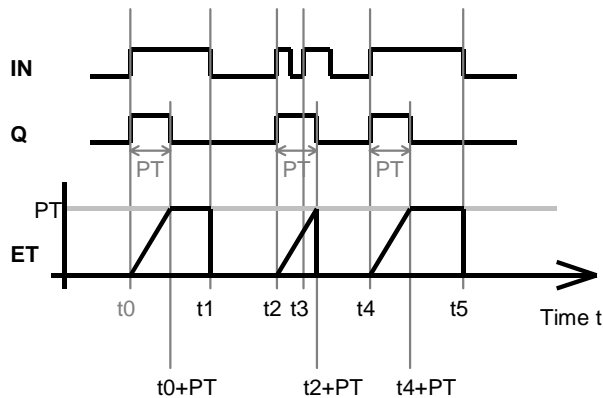
Figure B.5. Timing diagram for pulse timer TP depending on input IN

The standard FB "TP" acts as a pulse generator which supplies a pulse of constant length at output Q when a rising edge is detected at input IN. The time that has elapsed so far can be read off at output ET at any time.

   As can be seen from Figure B.5, timers of type TP are not "retriggerable". If the intervals between the input pulses at IN are shorter than the pre-set time period, the pulse duration still remains constant (see period [t2; t2+PT]). Timing therefore does *not* begin again with every rising edge at IN.
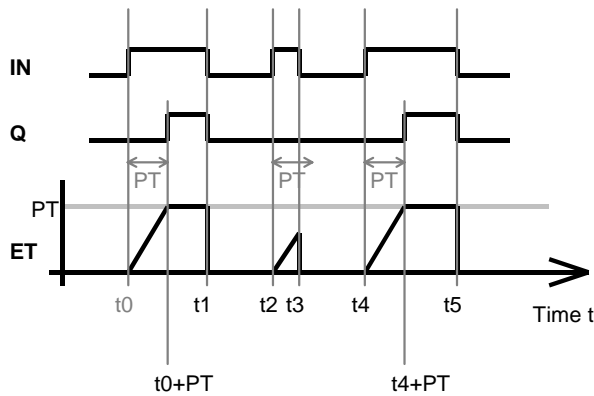
Figure B.6. Timing diagram for on-delay timer TON depending on input IN

The on-delay timer TON supplies the input value IN at Q with a time delay when a rising edge is detected at IN. If input IN is only "1" for a short pulse (shorter than PT), the timer is not started for this edge.

The elapsed time can be read off at output ET.
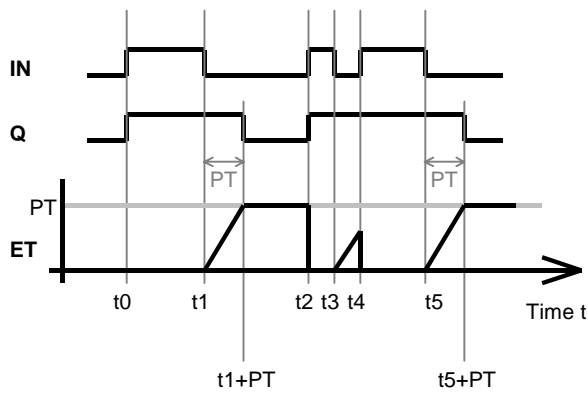
Figure B.7. Timing diagram for off-delay timer TOF depending on input IN

The off-delay timer performs the inverse function to TON i.e. it delays a falling edge in the same way as TON delays a rising one.

The behaviour of the timer TOF if PT is modified during timer operation is implementation-dependent.

```
              ┌──────────────┐
              │     RTC      │
BOOL──────────┤IN         Q  ├──────BOOL
   DT─────────┤PDT     CDT   ├──DT
              └──────────────┘
```
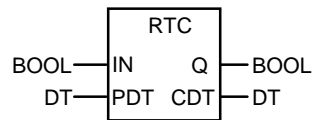
Figure B.8. Graphical declaration of the function block RTC (real time clock)

```
FUNCTION_BLOCK     RTC          (* real time clock *)
  VAR_INPUT
    IN   :   BOOL;
    PDT  :   DATE_AND_TIME;
  END_VAR
  VAR_OUTPUT
    Q    :   BOOL;
    CDT  :   DATE_AND_TIME;
  END_VAR
  VAR
    IN_Edge    : BOOL  R_EDGE;
    Actual_Time : DATE_AND_TIME;        (* PLC hardware clock *)
  END_VAR
  IF  IN_Edge (IN)  THEN                (* edge recognised? *)
      Actual_Time := PDT;              (* load PLC clock *)
  ENDIF;
  CDT    := Actual_Time;               (* undefined without edge at IN *)
  Q       := IN;                       (* Q=1: no error *)
END_FUNCTION_BLOCK
```

Example B.5. Specification of the function block RTC in ST

The real-time clock RTC supplies the current time including date. When a rising edge is detected at IN, the time is set and runs as long as IN remains "1". In a real PLC system, input IN could indicate that the central power supply is on and that the clock can therefore operate. After a power failure and/or cold restart of the system, the time would be set on a "0→1" transition. If no rising edge occurs at input IN, CDT is regarded as "undefined".

ActualTime is used here (informally) to mean the central time in the PLC, which can be recorded by software or, more usually, by a hardware clock. The realisation of ActualTime is therefore implementation-dependent.

C IL Examples

This appendix contains full examples of PLC programming with IEC 61131-3 for each type of POU, to supplement the information given in Chapters 2 and 4.

These examples are to be found on the CD enclosed in this book.

C.1 Example of a FUNCTION

The function ByteExtr extracts the upper or lower byte of an input word and returns it as the function value:

```
FUNCTION  ByteExtr : BYTE    (* extract byte from word *)
                             (* beginning of declaration part *)
VAR_INPUT                    (* input variables *)
  Word  :   WORD;            (* word consists of upper + lower byte *)
  Upper :   BOOL;            (* TRUE: take upper byte, else lower *)
END_VAR
                             (* beginning of instruction part *)
LD      Upper                (*extract upper or lower byte? *)
EQ      FALSE                (* lower? *)
JMPCN  UpByte                (* jump in the case of extraction of the upper byte *)
LD      Word                 (* load word *)
JMP     End                  (* nothing to do *)
UpByte:                      (* jump label *)
LD      Word                 (* load word *)
SHR     8                    (* shift upper byte 8 bits to the right *)
End:                         (* ready *)
WORD_TO_BYTE                 (* conversion for type compatibility *)
ST      ByteExtr             (* assignment to the function value *)
RET                          (* return with function value in CR *)
                             (* end of FUN *)
END_FUNCTION
```

Example C.1. Example of the declaration of a function in IL

The ByteExtr function in Example C.1 has the input parameter "Word" of type WORD and the Boolean input Upper. The value returned by the function is of type BYTE. This function requires no local variables. The VAR ... VAR_END section is therefore missing from the declaration part.

The returned value is in the current result (CR) when the function returns to the calling routine with RET. At this point (jump label End:) the CR is of data type WORD, because Word was previously loaded. The function value is of data type BYTE after type conversion.

   IEC 61131-3 always requires a strict "type compatibility" in cases like this. It is the job of the programming system to check this consistently. This is why a standard type conversion function (WORD_TO_BYTE) is called in Example C.1.

Example C.2 shows the instruction part of ByteExtr in the ST programming language.

```
FUNCTION ByteExtr : BYTE      (* extraction byte from word *)
VAR_INPUT ... END_VAR         (* as above *)
IF Upper THEN
  ByteExtr := WORD_TO_BYTE (SHR (Word, 8) );
ELSE
  ByteExtr := WORD_TO_BYTE (Word);
END_IF;
END_FUNCTION
```

Example C.2. Instruction part of Example C.1 in ST

FUNCTION

...
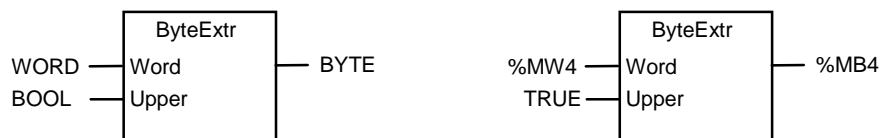
END_FUNCTION

Example C.3. Graphical declaration part of the function declaration in Example C.1 (left) with an example of a call (right)

Example C.3 shows the declaration part and an example of a call for the function ByteExtr in graphical representation. The call replaces the lower byte of flag word 4 by its upper byte.

C.2 Example of a FUNCTION_BLOCK

The function block DivWithRem calculates the result of dividing two integers and returns both the division result and the remainder. "Division by zero" is indicated by an output flag.

```
FUNCTION_BLOCK DivWithRem    (* division with remainder *)
                             (* beginning of declaration part *)
VAR_INPUT                    (* input parameter *)
  Dividend :    INT;         (* integer to be divided *)
  Divisor  :    INT;         (* integral divisor *)
END_VAR
VAR_OUTPUT RETAIN            (* retentive output parameters *)
  Quotient :    INT;         (* result of the division *)
  DivRem   :    INT;         (* remainder after division *)
  DivError :    BOOL;        (* flag for division by zero *)
END_VAR
                             (* beginning of instruction part *)
LD       0                   (* load zero *)
EQ       Divisor             (* divisor equal to zero? *)
JMPC     Error               (* catch error condition*)
LD       Dividend            (* load dividend, divisor not equal to zero *)
DIV      Divisor             (* carry out division *)
ST       Quotient            (* store integral division result *)
MUL      Divisor             (* multiply division result by divisor *)
ST       DivRem              (* store interim result *)
LD       Dividend            (* load dividend *)
SUB      DivRem              (* subtract interim result *)
ST       DivRem              (* yields "remainder" of the division as an integer*)
LD       FALSE               (* load logical "0" for error flag *)
ST       DivError            (* reset error flag *)
JMP      End                 (* ready, jump to end *)
Error:                       (* handling routine for error "division by zero" *)
LD       0                   (* zero, since outputs are invalid in event of error *)
ST       Quotient            (* reset Result *)
ST       DivRem              (* reset Remainder *)
LD       TRUE                (* load logical "1" for error flag *)
ST       DivError            (* set error flag *)
End:
RET
                             (* end of FB *)
END_FUNCTION_BLOCK
```

Example C.4. Example of the declaration of a function block in IL

The FB DivWithRem in Example C.4 performs integer division with remainder on the two input parameters Dividend and Divisor. In the case of division by zero, the error output DivError is set and the other two outputs are set in a defined manner to zero since they are invalid. The outputs are retentive, i.e. they are retained within the FB instance from which DivWithRem was called.

This example cannot be formulated as a function because there are three output variables.

Example C.5 shows the instruction part of DivWithRem in the ST programming language.

```
FUNCTION_BLOCK DivWithRem    (* division with remainder *)
VAR_INPUT ... END_VAR          (* as above *)
VAR_OUTPUT RETAIN ... END_VAR
IF Divisor = 0 THEN
  Quotient := 0;
  DivRem := 0;
  DivError := TRUE;
ELSE
  Quotient := Dividend / Divisor;
  DivRem := Dividend - (Quotient * Divisor);
  DivError := FALSE;
END_IF;
END_FUNCTION_BLOCK
```

Example C.5. Instruction part of Example C.4 in ST

Example C.6 shows the declaration part and an example of a call of the function DivWithRem in graphical representation. The FB must be instantiated (DivInst) before it can be called.

After execution of the function block, the output variables of this instance have the following values: DivInst.Quotient = 33, DivInst.DivRem = 1 and DivInst.DivError = FALSE. These return values are assigned to the variables Result, Remainder and Error respectively.

FUNCTION_BLOCK DivWithRem

```
            ┌─────────────────────────┐
INT ────────┤Dividend      Quotient├──── INT
INT ────────┤Divisor       DivRem  ├──── INT
            │              DivError├──── BOOL
            └─────────────────────────┘
```

...

END_FUNCTION_BLOCK

```
                  DivInst
            ┌─────────────────────────┐
            │       DivWithRem         │
100 ────────┤Dividend      Quotient├──── Result
  3 ────────┤Divisor       DivRem  ├──── Remainder
            │              DivError├──── Error
            └─────────────────────────┘
```
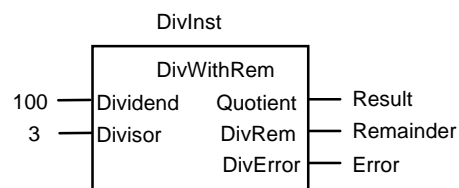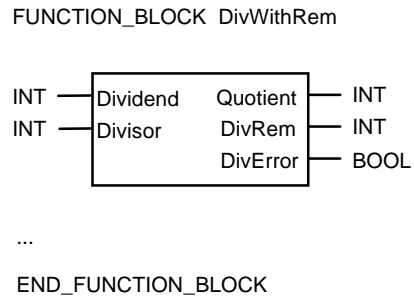
Example C.6. Graphical declaration part for Example C.4 (top) and example of a call of the FB instance DivInst (bottom)

C.3 Example of a PROGRAM

The program MainProg in Example C.7 is not a complete programming example, but shows ways of implementing problems and illustrates the use of variables in POUs of type PROGRAM.

MainProg first starts a real-time clock DateTime that records the date and time (standard FB RTC). This clock is used to find out how long an interruption in the program has lasted (TimeDiff). The PLC system must be able to detect the interruption (Ress_Running) and must be able to access a hardware clock with an I/O address (ActDateTime).

```
PROGRAM    MainProg              (* example of a main program *)
                                 (* beginning of declaration part *)
VAR_INPUT                        (* input variables *)
  T_Start     : BOOL := FALSE;   (* input starting condition *)
END_VAR
VAR_OUTPUT                       (* output variables *)
  T_Failure   : BOOL := FALSE;   (* output "failure" *)
END_VAR
VAR_GLOBAL RETAIN                (* global retentive data area *)
  Ress_Running AT %MX255.5 : BOOL; (* running flag for resource/PLC-CPU *)
  DateTime    : RTC;             (* programm clock: date and time *)
  ActDateTime AT %MD2 : DT;      (* hardware clock: actual date with time *)
END_VAR
VAR_GLOBAL                       (* global data area *)
  EmergOff  AT %IX255.0 : BOOL;  (* contact Emergency-Off *)
  ProgRun     : BOOL := FALSE;   (* "running" flag *)
  Error       : BOOL;            (* error flag *)
  Err_Code    : UDINT := 0;      (* Error code, 32 bit unsigned *)
END_VAR
VAR                              (* local variables *)
  AT %IX250.2 : BOOL;            (* directly represented variables *)
  ErrorProc   : CErrProc;        (* FB instance of CErrProc *)
  Edge        : R_TRIG;          (* edge detection *)
  TimeDiff    : TIME := t#0s;    (* time difference *)
END_VAR

                                 (* beginning of instruction part *)
LD     FALSE
ST     Error                     (* reset error flag *)

...                    (* determine how long a power failure and/or an interruption *)
                          (* of the CPU lasted; Clock "DateTime" is battery-backed *)
LD     t#0s                      (* zero seconds *)
ST     TimeDiff                  (* reset *)
LD     DateTime.Q                (* time valid = clock running *)
JMPC  Goon                       (* valid - nothing to do *)
LD     ActDateTime               (* check current time *)
SUB   DateTime.CDT               (* last time before power failure *)
ST     TimeDiff                  (* duration of power failure *)
Goon:
...
```

Example C.7. Example of the declaration of a main program in IL. The FB CErrProc ("central error processing") must already be available. (Continued on next page)

```
LD      Ress_Running               (* CPU is running *)
ST      DateTime.IN                (* clock running if CPU is running *)
LD      ActDateTime                (* initial value Date/Time *)
ST      DateTime.PDT               (* initial value of clock is loaded *)
                                   (* if there is a rising edge at IN *)
CAL     DateTime                   (* start real-time clock)
...
LD      TimeDiff                   (* interruption duration *)
LT      t#50m                      (* less than 50 seconds? *)
JMPC    Continue                   (* plant still warm enough ... *)
NOT
S       Error                      (* set error flag *)
LD      16#000300F2
ST      Err_Code                   (* store error cause *)
Continue:
LD      EmergOff                   (* "emergency off" pressed? *)
ST      Edge.CLK                   (* input edge detection *)
CAL     Edge                       (* compare input with edge flag *)
LDN     Edge.Q                     (* edge at EmergOff recognized? *)
AND     T_Start                    (* AND start flag set *)
ANDN    Error                      (* AND no new error *)
AND     ErrProc.Ack                (* AND error cause repaired *)
ST      ProgRun                    (* global "running"-condition *)

....                        (* ...real instructions, calls of FUN/FBs... *)

LD      Error                      (* error occurred? *)
AND     %IX250.2
R       ProgRun                    (* reset global starting condition *)
LD      ProgRun
JMPNC   End                        (* in the case of error: start error handler *)
CALC    ErrProc(Code := Err_Code)  (* FB with central error handling *)
LD      ErrProc.Ack                (* flag: error acknowledged *)
End:
LD      ProgRun                    (* program not running *)
ST      T_Failure                  (* set output parameter *)
RET                                (* return and/or end *)
                                   (* end of program *)

END_PROGRAM
```

Example C.7. (Continued)

Edge detection is also employed in this program in order to find out whether the Emergency Off button has been pressed.

In the global data area, the variable ProgRun is declared, which is available to all function blocks called under MainProg (as VAR_EXTERNAL). This is linked with the starting condition provided EmergOff has not been pressed.

FB instance ErrProc can handle errors with error code Err_Code. When the error has been corrected and acknowledged, the corresponding output is set to TRUE.

```
RESOURCE CentralUnit_1 ON CPU_001
  TASK Periodic (INTERVAL := time#13ms, PRIORITY := 1);
  PROGRAM Applic WITH Periodic : MainProg (   T_Start := %I250.0,
                                              T_Failure => %Q0.5);
END_RESOURCE
```

Example C.8. Resource definition with run-time program Applic for Example C.7

The program MainProg is provided with the properties of the periodic task Periodic and becomes the run-time program Applic of the resource (PLC-CPU) CentralUnit_1. This run-time program runs with highest priority (1) as a periodic task with a maximum cycle time of 13 ms.

MainProg is called with the value of input bit %I250.0 for the input variable T_Start and sets output bit %Q0.5 with the value of T_Failure.

D Standard Data Types

This appendix summarises all the elementary data types and their features in tabular form. Their use is explained in Chapter 3.

IEC 61131-3 defines five groups of elementary data types. Their generic data types are indicated in brackets (see Table 3.9):

- Bit string   (ANY_BIT),
- Integer, signed and unsigned   (ANY_INT),
- Real (floating-point)   (ANY_REAL),
- Date, time-of-day   (ANY_DATE),
- Character string, duration, derived (ANY).

The following information is given for each data type, listed in separate tables for each group:

- Name (keyword),
- Description,
- Number of bits (data width),
- Value range (using of the IEC literals),
- Initial values (default values).

The data width and permissible range of the data types in Tables D.5 and D.6 are application-dependent.

Data type	Description	Bits	Range	Initial
BOOL	Boolean	1	[0,1]	0
BYTE	Bit string 8	8	[0,...,16#FF]	0
WORD	Bit string 16	16	[0,...,16#FFFF]	0
DWORD	Bit string 32	32	[0,...,16#FFFF FFFF]	0
LWORD	Bit string 64	64	[0,...,16#FFFF FFFF FFFF FFFF]	0

Table D.1. "Boolean and Bit String" data types

Data type	Description	Bits	Range	Initial

SINT	Short integer	8	[-128,...,+127]	0
INT	Integer	16	[-32768,...,+32767]	0
DINT	Double integer	32	$[-2^{31},...,+2^{31}-1]$	0
LINT	Long integer	64	$[-2^{63},...,+2^{63}-1]$	0

Table D.2. . "Signed Integer" data types

Data type	Description	Bits	Range	Initial
USINT	Unsigned short integer	8	[0,...,+255]	0
UINT	Unsigned integer	16	[0,...,+65535]	0
UDINT	Unsigned double integer	32	$[0,...,+2^{32}-1]$	0
ULINT	Unsigned long integer	64	$[0,...,+2^{64}-1]$	0

Table D.3. "Unsigned Integer" data types

Data type	Description	Bits	Range	Initial
REAL	Real numbers	32	See IEC 559	0.0
LREAL	Long reals	64	See IEC 559	0.0

Table D.4. "Real Numbers" data types (floating-point numbers)

Data type	Description	initial
DATE	Date (only)	d#0001-01-01
TOD	Time of day (only)	tod#00:00:00
DT	Date and time of day	dt#0001-01-01-00:00:00

Table D.5. "Date and Time" data types

The full keyword TIME_OF_DAY can also be used in place of TOD, and DATE_AND_TIME in place of DT.

Data type	Description	Initial
TIME	Duration	t#0s
STRING	Character string (variable length)	''

Table D.6. "Duration and Character String" data types. The initial value for STRING is an "empty" character string.

E Causes of Error

IEC 61131-3 requires manufacturers to provide a list of responses to the following error conditions. See also Section 7.10. The responses fall into four different categories:

1) No system response (%),
2) Warning during program creation (WarnPc),
3) Error message during program creation (ErrPc),
4) Error message and response to error at run time (ErrRun).

No.	Error cause	Response
1	Value of a variable exceeds the specified range.	ErrRun
2	Length of the initialisation list does not match the number of array entries.	ErrPc
3	Type conversion errors	ErrPc
4	Numerical result (of a standard function) exceeds the range for the data type; Division by zero (in a standard function).	ErrRun ErrPc, ErrRun
5	Mixed input data types to a selection function (standard function); Selector (K) out of range for MUX function.	ErrPc ErrRun
6	Invalid character position specified; Result exceeds maximum string length (INSERT/CONCAT result is too long).	WarnPc, ErrRun WarnPc, ErrRun
7	Result exceeds range for data type TIME	ErrRun

Table E.1. Error causes. The manufacturer supplies a table specifying the system response to the errors described above. The entry in the *"Response"* column is the appropriate time for the response in accordance with IEC 61131-3. (Continued on next page.)

8	An FB instance used as an input parameter has no parameter values.	ErrPc
9	A VAR_IN_OUT parameter has no value.	ErrPc
10	Zero or more than one initial steps in SFC network; User program attempts to modify step state or time.	ErrPc ErrPc
11	Simultaneously true, non-prioritised transitions in a selection divergence.	WarnPc, ErrPc
12	Side effects in evaluation of transition conditions.	ErrPc
13	Action control contention error.	WarnPc, ErrRun
14	Unsafe or unreachable SFC.	ErrPc
15	Data type conflict in VAR_ACCESS.	ErrPc
16	Tasks require too many processor resources; Execution deadline not met; Other task scheduling conflicts.	ErrPc ErrPc WarnPc, ErrRun
17	Numerical result exceeds range for data type (IL).	WarnPc, ErrRun
18	Current result (CR) and operand type do not match (IL).	ErrPc
19	Division by zero (ST); Invalid data type for operation (ST).	WarnPc, ErrPc,ErrRun ErrPc
20	Return from function without value assigned (ST).	ErrPc
21	Iteration fails to terminate (ST).	WarnPc,ErrPc, ErrRun
22	Same identifier used as connector label and element name (LD / FBD).	ErrPc
23	Uninitialised feedback variable	ErrPc

Table E.1. (Continued)

This table is intended as a guide. There are other possible errors which are not included in the IEC table. It should therefore be extended by every manufacturer as appropriate.

F Implementation-Dependent Parameters

Table F.1 lists the implementation-dependent parameters defined by IEC 61131-3. See also Section 7.11.

No.	Implementation-dependent parameters
1	Error handling procedures which are supported by the manufacturer.
2	Information about use of the following national characters and/or their substitutes: £ instead of #, if # occupied by national character, Fitting currency symbol instead of $, if $ occupied by national character, ! instead of |, if | occupied by national character.
3	Maximum length of identifiers.
4	Maximum comment length (without leading or trailing brackets).
5	Range of values of duration (e.g. 0<=Hours<24; 0<=Min<60).
6	Range of values for variables of data type TIME (e.g. 0<=TimeVar<365 Days). Precision of representation of seconds in data types TIME_OF_DAY and DATE_AND_TIME.
7	Maximum: - number of elements in an array (number of array subscripts), - array size (number of bytes), - number of structure elements, - structure size (number of bytes), - number of variables per declaration that can be declared with the same data type (separated by commas).
8	Maximum number of enumerated values (data type ENUM).

Table F.1. Implementation-dependent parameters which every manufacturer of an IEC 61131-3 system must describe. The division into individual groups relates to individual sections in the standard. If no concrete figures are required, the manufacturer can add an informal description. (Continued on next page.)

9	Default maximum length of STRING variables. Maximum allowed length of STRING variables.
10	Maximum number of hierarchical levels for directly represented variables (e.g. %QX1.1.1.2...). Logical or physical mapping (symbols %IX.... onto the real hardware).
11	Maximum number of subscripts for access to an array element. Maximum range of subscript values. Maximum number of structure levels (depth of sub-structuring).
12	Initialisation of system inputs (%IX...) at system start time.
13	Information to determine execution times of program organisation units (POU) (no details are given in IEC 61131-3).
14	Method of function representation: textual or graphic (symbols, such as "+", or names, such as "ADD").
15	Maximum number of function specifications (if limited).
16	Maximum number of inputs of extensible functions.
17	Effects of type conversions on accuracy (REAL_TO_INT, ...).
18	Accuracy of functions of one variable (LOG, SIN, ...). Implementation of arithmetic (overloaded) functions.
19	Maximum number of function block specifications and instantiations (if limited).
20	Range of parameter PV (end value of counter function blocks).
21	System reaction to a change of the input PT (end value of timer function blocks) during operation .
22	Number and length of SEND inputs and RCV outputs specified in IEC 61131-5.
23	Program size limitations (executable code)
24	Timing and portability effects of execution control elements (SFC). Functional (and/or informal) description of SFC implementation.
25	Precision of step elapsed time. Maximum number of steps per SFC network and per POU.
26	Maximum number of transitions per SFC network and per POU.
27	Action control mechanism (if available).
28	Maximum number of action blocks per step.
29	Graphic indication of step state (e.g. by inverse representation or additional characters). This is not prescribed by the standard. (Minimum) transition clearing time (caused by PLC cycle time). Maximum number of predecessor and successor steps in diverge/converge constructs.
30	Contents of RESOURCE libraries. Every processing unit (e.g. processor type) receives a description of all functions (standard functions, standard FBs, data types, ...) that can be processed by this resource.

Table F.1. (Continued on next page)

31	Maximum number of tasks / resource. Task interval resolution (time between calls for periodic tasks); Type of task priority control (pre-emptive or non-pre-emptive scheduling).
32	Maximum length of expressions (ST) (number of operands, operators). Partial evaluation of Boolean expressions (for avoidance of unwanted side-effects).
33	Maximum length of statements (ST) (IF; CASE; FOR; ...); restrictions.
34	Maximum number of CASE selections (ST).
35	Value of control variable upon termination of FOR loop.
36	Kind of graphic representation (semigraphic or graphic) for graphic programming languages. Restrictions on network topology (LD/FBD)
37	Evaluation order of feedback loops.
38	Description of the execution order of networks.

Table F.1. (Continued)

There are a large number of other parameters that need to be considered when implementing a programming system compliant with IEC 61131-3.

Table F.2 gives a subjective selection of these parameters.

No.	**Implementation-dependent parameters**
1	Extent of syntax and semantic checking provided by the textual and graphical editors (during input).
2	Free placement of graphic elements in the case of editors with line updating ("elastic band") or automatic placement according to syntax rules.
3	Declaration and use of Directly Represented Variables (DRVs): - In the declaration the symbolic name is always used. In the code part *either* the symbol alone *or* both may be used (symbol; direct physical address). Or - DRVs may also be declared without symbolic names (use of the direct physical address only). Or - The programming system declares DRVs implicitly.
4	Sequence order of VAR_*...VAR_END blocks as well as multiple use of blocks with the same name (variable type).
5	Function calling mechanisms implemented in ST (with/without formal parameters).
6	Extent of implementation of the EN/ENO parameters in LD and FBD, and possible effects on IL/ST program sources.

Table F.2. Other implementation-dependent parameters (not part of IEC 61131-3) (Continued on next page)

7	Possibility of passing user-defined structures (TYPE) as function and FB input parameters (not defined in the standard at present).
8	Global and POU-wide publication of user-defined data types (TYPE...END_TYPE) (not defined in the standard at present).
9	Extent of data area checking during program creation and at run time.
10	Graphical representation of qualifiers in variable declarations.
11	Multiple instantiation of the real time clock RTC or use of *a single* instance for all calls.
12	Restrictions on use of complex data types (function type also permitted for type "string" or user-defined types, ...?).
13	Algorithm for the evaluation of LD/FBD networks (see Sections 4.3.4 and 7.4.1).
14	Aids for comprehensibility of cross-compiled programs (if implemented).
	...

Table F.2. (Continued)

IEC 61131-3 expressly allows functionality beyond the standard. However, they must be described. Table F.3 gives some examples.

No.	Extensions
1	Extending range checking to more data types than only integer (ANY_INT), e.g. ANY_NUM.
2	Accepting FB instances as array variables.
3	Permitting FB instances in structures.
4	Allowing overloading for user-defined functions, function blocks and programs.
5	Time when the step width is calculated in the case of FOR statements.
6	Possible use of pre-processor statements for literals, macros, conditional compiling, Include statements (for input of files with FB interface information/prototypes, with the list of directly represented variables or EXTERNAL declarations employed, ...).
7	Additional declaration facility in FBs for creating dynamic (non-static) variables (e.g. VAR_DYN...END_VAR), to reduce storage space requirements in the PLC.
8	Use of different memory models in the PLC (Small; Compact; Large,...)
	...

Table F.3. Possible extensions (not part of IEC 61131-3)

G IL Syntax Example

Many of the examples given in this book are formulated in Instruction List (IL). This programming language is widely used and is supported by most programming systems. By including data types previously only found in high-level languages, such as arrays or structures, the IEC 61131-3 Instruction List language opens up new possibilities compared with conventional IL.

The IL syntax descriptions in this appendix are presented in simplified form in syntax diagrams.

Syntax diagrams explain the permissible use of delimiters, keywords, literals and names in a readily comprehensible format. They can easily be put into textual form for the development of compilers and define the formal structure of a programming language *(syntax* of a language). The reader can use the diagrams for reference.

The syntax descriptions in this appendix go beyond IEC 61131-3 because, in addition to the pure syntax definitions, they also include semantic conditions (consistent use of the current result, use of function parameters, etc.). IEC 61131-3 only offers an informal description of this.

The rules are outlined in Section G.1. The use of the diagrams is explained in Section G.2 by means of an example.

G.1 Syntax Diagrams for IL

If a node in the syntax diagram has further subdivisions (sub-diagram), its name appears in italics. Keywords or terminal symbols which are not further subdivided appear in standard type.

IL instruction sequence:
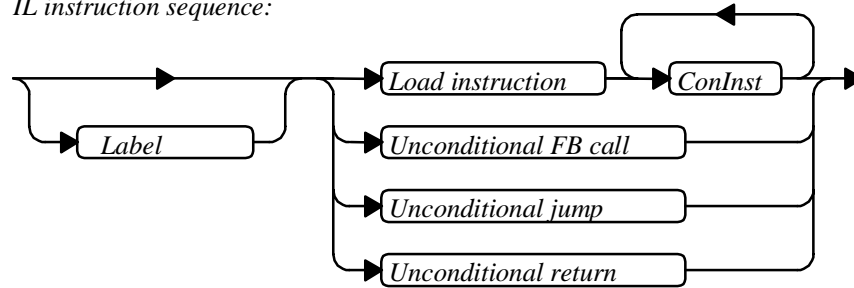
ConInst:
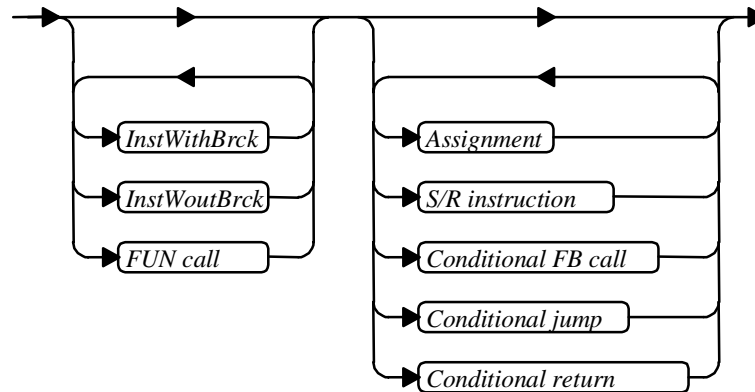
Label:
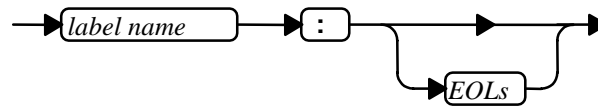
Figure G.1. Syntax diagrams of an IL instruction sequence with the sub-elements "conditional instruction" and "label".

An IL instruction sequence begins optionally with a (jump) label. This is followed either by a Load instruction followed by one or more conditional instructions CondInst, an unconditional instruction with FB call, a jump or a return (see syntax diagram in Figure G.1).

The conditional instruction begins with a sequence of instructions with and without brackets and/or function calls. This is followed by a series of (S/R) assignments, conditional calls or jumps.

The label consists of a label identifier, followed by a colon. It either immediately precedes the first instruction of the sequence or is followed by EOLs (end of line). The latter is an extension of IEC 61131-3, but it is accepted by some programming systems because it improves the optical structuring of instruction sequences.

EOLs represent the end of a line. An EOL can be directly preceded by *one* comment field.

The syntax diagrams for instructions, calls and jumps are given below.

InstWoutBrck:

Figure G.2. Syntax diagram of an instruction without brackets

An instruction without brackets (Figure G.2) consists of an IL operator with one operand and the end-of-line EOLs.

The syntax diagrams in Figure G.3 show the structure of an instruction with brackets. This type of instruction begins with an instruction consisting of an operator followed by an opening bracket and an operand. This can be followed by any number of instructions InstInBck inside the brackets, which are concluded with a closing bracket and EOLs.

These inner instructions can themselves contain brackets (nesting), as well as FUN calls and assignments.

As Figure G.4 shows, a function call consists of the function name together with a number of operands separated by commas as actual parameters of the function.

InstWithBrck:

InstInBck:
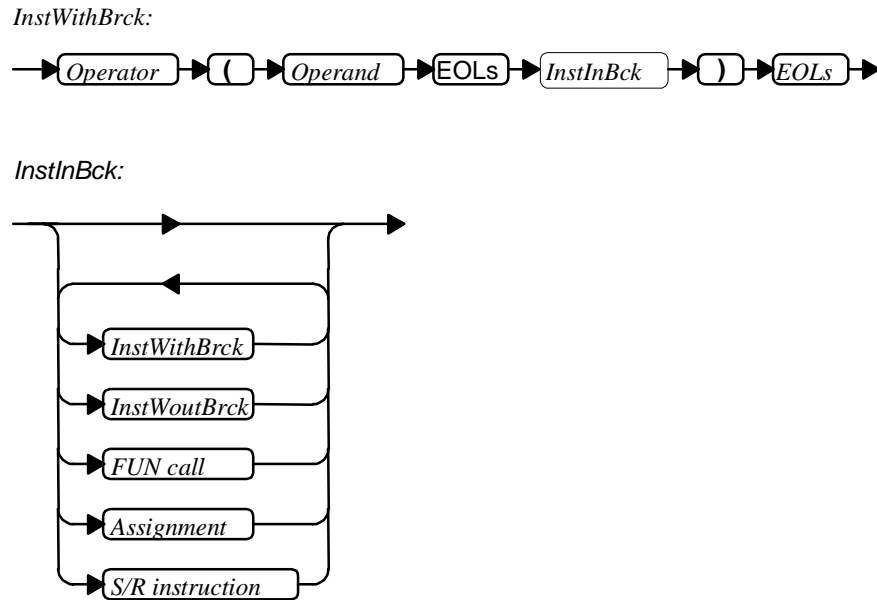
Figure G.3. Syntax diagrams of an instruction with brackets
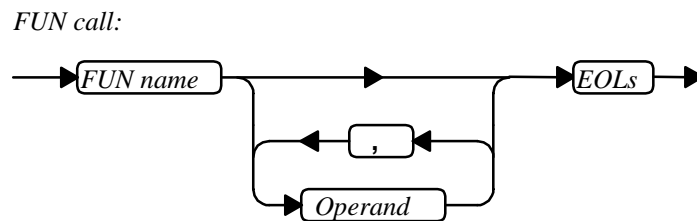
FUN call:

Figure G.4. Syntax diagram for a function call

Figure G.5 shows the syntax diagrams for conditional and unconditional calls of a function block. The unconditional call begins with CAL, the conditional call with CALC or CALCN. This is followed by the name of the FB instance, and the FB parameters in brackets.

The assignment of an actual parameter to a formal parameter is represented by the symbol ":=". Such assignments are required for every parameter and are separated by commas.

Unconditional FB call:
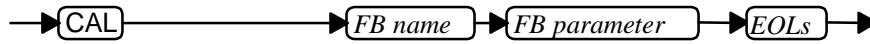
Conditional FB call:
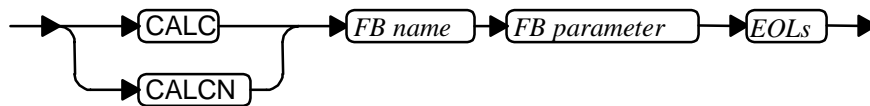
FB parameter:
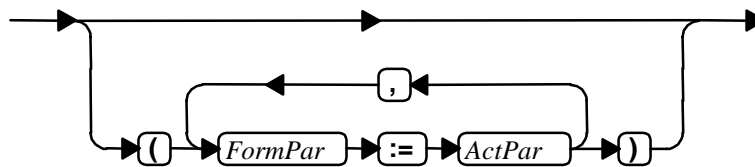
Figure G.5. Syntax diagrams for a function block call

Unconditional jump:

Conditional jump:
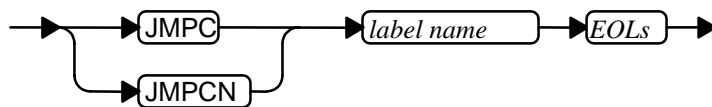
Figure G.6. Syntax diagrams for conditional and unconditional jumps

For jumps, the label name is specified after the jump operator JMP (unconditional) or JMPC/JMPCN (conditional) (Figure G.6).

Unconditional return:

```
─►─[RET]────────────────────────►─[EOLs]─►─
```
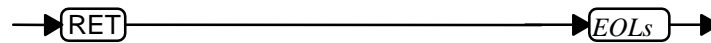
Conditional return:

```
─►─┬─►─[RETC]──┬──────────────►─[EOLs]─►─
   └─►─[RETCN]─┘
```
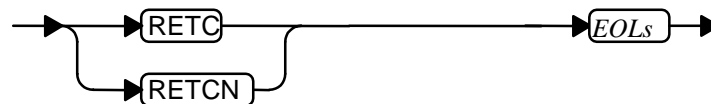
Figure G.7. Syntax diagrams for conditional and unconditional return

The returns shown in Figure G.7 have no operands or parameters, but they can have a comment, like any IL instruction.

Load instruction:

```
─►─┬─►─[LD]──┬──►─[Operand]─►─[EOLs]─►─
   └─►─[LDN]─┘
```
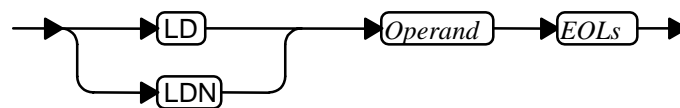
Figure G.8. Syntax diagram for the Load instruction

The Load instruction in Figure G.8 has a single (negatable) operand. It cannot be combined with a bracket or used inside a bracket.

Assignment:

```
─►─┬─►─[ST]──┬──►─[Operand]─►─[EOLs]─►─
   └─►─[STN]─┘
```
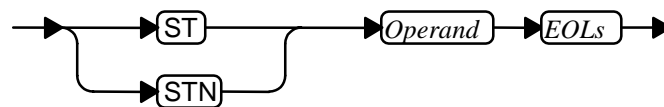
Figure G.9. Syntax diagram for assignment

Assignments (Figure G.9) consist of the operator ST or STN and the specification of the operand to be stored.

S/R instruction:
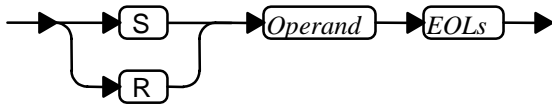
Figure G.10. Syntax diagram for the S/R instruction

An S/R instruction (Figure G.10) consists of the IL operators S or R and one operand.

Operator:
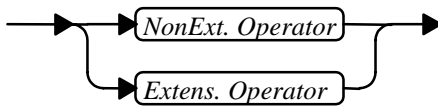
Figure G.11. Extensible and non-extensible operators

The operators represented in Figure G.11 perform logic operations, and are not used for loading or storage. A distinction is made between extensible and non-extensible operators, as shown below.

Extens. Operator:
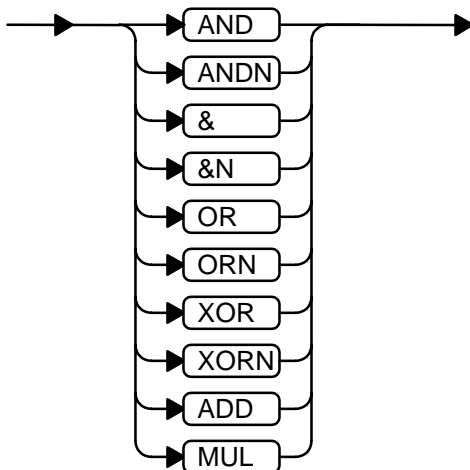
Figure G.12. Extensible operators: bitwise Boolean operations, addition and multiplication

Figure G.12 shows the extensible operators. They can have more than two input parameters. The bitwise Boolean operators (standard functions) can also be used with inversion.

NonExt. Operator:
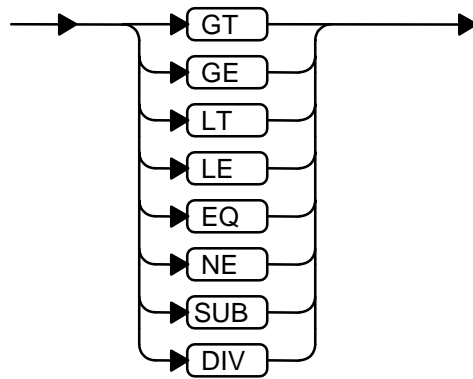
Figure G.13. Non-extensible operators: comparison, subtraction, division

The non-extensible operators in Figure G.13 have exactly two input parameters (including the current result CR).

EOLs:
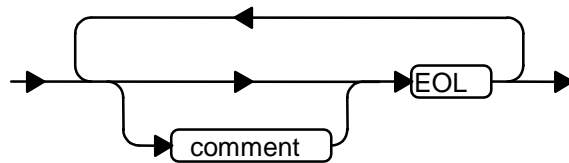
Figure G.14. Syntax diagram for the EOL (end of line) of an IL instruction with comment

An IL line is concluded with a single EOL character (e.g. carriage return / line feed) or a comment followed by EOL (Figure G.14). These elements can occur once or any number of times in sequence.

   A comment begins with (*, ends with *) and contains any number of alpha-numeric characters in between **without** EOL. Comments cannot be nested.

ActPar:

Operand:

label name:
FUN name:
FB name:
FormPar:

VarId:
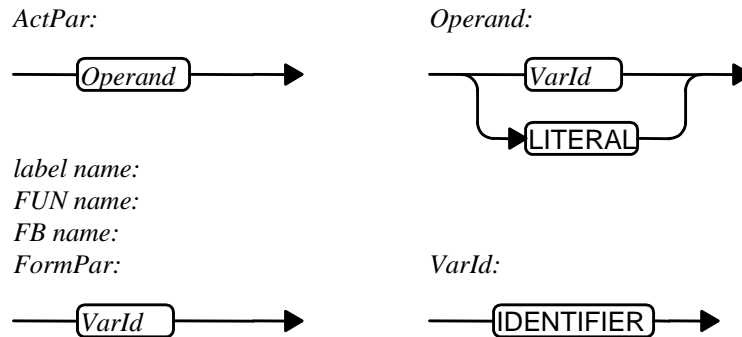
Figure G.15. Operands, parameters and other elements are represented by identifiers and literals.

Figure G.15 shows how parameters, operands and other elements are represented using IDENTIFIERs and LITERALs.

For simplicity the syntax diagrams of identifiers and literals are not shown here. The basic principles of their representation are explained in Section 3.2.

G.2 IL Example from Syntax Diagrams

The IL syntax diagrams shown on the previous pages will now be used to produce an IL example. This shows how sample programs are constructed from syntax diagrams and vice versa, enabling IL examples to be checked for correctness.

```
                              Beginning of IL sequence
0001        SequenceOne:      (* label *)
0002                          (* simple logic operation with jump *)
0003        LD      Var1      (* load instruction *)
0004                          (* beginning of conditional instructions *)
0005        ANDN  Var2        (* instruction without bracket *)
0006        ORN    ( Var3     (* instruction with bracket *)
0007        AND    Var4
0008        )                 (* end of bracketing *)
0009        AND    Var5
0010        ST      Var6      (* assignment *)
0011        S       Var7      (* S/R instruction *)
0012        RETC              (* conditional return *)
0013                          (* end of conditional instructions *)
0014                          (* end of IL sequence *)
```

Example G.1. IL example. The comments refer to the corresponding syntax diagram. The line numbers on the left are used for reference in Table G.1.

To show how the IL example in Example G.1 can be built up from the syntax diagrams in Section G.1, Table G.1 shows the relevant syntax diagrams for each IL line.

Line in Ex. G.1	Syntax diagram	Figures
0001-0014	IL instruction sequence	Figure G.1
0001-0002	(Jump) label	Figure G.1
0003-0004	Load instruction	Figure G.8
0005	Instruction without bracket	Figure G.2
0005	Extensible operator ANDN	Figure G.12
0006-0008	Instruction with bracket	Figure G.3
0006,0007	Extensible operators ORN, AND	Figure G.12
0007,0009	Instruction without bracket	Figure G.2
0007,0009	Extensible operator AND	Figure G.12
0010	Assignment	Figure G.9
0011	S/R instruction	Figure G.10
0012-0014	Conditional return	Figure G.7

Table G.1. Syntax diagrams to be used for each line of the IL sequence in Example G.1

This example shows how a concrete IL program is built up using syntax diagrams. In the syntax diagram for an IL instruction sequence (Figure G.1), first the label with name, colon and comments is inserted, followed by the first (Load) instruction.

The "conditional instructions" part is made up of two instructions, the first of which with a bracket containing further instructions. After the conditional instructions, the sequence is terminated with assignments and return.

In this way it is possible to create valid IL sequences from the individual syntax diagrams. Conversely, the relevant syntax diagrams for each IL line can be found in order to determine whether a program section is syntactically correct.

H Reserved Keywords and Delimiters

IEC 61131-3 expressly permits the use of translation tables for adapting keywords and delimiters to national character sets.

H.1 Reserved Keywords

Table H.1 lists all the reserved keywords for programming languages of IEC 61131-3 in alphabetical order. They must *not* be employed as names for user-defined elements.

A	ABS	ACOS
	ACTION	ADD
	AND	ANDN
	ANY	ANY_BIT
	ANY_DATE	ANY_INT
	ANY_NUM	ANY_REAL
	ARRAY	ASIN
	AT	ATAN
B	BOOL	BY
	BYTE	

Table H.1. Reserved keywords of IEC 61131-3 (continued on next page)

C	CAL	CALC
	CALCN	CASE
	CD	CDT
	CLK	CONCAT
	CONFIGURATION	CONSTANT
	COS	CTD
	CTU	CTUD
	CU	CV
D	D	DATE
	DATE_AND_TIME	DELETE
	DINT	DIV
	DO	DS
	DT	DWORD
E	ELSE	ELSIF
	END_ACTION	END_CASE
	END_CONFIGURATION	END_FOR
	END_FUNCTION	END_FUNCTION_BLOCK
	END_IF	END_PROGRAM
	END_REPEAT	END_RESOURCE
	END_STEP	END_STRUCT
	END_TRANSITION	END_TYPE
	END_VAR	END_WHILE
	EN	ENO
	EQ	ET
	EXIT	EXP
	EXPT	
F	FALSE	F_EDGE
	F_TRIG	FIND
	FOR	FROM
	FUNCTION	FUNCTION_BLOCK
G	GE	GT
I	IF	IN
	INITIAL_STEP	INSERT
	INT	INTERVAL

Table H.1. (Continued on next page)

J	JMP	JMPC
	JMPCN	
L	L	LD
	LDN	LE
	LEFT	LEN
	LIMIT	LINT
	LN	LOG
	LREAL	LT
	LWORD	
M	MAX	MID
	MIN	MOD
	MOVE	MUL
	MUX	
N	N	NE
	NEG	NOT
O	OF	ON
	OR	ORN
P	P	PRIORITY
	PROGRAM	PT
	PV	
Q	Q	Q1
	QU	QD
R	R	R1
	R_TRIG	READ_ONLY
	READ_WRITE	REAL
	RELEASE	REPEAT
	REPLACE	RESOURCE
	RET	RETAIN
	RETC	RETCN
	RETURN	RIGHT
	ROL	ROR
	RS	RTC
	R_EDGE	

Table H.1. (Continued on next page)

S	S	S1
	SD	SEL
	SEMA	SHL
	SHR	SIN
	SINGLE	SINT
	SL	SQRT
	SR	ST
	STEP	STN
	STRING	STRUCT
	SUB	
T	TAN	TASK
	THEN	TIME
	TIME_OF_DAY	TO
	TOD	TOF
	TON	TP
	TRANSITION	TRUE
	TYPE	
U	UDINT	UINT
	ULINT	UNTIL
	USINT	VAR
V	VAR_ACCESS	VAR_EXTERNAL
	VAR_GLOBAL	VAR_INPUT
	VAR_IN_OUT	VAR_OUTPUT
W	WHILE	WITH
	WORD	
X	XOR	XORN

Table H.1. (Continued)

H.2 Delimiters

Delimiters are "symbols" in the syntax of programming languages and have different meanings depending on where they are used. For example, round brackets can be used to indicate the beginning and end of a list of actual parameters in a function call, or they can be used together with the asterisk to frame comments.

All the delimiters and their combinations are listed in Table H.2 together with their possible meanings.

Characters for the graphical representation of lines are not included here.

Delimiter		Meaning, explanations
Space		Can be inserted anywhere - except within keywords, literals, identifiers, directly represented variables or combinations of delimiters (such as "(*" or "*)"). IEC 61131-3 does not specify any rules about tabulators (TABs). They are usually treated as spaces.
End of line (EOL)		Permissible at the end of a line in IL. In ST also permissible within statements. Not permitted in IL comments. EOL (end of line) is normally implemented by CR&LF (Carriage Return & Line Feed).
Begin comment	(*	Beginning of a comment (nesting not allowed)
End comment	*)	End of a comment
Plus	+	1. Leading sign of a decimal literal, also in the exponent of a real (floating-point) literal 2. Addition operator in expressions
Minus	-	1. Leading sign of a decimal literal, also in the exponent of a real (floating-point) literal 2. Subtraction operator in expressions 3. Negation operator in expressions 4. Year-month-day separator in time literals
Number sign ("hash")	#	1. Based number separator in literals 2. Time literal separator

Table H.2. Delimiters of IEC 61131-3 (continued on next page)

Delimiter	Meaning, explanations
Point .	1. Integer/fraction separator 2. Separator in the hierarchical addresses of directly represented and symbolic variables 3. Separator between components of a data structure (for access) 4. Separator for components of an FB instance (for access)
e, E	Leading character for exponents of real (floating-point) literals
Quotation mark '	Beginning and end of character strings
Dollar sign $	Beginning of special characters within character strings
Prefix time literals **t#, T#** **d#, D#** **d, D** **h, H** **m, M** **s, S** **ms, MS** **date#, DATE#** **time#, TIME#** **time_of_day#** **TIME_OF_DAY#** **tod#, TOD#** **date_and_time#** **DATE_AND_TIME#** **dt#, DT#**	Characters introducing time literals. Combinations of lower-case and upper-case letters are also permissible..
Colon :	Separator for: 1. Time within time literals 2. Data type specification in variable declarations 3. Data type name specification 4. Step names 5. PROGRAM...WITH... 6. Function name/data type 7. Access path: Name/type 8. Jump label before next statement 9. Network label before next statement
Assignment :=	1. Operator for initial value assignment 2. Input connection operator (assignment of actual parameter to formal parameter in POU-call) 3. Assignment operator

Table H.2. (Continued on next page)

Delimiter	Meaning, explanations
Round brackets (...)	Beginning and end of: 1. Enumeration list 2. Initial value list, also: multiple initial values (with repetition number) 3. Range specification 4. Operator in IL (computation level) 5. Parameter list in POU call 6. Sub-expression hierarchy
Square brackets [...]	Beginning and end of: 1. Array subscript (access to an array element) 2. Character string length (in declaration)
Comma ,	Separator for: 1. Enumeration list 2. Initial value list 3. Array subscripts (multidimensional) 4. Variable names (in the case of multiple declarations with the same data type) 5. Parameter list in POU call 6. Operand list in IL 7. CASE value list
Semicolon ;	End of: 1. Definition of a (data) type 2. Declaration (e.g. variables) 3. ST statement
Two points ..	Separator for: 1. Range specification 2. CASE range
Percent %	Leading character for hierarchical addresses of directly represented and symbolic variable
Assignment2 =>	Output connection operator (assignment of formal parameters to actual parameters in a PROGRAM call)
Comparison >, < >=, <=, =, <>,	Relational operators in expressions
Exponent **	Operator in expressions
Multiplication *	Multiplication operator in expressions
Division /	Division operator in expressions
Ampersand &	AND operator in expressions

Table H.2. (Continued)

I Planned Amendments to the Standard

This edition incorporates all the corrections contained in the current Corrigendum. As mentioned at the beginning, this is a document which corrects errors and clarifies points in the standard.

   The Amendments, on the other hand, describe extensions and significant changes to the standard. As these have not yet been finalised at the time of writing (2000), only the most important points are summarised here.

- Introduction of *typed literals* for Boolean and numerical constants. A constant can be identified by a prefix <elementary data type>#<data value>. Example: SINT#20 or BOOL#0
- Use of the *ISO 10646 character code* (1 / 2 bytes long) for character strings. This also enables national characters like ä, ü, ö,… to be represented.
- Introduction of the variable type *VAR_TEMP … END_VAR*. Variables declared with this construct are initialised every time an FB is called and do not retain their value between calls (corresponds to the VAR declaration for functions).
- Sets of variables are given the attributes RETAIN or *NON_RETAIN*. If neither of these attributes is specified, the "implementation-dependent parameters" decide whether these variables are to be retentive or not.
- Physical addresses can be partially specified at the programming stage by using the wild card character "*", for example: *%Q**. The final address is specified at the configuration stage.
- The initial values of call parameters and local variables for a function block instance can be defined when the instance is declared. This enables different instances of the same block type to have different initial values.
- Actual parameters are assigned to formal parameters with ":=" as before. Output parameter values are assigned with "=>".
  Example:
  CAL fb_instance (input_parameter:= 2, return_value => call_variable).

- Introduction of the generic data type *ANY_MAGNITUDE* to include the data types ANY_NUM and TIME. *ANY_STRING* describes the two types STRING (character length 1 byte) and *WSTRING* (character length 2 bytes). ANY is subdivided into *ANY_DERIVED* and *ANY_ELEMENTARY*.

J Glossary

In this chapter important terms and abbreviations employed in the book are listed in alphabetical order and explained in detail.

Terms, which are defined by IEC 61131 (parts 1 and 3), are marked "IEC".

Action	IEC	Boolean variable or a series of statements which can be accessed via an *action block* (in *SFC*).
Action block	IEC	Activation description of *actions* (in *SFC)* using an associated control structure.
Action control	IEC	Control unit for every *action* in SFC which is supplied with the input condition for activating the assigned action by means of one or more *action blocks*; also: action control function block
Actual parameter		Actual value for an input variable (*formal parameter*) of a *POU;* also: current address
Allocation table		List which contains the assignment of all symbols or *symbolic variables* to *PLC* addresses.
Array		Sequence of elements of the same *data type.*
Block		(Sub-) programming unit, from which PLC-programs are built. Blocks can often be loaded independently from each other into the PLC, see also *POU*.
CIM		Abbreviation for Computer Integrated Manufacturing
Cold restart	IEC	Restart of the PLC-system and its application program, whereby all variables and memory areas (such as internal registers, timers, counters) are (newly) initialised with predefined values. This process can occur automatically after specific events (e.g. after a power failure) or also manually by the user (e.g. Reset button). Also: new start.

Comment	IEC	Text written between parentheses and asterisks used to explain the program (cannot be nested!). This is not interpreted by the *programming system.*
Configuration	IEC	Language element CONFIGURATION which corresponds to a *PLC system*
CPU		Abbreviation for Central Processing Unit (e.g. of a PLC)
CR		Abbreviation for *Current Result*
Cross-compilation		Conversion of the representation of a *POU* from one programming language to another, typically between ladder and function block diagram, but also between textual and graphical languages; also: cross-compiling
Current result	IEC	Interim result in *IL* of any *data type*
Cycle		A single run of the (periodically called) application program.
Cycle time		The time which an application program requires for one *cycle*; also: scan time.
Data block		Shared data area, which is accessible throughout a program, see also *block*. In IEC 61131-3, there is no direct analogy. They are replaced here by global (non-local), structured data areas and *FB instance* data areas.
Data type		Defines bit length and range properties of a *variable.*
Declaration		Definition of *variables* and *FB instances* takes place in a *declaration block* with information about the data name, the *data type* or the *FB type* as well as appropriate *initial values*, *range specification* and *array attributes* (data template declaration).
		The definition or programming of *POUs* is also designated as a *declaration* since their interfaces are made known to the *programming system* here.
Declaration block	IEC	Combination of *declarations* of one variable type at the beginning of the *POU*.
Derived data type	IEC	With the aid of a *type definition,* a user-specific *data type* is created. Its elements are *elementary data types* or *derived data types*.
Directly represented variable	IEC	*Variable* without further name which corresponds to a *hierarchical address.*
Edge		The $0 \rightarrow 1$ transition of a Boolean variable is known as "rising edge". Accordingly, the $1 \rightarrow 0$ transition is known as "falling" edge.
Elementary data type	IEC	A standard *data type* predefined by IEC 61131-3.
Enumeration		Special *data type* for the definition of integer values.
Extension of functions	IEC	A *function* can have a variable number of inputs.

FB	IEC	Abbreviation for *function block*
FB instance	IEC	see *instance*
FB type	IEC	Name of a *function block* with call and return interface
FBD	IEC	Abbreviation for *Function Block Diagram*
Formal parameter		Name or placeholder of an input variable (all *POUs*) or output variable (*function block* and *program*).
Function	IEC	A *POU* of type FUNCTION
Function block	IEC	A *POU* of type FUNCTION_BLOCK
Function Block Diagram	IEC	Function Block Diagram (FBD) is a programming language used to describe networks with Boolean, arithmetic and similar elements.
Generic data type	IEC	Combination of *elementary data types* into groups using the prefix 'ANY', in order to describe *overloaded functions*.
Hierarchical address	IEC	Physical slot address of I/O modules of a *PLC system* (see also *I/O*).
Hot restart	IEC	Program restart at the place in the program where a power failure occurred. All battery-backed data areas as well as the application program context will be restored and the program can go on running, as if there had been no power failure. In contrast to *warm restart,* the interruption duration must be within a given value range depending on the process. For this purpose, the PLC system must have a separately secured real-time clock in order to be able to compute the interruption duration.
I/O		The addresses of input and output modules belonging to a *PLC system* with *hierarchical addresses*.
IL	IEC	Abbreviation for *Instruction List*
Indirect FB call		Call of an *FB instance* whose name is passed to the *POU* as a VAR_IN_OUT parameter.
Initial value		Value of a *variable*, which will be assigned during initialisation, i.e. at system start-up time; also: starting count.
Instance	IEC	Structured data set of an *FB* obtained by *declaration* of a *function block* indicating the *FB type*.
Instruction List	IEC	Instruction List (IL)) is a much used Assembler-like programming language for PLC-systems. Sometimes it is also called Statement List Language (STL).
Ladder Diagram	IEC	Ladder Diagram (LD) is a programming language to describe networks with Boolean and electromechanical elements, such as contacts and coils, working together concurrently.

LD	IEC	Abbreviation for *Ladder Diagram*
Multi-element variable	IEC	*Variable* of type *array* or *structure*, which is put together from several different *data types.*
New start		see *Cold restart*
Overloading of functions	IEC	The capability of an operation or *function* to operate with different input *data types* (but each of the same type). By this means several function classes are available under the same name.
PC		Abbreviation for personal computer.
		Also Abbreviation for Programmable Controllers as employed in IEC 61131
PLC		Abbreviation for Programmable Logic Controller
PLC programming computer		Unit consisting of computer, *programming system* and other peripherals for programming the *PLC*.
PLC programming system		Set of programs which are necessary for programming a *PLC system:* program creation and compilation, transfer into the *PLC* as well as program test and commissioning functions.
PLC system		All hardware parts required for executing a PLC program.
POU	IEC	Abbreviation for *program organisation unit*
Program	IEC	A *POU* of type PROGRAM
Program organisation unit	IEC	A *block* of type *function*, *function block* or *program*, from which application programs are built.
Programming computer		see *PLC programming computer*
Programming system		see *PLC programming system*
Range specification		Specification of a permissible range of values for a *data type* or a *variable*.
Recursion		Illegal in IEC 61131-3. It means:
		a) the *declaration* of an *FB* using its own name or *FB type*,
		b) mutual *FB* calls.
		Recursion is considered an error and must be recognised while programming and/or at run time.
Resource	IEC	Language element RESOURCE which corresponds to a central processing unit of the *PLC system.*
Retentive data	IEC	Ability of a PLC to protect specific process data against loss during a power failure. The keyword RETAIN is used for this in IEC 61131-3.
Reverse compiling		Recovery of the *POU* source back from the *PLC* memory.
Run-time program		Program of *POU type* PROGRAM as an executable unit (by association with a *task*).

Semantics		Meaning of language elements of a programming language as well as their description and their application.
Sequential Function Chart	IEC	Sequential Function Chart (SFC) is a programming language used to describe sequential and parallel control sequences with time and event control.
SFC	IEC	Abbreviation for *Sequential Function Chart*
Single-element variable	IEC	*Variable* which is based on a single *data type.*
ST	IEC	Abbreviation for *Structured Text*
Standard function blocks	IEC	Set of *function blocks* predefined by IEC 61131-3 for implementation of typical PLC requirements.
Standard functions	IEC	Set of *functions* predefined by IEC 61131-3 for the implementation of typical PLC requirements.
Std. FB		Abbreviation for *standard function block*
Std. FUN		Abbreviation for *standard function*
Step	IEC	State element of an SFC program in which statements of the *action* corresponding to this *step* can be started.
Structured Text	IEC	Structured Text is a programming language used to describe algorithms and control tasks by means of a modern high programming language similar to PASCAL.
Symbolic variable	IEC	*Variable* with name (identifier) to which a *hierarchical address* is assigned.
Syntax		Structure and interaction of elements of a programming language.
Task	IEC	Definition of run-time properties of programs.
Transition	IEC	Element of an SFC program for movement from one SFC *step* to the next by evaluating the transition condition.
Type definition		Definition of a user-specific *data type* based on already available *data types.*
Variable		Name of a data memory area which accepts values defined by the corresponding *data type* and by information in the variable *declaration.*
Warm reboot		Term used for either *hot restart* or *warm restart.*

| **Warm restart (at the beginning of the program)** | IEC | Program restart similar to *hot restart* with the difference that the program starts again at the beginning, if the interruption duration exceeded the maximum time period allowed. The user program can recognise this situation by means of a corresponding status flag and can accordingly pre-set specific data. |

K Bibliography

Books concerning PLC programming:

[JohnTiegel-99]   K.-H. John and M.Tiegelkamp
„SPS-Programmierung mit IEC 61131-3", Springer Berlin
Heidelberg New York, 1999, 3rd Ed. (German),
ISBN 3-540-66445-9

[Lewis-98]   R. W. Lewis
„Programming industrial control systems using
IEC 1131-3", IEE Control Engineering, The Institution of
Electrical Engineers, 1998,
ISBN 0-852-96950-3

[Bonfatti]   Dr. Monari, Prof. Bonfatti and Dr. Sampieri
„IEC 1131-3 Programming Methodology; Software
engineering methods for industrial automated systems",
ISBN 2-9511585-0-5

Standards concerning PLC programming:

[IEC EN 61131-3 B1] Committee Draft - IEC 61131-3, 2nd Ed.
„Programmable controllers - programming languages",
IEC 65B/WG7/TF3(PT3E2ACD)1
Committee Draft, Houston, 10/1998

[IEC AMEND-98]   IEC SC65B/WG7/TF3, Proposals to IEC 1131-3:
„Draft Amendments to IEC 1131-3",
Draft Version, Venedig, Italy, 8/98

[IEC CORR-97]   IEC SC65B/WG7/TF3, correction of IEC 1131-3:
„Revised Technical Corrigendum to IEC 1131-3",
Draft Version, Yokohama, Japan, 5/97

[IEC TR2-97]   IEC SC65B/WG7/TF3, Type 2 Technical Report
                       „Proposed Extensions to IEC 1131-3",
                       Committee Draft, Paris, France, 9/96
                       Version: 05/1997

[IEC 61499-98]   IEC TC65/WG6(PT1CD+PT2CD)
                       „Function blocks for industrial-process measurement and
                       control systems",
                       Committee Draft, Parts 1+2, 1998

Papers concerning IEC 61131-3

[Frost & Sullivan-95] "World programmable logic controller markets: increasing
                       functionality promises continued growth"
                       Study on the PLC market 1993-2000
                       Frost & Sullivan

[OMAC-94]   „Requirements of Open, Modular Architecture Controllers
                       for Applications in the Automotive Industry"
                       Version 1.1
                       13.12.94; Chrysler; Ford; G M

[PLCopen-96]   PLCopen / Michael Babb
                       „IEC 1131-3: A standard programming resource for PLCs"
                       in: Control Engineering, 2/96

[PLCopen-00]   „PLCopening" News of PLCopen, quarterly newspaper
                       Years 1992-00
                       PLCopen, Zaltbommel, The Netherlands

[Wal-99]   Eelco van der Wal
                       " IEC 1131-3: a standard programming resource"
                       ISA Show, 06.07.1999

Important references

[PLCopen Europe]   Eelco van der Wal
Postbus 2015
5300 CA Zaltbommel, The Netherlands

Tel: +31-418-541139
Fax: +31-418-516336
Email: evdwal@plcopen.org
www.plcopen.org

[PLCopenNorth America]
Jeremy Pollard
8 Vine Crescent,
Barrie, Ontario L4N 2B3
CANADA

Tel: 705-739-7155
Fax: 705-739-7157
Email: plcopenna@tsuonline.com

[PLCopenNorth Japan]
Yoshio Yamaguchi
3F, 3-61-8 Wada
Suginami - Ku
Tokyo 166-0012
Japan

Tel: +81-3-3315-0194
Fax :+81-3-3315-0192
Email: plcopen-japan@mugen.com

L Index