Common Elements (version 1996)

1.4.1 Software model

The basic high-level language elements and their interrelationships are illustrated in figure 1. These consist of elements which are *programmed* using the languages defined in this part, that is, *programs* and *function blocks*; and *configuration elements*, namely, *configurations, resources, tasks, global variables,* and *access paths*, which support the installation of programmable controller *programs* into programmable controller systems.

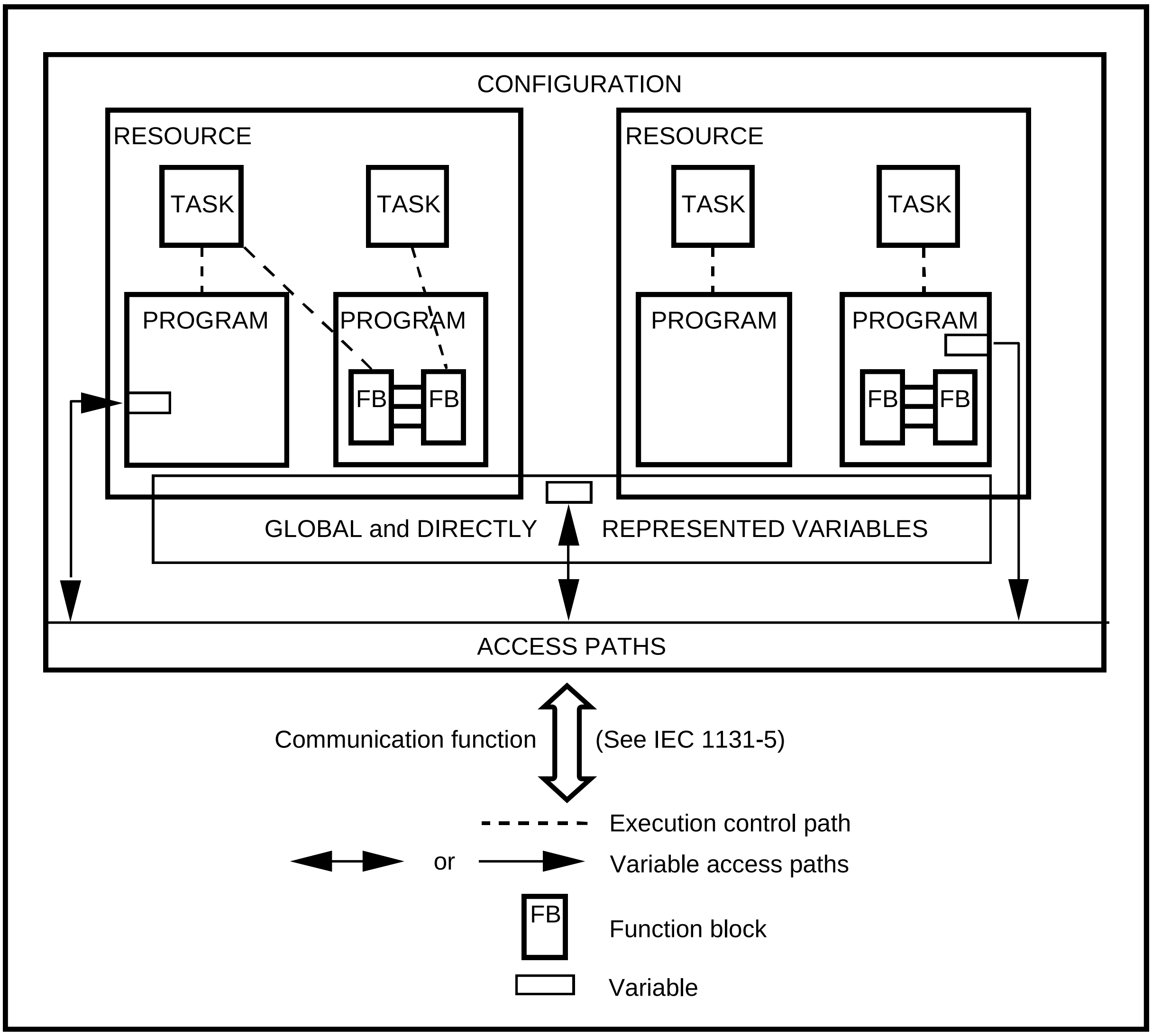


Figure 1 - Software model

A *configuration* is the language element which corresponds to a *programmable controller system* as defined in IEC 1131-1. A *resource* corresponds to a "signal processing function" and its "man-machine interface" and "sensor and actuator interface" functions (if any) as defined in IEC 1131-1. A *configuration* contains one or more *resources*, each of which contains one or more *programs* executed under the control of zero or more *tasks*. A *program* may contain zero or more *function blocks* or other language elements as defined in the standard.

Configurations and *resources* can be started and stopped via the "operator interface", "programming, testing, and monitoring", or "operating system" functions defined in IEC 1131-1. The starting of a *configuration* shall cause the initialization of its *global variables*, followed by the starting of all the *resources* in the configuration. The starting of a *resource* shall cause the initialization of all the *variables* in the resource, followed by the enabling of all the *tasks* in the resource. The stopping of a *resource* shall cause the disabling of all its *tasks*, while the stopping of a *configuration* shall cause the stopping of all its *resources*.

Programs, resources, global variables, access paths (and their corresponding access privileges, and *configurations* can be loaded or deleted by the "communication function" defined in IEC 1131-1. The loading or deletion of a *configuration* or *resource* shall be equivalent to the loading or deletion of all the elements it contains.

The mapping of the language elements defined in this subclause on to communication objects is defined in IEC 1131-5.

1.4.2 Communication model

Figure 2 illustrates the ways that values of variables can be communicated among software elements.

As shown in figure 2a, variable values within a program can be communicated directly by connection of the output of one program element to the input of another. This connection is shown explicitly in graphical languages and implicitly in textual languages.

Variable values can be communicated between programs in the same configuration via *global variables* such as the variable *x* illustrated in figure 2b. These variables shall be declared as GLOBAL in the configuration, and as EXTERNAL in the programs, as specified in 2.4.3.

As illustrated in figure 2c, the values of variables can be communicated between different parts of a program, between programs in the same or different configurations, or between a programmable controller program and a non-programmable controller system, using the communication function blocks defined in IEC 1131-5. In addition, programmable controllers or non-programmable controller systems can transfer data which is made available by *access paths,* as illustrated in figure 2d, using the mechanisms defined in IEC 1131-5.

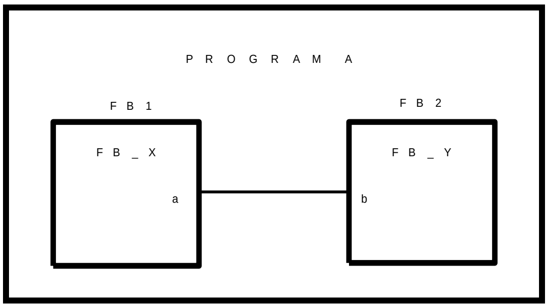


Figure 2a - Data flow connection within a program

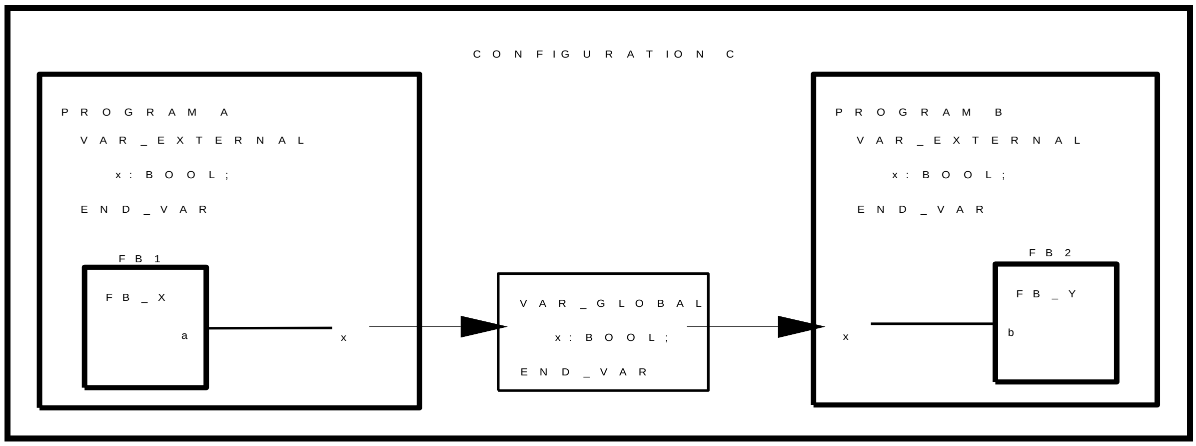


Figure 2b - Communication via GLOBAL variables

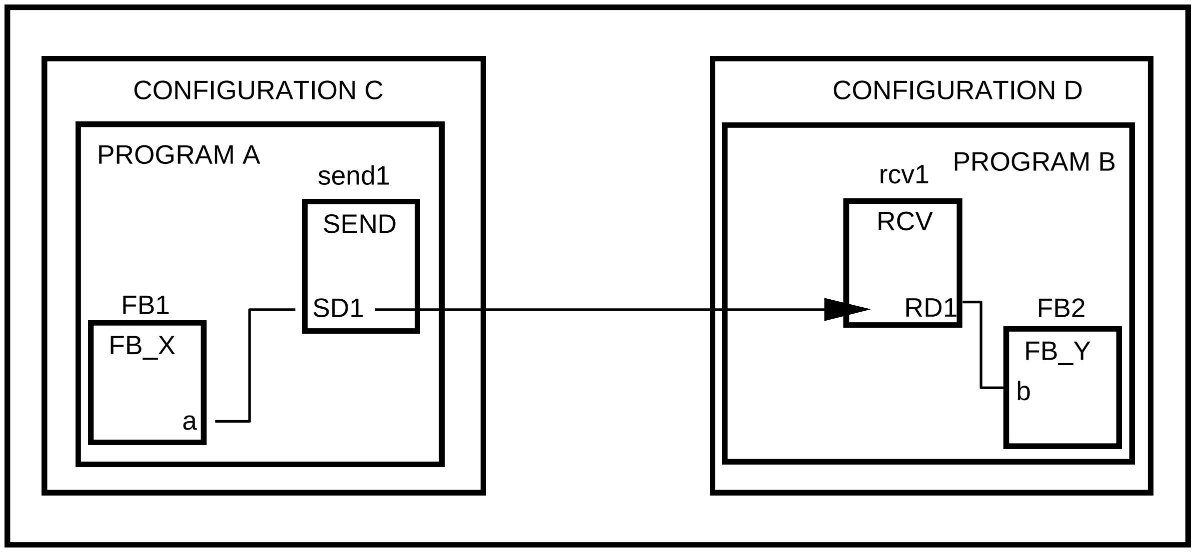


Figure 2c - Communication function blocks

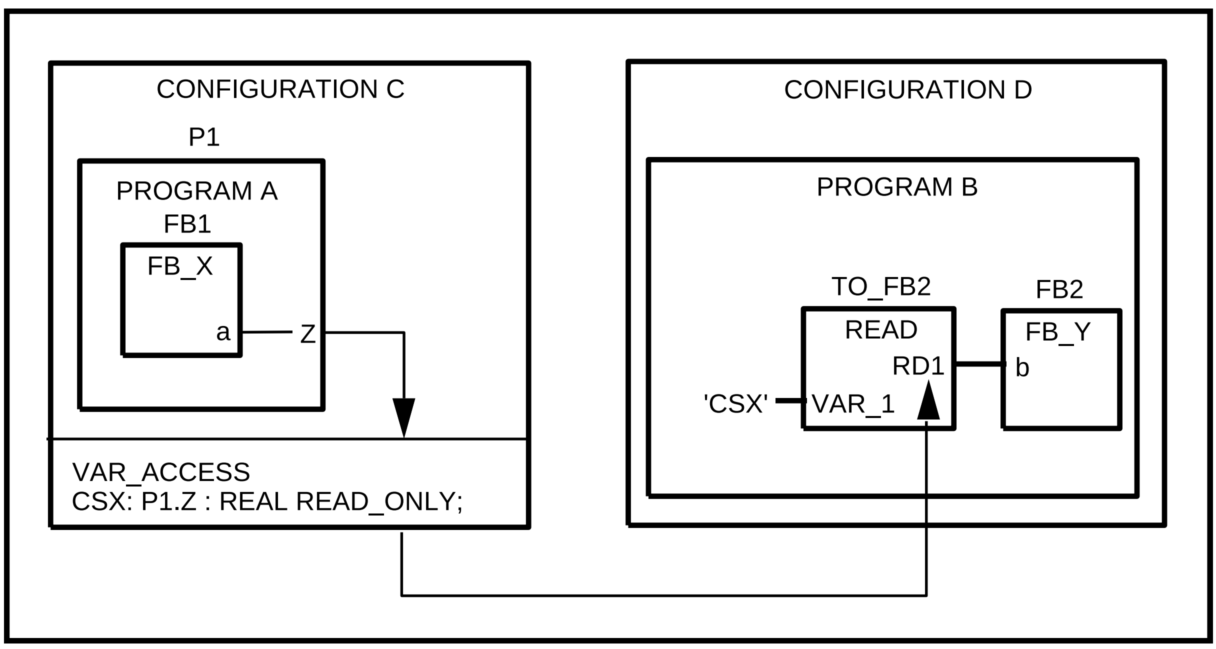


Figure 2d - Communication via access paths

NOTE: This figure is illustrative only. The graphical representation is not normative. The details of the communication function blocks are not shown in this figure. See the standard itself and IEC 1131-5.

Figure 2 - Communication model

1.4.3 Programming model

The elements of programmable controller programming languages, and the subclauses in which they appear in this part, are classified as follows:

- Data types
- Program organization units
  - Functions
  - Function blocks
  - Programs
- Sequential Function Chart (SFC) elements
- Configuration elements
  - Global variables
  - Resources
  - Tasks
  - Access paths

As shown in figure 3, the combination of these elements shall obey the following rules:

1) Derived *data types* shall be declared, using the standard data types and any previously derived data types.

2) Derived *functions* can be declared, using standard or derived data types, the standard functions, and any previously derived functions. This declaration shall use the mechanisms defined for the IL, ST, LD or FBD language.

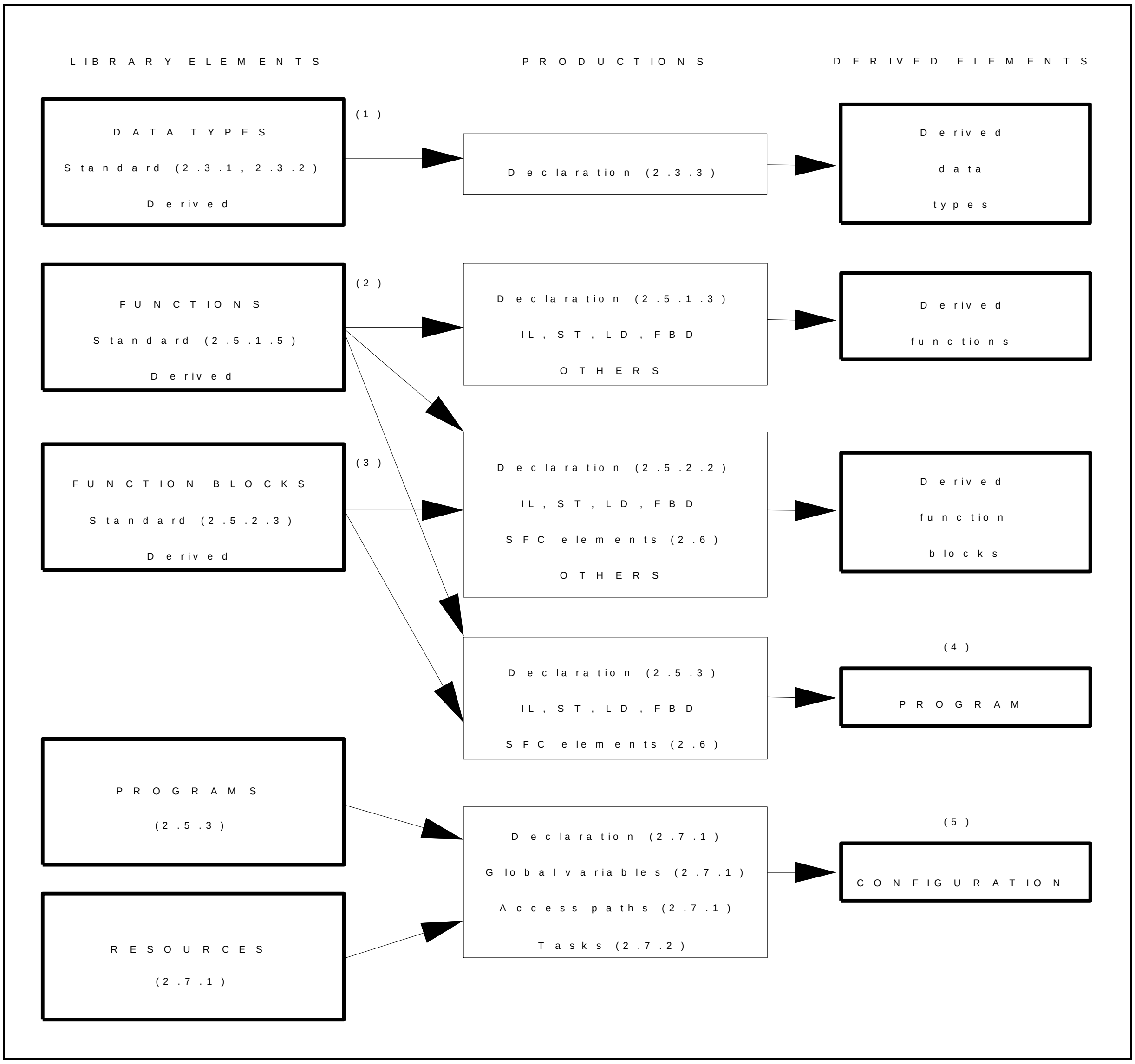
3) Derived *function blocks* can be declared, using standard or derived data types and functions, the standard function blocks, and any previously derived function blocks. This declaration shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements.

4) A *program* shall be declared, using standard or derived data types, functions, and function blocks. This declaration shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements.

5) *Programs* can be combined into *configurations* using the elements, that is, *global variables, resources, tasks,* and *access paths*.

Reference to "previously derived" data types, functions, and function blocks in the above rules is intended to imply that once such a derived element has been declared, its definition is available, e.g., in a "library" of derived elements, for use in further derivations. Therefore, the declaration of a derived element type shall not be contained within the declaration of another derived element type.

A programming language other than one of those defined in this standard may be used in the declaration of a *function* or *function block*. The means by which a user program written in one of the languages defined in this standard invokes the execution of, and accesses the data associated with, such a derived function or function block shall be as defined in this standard.



NOTE - For the references please refer to the standard itself.

Figure 3 - Combination of programmable controller language elements
(LD - Ladder Diagram, FBD - Function Block Diagram, IL - Instruction List, ST - Structured Text, OTHERS - Other programming languages)

1.5 Compliance

See IEC 1131-3 standard for details.

1.5.2 Programs

A programmable controller program complying with the requirements of IEC 1131-3:

a) shall use only those features specified in this part for the particular language used;

b) shall not use any features identified as extensions to the language;

c) shall not rely on any particular interpretation of implementation-dependent features.

The results produced by a complying program shall be the same when processed by any complying system which supports the features used by the program, except as these results are influenced by program execution timing, the use of implementation-dependent features in the program, and the execution of error handling procedures.

2. Common elements

This clause defines textual and graphic elements which are common to all the programmable controller programming languages specified in IEC 1131.

2.1 Use of printed characters

2.1.1 Character set

Textual languages and textual elements of graphic languages shall be represented in terms of the "Basic code table" of the ISO/IEC 646 character set.

The encoding of characters from national or extended (8-bit) character sets shall be consistent with ISO/IEC 646.

The *required character set* shown as feature 1 in table 1 consists of all the characters in columns 3 to 7 of the "Basic code table" given as table 1 in ISO/IEC 646, except for lower-case letters and those character positions which are reserved or optionally available for use in national character sets.

NOTE - The use of characters from national character sets is a typical extension of this standard.

Table 1 - Character set features

No.	Description
1	Required character set
2	Lower case characters
3a 3b	Number sign (#) OR Pound sign (£)
4a 4b	Dollar sign ($) OR Currency sign
5a 5b	Vertical bar (|) OR Exclamation mark (!)
 6a 6b	Subscript delimiters: Left and right brackets "[ ]" OR Left and right parentheses "( )"
	NOTE - When lower-case letters are supported, the case of letters shall not be significant in language elements (except within terminal symbols as defined in annexes A and B, comments, string literals, and variables of type STRING), e.g., the identifiers "abcd", "ABCD", and "aBCd" shall be interpreted identically.

2.1.2 Identifiers

An *identifier* is a string of letters, digits, and underline characters which shall begin with a letter or underline character.

Underlines shall be significant in identifiers, e.g., "A_BCD" and "AB_CD" shall be interpreted as different identifiers. Multiple leading or multiple embedded underlines are not allowed.

Identifiers shall not contain imbedded space (SP) characters.

At least six characters of uniqueness shall be supported in all systems which support the use of identifiers, e.g., "ABCDE1" shall be interpreted as different from "ABCDE2" in all such systems.

Identifier features and examples are shown in table 2.

Table 2 - Identifier features

No.	Feature description	Examples
1	Upper case and numbers	IW215 IW215Z QX75 IDENT
2	Upper and lower case, numbers, embedded underlines	All the above plus: LIM_SW_5 LimSw5 abcd ab_Cd
3	Upper and lower case, numbers, leading or embedded underlines	All the above plus: _MAIN _12V7

2.1.3 Keywords

Keywords are unique combinations of characters utilized as individual syntactic elements as defined in annex B. All keywords used in this part are listed in annex C. Keywords shall not contain imbedded spaces. The keywords listed in annex C shall not be used for any other purpose, e.g., variable names or extensions.

NOTE - National standards organizations can publish tables of translations of the keywords given in annex C.

2.1.4 Use of spaces

The user shall be allowed to insert one or more spaces (code position 2/0 in the ISO/IEC 646 character set) anywhere in the text of programmable controller programs except within keywords, literals, identifiers, directly represented variables, or delimiter combinations (e.g., for comments as defined below.

2.1.5 Comments

User comments shall be delimited at the beginning and end by the special character combinations "(*" and "*)", respectively, as shown in table 3. Except in the IL language, comments shall be permitted anywhere in the program where spaces are allowed, except within character string literals. Comments shall have no syntactic or semantic significance in any of the languages defined in this part.

Nested comments are not allowed, e.g., (* (* NESTED *) *).

Table 3 - Comment feature

No.	Feature description	Examples
1	Comments	(******************************) (* A framed comment *) (******************************)

2.2 External representation of data

External representations of data in the various programmable controller programming languages shall consist of numeric literals, character strings, and time literals. (Note: see the standard for details)

2.2.1 Numeric literals

Numeric literal features and examples are shown in table 4.

Table 4 - Numeric literals

No.	Feature description	Examples
1	Integer literals	-12 0 123_456 +986
2	Real literals	-12.0 0.0 0.4560 3.14159_26

3	Real literals with exponents	-1.34E-12 or -1.34e-12 1.0E+6 or 1.0e+6 1.234E6 or 1.234e6
4	Base 2 literals	2#1111_1111 (255 decimal) 2#1110_0000 (240 decimal)
5	Base 8 literals	8#377 (255 decimal) 8#340 (240 decimal)
6	Base 16 literals	16#FF or 16#ff (255 decimal) 16#E0 or 16#e0 (240 decimal)
7	Boolean zero and one	0 1
8	Boolean FALSE and TRUE	FALSE TRUE
	NOTE - The keywords FALSE and TRUE correspond to Boolean values of 0 and 1, respectively.	

2.2.2 Character string literals

Table 5 - Character string literal feature

No.	Example	Explanation
1	''	Empty string (length zero)
	'A'	String of length one containing the single character A
	' '	String of length one containing the "space" character
	'$''	String of length one containing the "single quote" character
	'RL' '$0D$0A'	Strings of length two containing CR and LF characters
	'$$1.00'	String of length five which would print as "$1.00"

Table 6 - Two-character combinations in character strings

No.	Combination	Interpretation when printed
2	$$	Dollar sign
3	$'	Single quote
4	$L or $l	Line feed
5	$N or $n	Newline
6	$P or $p	Form feed (page)
7	$R or $r	Carriage return
8	$T or $t	Tab
	NOTE - The "newline" character provides an implementation-independent means of defining the end of a line of data for both physical and file I/O; for printing, the effect is that of ending a line of data and resuming printing at the beginning of the next line.	

2.2.3 Time literals

The need to provide external representations for two distinct types of time-related data is recognized: *duration* data for measuring or controlling the elapsed time of a control event, and *time of day* data (which may also include date information) for synchronizing the beginning or end of a control event to an absolute time reference.

2.2.3.1 Duration

Table 7 - Duration literal features

No.	Feature description	Examples
1a	Duration literals without underlines: short prefix	T#14ms T#-14ms T#14.7s T#14.7m T#14.7h t#14.7d t#25h15m t#5d14h12m18s3.5ms
1b	long prefix	TIME#14ms TIME#-14ms time#14.7s
2a	Duration literals with underlines: short prefix	t#25h_15m t#5d_14h_12m_18s_3.5ms
2b	long prefix	TIME#25h_15m time#5d_14h_12m_18s_3.5ms

2.2.3.2 Time of day and date

Table 8 - Date and time of day literals

No.	Feature description	Prefix Keyword
1	Date literals (long prefix)	DATE#
2	Date literals (short prefix)	D#
3	Time of day literals (long prefix)	TIME_OF_DAY#
4	Time of day literals (short prefix)	TOD#
5	Date and time literals (long prefix)	DATE_AND_TIME#
6	Date and time literals (short prefix)	DT#

Table 9 - Examples of date and time of day literals

Long prefix notation	Short prefix notation
DATE#1984-06-25 date#1984-06-25	D#1984-06-25 d#1984-06-25
TIME_OF_DAY#15:36:55.36 time_of_day#15:36:55.36	TOD#15:36:55.36 tod#15:36:55.36
DATE_AND_TIME#1984-06-25-15:36:55.36 date_and_time#1984-06-25-15:36:55.36	DT#1984-06-25-15:36:55.36 dt#1984-06-25-15:36:55.36

2.3 Data types

A number of elementary (pre-defined) data types are recognized by this standard. Additionally, generic data types are defined for use in the definition of overloaded functions. A mechanism for the user or manufacturer to specify additional data types is also defined.

2.3.1 Elementary data types

Table 10 - Elementary data types

No.	Keyword	Data type	Bits	Range
1	BOOL	Boolean	1	Note 8
2	SINT	Short integer	8	Note 2
3	INT	Integer	16	Note 2
4	DINT	Double integer	32	Note 2
5	LINT	Long integer	64	Note 2
6	USINT	Unsigned short integer	8	Note 3
7	UINT	Unsigned integer	16	Note 3
8	UDINT	Unsigned double integer	32	Note 3
9	ULINT	Unsigned long integer	64	Note 3
10	REAL	Real numbers	32	Note 4
11	LREAL	Long reals	64	Note 5
12	TIME	Duration	Note 1	Note 6
13	DATE	Date (only)	Note 1	Note 6
14	TIME_OF_DAY or TOD	Time of day (only)	Note 1	Note 6
15	DATE_AND_TIME or DT	Date and time of Day	Note 1	Note 6
16	STRING	Variable-length character string	Note 1	Note 7
17	BYTE	Bit string of length 8	8	Note 7
18	WORD	Bit string of length 16	16	Note 7
19	DWORD	Bit string of length 32	32	Note 7
20	LWORD	Bit string of length 64	64	Note 7

2.3.2 Generic data types

Table 11 - Hierarchy of generic data types

```
ANY
    ANY_NUM
        ANY_REAL
            LREAL
            REAL
        ANY_INT
            LINT, DINT, INT, SINT
            ULINT,  UDINT, UINT, USINT
```

ANY_BIT
   LWORD, DWORD, WORD, BYTE, BOOL
STRING
ANY_DATE
   DATE_AND_TIME
   DATE
   TIME_OF_DAY
TIME
Derived (see notes)

2.3.3 Derived data types

2.3.3.1 Declaration

Derived (i.e., user- or manufacturer-specified) data types can be declared using the TYPE...END_TYPE textual construction shown in table 12. These derived data types can then be used, in addition to the elementary data types, in variable declarations.

An *enumerated* data type declaration specifies that the value of any data element of that type can only take on one of the values given in the associated list of identifiers, as illustrated in table 12.

A *subrange* declaration specifies that the value of any data element of that type can only take on values between and including the specified upper and lower limits, as illustrated in table 12.

A STRUCT declaration specifies that data elements of that type shall contain sub-elements of specified types which can be accessed by the specified names. For instance, an element of data type ANALOG_CHANNEL_CONFIGURATION as declared in table 12 will contain a RANGE sub-element of type ANALOG_SIGNAL_RANGE, a MIN_SCALE sub-element of type ANALOG_DATA, and a MAX_SCALE element of type ANALOG_DATA.

An ARRAY declaration specifies that a sufficient amount of data storage shall be allocated for each element of that type to store all the data which can be indexed by the specified index subrange(s). Thus, any element of type ANALOG_16_INPUT_CONFIGURATION as shown in table 12 contains (among other elements) sufficient storage for 16 CHANNEL elements of type ANALOG_CHANNEL_CONFIGURATION. Mechanisms for access to array elements are defined in 2.4.1.2.

2.3.3.2 Initialization

The default initial value of an *enumerated* data type shall be the first identifier in the associated enumeration list, or a value specified by the assignment operator ":=". For instance, as shown in tables 12 and 14, the default initial values of elements of data types ANALOG_SIGNAL_TYPE and ANALOG_SIGNAL_RANGE are SINGLE_ENDED and UNIPOLAR_1_5V, respectively.

For data types with *subranges*, the default initial values shall be the first (lower) limit of the subrange, unless otherwise specified by an assignment operator. For instance, as declared in table 12, the default initial value of elements of type ANALOG_DATA is -4095, while the default initial value for the FILTER_PARAMETER sub-element of elements of type ANALOG_16_INPUT_CONFIGURATION is zero. In contrast, the default initial value of elements of type ANALOG_DATAZ as declared in table 14 is zero.

For other derived data types, the default initial values, unless specified otherwise by the use of the assignment operator ":=" in the TYPE declaration, shall be the default initial values of the underlying elementary data types as defined in table 13. Further examples of the use of the assignment operator for initialization are given in 2.4.2.

Table 12 - Data type declaration features

No.	Feature/textual example
1	Direct derivation from elementary types, e.g.: TYPE R : REAL ; END_TYPE

2	Enumerated data types, e.g.: TYPE ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ; END_TYPE
3	Subrange data types, e.g.: TYPE ANALOG_DATA : INT (-4095..4095) ; END_TYPE
4	Array data types, e.g.: TYPE ANALOG_16_INPUT_DATA : ARRAY [1..16] OF ANALOG_DATA ; END_TYPE
5	Structured data types, e.g.: TYPE ANALOG_CHANNEL_CONFIGURATION : STRUCT RANGE : ANALOG_SIGNAL_RANGE ; MIN_SCALE : ANALOG_DATA ; MAX_SCALE : ANALOG_DATA ; END_STRUCT ; ANALOG_16_INPUT_CONFIGURATION : STRUCT SIGNAL_TYPE : ANALOG_SIGNAL_TYPE ; FILTER_PARAMETER : SINT (0..99) ; CHANNEL : ARRAY [1..16] OF ANALOG_CHANNEL_CONFIGURATION ; END_STRUCT ; END_TYPE

Table 13 - Default initial values

Data type(s)	Initial value
BOOL, SINT, INT, DINT, LINT	0
USINT, UINT, UDINT, ULINT	0
BYTE, WORD, DWORD, LWORD	0
REAL, LREAL	0.0
TIME	T#0S
DATE	D#0001-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#0001-01-01-00:00:00
STRING	'' (the empty string)

Table 14 - Data type initial value declaration features

No.	Feature/textual example
1	Initialization of directly derived types, e.g.: TYPE FREQ : REAL := 50.0 ; END_TYPE
2	Initialization of enumerated data types, e.g.: TYPE ANALOG_SIGNAL_RANGE : (BIPOLAR_10V, (* -10 to +10 VDC *) UNIPOLAR_10V, (* 0 to +10 VDC *) UNIPOLAR_1_5V, (* + 1 to + 5 VDC *) UNIPOLAR_0_5V, (* 0 to + 5 VDC *) UNIPOLAR_4_20_MA, (* + 4 to +20 mADC *) UNIPOLAR_0_20_MA (* 0 to +20 mADC *)) := UNIPOLAR_1_5V ; END_TYPE
3	Initialization of subrange data types, e.g.: TYPE ANALOG_DATAZ : INT (-4095..4095) := 0 ; END_TYPE
4	Initialization of array data types, e.g.: TYPE ANALOG_16_INPUT_DATAI : ARRAY [1..16] OF ANALOG_DATA := [8(-4095), 8(4095)] ; END_TYPE
5	Initialization of structured data type elements, e.g.: TYPE ANALOG_CHANNEL_CONFIGURATIONI : STRUCT RANGE : ANALOG_SIGNAL_RANGE ; MIN_SCALE : ANALOG_DATA := -4095 ; MAX_SCALE : ANALOG_DATA := 4095 ; END_STRUCT ; END_TYPE
6	Initialization of derived structured data types, e.g.: TYPE ANALOG_CHANNEL_CONFIGZ : ANALOG_CHANNEL_CONFIGURATIONI(MIN_SCALE := 0, MAX_SCALE := 4000); END_TYPE

2.3.3.3 Usage

The usage of variables which are declared to be of derived data types shall conform to the following rules:

(1) A single-element variable of a derived type, can be used anywhere that a variable of its "parent's" type can be used, e.g. variables of the types R and FREQ as shown in tables 12 and 14 can be used anywhere that a variable of type REAL could be used, and variables of type ANALOG_DATA can be used anywhere that a variable of type INT could be used.

This rule can be applied recursively. For example, given the declarations below, the variable R3 of type R2 can be used anywhere a variable of type REAL can be used:

```
TYPE R1 : REAL := 1.0 ; END_TYPE
TYPE R2 : R1 ; END_TYPE
VAR R3: R2; END_VAR
```

(2) An element of a multi-element variable can be used anywhere the "parent" type can be used, e.g., given the declaration of ANALOG_16_INPUT_DATA in table 12 and the declaration

```
VAR INS : ANALOG_16_INPUT_DATA ; END_VAR
```

the variables INS[1] through INS[16] can be used anywhere that a variable of type INT could be used.

This rule can also be applied recursively, e.g., given the declarations of ANALOG_16_INPUT_CONFIGURATION, ANALOG_CHANNEL_CONFIGURATION, and ANALOG_DATA in table 12 and the declaration

VAR CONF : ANALOG_16_INPUT_CONFIGURATION ; END_VAR

the variable CONF.CHANNEL[2].MIN_SCALE can be used anywhere that a variable of type INT could be used.

2.4 Variables

In contrast to the external representations of data described in 2.2, *variables* provide a means of identifying data objects whose contents may change, e.g., data associated with the inputs, outputs, or memory of the programmable controller. A variable can be declared to be one of the elementary types, or one of the derived types.

2.4.1 Representation

2.4.1.1 Single-element variables

A *single-element variable* is defined as a variable which represents a single data element of one of the elementary types; a derived enumeration or subrange type; or a derived type whose "parentage, is traceable to an elementary, enumeration or subrange type. This subclause defines the means of representing such variables *symbolically,* or alternatively in a manner which *directly* represents the association of the data element with physical or logical locations in the programmable controller's input, output, or memory structure.

Identifiers shall be used for symbolic representation of variables.

Direct representation of a single-element variable shall be provided by a special symbol formed by the concatenation of the percent sign "%" (position 2/5 in the ISO/IEC 646 code table), a *location prefix* and a *size prefix* from table 15, and one or more unsigned integers, separated by periods.

Examples of directly represented variables are:

%QX75 and %Q75	Output bit 75
%IW215	Input word location 215
%QB7	Output byte location 7
%MD48	Double word at memory location 48
%IW2.5.7.1	See explanation below

The use of directly represented variables is only permitted in *programs, configurations* and *resources*. The maximum number of levels of hierarchical addressing is an implementation-dependent parameter.

Table 15 - Location and size prefix features for directly represented variables

No.	Prefix	Meaning	Default data type
1	I	Input location	
2	Q	Output location	
3	M	Memory location	
4	X	Single bit size	BOOL
5	None	Single bit size	BOOL
6	B	Byte (8 bits) size	BYTE
7	W	Word (16 bits) size	WORD
8	D	Double word (32 bits) size	DWORD
9	L	Long (quad) word (64 bits) size	LWORD

2.4.1.2 Multi-element variables

The *multi-element variable* types defined in this standard are *arrays* and *structures*.

An *array* is a collection of data elements of the same data type referenced by one or more *subscripts* enclosed in brackets and separated by commas. An example of the use of array variables in the ST language is:

```
OUTARY[%MB6,SYM] := INARY[0] + INARY[7] - INARY[%MB6] * %IW62 ;
```

A *structured variable* is a variable which is declared to be of a type which has previously been specified to be a *data structure*, i.e., a data type consisting of a collection of named elements.

Example: if the variable MODULE_5_CONFIG has been declared to be of type ANALOG_16_INPUT_CONFIGURATION as shown in table 12, the following statements in the ST language would cause the value SINGLE_ENDED to be assigned to the element SIGNAL_TYPE of the variable MODULE_5_CONFIG, while the value BIPOLAR_10V would be assigned to the RANGE sub-element of the fifth CHANNEL element of MODULE_5_CONFIG:

```
MODULE_5_CONFIG.SIGNAL_TYPE := SINGLE_ENDED;
MODULE_5_CONFIG.CHANNEL[5].RANGE := BIPOLAR_10V;
```

2.4.2 Initialization

When a configuration element (*resource* or *configuration*) is "started", each of the variables associated with the configuration element and its *programs* can take on one of the following initial values:

- the value the variable had when the configuration element was "stopped" (a *retained* value);
- a user-specified initial value;
- the default initial value for the variable's associated data type.

The user can declare that a variable is to be *retentive* by using the RETAIN qualifier specified in table 16, when this feature is supported by the implementation.

The initial value of a variable upon starting of its associated configuration element shall be determined according to the following rules:

1) If the starting operation is a "warm restart" as defined in IEC 1131-1, the initial values of *retentive* variables shall be their *retained* values as defined above.
2) If the operation is a "cold restart as defined in IEC 1131-1, the initial values of retentive variables shall be the user-specified initial values, or the default value for the associated data type of any variable for which no initial value is specified by the user.
3) Non-retained variables shall be initialized to the user-specified initial values, or to the default value for the associated data type of any variable for which no initial value is specified by the user.
4) Variables which represent *inputs* of the *programmable controller system* as defined in IEC 1131-1 shall be initialized in an implementation-dependent manner.

2.4.3 Declaration

Each programmable controller program organization unit type declaration (i.e., each declaration of a *program*, *function*, or *function block*, as defined in 2.5) shall contain at its beginning at least one *declaration part* which specifies the types (and, if necessary, the physical or logical location) of the variables used in the organization unit. This declaration part shall have the textual form of one of the keywords VAR, VAR_INPUT, or VAR_OUTPUT, followed in the case of VAR by zero or one occurrence of the qualifier RETAIN or the qualifier CONSTANT, and in the case of VAR_OUTPUT by zero or one occurrence of the qualifier RETAIN, followed by one or more declarations separated by semicolons and terminated by the keyword END_VAR. When a programmable controller supports the declaration by the user of initial values for variables, this declaration shall be accomplished in the declaration part(s) as defined in this subclause.

The *scope* (range of validity) of the declarations contained in the declaration part shall be *local* to the program organization unit in which the declaration part is contained. That is, the declared variables shall not be accessible to other program organization units except by explicit parameter passing via variables which have been declared as *inputs* or *outputs* of those units. The one exception to this rule is the case of variables which have been declared to be *global*. Such variables are only accessible to a program organization unit via a VAR_EXTERNAL declaration. The type of a variable declared in a VAR_EXTERNAL block shall agree with the type declared in the VAR_GLOBAL block of the associated *program, configuration* or *resource*.

Table 16 - Variable declaration keywords

Keyword	Variable usage
VAR	Internal to organization unit
VAR_INPUT	Externally supplied, not modifiable within organization unit
VAR_OUTPUT	Supplied by organization unit to external entities
VAR_IN_OUT	Supplied by external entities Can be modified within organization unit NOTE - Examples of the use of these variables are given in figures 11b and 12
VAR_EXTERNAL	Supplied by configuration via VAR_GLOBAL .Can be modified within organization unit
VAR_GLOBAL	Global variable declaration
VAR_ACCESS	Access path declaration
RETAIN	Retentive variables
CONSTANT	Constant (variable cannot be modified)
AT	Location assignment
NOTE - The usage of these keywords is a feature of the program organization unit or configuration element in which they are used.	

2.4.3.1 Type assignment

As shown in table 17, the VAR...END_VAR construction shall be used to specify data types and retentivity for directly represented variables. This construction shall also be used to specify data types, retentivity, and (where necessary, in *programs* only) the physical or logical location of symbolically represented single- or multi-element variables. The usage of the VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT constructions is defined in 2.5.

The assignment of a physical or logical address to a symbolically represented variable shall be accomplished by the use of the AT keyword. Where no such assignment is made, automatic allocation of the variable to an appropriate location in the programmable controller memory shall be provided.

Table 17 - Variable type assignment features

<table>
<tr><th>No.</th><th colspan="2">Feature/examples</th></tr>
<tr><td rowspan="2">1</td><td colspan="2">Declaration of directly represented, non-retentive variables</td></tr>
<tr><td>VAR
   AT %IW6.2 : WORD;
   AT %MW6   : INT ;
END_VAR</td><td>16-bit string (note 2)
16-bit integer, initial value = 0</td></tr>
<tr><td rowspan="2">2</td><td colspan="2">Declaration of directly represented retentive variables</td></tr>
<tr><td>VAR RETAIN
  AT %QW5 : WORD ;
END_VAR</td><td>At cold restart, %QW5 will be initialized to a 16-bit string with value 0</td></tr>
<tr><td rowspan="2">3</td><td colspan="2">Declaration of locations of symbolic variables</td></tr>
<tr><td>VAR_GLOBAL
  LIM_SW_S5 AT %IX27 : BOOL;
  CONV_START AT %QX25 : BOOL;
  TEMPERATURE AT %IW28: INT ;
END_VAR</td><td>Assigns input bit 27 to the Boolean variable LIM_SW_5 (note 2)
Assigns output bit 25 to the Boolean variable CONV_START
Assigns input word 28 to the integer variable TEMPERATURE (note 2)</td></tr>
<tr><td rowspan="2">4</td><td colspan="2">Array location assignment</td></tr>
<tr><td>VAR
  INARY AT %IW6 :
    ARRAY [0..9] OF INT ;
END_VAR</td><td>Declares an array of 10 integers to be allocated to contiguous input locations starting at %IW6 (note 2)</td></tr>
<tr><td rowspan="2">5</td><td colspan="2">Automatic memory allocation of symbolic variables</td></tr>
<tr><td>VAR
   CONDITION_RED : BOOL;
   IBOUNCE : WORD ;
    MYDUB : DWORD ;
    AWORD, BWORD, CWORD : INT;
    MYSTR: STRING[10] ;
END_VAR</td><td>Allocates a memory bit to the Boolean variable CONDITION_RED.
Allocates a memory word to the 16-bit string variable IBOUNCE.
Allocates a double memory word to the 32-bit-string variable MYDUB.
Allocates 3 separate memory words for the integer variables AWORD, BWORD, and CWORD.
Allocates memory to contain a string with a maximum length of 10 characters. After initialization, the string has length 0 and contains the empty string ''.</td></tr>
<tr><td rowspan="2">6</td><td colspan="2">Array declaration</td></tr>
<tr><td>VAR THREE :
  ARRAY[1..5,1..10,1..8] OF INT;
END_VAR</td><td>Allocates 400 memory words for a three-dimensional array of integers</td></tr>
<tr><td rowspan="2">7</td><td colspan="2">Retentive array declaration</td></tr>
<tr><td>VAR RETAIN RTBT:
  ARRAY[1..2,1..3] OF INT;
END_VAR</td><td>Declares retentive array RTBT with "cold restart initial values of 0 for all elements</td></tr>
<tr><td rowspan="2">8</td><td colspan="2">Declaration of structured variables</td></tr>
<tr><td>VAR MODULE_8_CONFIG :
   ANALOG_16_INPUT_CONFIGURATION;
END_VAR</td><td>Declaration of a variable of derived data type (see table 12)</td></tr>
</table>

NOTES

1 Features 1 to 4 can only be used in PROGRAM and VAR_GLOBAL declarations.

2 Initialization of system inputs is implementation-dependent.

2.4.3.2 Initial value assignment

The VAR...END_VAR construction shown in table 18 shall be used to specify initial values of directly represented variables. This construction shall also be used to assign initial values of symbolically represented single- or multi-element variables (the usage of the VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT constructions is defined in 2.5).

Initial values cannot be given in VAR_EXTERNAL declarations.

When a variable is declared to be of a derived, structured data type, initial values for the elements of the variable can be declared in a parenthesized list following the data type identifier, as shown in table 18. Elements for which initial values are not listed in the initial value list shall have the default initial values declared for those elements in the data type declaration.

Table 18 - Variable initial value assignment features

No.	Feature/examples	
1	Initialization of directly represented, non-retentive variables	
	VAR AT %QX5.1 : BOOL :=1; AT %MW6 : INT := 8 ; END_VAR	Boolean type, initial value =1 Initializes a memory word to integer 8
2	Initialization of directly represented retentive variables	
	VAR RETAIN AT %QW5 : WORD := 16#FF00 ; END_VAR	At cold restart, the 8 most significant bits of the 16-bit string at output word 5 are to be initialized to 1 and the 8 least significant bits to 0
3	Location and initial value assignment to symbolic variables	
	VAR VALVE_POS AT %QW28 : INT := 100; END_VAR	Assigns output word 28 to the integer variable VALVE_POS, with an initial value of 100
4	Array location assignment and initialization	
	VAR OUTARY AT %QW6 : ARRAY [0..9] OF INT := [10(1)] ; END_VAR	Declares an array of 10 integers to be allocated to contiguous output locations starting at %QW6, each with an initial value of 1
5	Initialization of symbolic variables	
	VAR MYBIT : BOOL := 1 ; OKAY : STRING[10] := 'OK'; END_VAR	Allocates a memory bit to the Boolean variable MYBIT with an initial value of 1. Allocates memory to contain a string with a maximum length of 10 characters. After initialization, the string has length 2 and contains the two-byte sequence of characters 'OK' in the ISO/IEC 646 character set, in an order appropriate for printing as a character string.
6	Array initialization	
	VAR BITS : ARRAY[0..7] OF BOOL := [1,1,0,0,0,1,0,0] ; TBT : ARRAY [1..2,1..3] OF INT := [1,2,3(4),6] ; END_VAR	Allocates 8 memory bits to contain initial values BITS[0]:= 1, BITS[1] := 1,..., BITS[6]:= 0, BITS[7] := 0. Allocates a 2-by-3 integer array TBT with initial values TBT[1,1]:=1, TBT[1,2]:=2, TBT[1,3]:=4, TBT[2,1]:=4, TBT[2,2]:=4, TBT[2,3]:=6.
7	Retentive array declaration and initialization	

	VAR RETAIN RTBT : ARRAY(1..2,1..3) OF INT := [1,2,3(4)]; END_VAR	Declares retentive array RTBT with "cold restart initial values of: RTBT[1,1] := 1, RTBT[1,2] := 2, RTBT[1,3] := 4, RTBT[2,1] := 4, RTBT[2,2] := 4, RTBT[2,3] := 0.
8	Initialization of structured variables	
	VAR MODULE_8_CONFIG : ANALOG_16_INPUT_CONFIGURATION (SIGNAL_TYPE := DIFFERENTIAL, CHANNEL := [4((RANGE := UNIPOLAR_1_5)), (RANGE := BIPOLAR_10_V, MIN_SCALE := 0, MAX_SCALE := 500)]) ; END_VAR	Initialization of a variable of derived data type (see table 12) NOTE - This example illustrates the declaration of a non-default initial value for the fifth element of the CHANNEL array of the variable MODULE_8_CONFIG.
9	Initialization of constants	
	VAR CONSTANT PI : REAL := 3.141592 ; END_VAR	
	NOTE - Features 1 to 4 can only be used in PROGRAM and VAR_GLOBAL declarations, as defined in 2.5.3 and 2.7.1 respectively.	

2.5 Program organization units

The program organization units defined in this Part of IEC 1311 are the *function, function block*, and *program*. These program organization units can be delivered by the manufacturer, or programmed by the user by the means defined in this part of the standard.

Program organization units shall not be *recursive*; that is, the invocation of a program organization unit shall not cause the invocation of another program organization unit of the same type.

2.5.1 Functions

For the purposes of programmable controller programming languages, a *function* is defined as a program organization unit which, when executed, yields exactly one data element (which can be multi-valued, e.g., an array or structure, and whose invocation can be used in textual languages as an operand in an expression. For example, the SIN and COS functions could be used as shown in figure 4.

a)

```
Z := SIN(X)*COS(Y) + COS(X)*SIN(Y) ;
```

b)

```
             +-----+
    X----+--| SIN |--+
         |  +-----+  |
         |           |  +---+     +---+
         |  +-----+  +--| * |-----| + |---Z
    Y--+----| COS |-----|   |  +--|   |
       | |  +-----+     +---+  |  +---+
       | |                     |
       | |  +-----+     +---+  |
       | +--| COS |-----| * |--+
       |    +-----+  +--|   |
       |             |  +---+
       |    +-----+  |
       +----| SIN |--+
            +-----+
```

Figure 4 - Examples of function usage
a) Structured Text (ST) language; b) Function Block Diagram (FBD) language;

Functions shall contain no internal state information, i.e., invocation of a function with the same arguments (input parameters) shall always yield the same value (output).

Any function type which has already been declared can be used in the declaration of another program organization unit, as shown in figure 3.

2.5.1.1 Representation

Functions and their invocation can be represented either graphically or textually.

In the graphic languages defined in clause 4 of this part, functions shall be represented as graphic blocks according to the following rules:

1) The form of the block shall be rectangular or square.
2) The size and proportions of the block may vary depending on the number of inputs and other information to be displayed.
3) The direction of processing through the block shall be from left to right (input parameters on the left and output parameter on the right).
4) The function name or symbol, as specified below, shall be located inside the block.
5) Provision shall be made for formal input parameter names appearing at the inside left of the block when the block represents:

   - one of the standard functions when the given graphical form includes the formal parameter names; or

   - any additional function declared.
6) Since the name of the function is used for the assignment of its output value, no formal output parameter name shall be shown at the right side of the block.
7) Actual parameter connections shall be shown by signal flow lines.
8) Negation of Boolean signals shall be shown by placing an open circle just outside of the input or output line intersection with the block. In the ISO/IEC 646 character set, this shall be represented by the upper case alphabetic "O", as shown in table 19.
9) The output of a graphically represented function shall be represented by a single line at the right side of the block, even though the output may be a multi-element variable.

Table 19 - Graphical negation of Boolean signals

No.	Feature	Representation

1	Negated input	`   +---+`
		`---O|   |---`
		`   +---+`

2	Negated output	`    +---+`
		`----|   |O---`
		`    +---+`

NOTE - If either of these features is supported for functions, it shall also be supported for function blocks, and vice versa.		

Figure 5 illustrates both the graphical and equivalent textual use of functions, including the use of a standard function (ADD) with no defined formal parameter names, and another standard function (SHL) with defined formal parameter names.

Example	Explanation

<pre>   +-----+
    | ADD |
B---| |---A
C---| |
D---| |
    +-----+</pre>
	Graphical use of "ADD" function
	(FBD language; see 4.3)
	(No formal parameter names)

`A := ADD(B,C,D);`	Textual use of "ADD" function (ST language)

<pre>   +-----+
    | SHL |
B---|IN |---A
C---|N |
    +-----+</pre>
	Graphical use of "SHL" function
	(FBD language)
	(Formal parameter names)

`A := SHL(IN := B,N := C);`	Textual use of "SHL" function (ST language)

Figure 5 - Use of formal parameter names

TABLE 19a - Textual invocation of functions

No.	Feature			Example
	Parameter assignment	Parameter order	Number of parameters	
1	yes	fixed	fixed	`A := LIMIT(MN:=1, IN:= B, MX:= 5);`
2	yes	any	any	`A := LIMIT(IN := B, MX := 5) ;`
3	no	fixed	fixed	`A := ADD(B,C,D) ;`

2.5.1.2 Execution control

As shown in table 20, an additional Boolean "EN" (Enable) input and "ENO" (Enable Out output shall be used with functions in the LD language, and their use shall also be possible in the FBD language defined in this part. These variables are considered to be available in every function according to the implicit declarations

```
VAR_INPUT   EN: BOOL := 1;  END_VAR
VAR_OUTPUT  ENO: BOOL;  END_VAR
```

When these variables are used, the execution of the operations defined by the function shall be controlled according to the following rules:

1) If the value of EN is FALSE (0) when the function is invoked, the operations defined by the function body shall not be executed and the value of "ENO" shall be reset to FALSE (0) by the programmable controller system.

2) Otherwise, the value of ENO shall be set to TRUE (1) by the programmable controller system, and the operations defined by the function body shall be executed. These operations can include the assignment of a Boolean value to ENO.

3) If one of the errors defined in annex E occurs during the execution of one of the standard functions, the ENO output of that function shall be reset to FALSE (0) by the programmable controller system.

NOTE - The use of the ENO output is an allowable exception to the rule that the execution of a function yields exactly one output.

Table 20 - Use of EN input and ENO output

No.	Feature	Example

1	Use of "EN" and "ENO" Required for LD (Ladder Diagram) language	see diagram 1 below
2	Use of "EN" and "ENO" Optional for FBD (Function Block Diagram) language	see diagram 2 below
3	FBD without "EN" and "ENO"	see diagram 3 below

Diagram 1:

```
          +-------+         |
| ADD_EN  |   +   |  ADD_OK |
+---||---|EN  ENO|---( )---+
|         |       |         |
|     A---|       |---C     |
|     B---|       |         |
          +-------+         |
```

Diagram 2:

```
          +-------+
          |   +   |
 ADD_EN--|EN  ENO|---ADD_OK
     A---|       |---C
     B---|       |
          +-------+
```

Diagram 3:

```
      +-----+
  A---|  +  |---C
  B---|     |
      +-----+
```

2.5.1.3 Declaration

a)

```
FUNCTION SIMPLE_FUN : REAL
  VAR_INPUT
    A,B : REAL ;          (* External interface specification *)
      C : REAL := 1.0;
   END_VAR
   SIMPLE_FUN := A*B/C;   (* Function body specification *)
END_FUNCTION
```

b)

```
FUNCTION
            +-------------+
            | SIMPLE_FUN  |
    REAL----|A            |----REAL
    REAL----|B            |(* External interface specification *)
    REAL----|C            |
            +-------------+

        +---+              (* Function body specification *)
    A---| * |   +---+
    B---|   |---| / |---SIMPLE_FUN
        +---+   |   |
    C-----------|   |
                +---+
END_FUNCTION
```

NOTE - In example a), the input variable C is given a default value of 1.0 to avoid a "division by zero" error if the input is not specified when the function is invoked, for example, if a graphical input to the function is left unconnected.

Figure 6 - Examples of function declarations
a) Textual declaration in ST language
b) Graphical declaration in FBD language

2.5.1.4 Typing, overloading, and type conversion

A standard function, function block type, operator, or instruction is said to be *overloaded* when it can operate on input data elements of various types within a generic type designator. For instance, an overloaded addition function on generic type ANY_NUM can operate on data of types LREAL, REAL, DINT, INT, and SINT.

Table 21 - Typed and overloaded functions

<table>
<tr><th>No.</th><th>Feature</th><th>Example</th></tr>
<tr><td>1</td><td>Overloaded functions</td><td><pre>
             +-----+
             | ADD |
ANY_NUM-----| |----ANY_NUM
ANY_NUM-----| |
   .   -----| |
   .   -----| |
ANY_NUM-----| |
             +-----+
</pre></td></tr>
<tr><td>2</td><td>Typed functions</td><td><pre>
          +---------+
          | ADD_INT |
INT-----| |----INT
INT-----| |
 . -----| |
 . -----| |
INT-----| |
          +---------+
</pre></td></tr>
</table>

<table>
<tr><th>Type declaration
(ST language)</th><th>Usage
(FBD language)
(ST language)</th></tr>
<tr><td><pre>
VAR
. A : INT ;
. B : INT ;
. C : INT ;
END_VAR
</pre></td><td><pre>
    +---+
A---| + |---C
B---| |
    +---+
C := A+B;
</pre></td></tr>
<tr><td><pre>
VAR
  A : INT ;
  B : REAL ;
  C : REAL;
END_VAR
</pre></td><td><pre>
    +-----------+   +---+
A---|INT_TO_REAL|---| + |---C
    +-----------+ | |
B-------------------| |
                    +---+
    C := INT_TO_REAL(A)+B;
</pre></td></tr>
<tr><td><pre>
VAR
  A : INT ;
  B : INT ;
  C : REAL;
END_VAR
</pre></td><td><pre>
     +---+   +-----------+
A----| + |---|INT_TO_REAL|---C
B----| | +-----------+
     +---+
    C := INT_TO_REAL(A+B);
</pre></td></tr>
</table>

NOTE - Type conversion is not required in the first example shown above.

Figure 7 - Examples of explicit type conversion with overloaded functions

<table>
<tr><th>Type declaration
(ST language)</th><th>Usage
(FBD language)
(ST language)</th></tr>
<tr><td>VAR
A : INT ;
B : INT ;
C : INT ;
END_VAR</td><td>+---------+
A---| ADD_INT |---C
B---| |
+---------+
C := ADD_INT(A,B);</td></tr>
<tr><td>VAR
A : INT ;
B : REAL ;
C : REAL;
END_VAR</td><td>+-----------+   +----------+
A---|INT_TO_REAL|---| ADD_REAL |---C
+-----------+ | |
B-----------------| |
+----------+
C := ADD_REAL(INT_TO_REAL(A),B);</td></tr>
<tr><td>VAR
A : INT ;
B : INT ;
C : REAL;
END_VAR</td><td>+---------+   +-----------+
A---| ADD_INT |---|INT_TO_REAL|---C
| | +-----------+
B---| |
+---------+
C := INT_TO_REAL(ADD_INT(A,B));</td></tr>
</table>

NOTE - Type conversion is not required in the first example shown above.

Figure 8 - Examples of explicit type conversion with typed functions

2.5.1.5 Standard functions

Definitions of functions common to all programmable controller programming languages are given in this subclause. Where graphical representations of standard functions are shown in this subclause, equivalent textual declarations may be written.

A standard function specified in this subclause to be *extensible* is allowed to have a variable number of inputs, and shall be considered as applying the indicated operation to each input in turn, e.g., extensible addition shall give as its output the sum of all its inputs. The maximum number of inputs of an extensible function is an implementation-dependent parameter.

2.5.1.5.1 Type conversion functions

Table 22 - Type conversion function features

<table>
<tr><th>No.</th><th>Graphical form</th><th>Usage example</th><th>Notes</th></tr>
<tr><td>1</td><td>+---------+
* ---| *_TO_** |--- **
+---------+

(*) - Input data type, e.g., INT
(**) - Output data type, e.g.,REAL
(*_TO_**) - Function name, e.g., INT_TO_REAL</td><td>A := INT_TO_REAL(B) ;</td><td>1
2
5</td></tr>
<tr><td>2</td><td>+-------+
ANY_REAL---| TRUNC |---ANY_INT
+-------+</td><td>A := TRUNC(B) ;</td><td>3</td></tr>
<tr><td>3</td><td>+-----------+
ANY_BIT--| BCD_TO_** |---ANY_INT
+-----------+</td><td>A := BCD_TO_INT(B) ;</td><td>4</td></tr>
<tr><td>4</td><td>+----------+
ANY_INT--| *_TO_BCD |---ANY_BIT
+----------+</td><td>A := INT_TO_BCD(B) ;</td><td>4</td></tr>
</table>

2.5.1.5.2 Numerical functions

Table 23 - Standard functions of one numeric variable

Graphical form

```
          +---------+
   *  ---|    **     |---  *
          +---------+
   (*) - Input/Output (I/O) type
   (**) - Function name
```

Usage example

A := SIN(B) ;
(ST language)

No.	Function name	I/O type	Description
	General functions		
1	ABS	ANY_NUM	Absolute value
2	SQRT	ANY_REAL	Square root
	Logarithmic functions		
3	LN	ANY_REAL	Natural logarithm
4	LOG	ANY_REAL	Logarithm base 10
5	EXP	ANY_REAL	Natural exponential
	Trigonometric functions		
6	SIN	ANY_REAL	Sine of input in radians
7	COS	ANY_REAL	Cosine in radians
8	TAN	ANY_REAL	Tangent in radians
9	ASIN	ANY_REAL	Principal arc sine
10	ACOS	ANY_REAL	Principal arc cosine
11	ATAN	ANY_REAL	Principal arc tangent

Table 24 - Standard arithmetic functions

Graphical form

```
              +-----+
ANY_NUM ---| *** |--- ANY_NUM
ANY_NUM ---|     |
   .    ---|     |
   .    ---|     |
ANY_NUM ---|     |
              +-----+
     (***) - Name or Symbol
```

Usage example

A := ADD(B,C,D) ;
or
A := B+C+D ;

No.	Name	Symbol	Description
	Extensible arithmetic functions		
12	ADD	+	OUT := IN1 + IN2 + ... + INn
13	MUL	*	OUT := IN1 * IN2 * ... * INn
	Non-extensible arithmetic functions		
14	SUB	-	OUT := IN1 - IN2
15	DIV	/	OUT := IN1 / IN2

16	MOD		OUT := IN1 modulo IN2
17	EXPT	**	Exponentiation: OUT := $IN1^{IN2}$
18	MOVE	:=	OUT := IN

2.5.1.5.3 Bit string functions

Table 25 - Standard bit shift functions

Graphical form	Usage example

<pre>   +------+
             | *** |
ANY_BIT ---|IN |--- ANY_BIT
UINT ------|N |
             +------+</pre>(***) - Function Name
	A := SHL(IN:=B, N:=5) ;
	
	(ST language)

No.	Name	Description
1	SHL	OUT := IN left-shifted by N bits, zero-filled on right
2	SHR	OUT := IN right-shifted by N bits, zero-filled on left
3	ROR	OUT := IN right-rotated by N bits, circular
4	ROL	OUT := IN left-rotated by N bits, circular
NOTE - The notation "OUT" refers to the function output.		

2.5.1.5.4 Selection and comparison functions

Graphical form	Usage examples

<pre>   +------+
ANY_BIT ---| *** |--- ANY_BIT
ANY_BIT ---| |
   .   ---| |
   .   ---| |
ANY_BIT ---| |
            +------+</pre>(***) - Name or symbol
	A := AND(B,C,D) ;
	or
	A := B & C & D ;

No.	Name	Symbol	Description
5	AND	&	OUT := IN1 & IN2 & ... & INn
6	OR	>=1	OUT := IN1 OR IN2 OR ... OR INn
7	XOR	=2k+1	OUT := IN1 XOR IN2 XOR ... XOR INn
8	NOT		OUT := NOT IN1

Table 27 - Standard selection functions

<table>
<tr><th>No.</th><th>Graphical form</th><th>Explanation/example</th></tr>
<tr><td>1</td><td>+------+
| SEL |
BOOL----|G |----ANY
ANY-----|IN0 |
ANY-----|IN1 |
+------+</td><td>Binary selection:
OUT := IN0 if G = 0
OUT := IN1 if G = 1
Example:
A := SEL(G:=0,IN0:=X,IN1:=5) ;</td></tr>
<tr><td>2a</td><td>+------+
| MAX |
(Note 1)---| |----ANY
: ---| |
(Note 1)---| |
+------+</td><td>Extensible maximum function:
OUT := MAX (IN1,IN2, ...,INn)
Example:
A := MAX(B,C,D) ;</td></tr>
<tr><td>2b</td><td>+------+
| MIN |
(Note 1)---| |----ANY
: ---| |
(Note 1)---| |
+------+</td><td>Extensible minimum function:
OUT := MIN (IN1,IN2, ...,INn)
Example:
A := MIN(B,C,D) ;</td></tr>
<tr><td>3</td><td>+--------+
| LIMIT |
(Note 1)--|MN |----ANY
(Note 1)--|IN |
(Note 1)--|MX |
+--------+</td><td>Limiter:
OUT := MIN(MAX(IN,MN),MX)
Example:
A := LIMIT(IN:=B,MN:=0,MX:=5);</td></tr>
<tr><td>4</td><td>+------+
| MUX |
ANY_INT---|K |----ANY
ANY---| |
: ---| |
ANY---| |
+------+</td><td>Extensible multiplexer:
Select one of "N" inputs
depending on input K
Example:
A := MUX(0, B, C, D);
would have the same effect as
A := B ;</td></tr>
</table>

Table 28 - Standard comparison functions

<table>
<tr><th>Graphical form</th><th>Usage examples</th></tr>
<tr><td>+------+
(Note 1)--| *** |--- BOOL
: --| |
(Note 1)--| |
+------+
(***) - Name or Symbol</td><td>A := GT(B,C,D) ;
or
A := (B>C) & (C>D) ;</td></tr>
</table>

No.	Name	Symbol	Description
5	GT	>	Decreasing sequence: OUT := (IN1>IN2) & (IN2>IN3) & ... & (INn-1 > INn)
6	GE	>=	Monotonic sequence: OUT := (IN1>=IN2) & (IN2>=IN3) & ... & (INn-1 >= INn)
7	EQ	=	Equality: OUT := (IN1=IN2) & (IN2=IN3) & ... & (INn-1 = INn)
8	LE	<=	Monotonic sequence: OUT := (IN1<=IN2) & (IN2<=IN3) & ... & (INn-1 <= INn)
9	LT	<	Increasing sequence: OUT := (IN1<IN2) & (IN2<IN3) & ... & (INn-1 < INn)

10	NE	<>	Inequality (non-extensible) OUT := (IN1 <> IN2)

2.5.1.5.5 Character string functions

Table 29 - Standard character string functions

<table>
<tr><th>No.</th><th>Graphical form</th><th>Explanation/example</th></tr>
<tr><td>1</td><td>+------+
STRING---| LEN |---INT
+------+</td><td>String length function
Example:
A := LEN('ASTRING')
is equivalent to A := 7;</td></tr>
<tr><td>2</td><td>+-------+
| LEFT |
STRING----|IN |--STRING
UINT------|L |
+-------+</td><td>Leftmost L characters of IN
Example:
A := LEFT(IN:='ASTR',L:=3);
is equivalent to
A := 'AST' ;</td></tr>
<tr><td>3</td><td>+--------+
| RIGHT |
STRING----|IN |--STRING
UINT------|L |
+--------+</td><td>Rightmost L characters of IN
Example:
A := RIGHT(IN:='ASTR',L:=3);
is equivalent to
A := 'STR' ;</td></tr>
<tr><td>4</td><td>+--------+
| MID |
STRING----|IN |--STRING
UINT------|L |
UINT------|P |
+--------+</td><td>L characters of IN,
beginning at the P-th
Example:
A := MID(IN:='ASTR',L:=2,P:=2);
is equivalent to
A := 'ST' ;</td></tr>
<tr><td>5</td><td>+---------+
| CONCAT |
STRING---| |--STRING
: ---| |
STRING---| |
+---------+</td><td>Extensible concatenation
Example:
A := CONCAT('AB','CD','E') ;
is equivalent to
A := 'ABCDE' ;</td></tr>
<tr><td>6</td><td>+---------+
| INSERT |
STRING---|IN1 |--STRING
STRING---|IN2 |
UINT-----|P |
+---------+</td><td>Insert IN2 into IN1 after the
P-th character position
Example:
A:=INSERT(IN1:='ABC',IN2:='XY',P=2);
is equivalent to
A := 'ABXYC' ;</td></tr>
<tr><th>No.</th><th>Graphical form</th><th>Explanation/example</th></tr>
<tr><td>7</td><td>+---------+
| DELETE |
STRING---|IN |--STRING
UINT-----|L |
UINT-----|P |
+---------+</td><td>Delete L characters of IN, beginning
at the P-th character position
Example:
A := DELETE(IN:='ABXYC',L:=2, P:=3) ;
is equivalent to
A := 'ABC' ;</td></tr>
</table>

8	<pre> +---------+
	         | REPLACE |
	STRING---|IN1 |--STRING
	STRING---|IN2 |
	UINT-----|L |
	UINT-----|P |
	         +---------+</pre>
		Replace L characters of IN1 by IN2, starting at the P-th character position
		Example:
		A := REPLACE(IN1:='ABCDE',IN2:='X', L:=2, P:=3) ;
		is equivalent to
		A := 'ABXE' ;

9	<pre> +--------+
	         | FIND |
	STRING---|IN1 |---INT
	STRING---|IN2 |
	         +--------+</pre>
		Find the character position of the beginning of the first occurrence of IN2 in IN1. If no occurrence of IN2 is found, then OUT := 0.
		Example:
		A := FIND(IN1:='ABCBC',IN2:='BC') ;
		is equivalent to A := 2 ;

NOTE - The examples in this table are given in the Structured Text (ST) language.		

2.5.1.5.6 Functions of time data types

In addition to the comparison and selection functions defined in 2.5.1.5.4, the combinations of input and output time data types shown in table 30 shall be allowed with the associated functions.

2.5.1.5.7 Functions of enumerated data types

The selection and comparison functions listed in table 31 can be applied to inputs which are of an enumerated data type.

Table 30 - Functions of time data types

	Numeric and concatenation functions				
No.	**Name**	**Symbol**	**IN1**	**IN2**	**OUT**
1	ADD	+	TIME	TIME	TIME
2			TIME_OF_DAY	TIME	TIME_OF_DAY
3			DATE_AND_TIME	TIME	DATE_AND_TIME
4	SUB	-	TIME	TIME	TIME
5			DATE	DATE	TIME
6			TIME_OF_DAY	TIME	TIME_OF_DAY
7			TIME_OF_DAY	TIME_OF_DAY	TIME
8			DATE_AND_TIME	TIME	DATE_AND_TIME
9			DATE_AND_TIME	DATE_AND_TIME	TIME
10	MUL	*	TIME	ANY_NUM	TIME
11	DIV	/	TIME	ANY_NUM	TIME
12	CONCAT		DATE	TIME_OF_DAY	DATE_AND_TIME
	Type conversion functions				
13 14	DATE_AND_TIME_TO_TIME_OF_DAY DATE_AND_TIME_TO_DATE				

Table 31 - Functions of enumerated data types

No.	Name	Symbol	Feature No. in Tables 27 and 28
1	SEL		1
2	MUX		4
3	EQ	=	7
4	NE	<>	10

NOTE - The provisions of NOTES 2-5 of table 28 apply to this table.

2.5.2 Function blocks

For the purposes of programmable controller programming languages, a *function block* is a program organization unit which, when executed, yields one or more values. Multiple, named *instances* (copies) of a function block can be created. Each instance shall have an associated identifier (the *instance name*), and a data structure containing its output and internal variables, and, depending on the implementation, values of or references to its input parameters. All the values of the output variables and the necessary internal variables of this data structure shall persist from one execution of the function block to the next; therefore, invocation of a function block with the same arguments (input parameters) need not always yield the same output values.

Only the input and output parameters shall be accessible outside of an instance of a function block, i.e., the function block's internal variables shall be hi31Ôdden from the user of the fction block.

Execution of the operations of a function block shall be invoked as defined in clause 3 for textual languages, according to the rules of network evaluation given in clause 4 for graphic languages, or under the control of sequential function chart (SFC) elements.

Any function block type which has already been declared can be used in the declaration of another function block type or program type as shown in figure 3.

The scope of an instance of a function block shall be local to the program organization unit in which it is instantiated, unless it is declared to be global in a VAR_GLOBAL block as defined in 2.7.1.

As illustrated in 2.5.2.2, the instance name of a function block instance can be used as the input to a function or function block if declared as an input variable in a VAR_INPUT declaration, or as an input/output variable of a function block in a VAR_IN_OUT declaration, as defined in 2.4.3.

2.5.2.1 Representation

Graphical (FBD language)

```
        FF75
      +------+
      |  SR  |
%IX1---|S1  Q1|---%QX3
%IX2---|R     |
      +------+
```

Textual (ST language)

```
VAR FF75: SR; END_VAR      (* Declaration *)
FF75(S1:=%IX1, R:=%IX2);  (* Invocation *)
%QX3 := FF75.Q1 ;          (* Assign Output *)
```

Figure 9 - Function block instantiation example

Assignment of a value to an output variable of a function block is not allowed except from within the function block. The assignment of a value to the input of a function block is permitted only as part of the invocation of the function block.

Table 32 - Examples of function block I/O parameter usage

Usage	Inside function block	Outside function block
Input read	IF S1 THEN ...	Not allowed
Input write	Not allowed (notes 1 and 3)	FF75(S1:=%IX1,R:=%IX2);
Output read	Q1 := Q1 AND NOT R;	%QX3 := FF75.Q1;
Output write	Q1 := 1;	Not Allowed

2.5.2.2 Declaration

As illustrated in figure 10, a function block shall be declared textually or graphically in the same manner as defined for functions in 2.5.1.3, with the differences summarized in table 33:

As illustrated in figure 12, only variables or function block instance names can be passed into a function block via the VAR_IN_OUT construct, i.e., function or function block outputs cannot be passed via this construction. This is to prevent the inadvertent modifications of such outputs. However, "cascading" of VAR_IN_OUT constructions is permitted, as illustrated in figure 12c.

a)

```
FUNCTION_BLOCK DEBOUNCE
(*** External Interface ***)
VAR_INPUT
  IN : BOOL ;                   (* Default = 0 *)
  DB_TIME : TIME := t#10ms ;    (* Default = t#10ms *)
END_VAR
VAR_OUTPUT OUT : BOOL ;        (* Default = 0 *)
    ET_OFF : TIME ;            (* Default = t#0s *)
END_VAR
VAR DB_ON : TON ;              (** Internal Variables **)
    DB_OFF : TON ;             (**  and FB Instances  **)
    DB_FF : SR ;
END_VAR
(** Function Block Body **)
DB_ON(IN := IN, PT := DB_TIME) ;
DB_OFF(IN := NOT IN, PT:=DB_TIME) ;
DB_FF(S1 :=DB_ON.Q, R := DB_OFF.Q) ;
OUT := DB_FF.Q ;
ET_OFF := DB_OFF.ET ;
END_FUNCTION_BLOCK
```

b)

```
FUNCTION_BLOCK
(** External Interface **)
                  +----------------+
                  |    DEBOUNCE    |
          BOOL---|IN           OUT|---BOOL
          TIME---|DB_TIME   ET_OFF|---TIME
                  +----------------+
(** Function Block Body **)
              DB_ON        DB_FF
             +------+     +-----+
             | TON  |     | SR  |
 IN----+------|IN   Q|-----|S1  Q|---OUT
       |   +---|PT  ET|  +--|R    |
       |   |   +------+  |  +-----+
       |   |             |
       |   |    DB_OFF   |
       |   |   +------+  |
       |   |   | TON  |  |
       +--|--O|IN   Q|--+
 DB_TIME--+---|PT  ET|--------------ET_OFF
             +------+
 END_FUNCTION_BLOCK
```

Figure 10 - Examples of function block declarations
a) Textual declaration in ST language
b) Graphical declaration in FBD language

Table 33 - Function block declaration features

<table>
<tr><th>No.</th><th>Description</th><th>Example</th></tr>
<tr><td>1</td><td>RETAIN qualifier on internal variables</td><td>VAR RETAIN X : REAL ; END_VAR</td></tr>
<tr><td>2</td><td>RETAIN qualifier on output variables</td><td>VAR_OUTPUT RETAIN X : REAL ; END_VAR</td></tr>
<tr><td>3</td><td>RETAIN qualifier on internal function blocks</td><td>VAR RETAIN TMR1: TON ; END_VAR</td></tr>
<tr><td>4a</td><td>Input/output declaration (textual)</td><td>VAR_IN X: INT; END_VAR
VAR_IN_OUT A: INT ; END_VAR
A := A+X ;</td></tr>
<tr><td>4b</td><td>Input/output declaration (graphical)</td><td>See figure 12</td></tr>
<tr><td>5a</td><td>Function block instance name as input (textual)</td><td>VAR_INPUT I_TMR: TON ; END_VAR
EXPIRED := I_TMR.Q;   (* Note 1 *)</td></tr>
<tr><td>5b</td><td>Function block instance name as input (graphical)</td><td>See figure 11a</td></tr>
<tr><td>6a</td><td>Function block instance name as input/output (textual)</td><td>VAR_IN_OUT IO_TMR: TOF ; END_VAR
IO_TMR(IN:=A_VAR, PT:=T#10S);
EXPIRED := IO_TMR.Q;   (* Note 1 *)</td></tr>
<tr><td>6b</td><td>Function block instance name as input/output (graphical)</td><td>See figure 11b</td></tr>
<tr><td>7a</td><td>Function block instance name as external variable (textual)</td><td>VAR_EXTERNAL EX_TMR : TOF ;END_VAR
EX_TMR(IN:=A_VAR, PT:=T#10S);
EXPIRED := EX_TMR.Q;   (* Note 1 *)</td></tr>
<tr><td>7b</td><td>Function block instance name as external variable (graphical)</td><td>See figure 11c</td></tr>
<tr><td>8a
8b</td><td>Textual declaration of:
rising edge inputs
falling edge inputs</td><td>FUNCTION_BLOCK AND_EDGE   (* Note 2 *)
VAR_INPUT X : BOOL R_EDGE;
Y : BOOL F_EDGE;
END_VAR
VAR_OUTPUT Z : BOOL ; END_VAR
Z := X AND Y ;   (* ST language example *)
END_FUNCTION_BLOCK   (*- see 3.3 *)</td></tr>
<tr><td>9a

9b</td><td>Graphical declaration of:
rising edge inputs

falling edge inputs</td><td><pre>FUNCTION_BLOCK   (* Note 2 *)
        +-----------+   (* External interface *)
        | AND_EDGE |
BOOL--->X   Z|---BOOL
        | |
BOOL---<Y |
        | |
        +-----------+
         +---+   (* Function block body *)
     X---| & |---Z   (* FBD language example *)
     Y---| | (* - see 4.3 *)
         +---+
END_FUNCTION_BLOCK</pre></td></tr>
</table>

NOTES

1 It is assumed in these examples that the variables EXPIRED and A_VAR have been declared of type BOOL.

2 The declaration of function block AND_EDGE in the above examples is equivalent to:

```
FUNCTION_BLOCK AND_EDGE
    VAR INPUT X : BOOL; Y : BOOL; END_VAR
    VAR X_TRIG : R_TRIG ; Y_TRIG : F_TRIG ; END_VAR
    X_TRIG(CLK := X) ;
    Y_TRIG(CLK := Y) ;
    Z := X_TRIG.Q AND Y_TRIG.Q;
END_FUNCTION_BLOCK
```

See 2.5.2.3.2 for the definition of the edge detection function blocks R_TRIG and F_TRIG.

```
FUNCTION_BLOCK

        +---------------+     (* External interface *)
        |   INSIDE_A    |
  TON---|I_TMR   EXPIRED|---BOOL
        +---------------+

              I_TMR           (* Function Block body *)
             +------+
             | TON  |
             |IN   Q|---EXPIRED
             |PT  ET|
             +------+
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK

         +---------------+      (* External interface *)
         |  EXAMPLE_A    |
  BOOL---|GO         DONE|---BOOL
         +---------------+

              E_TMR             (* Function Block body *)
             +------+                  I_BLK
             | TON  |            +---------------+
        GO---|IN   Q|            |   INSIDE_A    |
   t#100ms---|PT  ET|     E_TMR---|I_TMR   EXPIRED|---DONE
             +------+            +---------------+
END_FUNCTION_BLOCK
```

Figure 11a - Graphical use of a function block name as an input variable (table 33, feature 5b)

```
FUNCTION_BLOCK

        +---------------+      (* External interface *)
        |   INSIDE_B    |
  TON---|I_TMR----I_TMR|---TON
  BOOL--|TMR_GO EXPIRED|---BOOL
        +---------------+

             I_TMR           (* Function Block body *)
            +------+
            | TON  |
    TMR_GO---|IN   Q|---EXPIRED
            |PT  ET|
            +------+
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK

         +---------------+      (* External interface *)
         |  EXAMPLE_B    |
  BOOL---|GO        DONE|---BOOL
         +---------------+

             E_TMR             (* Function Block body *)
            +------+                  I_BLK
            | TON  |            +----------------+
            |IN   Q|            |    INSIDE_B    |
   t#100ms---|PT  ET|     E_TMR---|I_TMR-----I_TMR|
            +------+     GO------|TMR_GO  EXPIRED|---DONE
                                +----------------+
END_FUNCTION_BLOCK
```

Figure 11b - Graphical use of a function block name as an input/output variable (table 33, feature 6b)

```
FUNCTION_BLOCK
        +---------------+       (* External interface *)
        |   INSIDE_C    |
  BOOL--|TMR_GO EXPIRED|---BOOL
        +---------------+

VAR_EXTERNAL X_TMR : TON ; END_VAR

              X_TMR             (* Function Block body *)
             +------+
             | TON  |
    TMR_GO---|IN   Q|---EXPIRED
             |PT  ET|
             +------+
END_FUNCTION_BLOCK
```

```
PROGRAM
         +---------------+      (* External interface *)
         |  EXAMPLE_C    |
  BOOL---|GO         DONE|---BOOL
         +---------------+

  VAR_GLOBAL X_TMR : TON ; END_VAR


              I_BLK             (* Program body *)
         +----------------+
         |    INSIDE_C    |
 GO------|TMR_GO  EXPIRED|---DONE
         +----------------+
END_PROGRAM
```

NOTE - PROGRAM declaration is defined in 2.5.3.

Figure 11c - Graphical use of a function block name as an external variable (table 33, feature 7b)

a)

```
          +--------+
          | ACCUM  |
    INT---|A-----A|---INT
    INT---|X       |
          +--------+
            +---+
        A---| + |---A
        X---|   |
            +---+
```

```
FUNCTION_BLOCK ACCUM
  VAR_IN_OUT A : INT ; END_VAR
  VAR_INPUT X : INT ; END_VAR
  A := A+X ;
END_FUNCTION_BLOCK
```

b)

```
                  ACC1
                +--------+
                | ACCUM  |
   ACC----------|A-----A|---ACC
        +---+   |        |
   X1---| * |---|X       |
   X2---|   |   +--------+
        +---+
```

c)

```
                ACC1                      ACC2
              +--------+               +--------+
              | ACCUM  |               | ACCUM  |
 ACC----------|A-----A|---------------|A-----A|---ACC
      +---+   |        |       +---+   |        |
 X1---| * |---|X       |  X3---| * |---|X       |
 X2---|   |   +--------+  X4---|   |   +--------+
      +---+                    +---+
```

<table>
<tr><td>d)</td><td><pre>
                  ACC1
     +---+   +-------+
X1---| * | | ACCUM |
X2---| |---|A-----A|---ACC
     +---+ | |
X3-----------|X |
              +-------+
</pre></td><td></td></tr>
</table>

Figure 12 - Examples of use of input/output variables
a) Graphical and textual declarations
b), c) Legal usage
d) Illegal usage

2.5.2.3 Standard function blocks

2.5.2.3.1 Bistable elements

Table 34 - Standard bistable function blocks

<table>
<tr><th>No.</th><th>Graphical form</th><th>Function block body</th></tr>
<tr><td>1</td><td colspan="2">Bistable Function Block (set dominant) (notes 1 and 2)</td></tr>
<tr><td></td><td><pre>
       +-----+
       | SR |
BOOL---|S1 Q1|---BOOL
BOOL---|R |
       +-----+
</pre></td><td><pre>
                      +-----+
S1----------------| >=1 |---Q1
         +---+ | |
 R------O| & |----| |
 Q1------| | | |
         +---+   +-----+
</pre></td></tr>
<tr><td>2</td><td colspan="2">Bistable Function Block (reset dominant) (notes 1 and 2)</td></tr>
<tr><td></td><td><pre>
       +-----+
       | RS |
BOOL---|S Q1|---BOOL
BOOL---|R1 |
       +-----+
</pre></td><td><pre>
                        +---+
 R1----------------O| & |---Q1
         +-----+ | |
 S-------| >=1 |----| |
 Q1------| | | |
         +-----+   +---+
</pre></td></tr>
</table>

2.5.2.3.2 Edge detection

The graphic representation of standard rising- and falling-edge detecting function blocks shall be as shown in table 35. The behaviors of these blocks shall be equivalent to the definitions given in this table. This behavior corresponds to the following rules:

1) The "Q" output of an R_TRIG function block shall stand at the Boolean "1" value from one execution of the function block to the next, following the "0" to "1" transition of the "CLK" input, and shall return to "0" at the next execution.

2) The "Q" output of an F_TRIG function block shall stand at the Boolean "1" value from one execution of the function block to the next, following the "1" to "0" transition of the "CLK" input, and shall return to "0" at the next execution.

Table 35 - Standard edge detection function blocks

No.	Graphical form	Definition (ST language)
1	Rising edge detector	

<table>
<tr><td></td><td><pre>
      +--------+
      | R_TRIG |
BOOL---|CLK   Q|---BOOL
      +--------+
</pre></td><td><pre>
FUNCTION_BLOCK R_TRIG
   VAR_INPUT CLK: BOOL; END_VAR
   VAR_OUTPUT Q: BOOL; END_VAR
   VAR RETAIN M: BOOL; END_VAR
Q := CLK AND NOT M;
M := CLK;
END_FUNCTION_BLOCK
</pre></td></tr>
<tr><td rowspan="2">2</td><td colspan="2">Falling edge detector</td></tr>
<tr><td><pre>
      +--------+
      | F_TRIG |
BOOL---|CLK   Q|---BOOL
      +--------+
</pre></td><td><pre>
FUNCTION_BLOCK F_TRIG
   VAR_INPUT CLK: BOOL; END_VAR
   VAR_OUTPUT   Q: BOOL; END_VAR
   VAR RETAIN M: BOOL; END_VAR
Q := NOT CLK AND NOT M;
M := NOT CLK;
END_FUNCTION_BLOCK
</pre></td></tr>
</table>

2.5.2.3.3 Counters

Table 36 - Standard counter function blocks

<table>
<tr><th>No.</th><th>Graphical form</th><th>Function block body
(ST language)</th></tr>
<tr><td rowspan="2">1</td><td colspan="2">Up-counter</td></tr>
<tr><td><pre>
        +-----+
        | CTU |
BOOL--->CU Q|---BOOL
BOOL---|R |
 INT---|PV CV|---INT
        +-----+
</pre></td><td><pre>
IF R THEN CV := 0 ;
ELSIF CU AND (CV < PVmax)
  THEN CV := CV+1;
END_IF ;
Q := (CV >= PV) ;
</pre></td></tr>
<tr><td rowspan="2">2</td><td colspan="2">Down-counter</td></tr>
<tr><td><pre>
        +-----+
        | CTD |
BOOL--->CD Q|---BOOL
BOOL---|LD |
 INT---|PV CV|---INT
        +-----+
</pre></td><td><pre>
IF LD THEN CV := PV ;
ELSIF CD AND (CV > PVmin)
  THEN CV := CV-1;
END_IF ;
Q := (CV <= 0) ;
</pre></td></tr>
<tr><td rowspan="2">3</td><td colspan="2">Up-down counter</td></tr>
<tr><td><pre>
        +------+
        | CTUD |
BOOL--->CU QU|---BOOL
BOOL--->CD QD|---BOOL
BOOL---|R |
BOOL---|LD |
 INT---|PV CV|---INT
        +------+
</pre></td><td><pre>
IF R THEN CV := 0 ;
ELSIF LD THEN CV := PV ;
ELSE
  IF NOT (CU AND CD) THEN
    IF CU AND (CV < PVmax)
    THEN CV := CV+1;
    ELSIF CD AND (CV > PVmin)
    THEN CV := CV-1;
    END_IF;
  END_IF;
END_IF ;
QU := (CV >= PV) ;
QD := (CV <= 0) ;
</pre></td></tr>
<tr><td colspan="3">NOTE - The numerical values of the limit variables PVmin and PVmax are implementation-dependent.</td></tr>
</table>

2.5.2.3.4 Timers

Table 37 - Standard timer function blocks

No.	Description	Graphical form

1
2a
2b
3a
3b
	`*** is: TP   (Pulse)`
	`TON  (On-delay)`
	`T---0  (On-delay)`
	`TOF   (Off-delay)`
	`0---T  (Off-delay)`
		`+--------+`
		`|  ***   |`
		`BOOL---|IN     Q|---BOOL`
		`TIME---|PT    ET|---TIME`
		`+--------+`

4	Real-time clock	

	PDT = Preset date and time, loaded on rising edge of EN
	CDT = Current date and time, valid when EN = 1
	Q = copy of EN
		`+--------+`
		`|  RTC   |`
		`BOOL---|IN     Q|---BOOL`
		`DT-----|PDT CDT|-----DT`
		`+--------+`

Table 38 - Standard timer function blocks - timing diagrams

Pulse (TP) timing

```
     +---------+    ++ ++   +---------+
IN   |         |    || ||   |         |
   --+         +----++-++---+         +---------
     t0       t1    t2 t3    t4        t5

     +----+         +----+  +----+
Q    |    |         |    |  |    |
   --+    +---------+    +--+    +-------------
     t0   t0+PT    t2 t2+PT t4  t4+PT

  PT      +---+          +       +---+
   :     /    |         /|      /    |
ET :    /     |        / |     /     |
   :   /      |       /  |    /      |
   :  /       |      /   |   /       |
   0-+        +------+   +---+       +---------
     t0       t1    t2       t4       t5
```

(continued on following page)

Table 38 - Standard timer Function Blocks - timing diagrams - continued

On-delay (TON) timing

```
     +--------+        +---+   +--------+
IN   |        |        |   |   |        |
   --+        +--------+   +---+        +-------------
     t0       t1       t2  t3  t4       t5

          +---+                     +---+
Q         |   |                     |   |
   -------+   +---------------------+   +-------------
       t0+PT  t1                t4+PT   t5

  PT      +---+                     +---+
   :     /    |            +       /    |
ET :    /     |           /|      /     |
   :   /      |          / |     /      |
   :  /       |         /  |    /       |
   0-+        +--------+   +---+        +-------------
     t0       t1       t2  t3  t4       t5
```

Off-delay (TOF) timing

```
       +--------+        +---+   +--------+
IN     |        |        |   |   |        |
    ---+        +--------+   +---+        +-----------
       t0       t1       t2  t3  t4       t5

       +-------------+   +---------------------+
Q      |             |   |                     |
    ---+             +---+                     +------
       t0          t1+PT t2                  t5+PT

  PT                 +---+                     +------
   :                /    |       +            /
ET :               /     |      /|           /
   :              /      |     / |          /
   :             /       |    /  |         /
   0------------+        +---+   +--------+
                t1           t3           t5
```

2.5.2.3.5 Communication function blocks

Standard communication function blocks for programmable controllers are defined in IEC 1131-5. These function blocks provide programmable communications functionality such as device verification, polled data acquisition, programmed data acquisition, parametric control, interlocked control, programmed alarm reporting, and connection management and protection.

2.5.3 Programs

A *program* is defined in IEC 1131-1 as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system."

The declaration and usage of *programs* is identical to that of *function blocks* as defined in 2.5.2.1 and 2.5.2.2, with the additional features shown in table 39 and the following differences:

1) The delimiting keywords for program declarations shall be PROGRAM...END_PROGRAM.

2) A program can contain a VAR_ACCESS...END_VAR construction, which provides a means of specifying named variables which can be accessed by some of the communication services specified in IEC 1131-5. An *access path* associates each such variable with an input, output or internal variable of the program The format and usage of this declaration shall be as described in 2.7.1 and in IEC 1131-5.

3) *Programs* can only be instantiated within *resources*, as defined in 2.7.1, while *function blocks* can only be instantiated within *programs* or other *function blocks*.

The declaration and use of programs are illustrated in figure 19, and in examples F.7 and F.8 of annex F.

Table 39 - Program declaration features

No.	DESCRIPTION
1 to 9b	Same as features 1 to 9b, respectively, of table 33
10	Formal input and output parameters
11 to 14	Same as features 1 to 4, respectively, of table 17
15 to 18	Same as features 1 to 4, respectively, of table 18
19	Use of directly represented variables (subclause 2.4.1.1)
20	VAR_GLOBAL...END_VAR declaration within a PROGRAM (see 2.4.3 and 2.7.1)
21	VAR_ACCESS...END_VAR declaration within a PROGRAM

2.7 Configuration elements

As described in 1.4.1, a *configuration* consists of *resources*, *tasks* (which are defined within *resources*), *global variables*, and *access paths*. Each of these elements is defined in detail in this subclause.

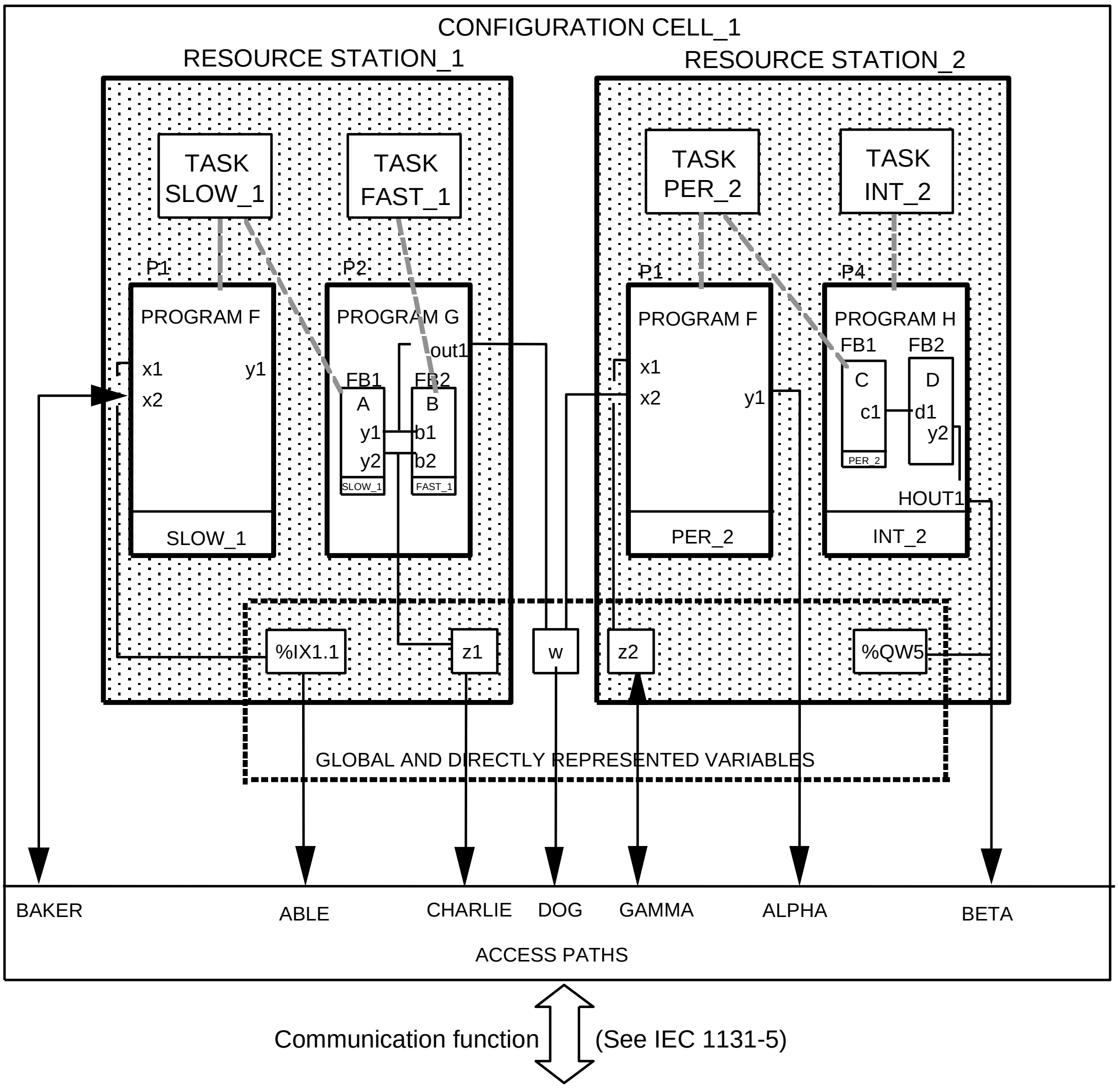


Figure 19a - Graphical example of a configuration

```
FUNCTION_BLOCK A
  VAR_OUTPUT y1 : UINT ;
             y2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK B
  VAR_INPUT b1 : UINT ;
            b2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK C
  VAR_OUTPUT c1 : BOOL ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK D
   VAR_INPUT d1 : BOOL ; END_VAR
   VAR_OUTPUT y2 : INT ; END_VAR
END_FUNCTION_BLOCK
```

```
PROGRAM F
  VAR_INPUT  x1 : BOOL ;  x2 : UINT ; END_VAR
  VAR_OUTPUT y1 : BYTE ; END_VAR
END_PROGRAM
```

```
PROGRAM G
  VAR_OUTPUT out1 : UINT ;  END_VAR
  VAR_EXTERNAL z1 : BYTE ;  END_VAR
  VAR  FB1 : A ;  FB2 : B ; END_VAR
  FB1(...);  out1 := FB1.y1;  z1 := FB1.y2;
  FB2(b1 := FB1.y1, b2 := FB1.y2) ;
END_PROGRAM
```

```
PROGRAM H
  VAR_OUTPUT HOUT1: INT ;  END_VAR
  VAR  FB1 : C ;  FB2 : D ; END_VAR
  FB1(...) ;
  FB2(d1 := FB1.c1);  HOUT1 := FB2.y2;
END_PROGRAM
```

Figure 19b - Skeleton function block and program declarations for configuration example

2.7.1 Configurations, resources, and access paths

Table 49 enumerates the language features for declaration of *configurations, resources, global variables,* and *access paths*. Partial enumeration of TASK declaration features is also given; additional information on *tasks* is provided in 2.7.2. The formal syntax for these features is given in B.1.7. Figure 20 provides examples of these features, corresponding to the example configuration shown in figure 19a and the supporting declarations in figure 19b.

The ON qualifier in the RESOURCE...ON...END_RESOURCE construction is used to specify the type of "processing function" and its "man-machine interface" and "sensor and actuator interface" functions upon which the *resource* and its associated *programs* and *tasks* are to be implemented. The manufacturer shall supply a *resource library* of such functions, as illustrated in figure 3. Associated with each element in this library shall be an identifier (the *resource type name*) for use in resource declaration.

The *scope* of a VAR_GLOBAL declaration shall be limited to the *configuration* or *resource* in which it is declared, with the exception that an *access path* can be declared to a *global* variable in a *resource* using feature 10d in table 49.

Table 49 - Configuration and resource declaration features

No.	DESCRIPTION
1	CONFIGURATION...END_CONFIGURATION construction
2	VAR_GLOBAL...END_VAR construction within CONFIGURATION
3	RESOURCE...ON...END_RESOURCE construction
4	VAR_GLOBAL...END_VAR construction within RESOURCE
5a	Periodic TASK construction within RESOURCE (Note 1)

5b	Non-periodic TASK construction within RESOURCE (Note 1)
6a	PROGRAM declaration with PROGRAM-to-TASK association using the WITH construction (Note 1)
6b	PROGRAM declaration with Function Block-to-TASK association using the WITH construction (Note 1)
6c	PROGRAM declaration with no TASK association (Note 1)
7	Declaration of directly represented variables in VAR_GLOBAL (Note 2)
8a	Connection of directly represented variables to PROGRAM inputs
8b	Connection of GLOBAL variables to PROGRAM inputs
9a	Connection of PROGRAM outputs to directly represented variables
9b	Connection of PROGRAM outputs to GLOBAL variables
10a	VAR_ACCESS...END_VAR construction
10b	Access paths to directly represented variables
10c	Access paths to PROGRAM inputs
10d	Access paths to GLOBAL variables in RESOURCES
10e	Access paths to GLOBAL variables in CONFIGURATIONS
10f	Access paths to PROGRAM outputs
NOTES 1. See 2.7.2 for further description of TASK features. 2. See 2.4.3.1 for further description of related features.	

No.	EXAMPLE
1	`CONFIGURATION CELL_1`
2	`  VAR_GLOBAL  w: UINT;  END_VAR`
3	`  RESOURCE STATION_1 ON PROCESSOR_TYPE_1`
4	`    VAR_GLOBAL  z1: BYTE;  END_VAR`
5a	`    TASK SLOW_1(INTERVAL := t#20ms, PRIORITY := 2) ;`
5a	`    TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1) ;`
6a	`    PROGRAM P1 WITH SLOW_1 :`
8a	`                 F(x1 := %IX1.1) ;`
9b	`    PROGRAM P2 : G(OUT1 => w,`
6b	`                   FB1 WITH SLOW_1,`
6b	`                   FB2 WITH FAST_1)  ;`
3	`  END_RESOURCE`
3	`  RESOURCE STATION_2 ON PROCESSOR_TYPE_2`
4	`    VAR_GLOBAL  z2      : BOOL ;`
7	`                AT %QW5 : INT  ;`
4	`    END_VAR`
5a	`    TASK PER_2(INTERVAL := t#50ms, PRIORITY := 2) ;`
5b	`    TASK INT_2(SINGLE := z2,       PRIORITY := 1) ;`
6a	`    PROGRAM P1 WITH PER_2 :`
8b	`          F(x1 := z2, x2 := w)  ;`
6a	`    PROGRAM P4 WITH INT_2 :`
9a	`          H(HOUT1 => %QW5,`
6b	`            FB1 WITH  PER_2);`

3	`  END_RESOURCE`
10a	`  VAR_ACCESS`
10b	`    ABLE    : STATION_1.%IX1.1     : BOOL READ_ONLY  ;`
10c	`    BAKER   : STATION_1.P1.x2      : UINT READ_WRITE ;`
10d	`    CHARLIE : STATION_1.z1         : BYTE            ;`
10e	`    DOG     : w                    : UINT READ_ONLY  ;`
10f	`    ALPHA   : STATION_2.P1.y1      : BYTE READ_ONLY  ;`
10f	`    BETA    : STATION_2.P4.HOUT1  : INT READ_ONLY   ;`
10d	`    GAMMA   : STATION_2.z2         : BOOL READ_WRITE ;`
10a	`  END_VAR`
1	`END_CONFIGURATION`

Figure 20 - Examples of CONFIGURATION and RESOURCE declaration features

2.7.2 Tasks

For the purposes of IEC 1131-3, a *task* is defined as an execution control element which is capable of invoking, either on a periodic basis or upon the occurrence of the rising edge of a specified Boolean variable, the execution of a set of program organization units, which can include *programs* and *function blocks* whose instances are specified in the declaration of *programs*.

Tasks and their association with program organization units can be represented graphically or textually using the WITH construction, as shown in table 50, as part of *resources* within *configurations*. A task is implicitly enabled or disabled by its associated resource according to the mechanisms defined in 1.4.1. The control of program organization units by enabled tasks shall conform to the following rules:

1) The associated program organization units shall be scheduled for execution upon each rising edge of the SINGLE input of the task.

2) If the INTERVAL input is non-zero, the associated program organization units shall be scheduled for execution periodically at the specified interval as long as the SINGLE input stands at zero (0). If the INTERVAL input is zero (the default value), no periodic scheduling of the associated program organization units shall occur.

3) The PRIORITY input of a task establishes the scheduling priority of the associated program organization units, with zero (0) being highest priority and successively lower priorities having successively higher numeric values. As shown in table 50, the priority of a program organization unit (that is, the priority of its associated task) can be used for *preemptive* or *non-preemptive* scheduling.

   a) In *non-preemptive* scheduling, processing power becomes available on a *resource* when execution of a program organization unit or operating system function is complete. When processing power is available, the program organization unit with highest scheduled priority shall begin execution. If more than one program organization unit is waiting at the highest scheduled priority, then the program organization unit with the longest waiting time at the highest scheduled priority shall be executed.

   b) In *preemptive* scheduling, when a program organization unit is scheduled, it can *interrupt* the execution of a program organization unit of lower priority on the same *resource*, that is, the execution of the lower-priority unit can be suspended until the execution of the higher-priority unit is completed. A program organization unit shall not interrupt the execution of another unit of the same or higher priority.

   NOTE - Depending on schedule priorities, a program organization unit might not begin execution at the instant it is scheduled. However, in the examples shown in table 50, all program organization units meet their *deadlines*, that is, they all complete execution before being scheduled for re-execution. The manufacturer shall provide information to enable the user to determine whether all deadlines will be met in a proposed configuration.

4) A *program* with no task association shall have the lowest system priority. Any such program shall be scheduled for execution upon "starting" of its *resource*, as defined in 1.4.1, and shall be re-scheduled for execution as soon as its execution terminates.

5) When a *function block instance* is associated with a task, its execution shall be under the exclusive control of the task, independent of the rules of evaluation of the program organization unit in which the task-associated function block instance is declared.

6) Execution of a *function block instance* which is not directly associated with a task shall follow the normal rules for the order of evaluation of language elements for the program organization unit (which can itself be under the control of a task) in which the function block instance is declared.

7) The execution of function blocks within a program shall be synchronized to ensure that data concurrency is achieved according to the following rules:

   a) If a function block receives more than one input from another function block, then when the former is executed, all inputs from the latter shall represent the results of the same evaluation. For instance, in the example represented by figure 21a, when Y2 is evaluated, the inputs Y2.A and Y2.B shall represent the outputs Y1.C and Y1.D from the same (not two different) evaluations of Y1.

   b) If two or more function blocks receive inputs from the same function block, and if the "destination" blocks are all explicitly or implicitly associated with the same task, then the inputs to all such "destination" blocks at the time of their evaluation shall represent the results of the same evaluation of the "source" block. For instance, in the example represented by figures 21b and 21c, when Y2 and Y3 are evaluated in the normal course of evaluating program P1, the inputs Y2.A and Y2.B shall be the results of the same evaluation of Y1 as the inputs Y3.A and Y3.B.

Provision shall be made for storage of the outputs of functions or function blocks which have explicit task associations, or which are used as inputs to program organization units which have explicit task associations, as necessary to satisfy the rules given above.

Table 50 - Task features

<table>
<tr><th>No.</th><th>Description/Examples</th></tr>
<tr><td>1a</td><td>Textual declaration of periodic TASK (feature 5a of table 49)</td></tr>
<tr><td>1b</td><td>Textual declaration of non-periodic TASK (feature 5b of table 49)</td></tr>
<tr><td rowspan="6">2a

2b</td><td>Graphical representation of TASKs within a RESOURCE</td></tr>
<tr><td><pre>
          TASKNAME
        +---------+
        | TASK |
BOOL---|SINGLE |
TIME---|INTERVAL |
UINT---|PRIORITY |
        +---------+
</pre></td></tr>
<tr><td>Graphical representation of periodic TASKs</td></tr>
<tr><td><pre>
             SLOW_1   FAST_1
           +---------+   +---------+
           | TASK | | TASK |
           |SINGLE | |SINGLE |
t#20ms---|INTERVAL | t#10ms---|INTERVAL |
      2---|PRIORITY | 1---|PRIORITY |
           +---------+   +---------+
</pre></td></tr>
<tr><td>Graphical representation of non-periodic TASK</td></tr>
<tr><td><pre>
           INT_2
         +---------+
         | TASK |
%IX2---|SINGLE |
         |INTERVAL |
    1---|PRIORITY |
         +---------+
</pre></td></tr>
</table>

<table>
<tr><td>3a</td><td>Textual association with PROGRAMs (feature 6a of table 49)</td></tr>
<tr><td>3b</td><td>Textual association with FUNCTION BLOCKs (feature 6b of table 49)</td></tr>
<tr><td rowspan="2">4a</td><td>Graphical association with PROGRAMs (within RESOURCEs)</td></tr>
<tr><td><pre>
RESOURCE STATION_2
                     P1   P4
                 +--------+   +--------+
                 | F | | H |
                 | | | |
                 | | | |
                 +--------+   +--------+
                 | PER_2 | | INT_2 |
                 +--------+   +--------+

END_RESOURCE
</pre></td></tr>
<tr><td rowspan="2">4b</td><td>Graphical association with FUNCTION BLOCKs
(within PROGRAMs inside RESOURCEs)</td></tr>
<tr><td><pre>
RESOURCE STATION_1
                                P2
       +---+
       | G |
       | |
       | FB1   FB2 |
       | +-------+   +-------+ |
       | | A | | B | |
       | | | | | |
       | | | | | |
       | +-------+   +-------+ |
       | |SLOW_1| |FAST_1| |
       | +-------+   +-------+ |
       +---+
END_RESOURCE
</pre></td></tr>
<tr><td rowspan="3">5a</td><td>Non-preemptive scheduling</td></tr>
<tr><td>Example 1:
- RESOURCE STATION_1 as configured in figure 20
- Execution times: P1 = 2 ms; P2 = 8 ms;
- P2.FB1 = P2.FB2 = 2 ms (NOTE 3)
- STATION_1 starts at t = 0</td></tr>
<tr><td>SCHEDULE (repeats every 40 ms)
<table>
<tr><th>t(ms)</th><th>Executing</th><th>Waiting</th></tr>
<tr><td>0</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
<tr><td>2</td><td>P1 @ 2</td><td>P2.FB1 @ 2, P2</td></tr>
<tr><td>4</td><td>P2.FB1 @ 2</td><td>P2</td></tr>
<tr><td>6</td><td>P2</td><td></td></tr>
<tr><td>10</td><td>P2</td><td>P2.FB2 @ 1</td></tr>
<tr><td>14</td><td>P2.FB2 @ 1</td><td>P2</td></tr>
<tr><td>16</td><td>P2</td><td>(P2 restarts)</td></tr>
<tr><td>20</td><td>P2</td><td>P2.FB2 @ 1, P1 @ 2, P2.FB1 @ 2</td></tr>
<tr><td>24</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
<tr><td>26</td><td>P1 @ 2</td><td>P2.FB1 @ 2, P2</td></tr>
<tr><td>28</td><td>P2.FB1 @ 2</td><td>P2</td></tr>
<tr><td>30</td><td>P2.FB2 @ 1</td><td>P2</td></tr>
</table></td></tr>
</table>

	32	P2	
	40	P2.FB2 @ 1	P1 @ 2, P2.FB1 @ 2, P2

5a **Non-preemptive scheduling**

Example 2:
- RESOURCE STATION_2 as configured in figure 20
- Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (NOTE 4)
- INT_2 is triggered at t = 25, 50, 90, ... ms
- STATION_2 starts at t = 0

SCHEDULE

t(ms)	Executing	Waiting
0	P1 @ 2	P4.FB1 @ 2
25	P1 @ 2	P4.FB1 @ 2, P4 @ 1
30	P4 @ 1	P4.FB1 @ 2
35	P4.FB1 @ 2	
50	P4 @ 1	P1 @ 2, P4.FB1 @ 2
55	P1 @ 2	P4.FB1 @ 2
85	P4.FB1 @ 2	
90	P4.FB1 @ 2	P4 @ 1
95	P4 @ 1	
100	P1 @ 2	P4.FB1 @ 2

5b **Preemptive scheduling**

Example 3:
- RESOURCE STATION_1 as configured in figure 20
- Execution times: P1 = 2 ms; P2 = 8 ms; P2.FB1 = P2.FB2 = 2 ms (NOTE 3)
- STATION_1 starts at t = 0

SCHEDULE

t(ms)	Executing	Waiting
0	P2.FB2 @ 1	P1 @ 2, P2.FB1 @ 2, P2
2	P1 @ 2	P2.FB1 @ 2, P2
4	P2.FB1 @ 2	P2
6	P2	
10	P2.FB2 @ 1	P2
12	P2	
16	P2	(P2 restarts)
20	P2.FB2 @ 1	P1 @ 2, P2.FB1 @ 2, P2

5b **Preemptive scheduling**

Example 4:
- RESOURCE STATION_2 as configured in figure 20
- Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (NOTE 4)
- INT_2 is triggered at t = 25, 50, 90, ... ms
- STATION_2 starts at t = 0

SCHEDULE

t(ms)	Executing	Waiting
0	P1 @ 2	P4.FB1 @ 2
25	P4 @ 1	P1 @ 2, P4.FB1 @ 2
30	P1 @ 2	P4.FB1 @ 2
35	P4.FB1 @ 2	
50	P4 @ 1	P1 @ 2, P4.FB1 @ 2
55	P1 @ 2	P4.FB1 @ 2
85	P4.FB1 @ 2	

90	P4 @ 1	P4.FB1 @ 2
95	P4.FB1 @ 2	
100	P1 @ 2	P4.FB1 @ 2

```
RESOURCE R1
             fast1                          slow1
         +-----------+                  +-----------+
         |   TASK    |                  |   TASK    |
t#10ms---|INTERVAL   |         t#20ms---|INTERVAL   |
     1---|PRIORITY   |              2---|PRIORITY   |
         +-----------+                  +-----------+


      P1
      PROGRAM X
            Y1                  Y2
          +-----+             +-----+
          |  Y  |             |  Y  |
       ---|A   C|----+--------|A   C|---
       ---|B   D|----|--+-----|B   D|---
          +-----+    |  |     +-----+
          |slow1|    |  |     |fast1|
          +-----+    |  |     +-----+
                     |  |
                     |  |     Y3
                     |  |  +-----+
                     |  |  |  Y  |
                     +--|--|A   C|---
                        +--|B   D|---
                           +-----+
                           |fast1|
                           +-----+

      END_PROGRAM
```

Figure 21a - Synchronization of function blocks with explicit task associations

```
RESOURCE R1
             fast1                          slow1
         +-----------+                  +-----------+
         |   TASK    |                  |   TASK    |
t#10ms---|INTERVAL   |         t#20ms---|INTERVAL   |
     1---|PRIORITY   |              2---|PRIORITY   |
         +-----------+                  +-----------+


      P1
```

```
PROGRAM X
      Y1                   Y2
    +------+             +------+
    |  Y   |             |  Y   |
 ---|A    C|-----+---------|A    C|---
 ---|B    D|-----|--+------|B    D|---
    +------+     |  |      +------+
    |fast1|      |  |
    +------+     |  |
                 |  |
                 |  |      Y3
                 |  |   +------+
                 |  |   |  Y   |
                 +--|--|A    C|---
                    +--|B    D|---
                       +------+
 END_PROGRAM

                  slow1
```

Figure 21b - Synchronization of function blocks with implicit task associations

```
RESOURCE R1
              fast1                              slow1
          +-----------+                      +-----------+
          |   TASK    |                      |   TASK    |
t#10ms---|INTERVAL   |          t#20ms---|INTERVAL   |
      1---|PRIORITY   |               2---|PRIORITY   |
          +-----------+                      +-----------+


      P1
PROGRAM X
      Y1                   Y2
    +------+             +------+
    |  Y   |             |  Y   |
 ---|A    C|-----+---------|A    C|---
 ---|B    D|-----|--+------|B    D|---
    +------+     |  |      +------+
    |fast1|      |  |      |slow1|
    +------+     |  |      +------+
                 |  |
                 |  |      Y3
                 |  |   +------+
                 |  |   |  Y   |
                 +--|--|A    C|---
                    +--|B    D|---
                       +------+
                       |slow1|
                       +------+

END_PROGRAM
```

Figure 21c - Explicit task associations equivalent to figure 21b