# A Standard Functions

This appendix contains a complete overview of all the standard PLC functions described by means of examples in Chapter 2. For every standard function of IEC 61131-3, the following information is given:

- Graphical declaration
- (Semantic) description
- Specification of some functions in Structured Text (ST).

Standard functions have input variables (formal parameters) as well as a function value (the returned value of the function). Some input variables are not named. In order to describe their functional behaviour the following conventions apply:

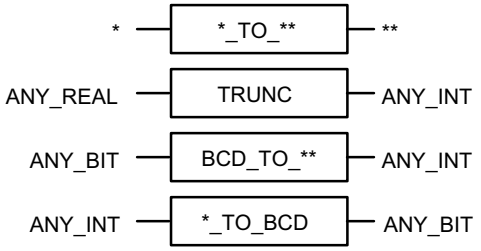
- An individual input variable without name is designated as "IN"
- Several input variables without names are numbered "IN1, IN2, ..., INn"
- The function value is designated as "F".

General data types (such as ANY or ANY_BIT) are used in the description. Their meaning is explained in Section 3.4.3 and they are summarised in Table 3.9. The designator ANY here stands for one of the data types: ANY_BIT, ANY_NUM, ANY_STRING, ANY_DATE or TIME.

Many standard functions have a textual name as well as an alternative representation with a symbol (e.g. ADD and "+"). In the figures and tables, both versions are given.

K.-H. John, M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems*, 2nd ed., DOI 10.1007/978-3-642-12015-2,

## A.1 Type Conversion Functions




| | |
|---|---|
| * | Input data type, e.g. REAL |
| ** | Output data type, e.g. INT |
| *_TO_** | Function name, e.g. REAL_TO_INT |

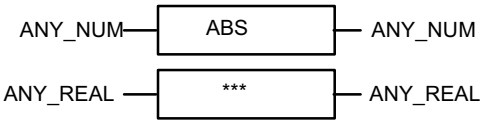
**Figure A.1.** Graphical declarations of the type conversion functions

These standard functions convert the input variable into the data type of their function value (type conversion).

| Name | Function / Description |
|---|---|
| *_TO_** | When REAL values are converted to INT values, they are rounded up or down to the next whole number. Halves, e.g. 0.5 or 0.05, are rounded up. |
| TRUNC | This function cuts off the places of a REAL value after the decimal point to form an integer value. |
| BCD | The input and/or output values of type ANY_BIT represent BCD-coded bit strings for the data types BYTE, WORD, DWORD and LWORD. BCD coding is not defined by IEC 61131-3, it is implementation-dependent. |

**Table A.1.** Description of the type conversion functions

## A.2 Numerical Functions



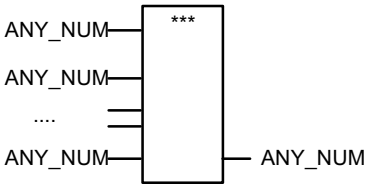

*** stands for: SQRT, LN, LOG, EXP,
SIN, COS, TAN,
ASIN, ACOS, ATAN

**Figure A.2.** Graphical declarations of the numerical functions

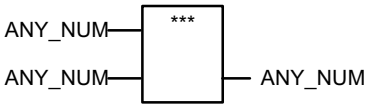
| Name | Function | Description |
|---|---|---|
| ABS | Absolute value | F := \| IN \| |
| SQRT | Square root | F := $\sqrt{}$IN |
| LN | Natural logarithm | F := $\log_e$ (IN) |
| LOG | Logarithm base 10 | F := $\log_{10}$ (IN) |
| EXP | Exponent base e | F := $e^{IN}$ |
| SIN | Sine, IN in radians | F := SIN (IN) |
| COS | Cosine, IN in radians | F := COS (IN) |
| TAN | Tangent, IN in radians | F := TAN (IN) |
| ASIN | Principal arc sine | F := ARCSIN (IN) |
| ACOS | Principal arc cosine | F := ARCCOS (IN) |
| ATAN | Principal arc tangent | F := ARCTAN (IN) |

**Table A.2.** Description of the numerical functions

## A.3 Arithmetic Functions

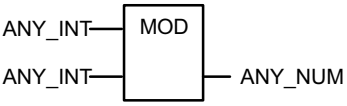


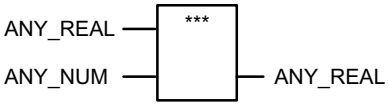

*** stands for: ADD, +
MULT, *




*** stands for: SUB, -
DIV, /

**Figure A.3.** Graphical declarations of the arithmetic functions ADD, MUL, SUB and DIV




*** Stands for: EXPT, **


*** stands for: MOVE, :=

**Figure A.4.** Graphical declarations of the standard arithmetic functions MOD, EXPT and MOVE

| Name | Symbol | Function | Description |
|---|---|---|---|
| ADD | + | Addition | F := IN1 + IN2 + ... + INn |
| MUL | * | Multiplication | F := IN1 * IN2 * ... * INn |
| SUB | - | Subtraction | F := IN1 - IN2 |
| DIV | / | Division | F := IN1 / IN2 |
| MOD | | Remainder formation | F := IN1 - (IN1/ IN2)*IN2 |
| EXPT | ** | Exponentiation | F := $IN1^{IN2}$ |
| MOVE | := | Assignment | F := IN |

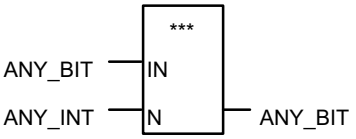
**Table A.3.** Description of the arithmetic functions

ADD and SUB use the type ANYMAGNITUDE, which includes operations on TIME. EXPT uses the type ANY_REAL for IN1 and the type ANY_NUM for IN2. MOVE uses ANY for input and output.

In the case of the division of integers, the result must also be an integer. If necessary, the result is truncated in the direction of zero.

If the input parameter IN2 is zero, an error is reported at run time with error cause "division by zero", see also Appendix E.

## A.4 Bit-Shift Functions



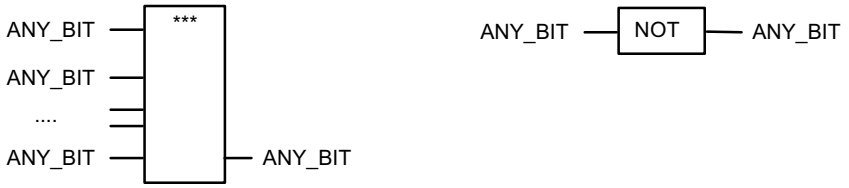

*** stands for: SHL, SHR, ROL, ROR

**Figure A.5.** Graphical declarations of the bit-shift functions SHL, SHR, ROR and ROL

| Name | Function | Description |
|---|---|---|
| SHL | Shift to the left | Shift IN to the left by N bits, fill with zeros from the right |
| SHR | Shift to the right | Shift IN to the right by N bits, fill with zeros from the left |
| ROR | Rotate to the right | Shift IN to the right by N bits, in a circle |
| ROL | Rotate to the left | Shift IN to the left by N bits, in a circle |

**Table A.4.** Description of the bit-shift functions. Input N must be greater than or equal to zero.

## A.5 Bitwise Boolean Functions




*** stands for: AND, &, OR, >=1, XOR, =2k+1

**Figure A.6.** Graphical declarations of the bitwise Boolean functions AND, OR, XOR and NOT

| Name | Symbol | Function | Description |
|---|---|---|---|
| AND | & | Bit-by-bit AND | F := IN1 & IN2 & ... & INn |
| OR | >=1 | Bit-by-bit OR | F := IN1 v IN2 v ... v INn |
| XOR | =2k+1 | Bit-by-bit XOR | F := IN1 XOR IN2 XOR ... XOR INn |
| NOT | | Negation | F := ¬ IN |

**Table A.5.** Description of the bitwise Boolean functions

The logic operation is performed on the input parameters bit-by-bit. That is, every bit position of one input is gated with the corresponding bit position of the other input and the result is stored in the same bit position of the function value.

An inversion can also be represented graphically by a circle "o" at the Boolean input or output of a function.

## A.6 Selection Functions for Max., Min. and Limit

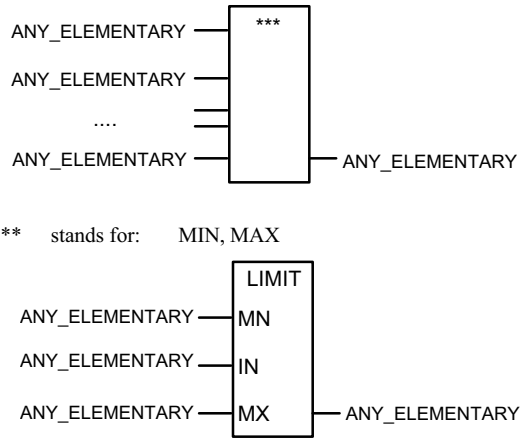



**Figure A.7.** Graphical declarations of the selection functions MAX, MIN and LIMIT

| Name | Function | Description |
|---|---|---|
| MAX | Maximum formation | F := MAX (IN1, IN2, ... , INn) |
| MIN | Minimum formation | F := MIN (IN1, IN2, ... , INn) |
| LIMIT | Limit | F := MIN (MAX (IN, MN), MX) |

**Table A.6.** Description of the selection functions MAX, MIN and LIMIT

These three standard functions are specified by declarations in ST in Example A.1 and Example A.2.

```
FUNCTION      MAX : ANY_ELEMENTARY  (* maximum formation *)
  VAR_INPUT   IN1, IN2, ... INn   : ANY_ELEMENTARY;        END_VAR
  VAR         Elem                : ANY_ELEMENTARY;        END_VAR
              (* ANY_ELEMENTARY  stands for any data type*)


  IF IN1 > IN2 THEN        (* first comparison *)
      Elem    := IN1;
  ELSE
      Elem    := IN2;
  END_IF;
  IF IN3 > Elem THEN       (* next comparison *)
      Elem    := IN3;
  END_IF;
  ...
  IF INn > Elem THEN       (* last comparison *)
      Elem    := INn;
  END_IF;
  MAX    :=   Elem;        (* writing the function value *)
END_FUNCTION
```

**Example A.1.** Specification of the selection function MAX in ST; for MIN replace all „>“ with „<“.

The specification of the MIN function can be obtained by replacing all occurrences of ">" by "<" in the MAX specification in Example A.1.

```
FUNCTION      LIMIT : ANY_ELEMENTARY (* limit formation *)
  VAR_INPUT
    MN    :    ANY_ELEMENTARY;
    IN    :    ANY_ELEMENTARY;
    MX    :    ANY_ELEMENTARY;
  END_VAR
  MAX    :=   MIN ( MAX ( IN, MN), MX);        (* call of MIN of MAX *)
END_FUNCTION
```

**Example A.2.** Specification of the selection function LIMIT in ST

## A.7 Selection Functions for Binary Selection and Multiplexers

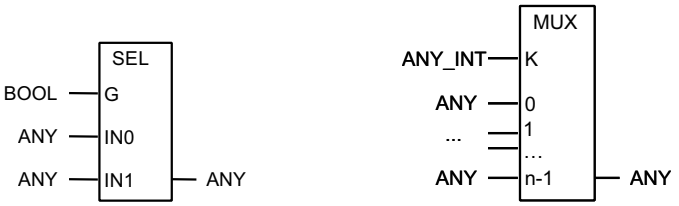



**Figure A.8.** Graphical declarations of the selection functions SEL and MUX

| Name | Function | Description |
|---|---|---|
| SEL | Binary selection | F := IN0, if G = 0, otherwise IN1 |
| MUX | Multiplexer | F := INi, if K = i and 0<=K<=n-1 |

**Table A.7.** Description of the selection functions SEL and MUX

These two standard functions are specified in the following examples by declaration in ST:

```
FUNCTION      SEL : ANY          (* binary selection *)
  VAR_INPUT
    G     :   BOOL;
    IN0   :   ANY;
    IN1   :   ANY;
  END_VAR
  IF G = 0 THEN
      SEL     :=  IN0;      (* selection of upper input *)
  ELSE
      SEL     :=  IN1;      (*selection of lower input *)
  END_IF;
END_FUNCTION
```

**Example A.3.** Specification of the selection function SEL in ST

```
FUNCTION      MUX : ANY          (* multiplexer *)
  VAR_INPUT
    K     :   ANY_INT;
    IN0   :   ANY;
    IN1   :   ANY;
    ...
    Inn-1 :   ANY;
  END_VAR
  IF (K < 0) OR (K>n-1) THEN
      ... error message ....  ;          (* K negative or too large *)
  END_IF;
  CASE  K  OF
      0: MUX := IN0;         (* selection of upper input *)
      1: MUX := IN1;         (* selection of second input *)
      ...
      n-1: MUX := INn;       (* selection of lowest input *)
  END_CASE;
END_FUNCTION
```

**Example A.4.** Specification of the selection function MUX in ST

## A.8 Comparison Functions

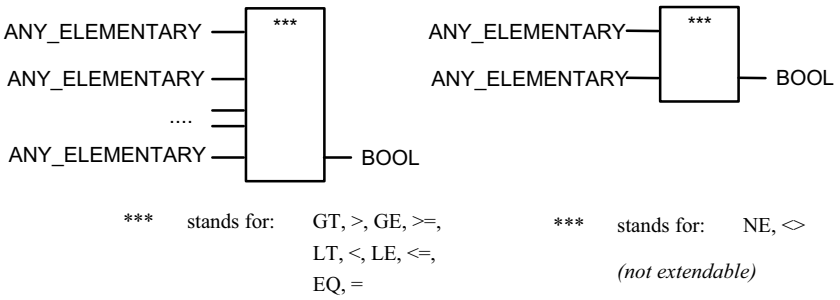



**Figure A.9.** Graphical declarations of the comparison functions GT, GE, LT, LE, EQ, NE

| Name | Function | Description |
|---|---|---|
| GT | Comparison for „> “ | F := 1, if INi > IN(i+1), otherwise 0 |
| GE | Comparison for„> =“ | F := 1, if INi >= IN(i+1), otherwise 0 |
| LT | Comparison for„< “ | F := 1, if INi < IN(i+1), otherwise 0 |
| LE | Comparison for„< =“ | F := 1, if INi <= IN(i+1), otherwise 0 |
| EQ | Comparison for„= “ | F := 1, if INi = IN(i+1), otherwise 0 |
| NE | Comparison for„<>“ | F := 1, if INi <> IN(i+1), otherwise 0 |

**Table A.8.** Description of the comparison functions

The specification of these standard functions is illustrated in Example A.5 by the declaration of GT in ST, from which the others can easily be derived.

```
FUNCTION      GT : BOOL          (* comparison for 'greater than' *)
  VAR_INPUT   IN1, IN2, ... INn  : ANY_ELEMENTARY;        END_VAR
  IF      (IN1 > IN2)
  AND     (IN2 > IN3)
  ...
  AND     (IN(n-1) > INn) THEN
      GT :=  TRUE;        (* inputs are "sorted": in increasing monotonic sequence *)
  ELSE
      GT :=  FALSE;       (* condition not fulfilled *)
  END_IF;
END_FUNCTION
```

**Example A.5.** Specification of the comparison function GT in ST

## A.9 Character String Functions

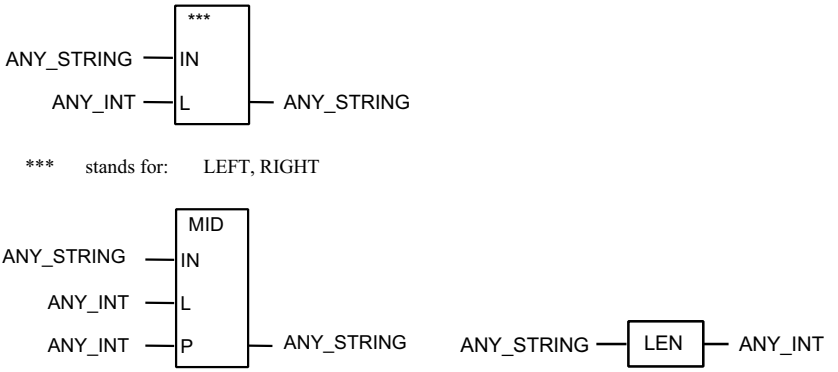



**Figure A.10.** Graphical declarations of the character string functions LEFT, RIGHT, MID and LEN

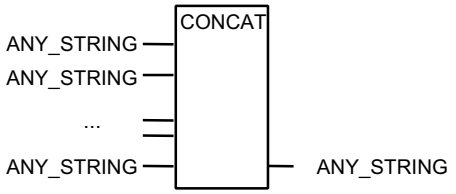



**Figure A.11.** Graphical declaration of the character string function CONCAT

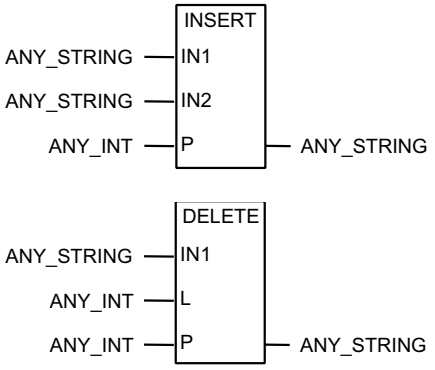



**Figure A.12.** Graphical declarations of the character string functions INSERT and DELETE

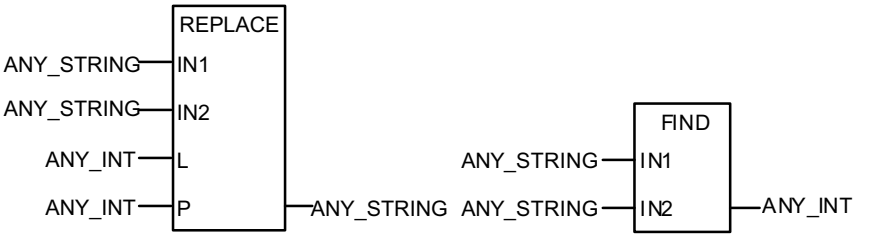



**Figure A.13.** Graphical declarations of the character string functions REPLACE and FIND

| Name | Function | Description |
|---|---|---|
| LEN | Determines the length of a character string | F := number of the characters in IN |
| LEFT | Starting section of a character string | F := starting section with L characters |
| RIGHT | Final section of a character string | F := final section with L characters |
| MID | Central section of a character string | F := middle section from position P with L characters |
| CONCAT | Sequence of character strings | F := total character string |
| INSERT | Inserts one character string into another one | F := total character string with new part from position P |
| DELETE | Deletes section in a character string | F := remaining character string with deleted part (L characters) from position P |
| REPLACE | Replaces a section of a character string with another one | F := total character string with replaced part (L characters) from position P |
| FIND | Determines the position of a section in a character string | F := index of the found position, otherwise 0 |

**Table A.9.** Description of the character string functions

The positions of the characters within a character string of type ANY_STRING are numbered starting with position "1".

Inputs L and P must be greater than or equal to zero.

## A.10 Functions for Time Data Types

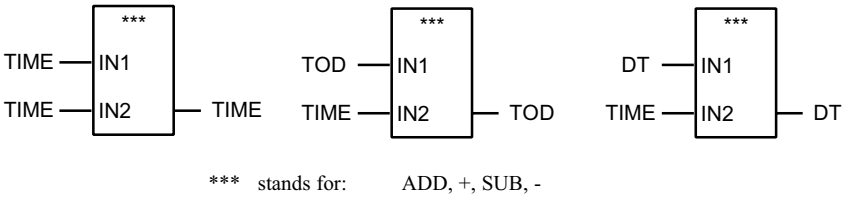



**Figure A.14.** Graphical declarations of the common functions for addition and subtraction of time

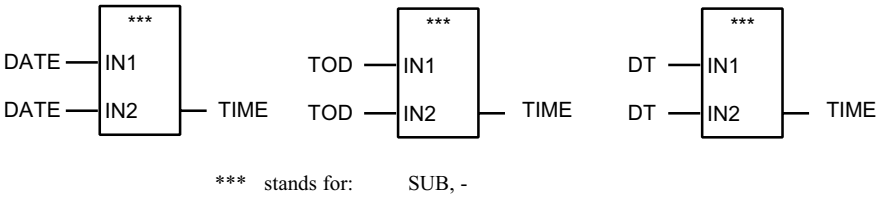



**Figure A.15.** Graphical declarations of the additional functions for subtraction of time

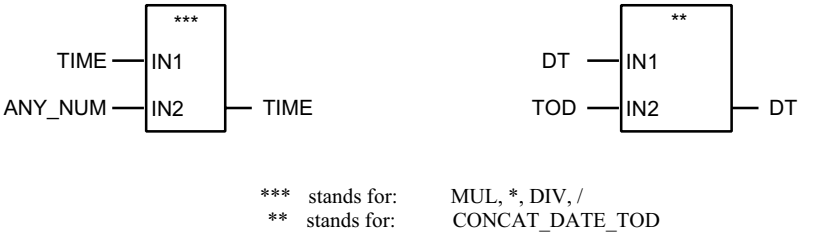



**Figure A.16.** Graphical declarations of the functions MUL, DIV and CONCAT_DATE_TOD for time

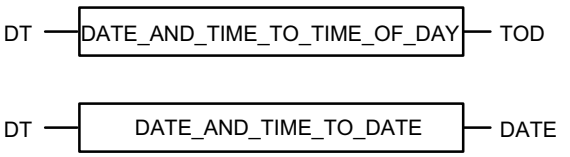



**Figure A.17.** Graphical declarations of the type conversion functions for time

The abbreviations TOD and DT can be employed as equivalents for the longer keywords TIME_OF_DAY and/or DATE_AND_TIME.
The function name versions ADD_TIME, SUB_TIME and the combinations ADD_TOD_TIME, ADD_DT_TIME, SUB_DATE_DATE, SUB_TOD_TIME, SUB_TOD_TOD, SUB_DT_TIME, MULTIME, DIVTIME and CONCAT_DATE_TOD can be used as equivalents for the function names ADD and SUB.
The standard explicitly advises against using the arithmetic symbols "+", "-", "/" and "*" and the function names ADD, SUB, DIV and MUL to prevent confusion and, as a result, incorrect use.

## A.11 Functions for Enumerated Data Types

The standard functions SEL, MUX, EQ and NE can also be employed in some cases for enumerated data types. They are then applied as for integers (values of enumerations correspond to constants "coded" by the programming system).

# B Standard Function Blocks

This appendix contains a complete overview of all the standard PLC function blocks described by means of examples in Chapter 2. For every standard function block of IEC 61131-3, the following information is given:

- Graphical declaration
- (Semantic) description
- Specification of some function blocks in Structured Text (ST).

In this book, the inputs and outputs of the standard FBs are given the names prescribed by the current version of IEC 61131-3. No allowance has been made for the fact that the possibility of calling FBs as "IL operators" can result in conflicts.

These conflicts occur with the IL operators (see also Section 4.1) LD, R and S, which need to be distinguished from the FB inputs LD, R and S when checking for correct program syntax. Depending on the programming system, these three formal operands could, for example, be called LOAD, RESET and SET in future.

## B.1 Bistable Elements (Flip-Flops)

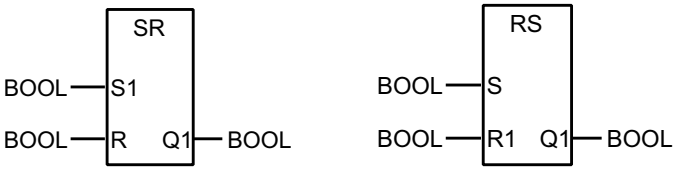



**Figure B.1.** Graphical declarations of the bistable function blocks SR and RS

```
FUNCTION_BLOCK      SR          (* flip-flop set dominant *)
  VAR_INPUT
    S1    :    BOOL;
    R     :    BOOL;
  END_VAR
  VAR_OUTPUT
    Q1    :    BOOL;
  END_VAR
  Q1      :=   S1 OR ( NOT R AND Q1);
END_FUNCTION_BLOCK

FUNCTION_BLOCK      RS          (* flip flop reset dominant *)
  VAR_INPUT
    S     :    BOOL;
    R1    :    BOOL;
  END_VAR
  VAR_OUTPUT
    Q1    :    BOOL;
  END_VAR
  Q1      :=   NOT R1 AND ( S OR Q1);
END_FUNCTION_BLOCK
```

**Example B.1,** Specification of the bistable function blocks SR and RS in ST

These two flip-flops implement dominant setting and resetting.

## B.2 Edge Detection

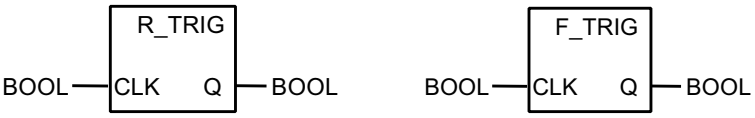



**Figure B.2.** Graphical declarations of the function blocks R_TRIG and F_TRIG

```
FUNCTION_BLOCK      R_TRIG     (* rising edge *)
  VAR_INPUT
    CLK  :    BOOL;
  END_VAR
  VAR_OUTPUT
    Q    :    BOOL;
  END_VAR
  VAR RETAIN                              (* example use of retentive variable *)
    MEM :     BOOL := 0;                  (* initialise edge flag *)
  END_VAR
  Q       :=  CLK AND NOT MEM;            (* recognise rising edge *)
  MEM     :=  CLK;                        (* reset edge flag *)
END_FUNCTION_BLOCK

FUNCTION_BLOCK      F_TRIG     (* falling edge *)
  VAR_INPUT
    CLK  :    BOOL;
  END_VAR
  VAR_OUTPUT
    Q    :    BOOL;
  END_VAR
  VAR RETAIN
    MEM :     BOOL := 1;                  (* initialise edge flag *)
  END_VAR
  Q       :=  NOT CLK AND NOT MEM;        (* recognise falling edge *)
  MEM     :=  NOT CLK;                    (* reset edge flag *)
END_FUNCTION_BLOCK
```

**Example B.2.** Specification of the function blocks R_TRIG and F_TRIG in ST

In the case of the function blocks R_TRIG and F_TRIG, it should be noted that they detect an "edge" on the **first** call if the input of R_TRIG is TRUE or the input of F_TRIG is FALSE.

## B.3 Counters

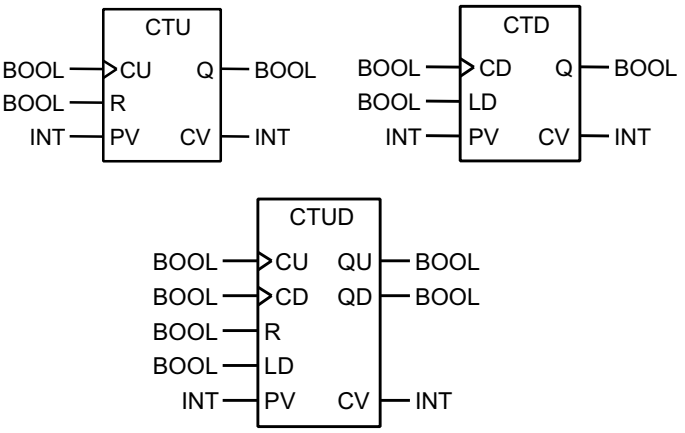



**Figure B.3.** Graphical declarations of the function blocks CTU, CTD and CTUD

```
FUNCTION_BLOCK          CTU       (* up counter *)
  VAR_INPUT
    CU    :    BOOL    R_EDGE;    (* CU with rising edge *)
    R     :    BOOL;
    PV    :    INT;
  END_VAR
  VAR_OUTPUT
    Q     :    BOOL;
    CV    :    INT;
  END_VAR
  IF  R  THEN                      (* reset counter *)
        CV      :=   0;
  ELSIF  CU AND ( CV < PV)  THEN
        CV      :=   CV + 1;       (* count up *)
  ENDIF;
  Q     :=     (CV >= PV);         (* limit reached *)
END_FUNCTION_BLOCK
```

**Example B.3.** Specification of the function blocks CTU and CTD in ST (continued on next page)

When used as versions for DINT, LINT, UDINT, ULINT, the counters CTU and CTD are accordingly designated CTU_DINT etc. up to CTD_ULINT.

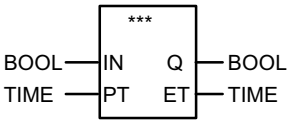
```
FUNCTION_BLOCK          CTD       (* down counter *)
  VAR_INPUT
    CD    :    BOOL     R_EDGE;   (* CD with falling edge *)
    LD    :    BOOL;
    PV    :    INT;
  END_VAR
  VAR_OUTPUT
    Q     :    BOOL;
    CV    :    INT;
  END_VAR
  IF  LD  THEN                     (* reset counter *)
        CV       :=   PV;
  ELSIF  CD AND ( CV > 0)  THEN
        CV       :=   CV - 1;     (* count down *)
  ENDIF;
  Q     :=     (CV <= 0);          (* zero reached *)
END_FUNCTION_BLOCK
```

**Example B.3.** (Continued)

```
FUNCTION_BLOCK          CTUD      (* up-down counter *)
  VAR_INPUT
    CU    :    BOOL     R_EDGE;   (* CU with rising edge *)
    CD    :    BOOL     R_EDGE;   (* CD with falling edge *)
    R     :    BOOL;
    LD    :    BOOL;
    PV    :    INT;
  END_VAR
  VAR_OUTPUT
    QU    :    BOOL;
    QD    :    BOOL;
    CV    :    INT;
  END_VAR
  IF  R  THEN                      (* reset counter (reset dominant) *)
        CV    :=     0;
  ELSIF  LD  THEN
        CV    :=     PV;          (* set to count value *)
  ELSE
    IF NOT (CU AND CD) THEN
      IF  CU AND ( CV < PV)  THEN
        CV    :=     CV + 1;      (* count up *)
      ELSIF  CD AND ( CV > 0)  THEN
        CV    :=     CV - 1;      (* count down *)
      ENDIF;
   ENDIF;
  ENDIF;
  QU    :=     (CV >= PV);        (* limit reached *)
  QD    :=     (CV <= 0);         (* zero reached *)
END_FUNCTION_BLOCK
```

**Example B.4.** Specification of the function block CTUD in ST

## B.4 Timers




*** stands for: TON, T---0, TOF, 0---T, TP

**Figure B.4.** Graphical declarations of the function blocks TON, TOF and TP

The timers TP, TON and TOF are specified here using timing diagrams.

This time behaviour is only possible if the cycle time of the cyclic PLC program in which the timer is used is negligibly small in comparison with the duration PT if the timer is called only once in the cycle.

The diagrams show the behaviour of outputs Q and ET depending on input IN. The time axis runs from left to right and is labelled "t". The Boolean variables IN and Q change between "0" and "1" and the time value ET increases as shown.

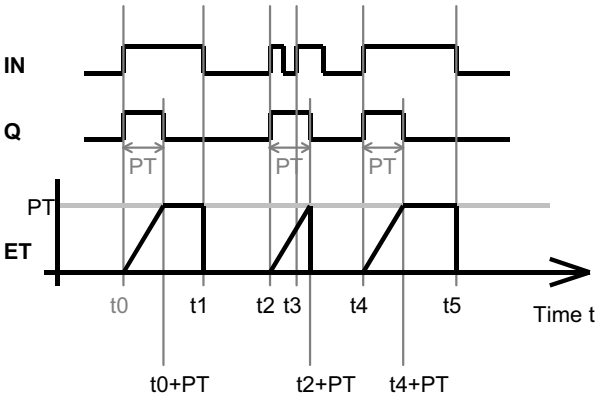



**Figure B.5**. Timing diagram for pulse timer TP depending on input IN

The standard FB "TP" acts as a pulse generator which supplies a pulse of constant length at output Q when a rising edge is detected at input IN. The time that has elapsed so far can be read off at output ET at any time.

As can be seen from Figure B.5, timers of type TP are not "retriggerable". If the intervals between the input pulses at IN are shorter than the pre-set time period, the pulse duration still remains constant (see period [t2; t2+PT]). Timing therefore does **not** begin again with every rising edge at IN.

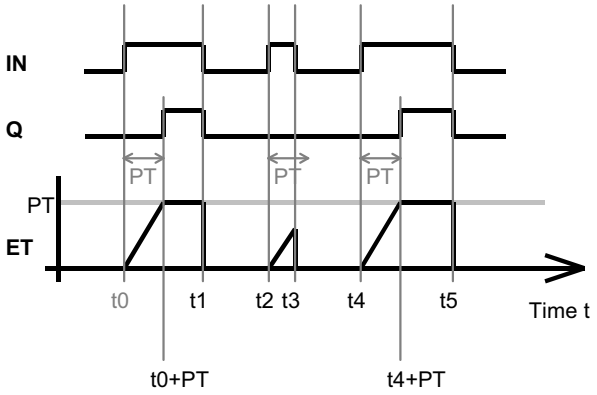



**Figure B.6.** Timing diagram for on-delay timer TON depending on input IN

The on-delay timer TON supplies the input value IN at Q with a time delay when a rising edge is detected at IN. If input IN is "1" only for a short pulse (shorter than PT), the timer is not started for this edge.

The elapsed time can be read off at output ET.

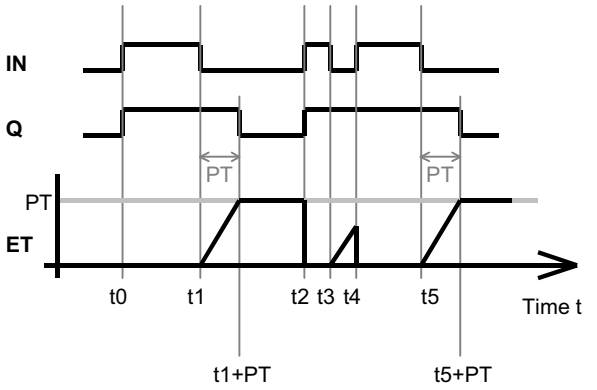



**Figure B.7.** Timing diagram for off-delay timer TOF depending on input IN

The off-delay timer performs the inverse function to TON i.e. it delays a falling edge in the same way as TON delays a rising one.

The behaviour of the timer TOF if PT is modified during timer operation is implementation-dependent.

# C IL Examples

This appendix contains full examples of PLC programming with IEC 61131-3 for each type of POU, to supplement the information given in Chapters 2 and 4.

These examples are to be found on the CD enclosed in this book.

## C.1 Example of a FUNCTION

The function ByteExtr extracts the upper or lower byte of an input word and returns it as the function value:

```
FUNCTION  ByteExtr : BYTE    (* extract byte from word *)
                             (* beginning of declaration part *)
VAR_INPUT                    (* input variables *)
  DbByte : WORD;             (* word consists of upper + lower byte *)
  Upper  : BOOL;             (* TRUE: take upper byte, else lower *)
END_VAR
                             (* beginning of instruction part *)
LD       Upper               (*extract upper or lower byte? *)
EQ       FALSE               (* lower? *)
JMPCN    UpByte              (* jump in the case of extraction of the upper byte *)
LD       DbByte              (* load word *)
WORD_TO_BYTE                 (* conversion for type compatibility *)
ST       ByteExtr            (* assignment to the function value *)
RET                          (* nothing to do *)
UpByte:                      (* jump label *)
LD       DbByte              (* load word *)
SHR      8                   (* shift upper byte 8 bits to the right *)
WORD_TO_BYTE                 (* conversion for type compatibility *)
ST       ByteExtr
RET                          (* return with function value in CR *)
                             (* end of FUN *)
END_FUNCTION
```

**Example C.1.** Example of the declaration of a function in IL

The ByteExtr function in Example C.1 has the input parameter DbByte of type WORD and the Boolean input Upper. The value returned by the function is of type BYTE. This function requires no local variables. The VAR ... VAR_END section is therefore missing from the declaration part.

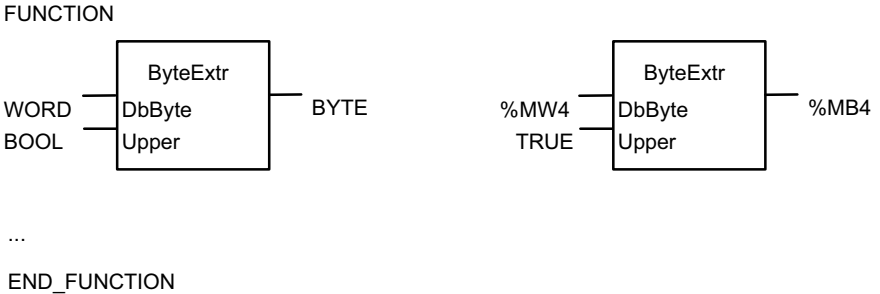
The returned value is in the current result (CR) when the function returns to the calling routine with RET. At this point (jump label End:) the CR is of data type WORD, because DbByte was previously loaded. The function value is of data type BYTE after type conversion.

IEC 61131-3 always requires a strict "type compatibility" in cases like this. It is the job of the programming system to check this consistently. This is why a standard type conversion function (WORD_TO_BYTE) is called in Example C.1.

Example C.2 shows the instruction part of ByteExtr in the ST programming language.

```
FUNCTION  ByteExtr : BYTE     (* extraction byte from word *)
VAR_INPUT ... END_VAR         (* as above *)
IF Upper THEN
  ByteExtr := WORD_TO_BYTE (SHR (DbByte, 8) );
ELSE
  ByteExtr := WORD_TO_BYTE (DbByte);
END_IF;
END_FUNCTION
```

**Example C.2.** Instruction part of Example C.1 in ST




**Example C.3.** Graphical declaration part of the function declaration in Example C.1 (left) with an example of a call (right)

Example C.3 shows the declaration part and an example of a call for the function ByteExtr in graphical representation. The call replaces the lower byte of flag word %MW4 by its upper byte.

## C.2 Example of a FUNCTION_BLOCK

The function block DivWithRem calculates the result of dividing two integers and returns both the division result and the remainder. "Division by zero" is indicated by an output flag.

```
FUNCTION_BLOCK DivWithRem   (* division with remainder *)
                            (* beginning of declaration part *)
VAR_INPUT                   (* input parameter *)
  Dividend :    INT;        (* integer to be divided *)
  Divisor  :    INT;        (* integral divisor *)
END_VAR
VAR_OUTPUT RETAIN           (* retentive output parameters *)
  Quotient :    INT;        (* result of the division *)
  DivRem   :    INT;        (* remainder after division *)
  DivError :    BOOL;       (* flag for division by zero *)
END_VAR

                            (* beginning of instruction part *)
LD      0                   (* load zero *)
EQ      Divisor             (* divisor equal to zero? *)
JMPC    Error               (* catch error condition *)
LD      Dividend            (* load dividend, divisor not equal to zero *)
DIV     Divisor             (* carry out division *)
ST      Quotient            (* store integral division result *)
MUL     Divisor             (* multiply division result by divisor *)
ST      DivRem              (* store interim result *)
LD      Dividend            (* load dividend *)
SUB     DivRem              (* subtract interim result *)
ST      DivRem              (* yields "remainder" of the division as an integer*)
LD      FALSE               (* load logical "0“ for error flag *)
ST      DivError            (* reset error flag *)
JMP     End                 (* ready, jump to end *)
Error:                      (* handling routine for error "division by zero" *)
LD      0                   (* zero, since outputs are invalid in event of error *)
ST      Quotient            (* reset Result *)
ST      DivRem              (* reset Remainder *)
LD      TRUE                (* load logical "1“ for error flag *)
ST      DivError            (* set error flag *)
End:
RET

                            (* end of FB *)
END_FUNCTION_BLOCK
```

**Example C.4.** Example of the declaration of a function block in IL

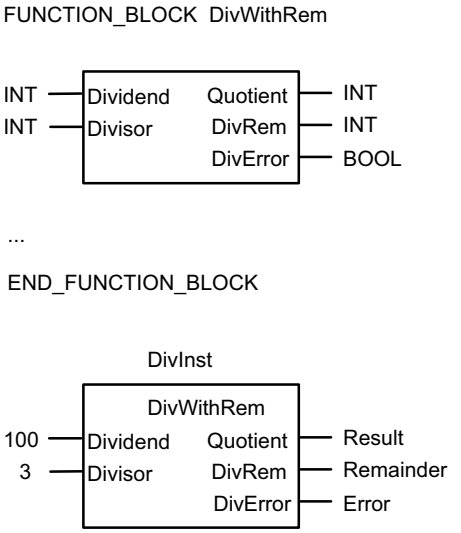
The FB DivWithRem in Example C.4 performs integer division with remainder on the two input parameters Dividend and Divisor. In the case of division by zero, the error output DivError is set and the other two outputs are set in a defined manner to zero since they are invalid. The outputs are retentive, i.e. they are retained within the FB instance from which DivWithRem was called.

This example can also be formulated as a function because no statistical information has to be retained between calls.

Example C.5 shows the instruction part of DivWithRem in the ST programming language.

```
FUNCTION_BLOCK DivWithRem    (* division with remainder *)
VAR_INPUT ... END_VAR        (* as above *)
VAR_OUTPUT RETAIN ... END_VAR
IF Divisor = 0 THEN
  Quotient := 0;
  DivRem := 0;
  DivError := TRUE;
ELSE
  Quotient := Dividend / Divisor;
  DivRem := Dividend - (Quotient * Divisor);
  DivError := FALSE;
END_IF;
END_FUNCTION_BLOCK
```

**Example C.5.** Instruction part of Example C.4 in ST



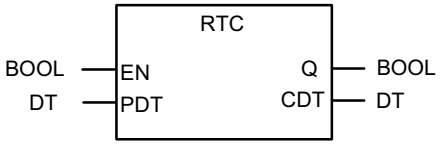

**Example C.6.** Graphical declaration part for Example C.4 (top) and example of a call of the FB instance DivInst (bottom)

Example C.6 shows the declaration part and an example of a call of the function DivWithRem in graphical representation. The FB must be instantiated (DivInst) before it can be called.
After execution of the function block, the output variables of this instance have the following values: DivInst.Quotient = 33, DivInst.DivRem = 1 and DivInst.DivError = FALSE. These return values are assigned to the variables Result, Remainder and Error respectively.

## C.3 Example of a PROGRAM

The program MainProg in Example C.8 is not a complete programming example, but shows ways of implementing problems and illustrates the use of variables in POUs of type PROGRAM.

MainProg first starts a real-time clock DateTime that records the date and time by using function block RTC. This function block can be either user specific or pre-defined by manufacturer – since edition 2 of the standard IEC 1131-3 this RTC is not avaliable as a standard FB any longer. Example C.7 shows one possible representation as example. This real-time clock RTC is used to find out how long an interruption in the program has lasted (TimeDiff). The PLC system must be able to detect the interruption (Ress_Running) and must be able to access a hardware clock with an I/O address (ActDateTime).




**Example C.7.** Graphical representation of example RTC, which is not a standard FB of IEC 1131-3. Rising edge at EN loads the real-time-clock, CDT is current date and time while EN; Q is copy of EN.

```
PROGRAM      MainProg                  (* example of a main program *)
  VAR_INPUT                            (* input variables *)
    T_Start       : BOOL := FALSE;     (* input starting condition *)
  END_VAR
  VAR_OUTPUT                           (* output variables *)
    T_Failure     : BOOL := FALSE;     (* output "failure“ *)
  END_VAR
  VAR_GLOBAL RETAIN                    (* global retentive data area *)
    Ress_Running  AT %MX255.5 : BOOL; (* running flag for resource/PLC-CPU *)
    DateTime      : RTC;               (* use buffered clock: date and time *)
    ActDateTime AT %MD2 : DT;          (* hardware clock: actual date with time *)
  END_VAR
  VAR_GLOBAL                           (* global data area *)
    EmergOff  AT %IX255.0 : BOOL;      (* contact Emergency-Off *)
    ProgRun       : BOOL := FALSE;     (* "running“ flag *)
    Error         : BOOL;              (* error flag *)
    Err_Code      : UDINT := 0;        (* Error code, 32 bit unsigned *)
  END_VAR
  VAR                                  (* local variables *)
    AT %IX250.2 :  BOOL;               (* directly represented variables *)
    ErrorProc     : CErrProc;          (* FB instance of CErrProc *)
    Edge          : R_TRIG;            (* edge detection *)
    TimeDiff      : TIME := t#0s;      (* time difference *)
  END_VAR
```

**Example C.8.** Example of the declaration of a main program in IL. The FB CErrProc ("central error processing") must already be available (continued on next page)

```
                                    (* beginning of instruction part *)
LD      FALSE
ST      Error                       (* reset error flag *)

...                         (* determine how long a power failure and/or an interruption *)
                              (* of the CPU lasted; Clock "DateTime" is battery-backed *)
LD      t#0s                        (* zero seconds *)
ST      TimeDiff                    (* reset *)
LD      DateTime.Q                  (* time valid = clock running *)
JMPC    Goon                        (* valid - nothing to do *)
LD      ActDateTime                 (* check current time *)
SUB     DateTime.CDT                (* last time before power failure *)
ST      TimeDiff                    (* duration of power failure *)
Goon:
...
LD      Ress_Running                (* CPU is running *)
ST      DateTime.IN                 (* clock running if CPU is running *)
LD      ActDateTime                 (* initial value Date/Time *)
ST      DateTime.PDT                (* initial value of clock is loaded *)
                                    (* if there is a rising edge at IN *)
CAL     DateTime                    (* start real-time clock)
...
LD      TimeDiff                    (* interruption duration *)
LT      t#50m                       (* less than 50 seconds? *)
JMPC    Continue                    (* plant still warm enough ... *)
NOT
S       Error                       (* set error flag *)
LD      16#000300F2
ST      Err_Code                    (* store error cause *)
Continue:
LD      EmergOff                    (* "emergency off“ pressed? *)
ST      Edge.CLK                    (* input edge detection *)
CAL     Edge                        (* compare input with edge flag *)
LDN     Edge.Q                      (* edge at EmergOff recognized? *)
AND     T_Start                     (* AND start flag set *)
ANDN    Error                       (* AND no new error *)
AND     ErrorProc.Ack               (* AND error cause repaired *)
ST      ProgRun                     (* global "running“ condition *)

....                        (* ...real instructions, calls of FUN/FBs... *)

LD      Error                       (* error occurred? *)
AND     %IX250.2
R       ProgRun                     (* reset global starting condition *)
LD      ProgRun
JMPCN End                           (* in the case of error: start error handler *)
CALC    ErrorProc(Code := Err_Code) (* FB with central error handling *)
LD      ErrorProc.Ack               (* flag: error acknowledged *)
End:
LD      ProgRun                     (* program not running *)
ST      T_Failure                   (* set output parameter *)
RET                                 (* return and/or end *)

END_PROGRAM
```

**Example C.8.** (Continued)

Edge detection is also employed in this program in order to find out whether the Emergency Off button has been pressed.

In the global data area, the variable ProgRun is declared, which is available to all function blocks called under MainProg (as VAR_EXTERNAL). This is linked with the starting condition provided EmergOff has not been pressed.

FB instance ErrProc can handle errors with error code Err_Code. When the error has been corrected and acknowledged, the corresponding output is set to TRUE.

```
RESOURCE CentralUnit_1 ON CPU_001
  TASK Periodic (INTERVAL := time#13ms, PRIORITY := 1);
  PROGRAM Applic WITH Periodic : MainProg (    T_Start := %I250.0,
                                               T_Failure => %Q0.5);
END_RESOURCE
```

**Example C.9.** Resource definition with run-time program Applic for Example C.8

The program MainProg is provided with the properties of the periodic task Periodic and becomes the run-time program Applic of the resource (PLC-CPU) CentralUnit_1. This run-time program runs with highest priority (1) as a periodic task with a maximum cycle time of 13 ms.

MainProg is called with the value of input bit %I250.0 for the input variable T_Start and sets output bit %Q0.5 with the value of T_Failure.

# D Standard Data Types

This appendix summarises all the elementary data types and their features in tabular form. Their use is explained in Chapter 3.

IEC 61131-3 defines five groups of elementary data types. Their generic data types are indicated in brackets (see Table 3.9):

- Bit string (ANY_BIT),
- Integer, signed and unsigned (ANY_INT),
- Real (floating-point) (ANY_REAL),
- Date, time-of-day (ANY_DATE),
- Character string, duration (ANYSTRING, TIME).

The following information is given for each data type, listed in separate tables for each group:

- Name (keyword),
- Description,
- Number of bits (data width),
- Value range (using the IEC literals),
- Initial values (default values).

The data width and permissible range of the data types in Tables D.5 and D.6 are application-dependent.

| Data type | Description | Bits | Range | Initial |
|---|---|---|---|---|
| BOOL | Boolean | 1 | [0,1] | 0 |
| BYTE | Bit string 8 | 8 | [0,...,16#FF] | 0 |
| WORD | Bit string 16 | 16 | [0,...,16#FFFF] | 0 |
| DWORD | Bit string 32 | 32 | [0,...,16#FFFF FFFF] | 0 |
| LWORD | Bit string 64 | 64 | [0,...,16#FFFF FFFF FFFF FFFF] | 0 |

**Table D.1.** “Boolean and Bit String” data types

| Data type | Description | Bits | Range | Initial |
|---|---|---|---|---|
| SINT | Short integer | 8 | [-128,...,+127] | 0 |
| INT | Integer | 16 | [-32768,...,+32767] | 0 |
| DINT | Double integer | 32 | [$-2^{31}$,...,+ $2^{31}$-1] | 0 |
| LINT | Long integer | 64 | [$-2^{63}$,...,+ $2^{63}$-1] | 0 |

**Table D.2. .** "Signed Integer" data types

| Data type | Description | Bits | Range | Initial |
|---|---|---|---|---|
| USINT | Unsigned short integer | 8 | [0,...,+255] | 0 |
| UINT | Unsigned integer | 16 | [0,...,+65535] | 0 |
| UDINT | Unsigned double integer | 32 | [0,...,+$2^{32}$-1] | 0 |
| ULINT | Unsigned long integer | 64 | [0,...,+$2^{64}$-1] | 0 |

**Table D.3.** "Unsigned Integer" data types

| Data type | Description | Bits | Range | Initial |
|---|---|---|---|---|
| REAL | Real numbers | 32 | See IEC 559 | 0.0 |
| LREAL | Long reals | 64 | See IEC 559 | 0.0 |

**Table D.4.** "Real Numbers" data types (floating-point numbers)

| Data type | Description | initial |
|---|---|---|
| DATE | Date (only) | d#0001-01-01 |
| TOD | Time of day (only) | tod#00:00:00 |
| DT | Date and time of day | dt#0001-01-01-00:00:00 |

**Table D.5.** "Date and Time" data types

The full keyword TIME_OF_DAY can also be used in place of TOD, and DATE_AND_TIME in place of DT.

| Data type | Description | Initial |
|---|---|---|
| TIME | Duration | t#0s |
| STRING | Character string (variable length) | '' |
| WSTRING | Character string (double bytes) | "" |

**Table D.6.** "Duration and Character String" data types. The initial values for STRING and WSTRING are "empty" character strings.

# E Causes of Error

IEC 61131-3 requires manufacturers to provide a list of responses to the following error conditions. See also Section 7.9. The responses fall into four different categories:

1) No system response (%),
2) Warning during program creation (WarnPc),
3) Error message during program creation (ErrPc),
4) Error message and response to error at run time (ErrRun).

| No. | Error cause | Response |
|---|---|---|
| 1 | Nested comment | ErrPc |
| 2 | Ambiguous value of enumeration type | ErrPc |
| 3 | Value of a variable exceeds the specified range. | ErrRun |
| 4 | Incomplete address resolution in configuration ("*") | ErrPc |
| 5 | Write access to variable declared as CONSTANT | ErrPc |
| 6 | Variable declared as VAR_GLOBAL CONSTANT, but referenced as VAR_EXTERNAL without CONSTANT. | ErrPc |
| 7 | Reference to a directly represented or external variable in a function | ErrPc |
| 8 | Impermissible assignment to VAR_IN_OUT variable (e.g. to a constant) | ErrPc |
| 9 | Ambiguous assignment to VAR_IN_OUT variable | ErrPc |
| 10 | Type conversion errors | ErrPc |
| 11 | Numerical result (of a standard function) exceeds the range for the data type;<br>Division by zero (in a standard function). | ErrRun<br>ErrPc, ErrRun |
| 12 | Bit shift function with negative shift number | ErrRun |
| 13 | Mixed input data types to a selection function (standard function);<br>Selector (K) out of range for MUX function. | ErrPc<br>ErrRun |

**Table E.1.** Error causes. The manufacturer supplies a table specifying the system response to the errors described above. The entry in the Response column marks the appropriate instant for the response in accordance with IEC 61131-3. (Continued on next page)

| | | |
|---|---|---|
| 14 | Character sequence functions:<br>Invalid character position specified;<br>Result exceeds maximum string length (INSERT/CONCAT result is too long).<br>ANY_INT input is negative. | <br>WarnPc, ErrRun<br>WarnPc, ErrRun<br><br>ErrRun |
| 15 | Result exceeds range for data type "Time" | ErrRun |
| 16 | An FB instance used as an input parameter has no parameter values. | ErrPc |
| 17 | A VAR_IN_OUT parameter has no value. | ErrPc |
| 18 | Zero or more than one initial steps in SFC network;<br>User program attempts to modify step state or time. | ErrPc<br>ErrPc |
| 19 | Side effects in evaluation of transition conditions. | ErrPc |
| 20 | Action control contention error. | WarnPc, ErrRun |
| 21 | Simultaneously true, non-prioritised transitions in a SFC selection divergence. | WarnPc, ErrPc |
| 22 | Unsafe or unreachable SFC. | WarnPc |
| 23 | Data type conflict in VAR_ACCESS. | ErrPc |
| 24 | Tasks require too many processor resources;<br>Execution deadline not met. | ErrPc<br>WarnPc, ErrRun |
| 25 | Numerical result exceeds range for data type (IL). | WarnPc, ErrRun |
| 26 | Current result (CR) and operand type do not match (IL). | ErrPc |
| 27 | Division by zero (ST);<br><br>Invalid data type for operation (ST).<br>Numerical result exceeds range for data type (ST). | WarnPc,ErrPc, ErrRun<br>ErrPc<br>ErrRun |
| 28 | Return from function without value assigned (ST). | ErrPc |
| 29 | Iteration fails to terminate (ST). | WarnPc,ErrPc, ErrRun |
| 30 | Same identifier used as connector label and element name (LD / FBD). | ErrPc |
| 31 | Uninitialised feedback variable | ErrPc |

**Table E.1.** (Continued)

This table is intended as a guide. There are other possible errors which are not included in the IEC table. It should therefore be extended by every manufacturer as appropriate.

# F Implementation-Dependent Parameters

Table F.1 lists the implementation-dependent parameters defined by IEC 61131-3. See also Section **7.**10.

| No. | Implementation-dependent parameters |
|---|---|
| 1 | Maximum length of identifiers. |
| 2 | Maximum comment length (without leading or trailing brackets). |
| 3 | Pragma syntax and semantics. |
| 4 | Syntax and semantics for ‘, unless used to declare double-byte strings. |
| 5 | Range of values for variables of data type TIME (e.g. 0<=TimeVar<1000 Days), DATE, TIME_OF_DAY and DATE_AND_TIME.<br>Precision of representation of seconds in data types TIME_OF_DAY and DATE_AND_TIME. |
| 6 | Maximum:<br>- number of elements in an enumeration data type,<br>- number of elements in an array (number of array subscripts),<br>- array size (number of bytes),<br>- number of structure elements,<br>- structure size (number of bytes),<br>- number of variables per declaration that can be declared with the same data type (separated by commas),<br>- number of structure nesting depth. |

**Table F.1.** Implementation-dependent parameters which every manufacturer of an IEC 61131-3 system must describe. The division into individual groups relates to individual sections in the standard. If no concrete figures are required, the manufacturer can add an informal description. (Continued on next page)

| | |
|---|---|
| 7 | Default maximum length of STRING and WSTRING variables.<br>Maximum allowed length of STRING and WSTRING variables. |
| 8 | Maximum number of hierarchical levels for directly represented variables (e.g. %QX1.1.1.2...).<br>Logical or physical mapping (symbols %IX.... onto the real hardware). |
| 9 | Initialisation of system inputs (%IX...) at system start time. |
| 10 | Maximum number of variables per declaration block. |
| 11 | Behaviour of an AT identifier as part of an FB declaration. |
| 12 | Hot-restart behaviour, if variable was declared neither with RETAIN nor with NON_RETAIN. |
| 13 | Information to determine execution times of program organisation units (POU) (no details are given in IEC 61131-3). |
| 14 | Block output values, if block terminates with ENO=FALSE (yes/no). |
| 15 | Maximum number of function specifications (if limited). |
| 16 | Maximum number of inputs of extensible functions. |
| 17 | Effects of type conversions on accuracy (REAL_TO_INT, ...) and implemented error handling. |
| 18 | Accuracy of numerical functions. |
| 19 | Results of type conversions between TIME and other data types. |
| 20 | Maximum number of user function block specifications and instantiations (if limited). |
| 21 | Assignment behaviour of block inputs,if EN= FALSE (yes/ no). |
| 22 | Range of parameter PV (end value of counter function blocks). |
| 23 | System reaction to a change of the input PT (end value of timer function blocks) during operation. |
| 24 | Program size limitations (executable code). |
| 25 | Precision of step elapsed time; see Section 4.6.3 |
| 26 | Maximum number of steps per SFC network. |
| 27 | Maximum number of transitions per SFC network and per step. |
| 28 | Maximum number of action blocks per step. |
| 29 | Action control mechanism (if available) and accessibility to Q and A output values. |
| 30 | (Minimum) transition clearing time (caused by PLC cycle time). |
| 31 | Maximum number of predecessor and successor steps in diverge/converge constructs. |
| 32 | Contents of RESOURCE libraries. Every processing unit (e.g. processor type) receives a description of all functions (standard functions, standard FBs, data types, ....) that can be processed by this resource. |
| 33 | Effect of READ_WRITE access on FB outputs |

**Table F.1.** (Continued on next page)

| | |
|---|---|
| 34 | Maximum number of tasks / resources.<br>Task interval resolution (time between calls for periodic tasks);<br>Type of task priority control (pre-emptive or non-pre-emptive scheduling). |
| 35 | Maximum length of expressions (ST) (number of operands, operators). |
| 36 | Maximum length of statements (ST) (IF; CASE; FOR; ...); restrictions. |
| 37 | Maximum number of CASE selections (ST). |
| 38 | Value of control variable upon termination of FOR loop. |
| 39 | Restrictions on network topology (LD/FBD). |
| 40 | Evaluation order of feedback loops. |

**Table F.1.** (Continued)

There are a large number of other parameters that need to be considered when implementing a programming system compliant with IEC 61131-3.

Table F.2 gives a subjective selection of these parameters.

| No. | **Implementation-dependent parameters** |
|---|---|
| 1 | Extent of syntax and semantic checking provided by the textual and graphical editors (during input). |
| 2 | Description of the execution order of networks. |
| 3 | Free placement of graphic elements in the case of editors with line updating ("elastic band") or automatic placement according to syntax rules. |
| 4 | Declaration and use of Directly Represented Variables (DRVs):<br>- In the declaration the symbolic name is always used. In the code part *either* the symbol alone *or* both may be used (symbol; direct physical address). Or<br>- DRVs may also be declared without symbolic names (use of the direct physical address only). Or<br>- The programming system declares DRVs implicitly. |
| 5 | Sequence order of VAR_*...VAR_END blocks as well as multiple use of blocks with the same name (variable type). |
| 6 | Function calling mechanisms implemented in ST (with/without formal parameters). |
| 7 | Extent of implementation of the EN/ENO parameters in LD and FBD, and possible effects on IL/ST program sources. |

**Table F.2.** Other implementation-dependent parameters (not part of IEC 61131-3). (Continued on next page)

| | |
|---|---|
| 8 | Possibility of passing user-defined structures (TYPE) as function and FB input parameters (not defined in the standard at present). |
| 9 | Global and POU-wide publication of user-defined data types TYPE...END_TYPE: (not defined in the standard at present). |
| 10 | Extent of data area checking during program creation and at run time. |
| 11 | Graphical representation of qualifiers in variable declarations. |
| 12 | Restrictions on use of complex data types (function type also permitted for type "string" or user-defined types, ...?). |
| 13 | Algorithm for the evaluation of LD/FBD networks (see Sections 4.3.4 and **7.**3.1). |
| 14 | Aids for comprehensibility of cross-compiled programs (if implemented). |
| | ... |
| | |

**Table F.2.** (Continued)

IEC 61131-3 expressly allows functionality beyond the standard. However, they must be described. Table F.3 gives some examples.

| **No.** | **Extensions** |
|---|---|
| 1 | Extending range checking to more data types than only integer (ANY_INT), e.g. ANY_NUM. |
| 2 | Accepting FB instances as array variables. |
| 3 | Permitting FB instances in structures. |
| 4 | Allowing overloading for user-defined functions, function blocks and programs. |
| 5 | Time when the step width is calculated in the case of FOR statements. |
| 6 | Possible use of pre-processor statements for literals, macros, conditional compiling, Include statements (for input of files with FB interface information/prototypes, with the list of directly represented variables or EXTERNAL declarations employed, ...). |
| 7 | Use of different memory models in the PLC (Small; Compact; Large,...). |
| | ... |

**Table F.3.** Possible extensions (not part of IEC 61131-3)

# G IL Syntax Example

Many of the examples given in this book are formulated in Instruction List (IL). This programming language is widely used and is supported by most programming systems. By including data types previously only found in high-level languages, such as arrays or structures, the IEC 61131-3 Instruction List language opens up new possibilities compared with conventional IL.

The IL syntax descriptions in this appendix are presented in simplified form in *syntax diagrams*.

Syntax diagrams explain the permissible use of delimiters, keywords, literals and names in a readily comprehensible format. They can easily be put into textual form for the development of compilers and define the formal structure of a programming language *(syntax* of a language). The reader can use the diagrams for reference.

The syntax descriptions in this appendix go beyond IEC 61131-3 because, in addition to the pure syntax definitions, they also include semantic conditions (consistent use of the current result, use of function parameters, etc.). IEC 61131-3 only offers an informal description of this.

The rules are outlined in Section G.1. The use of the diagrams is explained in Section G.2 by means of an example.

## G.1 Syntax Diagrams for IL

If a node in the syntax diagram has further subdivisions (sub-diagram), its name appears in italics. Keywords or terminal symbols which are not further subdivided appear in standard type. For clarity the use of comments is not included. As of version 2 of the standard, a comment is permissible wherever blanks (except for character strings) can be placed. A comment begins with (*, ends with *) and contains any number of alphanumeric characters in between **without** EOL. Comments cannot be nested.

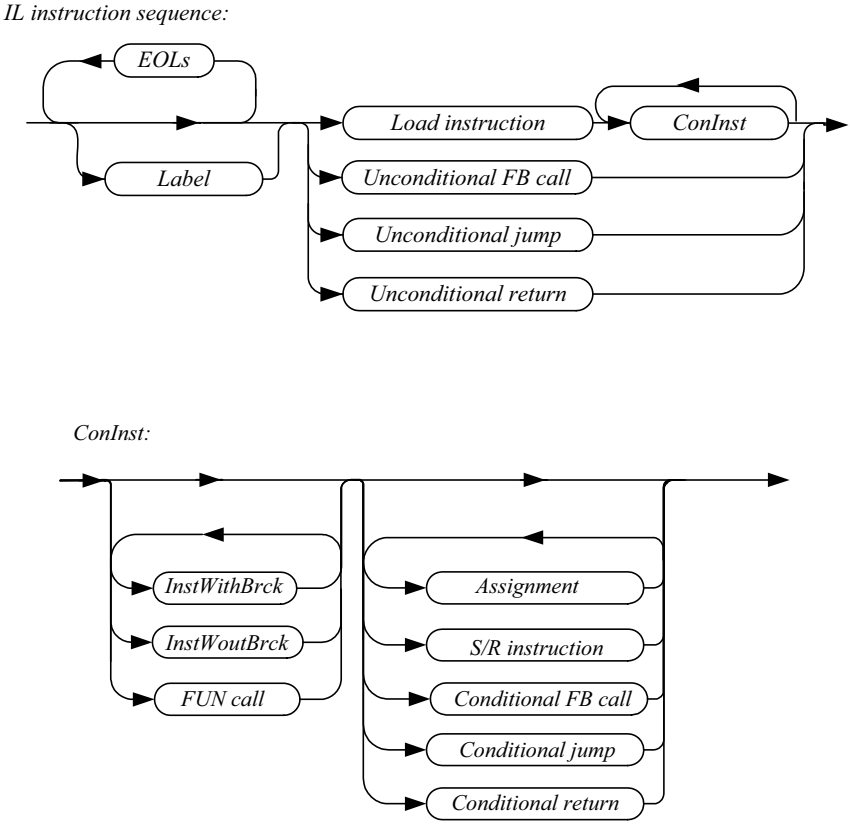



**Figure G.1.** Syntax diagrams of an IL instruction sequence with the sub-elements "conditional instruction" and "label" (continued on next page)

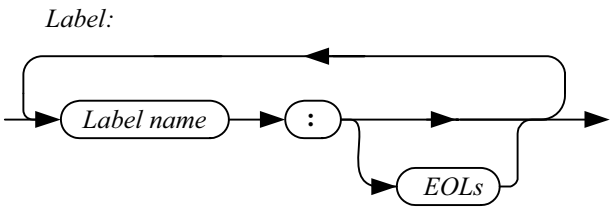



**Figure G.1.** (Continued)

An IL instruction sequence begins optionally with a (jump) label. This is followed either by a Load instruction followed by one or more conditional instructions ConInst, an unconditional instruction with FB call, a jump or a return (see syntax diagram in Figure G.1).

The conditional instruction begins with a sequence of instructions with and without brackets and/or function calls. This is followed by a series of (S/R) assignments, conditional calls or jumps.

The label consists of a label identifier, followed by a colon. It either immediately precedes the first instruction of the sequence or is followed by EOLs (end of line).

The syntax diagrams for instructions, calls and jumps are given below.

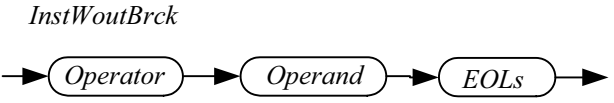



**Figure G.2.** Syntax diagram of an instruction without brackets

An instruction without brackets (Figure G.2) consists of an IL operator with one operand and the end-of-line EOLs.

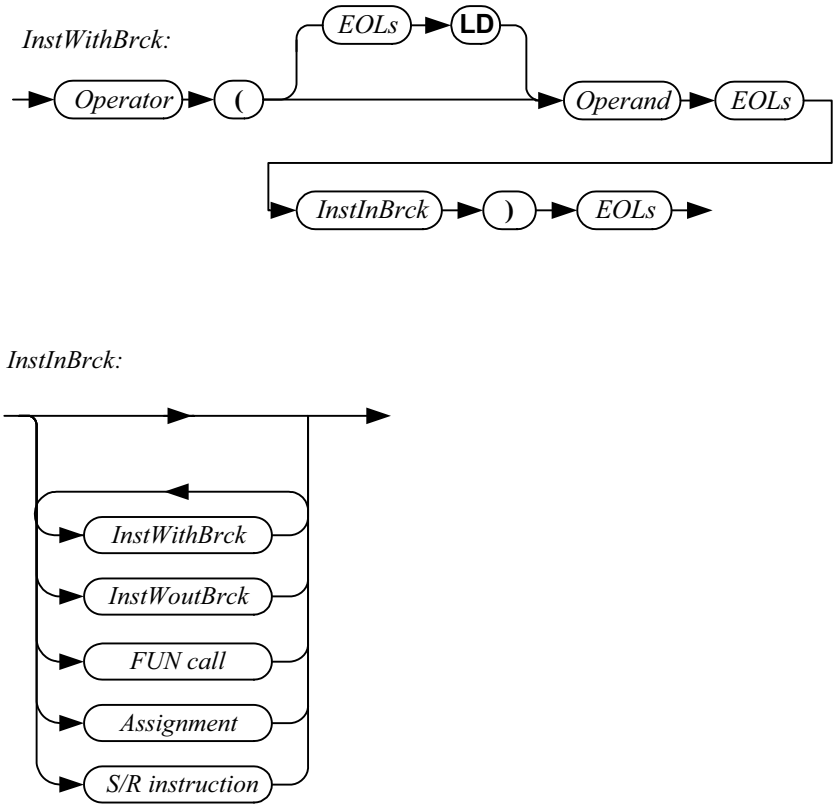



**Figure G.3.** Syntax diagrams of an instruction with brackets

The syntax diagrams in Figure G.3 show the structure of an instruction with brackets. This type of instruction begins with an instruction consisting of an operator followed by an opening bracket and an operand. This can be followed by any number of instructions InstInBck inside the brackets, which are concluded with a closing bracket and EOLs.

These inner instructions can themselves contain brackets (nesting), as well as FUN calls and assignments.

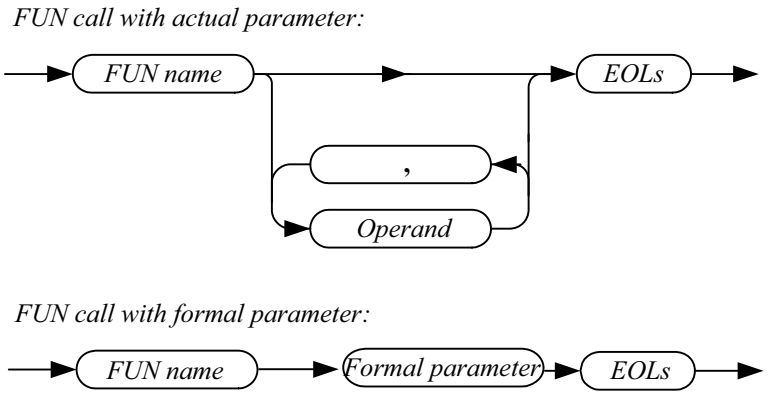



**Figure G.4.** Syntax diagram for a function call

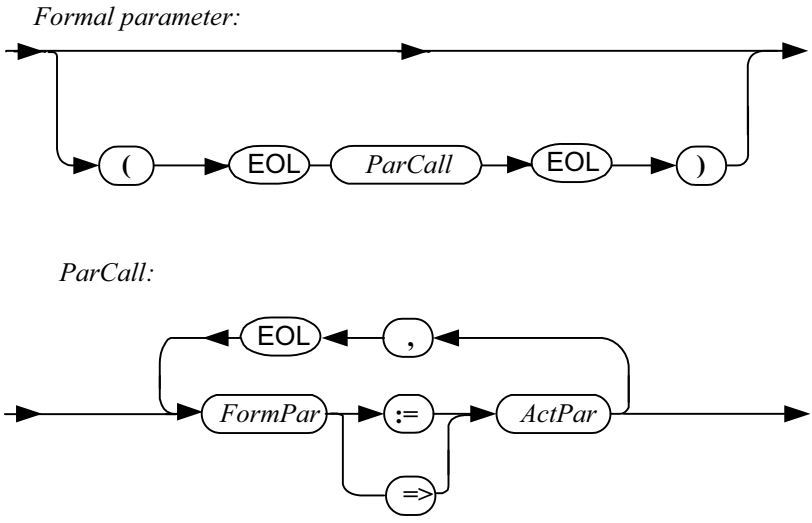



**Figure G.5.** Formal parameter assignment

As Figure G.4 shows, a function call consists of the function name together with a number of operands separated by commas as actual parameters of the function. Like a function block, a function can also be called with formal parameters.

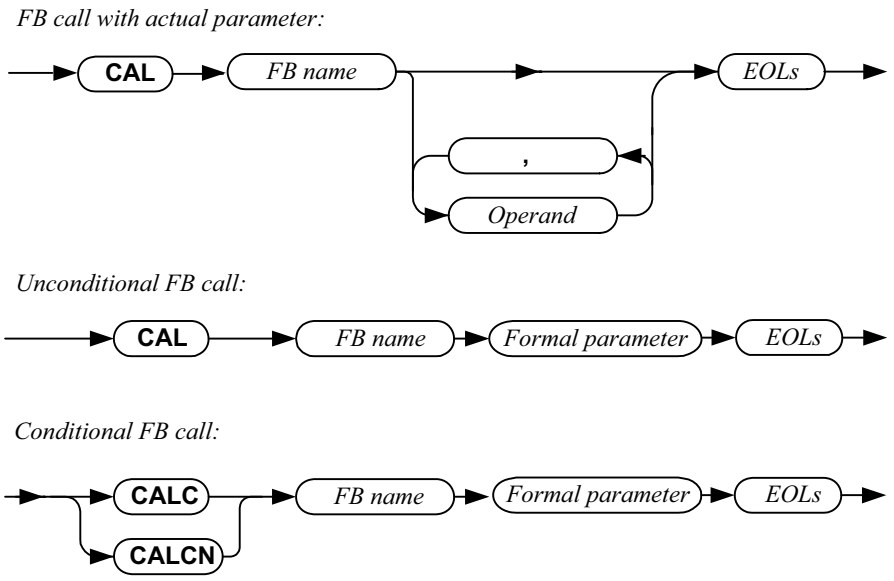



**Figure G.6.** Syntax diagrams for a function block call

Figure G.6 shows the syntax diagrams for conditional and unconditional calls of a function block. The unconditional call begins with CAL, the conditional call with CALC or CALCN. This is followed by the name of the FB instance, and the FB parameters in brackets.

The assignment of an actual parameter to a formal parameter is represented by the symbol ":=". Such assignments are required for every parameter and are separated by commas.

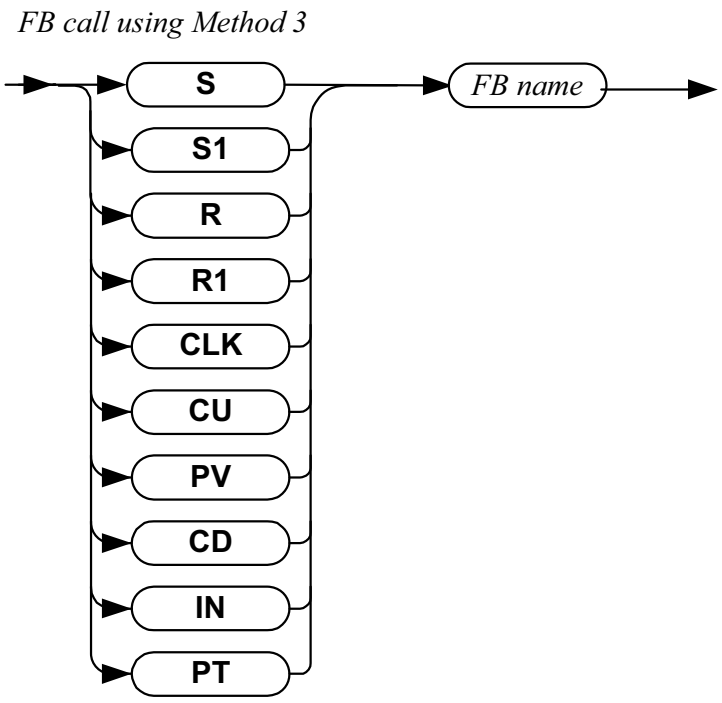



**Figure G.7.** FB call using Method 3

A call with the syntax shown in Figure G.7 results only in parameter initialisation or FB execution, depending on the parameters employed; see Section 4.12.

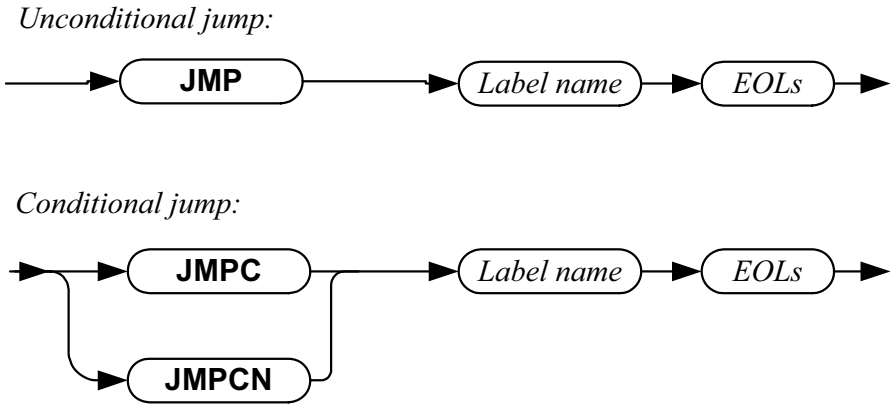



**Figure G.8.** Syntax diagrams for conditional and unconditional jumps

For jumps, the label name is specified after the jump operator JMP (unconditional) or JMPC/JMPCN (conditional) (Figure G.8).

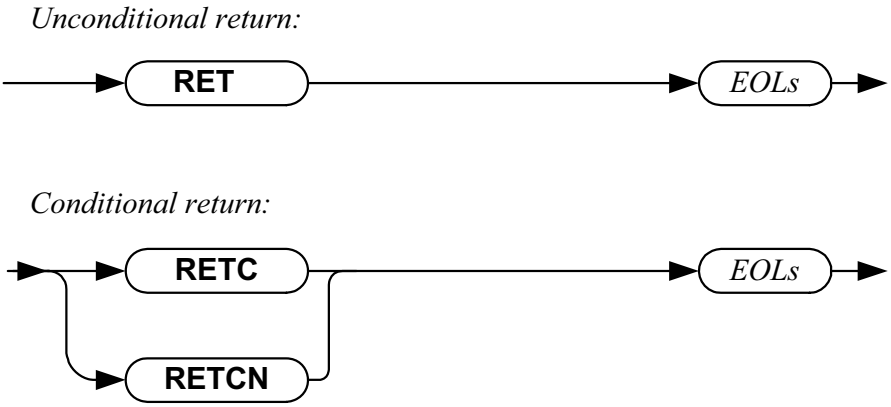



**Figure G.9.** Syntax diagrams for conditional and unconditional return

The returns shown in Figure G.9 have no operands or parameters.

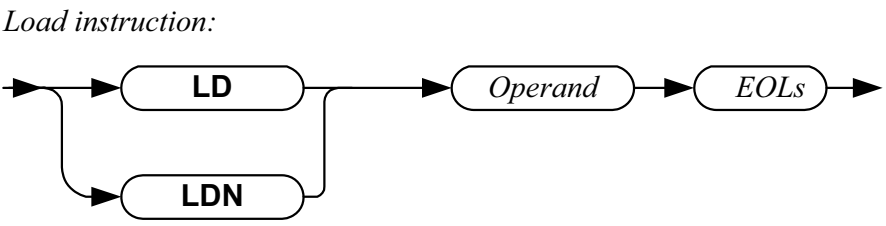



**Figure G.10.** Syntax diagram for the Load instruction

The Load instruction in Figure G.10 has a single (negatable) operand. It cannot be combined with a bracket or used inside a bracket.

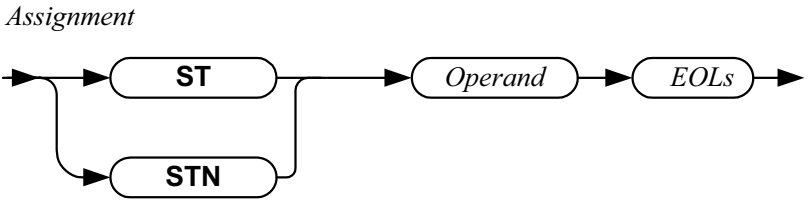



**Figure G.11.** Syntax diagram for assignment

Assignments (Figure G.11) consist of the operator ST or STN and the specification of the operand to be stored.

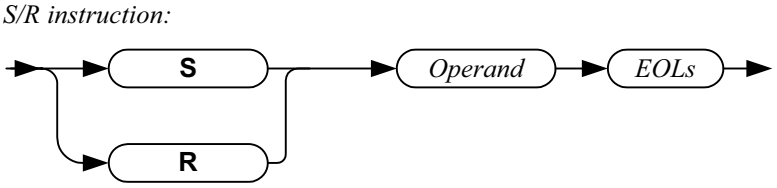



**Figure G.12.** Syntax diagram for the S/R instruction

An S/R instruction (Figure G.12) consists of the IL operators S or R and one operand.

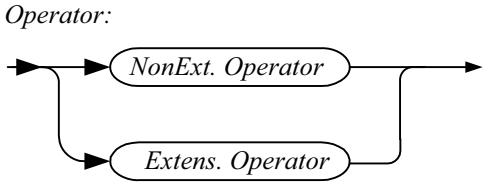



**Figure G.13.** Extensible and non-extensible operators

The operators represented in Figure G.13 perform logic operations, and are not used for loading or storage. A distinction is made between extensible and non-extensible operators, as shown below.

*Extens. Operator:*

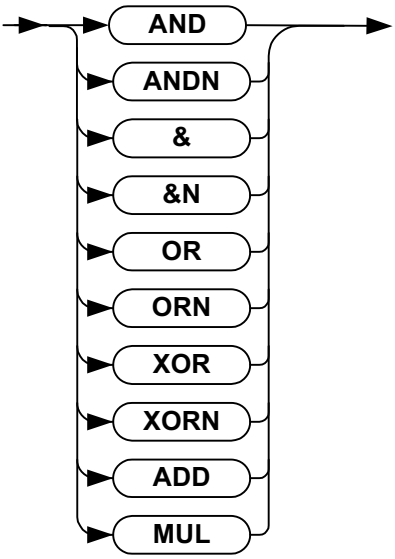



**Figure G.14.** Extensible operators: bitwise Boolean operations, addition and multiplication

Figure G.14 shows the extensible operators. They can have more than two input parameters. The bitwise Boolean operators (standard functions) can also be used with inversion.

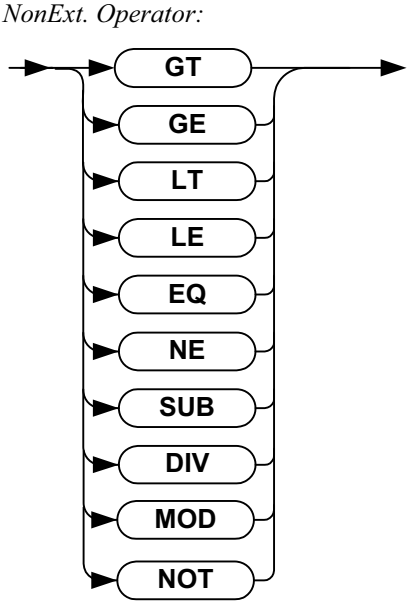



**Figure G.15.** Non-extensible operators: comparison, subtraction, division

The non-extensible operators in Figure G.15 have exactly two input parameters (including the current result CR).

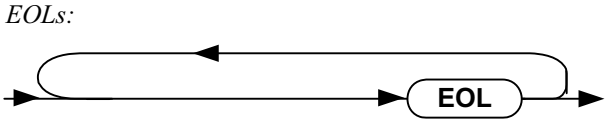



**Figure G.16.** Syntax diagram for the EOL (end of line) of an IL instruction

An IL line is concluded with a single EOL character (e.g. carriage return / line feed) or a comment followed by EOL (Figure G.16). These elements can occur once or any number of times in sequence.

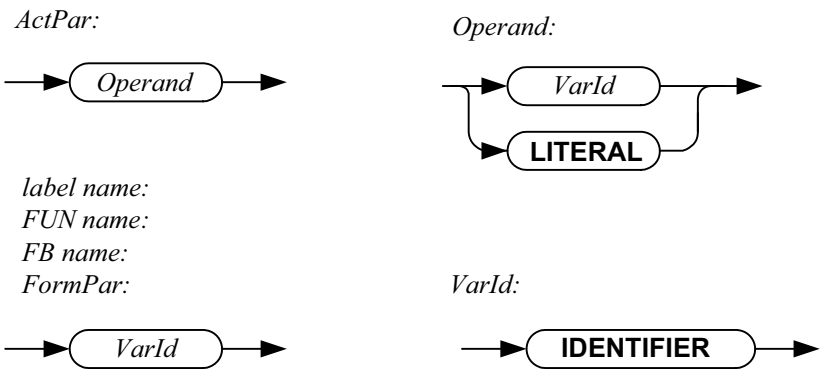



**Figure G.17.** Operands, parameters and other elements are represented by identifiers and literals.

Figure G.17 shows how parameters, operands and other elements are represented using IDENTIFIERs and LITERALs.

For simplicity the syntax diagrams of identifiers and literals are not shown here. The basic principles of their representation are explained in Section 3.2.

## G.2 IL Example from Syntax Diagrams

The IL syntax diagrams shown on the previous pages will now be used to produce an IL example. This shows how sample programs are constructed from syntax diagrams and vice versa, enabling IL examples to be checked for correctness.

```
                                    Beginning of IL sequence
0001    SequenceOne:                (* label *)
0002                                (* simple logic operation with jump *)
0003    LD      Var1                (* load instruction *)
0004                                (* beginning of conditional instructions *)
0005    ANDN    Var2                (* instruction without bracket *)
0006    ORN     ( Var3              (* instruction with bracket *)
0007    AND     Var4
0008    )                           (* end of bracketing *)
0009    AND     Var5
0010    ST      Var6                (* assignment *)
0011    S       Var7                (* S/R instruction *)
0012    RETC                        (* conditional return *)
0013                                (* end of conditional instructions *)
0014                                (* end of IL sequence *)
```

**Example G.1.** IL example. The comments refer to the corresponding syntax diagram. The line numbers on the left are used for reference in Table G.1.

To show how the IL example in Example G.1 can be built up from the syntax diagrams in Section G.1, Table G.1 shows the relevant syntax diagrams for each IL line.

| Line in Ex. G.1 | Syntax diagram | Figures |
|---|---|---|
| 0001-0014 | IL instruction sequence | Figure G.1 |
| 0001-0002 | (Jump) label | Figure G.1 |
| 0003-0004 | Load instruction | Figure G.10 |
| 0005 | Instruction without bracket | Figure G.2 |
| 0005 | Extensible operator ANDN | Figure G.14 |
| 0006-0008 | Instruction with bracket | Figure G.3 |
| 0006,0007 | Extensible operators ORN, AND | Figure G.14 |
| 0007,0009 | Instruction without bracket | Figure G.2 |
| 0007,0009 | Extensible operator AND | Figure G.14 |
| 0010 | Assignment | Figure G.11 |
| 0011 | S/R instruction | Figure G.12 |
| 0012-0014 | Conditional return | Figure G.9 |

**Table G.1.** Syntax diagrams to be used for each line of the IL sequence in Example G.1

This example shows how a concrete IL program is built up using syntax diagrams. In the syntax diagram for an IL instruction sequence (Figure G.1), first the label with name, colon and comments is inserted, followed by the first (Load) instruction.

The "conditional instructions" part is made up of two instructions, the first of which with a bracket containing further instructions. After the conditional instructions, the sequence is terminated with assignments and return.

In this way it is possible to create valid IL sequences from the individual syntax diagrams. Conversely, the relevant syntax diagrams for each IL line can be found in order to determine whether a program section is syntactically correct.

# H Reserved Keywords and Delimiters

IEC 61131-3 expressly permits the use of translation tables for adapting keywords and delimiters to national character sets.

## H.1 Reserved Keywords

Table H.1 lists all reserved keywords for programming languages of IEC 61131-3 in alphabetical order. They must **not** be employed as names for user-defined elements. In order to prevent confusion, the table also includes all formal parameters for functions and/or FB as well as data types beyond the scope of the standard.

| | | |
|---|---|---|
| **A** | ABS | ACOS |
| | ACTION | ADD |
| | AND | ANDN |
| | ANY | ANY_BIT |
| | ANY_DATE | ANY_DERIVED |
| | ANY_ELEMENTARY | ANY_INT |
| | ANY_MAGNITUDE | ANY_NUM |
| | ANY_REAL | ARRAY |
| | ASIN | AT |
| | ATAN | |
| **B** | BOOL | BY |
| | BYTE | |

**Table H.1.** Reserved keywords of IEC 61131-3 (continued on next page)

| | | |
|---|---|---|
| **C** | CAL | CALC |
| | CALCN | CASE |
| | CD | CDT |
| | CLK | CONCAT |
| | CONFIGURATION | CONSTANT |
| | COS | CTD |
| | CTU | CTUD |
| | CU | CV |
| **D** | D | DATE |
| | DATE_AND_TIME | DELETE |
| | DINT | DIV |
| | DO | DS |
| | DT | DWORD |
| **E** | ELSE | ELSIF |
| | END_ACTION | END_CASE |
| | END_CONFIGURATION | END_FOR |
| | END_FUNCTION | END_FUNCTION_BLOCK |
| | END_IF | END_PROGRAM |
| | END_REPEAT | END_RESOURCE |
| | END_STEP | END_STRUCT |
| | END_TRANSITION | END_TYPE |
| | END_VAR | END_WHILE |
| | EN | ENO |
| | EQ | ET |
| | EXIT | EXP |
| | EXPT | |
| **F** | FALSE | F_EDGE |
| | F_TRIG | FIND |
| | FOR | FROM |
| | FUNCTION | FUNCTION_BLOCK |
| **G** | GE | GT |
| **I** | IF | IN |
| | INITIAL_STEP | INSERT |
| | INT | INTERVAL |

**Table H.1.** (Continued on next page)

| | | |
|---|---|---|
| **J** | JMP | JMPC |
| | JMPCN | |
| **L** | L | LD |
| | LDN | LE |
| | LEFT | LEN |
| | LIMIT | LINT |
| | LN | LOG |
| | LREAL | LT |
| | LWORD | |
| **M** | MAX | MID |
| | MIN | MOD |
| | MOVE | MUL |
| | MUX | |
| **N** | N | NE |
| | NEG | NON_RETAIN |
| | NOT | |
| **O** | OF | ON |
| | OR | ORN |
| **P** | P | PRIORITY |
| | PROGRAM | PT |
| | PV | |
| **Q** | Q | Q1 |
| | QU | QD |
| **R** | R | R1 |
| | R_EDGE | R_TRIG |
| | READ_ONLY | READ_WRITE |
| | REAL | RELEASE |
| | REPEAT | REPLACE |
| | RESOURCE | RET |
| | RETAIN | RETC |
| | RETCN | RETURN |
| | RIGHT | ROL |
| | ROR | RS |

**Table H.1.** (Continued on next page)

| | | |
|---|---|---|
| **S** | S | S1 |
| | SD | SEL |
| | SEMA | SHL |
| | SHR | SIN |
| | SINGLE | SINT |
| | SL | SQRT |
| | SR | ST |
| | STEP | STN |
| | STRING | STRUCT |
| | SUB | |
| **T** | T | TAN |
| | TASK | THEN |
| | TIME | TIME_OF_DAY |
| | TO | TOD |
| | TOF | TON |
| | TP | TRANSITION |
| | TRUE | TYPE |
| **U** | UDINT | UINT |
| | ULINT | UNTIL |
| | USINT | VAR |
| **V** | VAR_ACCESS | VAR_CONFIG |
| | VAR_EXTERNAL | VAR_GLOBAL |
| | VAR_INPUT | VAR_IN_OUT |
| | VAR_OUTPUT | VAR_TEMP |
| **W** | WHILE | WITH |
| | WORD | WSTRING |
| **X** | XOR | XORN |

**Table H.1.** (Continued)

## H.2 Delimiters

Delimiters are "symbols" in the syntax of programming languages and have different meanings depending on where they are used. For example, round brackets can be used to indicate the beginning and end of a list of actual parameters in a function call, or they can be used together with the asterisk to frame comments.

All the delimiters and their combinations are listed in Table H.2 together with their possible meanings.

Characters for the graphical representation of lines are not included here.

| Delimiter | | Meaning, explanations |
|---|---|---|
| Space | | Can be inserted anywhere - except within keywords, literals, identifiers, directly represented variables or combinations of delimiters (such as "(*" or "*)").<br>IEC 61131-3 does not specify any rules about tabulators (TABs). They are usually treated as spaces. |
| End of line (EOL) | | Permissible at the end of a line in IL. In ST also permissible within statements. Not permitted in IL comments. EOL (end of line) is normally implemented by CR&LF (Carriage Return & Line Feed). |
| Begin comment | **(*** | Beginning of a comment (nesting not allowed) |
| End comment | ***)** | End of a comment |
| Plus | + | 1. Leading sign of a decimal literal, also in the exponent of a real (floating-point) literal<br>2. Addition operator in expressions |
| Minus | - | 1. Leading sign of a decimal literal, also in the exponent of a real (floating-point) literal<br>2. Subtraction operator in expressions<br>3. Negation operator in expressions<br>4. Year-month-day separator in time literals |
| Number sign ("hash") | # | 1. Based number separator in literals<br>2. Time literal separator |

**Table H.2.** Delimiters of IEC 61131-3 (continued on next page)

| Delimiter | | Meaning, explanations |
|---|---|---|
| Point | . | 1. Integer/fraction separator<br>2. Separator in the hierarchical addresses of directly represented and symbolic variables<br>3. Separator between components of a data structure (for access)<br>4. Separator for components of an FB instance (for access) |
| | e, E | Leading character for exponents of real (floating-point) literals |
| Quotation mark | ' | Beginning and end of character strings |
| Dollar sign | $ | Beginning of special characters within character strings |
| Prefix time literals | **t#, T#**<br>**d#, D#**<br>**d, D**<br>**h, H**<br>**m, M**<br>**s, S**<br>**ms, MS**<br>**date#, DATE#**<br>**time#, TIME#**<br>**time_of_day#**<br>**TIME_OF_DAY#**<br>**tod#, TOD#**<br>**date_and_time#**<br>**DATE_AND_TIME#**<br>**dt#, DT#** | Characters introducing time literals. Combinations of lower-case and upper-case letters are also permissible. |
| Colon | : | Separator for:<br>1. Time within time literals<br>2. Data type specification in variable declarations<br>3. Data type name specification<br>4. Step names<br>5. PROGRAM...WITH...<br>6. Function name/data type<br>7. Access path: Name/type<br>8. Jump label before next statement<br>9. Network label before next statement |
| Assignment (1) | := | 1. Operator for initial value assignment<br>2. Input connection operator (assignment of actual parameter to formal parameter in POU-call)<br>3. Assignment operator |
| Assignment (2) | => | Output connection operator (assignment of formal parameters to actual parameters in a PROGRAM call) |

**Table H.2.** (Continued on next page)

| Delimiter | Meaning, explanations |
|---|---|
| Round brackets **(...)** | Beginning and end of:<br>1. Enumeration list<br>2. Initial value list, also: multiple initial values (with repetition number)<br>3. Range specification<br>4. Array subscript<br>5. Character string length<br>6. Operator in IL (computation level)<br>7. Parameter list in POU call<br>8. Sub-expression hierarchy |
| Square brackets **[...]** | Beginning and end of:<br>1. Array subscript (access to an array element)<br>2. Character string length (in declaration) |
| Comma **,** | Separator for:<br>1. Enumeration list<br>2. Initial value list<br>3. Array subscripts (multidimensional)<br>4. Variable names (in the case of multiple declarations with the same data type)<br>5. Parameter list in POU call<br>6. Operand list in IL<br>7. CASE value list |
| Semicolon **;** | End of:<br>1. Definition of a (data) type<br>2. Declaration (e.g. variables)<br>3. ST statement |
| Two points **..** | Separator for:<br>1. Range specification<br>2. CASE range |
| Percent % | Leading character for hierarchical addresses of directly represented and symbolic variables |
| Comparison **>, <**<br>**>=, <=,**<br>**=, <>,** | Relational operators in expressions |
| Exponent ****** | Operator in expressions |
| Multiplication ***** | Multiplication operator in expressions |
| Division / | Division operator in expressions |
| Ampersand & | AND operator in expressions |

**Table H.2.** (Continued)

# I Glossary

In this chapter important terms and abbreviations employed in the book are listed in alphabetical order and explained in detail.

Terms, which are defined by IEC 61131-3, are marked "IEC".

| | | |
|---|---|---|
| **Action** | IEC | Boolean variable or a series of statements which can be accessed via an *action block* (in *SFC*). |
| **Action block** | IEC | Activation description of *actions* (in *SFC)* using an associated control structure. |
| **Action control** | IEC | Control unit for every *action* in SFC which is supplied with the input condition for activating the assigned action by means of one or more *action blocks*; also: action control block. |
| **Actual parameter** | | Actual value for an input variable (*formal parameter*) of a *POU.* |
| **Allocation list** | | List which contains the assignment of all symbols or *symbolic variables* to *PLC* addresses. |
| **Array** | | Sequence of elements of the same *data type.* |
| **Block** | | Programming unit, from which PLC-programs are built. Blocks can often be loaded independently from each other into the PLC, see also *POU*. Also referred to as sub-program in other programming languages. |
| **Cold restart** | IEC | Restart of the PLC-system and its application program, whereby all variables and memory areas (such as internal registers, timers, counters) are (newly) initialised with predefined values. This process can occur automatically after specific events (e.g. after a power failure) or also manually by the user (e.g. Reset button). |

| | | |
|---|---|---|
| **Comment** | IEC | Text written between parentheses and asterisks used to explain the program (cannot be nested!). This is not interpreted by the *programming system.* |
| **Configuration** | IEC | Language element CONFIGURATION which corresponds to a *PLC system* |
| **CPU** | | Abbreviation for Central Processing Unit (e.g. of a PLC) |
| **CR** | | Abbreviation for *Current Result* |
| **Cross-compilation** | | Conversion of the representation of a *POU* from one programming language to another, typically between ladder and function block diagram, but also between textual and graphical languages; also: cross-compiling |
| **Current result** | IEC | Interim result in *IL* of any *data type* |
| **Cycle** | | A single run of the (periodically called) application program. |
| **Cycle time** | | The time which an application program requires for one *cycle*; also: scan time. |
| **Data block** | | Shared data area, which is accessible throughout a program, see also *block*. In IEC 61131-3, there is no direct analogy. They are replaced here by global (non-local), structured data areas and *FB instance* data areas. |
| **Data type** | | Defines properties of a *variable* as bit length and value range. |
| **Declaration** | | Definition of *variables* and *FB instances* takes place in a *declaration block* with information about the data name, the *data type* or the *FB type* as well as appropriate *initial values*, *range specification* and *array attributes* (data template declaration).<br>The definition or programming of *POUs* is also designated as a *declaration* since their interfaces are made known to the *programming system* here. |
| **Declaration block** | IEC | Combination of *declarations* of one variable type at the beginning of the *POU.* |
| **Derived data type** | IEC | With the aid of a *type definition,* a user-specific *data type* is created. Its elements are *elementary data types* or *derived data types*. |
| **Directly represented variable** | IEC | *Variable* without further name which corresponds to a *hierarchical address.* |
| **Edge** | | The 0→1 transition of a Boolean variable is known as “rising edge”. Accordingly, the 1→0 transition is known as “falling” edge. |

| | | |
|---|---|---|
| **Elementary data type** | IEC | A standard *data type* predefined by IEC 61131-3. |
| **Enumeration** | | Special *data type* for the definition of a range of textual sequences. Mostly implemented internally as integer values. |
| **Extension of functions** | IEC | A *function* can have a variable number of inputs. |
| **FB** | IEC | Abbreviation for *function block* |
| **FB instance** | IEC | see *instance* |
| **FB type** | IEC | Name of a *function block* with call and return interface |
| **FBD** | IEC | Abbreviation for *Function Block Diagram* |
| **Formal parameter** | | Name or placeholder of an input variable (all *POUs*) or output variable (*function block* and *program*). |
| **Function** | IEC | A *POU* of type FUNCTION |
| **Function block** | IEC | A *POU* of type FUNCTION_BLOCK |
| **Function Block Diagram** | IEC | Function Block Diagram (FBD) is a programming language used to describe networks with Boolean, arithmetic and similar elements, working together concurrently. |
| **Generic data type** | IEC | Combination of *elementary data types* into groups using the prefix 'ANY', in order to describe *overloaded functions*. |
| **Hierarchical address** | IEC | Physical slot address of I/O modules of a *PLC system* (see also *I/O*). |
| **Hot restart** | IEC | Program restart at the place in the program where a power failure occurred. All battery-backed data areas as well as the application program context will be restored and the program can go on running, as if there had been no power failure. In contrast to *warm restart,* the interruption duration must be within a given value range depending on the process. For this purpose, the PLC system must have a separately secured real-time clock in order to be able to compute the interruption duration. |
| **I/O** | | The addresses of input and output modules belonging to a *PLC system* with *hierarchical addresses*. |
| **IL** | IEC | Abbreviation for *Instruction List* |
| **Indirect FB call** | | Call of an *FB instance* whose name is passed to the *POU* as a VAR_IN_OUT parameter. |
| **Initial value** | | Value of a *variable*, which will be assigned during initialisation, i.e. at system start-up time; also: starting count. |

| | | |
|---|---|---|
| **Instance** | IEC | Structured data set of an *FB* obtained by *declaration* of a *function block* indicating the *FB type*. |
| **Instruction List** | IEC | Instruction List (IL) is a much used Assembler-like programming language for PLC systems. |
| **Ladder Diagram** | IEC | Ladder Diagram (LD) is a programming language to describe networks with Boolean and electromechanical elements, such as contacts and coils, working together concurrently. |
| **LD** | IEC | Abbreviation for *Ladder Diagram* |
| **Multi-element variable** | IEC | *Variable* of type *array* or *structure*, which is put together from several different *data types.* |
| **New start** | | see *Cold restart* |
| **Overloading of functions** | IEC | The capability of an operation or *function* to operate with different input *data types* (but each of the same type). By this means several function classes are available under the same name. |
| **PC** | | Abbreviation for personal computer. |
| | IEC | Also Abbreviation for Programmable Controllers as employed in IEC 61131 |
| **PLC** | | Abbreviation for Programmable Logic Controller |
| **PLC programming computer** | | Unit consisting of computer, *programming system* and other peripherals for programming the *PLC*. |
| **PLC programming system** | | Set of programs which are necessary for programming a *PLC system:* program creation and compilation, transfer into the *PLC* as well as program test and commissioning functions. |
| **PLC system** | | All hardware, firmware and operating system parts required for executing a PLC program. |
| **POU** | IEC | Abbreviation for *program organisation unit* |
| **Pragma** | | Pragmas are offered by the *programming system* and are typically employed for program pre-processing and post-processing. |
| **Program** | IEC | A *POU* of type PROGRAM |
| **Program organisation unit** | IEC | A *block* of type *function*, *function block* or *program*, from which application programs are built. |
| **Programming computer** | | see *PLC programming computer* |
| **Programming system** | | see *PLC programming system* |
| **Range specification** | | Specification of a permissible range of values for a *data type* or a *variable*. |

| | | |
|---|---|---|
| **Recursion** | | Illegal in IEC 61131-3. It means:<br>a) the *declaration* of an *FB* using its own name or *FB type*,<br>b) mutual *FB* calls.<br>Recursion is considered an error and must be recognised while programming and/or at run time. |
| **Resource** | IEC | Language element RESOURCE which corresponds to a central processing unit of the *PLC system.* |
| **Retentive data** | IEC | Ability of a PLC to protect specific process data against loss during a power failure. The keyword RETAIN is used for this in IEC 61131-3. |
| **Reverse compiling** | | Recovery of the *POU* source back from the *PLC* memory. |
| **Run-time program** | | Program of *POU type* PROGRAM as an executable unit (by association with a *task*). |
| **Semantics** | | Meaning of language elements of a programming language as well as their description and their application. |
| **Sequential Function Chart** | IEC | Sequential Function Chart (SFC) is a programming language used to describe sequential and parallel control sequences with time and event control. |
| **SFC** | IEC | Abbreviation for *Sequential Function Chart* |
| **Single-element variable** | IEC | *Variable* which is based on a single *data type.* |
| **ST** | IEC | Abbreviation for *Structured Text* |
| **Standard function blocks** | IEC | Set of *function blocks* predefined by IEC 61131-3 for implementation of typical PLC requirements. |
| **Standard functions** | IEC | Set of *functions* predefined by IEC 61131-3 for the implementation of typical PLC requirements. |
| **Std. FB** | | Abbreviation for *standard function block* |
| **Std. FUN** | | Abbreviation for *standard function* |
| **Step** | IEC | State element of an SFC program in which statements of the *action* corresponding to this *step* can be started. |
| **Structured Text** | IEC | Structured Text is a programming language used to describe algorithms and control tasks by means of a modern high programming language similar to PASCAL. |
| **Symbolic variable** | IEC | *Variable* with name (identifier) to which a *hierarchical address* is assigned. |
| **Syntax** | | Structure and interaction of elements of a programming language. |

| | | |
|---|---|---|
| **Task** | IEC | Definition of run-time properties of programs. |
| **Transition** | IEC | Element of an SFC program for movement from one SFC *step* to the next by evaluating the transition condition. |
| **Type definition** | | Definition of a user-specific *data type* based on already available *data types*. |
| **Variable** | | Name of a data memory area which accepts values defined by the corresponding *data type* and by information in the variable *declaration*. |
| **Warm reboot** | | Term used for either *hot restart* or *warm restart* |
| **Warm restart (at the beginning of the program)** | IEC | Program restart similar to *hot restart* with the difference that the program starts again at the beginning, if the interruption duration exceeded the maximum time period allowed. The user program can recognise this situation by means of a corresponding status flag and can accordingly pre-set specific data. |

# J Bibliography

***Books concerning PLC programming according to IEC 61131-3:***


[JohnTiegel-09] K.-H. John and M. Tiegelkamp
„SPS-Programmierung mit IEC 61131-3“. Springer Berlin Heidelberg New York, 2009, 4th Ed. (German),
ISBN 3-540-67752-6

[Lewis-98] R. W. Lewis
„Programming industrial control systems using IEC 1131-3“, IEE Control Engineering, The Institution of Electrical Engineers, 1998,
ISBN 0-852-96950-3

[Lewis-01] R. W. Lewis
„Modelling Control Systems Using Iec 61499: Applying Function Blocks to Distributed Systems “, IEE Control Engineering, The Institution of Electrical Engineers, 2001,
ISBN 0-852-96796-9

[Bonfatti-03] Dr. Monari, Prof. Bonfatti and Dr. Sampieri
„IEC 61131-3 Programming Methodology; Software engineering methods for industrial automated systems“, 2003, ISBN 978-0973467000

***Standards concerning PLC programming:***

[IEC 61131-1] IEC 61131-1 Edition 2.0 (2003-05), TC/SC 65B. Programmable controllers - Part 1: General information www.iec.ch

[IEC 61131-2] IEC 61131-2 Edition 3.0 (2007-07), TC/SC 65B, Programmable controllers - Part 2: Equipment requirements and tests www.iec.ch

[IEC 61131-3] IEC 61131-3 Edition 2.0 (2003-01), TC/SC 65B, Programmable controllers - Part 3: Programming languages www.iec.ch

[IEC 61131-4] IEC 61131-4 Edition 2.0 (2004-07), TC/SC 65B, Programmable controllers - Part 4: User guidelines www.iec.ch

[IEC 61131-5] IEC 61131-5 Edition 1.0 (2000-11), TC/SC 65B, Programmable controllers - Part 5: Communications www.iec.ch

[IEC 61131-7] IEC 61131-7 Edition 1.0 (2000-08), TC/SC 65B, Programmable controllers - Part 7: Fuzzy control programming www.iec.ch

[IEC 61131-8] IEC 61131-8 Edition 2.0 (2003-09), TC/SC 65B, Programmable controllers - Part 8: Guidelines for the application and implementation of programming languages www.iec.ch

[IEC 61499-1] IEC 61499-1 Edition 1.0 (2005-01), TC/SC 65B, Function blocks - Part 1: Architecture www.iec.ch

[IEC 61499-2] IEC 61499-2 Edition 1.0 (2005-01), TC/SC 65B, Function blocks - Part 2: Software tool requirements www.iec.ch

[IEC 61499-4] IEC 61499-4 Edition 1.0 (2005-08), TC/SC 65B, Function blocks - Part 4: Rules for compliance profiles www.iec.ch

***Other important references***

[Holobloc] HOLOBLOC, Inc., FBDK (Function Block Development Kit) technology for 61499
http://www.holobloc.com/

[IEC 65B WG7] Working group for IEC 61131: „SC 65B-WG 7: Programmable control systems for discontinuous industrial-processes",
http://www.iec.ch/dyn/www/f?p=102:14:0::::FSP_ORG_ID:2598

[OPC Foundation] OPC Foundation, dedicated to ensuring interoperability in automation (formerly: OLE for Process Control)
http://www.opcfoundation.org

[PLCopen Europe] Eelco van der Wal
Molenstraat 34
4201 CX Gorinchem, The Netherlands

*Tel:* +31-183-660261
*Fax:* +31-183-664821
Email: evdwal@plcopen.org
http://ww.plcopen.org

[PLCopen North America]
Bill Lydon
10308 W. Cascade Dr.
Franklin, WI 53132
USA

*Tel:* +1 414-427-5853
Email: blydon@plcopen-na.org
http://www.plcopen-na.org/

[PLCopen Japan] Shigeo Kawashima
Mitsui Sumitomo Bank Ningyo-cho
5-7, Nihonbashi Ohdemma-cho, Chuo-lu
Tokyo, 103-0011
Japan

*Tel*: +81-5847-8058
*Fax*: +81-5847-8180
Email: kawashima-shigeo@fujielectric.co.jp
http://www.plcopen-japan.jp

[PLCopen China] No. 1, JiaoChangKuo
DeShengMenWai
Beijing, 100011
China

*Tel*: +86-(010) 6202-9216
*Fax*: +86-(010) 6202-7873
http://www.plcopen.org.cn

# K Index

## —S—

# Author Biographies

**Karl-Heinz John**

Born 09 July 1955 in Ulm, Germany. Married, 2 children.

Studied Computer Science from 1975 to 1981 at the Friedrich-Alexander-University Erlangen-Nürnberg, specialising in computer architecture. Degree in Computer Science, degree thesis on microprogramming. From 1981 to 1983 research assistant at the Faculty of Engineering Sciences, Chair of Computer Architecture and Performance Evaluation.

Since 1984 at infoteam Software GmbH (www.infoteam.de), a software company specialised in automation and medical device engineering, with headquarters in Bubenreuth / Nürnberg, Germany and subsidiaries in Stäfa, Switzerland and Beijing, China. As co-owner and chief executive officer (CEO), his areas of responsibility include the development of IEC 61131 programming systems, such as OpenPCS.
Founder member of PLCopen (www.plcopen.org), vice-president of ASQF (www.asqf.de), the largest association for software quality in German-speaking European countries; chairman of the ASQF Special Interest Group 'Automation' (ASQF Automation Days in Nürnberg, Germany and St. Gallen, Switzerland).

**Michael Tiegelkamp**

Born 23 December 1959 in Krefeld, Germany. Divorced, 2 children.

Studied Computer Science from 1982 to 1988 at the Friedrich-Alexander-University Erlangen-Nürnberg, specialising in computer architecture and communication systems. Degree in Computer Science, degree thesis on PLC architectures.

From 1988 to 1994 at infoteam Software GmbH as project manager and head of Marketing, responsible for PLC programming systems. Until October 1997 project manager at SIEMENS AG, responsible for development of programming device

software for SIMATIC S5/S7. Between 1998 and 2002 group manager for product definition of SIMATIC industrial software and from 2002 to 2004 project manager for system roadmap & innovation at Siemens Industry Automation and Drive Technologies. From 2004 to 2005 product management group manager for low-voltage energy distribution; from 2005 to 2007 organisation of the Power Management business segment. Since 2007 integration manager, changing to Siemens subsidiary OEZ s.r.o., Czech Republic, as process and project manager.

We have devoted a great deal of time and energy writing this book and have made every effort to ensure accuracy. If you discover any errors, please let us know!

We would also be grateful for your comments and suggestions.

You can contact the authors at:


| | *Karl-Heinz John* | *Michael Tiegelkamp* |
|---|---|---|
| Address | Irrlrinnig 13<br>D-91301 Forchheim<br>Germany | Kurpfalzstr. 34<br>D-90602 Pyrbaum<br>Germany |
| e-mail | karlheinz.john@gmx.de<br>*or (at infoteam):*<br>john@infoteam.de | michael.tiegelkamp@gmx.de<br>*or (at Siemens):*<br>michael.tiegelkamp@siemens.com |
| Homepage | www.fen-net.de/karlheinz.john | |
| Tel./Fax: | +49/9191/14672 (Tel.)<br>+49/9191/14672 (Fax) | +49/173/3112532 (Tel.) |