MOELLER

Language Elements for PS4-150/-200/-300 and PS416

Sucosoft S40 Programming Software

07/01 AWB2700-1306-GB

1st edition 12/97
2nd published 1998, edition 06/98
3rd published 1999, edition 03/99
4th published 2000, edition 02/00
5th published 2001, edition 07/01

See List of Revisions on Page 3

© Moeller GmbH, Bonn

Authors: Arno Dielmann, Eberhard Kastner
Editor: Thomas Kracht
Translator: Terence Osborn

Contents

About This Manual 3
List of revisions 3

1 Program Structure of SucosoftS40 5
Program POU 7
Function POU 8
Function block POU 9

2 Variables and Constants 11
Variables declaration 11
Variables 20
Constants 32

3 Textual Programming Languages 37
Introduction 37
Programming in instruction list (IL) 37
Programming in structured text (ST) 75

4 Graphical Programming Languages 91
Introduction 91
Programming in ladder diagram (LD) 95
Programming in function block diagram (FBD) 108

5 Functions 127
Overview 127

6 Function Blocks 221
Overview 221

7 Language Extension 587
Mixing IL and ST language elements 587
Single-bit access 587

07/01 AWB2700-1306-GB

8 Compliance with the IEC Standard 591
Common Elements 592
Language-specific elements 610
D.1: Implementation-dependent parameters 614
Reaction to errors 617

Appendix 619
Sucosoft conventions 619
Sucosoft limit values 621
Program examples LD/FBD 622

Index 641

About This Manual

The manual “Language Elements” describes programming languages compliant with the IEC/EN 61131-3 standard including functions and function blocks.

The language elements are identical for all controllers and CPUs in the PS4 and PS416 series. There may, however, be differences in the descriptions due to variations in the memory sizes of the controllers, or data types supported or a different I/O systems. In such cases your attention will be drawn to the differences by means of marginal comments or footnotes. “PS4-200” stands for the PS4-201-MM1, PS4-271-MM1, PS4-151-MM1 and PS4-141-MM1 controllers. “PS4-300” stands for the PS4-341-MM1 controller. References to “PS416” includes all CPU type, such as CPU-200, CPU-300 and CPU-400.

Unless explicitly indicated by comments, all descriptions apply to all controller and CPU types.

List of revisions

The following revisions have been made in comparison to the 08/00 edition of this manual:

Page	Description	new	Modification
75	Programming in Structured Text (ST)	×	
344 f.	Description of the IncEncoder function block		×
258	Description of the ClearDiagnosticBits function block	×	×
591	Chapter “Compliance with the Standard”	×	
437 ff.	SCO		×
533	Pulse length of Predivide input		×

07/01 AWB2700-1306-GB

07/01 AWB2700-1306-GB

1 Program Structure of Sucosoft S40

The Sucosoft programming languages comply with the IEC/EN 61131-3 standard. This document provides a description of the relevant language elements.

Three types of **P**rogram **O**rganisation **U**nits (POUs) are available for structuring a user application:

- Program
- Function (FU)
- Function block(FB).

Sucosoft provides a number of standard functions and standard function blocks for common tasks:

- Functions and function blocks which are defined in the IEC/EN 61131-3 standard are marked with an IEC symbol in the functions and function blocks.
- Manufacturer-specific functions and function blocks are marked with a Moeller symbol in the description of the functions and function blocks.

When a structured program is being processed, the invoking POU (program, function or function block) will be interrupted and the program process continued in the invoked POU (function or function block).

Every POU consists of a declaration section and an instruction section. All data elements which are used in an instruction section must be declared in the declaration section. The variable declaration of different data types is carried out in the individual declaration blocks which are entered in front of the instruction section.

A POU can also contain comments which are entered between the character combination (* and *).

Comments can be nested so that comments can include other comments. In this way, the commenting of program sections is simplified.

The content of a POU is enclosed in keywords which are automatically generated by the POU Editor:

PROGRAM...END_PROGRAM or
FUNCTION...END_FUNCTION or
FUNCTION_BLOCK...END_FUNCTION_BLOCK.

Example of an IL program “Typical elements of a POU”

```
PROGRAM exp_pou
              (* Declaration section*)
VAR
   Memory_1 : SR;                    (* Declaration of the FB "SR" as Memory_1 *)
   ON AT %I0.0.0.0.0 : BOOL;         (* Declaration of variables *)
   OFF AT %I0.0.0.0.1 : BOOL;
   Enable : BOOL;
   Start : BOOL;
   .                                 (* Declaration of further variables*)
   .
END_VAR
              (* Instruction section*)

   CAL Memory_1( Set1 := ON          (* FB invocation with parameter setting*)
                 ReSet := OFF)
   LD Memory_1.Q1
   ST Enable
   .                                 (*Program continued*)
   .

END_PROGRAM                          (*End of program*)
```


07/01 AWB2700-1306-GB

Program POU

A program POU is the highest structural level of Sucosoft and must be present at least once in each application. The name of this program POU is at the same time the name of the PLC application that you will later transfer to your PLC.

The program contains control instructions and invocations of other POUs (i.e. functions and function blocks). How a function or a function block is called depends on the programming language used.

IL example “program pou1”
Program pou1 with invocation of the SHL function (Shift left) and the CTU function block (up counter).

```
PROGRAM pou1
                                        (* Declaration section *)

VAR
   Counter1 : CTU;                      (* Declaration of the FB CTU as counter1 *)
   ShiftNumber AT %IW0.0.0.2: WORD;     (* Declaration of variables *)
   ShiftPlaces : INT := 2;              (* which are used by
                                        function SHL *)
   ShiftResult : WORD;
   Start AT %I0.0.0.0.0 : BOOL;         (* Declaration of further variables *)
   AT %I0.0.0.0.1 : BOOL;
   Pulse AT %I0.0.0.0.2 : BOOL;
   AT %I0.0.0.0.3 : BOOL;
   Belt AT %Q0.0.0.0.1 : BOOL;
   State : INT;
END_VAR
                                        (* Instruction section *)
   LD %I0.0.0.0.1
   AND Start
   ST Belt
   LD ShiftNumber                       (* FU invocation with parameter setting *)
   SHL ShiftPlaces
   ST ShiftResult
   CAL Counter1(CU := Pulse,            (* FB invocation with parameter setting*)
               ReSet := %I0.0.0.0.3)
   LD Counter1.CV
   ST State

END_PROGRAM                             (*End of program*)
```

Function POU

A function can be used where a program POU contains control tasks that are repeated several times. A function can contain several input parameters but only one output parameter, i.e. the process can only provide one data element as a result. This can be a single value or can involve several values, in the case of arrays.

The function has a global scope; it is thus available for all POUs and does not therefore need to be declared in the POU containing the invocation.

Functions cannot store information. A function with the same parameters thus always supplies the same return value when it is called up.

With every function call, local variables are initialised according to their data type (see Table 2-1 on Page 12). If a function call has been terminated without the function supplying a result, this automatically resets the initialisation value of the type concerned.

Further functions can be called up within a function. Function blocks may not be instantiated in a function and instance names of function blocks may not be transferred as input parameters in a function.

Sucosoft provides various manufacturer-defined functions for frequently required tasks.



Sucosoft allows online modifications and the use of the status display in the program POU and in function blocks but not in user-defined functions. In order to test the algorithm of a user-defined function, first accept it as function block, test it and then use it as a function.
Functions written by the user are currently not supported in PS4-200 controllers.

07/01 AWB2700-1306-GB

Function block POU

Frequently used tasks can also be implemented with function blocks as well as functions. Unlike functions, function blocks can supply several output operands as a result. A function block saves its variable values from one invocation to the next, and can use these values again the next time the function block is invoked, provided the variables have not been assigned new values.

A function block can call other function blocks. The following table shows the permissible uses of the inputs and outputs of function blocks:

Use	Inside the function block,	Outside the function block
Read input	LD IN1 AND IN2 ST Q1	LD FBx.IN1 ST %QW0.0.0.3
Write input	LD 1000 ST IN1	LD Start ST FBx.IN1 CAL FBx
Read output	LD OUT1 AND IN1 ST OUT1	CAL FBx(IN1 := On, ...) LD FBx.OUT1 ST %QW0.0.0.3
Write output	LD 1000 ST OUT1	Not permitted

If an input operand is used as an edge, it must be declared accordingly.

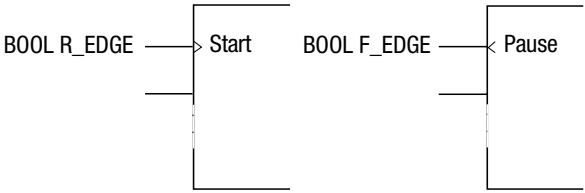
Example in IL/FBD: “Operand as edge”
Declaration of an input operand as edge shown in IL and FBD. In S40 operands can only be used as edges with the PS4-200 and PS4-300 series.

Rising edge

```
VAR_INPUT
Start : BOOL R_EDGE;
END_VAR
```

Falling edge

```
VAR_INPUT
Pause : BOOL F_EDGE;
END_VAR
```



Multiple instantiation of an FB
The single function block meanval in the program pou3 can be instantiated as often as required under different names. Define the instances in the declaration section of the invoking POU by assigning each application of the function block a different instance name (here: 'Instance1_Name' and 'Instance2_Name').

Sucosoft assigns each instance the required memory space so that the internal data of each instance of the function block can be stored. The function block is called with CAL and the appropriate instance name (CAL Instance1_Name).

07/01 AWB2700-1306-GB

2 Variables and Constants

Variables declaration

Data types

The IEC/EN 61131-3 Standard defines elementary and derived data types.

The keywords for designating data types can be written in upper or lower case letters.

Elementary data types

The elementary data types are predefined data types which are identified by means of keywords. The initialisation value of a data type is defined by the assignment operator ":=". If no initialisation value is assigned, the default value 0 will be assigned for numeric data types.

Table 2-1: Elementary data types

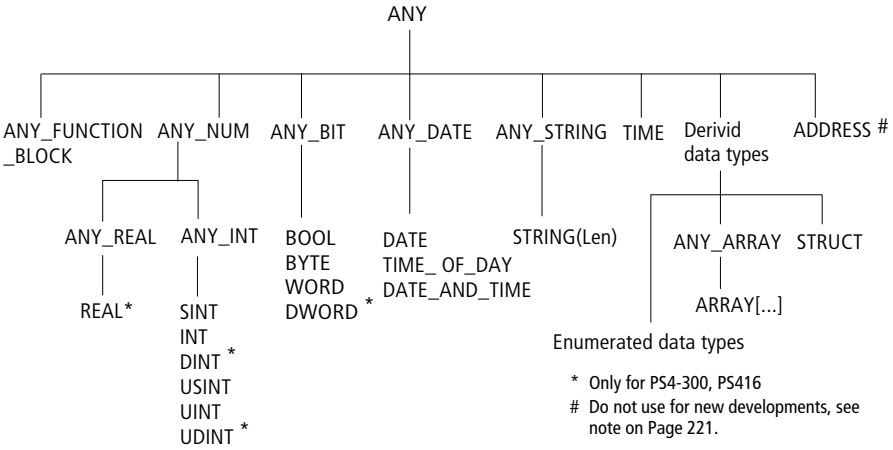
Keyword	PLC	Bits	Default value
BOOL	Boolean number; accepts value 1 or 0 corresponding to TRUE or FALSE	1	0
SINT	Short Integer; with the value range -128 to +127	8	0
INT	Integer; with the value range -32768 to +32767	16	0
DINT[1)]	Double Integer; with the value range -2147483648 to +2147483647	32	0
USINT	Unsigned Short Integer; with the value range 0 to 255	8	0
UINT	Unsigned Integer; with the value range 0 to 65535	16	0
UDINT[1)]	Unsigned Double Integer; with the value range 0 to 4294967295	32	0
REAL[1)]	Real Number (other designation "equal point number") with the following value range –3.402823E+38 to –1.75494E-38 +/–0 +1.75494E-38 to + 3.402823E+38	32	0.0
TIME	Duration Specified in days, hours, minutes or milliseconds Value range –129d +1 ms to +128 –1 ms; For PS4-200: –65d +1 ms to +64d –1 ms in conjunction with time arithmetic operations 0d to 64d –1 ms	–	T#0s
DATE	Calendar date in format YYYY-MM-DD	–	D#1900-01-01
TIME_OF_DAY	Time in format HH:MM:SS	–	TOD#00:00:00
DATE_AND_TIME	Date and time	–	DT#1900-01-01-00:00:00
STRING	Character string of variable length	[2)]	' ' (empty string)
BYTE	Bit sequence of 8 bits	8	0
WORD	Bit sequence of 16 bits	16	0
DWORD[1)]	Bit sequence of 32 bits	32	0

1) Only for PS4-300, PS416

2) The default length of a character string is 32 characters. If another length is required, this must be defined between parentheses in the declaration, e.g. text : STRING (12) := 'ABC';
Max. string length: 253 characters for PS4-200
1024 characters for PS4-300, PS416

07/01 AWB2700-1306-GB

If several data types are possible for an operand they are specified according to the following structure:



The data type ADDRESS as well as the category designation ANY in conjunction with data types cannot be used for the declaration of derived data types, variables, parameters, functions or function blocks. 'Generic' data types of this kind are only provided with input and output parameters of manufacturer functions and function blocks.

BOOL R_EDGE and BOOL F_EDGE (only PS4-200, PS4-300)

The data types BOOL R_EDGE (rising edge) and BOOL F_EDGE (falling edge) are provided in the Elementary Data Types list box or can be entered directly in the variables declaration. The two edge-triggered data types can only be used as input variables (VAR_INPUT) for user-defined function blocks. If the value is present at the input during a rising or falling edge, the input declared with R_EDGE and F_EDGE is high for the next call.

Derived data types

Derived data types are special manufacturer or user-defined data types derived from elementary data types and which have been assigned a new name. They are declared within the keywords **TYPE...END_TYPE** and can thus be used with the new names in variable declarations.

Example of derived data types

Declaring derived data types enables you to create your own terms for the designation of data types in declaration blocks VAR...END_VAR. The term analog_value is defined here for the data type INT:

```
TYPE
  analog_value : INT;
END_TYPE


VAR
  Value1: analog_value;
END_VAR
```

Derived data types are only known in the POU containing the type definitions concerned. They cannot therefore be used for declaring function and function block parameters or for global and external variables. This is only possible with global data types that are valid for the entire project.

07/01 AWB2700-1306-GB

To define derived data types for an entire project, declare these with the editor for global types (see “S40 User Interface” manual, AWB2700-1305-GB).

Declaration of a data type with restricted value range

The use of derived data types enables you to restrict the value range for the used data type.



If values outside of the value range are assigned, the nearest limit value is automatically used. Example: Value range –128 ... 128. If the value 256 is assigned, 128 will be used instead.

Example of restricted value ranges

Here the data type INT is not defined within the full range of values –32768 to +32767. Due to the assignment of the derived data type ‘Temperature’ the variable ‘BoilerTemperature’ can only take on values falling within the range –10 to +10. A similar restriction applies to the variables ’MeasValue1’ and ’MeasValue2’.

```
TYPE
  analog_VALUE_1 : INT (-128..128);
  Voltage : INT (0..150);
  Temperature : INT (-10..10);
END_TYPE

VAR
  MeasValue1 : analog_VALUE_1;
  MeasValue2 : Voltage;
  BoilerTemperature : Temperature;
END_VAR
```



The limit values are entered by using two dots without spaces to separate them. The positive numbers can be shown without the + sign.

Declaration of a data type as enumeration
Enumerated data types are declared to define a list of identifiers whose values may contain one data element. The enumerated data type is declared by entering the variable list in parentheses after the name; the individual elements must be separated with a comma. Enter the initialisation value after the ")" parenthesis. If an initialisation is not made within the declaration, the data element will be assigned the value of the first element in the enumeration list when processing the program for the first time.

Example of type initialization
The type 'Traffic_lights' can take one of the three states 'red', 'amber' or 'green'. A variable is initialized with the state 'Amber'. If there is no initialisation the variable would automatically take on the value of the first element - i.e. the state 'Red'.

07/01 AWB2700-1306-GB

```
 TYPE traffic_lights:
  (red,
  amber,
  green):= amber;
END_TYPE

VAR
  Main_street1 : Traffic_lights;
  Side_street1 : Traffic_lights;
  Side_street2 : Traffic_lights;
  Car_stop : BOOL;
END_VAR
```

The defined, permitted elements can be used directly in the instruction section of the program.

Example IL

```
LD     Main_street1
EQ     red
ST     Car_stop
```

Declaration of a data type as array

Several data elements of the same type can be combined into an array which can consist of elementary or derived data types. The data type is declared with the keyword ARRAY and the definition of the number of array elements and the data type involved.

Array variables may be assigned with a direct address. For this assign an array with an initial address, e.g. with %IB0.0.0.0 via the AT keyword or in the address field of the Syntax-Controlled Variable Editor. See also Page 22.

Example of declaring a data array
This declaration makes the Array1 variable into a data array with five data elements of type INT. The Array2 variable is a three-dimensional data array with 24 data elements.

The data element of Array2 which is initialized with the value 17 is overwritten with the value 5.

```
PROGRAM field
TYPE
 Field_5_INT: ARRAY [1..5] OF INT:=[1,13,5,34,2];
 Field_24_INT:ARRAY [1..2,1..3,1..4] OF INT
              :=[[[3,7,2,9],[2,8,6,1],[5,7,2,17]],
               [[2,3,0,5],[6,1,2,4],[3,0,1,4]]];
END_TYPE

VAR
 Array1 : Array_5_INT;
 Array2 : Array_24_INT;
END_VAR
 .
 .
 LD 5
 ST Array2 [1,3,4]
 .
 END_PROGRAM
```

Declaration of a data type as a structure
Several data elements of different types can be grouped into a single structure. A structure may consist of elementary or derived data types. A structure is declared with the keyword **STRUCT** and a list of the structure elements specifying their data types.

To access a structure element, enter the variable name declared as a structure and – separated by a dot – the name of the structure element.

07/01 AWB2700-1306-GB

Example of initialization in the declaration

```
PROGRAM structure

TYPE
  Plant_data :
      STRUCT
           MeasValue1 : BOOL := 1;
           MeasValue2 : INT := -15;
           MeasValue3 : UINT := 100;
           MeasValue4 : REAL := 10.2;
      END_STRUCT;
END_TYPE

VAR
  Station1 : Plant_data;
  Station2 : Plant_data;
END_VAR

  LD Station1.MeasValue3
  .
  .
END_PROGRAM
```

Example of initialization after the declaration

```
TYPE
  System data
       STRUCT
           MeasValue1 : BOOL;
           MeasValue2 : INT;
           MeasValue3 : UINT;
           MeasValue4 : REAL;
       END_STRUCT := (MeasValue1 := 1,
                      MeasValue2 := -15,
                      MeasValue3 := 100,
                      MeasValue4 := 10.2);
END_TYPE

VAR
  Station1 : Plant_data;
  Station2 : Plant_data;
END_VAR
```

Assignment of the type ‘Plant_data’ causes the variables ‘Station1’ and ‘Station2’ to take on the properties of the structure ‘Plant_data’. In other words, they are given in each case a data element of the type BOOL, INT, UINT or REAL.



If no initialization parameters are transferred in the type declaration of the derived data types, the individual data elements will be assigned the type-dependent default values.

Variables

Variables are data elements whose content can change. The variables can be divided into two groups:

Variables used for temporary storage of internal data and

directly represented variables which are connected with the inputs and outputs or specified markers or communication memories of the PLC.

A variable is identified by a name – the identifier. The name starts with a letter (a–z) or an underline character and can consist of a string of upper and lower case letters, numbers and underline characters.



Spaces and umlauts are not allowed in the variable names. Keywords must not be used as variable names.
Variable names have a maximum length of 64 characters. No distinction is made between upper and lower case.

07/01 AWB2700-1306-GB

Directly represented variables

The physical PLC addresses (PLC inputs and outputs) can also be addressed directly. Directly represented variables must also be declared. Symbolic names do not have to be assigned. In this case the operands are addressed with their physical addresses – starting with the % character. In the declaration block the keyword AT is entered in front of the physical address separated by a space.

Example of declaring physical operands

Declaration of physical operands with and without assignment of a symbolic name

```
PROGRAM pou4
VAR
  AT %I0.0.0.0.0 : BOOL;
  Input_1 AT %I0.0.0.0.1 : BOOL;
  Result : BOOL;
END_VAR

  LD Input_1
  AND %I0.0.0.0.0
  ST Result
END_PROGRAM
```

The use of symbolic names also with the directly represented variables is recommended. If a modification is made to input I0.0.0.0.0 that has no symbolic name, an appropriate correction would have to be made in both the declaration block and in the entire program at all points where the input I0.0.0.0.0 is used. On the other hand, to change the address I0.0.0.0.1, you only need to make a correction in the declaration block since in the instruction section the symbolic name Input_1 and not the address I0.0.0.0.1 is used.



Directly represented variables must only be used in program POUs.

You can also assign an array to a direct address. In this case, observe the following requirements:

All array elements must have the same input/ output type, e.g. %I, %Q or %M.

With bit addresses (%IX) the element data type must be of type BOOL or be derived from this type.

With byte addresses (%IB) the element data type must be of type BYTE, SINT or USINT or be derived from these types.

With word addresses (%IW) the element data type must be of type WORD, INT or UINT or be derived from these types.

With double word addresses (%ID), the element data type must be of type DWORD, DINT, UDINT or REAL or be derived from these types.

With arrays of BOOL elements for PS4-200 and PS4-300 the number of array elements must be a multiple of 8 and the bit number must always be set to 0.

Example of array declaration on direct variables

```
VAR
   Marker           AT%MW0.0.0.4:    ARRAY[1..10] of int;
   CounterFlags     AT%MB0.0.0.50:   ARRAY[1..5, 1..3] of usint;
   Flags            AT%MX0.0.0.7.0:  ARRAY[1..16] of bool;
   Status           AT%ISX1.2.0.0.0: ARRAY[0..7] of bool;
   Array            AT%MD0.0.0.400:  ARRAY[1..2] of REAL;
END_VAR
```


07/01 AWB2700-1306-GB

If directly represented variables are declared as global variables in the program POU, they can be used as external variables in the function block. The second option is to transfer the variable parameters to the function block via VAR_IN_OUT. This is only possible for PLC outputs, markers and RD, SD communication data.

Sucosoft supports the following physical addresses:

I: Digital input
Q: Digital output
M: Marker
IA: Analog input
QA: Analog output
IC: Counter input
IS: Status information (diagnostics)
Assignment of IS depends on the device; refer to the respective manual for details.
SD: Send data for intelligent Suconet K stations
RD: Receive data from intelligent Suconet K stations

The following physical addresses are used for direct access of inputs/outputs (→ Chapter 7):

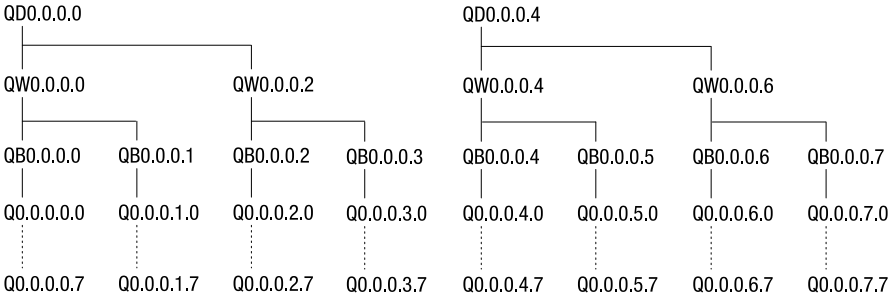
IP: Digital input (PS4-200/-300)
QP: Digital output (with PS4-200/-300)
IPA: Analog input (with PS4-300)
QPA: Analog output (with PS4-300)
IPC: Counter input (with PS4-300)

The available addresses depend on the hardware configured.

The following symbols can be used for Length:

Symbol	Meaning	Length	Example
X or none:	Bit	1 bit	%IX0.0.0.0.0 or %I0.0.0.0.0
B:	Byte	8 bits	%IB0.0.0.0
W:	Word	16 bits	%QW0.0.0.0
D:	Double word	32 bits	%ID0.0.0.4

The organisation of the directly represented variables supported by Sucosoft is shown in the example of the output operands (Q):



Multiple declaration on directly represented variables (PS4-200 and PS4-300 only)
The directly represented variables can be declared in such a way that a variable with a smaller data type is located within this variable with a larger data type.

07/01 AWB2700-1306-GB

Example of multiple declaration on variables
The variable 'Bit_Var' in the first declaration forms the fifth bit in the variable 'Word_Var'. In the second declaration the variable 'Byte_Var' is the high byte in the variable 'Word_Var'.

```
VAR
  Bit_Var AT %IO.0.0.0.4:BOOL;
  Word_Var AT %IWO.0.0.0:WORD;
END_VAR

VAR
  Byte_Var AT %IO.0.0.1:BYTE;
  Word_Var AT %IWO.0.0.0:WORD;
END_VAR
```

Multiple declaration is not supported with the PS416.

Variable types

The types of all variables are defined in the declaration section of a program organisation unit. All variables that you use in the instruction section of a POU must be declared in the declaration section. The individual variable types are identified with keywords. Variables of the same type are stored in the declaration block. A declaration block starts with a keyword, e.g. VAR or VAR_GLOBAL – depending on the variable type – and ends with the keyword END_VAR. The following table shows the different variable types and their use:

Keyword	Use
VAR	'Local variable'; valid only within the POU where it was declared.
VAR_GLOBAL	A variable is declared as a 'global variable' if it is to be used in a program and in all FBs which can be called by this program. A global variable called up in a program is known within this program and within the FBs that are called up by this program. The variable must be declared with the same identifier (name) as VAR_EXTERNAL in all invoked FBs in which this global value is used.
VAR_EXTERNAL	If a global variable is used within a FB, it must be declared as 'VAR_EXTERNAL' with the same identifier (name).
VAR_INPUT	Declare an 'input variable' if the variable is to be read only within a POU (FB, FU) or if it is to be used for transferring parameters in a function or a function block. The variable cannot then be changed in this POU.
VAR_OUTPUT	'Output variable' of a function block
VAR_IN_OUT	An IN_OUT variable is read, processed and output under the same name by the FB; it is known as a passing parameter. Since an operation on an IN_OUT variable has a direct effect on the contents of the transferred variable, this transferred variable must not be of a type that does not permit write operations - in other words, e.g. no INPUT variable or no variable with the attribute CONSTANT.
TYPE	Use the keyword 'TYPE' to declare the derived data types. Derived data types are special data types which you can derive from the elementary data types. You can find detailed explanations in the section 'Derived data types' on Page 14.

Each declaration block must end with the keyword 'END_VAR' – the declaration of derived data types with the keyword 'END_TYPE'.

07/01 AWB2700-1306-GB

VAR_IN_OUT declarations can only be used in function blocks.

VAR_IN_OUT variables must be set again ("connected") every time a function block is called, even if this is not required for a particular call. The connection must be made within the parenthesized area of the call instruction.

Example of pointer to an array with 201 words
In the following example, the VAR_IN_OUT variable 'w_array' in the function block 'demo' is not an array of 201 words but a pointer to an array of 201 words. The 'demo' function block therefore accesses the global array 'word_array'.

This solution has the following advantages:

- When 'demo' is called it is not necessary to copy an array (runtime saving) and the VAR_IN_OUT variable can be accessed for both read and write operations.
- The function block is neutral since EXTERNAL declarations that are attached to variable names are not necessary.

```
PROGRAM user

VAR_GLOBAL
  word_array : array[0..200] of word;
END_VAR

VAR
  my_function_block : demo;
END_VAR

  cal my_function_block(w_array := word_array)
END_PROGRAM

FUNCTION_BLOCK demo

VAR_IN_OUT
  w_array : array[0..200] of word;
END_VAR

VAR
  Help1 : word ;
  Help2 : word ;
END_VAR

  ld w_array[100]
  st Help1
  ld Help2
  st w_array[200]
END_FUNCTION_BLOCK
```

Attributes and address allocations

The declaration of local and global variables can be supplemented as follows:

Keyword	Use
RETAIN	**Use RETAIN to declare local and global variables as retentive. Retentive means that with a warm start the retentively declared variable keeps the last valid value it had before the stop. RETAIN is written behind the keywords VAR, VAR_GLOBAL or VAR_OUTPUT after a space (see also the following section).**
CONSTANT	**If a local or global data element is to be constant, i.e. the contents cannot be changed, the attribute 'CONSTANT' is used. Place the attribute CONSTANT behind the keywords VAR, VAR_GLOBAL or VAR_OUTPUT and after a space.**
AT	**If you require a variable assigned to a physical address (a directly represented variable), mark it in the declaration block of a Program POU with AT (see also IL example on Page 30)**

07/01 AWB2700-1306-GB

Declaration of warm start retentive markers for different PLC types

PS4-200:
Marker variables are not made retentive by the attribute RETAIN. They are directly assigned to a marker from the range that is defined as retentive (AT %M....). The retentive marker range is defined in the Sucosoft Navigator via Generate → Program Parameters... → Markers.

If variables of a function block are to be used as retentive variables, this will not be possible in relation to individual variables. In this case the instance of the function block must be declared as retentive.

PS-300:
Retentive markers can be made retentive by the attribute RETAIN as well as by directly assigning them to a marker range that is defined as retentive.

The retentive marker range is defined in the Sucosoft Navigator via Generate → Program Parameters... → Markers.

PS416:
No definitions are required for the PS416 since all markers are retentive by default. This setting cannot be changed.

Example of retentive declaration with RETAIN
Declaration of a retentive data element, of a retentive function block and of a constant data element:

```
VAR RETAIN
  CounterValue : UINT;
END_VAR

VAR RETAIN
  Counter : CTU;
END_VAR

VAR_GLOBAL CONSTANT
  ZeroKelvin : INT := -273;
END_VAR
```

The keyword 'AT' identifies variables that are assigned to physical addresses. This keyword can only be used when declaring local and global variables within a program POU, e.g.:

```
VAR
  motor1 AT %Q0.0.0.0.3 : BOOL;
END_VAR
```

Example of assigning physical addresses
Two local variables with the symbolic names 'pump' and 'valve' and a global variable with the symbolic name 'setpoint' are declared. Two declaration blocks are necessary.

```
VAR_GLOBAL                (*Declaration block*)
  setpoint: INT:=25;      (* global variable*)
END_VAR

VAR                       (*Declaration block*)
  pump AT %Q0.0.0.0.3: BOOL;(* local variable*)
  valve AT %Q0.0.0.3.1: BOOL;
END_VAR
```


07/01 AWB2700-1306-GB

If a directly represented variable without a symbolic name is used in the declaration, e.g. %Q0.0.0.0.3, the output in the instruction section will be directly addressed under %Q0.0.0.0.3.

The following table shows the use of the individual types of variables in the three POU types:

Use of variable types	FU	FB	PRG	Global types
Var_Input	✓	✓		
Var_Output		✓		
Var_In_Out		✓		
Var_Global			✓	
Var_External		✓		
Var	✓	✓	✓	
Type	✓	✓	✓	✓



Observe the order:
VAR must not be placed in front of VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT.

Instantiating function blocks

If a function block is to be called in another function block or a program, it is necessary to create an instance of it, i.e. the function block must be assigned a user-defined name. Assign the instance name in the declaration section of the invoking POU, e.g.:

```
Counter1 : CTU;
```

The Counter1 variable is thus declared as a CTU function block. Set the parameters for the input/ output variables when the function block is called.

Unlike a function, you only need to set the parameters for the input variables which are relevant for the application. An exception to this is input variables of the type VAR_IN_OUT, for which the parameters have to be set with every call. If this does not happen, a syntax error will be reported.

Constants

If fixed data is used in Sucosoft, the data elements which are declared with the keyword CONSTANT can be initialised with the desired contents. It is also possible to use constants which are predefined values for direct processing within the program without using a data element with an identifier.

Example
The variable Value1 which has been increased by 115 is to be assigned to the variable Value2.

```
LD      Value1
ADD     115
ST      Value2
```

The constants (initial values) of the individual data types require a special notation. Underline characters can be used within a constant; they do not affect the value of the constant. Spaces may only be used in string constants.

07/01 AWB2700-1306-GB

Table 2-2: Constants overview

Constants	Example	Meaning
INT	–13 45165 or 45_165 +125	integer -13 integer 45165 integer 125
REAL[1)]	–13.12 123.45 0.123 –1.23E-3	real number -13.12 real number 123.45 real number 0.123 exponent notation of real number -0.00123
Dual number	2#0111_1110	126 decimal
Octal number	8#123	83 decimal
Hexadecimal number	16#123	291 decimal
BOOL	0 and 1 or FALSE and TRUE	Boolean zero and one
STRING	'ABC'	Character string ABC
TIME	T#12.3ms or TIME#12.3ms	Duration of 12.3 milliseconds
	T#12h34m or T#12h_34m	Duration of 12 hours and 34 minutes
DATE	DATE#1995-12-24 or D#1995-12-24	Date 24.12.1995
TIME_OF_DAY	TOD#12:05:14.56 or TIME_OF_DAY#12:05:14.56	Time 12 o'clock, 5 minutes and 14.56 seconds
DATE_AND_TIME	DT#1995-12-24-12:05:14.56 or DATE_AND_TIME#1995-12-24-12:05:14.56	Date and time 24.12.1995, 12 o'clock, 5 minutes and 14.56 seconds

1) Only for PS4-300 and PS416

Duration, time and date are represented by the keywords which can be written in full, e.g. TIME_OF_DAY#, or in short, e.g. TOD#.

A constant – string of characters – is a sequence of individual characters which are enclosed by apostrophes.

This notation allows all characters apart from the $ character to be represented. The $ character has a special meaning when representing constants. It is used for the representation of control characters for the formatted text output, e.g. $L for the line feed. The $ character also enables you to represent the individual characters in their two-digit hexadecimal form, e.g. $0A for the line feed.

Predefined Constants	**Meaning**
'$'	'Apostrophe '
'$$'	The $ character itself
'$L' or '$l'	Line feed, hexadecimal $0A
'$N' or '$n'	New line
'$P' or '$p'	Form feed
'$R' or '$r'	Carriage return, hexadecimal $0D
'$T' or '$t'	Tabulator, hexadecimal "$09"

07/01 AWB2700-1306-GB

Examples

Constants	Meaning, Length of string
'A'	The character A, length 1
' '	Space, length 1
''	No character, length 0
'RL'	Carriage return and line feed, length 2
'$0D$0A'	Carriage return and line feed, length 2

07/01 AWB2700-1306-GB

3 Textual Programming Languages

Introduction

Sucosoft S40 provides the two textual programming languages Instruction List (IL) and Structured Text .

ST is a higher-level language comparable with Pascal, and is used for more complex programming tasks.

Programming in instruction list (IL)

The IEC/EN 61131-3 standard made the instruction list programming language suitable for all processors. It also still retained its Assembler-like structure.

Structure of an IL instruction

An IL command line (instruction) is specifically defined according to the following structure. Each IL command line must be completed with a line break.

Label	Operator	Operand	Comment
L1:	LD	%I0.0.0.1	(*Load an input into the working register*)

Label Identifies an instruction as jump target (optional).

Operator Depending on the function, one of the commands shown in Table 3-1 on Page 41 is used.

Operand Variable name; depending on the declared variable type, the variable name is the assigned symbolic name; with directly represented variables the symbolic name or the physical

address, e.g. %I0.0.0.1.

Permissible operands:

Constants:	
Numeric	e.g. 101
Strings	'Moeller'
Time	T#60s
Variable:	
Single variable	e.g. var_1
Single-bit access	var_2.15, see Page 587,
Array elements	var_array[7,8], see Page 17,
Structure elements	var_3.struct_element1, see Page 18
Directly represented	e.g. %I0.0.0.0.0, see Page 21
Address operator	&var_4, see Page 159
FB instance name (poss. with FB parameters)	e.g. CAL FBx(IN1 := On1, ... , see Page 9

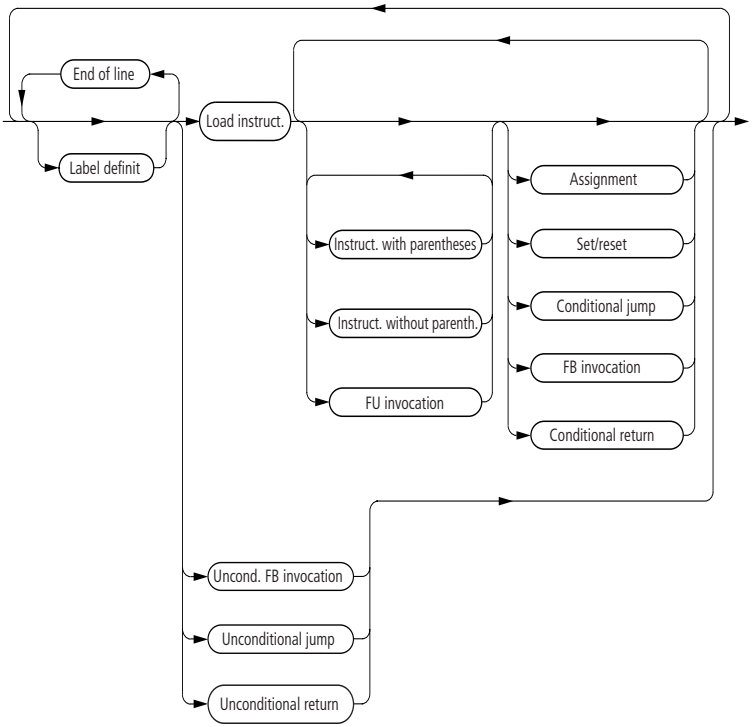
Comment The comment provides more detailed information on the executed operation (optional).

07/01 AWB2700-1306-GB

Programming rules

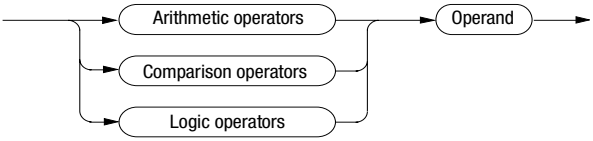
An IL sequence starts with a load instruction (LD or LDN) and is valid until a new load instruction is used. This does not apply to unconditional function block calls and jump/POU return operations.

IL sequences (overview)
The following syntax chart shows an overview of all IL sequences for all instruction groups. The use of the individual instruction groups is explained with detailed diagrams for each respective group.

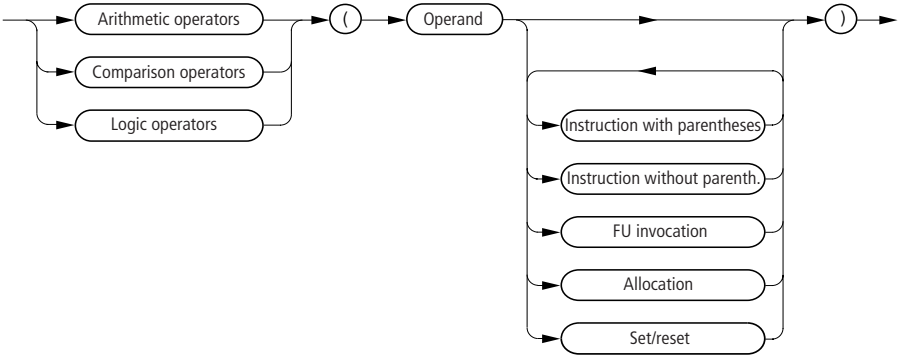


Use of logical, arithmetic and comparison operations

Instructions without parentheses:

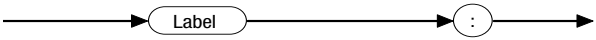


Instructions with parentheses:



Label definition

Labels are used as jump targets. They are entered at the beginning of an instruction and may only be positioned at the beginning of a sequence.



07/01 AWB2700-1306-GB

Description of the instructions

Table 3-1: IL instruction set

Operator	Operand type	Meaning
LD	ANY	Copies the operand value into the working register
LDN	ANY_BIT	Copies the negated operand value into the working register
ST	ANY	Sets the operand value to the current result (CR)
STN	ANY_BIT	Sets the operand value to the negated value of the current result
S	BOOL	Sets Boolean operand value to 1 if the current result is 1
R	BOOL	Resets Boolean operand value to 0 if the current result is 1
AND[1)]	ANY_BIT	Boolean AND
&[1)]	ANY_BIT	Boolean AND
ANDN[1)]	ANY_BIT	Boolean AND, negated
&N[1)]	ANY_BIT	Boolean AND, negated
OR[1)]	ANY_BIT	Boolean OR
ORN[1)]	ANY_BIT	Boolean OR, negated
XOR[1)]	ANY_BIT	Boolean Exclusive OR
XORN[1)]	ANY_BIT	Boolean Exclusive OR, negated
ADD[1)]	ANY_NUM	Addition
SUB[1)]	ANY_NUM	Subtraction
MUL[1)]	ANY_NUM	Multiplication
DIV[1)]	ANY_NUM	Division
GT[1)]	ANY	Comparison greater than
GE[1)]	ANY	Comparison greater than or equal to

Operator	Operand type	Meaning
EQ[1)]	ANY	Comparison equal
NE[1)]	ANY	Comparison not equal
LE[1)]	ANY	Comparison an or equal to
LT[1)]	ANY	Comparison less than
JMP	Label	Unconditional jump to label
JMPC	Label	Jump to label with current result 1
JMPCN	Label	Jump to label with current result 0
CAL	Instance name	Unconditional invocation of a function block
CALC	Instance name	Invocation of a function block with current result 1
CALCN	Instance name	Invocation of a function block with current result 0
RET	–	Unconditional return
RETC	–	Return with current result 1
RETCN	–	Return with current result 0

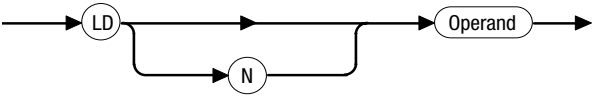
1) Instruction can be used in conjunction with a "(" parenthesis if required (deferring the operation). The instruction then refers to the result of the parenthesis expression, i.e. operand sequence. The deferred operation is continued after the ")".

The letter **N** at the last position of the instruction designation indicates the Boolean negation of the operand, e.g. instruction **LDN %I 0.0.0.0.0** when input I0.0.0.0.0 is 1 will enter a 0 into the working register.

07/01 AWB2700-1306-GB

The instruction sequences must be structured according to defined rules. The permissible instruction sequences are explained in detail in the sections 'Programming rules' and 'Description of the instructions' (see below in this chapter).

Load instructions



LD
Load

Permissible data types: all

Description
The value of the entered operand is loaded into the working register. The original content of the working register is overwritten. The operand is not changed. The data type of the stated operand defines the permitted data type for the following operands in this sequence.

Examples

Operation	Operand status	Comment
LD %I0.0.0.5	1	The signal status 1 on the I0.0.0.5 input is loaded into the working register
LD %IB0.0.0.0	11010100	The bit pattern at input byte IB0.0.0.0 is copied directly into the working register; the working register contains 11010100.

LDN
Load negated

Permissible data type: ANY_BIT

Description

The negated value of the entered operand is loaded into the working register. The original content of the working register is overwritten. The data type of the stated operand defines the permitted data type for the following operand in this sequence.

Example

Operation	Operand status	Comment
LDN %I0.0.0.0.3	1	The signal status at input I0.0.0.0.3 is negated and loaded into the working register; the working register is set to 0.

07/01 AWB2700-1306-GB

ST
Store instruction

Assignments

Permissible data types: all

Description

The content of the working register is assigned to the entered operand. The original value of the operand is overwritten. The data type of the entered operand must be the same as the data type of the data element in the working register. The data type of the working register content is defined by the data type of the variable that is assigned a value first. Further assignments can only be carried out with variables of the same data type. It is possible for one assignment to follow another one (see syntax chart on Page 39).

Examples

Operation	Status of working register	Comment
ST %Q0.0.0.0.0	1	The status 1 of the working register is assigned to the Q0.0.0.0.0 output. The previous operation forming the current result must provide a Boolean result.
ST Result1	1001100100001111	The bit pattern in the working register is copied into the Result1 variable which must be of type WORD.
LD 17 ST Value1 ST Value2		The variables Value1 and Value2 receive the value 17. They must be of the same data type.

STN
Store negated

Permissible data type: ANY_BIT

Description

The negated content of the working register is assigned to the operand entered. The original value of the operand is overwritten. The data type of the entered operand must be the same as the data type of the data element in the working register. The working register is not changed by the assignment instruction. A STN instruction can be followed by another ST or STN instruction (see syntax chart on Page 39).

Example

Operation	Status of working register	Comment
STN %Q0.0.0.0.1	1	The negated status 1 of the working register is assigned to output Q0.0.0.0.1 which receives the value 0. The previous operation forming the current result must provide a Boolean result.

Bistable data element

A bistable data element can be set to 1 by a fulfilled set condition, or it can be set to 0 by a fulfilled reset condition. The status is active until the condition is fulfilled for the opposite status. If both conditions are fulfilled at the same time, the data element accepts the status for which the condition was last processed. This is called priority setting or resetting.

07/01 AWB2700-1306-GB

S
Set

Permissible data type: BOOL

Description

The entered operand is set if the content of the working register is 1. If this set condition is not fulfilled, the operand will not be changed.
The working register is not changed.

Example

Operation	Operand status	Comment
LD %I0.0.0.0.0	1	If the I0.0.0.0.0 input has the status 1,
S %Q0.0.0.0.0	1	the Q0.0.0.0.0 output is set. The 0 status at input I0.0.0.0.0 has no effect on this output. Only a reset instruction R with fulfilled reset condition can set the output status to 0.

R
Reset

Permissible data type: BOOL

Description

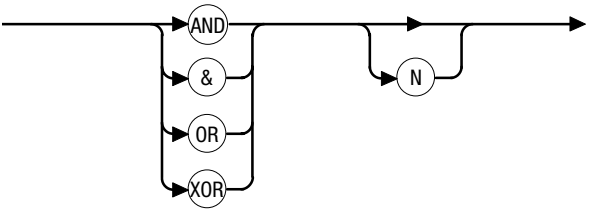
The entered operand is reset if the content of the working register is 1. If this reset condition is not fulfilled, the operand will not be changed. The working register is not changed.

Example

Operation	Operand status	Comment
LD %I0.0.0.0.1	1	If the I0.0.0.0.1 input has the status 1,
R %Q0.0.0.0.0	1	the Q0.0.0.0.0 output receives the status 0. The status 0 on the I0.0.0.0.1 input has no effect on this output. Only a set instruction with fulfilled set condition can set the output status to 1.

07/01 AWB2700-1306-GB

Logic operators



AND, &

Permissible data type: ANY_BIT

Description

AND operation linking the entered operand with the content of the working register. The result is stored in the working register. The original content of the working register is overwritten. The operand value is not changed.

With an AND operation of byte/word operands the individual bits of the entered operand are sequenced with the appropriate bits of the working register.

Examples

Operation	Operand status	Comment
LD %I0.0.0.0	1	The status 1 in the working register is ANDed with the status 0 of the variable Start and the result 0 is assigned to the variable Belt.
AND Start	0	
ST Belt	0	
LD %IB0.0.0	11001100	The individual bits of the bit pattern 11001100 in the working register are ANDed with the corresponding bits of the IB0.0.0.1 input byte and the result is transferred from the working register to the variable Result.
AND %IB0.0.0.1	10101010	
ST Result	10001000	

07/01 AWB2700-1306-GB

ANDN, &N negated

Permissible data type: ANY_BIT

Description

With binary operation the content of the entered operand is negated and is ANDed with the content of the working register. The result is stored in the working register. The original content of the working register is overwritten. The operand value is not changed.

With a negated AND sequence of byte/word operands, individual bits of the one's complement of the entered operand are ANDed with the corresponding bits of the working register.
The result is stored in the working register.

Examples

Operation	Operand status	Comment
LD %I0.0.0.1 ANDN Start ST Belt	1 0 1	The status 1 in the working register is ANDed with the negated status of the variable Start and the result 1 is assigned to the variable Belt.
LD %IB0.0.0 &N %IB0.0.0.1 ST Result	11001100 10101010 01000100	The individual bits of the IB0.0.0 bit pattern in the working register are ANDed with the corresponding bits of the one's complement of the IB0.0.0.1 input byte. The 01000100 result is transferred from the working register to the variable Result.

OR

Permissible data type: ANY_BIT

Description

OR operation linking the entered operand with the content of the working register. The result is stored in the working register. The original content of the working register is overwritten. The operand value is not changed.

With an OR sequence of byte/word operands the corresponding bits of each operand involved are sequenced.

Examples

Operation	Operand status	Comment
LD Input1 OR Input2	1 0	The status 1 of the variable Input1 is ORed with the status 0 of the variable Input2 and the result 1 is assigned to the variable OR_Bit.
ST OR_Bit	1	
LD Byte_1	11001100	The individual bits of the bit pattern 11001100 of the variable Byte_1 are ORed with the corresponding bits of the variable Byte_2 and the result is transferred from the working register to the variable OR_Byte.
OR Byte_2	10101010	
ST OR_Byte	11101110	

07/01 AWB2700-1306-GB

ORN
OR negated

Permissible data type: ANY_BIT

Description

With binary operations the content of the entered operand is negated and ORed with the contents of the working register. The result is stored in the working register. The original content of the working register is overwritten. The operand value is not changed.

With a negated OR sequence of byte/word operands, individual bits of the one's complement of the entered operand are ORed with the corresponding bits of the working register. The result is stored in the working register.

Examples

Operation	Operand status	Comment
LD Input1	0	The status 0 in the working register is ORed with the negated status of the variable Input2. The result 1 is assigned to the variable ORN_Bit.
ORN Input2	0	
ST ORN_Bit	1	
LD Byte_1	11001100	The individual bits of bit pattern 11001100 of the variable Byte_1 are ORed with the corresponding bits of the one's complement of the variable Byte_2. The result is transferred to the variable ORN_Byte.
ORN Byte_2	10101010	
ST ORN_Byte	11011101	

XOR
Exclusive OR

Permissible data type: ANY_BIT

Description

Exclusive OR operation of the entered operand with the content of the working register. The result is stored in the working register. The original content of the working register is overwritten. The operand value is not changed.

With an Exclusive OR sequence of byte/word operands, the corresponding bits of each operand involved are sequenced.

Examples

Operation	Operand status	Comment
LD Input1	1	The status 1 of the variable Input1 is Exclusive-ORed with
XOR Input2	1	the status 1 of the variable Input2 and the result 0 is assigned to the variable
ST XOR_Bit	0	XOR_Bit.
LD Byte_1	11101111	The individual bits of the bit pattern 11101111 of the variable Byte_1 are Exclusive-
XOR Byte_2	10111110	ORed with the corresponding bits of the variable Byte_2 and the result is transferred from the working register to the
ST XOR_Byte	01010001	variable XOR_Byte.

07/01 AWB2700-1306-GB

XORN
Exclusive OR negated

Permissible data type: ANY_BIT

Description

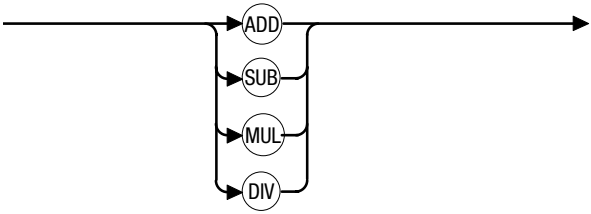
With binary operations the content of the entered operand is negated and Exclusive-ORed with the content of the working register. The result is stored in the working register. The original content of the working register is overwritten. The operand value is not changed.

With a negated Exclusive OR sequence of byte/word operands the individual bits of the one's complement of the entered operand are Exclusive-ORed with the corresponding bits of the working register. The result is stored in the working register.

Examples

Operation	Operand status	Comment
LD Input1	1	The status 1 of the variable Input1 is Exclusive-ORed with
XORN Input2	1	the negated status of the variable Input2 and the result 1 is assigned to the variable
ST XOR_Bit	1	XOR_Bit.
LD Byte_1	11101111	The individual bits of bit pattern 11101111 of the variable Byte_1 are Exclusive-ORed
XORN Byte_2	10111110	with the corresponding bits of the one's complement of the variable Byte_2. The result is transferred from the working
ST XORN_Byte	10101110	register to the variable XORN_Byte.

Arithmetic operators



ADD
Addition

Permissible data types: ANY_INT, REAL
(REAL only for PS4-300 and PS416)

Description

The entered operand value is added to the content of the working register and the result transferred to the working register. The original content of the working register is overwritten. The operand value is not changed.

Should the result of the addition exceed the range of values permitted for the data type selected, the carry will be transferred to the carry register.



The function ADD for the multiple addition of numeric operands and for adding time values and addresses is described in Chapter 5, “Functions”.

Example

Operation	Comment
LD 23	The content of the working register is added to the
ADD Summand	value of the variable Summand.
ST Sum	The result is transferred from the working register to the variable Sum.

07/01 AWB2700-1306-GB

SUB
Subtraction

Permissible data types: ANY_INT, REAL
(REAL only for PS4-300 and PS416)

Description
The entered operand (subtrahend) is subtracted from the content of the working register (minuend) and the result is stored in the working register. The original content of the working register is overwritten. The operand value is not changed.

Should the result of subtraction exceed the range of values permitted for the data type selected, the carry will be transferred to the carry register.



The function SUB for subtracting time values and addresses is described in Chapter 5, “Functions”.

Example

Operation	Comment
LD Minuend	The value of the variable Subtrahend is subtracted from the variable Minuend.
SUB Subtrahend	
ST Difference	The result is transferred from the working register to the variable Difference.

MUL
Multiplication

Permissible data types: ANY_INT, REAL
(REAL only for PS4-300 and PS416)

Description
The entered operand is multiplied with the content of the working register where the result is stored. The original content of the working register is overwritten. The operand value is not changed.

Should the result of multiplication exceed the range of values permitted for the data type selected, the carry will be transferred to the carry register.



The function MUL for the multiple multiplication of numeric operands and for multiplying addresses is described in Chapter 5, “Functions”.

Example

Operation	Comment
LD Multiplicand MUL Multiplier ST Product	The value of the variable Multiplicand is multiplied by the value of the variable Multiplier. The result is transferred from the working register to the variable Product.

DIV
Division

Permissible data types: ANY_INT, REAL
(REAL only for PS4-300 and PS416)

Description

The content of the working register (dividend) is divided by the value of the entered operand (divisor) and the result (quotient) is stored in the working register. The original content of the working register is overwritten. The operand value is not changed.

The remainder is stored in the carry register.



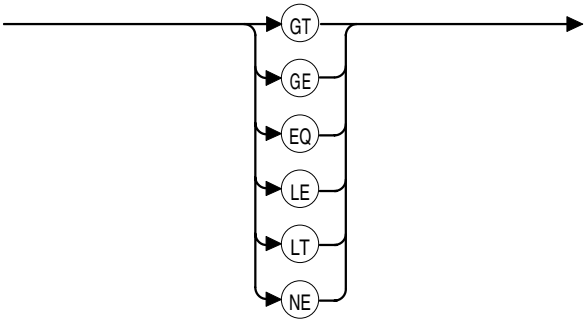
The function DIV for dividing addresses is described in Chapter 5, “Functions”.

Example

Operation	Comment
LD Dividend DIV Divisor ST Quotient	The value of the variable Dividend is divided by the value of the variable Divisor The result is transferred from the working register to the variable Quotient.

07/01 AWB2700-1306-GB

Comparison operators



Permissible data types: ANY_NUM, ANY_DATE, ANY_BIT, STRING, TIME

All data types except function blocks are permissible for EQ and NE.

Description
The following comparison operations are possible:

GT greater than

GE greater than or equal to

EQ equal to

LE less than or equal to

LT less than

NE not equal

The entered operand value is compared with the contents of the working register (operand value and working register must be of the same type!). The fulfilled comparison is indicated by the Boolean 1 in the working register; a 0 indicates an unfulfilled comparison. The original content of the working register is overwritten. The operand value is not changed.

The comparison operation involves a type conversion of the working register, i.e. the working register is of type ANY_INT before the comparison and afterwards always BOOL.

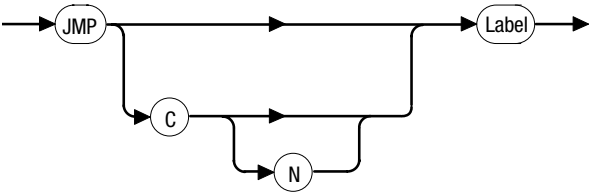
The Boolean result of a comparison can be used as a condition for invoking a function block, a jump to a jump label, further logic operations or a return from a POU to the next higher POU level.

Example

Label	Operation	Comment
	LD Number1	The value of the variable Number1 is
	EQ Number2	checked for equality with the value of the
	JMPC M1	variable Number2. If the comparison is
	LD 1	fulfilled, the conditional jump to label M1
	ADD Number1	is executed.
	ST Number1	The variable Number1 is not incremented if the comparison is fulfilled.
M1:	.	
	.	

07/01 AWB2700-1306-GB

Unconditional and conditional jumps



JMP
Unconditional jump

Permissible data type: Not relevant. In the jump instruction the jump target (label) is entered as the operand.

Description

The program is continued from the point specified as the jump target (label). The jump target must always be the start of a sequence which is marked with a label.

This instruction is to be considered as a sequence which consists of one instruction.

A jump is only possible within a POU.

Example

Label	Operation	Comment
M1:	LD Number1 EQ Number2 JMPC M2 . .	The program loop (incrementing the variable Number1 and the comparison by the variable Number2) is processed until the comparison is fulfilled.
	LD 1 ADD Number1 ST Number1 JMP M1	The unconditional jump to label M1 is carried out after each incrementation.
M2:	. .	

JMPC/JMPCN
Conditional jumps

Permissible data type: Not relevant. In the jump instruction the jump target (label) is entered as the operand.

Description
Conditional jump in conjunction with the Boolean content of the working register.

JMPC: Jump on "1"
If the working register has the value 1, the jump will be executed and the program will continue from the specified jump target. If the working register has the value 0, the jump will not be executed. The program continues with the next instruction after the jump instruction.

JMPCN: Jump on "0"
If the working register has the value 0, the jump will be executed and the program will continue from the specified jump target. If the working register has the value 1, no return will be executed. The program will continue with the instruction which follows the return instruction.

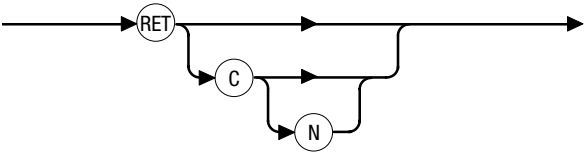
The jump target must always be a sequence start which is marked with a label. A jump is only possible within a POU.

Example

```
Label  Operation    Comment
M1:    LD Number1   If the comparison condition is fulfilled, the working
       EQ Number2   register will have the Boolean value 1 and thus
       JMPC M2      represent the fulfilled jump condition. Program
                    processing is continued at the jump label M2.
       .
       .
       LD 1         This section of the program is then skipped.
       ADD
       Number1
       ST Number1
M2:    JMP M1
       .
       .
```


07/01 AWB2700-1306-GB

Unconditional and conditional returns



RET
Unconditional return

Permissible data type: no operand is entered as the Return instruction.

Description

The RET instruction causes an unconditional return to the invoking POU, e.g. with a program POU, a return to the system program. With the return to a POU the invoking POU is continued at the point at which the subordinate POU was invoked. The deferred operations are now processed.

Example

Label	Operation	Comment
	LD Error	If Error = 0, the RET instruction is
	JMPCN M1	skipped. If Error = 1, RET is executed, i.e.
	RET	the POU is left and the program is
M1:	.	continued in the invoking POU.
	.	

RETC/RETCN
Conditional returns

Permissible data type: no operand is entered as the Return instruction.

Description
Conditional return in conjunction with the Boolean content of the working register.

RETC: Return on 1
If the working register has the value 1, the return will be to the next-higher POU, i.e. with a program POU to the system program. If the working register has the value 0, no return will be executed. The program will continue with the instruction which follows the return instruction.

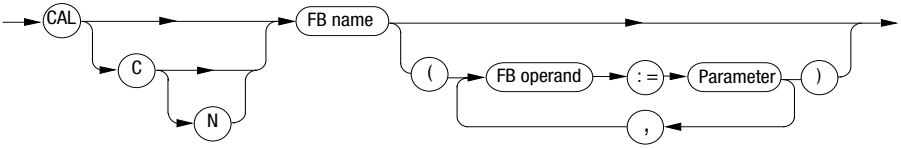
RETCN: Return on 0
If the working register has the value 0, the return will be to the next-higher POU, i.e. with a program POU to the system program. If the working register has the value 1, the return will not be executed. The program continues with the next instruction after the return instruction.

Example

Operation	Comment
LD Error	If Error = 1, the POU will be quit and the program
RETC	will continue in the invoking POU.
.	
.	

07/01 AWB2700-1306-GB

Unconditional and conditional invocation of function blocks



CAL
Unconditional invocation of function blocks

Description

The program is continued in the function block entered as operand. The unconditional invocation may only be programmed after a sequence is completed and is not permitted inside parenthesized operations.

Two FB call variations in accordance with the IEC standard are illustrated in the IL program example POU on Page 72.

The direct parameter passing within the function block call of inputs and outputs with instruction sequences is a Moeller alternative of the function block call. More information is given in Chapter 7, “Language Extension”.

Example

Operation	Comment
.	
.	
ST Output2	
CAL Pulse generator_2_Hz (Set:=start, period:=timevalue)	After the allocation (end of sequence) the function block pulse_generator_2_Hz is called.

CALC/CALCN
Conditional invocation of function blocks

Description
Conditional function block invocation in conjunction with the Boolean content of the working register.

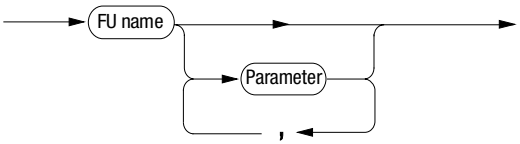
CALC: FB call if 1
If the working register has the value 1, the function block is called which is entered as the operand. If the working register has the value 0, no function block invocation takes place. The program is continued with the instruction after the CALC instruction.

CALCN: FB call if 0
If the working register has the value 0, the function block is called which is entered as the operand. If the working register has the value 1, no function block invocation takes place. The program is continued with the instruction after the CALCN instruction.

Example

Operation	Comment
...	
ST Output2	
LD Result3	If the Boolean variable Result3 = 1,
CALC Pulse generator_2_Hz	function block Pulse_generator_2_Hz
(Set:=Start,	is called.
Period:=TimeValue)	
...	

Invocation of functions



The function is called by entering its name – there are no keywords for calling functions (see also Section “Calling a function in IL” on Page 69).

07/01 AWB2700-1306-GB

Use of integer constants in the instruction section

The IEC/EN 61131-3 standard defines three different kinds of literals (constants) for integer values:

Unsigned decimal integers, e.g. 127

Signed decimal integers +3000

Unsigned literals with base value, e.g. 2#1111_1111.

These three types of constants do not require any explicit data type definition. The data type is assigned within the context of the command sequence it is used in.

The value of these constants can be determined when they are directly assigned to a fixed variable or parameter. However, the result will not be clear when used with operators and overloaded functions with the input parameters ANY or ANY_INT.



Overloaded functions are functions that can be used for several data types. Refer to Page 131 for more information.

Example of "Integer constants"
The instruction sequence in a PS4-200 program does not supply the required result –5 but the value 32763.

```
LD  1
SUB 11
DIV 2
ST  int_var
```

The compiler selects the data type UINT for the sequence result because of the unsigned decimal integers used. The subtraction causes an overflow. The result is converted from a binary to a decimal value when assigned to the INT variable int_var.

However, a small modification in the example will

ensure the correct result:

```
LD  +1
SUB +11
DIV +2
ST  int_var
```

Observe the following points when working with integers:

- Always use signed constants when working with signed data types.
- Always use unsigned constants when working with unsigned data types.
- When calling one of the IEC selection functions LIMIT, MAX, MIN, MUX or SEL avoid using unsigned and signed constants at the same time.
- Use constants with base definitions only in conjunction with ANY_BIT data types. The IEC standard knows no binary format for the unique absolute allocation to decimal values apart from one exception - the numbers 0 and 1 which are used to replace FALSE and TRUE.

07/01 AWB2700-1306-GB

Calling a function in IL

A function is called up by entering the function name and by transferring all necessary input parameters.

The first parameter is transferred to the function via the working register. Further parameters are entered after the function name, separated by a comma.

The result of a function process is supplied as a return value with the name of the function and is made available for the POU containing the invocation via the working register.

Functions cannot store information. A function with the same parameters thus always supplies the same return value when it is called up.

With every function call, local variables are initialised according to their data type as shown in the table Table 3-1 on Page 41. If a function call has been terminated without the function supplying a result, this automatically resets the initialisation value of the type concerned.

Example: Sum function
Function SUM and its use in the program pou2 displayed in IL

```
FUNCTION SUM : INT
VAR_INPUT
  A : INT;
  B : INT;                (* Declaration section*)
  C : INT;
END_VAR
  LD A
  ADD B                   (* Instruction section*)
  ADD C
  ST SUM
END_FUNCTION
```

Example: Program pou2
Invocation of the SUM function in the program pou2

```
PROGRAM pou2
VAR
   SUMMAND1 : INT:= 1;
   SUMMAND2 : INT:= 2;
   SUMMAND3 : INT:= 3;
   RESULT : INT;
END_VAR
   LD SUMMAND1                    (* First operand in the working register *)
   SUM SUMMAND2, SUMMAND3          (* Function invocation with parameter transfer *)
   ST RESULT                      (* Read sum of the working register and assign to
the variable RESULT *)
END_PROGRAM
```

Call of a function with direct assignment
A Moeller variation of function calls allows the inputs (VAR_INPUT parameters) to be assigned directly using instruction sequences when the function is called. This call variation is not part of the IEC standard.

The instruction sequence is processed before the function is called and the sequence result is assigned to the input directly. Other parameter allocations are then carried out and the function is called.
The parameter allocations are carried out from left to right.

Example “Connecting inputs of a function”

```
VAR
  a : word;
  b : word;
END_VAR;

ld    a
shr   (b
      word_to_uint
      )
```

Calling a function block in IL

The IEC standard provides two equal-ranking

07/01 AWB2700-1306-GB

procedures for the invocation of a function block. They are shown in the pou3 program with the aid of two instances of a function block.

Example: Function block meanval
A function block for simple averaging and its application in pou3

Prototype of the meanval function block in IL

```
FUNCTION_BLOCK meanval

VAR_INPUT
  Reset : BOOL;
  Value1 : USINT;
  Value2 : USINT;
END_VAR

VAR_OUTPUT
  Mean_value : USINT;
END_VAR

  LD Value1
  ADD Value2
  DIV 2
  ST Mean_value
END_FUNCTION_BLOCK
```

Example: Program pou3

Instantiation of the meanval function block in the pou3 program with two different invocation procedures:

```
PROGRAM Pou3

VAR
  Instance1_Name : meanval;
  Instance2_Name : meanval;
  MeasValue1 AT %IB0.0.0.0 : USINT;
  MeasValue2 AT %IB0.0.0.1 : USINT;
  Result1 AT %QB0.0.0.0 : USINT;
  Result2 AT %QB0.0.0.1 : USINT;
  Reset AT % I0.0.0.2.0 : BOOL;
END_VAR

          (* 1st procedure FB invocation *)
  CAL Instance1_Name (Reset:= reset,
                      Value1:= MeasValue1,
                      Value2:= MeasValue2)
  LD Instance1_Name.Mean_value
  ST Result1
          (* 2nd procedure FB invocation *)
  LD Reset
  ST Instance2_Name.Reset
  LD MeasValue1
  ST Instance2_Name.value1
  LD MeasValue2
  ST Instance2_Name.value2
  CAL Instance2_Name
  LD Instance2_Name.Mean_value
  ST Result2
END_PROGRAM
```

The parameters for the function block are assigned when it is called up with CAL and the appropriate instance name (CAL Instance1_Name). The previously defined instance variables are assigned to the input parameters of the function block. The individual operands are specified between round brackets, separated by commas. The value of the output parameter Instance1_Name.Mean_value is transferred after the invocation of the variable Result1:

07/01 AWB2700-1306-GB

```
LD Instance1_Name.Mean_value
ST Result1
```

In the second procedure all instance-defined input parameters are individually transferred to the function block parameters by means of the working register before the invocation of the function block. The invocation of the function block itself does not contain parameters. The function block results are stored and can be addressed with the LD instruction.

Another procedure for the function block invocation is a Moeller variant. The parameters for the input and output operands are transferred when invoking the function block, separated by the character ¦ .

Example: Function block invocation with ¦

```
CAL Instance3_name(Reset := Reset,
                   Value := Measvalue3
                   |
                   Result3 := Mean_value)
```

Call of a function block with direct assignment

A Moeller variation of function block calls allows the direct assignment of function block inputs (VAR_INPUT parameters) and outputs (VAR_OUTPUT parameters) via instruction sequences in the call. This call variation is not part of the IEC standard.

With conditional function block calls, the input and output allocations are only processed if the function block is called.

Assignment of FB input parameters

Before the function block call the instruction sequence is processed and the sequence result is assigned to the function block input. Other direct parameter passing is then carried out and the function block is called.

Example “Assignment of inputs”

```
(* FUNCTION_BLOCK meanval
  VAR_INPUT
       Value1 : USINT;
       Value2 : USINT;
  END_VAR

  VAR_OUTPUT
       meanval : USINT;
  END_VAR                *)

PROGRAM Pou4

VAR
  Instance : meanval;
  Measval1 AT %IB0.0.0.0 : USINT;
  Measval2 AT %IB0.0.0.1 : USINT;
END_VAR

CAL meanval(Value1 := (Measval1
                       add 2
                       ),
            Value2 := (Measval2
                       add 3
                       )
            )
```


07/01 AWB2700-1306-GB

Assignment of FB outputs

After the function block is called the instruction sequence is processed and the sequence result is assigned to the variable in front of the assignment operator ":=". The first operand of the sequence must be the output parameter of the function block called.

Example "Assignment of outputs"

```
VAR
  Instance : meanval;
  Measval1 AT %IB0.0.0.0 : USINT;
  Measval2 AT %IB0.0.0.1 : USINT;
  Result AT %QB0.0.0.0 : USINT;
  Average : USINT;
END_VAR

CAL meanval(Value1 := Measval1,
           Value2 := (Measval2
                            add 3
                          )
           |
           Result := (Average
                            sub 2
                          )
           )
```

Programming in structured text (ST)

Statements

The instruction section of a POU written in ST consists of at least one ST statement or a sequence of statements. Each statement must be terminated with a semicolon. This unambiguous separator allows a statement to cover several lines, and several statements can be written on the same line. Comments can be positioned at any place if required.

Elementary statements

Value assignment	`CV := CV +1;`
Empty statement	`;`

Branching within a POU

Conditional statement, Alternative statement	`IF d>10 THEN d:=1;`
	`ELSIF d=2 THEN RETURN;`
	`ELSE d:=d+1;`
	`END_IF;`

Multi-selection	`CASE i MOD 15 OF`
	`  -2,1,3: i := 6;`
	`  4..10,12: i := i+1;`
	`  ELSE RETURN;`
	`END_CASE;`

Loops

Loop with condition	`WHILE i<=100 DO`
	`  F[i]:=i; i:=i+2;`
	`END_WHILE;`

Loop with termination condition	`REPEAT`
	`  F[i]:=i; i:=i+2;`
	`  UNTIL i>100;`
	`END_REPEAT;`

Count loop	`FOR i:=0 TO 100 BY 2 DO`
	`  F[i]:=i;`
	`END_FOR;`

Premature termination of loop	`EXIT;`

Branching within the application

Call of a function block with assignment of inputs and outputs	`CTD_Instance( CD := %I0.0.0.0.0,`
	`  Load := 1,`
	`  PV := ,`
	`  Q =>,`
	`  CV => CounterStatus );`

POU Exit	`RETURN;`

07/01 AWB2700-1306-GB

Value assignment
The value of the expression to the right of the assignment operator is assigned to the data element on the left. The data element can be a variable, a function block parameter or a subrange of it, such as a structure component or an array element. The data type of the assigned value must correspond with that of the data element, i.e. the type is not adapted automatically!

Syntax:

```
Data element := Expression ;
```

Example:

```
Index := Index + 3;
F[Index]:= 3.5 + PI *2.0;
```

Empty statement
As instruction sequences must always consist of at least one statement, loops or alternative statements must not be empty. A single semicolon can then be used as an empty statement for a statement sequence that is not fully specified. This therefore fulfills syntax requirements without generating additional machine code.

The empty statement can only be used effectively in multiple-selection tasks:

Example:

```
(* Prevents the processing of the ELSE branch
when the expression yields 0 *)
CASE i MOD 15 OF
  0:       ;
  -2,1,3:  i := 6;
  ELSE     i := i+1;
END_CASE;
```

Conditional statement
A conditional statement will only execute one or several statements if a condition is fulfilled, i.e. corresponding to the value TRUE. Any expression that supplies a Boolean result can be specified as a condition. The expression is calculated directly before the construct is processed.

Syntax:

```
IF Expression THEN StatementSequence END_IF ;
```

Example:

```
IF     A < 4 THEN
       C := T#5s;
       A := 4;
END_IF;
```

Single alternative statement
A single alternative statement will either execute the first or the second sequence of statements, depending on whether the condition is true or false. Any expression that supplies a Boolean result can be specified as a condition. The expression is calculated directly before the construct is processed.

Syntax:

```
IF Expression THEN StatementSequence1
ELSE StatementSequence2 END_IF ;
```

Example:

```
IF A < 4 THEN
  C := T#500ms;
  A := 4;
ELSE
  C := T#100ms;
END_IF;
```


07/01 AWB2700-1306-GB

Multiple alternative statement

A multiple alternative statement executes the first of the statement sequences listed, that fulfills the preceding condition and then exits the IF construct. If none of the specified conditions are fulfilled, any statements specified after the ELSE are executed. Any expression that supplies a Boolean result can be specified as a condition. The expression is calculated directly before the IF or ELSIF keyword is processed. Processing is therefore more efficient than when processing several successive conditional statements. Any number of ELSIF branches can be listed, the ELSE branch is optional.

Syntax:

```
IF Expression1 THEN StatementSequence1
ELSIF Expression2 THEN StatementSequence2
ELSIF Expression3 THEN StatementSequence3
...
ELSE StatementSequenceN END_IF ;
```

Example:

```
(* The A = B comparison is only executed
 * if A < B is not fulfilled. If this is also
 * not fulfilled, C is unchanged.
 *)
IF A < B THEN
  C := T#500ms;
ELSIF A = B THEN
  C := T#100ms;
END_IF;
```

Multiple selection (CASE statement)
The CASE statement is used to select from several alternative statement sequences. The selection is based on the value of an ANY_INT expression.

When a CASE statement is processed, a check is made to see whether the value of the expression is contained in one of the specified value lists. If the value matches a specified value, the assigned statement sequence is executed and the CASE construct is closed. If the value of the expression is not contained in any of the value lists, the instruction sequence after the ELSE branch will be executed.

Either a sequence of individual values (integer constants separated by commas), a specified interval, or a combination of both can be used as a value list.

A value can only be listed once in a CASE statement, otherwise an unambiguous selection is not possible. Not all values have to be recorded that the expression can take on. All listed values must be able to be represented with the data type of the expression.

Complex expressions should be avoided to ensure optimum performance.

Syntax:

```
CASE Expression OF
ValueList1 : StatementSequence1
ValueList2 :StatementSequence2
...
ELSE StatementSequenceN
END_CASE ;
```


07/01 AWB2700-1306-GB

Example:

```
CASE i MOD 15 OF
  0:       RETURN;
  -2,1,3:  i := 6;
  4..10,12:i := i+1;
END_CASE;
```

WHILE loop

Executes a sequence of statements as long as the execution condition is fulfilled. The condition is formulated as a BOOL expression and is computed before every loop pass. The loop is skipped if the condition is not fulfilled from the start.

Syntax:

```
WHILE Expression DO Statement sequence END_WHILE ;
```

Example:

```
Maximum:=0; Index := FirstIndex;
WHILE Index<= LastIndex DO
  Maximum := MAX(Buffer[Index], Maximum);
  Index := Index +1;
END_WHILE;
```

REPEAT loop

Executes a sequence of statements until a termination condition is fulfilled. The condition is formulated as a BOOL expression and is computed after every loop pass. Unlike the WHILE loop, the statements are run through at least once, even if the termination condition was fulfilled from the start.

Syntax:

```
REPEAT Statement sequence UNTIL Expression END_REPEAT ;
```

Example:

```
Maximum:=0; Index := FirstIndex;
REPEAT
  Maximum := MAX(Buffer[Index], Maximum);
  Index := Index +1;
UNTIL Index = LastIndex
END_REPEAT;
```

FOR loop

A FOR statement is used to repeat a statement sequence as long as a loop variable is within a specified value range. The loop variable must be an ANY_INT type elementary variable. The loop requires an initial and end value to be specified. Both values must be loop variables of the same type.

Before the start of the loop, the loop variable is set to the initial value (start assignment) and after each pass incremented accordingly (with positive step width) or decremented (with negative step width) by the specified step width. If the step width is not specified, this is set to 1.

A check is made whether the variable is still within the specified value range before every pass. If this is the case, the statement sequence is executed. Otherwise, the program will branch to behind the end of the loop. The statement sequence is not executed if the initial value is greater than the end value with a positive step width. With a negative step width the loop will likewise be skipped, if the initial value is less than the end value.

The value of the loop variable within the loop can be used to compute expressions or for other branches. Sucosoft also allows the variable state to be manipulated directly within the loop. However, this programming method is not recommended since the resulting behaviour is not specified by the IEC standard and may be implemented differently in other programming systems.

07/01 AWB2700-1306-GB

Syntax:

```
FOR Loop variable := ExpressionInitialValue TO ExpressionEndValue BY Step width
DO StatementSequence
END_FOR ;
```

Initial and end value can be any expressions. Complicated expressions should not be used for the end value to ensure optimum performance. Only an integer or signed constant should be used as a step width.

Example:

```
Maximum:=0;
FOR Index:= FirstIndex TO LastIndex
  Maximum := MAX(Buffer[Index], Maximum);
END_FOR;
```

Loop exit

The EXIT statement can be used to exit a loop before the termination condition is satisfied. The control passes to the next statement after the loop end. EXIT causes the control to exit the loop directly surrounding the EXIT statement. With nested loops, only the inside loop is left.

Syntax:

```
EXIT ;
```

Example:

```
Maximum:=0; Index := FirstIndex;
WHILE Index<= LastIndex DO
 Maximum := MAX(Buffer[Index], Maximum);
 IF Maximum >= 1000 THEN EXIT; END_IF;
 Index := Index +1;
END_WHILE;
```

POU Exit
A RETURN statement causes the control to exit the currently processed POU and return to the call location. The command in the program POU causes the control to return to the PLC operating system, i.e. the program cycle is stopped prematurely.

A RETURN entry at the end of the instruction section of a POU is not necessary since a missing statement is automatically added by the compiler.

Syntax:

```
RETURN ;
```

Example:

```
IF Maximum >= 1000
THEN RETURN; END_IF;
```

Call of function blocks
A function block is called by stating the instance name (the variable name specified in the declaration) followed by a list of parameter assignments enclosed in parentheses.

The specified values are assigned to the function block inputs before it is called. This is followed by a branch to the function block and the outputs are assigned to the corresponding outputs after the function block is left.

There are two ways of transferring the parameters to the function block and calling it:

07/01 AWB2700-1306-GB

Method 1 Connection of inputs and outputs by assignments and calling the function block without parameters:

```
(* the following function block is predefined:

FUNCTION_BLOCK meanval

VAR_INPUT
  Reset : BOOL;
  Value1 : USINT;
  Value2 : USINT;
END_VAR

VAR_OUTPUT
  Result : USINT;
END_VAR

END_FUNCTION_BLOCK
*)

VAR
  Instance:MeanValue;
  ResetSwitch AT %I0.0.0.0.0 : BOOL;
  MeasValue AT %IB0.0.0.1 : BYTE;
  Result AT %QB0.0.0.0 : BYTE;
END_VAR

Instance.Reset := ResetSwitch;
Instance.Value1 := MeasValue;
Instance.Value2 := 2+MeasValue;
Instance(); (* the call is located here*)
Result := Instance.Result;
```

Method 2 Connection of inputs and outputs by assignments within the call command:

```
VAR
   Instance:meanval;
   ResetSwitch AT %I0.0.0.0.0 : BOOL;
   MeasValue AT %IB0.0.0.1 : BYTE;
   Result AT %QB0.0.0.0 : BYTE;
END_VAR

Instance(Reset := ResetSwitch, (* := for connecting inputs *)
Value1 := MeasValue ,
Value2 := 2+MeasValue ,
Result => Result (* => for connecting the outputs *)
);
```

The connection of VAR_IN_OUT inputs (passing parameters) can only be called using the second method with the assignment operator ":=". Only a writable data element and no other expression can be located on the right hand side of the assignment. VAR_IN_OUT inputs must be connected with every function block call.

Call of functions
Unlike function block calls, function calls are not separate statements but expressions. They can only therefore be used within expressions.

A function is called by stating its name, and by transferring the parameter list for the individual operands of the function, the parameter list must be set in parentheses. The operands are separated by commas.

The order of operands must match the interface of the function and all inputs of the function must be assigned. Unlike the call of function blocks, the use of formal parameters (stating of input names) is not supported by Sucosoft.

Example: call of the string function REPLACE

```
(* Interface assignment *)

FUNCTION REPLACE : STRING
  VAR_INPUT
       IN1 : STRING;
       IN2 : STRING;
       L : ANY_INT;
       P : ANY_INT;
  END_VAR
END_FUNCTION
```


07/01 AWB2700-1306-GB

```
VAR
  OldText:STRING:='abbCDEe';
  NewText:STRING;
END_VAR

NewText:=REPLACE(OldText, 'cd', 4, 2+1);
(* NewText = 'abcde' *)
```

In the previous example, the value of the variable OldText is assigned to function input IN1, the constant cd to input IN2, the constant 4 to input L and the result of the express 2+1=3 assigned to input P. This is followed by the call of the function REPLACE. At the end the function's return value is assigned to the variable NewText.

A function input can therefore be assigned any expression:

- Constant/Literal
- Data element (variable, array element, function block parameter, structure component)
- Call of another function
- Expression with operators.

Expressions, operands and operators
An expression is not an independent statement but a calculation that yields a result for further processing in the statement. An expression consists of one or several operands that are linked together by means of operators. The following operands are permissible:

- Constant/Literal
- Data element (variable, array element, function block parameter, structure component)
- Function call
- Another expression

Most ST operators link two operands together and are therefore called binary operators. They are written between the operands (e.g. A+B).

The sign operators “+” and “–”, as well as the NOT operator are unary operators, i.e. they only work with one operand. A unary operator is located directly in front of the operand (e.g. –A).

Table 3-2: Overview of operators

Operation	Symbol	Example	Precedence
Parenthesization	( )	3*(4+5) yields 27	1 (high)
Function call	Name( )	CONCAT ('A','B','C') yields 'ABC'	2
Two’s complement	-	-5	3
Two's complement used twice	+	+5 yields 5 with signed type	3
One's complement	NOT	NOT 2#00001111 yields 2#11110000	3
Exponentiation	**	2**3 yields 8	4
Multiplication	*	10.0*3.5 yields 35.0	5
Division	/	5/2 yields 2	5
Modulo	MOD	5 MOD 2 yields 1	5
Addition	+	T#5s + T#1s yields T#6s	6
Subtraction	-	2 - 3 yields -1	6
Comparison	<, >, <=, >=	20 > 10 yields TRUE	7
Equality	=	20 = 10 yields FALSE	8
Inequality	<>	3.5 <> 2.0 yields TRUE	8
Logic AND	AND, &	TRUE & FALSE yields FALSE	9
Logic EXCLUSIVE-OR	XOR	TRUE XOR FALSE yields TRUE	10
Logic OR	OR	16#F0 OR 16#0F yields 16#FF	11 (low)

07/01 AWB2700-1306-GB

The precedence of the operators determines the order in which sub-expressions are calculated in complex expressions. A typical example of this is the multiplication calculation in arithmetic:

5+3*2 = 5+(3*2) =11 * greater precedence than +

–2+3 = (–2)+3 = 1 sign has greater precedence

The left operator has greater precedence over the right operator in expressions with adjoining binary operators of the same precedence:

3–2+1 = (3–2)+1 = 2 – greater precedence than +

Parenthesization can be used to break the specified precedence rules. The use of parentheses is recommended to ensure greater legibility, especially when the precedence rules are not immediately obvious:

(5+3)*2 = 16

–(2+3) = –5

3–(2+1) = 0

((A=B) OR (A=C)) AND (A<>0)

An equivalent function call can always be used instead of the operators. Some of these functions can also be called with more than two input operands:

-A	NEG(A)
NOT A	NOT(A)
A**B	EXPT(A,B)
A*B*C*D	MUL(A,B,C,D)
A/B	DIV(A,B)
A MOD B	MOD(A,B)
A+B+C+D	ADD(A,B,C,D)
A-B	SUB(A,B)
A<B	LT(A,B)
A>B	GT(A,B)
A<=B	LE(A,B)
A>=B	GE(A,B)
A=B	EQ(A,B)
A<>B	NE(A,B)
A&B&C&D	AND(A,B,C,D)
A XOR B XOR C	XOR(A,B,C)
A OR B OR C	OR(A,B,C)

The result type of an operator call, and the permissible data types for the input operands are the same as for the function calls. Further information on this is provided in Chapter 5, “Functions” from Page 127.

The function EXPT( ) has never been available in any PLC series. The exponentiation operator ** is therefore also not supported by the code generation functions.

07/01 AWB2700-1306-GB

4 Graphical Programming Languages

Introduction

Sucosoft provides two graphical programming languages that comply with the IEC/EN 61131-3 standard: Ladder diagram (LD) and Function block diagram (FBD).

The first section of this manual provides a general description of the programming languages LD and FBD and an overview of the manufacturer-defined functions and function blocks, together with practical programming examples in FBD. The second section describes working with LD and FBD in the POU editor and when testing and commissioning. The Sucosoft S40 User Interface reference manual provides basic information on working with the software.

A program organisation unit created in the programming languages LD and FBD consists of a declaration section and an instruction section. It therefore has the same structure as a POU in an IL or ST version.

In the programming languages LD and FBD the declaration section is shown as text just as it is in the IL programming language. The instruction section is represented graphically in LD and FBD. The programming languages IL, LD and FBD are interchangeable. You can thus create a program in LD and display and process it in IL and FBD. You will find further information in the last section of the description of the POU editor LD/FBD under ‘Changing between the programming languages’.

In the instruction section the individual program sequences are represented in networks. The network header includes the serial network number, below which it is possible to enter a comment. When changing from IL to LD or FBD all line comments of a sequence which were entered in the IL programming language will automatically be shown in the header of the corresponding network.

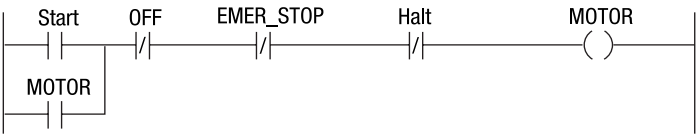
As in the programming language IL, the user is provided with all operations, functions and function blocks in the programming languages LD and FBD. Binary operations are represented by different graphical symbols in LD and FBD. In both languages the functions and function blocks are represented as rectangular graphical symbols with input operands on the left and output operands on the right side.

07/01 AWB2700-1306-GB

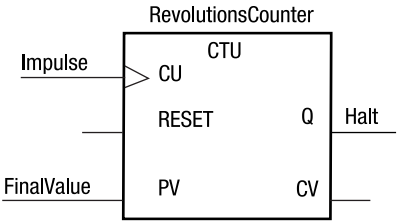
Ladder diagram

```
VAR
   START       AT %I0.0.0.0.0: BOOL;     (*Motor ON (make contact)*)
   OFF         AT %I0.0.0.0.1: BOOL;     (*Motor OFF (break contact)*)
   EMERG_STOP  AT %I0.0.0.0.2: BOOL;     (*Emergency-stop switch (break contact)*)
   Motor       AT %Q0.0.0.0.0: BOOL;     (*Motor drive*)
   Impulse     AT %I0.0.0.0.3: BOOL;     (*Impulse encoder*)
   FinalValue        : INT:=20;          (*Comparison value for the counter*)
   Halt              : BOOL;             (*Stop signal if counter=20*)
   RevolutionsCounter : CTU;             (*Impulse counter*)
END_VAR
```

0001

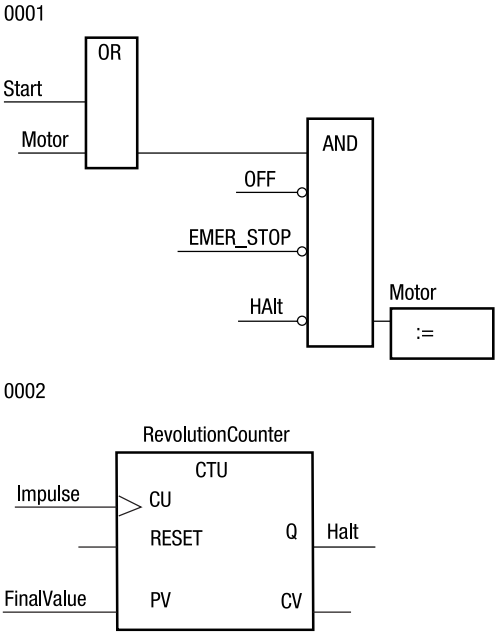


0002



Function block diagram

```
VAR
   START       AT %IO.0.0.0.0: BOOL;(*Motor ON (make contact)*)
   OFF         AT %IO.0.0.0.1: BOOL;(*Motor OFF (break contact)*)
   EMERG_STOP  AT %IO.0.0.0.2: BOOL;(*Emergency-stop switch (break contact)*)
   Motor       AT %QO.0.0.0.0: BOOL;(*Motor drive*)
   Impulse     AT %IO.0.0.0.3: BOOL;(*Impulse encoder*)
   FinalValue       : INT:= 20;(*Comparison value for the counter*)
   Halt             : BOOL; (*Stop signal if counter=20*)
   RevolutionsCounter : CTU; (*Impulse counter*)
END_VAR
```

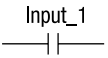


07/01 AWB2700-1306-GB

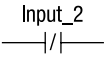
Programming in ladder diagram (LD)

The ladder diagram programming language is designed for programming logic operations of binary variables.

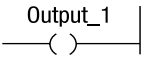
Programs are represented in LD by the following graphical elements:



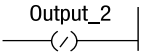
Scan a variable for status 1



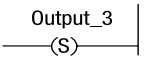
Scan a variable for status 0



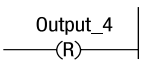
Assign a current result (CR) or a constant to a variable



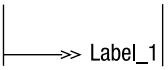
Assign a negated current result or a negated constant to a variable



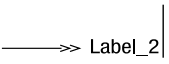
Set a variable if current result = 1



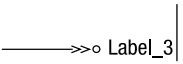
Reset a variable if current result = 1



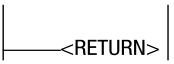
Unconditional jump to a label



Jump to a label if current result = 1



Jump to a label if current result = 0



Unconditional return to the invoking POU



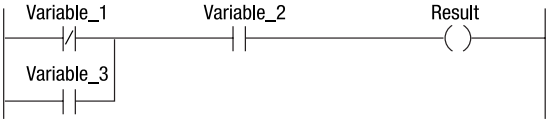
Return to the calling POU if CR = 1



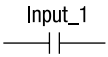
Return to the calling POU if CR = 0

The representation of a program sequence in LD is similar to a current path running between two current rails. The rails are the vertical outer lines in the display. A logic sequence is made by combining individual contacts in series or in parallel. There is no

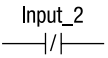
restriction in the number of contacts that can be connected in parallel or in series.



Two contact symbols are available for Boolean input variables:



Contact symbol for a variable that requires the status 1 so that a logic sequence is fulfilled (open contact). The status 1 of variables assigned to physical addresses corresponds to a break contact that has not been actuated or a make contact that has been actuated.

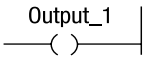


Contact symbol for a variable that requires the status 0 so that a logic sequence is fulfilled (closed contact). The status 0 of variables assigned to physical addresses corresponds to a break contact that has been actuated or a make contact that has not been actuated.

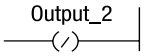
The result of a logic sequence of input variables can be used as the condition of a subsequent operation. These operations may consist of:

- Assignments
- Jumps to a point in a program that has been assigned a jump label
- A return to the invoking POU

The current result can be assigned to the output variable directly or after being negated. There are therefore two symbols for output variables:



Direct assignment of the result to the output variable.(“Coil”)



Assignment of the negated result to the output variable.(“Negated coil”)

07/01 AWB2700-1306-GB

There is no limit to the number of symbols that can be placed next to each other or above each other.

Variable names can be specified with a maximum length of 64 characters. Longer variable names are shown, however, with up to 3 lines above the contact or output symbol.

AND sequence

An AND sequence is created in LD by connecting contacts in series. In order to fulfill the AND condition, the individual input variables must simultaneously have the states required by the contact type.

0001

AND_Sequence



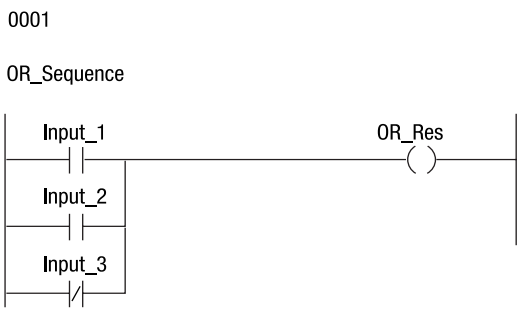
The AND condition shown is fulfilled if the input variables have the following states:

Input variable	**Status**
Input_1	1
Input_2	0
Input_3	1

The AND_Result output variable is then in status 1.

OR sequence

An OR condition is created in LD by connecting contacts in parallel. The OR condition is fulfilled if at least one input variable has the status required for contacting.



The OR condition shown is fulfilled if at least one of the input variables has the following status:

Input variable	**Status**
Input_1	1
Input_2	1
Input_3	0

The output variable OR_Result has the status 1.

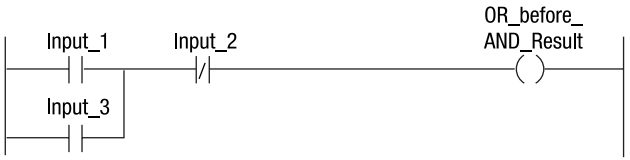
Mixed sequences

Mixed sequences are combinations of AND and OR sequences. The sequence is fulfilled if the current path between the left current rail and the output symbol is closed via at least one path.

07/01 AWB2700-1306-GB

0001

OR-before-AND-Sequence



The condition shown is fulfilled if the input variables have the following states:

Input variable	**Status**
Input_1 or Input_3	1
Input_2	0

The output variable OR_before_AND_Result then has status 1.

0001

Exclusive-OR-Sequence



The sequence is fulfilled if one of the two input variables has the status 1, i.e. the input variables are different. The output variable XOR_Result then has status 1.

Negated assignment

An output variable can also be assigned a negated current result. This status is indicated by an oblique stroke in the middle of the output symbol.

0001

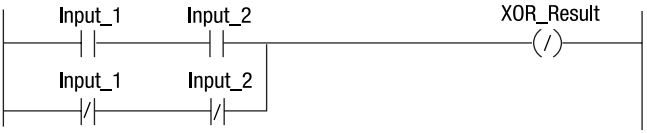
Negated AND-Sequence



The input variables are ANDed. The current result is negated and assigned to the AND_negated output variable.

0001

Negated equivalance results in an Exclusive-OR-Sequence



The sequence shown represents an equivalence function. The sequence is only fulfilled if both input variables have the same status. The current result will be 0 if the states of the input variables are different. The output variable XOR_Result is therefore only 1 if the equivalence is not fulfilled, i.e. if the input variables are different. The sequence shown therefore corresponds to an Exclusive OR sequence.

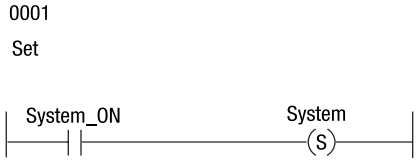
07/01 AWB2700-1306-GB

Retentive assignment

The current result can be assigned to the output variable retentively. The output will then retain its set or reset status.

Set (latch coil)

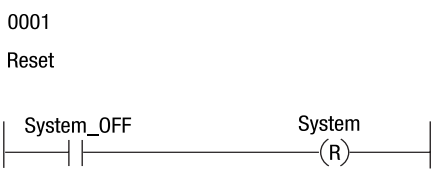
If the current result is 1, the output variable is set, i.e. is set to 1. The 0 status of the current result has no effect on the value of the output variable. Setting is represented by an output symbol with an S in the middle.



Just a momentary switching to 1 status on the System_ON input will be enough to switch on the plant permanently.

Reset (unlatch coil)

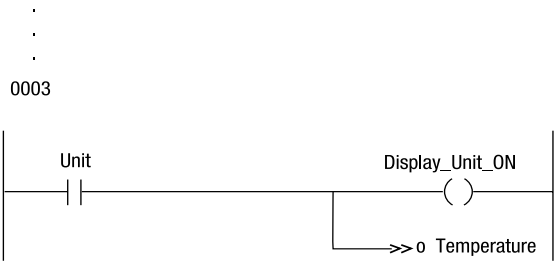
If the current result is 1, the output variable is reset, i.e. is reset to 0. The current result 0 has no effect on the value of the output variable. Resetting is represented by an output symbol with the letter R in the middle.



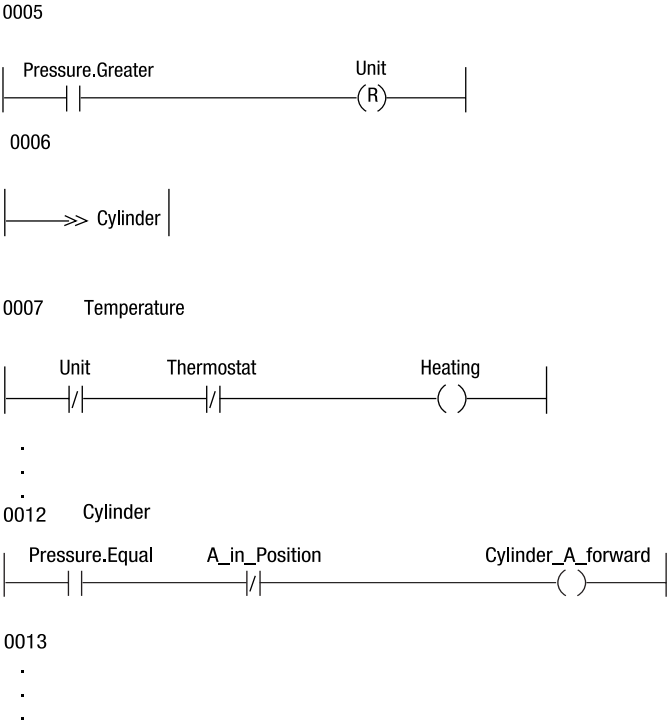
Just a momentary switching to 1 status on the System_OFF input will be enough to switch off the plant permanently.

Jumps

The use of jump commands is a way of structuring programs. They enable networks to be skipped which are only processed occasionally. Jumps can be executed depending on or irrespective of a particular Boolean current result. The jump destination is always the start of the network, i.e. the beginning of a program sequence. A jump label must be specified in addition to the jump symbol. The same jump label name must be entered in the destination network next to the network number.



07/01 AWB2700-1306-GB



Unconditional jump

The unconditional jump JMP is executed irrespective of the current result and is only useful in a program section that is processed occasionally. An unconditional jump creates an independent network.

Network 0006 of the program shown contains the unconditional jump to the label Cylinder. The jump destination is the network 0012, the jump label name Cylinder is shown in the header of this network.

Conditional jump

JMPC: The conditional jump JMPC is executed if the current result is 1. The current result 0 causes the program directly behind the jump operation to be processed.

JMPCN: The conditional jump JMPCN is executed if the current result is 0. If the current result is 1, the program is processed directly after the jump operation. Network 0003 of the program shown on page 102 executes a jump to network 0007 if the value of the variable Aggregate in network 0003 is 0.

Returns

Returns enable the program execution in the current POU to be interrupted and returned to the program level from which the POU was invoked. Returns can be executed depending on or irrespective of the current result.

Unconditional return

The unconditional return RET is executed irrespective of the current result. It is only useful in a program section that is only processed occasionally and which is not the last network. An unconditional return starts a separate network.

Conditional return

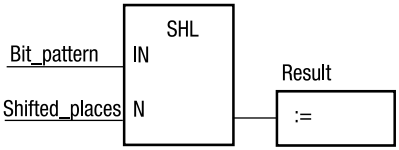
RETC: The conditional return RETC is executed if the current result is 1. If the current result is 0, the program will be processed directly after the return operation.

07/01 AWB2700-1306-GB

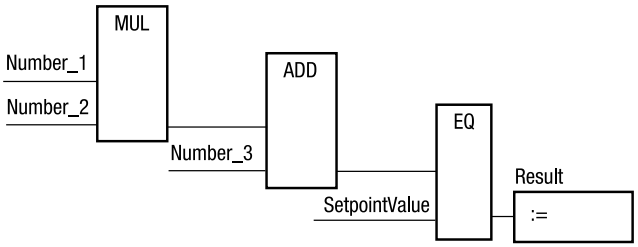
RETCN: The conditional return RETCN is executed if the current result is 0. If the current result is 1, the program will be continued directly after the return operation.

Invocation of functions

Functions in LD and in FBD are represented by a rectangular graphical symbol. The input operands are shown on the left-hand connection lines and the output variables the right. The name of the function and the names of the inputs are shown inside the block.



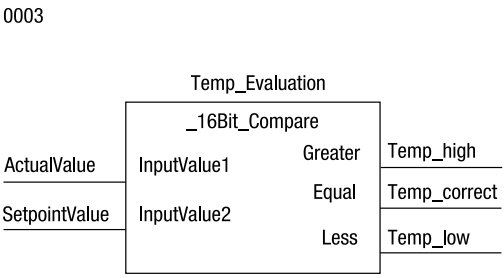
A network can consist of a series of cascaded functions.



Call of function blocks

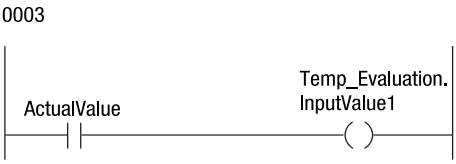
Function blocks in LD and FBD are shown in the same way. The function block is shown as a rectangular block The input operands are shown on the left-hand connection line and the output operands on the right. The name of the function block is contained inside the block, with the names of the inputs on the left and the names of the outputs on the right. The instance name of the function block is shown above the block.

A function block counts as an independent network.



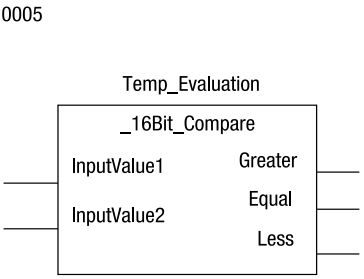
The unused input and output operands of the function block are not assigned a variable name and their corresponding connection lines are shown without any designation.

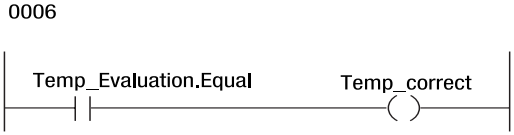
The parameters can be transferred outside of the function block. In this case the function block is shown without any external connection.



07/01 AWB2700-1306-GB









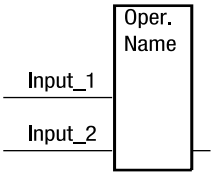
Only an unconditional function block invocation can be shown in the graphical programming languages. If the function block is to be invoked according to the current result, a conditional jump to the network must be programmed with the function block invocation.

Programming in function block diagram (FBD)

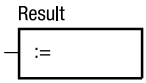
In FBD the individual language elements are represented as graphical symbols.

In FBD several functions can be combined within a network. A function block is represented in an individual network.

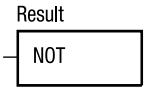
The following graphical symbols are available for displaying the program in FBD:



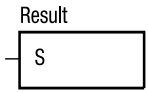
Graphical symbol to show an operation with two variables. It is used for logical, arithmetic and comparison functions. In the case of logical sequences the values of input variables can be negated before sequencing. This kind of input variable is marked with a negation symbol such as Input_2, for example.



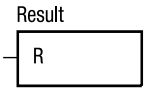
Assignment of a current result (CR) or a constant to a variable



Assign a negated current result or a negated constant to a variable



Setting of a Boolean variable if CR = 1



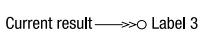
Resetting of a Boolean variable if CR = 1



Unconditional jump to a label



Jump to a label if current result = 1



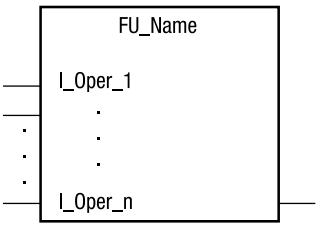
Jump to a label if current result = 0

Unconditional return to the calling POU

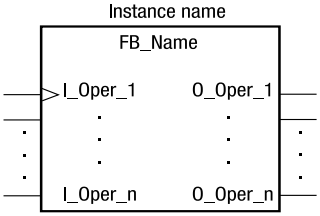
Return to the calling POU if CR = 1

07/01 AWB2700-1306-GB

Return to the calling POU if CR = 0



Graphical symbol which indicates a function. Functions have any number of input operands but only one output operand.



Graphical symbol which indicates a function block. Function blocks have any number of input and output operands.

Variable names of the inputs and outputs can be entered with a maximum length of 64 characters. Long variable names, however, will appear in up to 3 lines above the input or output symbol.

Logic (Boolean) functions

The logic sequences AND, OR and XOR are shown as graphical symbols with several input operands and one output operand.

The value of an input operand can also be negated. In this case the operand value is negated first of all, after which the sequence is formed. These sequences are also shown with the three basic graphical symbols for AND, OR and XOR. The operand to be negated is marked by a negation symbol on the connection line.

The result of a logic sequence of Boolean input variables is a condition for one of the following operations:

- a store operation
- a jump to a point in the program which is marked by a jump label
- a return to the calling POU

Logic sequences can refer to variables of the ANY_BIT data type.

AND sequence

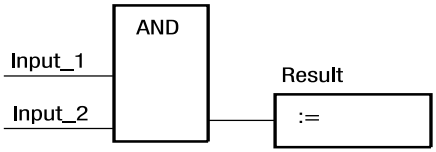
The input operands are ANDed. The current result is assigned to the output operand.

The individual bits of the first operand of an AND sequence of byte or word operands are sequenced with the corresponding bits of the second operand.

07/01 AWB2700-1306-GB

0001

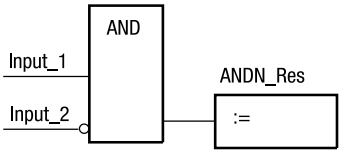
Sequencing binary variables acc. AND



The output variable Result is 1 if the two input variables are 1.

0001

ANDN-Sequence



The value of the Input_2 operand is negated in the ANDN function shown. The result is ANDed with the value of the operand Input_1 and assigned to the ANDN_Res output operand.

The individual bits of the second operand of an ANDN sequence of byte or word operands are negated and sequenced with the corresponding bits of the first operand.

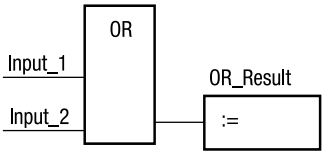
OR sequence

The input operands are ORed. The current result is assigned to the output operand.

The individual bits of the first operand of an OR sequence of byte or word operands are sequenced with the corresponding bits of the second operand.

0001

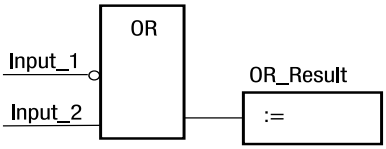
Sequencing Input_1 with Input_2
acc. OR



The output variable OR_Result is 1 if at least one of the input variables is 1.

0001

Negating Input_1, then sequencing
with Input_2 acc. OR



In the sequence shown the value of the operand Input_1 is negated. The result is ORed with the value of the operand Input_2 and assigned to the OR_Result output operand.

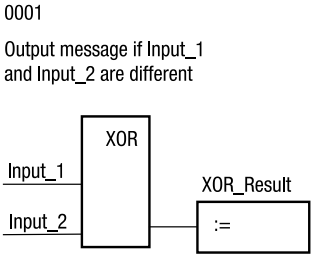
In a sequence of byte or word operands the individual bits of the operand Input_1 are negated and sequenced with the corresponding bits of the operand Input_2.

07/01 AWB2700-1306-GB

Exclusive OR sequence

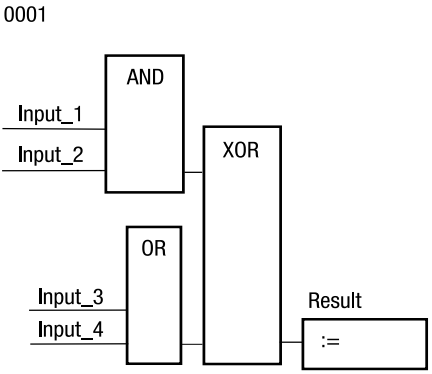
The input operands are XORed. The current result is assigned to the output operand.

In an XOR sequence of byte or word operands, the individual bits of the first operand are sequenced with the corresponding bits of the second operand.



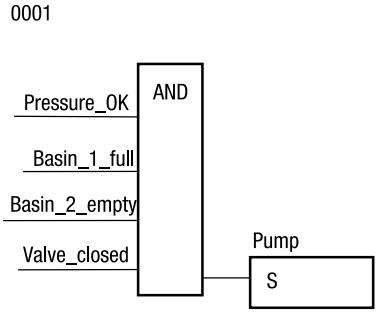
Mixed sequences

Mixed sequences consist of combinations of several different graphical symbols in the same network.

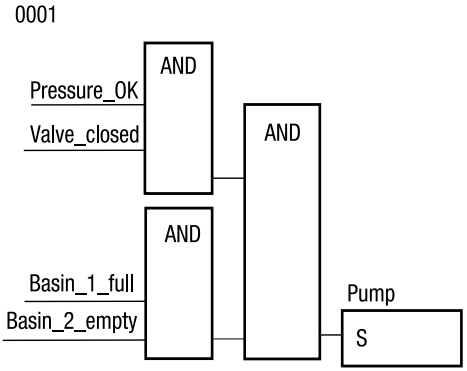


A sequence consisting of several input variables can be shown in different ways:

By increasing the number of connection lines on the existing sequence symbol.



By adding a sequence symbol of the same type upstream or downstream.

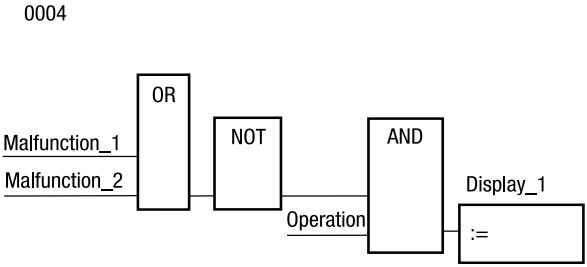


Negating intermediate result

When several functions are combined in a network, an intermediate result can be continued after a negation with a NOT function in the same network.

The negation can be used on ANY_BIT data elements of type.

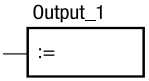
07/01 AWB2700-1306-GB



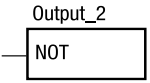
The variable Display_1 indicates trouble-free operation.

Assignment

The current result or a constant can be assigned directly to an output variable or after it has been negated. There are two output symbols for both options:



Direct assignment of a current result or a constant to the output variable.



Assignment of a negated current result or a negated constant to the output variable.

The negation of byte, word or double word results corresponds to the negation of the individual bits.

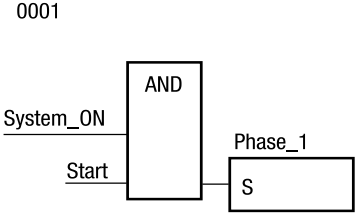
Only binary variables can be negated.

Retentive assignment

The Boolean current result can be assigned retentively to the output variable, which can also be set or reset retentively.

Set

If the current result is 1, the output variable is set, i.e. is set to 1. The 0 status of the current result has no effect on the value of the output variable. Setting is represented by an output symbol with an S in the middle.

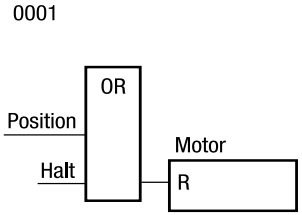


If System_ON is 1 just a momentary switching to 1 status on the Start input will be enough to switch on the first phase of the plant permanently.

07/01 AWB2700-1306-GB

Reset

If the current is 1, the output variable is reset, i.e. is reset to 0. The current result 0 has no effect on the value of the output variable. Resetting is represented by an output symbol with an R in the middle.

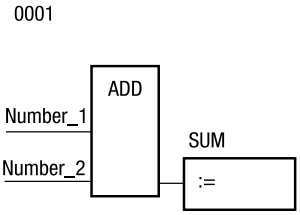


Just a momentary switching to 1 status on the Position input or the Halt input is enough to switch off the motor permanently.

Arithmetic operations

The arithmetic operations addition, subtraction, multiplication and division can be used on numeric data types.

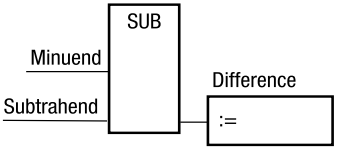
Addition



The values of variables Number_1 and Number_2 are added and the result is assigned to the variable Sum.

Subtraction

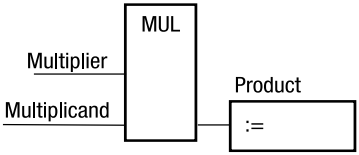
0001



The value of the variable Subtrahend is subtracted from the variable Minuend and the result is assigned to the variable Difference.

Multiplication

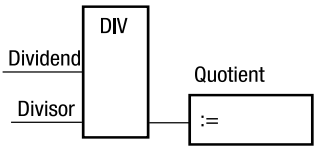
0001



The value of the variable Multiplier is multiplied with the value of the variable Multiplicand and the result is assigned to the variable Product.

Division

0001



The value of the variable Dividend is divided by the value of the variable Divisor and the result is assigned to the variable Quotient.

07/01 AWB2700-1306-GB

Compare operations

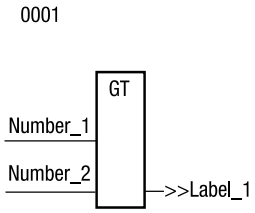
A compare operation involves the comparison between the value of the first operand and the value of the second operand. The completion of the comparison is indicated by the status 1 on the output operand.

All elementary data types can be used.

The graphical symbols are marked with the appropriate operation name:

GT	greater than
GE	greater than or equal to
EQ	equal
NE	not equal to
LE	less than or equal to
LT	less than

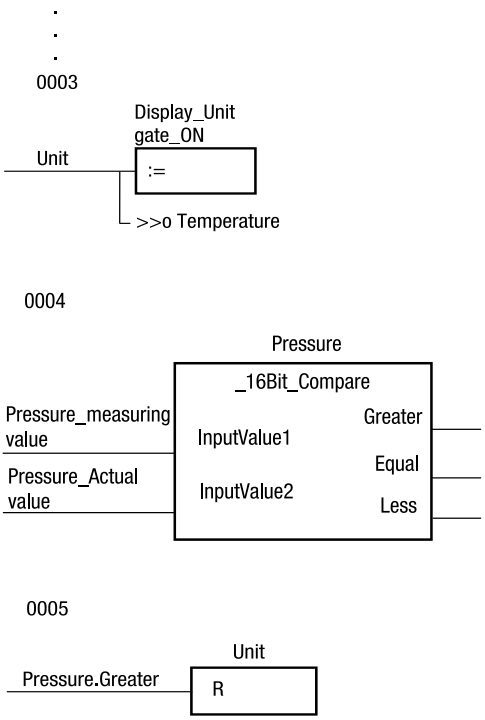
Example: Greater than comparison



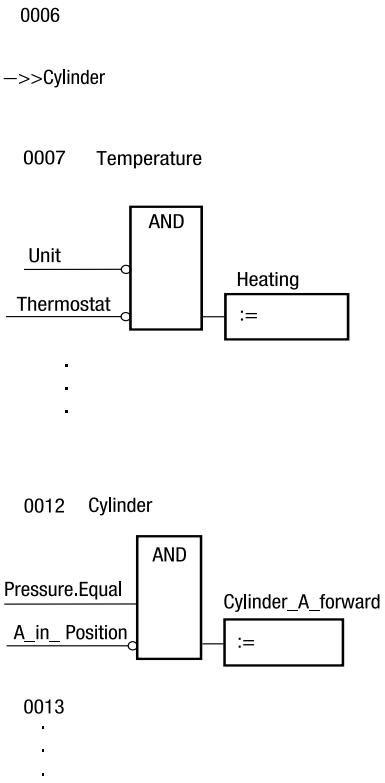
If the value of the variable Number_1 is greater than the value of the variable Number_2, the jump condition is fulfilled. Program execution is continued in the network marked Label_1.

Jumps

The use of jump commands is a way of structuring programs. They enable networks to be skipped which are only processed occasionally. Jumps can be executed depending on or irrespective of a particular Boolean current result. The jump destination is always the start of the network, i.e. the beginning of a program sequence. A jump label must be specified in addition to the jump symbol. The same jump label name must be entered in the destination network next to the network number.



07/01 AWB2700-1306-GB



Unconditional jump

The unconditional jump JMP is executed irrespective of the current result and is only useful in a program section that is processed occasionally. An unconditional jump creates an independent network.

Network 0006 of the program shown contains the unconditional jump to the label Cylinder. The jump destination is the network 0012, the jump label name Cylinder is shown in the header of this network.

Conditional jump

JMPC: The conditional jump JMPC is executed if the current result is 1. If the current result is 0, the program will be processed directly after the jump operation.

JMPCN: The conditional jump JMPCN is executed if the current result is 0. If the current result is 1, the program is processed directly after the jump operation.

Network 0003 of the program shown on page 120 executes a jump to network 0007 if the value of the current result in network 0003 is 0.

Returns

Returns enable the program execution in the current POU to be interrupted and returned to the program level from which the POU was invoked. Returns can be executed depending on or irrespective of the current result.

Unconditional return

The unconditional return RET is executed irrespective of the current result. It is only useful in a program section that is only processed occasionally and which is not the last network. An unconditional return starts a separate network.

07/01 AWB2700-1306-GB

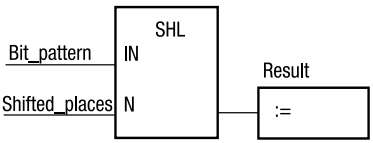
Conditional return

RETC: The conditional return RETC is executed if the current result is 1. If the current result is 0, the program will be processed directly after the return operation.

RETCN: The conditional return RETCN is executed if the current result is 0. If the current result is 1, the program will be processed directly after the return operation.

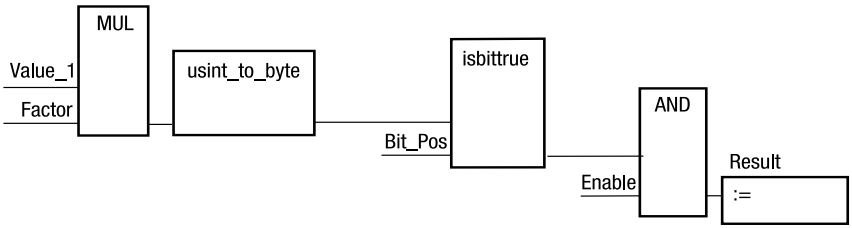
Invocation of functions

Functions in LD and in FBD are represented by a rectangular graphical symbol. The input operands are shown on the left-hand connection lines and the output variables the right. The name of the function and the names of the inputs are shown inside the block.



Several functions and operation symbols can be cascaded within a network. If the data types of the connected graphical symbols do not match, a type conversion function must be inserted between the data types concerned.

0001



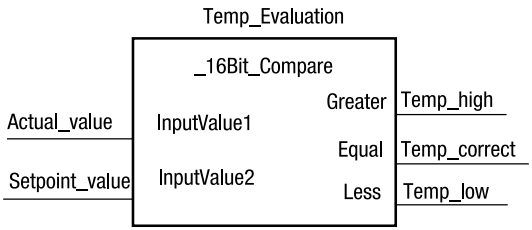
The numerical result of the multiplication is converted into the data type BYTE.

Invocation of function blocks

Function blocks in LD and FBD are shown in the same way. The function block is shown as a rectangular block The input operands are shown on the left-hand connection line and the output operands on the right. The name of the function block is contained inside the block, with the names of the inputs on the left and the names of the outputs on the right. The instance name of the function block is shown above the block.

A function block counts as an independent network.

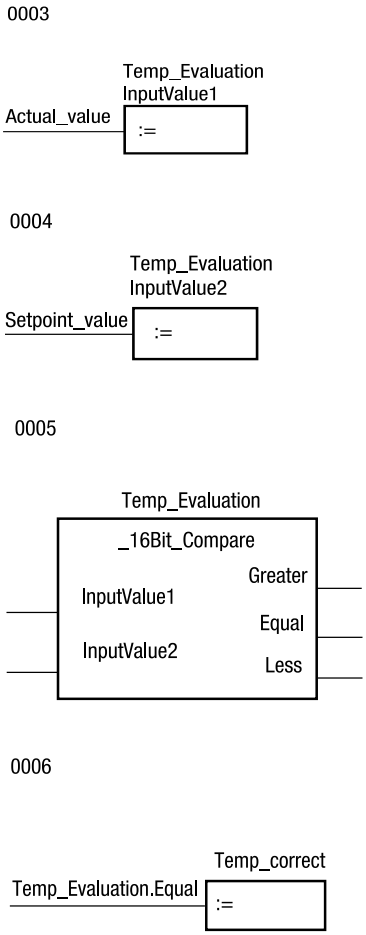
0003



07/01 AWB2700-1306-GB

The unused input and output operands of the function block are not assigned a variable name and their corresponding connection lines are shown without any designation.

The parameters can be transferred outside of the function block. In this case the function block is shown without any external connection.





Only an unconditional function block invocation can be shown in the graphical programming languages. If the function block is to be invoked according to the current result, a conditional jump to the network must be programmed with the function block invocation.

07/01 AWB2700-1306-GB

5 Functions

Overview

Sucosoft provides a wide variety of functions for your use. They can be classified into:

IEC Functions in compliance with the IEC/EN 61131-3 standard,

Moeller standard functions.

The functions in compliance with the IEC/EN 61131-3 standard are designed to cover all vendors. PLC programs which use only IEC functions can therefore be used with controllers made by different manufacturers.

Moeller has added further functions to the IEC functions; these are marked. With a few exceptions the these functions can be used with all Moeller controllers.

The following table shows an overview of all functions grouped according to the jobs they do. Each function is marked with 'IEC' or (in the table only marked with M) and the controller type specified where the function can be used without any modification.

Some of the functions described use the data type ADDRESS. They are used for the automatic conversion of programs from previous systems and for this reason cannot be used with all controllers. Such functions are marked # in the following table.

A detailed description of the functions (in alphabetical order) starts on Page 134.

Function	Meaning	PS4-200	PS4-300	PS416	Standard	See page
Conversion functions						
*_TO_**	Type conversion of elementary data types	✓	✓	✓	IEC, M	134
	Type conversion of time and date	–	✓	✓		146
TRUNC	INT part of a REAL number	–	✓	✓	IEC	220
Numerical functions						
ABS	Formation of amount	✓	✓	✓	IEC	154
NEG	Two's complement	✓	✓	✓	M	200
Arithmetic functions						
ADD	Addition of numeric data types	✓	✓	✓	IEC	155
	Addition of duration to time and date	✓	✓	✓	IEC	157
	Addition of addresses	✓, #	✓, #	–	M	159
MUL	Multiplication of numeric data types	✓	✓	✓	IEC	194
	Multiplication of duration	–	✓	✓	IEC	196
	Multiplication of addresses	✓, #	✓, #	–	M	197
SUB	Subtraction of numeric data types	✓	✓	✓	IEC	216
	Subtraction of duration and time and date	✓	✓	✓	IEC	217
	Subtraction of addresses	✓, #	✓, #	–	M	219
DIV	Division of numeric data types	✓	✓	✓	IEC	168
	Division of duration	–	✓	✓	IEC	169
	Division of addresses	✓, #	✓, #	–	M	170
MOD	Remainder Division	–	✓	✓	IEC	193
Edge detection						
IsCarry	Test of carry bit	✓	✓	✓	M	180
IsZero	Test of zero bit	✓	✓	✓	M	183
IsOverflow	Test of overflow bit	✓	✓	✓	M	181
IsPositive	Test whether result is positive	✓	✓	✓	M	182
IsBitTrue	Test of a bit in the result	✓	✓	✓	M	179
GetCarry Register	Get carry with multiplication or remainder of a division	✓	✓	✓	M	176
Shift functions						
SHL	Shift left	✓	✓	✓	IEC	214
SHR	Shift right	✓	✓	✓	IEC	215

07/01 AWB2700-1306-GB

Function	Meaning	PS4-200	PS4-300	PS416	Standard	See page
ShiftLeft Carry	Shift left with carry	✓	✓	✓	M	212
ShiftRight Carry	Shift right with carry	✓	✓	✓	M	213
ROL	Rotate left	✓	✓	✓	IEC	208
ROR	Rotate right	✓	✓	✓	IEC	209
Logic sequences						
AND, &	Logic AND function *)	✓	✓	✓	IEC	49
OR	Logic OR function *)	✓	✓	✓	IEC	52
XOR	Logic Exclusive OR function *)	✓	✓	✓	IEC	54
NOT	Bit complement formation	✓	✓	✓	IEC	201
Comparison functions						
GT	Comparison function: greater than *)	✓	✓	✓	IEC	59
GE	Comparison function: greater than or equal to *)	✓	✓	✓	IEC	59
EQ	Comparison function: equal to *)	✓	✓	✓	IEC	59
LE	Comparison function: less than or equal to *)	✓	✓	✓	IEC	59
LT	Comparison function: less than *)	✓	✓	✓	IEC	59
NE	Comparison function: not equal to *)	✓	✓	✓	IEC	59
OS system functions (see also OS system functions in table on Page 225)						
PLC_Message	System diagnostics, reading message word	✓	✓	✓	M	202
PLC_ReadClock	Reading single element from real-time clock	✓	✓	✓	M	204
Interrupt functions						
Disable_Interrupt	Disable alarm	✓	✓	✓	M	165
Enable_Interrupt	Enable alarm	✓	✓	✓	M	171
Character string functions						
LEN	Determine length of a character string	✓	✓	✓	IEC	185
CONCAT	Concatenation of strings Concatenation of time and date	–	✓	✓	IEC	160 162

Function	Meaning	PS4-200	PS4-300	PS416	Standard	See page
DELETE	Delete n characters from a string	–	✓	✓	IEC	163
FIND	Find a string within a string	–	✓	✓	IEC	174
INSERT	Insert a string in a string	–	✓	✓	IEC	177
LEFT	Select n characters left of a string	–	✓	✓	IEC	184
MID	Sections of a string	–	✓	✓	IEC	190
REPLACE	Replace characters in a string	–	✓	✓	IEC	205
RIGHT	Select n characters right of string	–	✓	✓	IEC	207
Selection functions						
LIMIT	Limitation	–	✓	✓	IEC	186
MAX	Maximum selection	–	✓	✓	IEC	188
MIN	Minimum selection	–	✓	✓	IEC	192
MUX	Multiplexer	–	✓	✓	IEC	198
SEL	Conditional operand selection	–	✓	✓	IEC	210

*) The functions that correspond to the IL operators are described in the section Instruction set starting on Page 41

This chapter describes the individual functions and their application by means of simple examples.

The interfaces for parameter transfer are described by means of the prototypes of the individual functions. A prototype gives an overview of the input operands of a function. In IL representation the prototype is the declaration section of the function with the declaration blocks of the input operands.

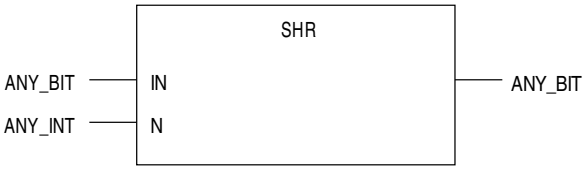
The functions are illustrated with a graphical representation of their prototypes. This consists of a square graphical symbol with input operands on the left and the output operands on the right. The name of the function is shown in the middle of the graphical symbol. The names of the input operands appear on the left-hand side. The data types of the operands

07/01 AWB2700-1306-GB

are shown outside the graphical symbol.

Example: prototype representation
Comparison between instruction list and graphical representations of a prototype function using the SHR function as an example.

```
FUNCTION SHR : ANY_BIT
VAR_INPUT
  IN : ANY_BIT;
  N : ANY_INT;
END_VAR
END_FUNCTION
```



For the IN operand of the SHR function the data types BYTE, WORD and DWORD (DWORD only for PS4-300/PS416) can be used for ANY_BIT. This function is an overloaded function, i.e. the function can be used for various data types.



Overloaded – non-type-dependent – functions are only available with standard (manufacturer) functions. Programming your own overloaded functions (user-specified functions) is not possible.

You can find the description of the individual elementary data types in the Section “Elementary data types” (Page 11) of this manual.

The invocation of a function is carried out by entering its name and transferring the parameter list for the individual input operands of the function. The parameters are listed beside the function name, separated by a comma.



In IL the first function parameter is provided in the working register before the function is called.

Type conversion functions

*_TO_**	Function name
*	Data type of the input operand
**	Data type of the output operand

Operands that are sequenced with each other must always be of the same data type. If, for example, the data type BYTE is required for an input operand of a function but the data type of the variable to be transferred is of type USINT, the type must be converted using the type converting function USINT_TO_BYTE.

In type conversion the value in the working register is used as the input operand. The result - the output operand in the new data type - is made available in the working register.



When type converting numerical data types make sure that the permitted value of the target data type is not exceeded. If the value of the input operand is too large, the result will be reduced to the available bit length of the output operand and will thus supply an incorrect value.

Sucosoft provides several type converting functions for type conversion between the elementary data types. The type conversion functions can be used for

07/01 AWB2700-1306-GB

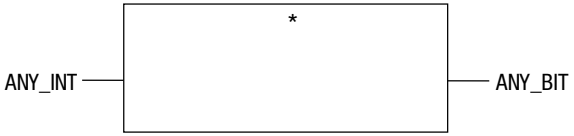
the following combinations:

Input \ Output	SINT	INT	DINT[1]	USINT	UINT	UDINT[1]	REAL[1]	BOOL	BYTE	WORD	DWORD[1]	TIME_OF_DAY	DATE	STRING[1]	BCD
SINT		x	x	x	x	x	x	x	x	x	x				x
INT	x		x	x	x	x	x	x	x	x	x				x
DINT[1]	x	x		x	x	x	x	x	x	x	x				x
USINT	x	x	x		x	x	x	x	x	x	x				x
UINT	x	x	x	x		x	x	x	x	x	x			x	x
UDINT[1]	x	x	x	x	x		x	x	x	x	x				x
REAL[1]	x	x	x	x	x	x									
BOOL	x	x	x	x	x	x			x	x	x				
BYTE	x	x	x	x	x	x		x		x	x				
WORD	x	x	x	x	x	x		x	x		x				
DWORD[1]	x	x	x	x	x	x		x	x	x					
DATE_AND_TIME												x	x		
TIME				x	x										
BCD	x	x	x	x	x	x									
STRING[1]					x										
ADDRESS[2]					x										

[1] only for PS4-300 and PS416
[2] only for PS4-200 and PS4-300

IEC

Type Conversion 'ANY_INT' to 'ANY_BIT'



Function prototype

Description

In a real case the character * would be replaced by the function name. The following type conversions are possible:

Function name	Comment
USINT_TO_BYTE UINT_TO_WORD UDINT_TO_DWORD[1)] INT_TO_WORD DINT_TO_DWORD[1)]	Type conversions without data type expansion
USINT_TO_WORD USINT_TO_DWORD[1)] UINT_TO_DWORD[1)]	Type conversion with data type expansion from a less significant data type to a more significant data type without evaluation of the most significant bit as sign bit.
SINT_TO_WORD SINT_TO_DWORD[1)] INT_TO_DWORD[1)]	Type conversion with data type expansion from a less significant data type to a more significant data type with evaluation of the more significant bit as sign bit.
UINT_TO_BYTE UDINT_TO_BYTE[1)] UDINT_TO_WORD[1)] SINT_TO_BYTE DINT_TO_BYTE[1)] DINT_TO_WORD[1)]	Type conversion from a more significant data type to a less significant data type. The final result is reduced to the length of the destination data type; the more significant bits are lost.

07/01 AWB2700-1306-GB

Function name	Comment
SINT_TO_BOOL UINT_TO_BOOL UDINT_TO_BOOL[1)] USINT_TO_BOOL INT_TO_BOOL DINT_TO_BOOL[1)]	The result has the value 0 with the value 0 of the input operand, otherwise the result is 1.

[1)] only for PS4-300 and PS416

Example: type conversion USINT_TO_WORD

The data element of type USINT (unsigned integer) is written to the low byte of the data element of type WORD the high byte is filled with zeros.

			High Low
Conversion from	00100101 USINT	to	00000000 00100101 WORD

			High Low
Conversion from	10100101 USINT	to	00000000 10100101 WORD

Example: type conversion SINT_TO_WORD
The bit pattern in the data element of type SINT is interpreted as a signed number. The most significant bit (bit 7) is evaluated as a sign bit. The bit pattern is written to the low byte of the data element of type INT. If the signed bit has a 0, it is a positive number. The high byte will therefore be filled with zeros. If bit 7 is 1, the high byte will be filled with ones.

			High Low
Conversion from	00100101 SINT	to	00000000 00100101 WORD
			High Low
Conversion from	10100101 SINT	to	11111111 10100101 WORD

Example: invocation of the function SINT_TO_BYTE

```
PROGRAM conv1

VAR
  Number1 : SINT := -1;
  Byte1 : BYTE;
END_VAR

  LD   Number1
  SINT_TO_BYTE
  ST   Byte1 (* Byte1: 11111111 *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC

Type Conversion of ANY_INT to ANY_INT



Function prototype

Description

In a real case the character * would be replaced by the function name. The following type conversions are possible:

Function name	Comment
USINT_TO_SINT SINT_TO_USINT UINT_TO_INT INT_TO_UINT UDINT_TO_DINT[1)] DINT_TO_UDINT[1)]	Type conversions without data type expansion
USINT_TO_UINT USINT_TO_INT USINT_TO_UDINT[1)] USINT_TO_DINT[1)] UINT_TO_UDINT[1)] UINT_TO_DINT[1)]	Type conversion with data type expansion from a less significant data type to a more significant data type without evaluation of the most significant bit as sign bit. The more significant bits are filled with zeros.
SINT_TO_UINT SINT_TO_INT SINT_TO_UDINT[1)] SINT_TO_DINT[1)] INT_TO_UDINT[1)] INT_TO_DINT[1)]	Type conversion with data type expansion from a less significant data type to a more significant data type with evaluation of the more significant bit as sign bit. The more significant bits are filled with the sign bit.

Function name	Comment
UINT_TO_USINT UINT_TO_SINT INT_TO_USINT INT_TO_SINT UDINT_TO_USINT[1)] UDINT_TO_SINT[1)] UDINT_TO_UINT UDINT_TO_INT[1)] DINT_TO_USINT[1)] DINT_TO_SINT[1)] DINT_TO_UINT[1)] DINT_TO_INT[1)]	Type conversion from a more significant data type to a less significant data type. The result is reduced to the length of the destination data type.

[1)] only for PS4-300 and PS416

07/01 AWB2700-1306-GB

IEC

Type Conversion ANY_BIT to ANY_INT



Function prototype

Description

In a real case the character * would be replaced by the function name. The following type conversions are possible:

Function name	Comment
BOOL_TO_USINT BOOL_TO_UINT BOOL_TO_UDINT[1)] BOOL_TO_SINT BOOL_TO_INT BOOL_TO_DINT[1)]	Type conversion with data expansion. The result has the value of the input operand (0 or 1).
BYTE_TO_USINT WORD_TO_UINT DWORD_TO_UDINT[1)] BYTE_TO_SINT WORD_TO_INT DWORD_TO_DINT[1)]	Type conversions without data type expansion
BYTE_TO_UINT BYTE_TO_UDINT[1)] WORD_TO_UDINT[1)]	Type conversion with data type expansion from a less significant data type to a more significant data type without evaluation of the most significant bit as sign bit. The more significant bits are filled with zeros.
BYTE_TO_INT BYTE_TO_DINT[1)] WORD_TO_DINT[1)]	Type conversion with data type expansion from a less significant data type to a more significant data type without evaluation of the most significant bit as sign bit. The more significant bits are filled with zeros.

Function name	Comment
WORD_TO_USINT DWORD_TO_USINT[1)] DWORD_TO_UINT[1)] WORD_TO_SINT DWORD_TO_SINT[1)] DWORD_TO_INT[1)]	Type conversion from a more significant data type to a less significant data type. The final result is reduced to the length of the destination data type; the more significant bits are lost.

[1)] only for PS4-300 and PS416

Example: type conversion BYTE_TO_UINT
The most significant bit of the data element of type BYTE is not evaluated. The bit pattern is written to the low byte of the data element of type UINT; the high byte is filled with zeros.

			High Low
Conversion from	00100101 BYTE	to	00000000 00100101 UINT
Conversion from	10100101 BYTE	to	00000000 10100101 UINT

Example: type conversion BYTE_TO_INT
The bit pattern is written to the low byte of the data element of type INT. The high byte is filled with zeros.

			High Low
Conversion from	00100101 BYTE	to	00000000 00100101 INT
Conversion from	10100101 BYTE	to	00000000 10100101 INT

07/01 AWB2700-1306-GB

Example: invocation of the function BYTE_TO_SINT

```
PROGRAM conv2

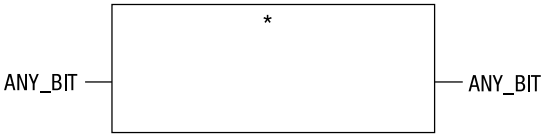
VAR
  Byte1 : BYTE := 2#0111;
  Number1 : SINT;
END_VAR

  LD   Byte1
  BYTE_TO_SINT
  ST Number1          (* Number1: 123 *)

END_PROGRAM
```

IEC

Type Conversion ANY_BIT to ANY_BIT



Function prototype

Description

In a real case the character * would be replaced by the function name. The following type conversions are possible:

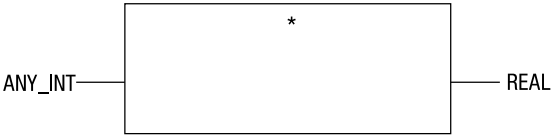
Function name	Comment
BYTE_TO_BOOL WORD_TO_BOOL DWORD_TO_BOOL[1)]	The result has the value 0 with the value 0 of the input operand, otherwise the result is 1.
BYTE_TO_WORD BYTE_TO_DWORD[1)] WORD_TO_DWORD[1)]	Type conversion with data type expansion from a less significant data type to a more significant data type. The more significant bits are filled with zeros.
WORD_TO_BYTE DWORD_TO_BYTE[1)] DWORD_TO_WORD[1)]	Type conversion from a more significant data type to a less significant data type. The result is reduced to the length of the destination data type.
BOOL_TO_BYTE BOOL_TO_WORD BOOL_TO_DWORD[1)]	Type conversion with data type expansion. The result has the value of the input operand (0 or 1).

1) only for PS4-300 and PS416

07/01 AWB2700-1306-GB

IEC **PS4-300**
PS416

Type conversion ANY_INT to REAL



Function prototype

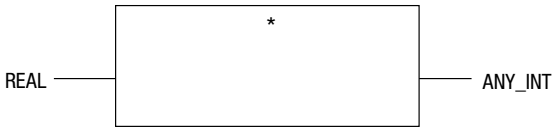
Description

In a real case the character * would be replaced by the function name. The following type conversions are possible:

SINT_TO_REAL
INT_TO_REAL
DINT_TO_REAL
USINT_TO_REAL
UINT_TO_REAL
UDINT_TO_REAL

IEC **PS4-300**
PS416

Type conversion REAL to ANY_INT



Function prototype

Description
In a real case the character * would be replaced by the function name. The following type conversions are possible:

REAL_TO_SINT
REAL_TO_INT
REAL_TO_DINT
REAL_TO_USINT
REAL_TO_UINT
REAL_TO_UDINT

In the case of floating point numbers (data type REAL) with decimal values less than 0.5 the amounts of the output values are rounded down, with decimal values greater than 0.5 they are rounded up.

Numbers with decimal values equal to 0.5 are rounded to the nearest even number.

07/01 AWB2700-1306-GB

Example: type conversion REAL_TO_INT

Input value :	Output value:
1.4	1
–1.4	–1
1.6	2
–1.6	–2
1.5	2
2.5	2
0.5	0
3.5	4

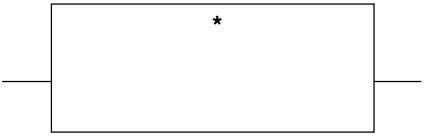
Conversion results which exceed the range of values of the destination data type are truncated - i.e. reduced to the bit length available.

Example: type conversion REAL_TO_SINT

Input value :	Output value:
2049.0	1
2178.0	–126

IEC

Type Conversion of Time and Date



Function prototype

Description

In a real case the character * would be replaced by the function name. The following type conversions are possible:

Function name	Comment
DATE_AND_TIME_TO_TIME_OF_DAY * or DT_TO_TOD	The time is loaded into the working register from the data element with the date and time data.
DATE_AND_TIME_TO_DATE * or DT_TO_DATE	The date is loaded into the working register from the data element with the date and time data.
TIME_TO_UINT	The duration is converted into milliseconds and loaded into the working register. The result must not exceed the value 65535.
TIME_TO_USINT	The duration is converted into milliseconds and loaded into the working register. The result must not exceed the value 255.

* Alternative notation

For the two functions TIME_TO_UINT and TIME_TO_USINT, results with values larger than 65535 and 255 are invalid. This can be scanned with the IsOverflow function.

07/01 AWB2700-1306-GB

Example: invocation of the function DATE_AND_TIME_TO_TIME_OF_DAY

```
PROGRAM time

VAR
  date1 : DATE_AND_TIME := DT#1980-01-12-
                           18:15:00.00;
  only_time : TIME_OF_DAY;
END_VAR

  LD   date1
  DATE_AND_TIME_TO_TIME_OF_DAY
  ST   only_time (*only_time: 18:15:00.00 *)

END_PROGRAM
```

Example: invocation of the function DATE_AND_TIME_TO_DATE

```
PROGRAM date

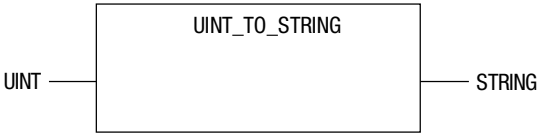
VAR
  Date1 : DATE_AND_TIME := DT#1980-01-12-
                           18;:15:00.00;
  only_date: DATE;
END_VAR

  LD   date1
  DATE_AND_TIME_TO_DATE
  ST   only_date            (*only_date: 1980-01-12 *)

END_PROGRAM
```

PS4-300
PS416

Type conversion UINT to STRING



Function prototype

Description

The two bytes of the input operand of data type UINT are each interpreted as an encoded text character and output in a two character string. This type conversion is not useful unless the value of the input variable actually corresponds to two encoded characters.

Example: invocation of the function UNIT_TO_STRING

```
PROGRAM main

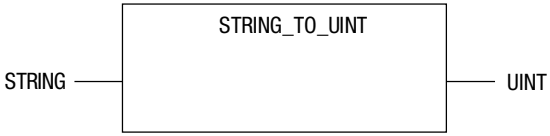
VAR
  str1 : STRING(5);
END_VAR
       (* The constant 16#4F4B is shown as
       the string 'OK' *)

  LD   16#4F4B
  UNIT_TO_STRING
  ST   str1
  LD   str1

END_PROGRAM
```


07/01 AWB2700-1306-GB

Type conversion STRING to UINT



Function prototype

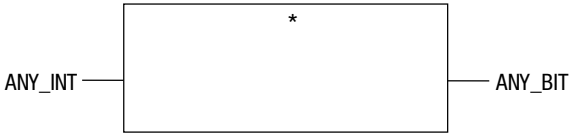
Description

The two-character string specified in the input operand of the data type STRING is converted into the data type UINT. The left-hand character of the string is supplied in the high byte and the right-hand character in the low byte. The bit pattern of the input and output operands remains unchanged.

If the character string consists of more than two characters only the two left-hand characters are evaluated. A single character is supplied in the low byte.

IEC

Type conversion of ANY_INT to BCD



Function prototype

Description

In a real case the character * would be replaced by the function name.

The input value is first converted into a BCD value which is then output as a bit pattern data type (ANY_BIT). The length of the output variable need not be the same as the size of the BCD value. The most significant bits will be lost if the bit pattern data type assigned for the representation of the BCD result is too small. The following type conversions are possible:

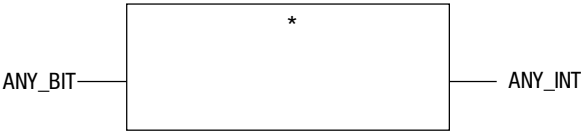
Function name	Comment
USINT_TO_BCD UINT_TO_BCD UDINT_TO_BCD[1)]	The result contains the BCD coded input value.
SINT_TO_BCD INT_TO_BCD DINT_TO_BCD[1)]	The result contains the BCD coded input value. A negative input number is converted beforehand to a positive number; the sign cannot be represented.

[1)] only for PS4-300 and PS416

07/01 AWB2700-1306-GB

IEC

Type conversion of BCD to ANY_INT



Function prototype

Description

In a real case the character * would be replaced by the function name.

The input value coded in a bit pattern of data type BCD is converted to a binary-coded decimal value.

The input variable can be assigned any bit pattern data type, irrespective the length of the output variable.

The following type conversions are possible:

Function name	Comment
BCD_TO_USINT	The positive BCD value of max. 255 (decimal) is converted to a decimal value. There is no overflow with higher input values.
BCD_TO_UINT	The positive BCD value of max. 65535 (decimal) is converted to a decimal value. There is no overflow with higher input values.
BCD_TO_UDINT[1)]	The positive BCD value of max. 99999 (decimal) is converted to a decimal value.
BCD_TO_SINT	The positive BCD value of max. 255 (decimal) is converted to a decimal value. There is no overflow with higher input values.
BCD_TO_INT	The positive BCD value of max. 65535 (decimal) is converted to a decimal value. There is no overflow with higher input values.
BCD_TO_DINT[1)]	The positive BCD value of max. 99999 (decimal) is converted to a decimal value.

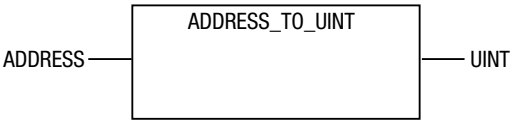
1) only for PS4-300 and PS416

Example: type conversion BCD_TO_USINT
The BCD-coded value 137 (decimal) is converted to a decimal value and output as data type USINT.

BCD value 137 (decimal)	USINT
0000 0001 0011 0111	1000 1001

07/01 AWB2700-1306-GB

Type Conversion ADDRESS to UINT



Function prototype

Description

The data type of a variable or a function block parameter is converted from ADDRESS to UINT.

With this, addresses of block copy functions, for example, can be saved for further evaluation in variables of type UINT since variables of type ADDRESS cannot be declared.

Example: invocation of the function ADDRESS_TO_UINT

```
PROGRAM addr_conv

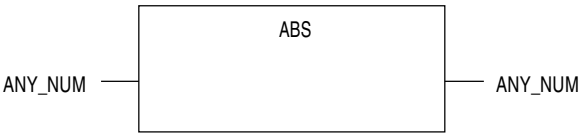
VAR
  Start_data_field AT %MB0.0.0.0 : BYTE ;
  Address1 : UINT ;
  Copy : BlockTransfer ;
END_VAR

  LD &Start_data_field
  ST Copy.SourceAddress
  ADDRESS_TO_UINT
  ST Address1

END_PROGRAM
```

IEC

ABS
Summation



Function prototype

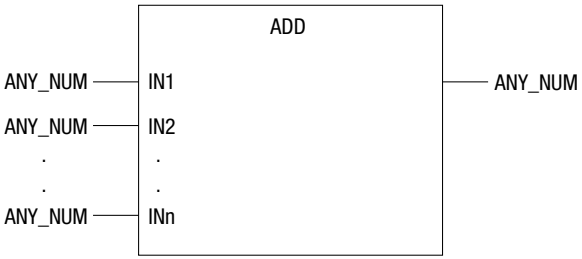
Description
The function ABS converts the numeric variable to a summation number; the data type remains unchanged.

The smallest SINT value (–128) and the smallest INT value (–32768) are not modified by the ABS function since the binary values for –128/+128 and –32768/+32768 are identical. If such values occur in the ABS function, an error message will not be generated.

07/01 AWB2700-1306-GB

IEC

ADD
Multiple Addition of Numeric Data Types



Function prototype

Meaning of operands

IN1 The first addend must be provided via the working register.

IN2 Further addends which are included in the addition.

.
.

INn

Description

The value in the working register is added to the values of the operands IN2 to INn. All input operands are of the same numeric data type.The result is made available in the working register for further processing. If the result exceeds the range of values permitted for the data type used an overflow will be formed.

The data type of the result corresponds to the selected data type of the input operand.

For this function all of the input operands must be of the same data type.

Example: addition of numeric data types

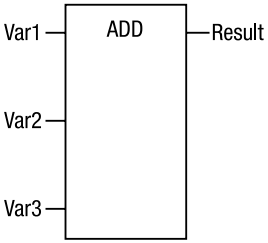
```
PROGRAM add3fold

VAR
  A : INT := 7;
  B : INT := 2;
  Sum : INT;
END_VAR

  LD   3
  ADD  A
  ADD  B
  ST   Sum                    (* Sum: 12 *)
END_PROGRAM
```

Example: invocation of the function ADD in FBD

The function ADD can be extended in the graphic representation in LD and FBD - more than two input operands can be processed.



Example: invocation of the function ADD in IL

In IL multiple input operands are handled by the sequential processing of two operands each time. The following IL listing shows how this is done. The second listing generates an error message with S40 versions before V 5.0.

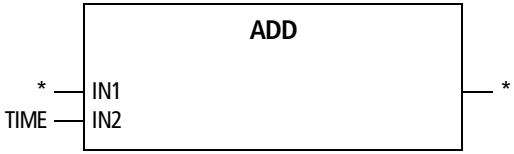
```
Listing 1: LD Var1
           ADD Var2
           ADD Var3
           ST Result

Listing 2: LD Var 1
           ADD Var2, Var3
           ST Result
```


07/01 AWB2700-1306-GB

IEC

ADD
Addition of Duration to Time and Date



Function prototype

The following data type combinations of the operands are possible:

Input operand IN1	Input operand IN2	Output operand
TIME	TIME	TIME
TIME_OF_DAY	TIME	TIME_OF_DAY
DATE_AND_TIME[1)]	TIME	DATE_AND_TIME

[1)] only for PS4-300 and PS416

Description
The input variable is increased by a time value and the result is loaded into the working register.

Only for PS-200:
An ADD operation with the TIME data type only yields correct results in the positive value range. Do not run this operation with negative values.

Example: addition of time values

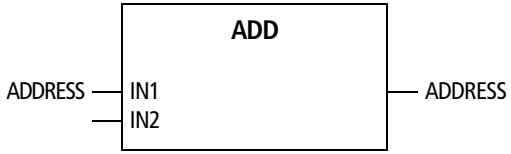
```
PROGRAM time_add

VAR
  Time1 : TIME_OF_DAY := TOD#15:25:00.00;
  Time_value : TIME := T#10h6m15s;
  New_time : TIME_OF_DAY;
END_VAR

  LD Time1
  ADD Time_value
  ST   New_time
       (* New_time: TOD#01:31:15.00 *)
END_PROGRAM
```


07/01 AWB2700-1306-GB

ADD
Addition of addresses



Function prototype

The following data type combinations of the operands are possible:

Input operand IN1	Input operand IN2	Output operand
ADDRESS	ANY_INT	ADDRESS
ADDRESS	ADDRESS	ADDRESS

Description

The input variable is increased and the result is loaded into the working register.

Loops or conditions for function blocks can be calculated with ADDRESS parameters, e.g. for SaveData or BlockTransfer. The address of a variable is loaded by writing the address operand & in front of the variable name.

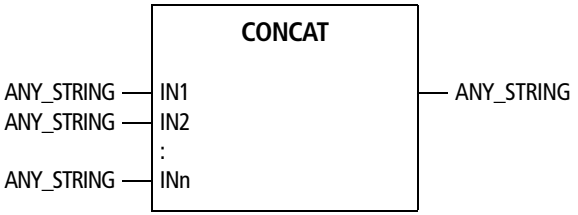
Example: addition of addresses

```
PROGRAM adr_add
VAR
  Start_data_field AT %MB0.0.0.0 : BYTE ;
  Shift: UINT := 10 ;
  Copy : BlockTransfer ;
END_VAR

  LD &Start_Data_Field
  ADD Offset
  ST Copy.SourceAddress
END_PROGRAM
```

IEC **PS4-300**
PS416

CONCAT
Concatenation of strings



Function prototype

Meaning of operands

IN1 Single strings to be concatenated into one string.

.
.

INn

Description

String IN1 in the working register and string IN2 to INn are concatenated. The resulting string is loaded into the working register. Strings IN1 to INn are written from left to right in ascending order.

07/01 AWB2700-1306-GB

Example “Concatenate strings”

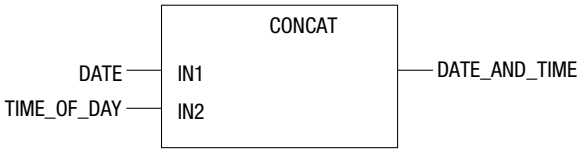
```
PROGRAM n_texts

VAR
  Text1 : STRING := 'ab';
  Text2 : STRING := 'cd';
  Text3 : STRING := 'ef';
  Texts_1_2_3 : STRING;
END_VAR

  LD       Text1
  CONCAT   Text2, Text3
  ST       Texts_1_2_3    (* Texts_1_2_3:
                          'abcdef' *)
END_PROGRAM
```

IEC **PS4-300**
PS416

CONCAT
Concatenation of time and date



Function prototype

Meaning of operands
IN1 Date

IN2 Time1

Description
Date IN1 and time IN2 are combined to form a data element of type DATE_AND_TIME and the result is loaded into the working register.

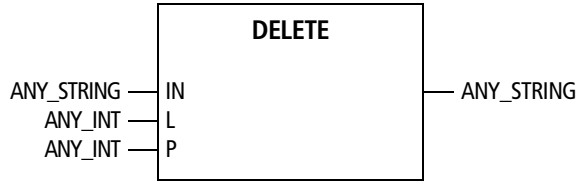
Example “Deadline from time and date”
```
PROGRAM deadline

VAR
  Date1 : DATE := D#1997-05-10;
  Time : TIME_OF_DAY := TOD#6:35:00.00;
  Date_and_Time1 : DATE_AND_TIME;
END_VAR

  LD       Date
  CONCAT   Time
  ST       Date_and_Time1
                   (* Date_and_Time1:
                   DT#1997-05-10-6:35 *)
END_PROGRAM
```

IEC **PS4-300**
PS416

DELETE
Delete n characters from a string



Function prototype

Meaning of operands

IN String containing the series of characters to be deleted, made available via the working register.

L Number of characters to be deleted.

P Position of the character to be deleted from. 1 corresponds to 1st character.

Description

L characters from position P are to be deleted from string IN. The resulting string is loaded into the working register.

If L = 0 or P > than the current length of IN, IN is returned unchanged.

Example “Delete three characters from a string”

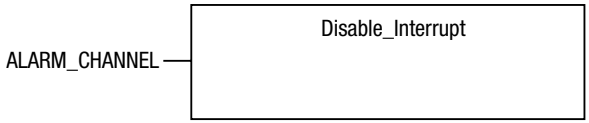
```
PROGRAM delete

VAR
  Text1 : STRING := 'abABCDcd';
  Length : UINT := 3;
  Position : UINT := 5;
  Text2 : STRING;
END_VAR

  LD       Text1
  DELETE   Length, Position
  ST       Text2          (* Text2: 'abABd' *)
END_PROGRAM
```


07/01 AWB2700-1306-GB

Disable_Interrupt
Disable Interrupt



Function prototype

Description
An interrupt is disabled to prevent it interrupting the read or write process of a main program. The CounterAlarm, EdgeAlarm and TimerAlarm function blocks are used as interrupt sources.

The invocation of the function Disable_Interrupt disables the Interrupt source which one of the function blocks mentioned above has previously started. This synchronises the access of the main program and the Interrupt routine to common data. Write accesses are affected (ST, S, R). The instruction is programmed as an independent sequence.

An element of the ALARM_CHANNEL enumeration data type is transferred as an argument for the function Disable_Interrupt:

For PS4-200

```
 TYPE ALARM_CHANNEL
   (CounterAlarm_Channel,
   TimerAlarm_Channel,
   EdgeAlarm_Channel):= CounterAlarm_Channel;
END_TYPE
```

For PS4-300

```
TYPE ALARM_CHANNEL
  (CounterAlarm_Channel, (*0*)
       TimerAlarm_Channel,  (*1*)
       EdgeAlarm_Channel_1, (*2*)
  EdgeAlarm_Channel_2):= CounterAlarm_Channel;
END_TYPE
```

For PS416

```
TYPE ALARM_CHANNEL
  (CounterAlarm_Channel, (*0*)
  TimerAlarm_Channel,    (*1*)
  EdgeAlarm_Channel_1,   (*2*)
  EdgeAlarm_Channel_2,   (*3*)
  EdgeAlarm_Channel_3,   (*4*)
  EdgeAlarm_Channel_4,   (*5*)
  EdgeAlarm_Channel_5,   (*6*)
  EdgeAlarm_Channel_6,   (*7*)
  EdgeAlarm_Channel_7,   (*8*) (*.._8: 9*)
  EdgeAlarm_Channel_8):= CounterAlarm_Channel;
END_TYPE
```

If the Interrupt source is disabled, all instances of alarm function blocks belonging to this interrupt source will also be disabled at the same time.

07/01 AWB2700-1306-GB

Example: disable interrupt

```
PROGRAM D_Alarm

VAR_GLOBAL
  Buffer : INT ;
  Edge_alarm : EdgeAlarm ;
  I_Routine : AlarmFB ;
END_VAR

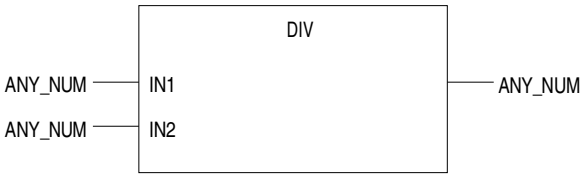
  CAL Edge_alarm    (EnableDisable := 1,
                     Number := 0,
                     RisingEdge := 0,
                     SetPointValue := 1,
                     Subroutine := I_Routine)
  LD EdgeAlarm_Channel
  Disable_Interrupt (* Alarm source for Edge *)
                     (* Alarm is disabled *)
  LD 500
  ST Buffer
  LD EdgeAlarm_Channel
  Enable_Interrupt  (* Alarm source is *)
                     (* enabled again *)
END_PROGRAM

FUNCTION_BLOCK AlarmFB
VAR_EXTERNAL
  Buffer : INT ;
END_VAR

  LD 30000
  ST Buffer
END_FUNCTION_BLOCK
```

IEC

DIV
Division of Numeric Data Types



Function prototype

Meaning of operands

IN1 The dividend is provided via the working register.

IN2 Divisor

For this function all of the input operands must be of the same data type.

Description

The value in the working register is divided by the value of the operand IN2. Both input operands are of the same data type. The result is made available in the working register for further processing. If a division operation results in a remainder, this will be added as a carry to the carry register.

Example: division of numeric data types

```
PROGRAM divide

VAR
  A : INT := 15;
  B : INT := 3;
  Quotient : INT;
END_VAR

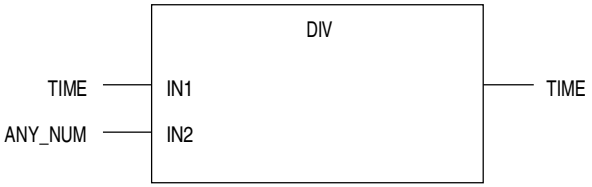
  LD   A
  DIV  B
  ST   Quotient              (* Quotient: 5 *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC **PS4-300**
PS416

DIV
Division of Duration



Function prototype

Meaning of operands

IN1 Time value of time duration data type

IN2 Value by which the address operand IN1 is divided.

Example: division of a time value

```
PROGRAM Divide_time

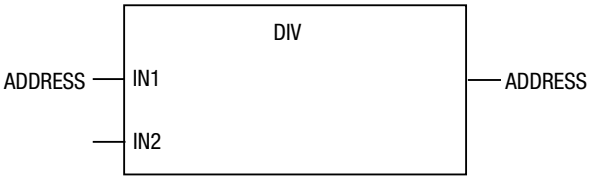
VAR
  Time_old : TIME := T#2h34m;
  Number : SINT := 2;
  Time_new : TIME;
END_VAR

  LD   Time_old
  DIV  Number
  ST   Time_new (* Time_new :T#1h17m *)

END_PROGRAM
```

PS4-200
PS4-300

DIV
Division of Addresses



Function prototype

The following data type combinations of the operands are possible:

Input operand IN1	Input operand IN2	Output operand
ADDRESS	ANY_INT	ADDRESS
ADDRESS	ADDRESS	ADDRESS

Description

The input variable is divided by a divisor and the result is loaded into the working register.

Loops or conditions for function blocks can be calculated with ADDRESS parameters, e.g. for SaveData or BlockTransfer. The address of a variable is loaded by writing the address operand & in front of the variable name.

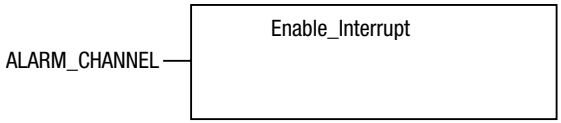
Example: division of addresses

```
PROGRAM addr_div
VAR
  Start_Data_Field AT %MB0.0.0.30 : BYTE ;
  Offset : UINT := 10 ;
  Copy : BlockTransfer ;
END_VAR

  LD &Start_Data_Field
  DIV Offset
  ST Copy.SourceAddress
END_PROGRAM
```


07/01 AWB2700-1306-GB

Enable_Interrupt
Enable Interrupt



Function prototype

Description
An interrupt which has previously been disabled via the function Disable_Interrupt is enabled via the Enable_Interrupt function. The CounterAlarm, EdgeAlarm and TimerAlarm function blocks are used as interrupt sources. The instruction is programmed as an independent sequence.

An element of enumerated data type ALARM_CHANNEL is transferred as argument for the function Enable_Interrupt.

For PS4-200

```
TYPE ALARM_CHANNEL
  (CounterAlarm_Channel,
   TimerAlarm_Channel,
   EdgeAlarm_Channel):= CounterAlarm_Channel;
END_TYPE
```

For PS4-300

```
TYPE ALARM_CHANNEL
  (CounterAlarm_Channel, (*0*)
   TimerAlarm_Channel,   (*1*)
   EdgeAlarm_Channel_1,  (*2*)
   EdgeAlarm_Channel_2):= CounterAlarm_Channel;
END_TYPE
```

For PS416

```
TYPE ALARM_CHANNEL
 (CounterAlarm_Channel, (*0*)
 TimerAlarm_Channel, (*1*)
 EdgeAlarm_Channel_1, (*2*)
 EdgeAlarm_Channel_2, (*3*)
 EdgeAlarm_Channel_3, (*4*)
 EdgeAlarm_Channel_4, (*5*)
 EdgeAlarm_Channel_5, (*6*)
 EdgeAlarm_Channel_6, (*7*)
 EdgeAlarm_Channel_7, (*8*) (*.._8: 9*)
 EdgeAlarm_Channel_8):= CounterAlarm_Channel;
END_TYPE
```

If the interrupt source is enabled, all instances of the alarm function blocks related to this interrupt source will be also enabled simultaneously.

Example: Enable alarm

```
PROGRAM E_Alarm

VAR_GLOBAL
  Buffer : INT ;
  Edge_alarm : EdgeAlarm ;
  I_Routine : AlarmFB ;
END_VAR

  CAL Edge_Alarm (EnableDisable := 1,
                  Number := 0,
                  RisingEdge := 0,
                  SetPointValue := 1,
                  Subroutine := I_Routine)
  LD EdgeAlarm_Channel
  Disable_Interrupt
  (*Alarm source for EdgeAlarm is disabled*)
  LD 500
  ST Buffer
  LD EdgeAlarm_Channel
  Enable_Interrupt
  (*Alarm source is enabled*)

END_PROGRAM
```


07/01 AWB2700-1306-GB

```
FUNCTION_BLOCK AlarmFB

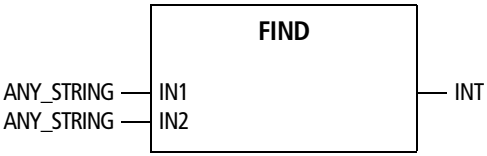
VAR_EXTERNAL
  Buffer : INT ;
END_VAR

  LD 30000
  ST Buffer

END_FUNCTION_BLOCK
```

IEC **PS4-300**
PS416

FIND
Find a string within a string



Function prototype

Meaning of operands

IN1 Basic string containing a specific sequence of characters that is to be found. It is made available via the working register.

IN2 Sequence of characters for which the basic string IN1 is searched.

Description

A specific sequence of characters is searched for in the basic string IN1. If the characters searched for are found, the position of the first character is entered in the working register. Otherwise the value 0 is output.

If the succession of characters is present several times in the basic string, the first string found is taken. A 0 is returned if one of the parameters is an empty string.

07/01 AWB2700-1306-GB

Example “Find character sequence in string”

```
PROGRAM find

VAR
  Basic_Text : STRING := 'On_condition';
  Find_Text : STRING := 'On';
  Position : INT;
END_VAR

  LD   Basic_Text
  FIND Find_Text
  ST   Position          (* Position: 4 *)
END_PROGRAM
```

GetCarryRegister
Get carry with multiplication or remainder of a division



Function prototype

Description

The function GetCarryRegister refers to the result of a multiplication or division that has just been executed. If a multiplication has produced a carry, this is stored in the carry register. If a division has produced a remainder, this is also stored in the carry register. The function GetCarryRegister therefore accesses the carry register and loads its contents into the working register.

No carry or remainder is formed after multiplication or division of USINT, SINT and real data types (PS416, PS4-300). The compiler supplies the corresponding error message.

Example: get remainder of a Division

```
PROGRAM remainder
VAR
  Number : INT := 127;
  Result_integer : INT;
  Result_remainder : INT;
END_VAR

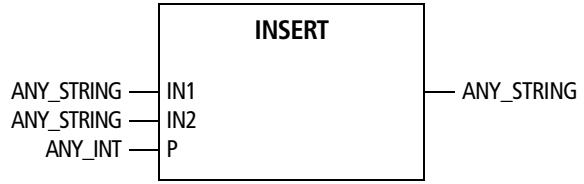
  LD Number
  DIV 4
  ST Result_integer       (* Result_integer : 31 *)
  GetCarryRegister
  ST Result_remainder     (* Result_remainder : 3*)
       .
       .

END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC **PS4-300**
PS416

INSERT
Insert String in String



Function prototype

Meaning of operands

IN1 Basic string in which another string (IN2) is to be inserted. The basic string is provided via the working register.

IN2 String to be inserted in the basic string.

P Position of the character in the basic string from which the string is inserted. 1 means the 1st character.

Description

String IN2 is inserted into the basic string IN1 after the character with the number specified by the P input operand. The resulting string is loaded into the working register. If P is greater than the current length of IN1, IN1 is returned unchanged.

Example “Insert string within string”

```
PROGRAM insert

VAR
  Text1 : STRING := 'abcd';
  Text2 : STRING := 'ABCD';
  Position : INT := 2;
  newText : STRING;
END_VAR

  LD       Text1
  INSERT   Text2, Position
  ST       newText        (* newText:
                           'abABCDcd' *)
END_PROGRAM
```


07/01 AWB2700-1306-GB

IsBitTrue
Test of a Single Bit in the Result



Function prototype

Meaning of operands

IN Bit pattern in which a specified bit is tested

N Bit position

Input operand	Input operand	Output operand
ANY_BIT	USINT	Bool
ANY_BIT	UINT	Bool

Description

The status of the bit specified by operand N in bit pattern IN is tested. The resulting value is sent to the working register as the result.

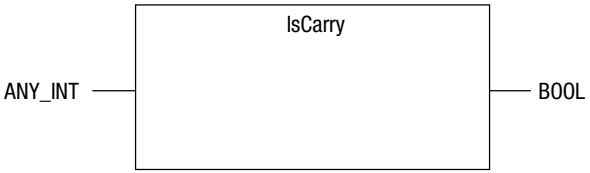
This function is best used when followed by conditional operations such as JMPC, JMPCN, CALC, CALCN, RETC or RETCN.

Example: Testing a single bit in the result

```
PROGRAM bittest
VAR
  BitPattern  : BYTE := 2#11010011;
  Position    :UINT := 4;
  BitValue    : BOOL;
END_VAR

  LD BitPattern
  IsBitTrue Position
  ST BitValue (* BitValue : 1 *)
END_PROGRAM
```

IsCarry
Test of carry bit



Function prototype

Description
The function IsCarry refers to the result of the previously executed arithmetic function which checks the Boolean value of the carry bit and transfers the result to the working register.

The IsCarry function should only be used with shift operations (SHL,SHR etc.) since it is the carry bit which is evaluated here. To evaluate the overshoot of the value range in arithmetical operations, you must use the IsOverflow function.

This function is best used when followed by conditional operations such as JMPC(N), CALC(N), or RETC(N).

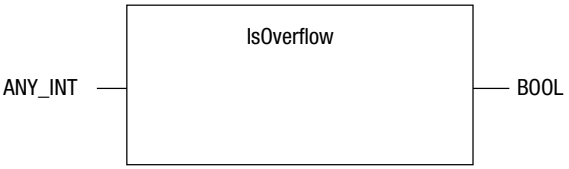
Example: Testing the carry bit

```
PROGRAM carrybit
VAR
  Edge : R_TRIG;
  Signal : AT %I0.0.0.0.0 : BOOL;
  Impulse : BOOL;
  Byte1 : BYTE := 2#11110000;
END_VAR

  LD Byte1
  SHL 3
  BYTE_TO_USINT
  IsCarry                (* Carry register = 1 *)
  CALC Edge (CLK := Signal)
  LD Edge.Q
  ST Impulse
END_PROGRAM
```


07/01 AWB2700-1306-GB

IsOverflow
Test of overflow bit



Function prototype

Description

The function refers to the result of the previously executed arithmetic function which transfers the Boolean value of the overflow bit to the working register. The overflow bit has the Boolean value 1 if the result of the previous operation exceeds the entered value range of the data type. If the result is within the value range, the overflow bit is 0.

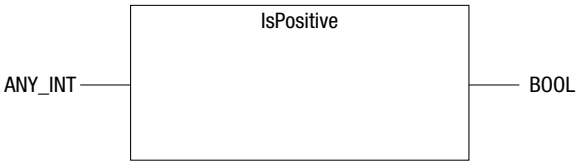
This function is best used when followed by conditional operations such as JMPC, JMPCN, CALC, CALCN, RETC or RETCN.

Example: Testing the overflow bit

```
PROGRAM overflow
VAR
  Edge : R_TRIG;
  Signal : AT %I0.0.0.0.0 : BOOL;
  Impulse : BOOL;
  Number1 : SINT := 127;
END_VAR

  LD Number1
  ADD 12
  IsOverflow
  CALC Edge (CLK := Signal)
  LD Edge.Q
  ST Impulse
END_PROGRAM
```

IsPositive
Test Whether Result is Positive



Function prototype

Description

The function IsPositive refers to the result of the previously executed arithmetic function. If the result is greater than or equal to zero, 1 is loaded into the working register.

The IsPositive function is compatible with Sucosoft S 30. When used with unsigned variables (UINT,USINT etc.) which are greater than the maximum range/2, IsPositive will deliver a 0 but in all other cases a 1.

This function is best used when followed by conditional operations such as JMPC(N), CALC(N), RETC(N).

Example: Testing whether the result is positive

```
PROGRAM res_pos

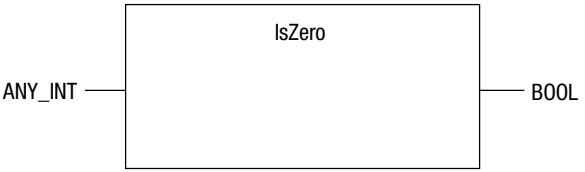
VAR
  Edge : R_TRIG;
  Signal : AT %IO.0.0.0.0 : BOOL;
  Impulse : BOOL;
  Number1 : INT := 12;
END_VAR

  LD Number1
  SUB 10
  IsPositive
  CALC Edge (CLK := Signal)
  LD Edge.Q
  ST Impulse

END_PROGRAM
```


07/01 AWB2700-1306-GB

IsZero
Test of zero bit



Function prototype

Description

The function IsZero refers to the result of the previously executed arithmetic function which transfers the Boolean value of the zero bit to the working register. The zero bit has the value 0 if the previous operation supplies the value 0. If the result is not equal 0, the zero bit is 1.

This function is best used when followed by conditional operations such as JMPC, JMPCN, CALC, CALCN, RETC or RETCN.

Example: Testing the zero bit

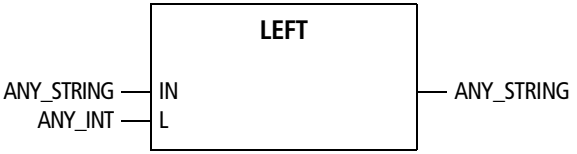
```
PROGRAM zero_bit
VAR
  Edge : R_TRIG;
  Signal : AT %IO.0.0.0.0 : BOOL;
  Impulse : BOOL;
  Number1 : INT := 12;
END_VAR

  LD Number1
  SUB 12
  IsZero
  CALC Edge (CLK := Signal)
  LD Edge.Q
  ST Impulse

END_PROGRAM
```

IEC **PS4-300**
PS416

LEFT
Select n characters left of a string



Function prototype

Meaning of operands

IN String containing the left section of length L to be taken. It is loaded into the working register.

L Number of characters left of the STRING operand IN to be loaded into the working register.

Description

The function LEFT takes the left section of the string present in the working register and enters this section in the working register. The number of characters to be taken is specified by input operand L.

Example “Take the first four characters”

```
PROGRAM le_text

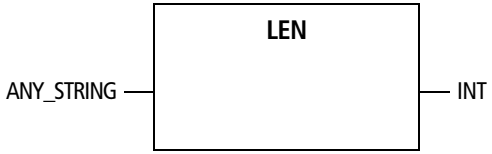
VAR
  String1 : STRING := 'Teststring';
  CharNo : INT := 4;
  n_left_chars : STRING;
END_VAR

  LD   String
  LEFT CharNo
  ST   n_left_chars       (* n_left_chars:
                          'Test' *)
END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC

LEN
Determination of the Character String Length



Function prototype

Description

The function LEN determines the length of the character string in the working register (input operand of data type STRING) and enters the determined value as INT number into the working register.

Example: determining the character string length

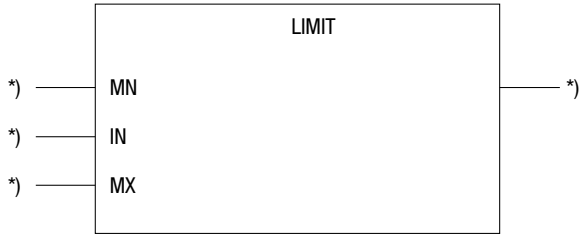
```
PROGRAM how_long
VAR
  Text : STRING := 'Term';
  TextLength : INT;
END_VAR

  LD   Text
  LEN
  ST   TextLength        (* TextLength: 7 *)

END_PROGRAM
```

IEC **PS4-300**
PS416

LIMIT
Limitation



Function prototype

*) Supported data types: ANY_BIT, ANY_NUM, ANY_DATE, TIME, STRING (1).
All parameters must be the same data type.

Meaning of operands

MN Lower limit value

IN Test value

MX Upper limit value

Description

The values MN and MX specify the upper and lower limit value. The function compares the test value IN with MN and MX. If the IN value is between the two limit values, it is loaded into the working register. If IN is less than MN, the MN value is output. The MX value is output if IN is greater than MX.

07/01 AWB2700-1306-GB

Example “Measured value between 17 and 124”

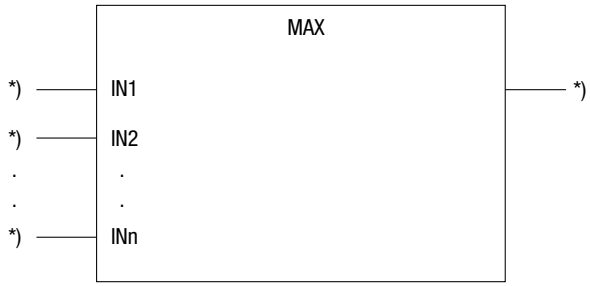
```
PROGRAM limits

VAR
  max_val : INT := 124;
  min_val : INT := 17;
Measval AT %IW0.0.0.10:INT; (*IW0.0.0.10 *)
                          (* with value 135 *)
  Result : INT;
END_VAR

  LD       min_val
  LIMIT    Measval, max_val
  ST       Result         (* Result: 124 *)
END_PROGRAM
```

IEC **PS4-300**
PS416

MAX
Maximum selection



Function prototype

*) Supported data types: ANY_BIT, ANY_NUM, ANY_DATE, TIME, STRING.
All parameters must be the same data type.

Meaning of operands

IN1 Input operand for which the highest value is searched.

.
.

INn

Description

The MAX function determines which of the input operands has the highest value. The specified operand is loaded into the working register.

With STRING operands the current length is first compared and the longer operand is loaded into the working register. If both operands are the same length, the operand is loaded which has a more significant character.

07/01 AWB2700-1306-GB

Example “find maximum”

```
PROGRAM maximum

VAR
  Value1 : SINT := 155;
  Value2 : SINT := -24;
  Value3 : SINT := -3;
  Value4 : SINT := 5;
  Maximumval : SINT;
END_VAR

  LD   Value1
  MAX  Value2, Value3, Value4
  ST   Maximumval         (* Maximumval: 155 *)
END_PROGRAM
```

IEC **PS4-300**
PS416

MID
Section of a String



Function prototype

Meaning of operands

IN String from which a sequence of characters is to be removed. It is made available via the working register.

L Number of IN string characters to be cut.

P Position of the first character in the string to be cut.
1 corresponds to 1st character.

Description

L characters are to be cut from string IN starting from position P. The cut string is loaded into the working register. If P > current length of IN, IN is returned unchanged.

07/01 AWB2700-1306-GB

Example “Take part of string”

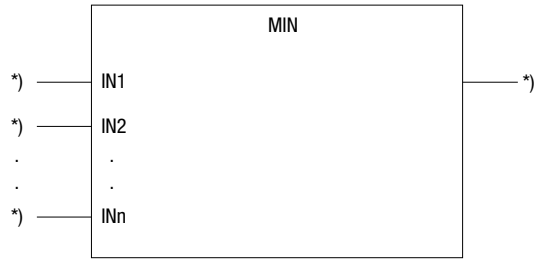
```
PROGRAM parttext

VAR
  String : STRING := 'Test_string';
  CharNo : INT := 7;
  Start : INT := 5;
  Text_Section : STRING;
END_VAR

  LD   String
  MID  CharNo, Start
  ST   Text_Section      (* Text_Section:
                         'Characters' *)
END_PROGRAM
```

IEC **PS4-300**
PS416

MIN
Minimum selection



Function prototype

*) Supported data types: ANY_BIT, ANY_NUM, ANY_DATE, TIME, STRING.
All parameters must be the same data type.

Meaning of operands

IN1 – INn Input operand for which the lowest value is searched.

Description

The MIN function determines which of the input operands has the lowest value. The specified operand is loaded into the working register.

With STRING operands the current length is first compared and the shorter operand is loaded into the working register. If both operands are the same length, the operand is loaded which has a less significant character.

Example “Find minimum”

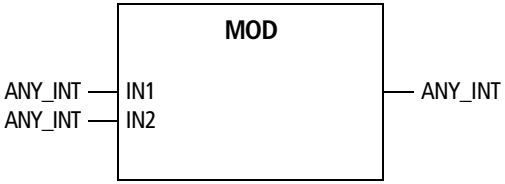
```
PROGRAM minimum

VAR
  Value1 : SINT := 155;
  Value2 : SINT := -24;
  Value3 : SINT := -3;
  Value4 : SINT := 5;
  Minimumval : SINT;
END_VAR
  LD   Value1
  MIN  Value2, Value3, Value4
  ST   Minimumval         (* Minimumval: -24 *)
END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC **PS4-300**
PS416

MOD
Remainder Division



Function prototype

Description:

The input variable is divided by a divisor and the remainder is loaded into the working register. The MOD function is permissible for all ANY_INT types and is only supported by the PS416 and PS-300.

On the PS-200 the remainder of a division is processed using the GetCarryRegister function.

Example “MOD function” in the program “mod_div”

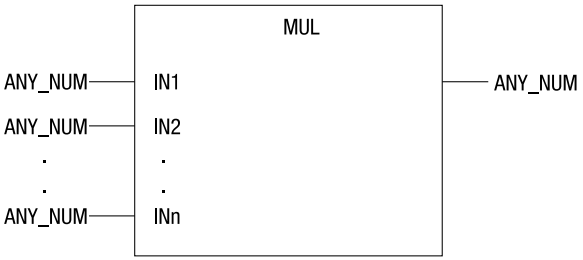
```
PROGRAM mod_div

VAR
  Dividend : UINT := 256;
  Divisor  : UINT := 50;
  Remainder: UINT ;
END_VAR

  LD   Dividend
  MOD  Divisor
  ST   3Remainder
END_PROGRAM
```

IEC

MUL
Multiplication of Numeric Data Types



Function prototype

Meaning of operands

IN1 The first operand is supplied via the working register.

IN2 Further operands involved in the multiplication.

.
.

INn

For this function all of the input operands must be of the same data type.

Description

The value in the working register is multiplied by the values of the operands IN2 to INn. All input operands are of the same numeric data type. The result is made available in the working register for further processing. If the result exceeds the range of values permitted for the data type used an overflow will be formed.

07/01 AWB2700-1306-GB

Example: multiplication of numeric operands

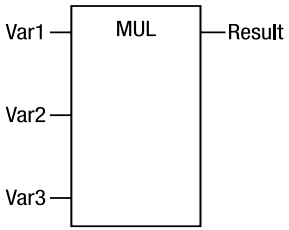
```
PROGRAM mul3times

VAR
  A : SINT := 5;
  B : SINT := 2;
  C : SINT := 3;
  Product_3times : SINT;
END_VAR

  LD   A
  MUL  B
  MUL  C
  ST Product_3times       (* Product_3times: 30 *)
END_PROGRAM
```

Example: invocation of the function MUL in FBD

The MUL function is extendable in the LD and FBD graphical programming languages – you can specify more than two input operands.



Example: invocation of the function MUL in IL

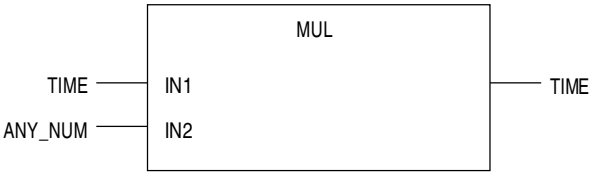
In IL multiple input operands are handled by the sequential processing of two operands each time. The following IL listing shows how this is done. The second listing generates an error message with S40 versions before V5.0.

```
Listing 1:LD Var1
          MUL Var2
          MUL Var3
          ST Result

Listing 2: LD Var 1
          MUL Var2, Var3
          ST Result
```

IEC **PS4-300**
PS416

MUL
Multiplication of Time



Function prototype

Meaning of operands

IN1 Time value of time duration data type

IN2 Value by which operand IN1 is to be multiplied.

Description

The time duration operand IN1 is multiplied by the value IN2 and the result is loaded into the working register.

Example: multiplication of a time value

```
PROGRAM time_mul

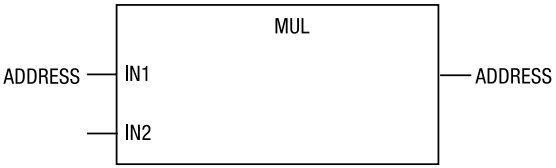
VAR
  Time_old : TIME := T#2h34m;
  Number : USINT:= 2;
  Time_new : TIME;
END_VAR

  LD   Time_old
  MUL  Number
  ST   Time_new        (* Time_new: T#5h8m *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

MUL
Multiplication of addresses



Function prototype

The following data type combinations of the operands are possible:

Input operand IN1	Input operand IN2	Output operand
ADDRESS	ANY_INT	ADDRESS
ADDRESS	ADDRESS	ADDRESS

Description

The input variable is increased by a factor and the result is loaded into the working register.

Loops or conditions for function blocks can be calculated with ADDRESS parameters, e.g. for SaveData or BlockTransfer. The address of a variable is loaded by writing the address operand & in front of the variable name.

Example: multiplication of addresses

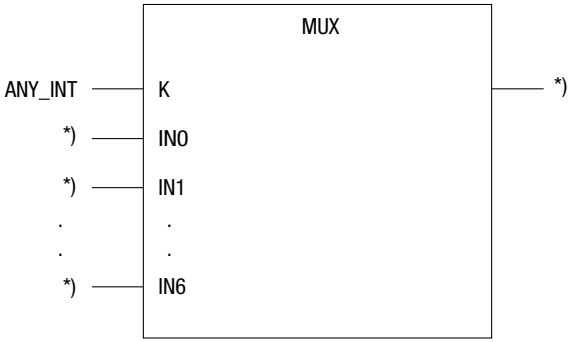
```
PROGRAM adr_mul

VAR
  Start_ Datafield AT %MB0.0.0.2 : BYTE ;
  Offset : UINT := 10 ;
  Copy : BlockTransfer ;
END_VAR

  LD &Start_ Datafield
  MUL Offset
  ST Copy.SourceAddress
END_PROGRAM
```

IEC **PS4-300**
PS416

MUX
Multiplexer



Function prototype

Meaning of operands

K Selection operand. Its numeric value depends on the selected input operand IN0 to INn. It is made available via the working register.

The valid value range for K is 0 to 6. If the value is outside of the valid range, the highest permissible value is accepted.

IN0 Up to seven input operands between which the selection is made. The input operand with the number is selected that matches the value K.

.

.

IN6

Description

The MUX function selects an input operand between IN0 and IN6 and transfers it to the working register. The selection criterion is the operand number 0 to 6 defined by input operand K.

07/01 AWB2700-1306-GB

Example “Select second input operand”

```
PROGRAM select

VAR
  IN_Nr : USINT := 2;
  Numb0 : INT := 15;
  Numb1 : INT := -123;
  Numb2 : INT := 1;
  SelectedNo : INT;
  a, b, c, d, e, f, g, h, i, j, K : BYTE;
  result : BYTE;
END_VAR

  LD   IN_Nr
  MUX  Numb0, Numb1, Numb2
  ST   SelectedNo           (* SelectedNo: 1 *)
END_PROGRAM
```

Example “Selection with more than seven input operands”

If more than seven parameters are required, the MUX command must be converted into several conditional calls

```
(* 10 input_operands..*)

  LD   K
  MUX  a, b, c, d, e, f, g, h, i, j
  ST   result

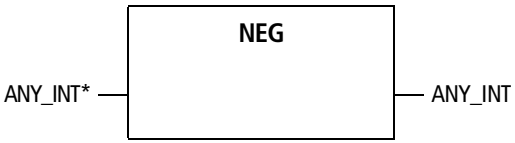
(* to be converted for operands 1 to 7: *)
  LD   K
  GE   7
  JMPC mux2

  LD   K
  MUX  a, b, c, d, e, f, g
  ST   result
  JMP  go_on

(* and operands 8 to 10: *)
MUX2:
  LD   K
  SUB 7
  MUX h, i, j
  ST result

go_on:
```

NEG
Two's complement



Function prototype

* Also supported: ANY_NUM, ANY_BIT (without BOOL) and Time

Description

The two's complement is formed for the numeric value of data type ANY_INT. The result is made available in the working register for further processing. The two's complement of signed numbers corresponds to the negation of a numeric value. For example, 100 is converted to –100. With unsigned values the two's complement formed is a different value.

Example: two's complement

```
  LD Signed_Number
  NEG
```

Invocation of the function NEG

```
PROGRAM complement

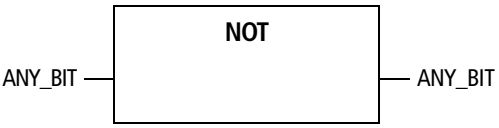
VAR
  Number : INT := 230;
  Complement_Number : INT;
END_VAR

  LD Number
  NEG
  ST   Complement_Number (* Now contains the
                          value -230 *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

NOT
Bit complement formation



Function prototype

Description

If the input operand is of data type BOOL, the bit in the working register is negated. If the input operand is of data type BYTE, WORD or DWORD[1)] the individual bits of the working register are negated, i.e. the one's complement is formed.

The result is made available in the working register for further processing.

The NOT function is used if it is necessary to load or assign a negated value of data type BOOL, BYTE, WORD or DWORD[1)].

Example: function NOT

```
  LD BitPattern
  NOT
```

Example: binary complement formation

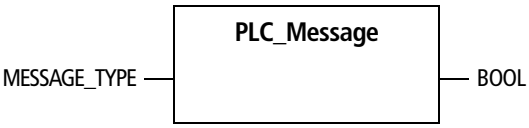
```
PROGRAM negation
VAR
  BitString : BYTE := 2#00110101;
  Negation_Result : BYTE;
END_VAR

  LD   BitString
  NOT
  ST   Negation_Result (* Negation_Result
                          2#11001010 *)

END_PROGRAM
```

[1)] Only for PS416 and PS4-300

PLC_Message
Evaluate System Diagnostics Message



Function prototype

Description

An element of the enumerated data type Message_Type is sent to the input of type Message_Type via the working register. The return value is a BOOL value and indicates the status of the bit in the message word selected via the enumerated data element.

The enumeration type Message_Type which should be used contains the following elements:

For PS4-200

```
TYPE
  MESSAGE_TYPE
  (ISA,    (*First cycle after start*)
   IRE,    (*RESET button has been pressed*)
   IFO,    (*Forcing in RUN active *)
   REC,    (*Remaining cycle processed after restart*)
   ICS,    (*PLC cold started *
   NKD_1,  (*New data to SBI 1*)
   NKD_2,  (*New data to SBI 2*)
   NKD_3   (*New data to SBI 3*)
  ) := ISA;
END_TYPE
```


07/01 AWB2700-1306-GB

For PS4-300

```
TYPE
  MESSAGE_TYPE
  (ISA, (* =0:First cycle after start *)
  IRE, (* =1:RESET button actuated *)
  IFO, (* =2:Forcing in RUN active *)
  REC, (* =3:Remaining cycle run
   after restart *)
   ICS, (* =4:PLC cold started*)
   NKD_1, (* =5:New data to SBI 1*)
   NKD_2, (* =6:New data to SBI 2*)
   NKD_3, (* =7:New data to SBI 3*)
   ICT, (* =8:ECT in first cycle *)
   IAH, (* =9:EAH in first cycle *)
   IRT, (* =10:ERT in first cycle *)
  ) := ISA;
END_TYPE
```

For PS416

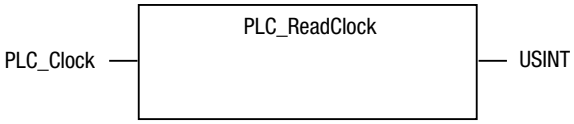
```
TYPE
  MESSAGE_TYPE
  (ICS, (* =0:PLC cold started*)
   ISA, (* =1:1st cycle after start*)
   RTW, (* =2:Runtime warning*)
   REC, (* =3:Remaining cycle completed after restart
           *)
   DAKE,(* =4:AK-Error; Card not addressable
           or fitted*)
   KOM, (* =5:Communication partner
           failed*)
  ) := ISA;
END_TYPE
```

The data type has been defined by Moeller and can be used in the POUs immediately. The standard setting is ISA.

Example: evaluating system diagnostics message

```
  LD                    ISA
  PLC_Message
  JMPCN                 No_new_start
```

PLC_ReadClock
Reading single element from real-time clock



Function prototype

Description
An element of the enumerated data type PLC_CLOCK is sent to the input via the working register. The return value is of type USINT and indicates the value of the specified element of the real-time clock.

The enumerated data type PLC_CLOCK to be used contains the following elements:

```
TYPE
  PLC_CLOCK
  (CK_YEAR, (*Year*)
       CK_MONTH, (*Month*)
       CK_DAY, (*Day*)
       CK_WEEKDAY, (*Weekday; 0 = Sunday*)
       CK_HOURS, (*Hours*)
       CK_MINUTES, (*Minutes*)
           CK_SECONDS (*Seconds*)
  ) := CK_YEAR;
END_TYPE
```

The data type has been defined by Moeller and can be used in the POUs immediately. CK_YEAR is the standard setting.

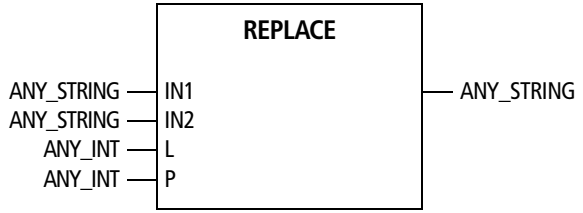
Example: read an element of the real-time clock

```
PROGRAM RD_Clock
VAR
  Hours : USINT ;
END_VAR
  LD CK_HOURS
  PLC_ReadClock
  ST Hours
END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC **PS4-300**
PS416

REPLACE
Replace characters in a string



Function prototype

Meaning of operands

IN1	String containing characters to be overwritten. It is made available via the working register.
IN2	String in which string IN1 is used.
L	Number of characters to be deleted.
P	Position of the character in the string from which the exchange is to be made. 1 corresponds to 1st character.

Description

A number of characters in string IN1 as specified by the input operand L is overwritten from position P with string IN2. The number of deleted and inserted characters can vary. The resulting string is loaded into the working register.

L = 0 corresponds to the INSERT function. If P > current length of IN1, IN1 is returned unchanged.

Example “Replace sections of a string”

```
PROGRAM replace

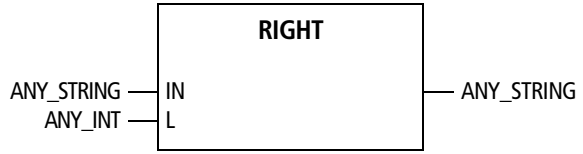
VAR
  Old_Text : STRING := 'abbCDEe';
  New_Chars : STRING := 'cd';
  New_Text : STRING;
END_VAR

  LD       Old_Text
  REPLACE  New_Chars, 4, 3 (*L=4, P=3*)
  ST       New_Text          (* New_Text:
                             'abcde *)
END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC **PS4-300**
PS416

RIGHT
Select n characters right of string



Function prototype

Meaning of operands

IN String containing right section of length L to be taken. It is made available via the working register.

L Number of characters from right of STRING operand IN to be loaded in the working register.

Description

The function RIGHT takes the right section of the string in the working register and transfers it to the working register. The number of characters to be taken is specified by input operand L.

Example “Take right section of a string”

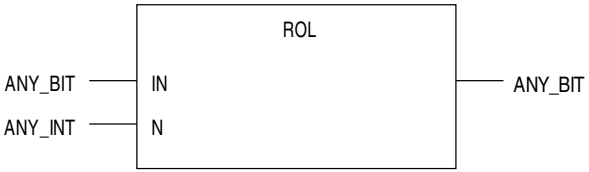
```
PROGRAM re_text

VAR
  String1 : STRING := 'Teststring';
  CharNo : INT := 5;
  n_right_chars : STRING;
END_VAR

  LD      String1
  RIGHT   CharNo
  ST      n_right_chars (* n_right_chars:
                           'string'*)
END_PROGRAM
```

IEC

ROL
Rotation Left



Function prototype

Meaning of operands

IN The number (bit pattern) to be rotated of the data type BYTE, WORD or DWORD[1].

N Number of places by which the operand IN is rotated to the left.

Description

The bit pattern entered as the operand IN is shifted to the left by the number of places specified in the operand N. The free places are filled with the shifted bits. This shifted bit pattern is provided in the working register. The result has the same data type as the input operand IN.

Example: bit pattern rotation to the left

```
PROGRAM rot_le

VAR
  Bit_pattern : BYTE := 2#01110001;
  Places : INT := 3;
  NewBitPattern : BYTE;
END_VAR

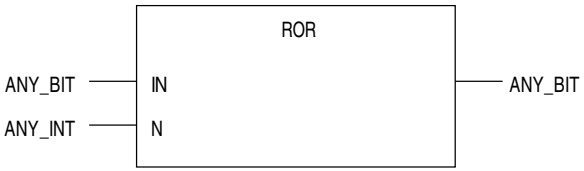
  LD   Bit_pattern
  ROL  Places
  ST   NewBitPattern      (* NewBitPattern:
                             2#10001011 *)
END_PROGRAM
```

[1] Only for PS4-300 and PS416

07/01 AWB2700-1306-GB

IEC

ROR
Rotation Right



Function prototype

Meaning of operands

IN The number (bit pattern) to be rotated of the data type BYTE, WORD or DWORD[1)].

N Number of places by which the operand IN is rotated to the right.

Description

The bit pattern entered as the operand IN is shifted to the right by the number of places specified in the operand N. The free places are filled with the shifted bits. This shifted bit pattern is provided in the working register. The result has the same data type as the input operand IN.

Example: bit pattern rotation to the right

```
PROGRAM rot_ri

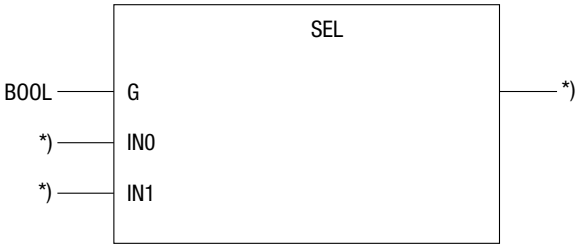
VAR
  Bit_pattern : BYTE := 2#01110001;
  Places : INT := 2;
  NewBitPattern : BYTE;
END_VAR

  LD   Bit_pattern
  ROR  Places
  ST   NewBitPattern      (* NewBitPattern:
                             2#01011100 *)
END_PROGRAM
```

1) Only for PS4-300 and PS416

IEC **PS4-300**
PS416

SEL
Conditional operand selection



Function prototype

*) Supported data types: All apart from ADDRESS
Parameter IN0 and IN1 must be the same data type.

Meaning of operands

G Boolean selection criterion

IN0 The first operand to be selected

IN1 The second operand to be selected

Description

The selection criterion SEL selects an input operand IN0 and IN1. The selection criterion is the logic state of the input operand G. The operand IN0 is selected with status 0 and input IN1 with status 1 of operand G. The specified operand is loaded into the working register.

07/01 AWB2700-1306-GB

Example “Select either 'ABC' or 'abc' ”

```
PROGRAM select

VAR
  Text1 : STRING := 'ABC';
  Text2 : STRING := 'abc';
  SelectorSwitch AT %I0.0.0.0.3 : BOOL;
  Selected : STRING;
END_VAR

  LD   SelectorSwitch
  SEL  Text1, Text2
  ST   Selected          (* The contents of the
                         variable selected
                         depends on the status
                         of input I0.0.0.0.3 *)
END_PROGRAM
```

ShiftLeftCarry
Shift left with carry

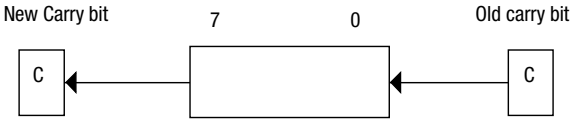


**BOOL is not supported*

Function prototype

Description
The bit pattern specified as the input operand is shifted one bit to the left in the working register. The bit position which becomes free is assigned the original carry bit and the bit removed becomes the new carry bit.

Data element of type BYTE:



Example: shift left with carry

```
PROGRAM shi_le

VAR
  Bit_pattern : BYTE := 2#01100110;
  NewBitPattern : BYTE;
END_VAR

  LD BitPattern
  ShiftLeftCarry
  ST NewBitPattern          (* NewBitPattern:
   .       1100110x, x at bit position 0 is
   .       the old value of the carry bit *)
END_PROGRAM
```


07/01 AWB2700-1306-GB

ShiftRightCarry
Shift Right with Carry



**BOOL is not supported*

Function prototype

Description

The bit pattern specified as the input operand is shifted one bit to the right in the working register. The bit position which becomes free is assigned the original carry bit and the bit removed becomes the new carry bit.

Data element of type BYTE:



Example: shift right with carry

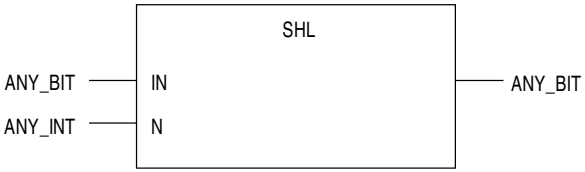
```
PROGRAM shi_ri

VAR
  Bit_pattern: BYTE := 2#01011100;
  NewBitPattern : BYTE;  (* NewBitPattern: x0101110,
                            x at bit position 7 is old
                            value of carry bit*)
END_VAR

  LD BitPattern
  ShiftRightCarry
  ST NewBitPattern
   .
   .
END_PROGRAM
```

IEC

SHL
Shift left



Function prototype

Meaning of operands

IN The number (bit pattern) to be shifted of the data type BYTE, WORD or DWORD[1)].

N Number of places by which the operand IN is shifted to the left.

Description

The bit pattern entered as the operand IN is shifted to the left by the number of places specified in the operand N. The free places are filled with the zeros. This shifted bit pattern is provided in the working register. The result has the same data type as the input operand IN.

Example: shift bit pattern left

```
PROGRAM shi_le

VAR
  Bit_pattern : BYTE := 2#00111011;
  Places : INT := 3;
  NewBitPattern : BYTE;
END_VAR

  LD   Bit_pattern
  SHL  Places
  ST   NewBitPattern (* NewBitPattern
                        2#11011000 *)
END_PROGRAM
```

1) Only for PS4-300 and PS416

07/01 AWB2700-1306-GB

IEC

SHR
Shift right



Function prototype

Meaning of operands

IN The number (bit pattern) to be shifted of the data type BYTE, WORD or DWORD[1)].

N Number of places by which the operand IN is shifted to the right.

Description

The bit pattern entered as the operand IN is shifted to the right by the number of places specified in the operand N. The free places are filled with the zeros. This shifted bit pattern is provided in the working register. The result has the same data type as the input operand IN.

Example: shift bit pattern right

```
PROGRAM shi_ri

VAR
  Bit_pattern : BYTE := 2#00111011;
  Places : INT := 2;
  NewBitPattern : BYTE;
END_VAR

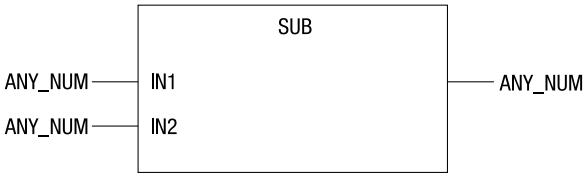
  LD   Bit_pattern
  SHR  Places
  ST   NewBitPattern      (* NewBitPattern:
                             2#00001110 *)

END_PROGRAM
```

1) Only for PS4-300 and PS416

IEC

SUB
Subtraction of Numeric Data Types



Function prototype

Meaning of operands

IN1 Minuend from which the operand IN2 is subtracted. The minuend is supplied via the working register.

IN2 Subtrahend which is subtracted from IN1.

Description

Operand IN2 is subtracted from the value in the working register (operand “IN1”). The result is made available in the working register for further processing. When negative numbers are subtracted, an overflow will be sent to the overflow register if the result exceeds the value range of the data type used.

For this function all of the input operands must be of the same data type.

Example: subtraction of numeric data types

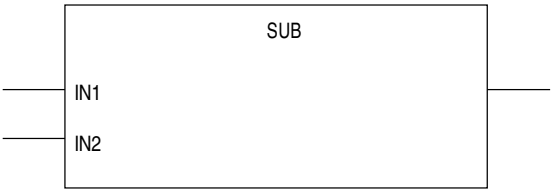
```
PROGRAM aminusb
VAR
  A : INT:= 123;
  B : INT:= 12;
  Difference : INT;
END_VAR

  LD   A
  SUB  B
  ST   Difference          (* Difference: 111 *)
END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC

SUB
Subtraction of Time and Date



Function prototype

The following data type combinations of the operands are possible:

Input operand IN1	Input operand IN2	Output operand
TIME	TIME	TIME
DATE[1)]	DATE	TIME
TIME_OF_DAY	TIME	TIME_OF_DAY
TIME_OF_DAY	TIME_OF_DAY	TIME
DATE_AND_TIME[1)]	TIME	DATE_AND_TIME
DATE_AND_TIME[1)]	DATE_AND_TIME	TIME

[1)] only for PS416 and PS4-300

Description
The input variable IN1 is reduced by the value of IN2 and the result is loaded into the working register.



Only for PS-200:
A SUB operation with the TIME data type only yields correct results in the positive value range. Do not run this operation with negative values.

Example: subtraction of time values

```
PROGRAM time_sub

VAR
  Time1 : TIME_OF_DAY := TOD#15:25:00.00;
  Time_value : TIME := T#3h6m15s;
  New_time : TIME_OF_DAY;
END_VAR

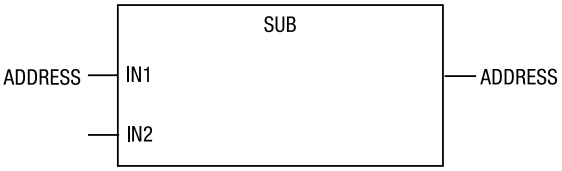
  LD Time1
  SUB Time_value
  ST New_time
          (* New_time: TOD#12:18:45 *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

PS4-200
PS4-300

SUB
Subtraction of addresses



Function prototype

The following data type combinations of the operands are possible:

Input operand IN1	Input operand IN2	Output operand
ADDRESS	ANY_INT	ADDRESS
ADDRESS	ADDRESS	ADDRESS

Description

The input variable is decreased and the result is loaded into the working register.

Loops or conditions for function blocks can be calculated with ADDRESS parameters, e.g. for SaveData or BlockTransfer. The address of a variable is loaded by writing the address operand & in front of the variable name.

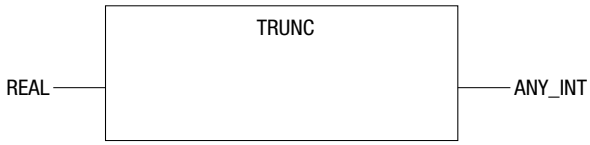
Example: subtraction of addresses

```
PROGRAM adr_sub
VAR
  Start_Data_Field AT %MB0.0.0.20 : BYTE ;
  Offset : UINT := 10 ;
  Copy : BlockTransfer ;
END_VAR

  LD &Start_Data_Field
  SUB Offset
  ST Copy.SourceAddress
END_PROGRAM
```

IEC **PS4-300**
PS416

TRUNC
Integer Part of a Floating Point Number



Function prototype

Description

The TRUNC function transfers the integer part of a floating-point number (REAL variable) to a signed integer variable.

The choice of data type for the result is dependent on the value of the integer part of the input operand.

Example: transfer integer part of number

The program integ_part transfers the integer part (up to 16 positions) of the variable Fl_Pt_Number to the variable Int_Value.

```
PROGRAM integ_part

VAR
  Fl_Pt_Number : REAL := 123.5;
  Int_Value : INT;
END_VAR

  LD   Fl_Pt_Number
  TRUNC
  ST   Int_Value          (* Int_Value: 123 *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

6 Function Blocks

Overview

Sucosoft provides a wide variety of function blocks for your use. They can be classified into:

IEC Function blocks in compliance with the IEC/EN 61131-3 standard,

Moeller standard function blocks.

The function blocks in compliance with the IEC/ EN 61131-3 standard are designed to cover all vendors. PLC programs which use only IEC function blocks can therefore be used with controllers made by different manufacturers.

Moeller has added further function blocks to the IEC functions. Apart from a few exceptions the Moeller function blocks can be used with all Moeller controllers.

The following table shows an overview of all function blocks grouped according to the jobs they do. Each function block is marked with 'IEC' or 'M' (in the table only "M") and the controller type specified where the function block can be used without any modification.

Some of the functions described used the data type ADDRESS. They are used for the automatic conversion of programs from previous systems and for this reason cannot be used with all controllers. Such function blocks are marked with # in the following table.
The use of ARRAYs means that you have functionally identical alternatives which can be used with all controllers. Please use these when you write new programs.

Function block	Meaning	PS4-200	PS4-300	PS416	Standard	See page
Edge detection						
F_TRIG	Edge evaluation, falling edge	✓	✓	✓	IEC	332
R_TRIG	Edge evaluation, rising edge	✓	✓	✓	IEC	423
Bistable function blocks						
RS	RS flip flop	✓	✓	✓	IEC	446
SR	SR flip flop	✓	✓	✓	IEC	521
Date and time						
DATconcat	Concatenate the data type DT	✓	✓	✓	M	291
DATsplit	Split the data type DT	✓	✓	✓	M	297
DateConcat	Concatenate the data type DATE	✓	✓	✓	M	293
DateSplit	Split the data type DATE	✓	✓	✓	M	295
GetRealTimeClock	Evaluation of the real-time clock	✓	✓	✓	M	342
RealTimeClock	Comparing the real-time clock	✓	✓	✓	M	438
RTC	Setting the real-time clock	✓	✓	✓	IEC	448
SetRealTimeClock*)	Setting the real-time clock	✓	✓	–	M	491
TimeConcat	Concatenate the data type TIME	✓	✓	✓	M	548
TimeSplit	Split the data type TIME	✓	✓	✓	M	561
TODconcat	Concatenate the data type TOD	✓	✓	✓	M	563
TODsplit	Split the data type TOD	✓	✓	✓	M	565
Timer function blocks						
MS_TimeFalling	Off-delay timer milliseconds	✓	✓	✓	M	361
MS_TimeRising	On-delay timer milliseconds	✓	✓	✓	M	364
S_TimeFalling	Off-delay timer seconds	✓	✓	✓	M	450
S_TimeRising	On-delay timer seconds	✓	✓	✓	M	453
TimeGenerator	Pulse generator	✓	✓	✓	M	550
TimePulse	Impulse timer	✓	✓	✓	M	552
TOF	Off-delay	✓	✓	✓	IEC	567
TON	On-delay	✓	✓	✓	IEC	569
TP	Impulse	✓	✓	✓	IEC	571

07/01 AWB2700-1306-GB

Function block	Meaning	PS4-200	PS4-300	PS416	Standard	See page
Alarm function blocks						
CounterAlarm	Counter alarm function block	✓	✓	✓	M	271
EdgeAlarm	Edge-controlled alarm function block	✓	✓	✓	M	326
TimerAlarm	Timer alarm function block	✓	✓	✓	M	554
Counter and comparison function blocks						
_16BitCounter	16-bit up/down counter	✓	✓	✓	M	232
_32BitCounter	32-bit up/down counter	–	✓	✓	M	238
CounterLE	Block for addressing the counter LE	✓	✓	–	M	277
CTD	Down counter	✓	✓	✓	IEC	283
CTU	Up counter	✓	✓	✓	IEC	284
CTUD	Up/down counter	✓	✓	✓	IEC	285
_16Bit_Compare	16-bit compare	✓	✓	✓	M	228
_32Bit_Compare	32-bit compare	–	✓	✓	M	234
SSIEncoder	Absolute encoder	–	✓	–	M	535
IncEncoder	Incremental encoder	–	✓	–	M	344
Register function blocks						
SR_x	1-Bit Shift Register	✓	✓	✓	M	523
SRB_x	8-Bit Shift Register	✓	✓	✓	M	527
SRW_x	16-Bit Shift Register	✓	✓	✓	M	531
LifoBx	8-bit FIFO register	✓	✓	✓	M	348
LifoWx	16-bit LIFO register	✓	✓	✓	M	352
FifoBx	8-bit FIFO register	✓	✓	✓	M	334
FifoWx	16-bit LIFO register	✓	✓	✓	M	338
Code converter						
_16BitBinaryToBCD	16-bit binary/decimal code converter	✓	✓	✓	M	230
_32BitBinaryToBCD	32-bit binary/decimal code converter	–	✓	✓	M	236
BCDTo16BitBinary	16-bit decimal/binary code converter	✓	✓	✓	M	245
BCDTo32BitBinary	32-bit decimal/binary code converter	–	✓	✓	M	247

Function block	Meaning	PS4-200	PS4-300	PS416	Standard	See page
Array operations						
BlockCompare*)	Data-block comparison or searching for values					
	Copy or initialise array	✓	✓	–	M	249
BlockTransfer*)	Data blocks					
	Comparison of arrays	✓	✓	–	M	254
CompareArray	Transfer of arrays	✓	✓	✓	M	265
TransferArray	Transfer data structures to an array	✓	✓	✓	M	573
Serialize	Transfer array to data structures	✓	✓	✓	M	485
Deserialize		✓	✓	✓	M	314
Character string processing						
ADRtoSTR	Copy marker area to string	✓	✓	–	M	240
STRtoADR	Copy string to a marker area	✓	✓	–	M	538
COMtoSTR	Copy received characters to string	✓	✓	✓	M	269
STRtoCOM	Copy string to marker area	✓	✓	✓	M	540
Sequential control function block						
SFC_x	Sequential control function block for sequence control	✓	✓	–	M	494
Communication function blocks						
SCO*)	Block for serial communication	✓	✓	–	M	459
COM	Serial communication	–	–	✓	M	264
DialOrHangup	Establish or clear a connection to a DTE via modem	–	✓	✓	M	317
Send ATCommand	Send AT commands to a modem	–	✓	✓	M	478
SUCOM_A	Emulation of programming protocol	–	✓	✓	M	542
MOD200	MODBUS communication	–	–	✓	M	360
ASI_PARAM	ASI slave parameter assignment	✓	✓	–	M	242
PSCO	PROFIBUS-FMS with LE4-503-BS1	✓	✓	–	M	416
SuconetP	PROFIBUS-FMS with PS416-NET-230	–	–	✓	M	546
SuconetS_BGKS	INTERBUS communication	–	–	✓	M	547
PdpStationDiag	PROFIBUS-DP communication	–	✓	✓	M	398
PdpFreezeSync	PROFIBUS-DP control commands	–	✓	✓	M	384
PdpAcyclicReadWrite	PROFIBUS-DPV1, acyclical communication	–	✓	✓	M	371
DE4netDP	DF4 frequency inverter	–	✓	✓	M	299
DE4netK	DF4 frequency inverter	✓	✓	✓	M	307
MI4netK	MI4 communication module	✓	✓	✓	M	358
MI4K	MI4 communication module	✓	✓	–	M	356

07/01 AWB2700-1306-GB

Function block	Meaning	PS4-200	PS4-300	PS416	Standard	See page
Reading and writing the memory card						
SetMCFileLength	Create and initialise data file on memory card					
	Load data from memory card	–	–	✓	M	488
ReloadData*)	Save data on memory card	✓	✓	–	M	443
SaveData*)	Read data from memory card	✓	✓	–	M	456
ReadMC	Write data to memory card	✓	✓	✓	M	433
WriteMC		✓	✓	✓	M	582
System functions (see also OS system functions in Table on page 129)						
ClearDiagnosticBits	Reset PS diagnostics bits	✓	✓	–	M	258
PS_Diagnostic	PS diagnostics words	–	–	✓	M	406
PS_Message	PS message words	–	–	✓	M	410
PS_SwitchPosition	PS switch positions	–	✓	✓	M	414
PS_GetDateAndTime	Access to the real-time clock	–	✓	✓	M	408
PS_ApplicationHalt	Halt the user program	–	✓	✓	M	402
PS_ReadOSInfo	Check the operating system version	✓	✓	✓	M	412
PS_ClearKOMBit	Acknowledge communication error	–	–	✓	M	404
CPUDataExchange	Data exchange between CPUs	–	–	✓	M	279
ColdstartRetention	Definition of a marker range which retains data even after cold start	–	–	✓	M	260
Peripheral access						
ReadDirect	Direct reading of digital inputs	–	–	✓	M	430
WriteDirect	Direct writing to digital outputs	–	–	✓	M	578
ReadAnalogDirect	Direct reading of analog inputs	–	–	✓	M	424
WriteAnalogDirect	Direct writing to analog outputs	–	–	✓	M	575
ReadCounterDirect	Direct reading of counter inputs	–	–	✓	M	427
Conversion functions						
DataScale	Scaling of data	✓	✓	✓	M	287
PTtoCelsius	Linearisation of PT1000 values	✓ 1)	–	–	M	419
PTtoFahrenheit	Linearisation of PT1000 values	✓ 1)	–	–	M	421
NItoCelsius	Linearisation of NI1000 values	✓ 1)	–	–	M	367
NItoFahrenheit	Linearisation of NI1000 values	✓ 1)	–	–	M	369
IEEE_to_Real	Number conversion	–	✓	✓	M	343
Real_to_IEEE	Number conversion	–	✓	✓	M	437

1) Only PS4-271

*) These function blocks operate in a marker range which must be defined under Generate → Program Parameters when generating the code.

A detailed description of the function blocks (in alphabetical order) starts on page 228.

The interfaces of the parameter transfer are described by means of the prototypes of the individual function blocks. A prototype provides an overview of the inputs and outputs of a function block. In IL representation the prototype is the declaration section of the function block with the declaration blocks contained in it.

For ease of understanding, the graphic prototype representation is used in the description of the function blocks. The function block is represented as a rectangular graphical symbol with input operands on the left and output operands on the right side. The name of the function block is shown in the middle of the graphical symbol. The data types of the inputs and outputs are shown outside the graphical symbol.

A continuous horizontal line indicates a passing parameter (VAR_IN_OUT).

The > symbol in front of an input operand indicates that the input requires a rising edge to execute the function.

The < symbol indicates that the input requires a falling edge to execute the function.

Taking the function block R_TRIG as an example the prototype representation of the function block is shown as an instruction list and also in graphic form.

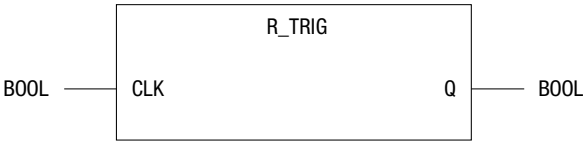
Example: prototype representation
Comparison of a function block prototype in IL and in graphical representation using the R_TRIG function.

```
FUNCTION_BLOCK R_TRIG

VAR_INPUT
  CLK : BOOL;
END_VAR

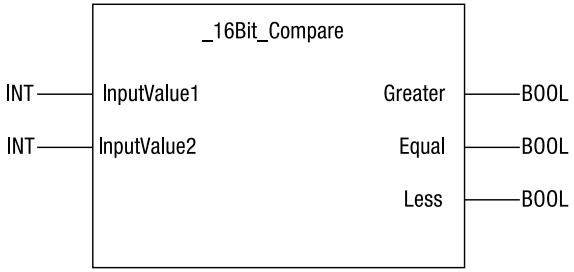
VAR_OUTPUT
  Q : BOOL;
END_VAR

END_FUNCTION_BLOCK
```



An instance is entered in the declaration section of the invoking POU in order to call a particular application of a function block. The function block invocation of the CAL command specifies the instance name of the function block concerned and transfers the parameters. More detailed information on function blocks and how to call them in the program is provided in Section “Function block POU” from page 9.

_16Bit_Compare
16-Bit Comparator



Function block prototype

Meaning of operands

InputValue1	First comparison value
InputValue2	Second comparison value
Greater	Result: first value greater
Equal	Result: both values equal
Less	Result: first value less

Description

The _16Bit_Compare function block compares InputValue1 with InputValue2 to see whether the value of the number InputValue1 is greater, equal or less than the value of the number InputValue2. The result of the comparison is indicated by the three output operands as follows:

	Less	Equal	Greater
InputValue1 < InputValue2	1	0	0
InputValue1 = InputValue2	0	1	0
InputValue1 > InputValue2	0	0	1

07/01 AWB2700-1306-GB

Example: 16-bit comparator

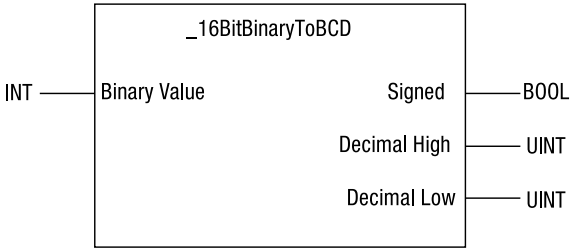
```
PROGRAM length

VAR
  Length_Compare : _16Bit_Compare;
  Length1 : INT := 10;
  Length2 : INT := 10;
  L1_equal_L2 : BOOL;
END_VAR

  CAL Length_Compare(InputValue1:=Length1,
                     InputValue2:=Length2)
  LD Length_Compare.Equal
  ST L1_equal_L2

END_PROGRAM
```

_16BitBinaryToBCD
16-Bit Binary/Decimal Code Converter



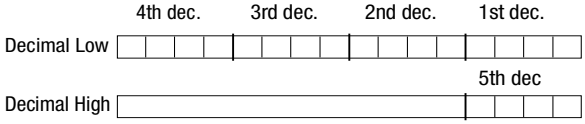
Function block prototype

Meaning of operands

BinaryValue	Binary-coded number
DecimalLow	Sign of BCD number, 0 for positive, 1 for negative number
DecimalHigh	5th decade of BCD number
DecimalLow	1st to 4th decade of BCD number

Description

A 16-bit binary number between –32768... +32767 at the BinaryValue operand is converted into a BCD value. The four least significant digits of the BCD value (1st – 4th decade) are written to the four nibbles of the 16-bit DecimalLow operand in the range. The most significant digit of the BCD value (5th decade) is written to the least significant nibble of the DecimalHigh operand. The sign of the BCD value is shown in the Boolean Signed operand.



07/01 AWB2700-1306-GB

Binary Value (hexadecimal)	Binary Value (decimal)	DecimalLow	DecimalHigh	DecimalLow
0000	0	0	0000	0000
0001	1	0	0000	0001
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
7FFF	32767	0	0003	2767
8000	–32768	1	0003	2768
8001	–32767	1	0003	2767
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
FFFF	–1	1	0000	0001

Example: 16-bit binary/decimal code converter

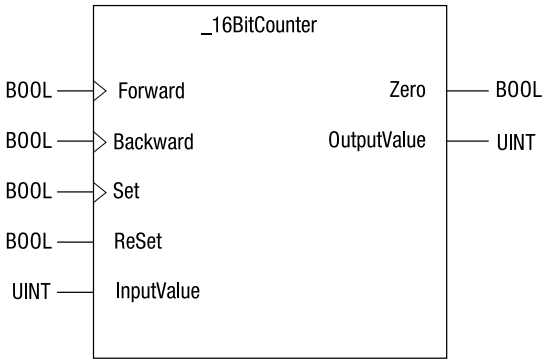
```
PROGRAM du16_bcd

VAR
  Converter1 : _16BitBinaryToBCD;
  BinaryNumber : INT;
  Sign : BOOL;
  HighNumber : UINT;
  LowNumber : UINT;
END_VAR

   .
   .
  CAL Converter1 (BinaryValue := BinaryNumber)
  LD   Converter1.Signed
  ST   Sign
  LD   Converter1.DecimalHigh
  ST   HighNumber
  LD   Converter1.DecimalLow
  ST   LowNumber
  .
   .


END_PROGRAM
```

_16BitCounter
16-Bit Up/Down Counter



Function block prototype

Meaning of operands

Forward	Up counter pulse, rising edge
Backward	Down counter pulse, rising edge
Set	Set condition, rising edge
ReSet	Reset condition
InputValue	Set value
Zero	Message: counter status = 0
OutputValue	Counter status

Description

The _16BitCounter function block is used for the upward and downward counting of signals. Each rising edge on the Forward input operand increments the counter by one. A rising edge on the Backward input operand decrements the counter by one. The current counter status is shown in the OutputValue operand. A rising edge on the Set input transfers the value of the InputValue to the counter. The counter is cleared when the ReSet input is 1, and other counting functions are disabled while the ReSet is 1.

07/01 AWB2700-1306-GB

The counter range is between 0 and 65535. The counter returns to 0 after it is incremented to 65535 and to 65535 after it is decremented to 0. A carry is not formed in either case.

If the counter is not instantiated as retentive, it is assigned the value 0 when it is initialised. The zero output operand will then be 1.A retentive instantiation of a counter retains its original value.

Example: 16-bit up counter

```
PROGRAM parts_16

VAR RETAIN
  Count_parts : _16BitCounter;
END_VAR

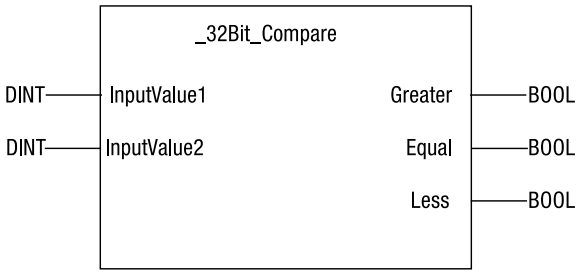
VAR
  Light_barrier AT %I0.0.0.0.0 : BOOL;
  Quantity : UINT;
  Clear AT %I0.0.0.0.1 : BOOL;
END_VAR

  CAL Count_parts(Forward := Light_barrier,
                          ReSet := Clear)
  LD Count_parts.OutputValue
  ST Quantity

END_PROGRAM
```

IEC **PS4-300**
PS416

_32Bit_Compare
32-Bit Comparator



Function block prototype

Meaning of operands

InputValue1	First comparison value
InputValue2	Second comparison value
Greater	Result: first value greater
Equal	Result: both values equal
Less	Result: first value less

Description

The _32Bit_Compare function block compares InputValue1 to InputValue2 to see whether the value of the number InputValue1 is greater, equal or less compared with the value of the number InputValue2. The result of the comparison is indicated by the three output operands as follows:

	Less	Equal	Greater
InputValue1< InputValue2	1	0	0
InputValue1 = InputValue2	0	1	0
InputValue1 > InputValue2	0	0	1

07/01 AWB2700-1306-GB

Example: 32-bit comparator

```
PROGRAM length

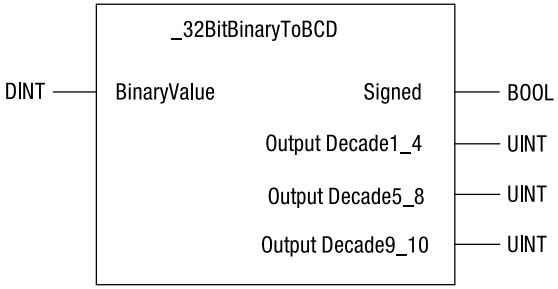
VAR
  Length_Compare : _32Bit_Compare;
  Length1 : DINT := 10;
  Length2 : DINT := 10;
  L1_equal_L2 : BOOL;
END_VAR

  CAL Length_Compare(InputValue1:=Length1,
                     InputValue2:=Length2)
  LD Length_Compare.Equal
  ST L1_equal_L2

END_PROGRAM
```

IEC **PS4-300**
PS416

_32BitBinaryToBCD
32-Bit Binary/Decimal Code Converter



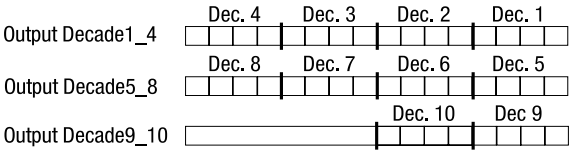
Function block prototype

Meaning of operands

BinaryValue	Binary-coded number
DecimalLow	Sign of BCD number, 0 for positive, 1 for negative number
OutputDecade1_4	1st to 4th decade of BCD number
OutputDecade5_8	5th to 8th decade of BCD number
OutputDecade9_10	9th to 10th decade of BCD number

Description

A 32-bit binary number between –2147483648... +2147483647 at the BinaryValue operand is converted to a BCD value. The individual digits of the BCD number are output in the nibbles of the operands OutputDecade1_4, OutputDecade 5_8 and OutputDecade 9_10. The sign of the BCD value is shown in the Boolean Signed operand.



07/01 AWB2700-1306-GB

Binary Value (hexadecimal)	Binary Value (decimal)	Decimal Low	OutputDecade1_4	OutputDecade5_8	OutputDecade9_10
00000000	0	0	0000	0000	0000
00000001	1	0	0000	0000	0001
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
7FFFFFFF	2147483647	0	0021	4748	3647
80000000	–2147483648	1	0021	4748	3648
80000001	–2147483647	1	0021	4748	3647
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
FFFFFFFF	–1	1	0000	0000	0001

Example: 32-bit binary/decimal code converter

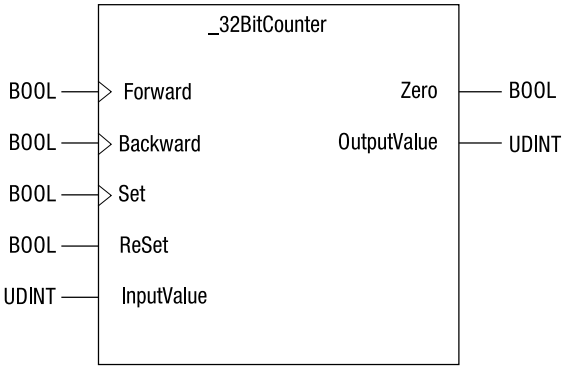
```
PROGRAM du32_bcd

VAR
  Converter2 : _32BitBinaryToBCD;
  BinaryNumber : DINT;
  Sign : BOOL;
  Decade1_4 : UINT;
  Decade5_8 : UINT;
  Decade9_10 : UINT;
END_VAR

  .
  .
  CAL  Converter2(BinaryValue := BinaryNumber)
  LD   Converter2.Signed
  ST   Sign
  LD   Converter2.OutputDecade1_4
  ST   Decade1_4
  LD   Converter2.OutputDecade5_8
  ST   Decade5_8
  LD   Converter2.OutputDecade9_10
  ST   Decade9_10
   .
   .
END_PROGRAM
```

IEC **PS4-300**
PS416

_32BitCounter
32-Bit Up/Down Counter



Function block prototype

Meaning of operands

Forward	Up counter pulse, rising edge
Backward	Down counter pulse, rising edge
Set	Set condition, rising edge
ReSet	Reset condition
InputValue	Set value
Zero	Message: counter status = 0
OutputValue	Counter status

Description

The _32BitCounter function block is used for the upward and downward counting of signals. Each rising edge on the Forward input operand increments the counter by one. A rising edge on the Backward input operand decrements the counter by one. The current counter status is shown in the OutputValue operand. A rising edge on the Set input transfers the value of the InputValue to the counter. The counter is

07/01 AWB2700-1306-GB

cleared when the ReSet input is 1, and other counting functions are disabled while the ReSet is 1.

The counter range is between 0 and 4294967295. The counter returns to 0 after it is incremented to 4294967295 and to 4294967295 after it is decremented to 0. A carry is not formed in either case.

If the counter is not instantiated as retentive, it is assigned the value 0 when it is initialised. The zero output operand will then be 1.
A retentive instantiation of a counter retains its original value.

Example: 32-bit up counter

```
PROGRAM parts_32

VAR RETAIN
  Count_parts : _32BitCounter;
END_VAR

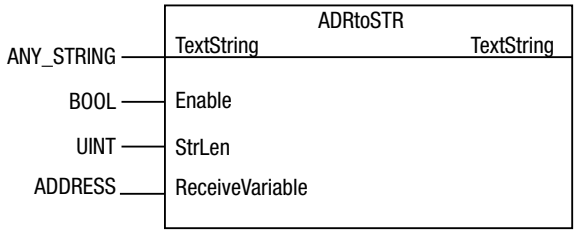
VAR
  Light_barrier : AT %IO.0.0.0.6 : BOOL;
  Quantity : UDINT;
  Clear : AT %IO.0.0.0.7 : BOOL;
END_VAR

  CAL Count_parts(Forward := Light_barrier,
                          ReSet := Clear)
  LD Count_parts.OutputValue
  ST Quantity

END_PROGRAM
```

PS4-200
PS4-300

ADRtoSTR
Convert Received Characters to String



Function block prototype

Meaning of operands

Enable	Number of converted characters
StrLen	Number of characters to be converted
ReceiveVariable	Start address of the marker area in which the data to be converted is stored
TextString	String created from the converted data in the marker area

Description

The function block is used to convert text data received via an interface and stored in a marker area to a variable of data type ANY_STRING.

A 1 or TRUE on the Enable input starts the conversion. The input parameter StrLen defines the number of characters to be converted and thus the length of the resulting string.

If the declared length of TextString is insufficient to hold all the converted data, characters are only copied into it until it is full.

07/01 AWB2700-1306-GB

Example: conversion of received characters into string

```
PROGRAM Str_Gen

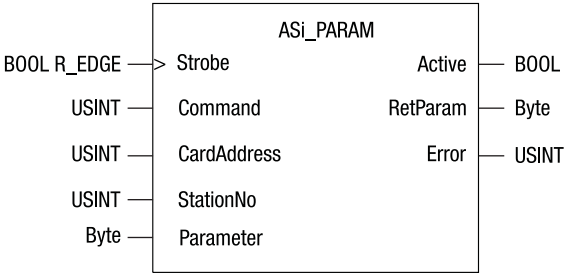
VAR
   Marker_to_String : ADRtoSTR;
   ReceiveText : STRING(4) := 'HALT';
   StartAddress AT %MB0.0.0.100 :
                    ARRAY[1..20] of BYTE;
   ReceiveLength : UINT := 4;
END_VAR

   CAL Marker_to_String(Enable := 1,
                        StrLen := ReceiveLength,
                        ReceiveVariable := &StartAddress,
                        TextString := ReceiveText)

END_PROGRAM
```

PS4-200
PS4-300

ASi_PARAM
Change Slave Parameters During Operation



Function block prototype

Meaning of operands

Strobe	Initiate an acyclical job (rising edge)
CardAddress	Module number of the LE4-505-BS1 (1 or 2)
Command	Command type
StationNo	Address of slave (1 – 31) to be assigned parameters
Parameter	Parameters for the slave
Active	Function block status
RetParam	Send or read parameters
Error	Result of job or detected errors

Description

The ASi-PARAM function block is used for changing slave parameters during operation. Only one function block needs to be active for each ASi LE.

A rising edge at the Strobe input causes the input parameters to be transferred. At the same time the Active output is set to 1 and keeps this state until the job is completed. If the Active output is 0, the values at the RetParam and Error outputs can be accepted.

07/01 AWB2700-1306-GB

The command type is input at the Command input:

02: Write parameters
03: Read parameters

Write parameters
When using the Write command the write address must be specified at the StationNo input and the parameter at the Parameter input. The parameter is in byte format. Only the 4 least significant bits are used. Each bit represents a function of the ASi slaves.

Correct command execution is indicated with Active = 0 and Error = 0, the slave has accepted the data and the sent parameters are shown at the RetParam output.

If the write command is not executed correctly, the Error output will send an error code (see table).

Read parameters
When using the Read command the read address must be specified at the StationNo input. Correct command execution is indicated with Active = 0 and Error = 0, the slave has accepted the data and the sent parameters are shown at the RetParam output.

If the command is not executed correctly, the Error output supplies the appropriate error code as a decimal value instead of 0 (see table).

Error code (dec.)	Meaning when write command incorrectly executed	Meaning when read command incorrectly executed
00	Everything OK	
02	Invalid station number (cause: number less than 1 or greater than 31)	
03	Existing station not configured	–
05	Station not responding (cause: line faulty or station missing)	–
08	–	Command not executable at present, e.g. due to absence of Asi voltage

Error code (dec.)	Meaning when write command incorrectly executed	Meaning when read command incorrectly executed
17	ASi master not present, not configured, incorrect module type or module type or CardAddress less than 1 or greater than 2	
35	Invalid command	
38	Command still active (the message only appears if a command has been started on a second function block for the same ASi-LE)	
39	Parameter is greater than 15	–

A more detailed description of this function block is given in the manual AWB 27-1314-GB “Hardware and Engineering, LE4-505-BS1 (ASi-LE)”.

Example “ASi_PARAM”

```
PROGRAM param

VAR
  ASIPAbau : ASi_PARAM ;
  Command1 : USINT : = 03 ;
  ASiadr : USINT : = 02 ;
  SLadr : USINT : = 01 ;
  Data : Byte : = 2 ;
  Pulse AT %IO.0.0.0.0 : BOOL ;
  Busy : BOOL ;
  Response : Byte ;
  Diagnostics : USINT ;
END_VAR

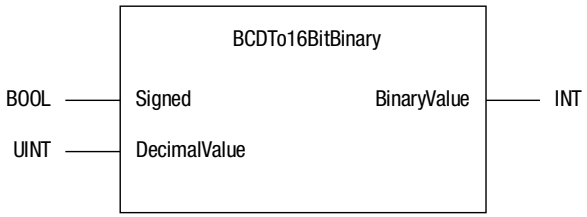
CAL ASIPAbau(Strobe : = Pulse,
             Command : = Command1,
             CardAddress : = ASiadr,
             StationNo : =SLadr,
             Parameter : = Data
             )
  Output*)

  LD ASIPAbau.Active
  ST Busy
  LD ASIPAbau.RetParam
  ST Response
  LD ASIPAbau.Error
  ST Diagnostics

END_PROGRAM
```


07/01 AWB2700-1306-GB

BCDTo16BitBinary
16-Bit Decimal/Binary Code Converter



Function block prototype

Meaning of operands

DecimalLow	Sign of the BCD number
DecimalValue	BCD number, 4 decades
BinaryValue	Binary-coded number

Description

A BCD number between –9999 .... +9999 in the 4 nibbles of the DecimalValue operand, specified as positive or negative by the Signed operand is converted to a binary number and output via the BinaryValue operand.

DecimalLow	Decimal Value	Binary Value (decimal)	Binary Value (hexadecimal)
0	0011	11	B
0	9999	9999	270F
1	–1311	–1311	FAE1
1	–9999	–9999	D8F1

Example: 16-bit decimal/binary code converter

```
PROGRAM bcd_du16

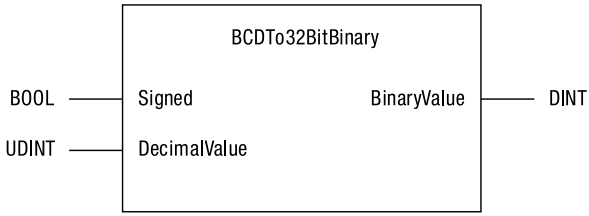
VAR
  Converter3 : BCDTo16BitBinary;
  DecimalNumber : UINT := 16#13;
  Sign : BOOL := 0;
  BinaryNumber : INT;
END_VAR

  .
  .
  CAL Converter3(Signed := Sign,
              DecimalValue := DecimalNumber)
  LD   Converter3.BinaryValue
  ST   BinaryNumber
  .
  .

END_PROGRAM
```


07/01 AWB2700-1306-GB

BCDTo32BitBinary
32-Bit Decimal/Binary Code Converter



Function block prototype

Meaning of operands

DecimalLow	Sign of the BCD number
DecimalValue	BCD number, 8 decades
BinaryValue	Binary-coded number

Description

A BCD number between 99999999.... +99999999 in the 8 digits of the DecimalValue operand, specified as positive or negative by the Signed operand is converted to a binary number and output via the BinaryValue operand.

Example: 32-bit decimal/binary code converter

```
PROGRAM bcd_du32

VAR
  Converter4 : BCDTo32BitBinary;
  DecimalNumber : UDINT := 13;
  Sign : BOOL := 0;
  BinaryNumber : DINT;
END_VAR

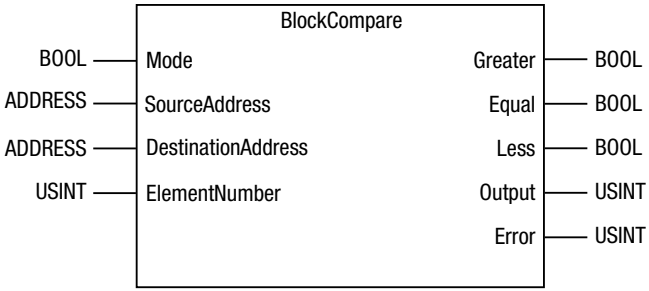
   .
   .
  CAL Converter4(Signed := Sign,
              DecimalValue := DecimalNumber)
  LD   Converter4.BinaryValue
  ST   BinaryNumber
  .
  .

END_PROGRAM
```


07/01 AWB2700-1306-GB

PS4-200
PS4-300

BlockCompare
Data Block Comparison or Value Search



Function block prototype

Meaning of operands

Mode	Operating mode (0 = search, 1 = block comparison)
SourceAddress	Start address of search range
DestinationAddress	Start address of destination range
ElementNumber	Number of elements to be compared
Greater	Greater indicator for comparison result
Equal	Equal indicator for comparison result
Less	Less indicator for comparison result
Output	Indicates where unequal data or the specified search value is located
Error	Error code

Description

The BlockCompare function block has two operating modes: Value Search and Block Comparison.

The mode is set with the Mode input.

Use CompareArray in new programs instead of BlockCompare.

Value Search mode
The Value Search mode is used to search for a specified value in a data block. The value to search for is specified in the marker entered under SourceAddress. The start of the data block to be searched is specified by DestinationAddress. The length of the data block to be searched for is specified by ElementNumber.

The Equal output will be set if the search criterion is found within the data range specified by SourceAddress and ElementNumber. Output indicates the position of this value within the data range:

Output = 0.. ElementNumber–1; Equal = 1;
The outputs Greater and Less are 0 in this case.

If the search criterion is not found, Output is set to the same value as ElementNumber. The Equal output is 0 and the outputs Greater and Less are set accordingly.

The Error output indicates the following error codes:

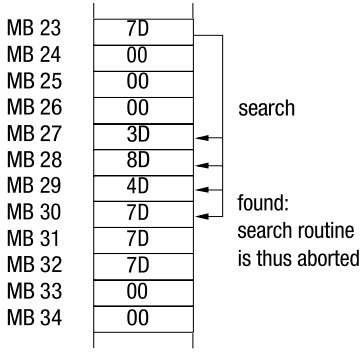
0 Data limits permissible
1 Number of elements (ElementNumber) is 0
2 Invalid source address at SourceAddress
3 Invalid destination address at DestinationAddress
4 Source address is same as destination address

07/01 AWB2700-1306-GB

Example: searching for data value

The following example searches in the marker area from %MB0.0.0.27 to %MB0.0.0.32 for the value 7D stored in %MB0.0.0.23.

Marker field:




```
PROGRAM search
VAR
  search_fb: BlockCompare;
  SourceAddress AT %MB0.0.0.23 : BYTE := 16#7D;
       (* Marker byte 23 initialised with the
       search value 7D hex *)
  DestinationAddress AT %MB0.0.0.27 : BYTE;
  Start AT %I0.0.0.0.0 : BOOL;
       (* Start search *)
  Found : USINT;
  Result_greater : BOOL;
  Result_equal : BOOL;
  Result_less : BOOL;
END_VAR

  LD Start
  CALC search_fb(Mode := 0,(*Data search mode*)
              SourceAddress := &SourceAddress,
              DestinationAddress :=
              &DestinationAddress,
              ElementNumber := 6)
              (*Search range of 6 bytes*)
  LD search_fb.Output
  ST Found
```

```
  (* Location where the search value is found *)
  (* "DestinationAddress" plus Output *)
  LD search_fb.Greater
  ST Result_greater
  LD search_fb.Equal
  ST Result_equal
  LD search_fb.Less
  ST Result_less
END_PROGRAM
```

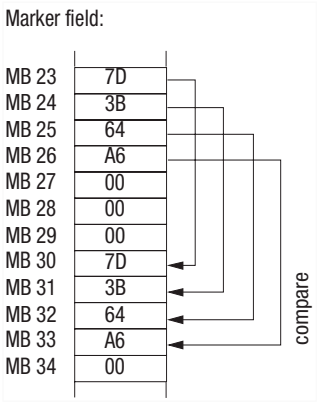
Block Comparison mode

The Block Comparison mode is used to compare two data blocks. The start of the two data blocks is specified with SourceAddress and DestinationAddress. The size of the data blocks is specified with ElementNumber.

If both data blocks are found to be identical, Output will be set to the same value as ElementNumber and the Equal output will be set to 1.

If the data blocks are not the same, Output will indicate the location where there is a difference. The Equal output will be set to 0 and the Greater or Less outputs will be set according to the result of comparison at the location where the discrepancy was found.

Example of block comparison



07/01 AWB2700-1306-GB

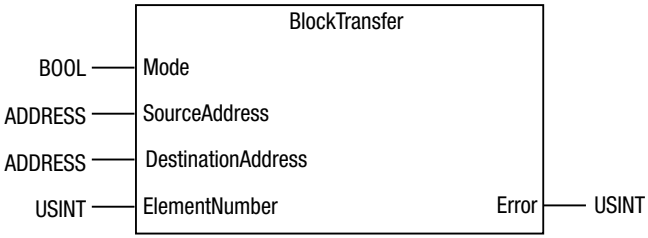
```
PROGRAM compare
VAR
   compare_fb : BlockCompare;
   Source_Address AT %MB0.0.0.23 : BYTE;
   Destination_Address AT %MB0.0.0.30 : BYTE;
   Start AT %I0.0.0.0.0 : BOOL;
       (*Start the comparison*)
   Deviation : USINT;
   Result_greater : BOOL;
   Result_equal : BOOL;
   Result_less : BOOL;
END_VAR

   LD Start
   CALC compare_fb(Mode := 1,(*Block comparison*)
                   SourceAddress := &Source_Address,
                   DestinationAddress := &Destination_Address,
                   ElementNumber := 4)
                   (*Comparison range of 4 Bytes *)
   LD compare_fb.Output
   ST Deviation
   (*Location at which the blocks are different*)
   (* the: DestinationAddress plus Output *)
   LD compare_fb.Greater
   ST Result_greater
   LD compare_fb.Equal
   ST Result_equal
   LD compare_fb.Less
   ST Result_less

END_PROGRAM
```

PS4-200
PS4-300

BlockTransfer
Copy or Initialise Data Blocks



Function block prototype

Meaning of operands

Mode	Operating mode (0 = Initialise, 1 = Copy)
SourceAddress	Start address of search range
DestinationAddress	Start address of destination range
ElementNumber	Number of elements to be compared
Error	Error code

Description

The BlockTransfer function block allows system support for transferring data blocks. Data is transferred from the source address defined at the SourceAddress input to the destination in the PLC memory defined at the DestinationAddress input.

The Mode parameter specifies the required operating mode: Copy mode or Initialise mode.

The size of the data block is defined in byte at the ElementNumber input.

The Error output indicates the following error codes:

0 Data limits permissible
1 Number of elements (ElementNumber) is 0
2 Invalid source address at SourceAddress

07/01 AWB2700-1306-GB

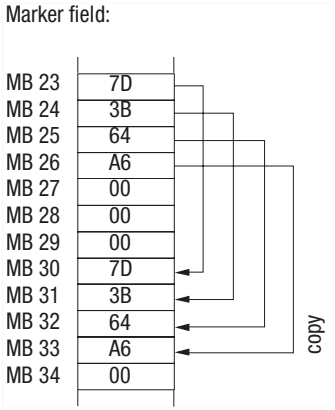
3 Invalid destination address at DestinationAddress
4 Source address is same as destination address



Use TransferArray in new programs instead of BlockTransfer.

Copy mode
In Copy mode the function block copies a complete data range from a source address to a destination address.

Example of Copy Mode
The contents of the data block marker bytes 23 to 26 are copied to marker area 30 to 33.




```
PROGRAM copy

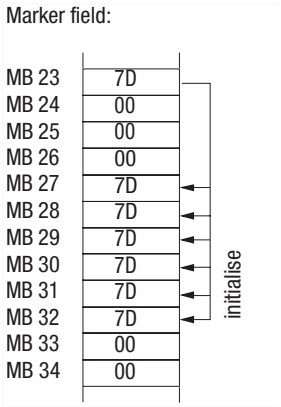
VAR
  copy_fb : BlockTransfer;
  SourceAddress AT %MB0.0.0.23 : BYTE;
  DestinationAddress AT %MB0.0.0.30 : BYTE;
  Start AT %I0.0.0.0.0 : BOOL;
       (*Start copying*)
END_VAR
```

```
  LD Start
  CALC copy_fb (Mode := 1,(* Copy Mode ! *)
              SourceAddress := &SourceAddress,
              DestinationAddress
              := &DestinationAddress,
              ElementNumber := 4)
              (*Constant number of 4 bytes *)

END_PROGRAM
```

Initialise mode
In Initialise mode the function block copies a specified value from the source address to a data block starting with the destination address.

Example of Initialise Mode
The value of the marker byte specified in SourceAddress is transferred into the number of marker bytes specified in ElementNumber. The copy operation starts from DestinationAddress. The marker field %MB0.0.0.27 to %MB0.0.0.32 is initialised with the data value 7D stored in %MB0.0.0.23.



07/01 AWB2700-1306-GB

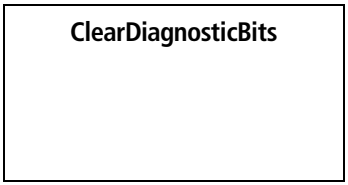
```
PROGRAM init

VAR
   init_fb : BlockTransfer;
   SourceAddress AT %MB0.0.0.23 : BYTE := 16#7D;
    (* MarkerByte 23 initialised with the data value 7D hex *)
   DestinationAddress AT %MB0.0.0.27 : BYTE;
   Start AT %I0.0.0.0.0 : BOOL;
    (*Start initialising*)
END_VAR

   LD Start
   CALC init_fb(Mode := 0, (* Initialise Mode *)
               SourceAddress := &SourceAddress,
               DestinationAddress := &DestinationAddress,
               ElementNumber := 6)
               (*Constant number of 6 bytes *)

END_PROGRAM
```

PS4-200
PS4-300
PS416



Function block prototype

Description

This function block is used to reset active diagnostic bits while the program is running. The operating system will check the diagnostics bits again in the next cycle and those bits will be set again that fulfill the set condition. In this case, the diagnostics counter will be incremented by 1. The diagnostics bits checked by the function block are listed below by PLC.

Indication diagnostics bits in the PS4-200 and PS4-300:

DAC	Power supply failure
DBM	Battery low or faulty
DMC	Memory module cannot be used for backup
DLK	Configuration error
DLS	Input/output error
DDK	Configuration error in remote expansion
DDS	Status error in remote expansion

07/01 AWB2700-1306-GB

Indication diagnostics bits in the PS416:

DAC	Power supply failure
DBM	Failure of the CPU battery
DBC	Failure of the MC battery
DAK	No card detected in slot
KOM	Communication partner failed

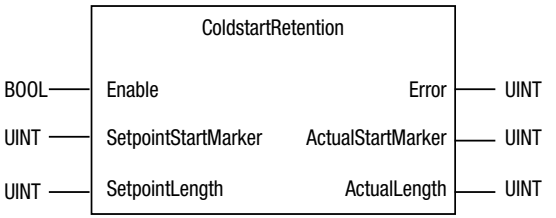
```
PROGRAM TestOfClearDiagnosticBits

VAR
  FBClear:ClearDiagnosticBits;
  Status AT %ISB0.0.0.0:BYTE;
END_VAR

ld     Status
eq     0
jmpc   _over
CAL    FBClear
_over:
END_PROGRAM
```

PS416

ColdstartRetention
Definition of a Marker Range with Cold Start Retention



Function block prototype

Meaning of operands

Enable	A rising edge enables the definition of the cold start retentive marker area
SetpointStartMarker	Starting byte address of marker area
SetpointLength	Number of marker bytes
Error	Error message 0 Last call without errors 1 Starting byte address too large or defined marker area exceeds MB4345 2 Operating system version older than V1.09
ActualStartMarker	Current starting byte address of cold start retentive marker area; only relevant if ActualLength > 0
ActualLength	Current size of the cold start retentive marker area (in bytes)

Description

The ColdstartRetention function block defines a block of marker bytes in the PS416-CPU (MB 0 – MB 4345) marker range which is retentive in the event of a cold start of the PLC, i.e. its data is retained instead of being initialised when the program is cold started. This allows several user programs to share the same data.

07/01 AWB2700-1306-GB

Please note the following constraints when defining the cold start retentive marker area.

- The OS40 operating system version must be V 1.09 or later.
- All markers which should also be retentive following a warm start must be declared in the program with the RETAIN keyword; according to requirements this also applies to markers defined with this function block as cold start retentive.
- The ColdstartRetention function block must be called in the first program cycle following a cold start.

The ColdstartRetention function block does not carry out management of the cold start retentive marker area.

After loading the operating system, the length of the cold start retentive marker area is zero until the ColdstartRetention function block has been called for the first time. In this case, all variables are initialised with the values specified in the program or with default values.

The cold start retentive marker area remains valid until it is redefined by calling the function block again, even if other user programs are started which do not use the defined marker area.

The definition of the cold start retentive marker area can be reset by loading the OS40 operating system into the PLC again or booting from a memory card.

The current definition of the cold start retentive marker area can be checked by calling the function block with the Enable input set to 0 or FALSE.

Example: cold start retention
Invocation in the program

```
PROGRAM Retention

VAR RETAIN

   (* Declaration of warm start retentive markers *)
   M10 AT%MB0.0.2.10 : USINT := 2;
   M12 AT%MB0.0.2.12 : USINT := 2;
   M14 AT%MB0.0.2.14 : USINT := 2;

   (* Declaration of cold and warm start retentive markers *)
   M20 AT%MB0.0.2.20 : USINT := 3;
   M22 AT%MB0.0.2.22 : USINT := 3;
   M24 AT%MB0.0.2.24 : USINT := 3;

END_VAR

VAR
   SetValues AT%i0.0.0.0.0 : BOOL ;
   RetentiveError AT%q0.0.0.0.0 : BOOL ;
   Retentive : ColdstartRetention ;
   FirstCycle : PS_Message ;

END_VAR

   (* The cold start retentive marker area must be defined
      in the first program cycle following a cold start.*)
   CAL   FirstCycle( MessageType := ICS )
   LD    FirstCycle.Result
   JMPCN NORMAL_CYCLE

   (* An existing cold start retentive marker area should not be changed
      Accordingly a check is made for a previously defined marker area.*)
   CAL   Retentive( Enable := FALSE )
   LD    Retentive.Error
   EQ    0
   JMPCN ERROR
   LD    Retention.ActualLength
   GT    0
   JMPC NORMAL_CYCLE
```

```
      (* If the length of a previously defined cold start retentive marker area is zero,
define a range for cold start retentive markers as follows.
      Start of range : MB 20; Length of range : 6 Byte *)
   CAL Retention(Enable := TRUE,
                 SetpointStartMarker := 20,
                 SetpointLength := 6
                 |
                 :=Error,
                 :=ActualStartMarker,
                 :=ActualLength )

   LD    Retention.Error
   EQ    0
   JMPCN ERROR
   JMP   NORMAL_CYCLE

ERROR:
   (* Execute error handling *)
   LD    1
   ST    RetentionError

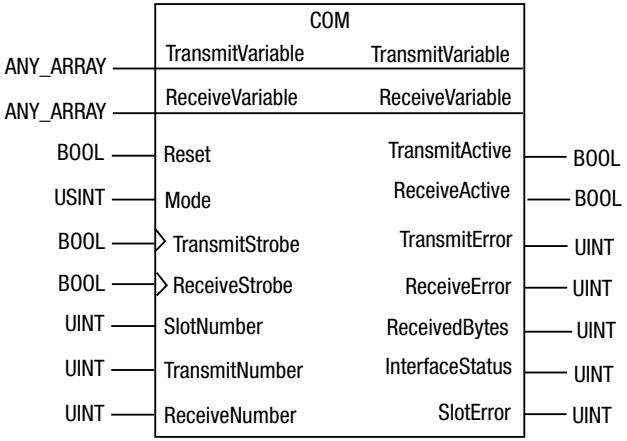
NORMAL_CYCLE:
   (* This range is only used to show the different
    retentive behaviour of the declared markers
    These markers are retentive when the user program
   is warm started *)
   LD    M10
   LD    M12
   LD    M14

   (* The following markers are cold and warm start retentive
         with a user program *)
   LD    M20
   LD    M22
   LD    M24
   LD    SetValues
   JMPCN END
   LD    44
   ST    M10
   ST    M12
   ST    M14
   ST    M20
   ST    M22
   ST    M24

END:
END_PROGRAM
```

PS416

COM
Serial Communication Function Block



Function block prototype

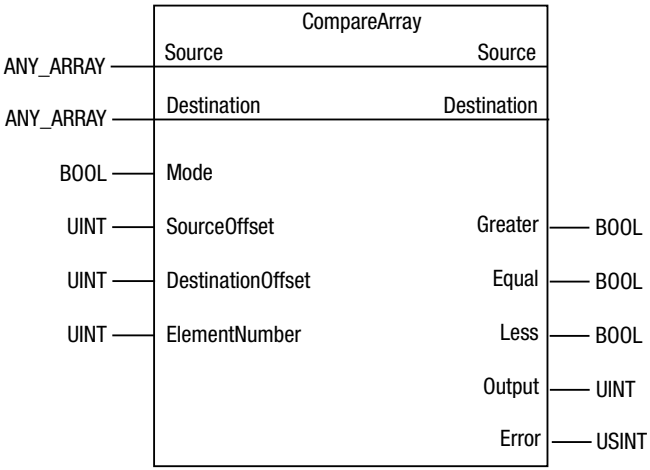
Description

The COM function block is used for handling the serial communication via the PS416-COM-200 card.

A detailed description and explanation of how to use the function block is provided in the manual Hardware and Engineering PS416-COM-200 – Communication Card, AWB27-1237-GB.

07/01 AWB2700-1306-GB

CompareArray
Data Block Comparison or Value Search



Function block prototype

Meaning of operands

Mode	Operating mode (0 = search, 1 = block comparison)
Source	Source area
Destination	Destination area
SourceOffset	Offset in Source area
DestinationOffset	Offset in Destination area
ElementNumber	Number of elements to be compared
Greater	Greater indicator for comparison result
Equal	Equal indicator for comparison result
Less	Less indicator for comparison result
Output	Indicates where unequal data or the specified search value is located
Error	Error code

Description
The CompareArray function block has two operating modes: Value Search and Block Comparison.

The mode is set with the Mode input.

The source and the destination can be of any required size. All types of type ANY_INT and ANY_BIT are permissible as array element types apart from BOOL.

The Error output indicates the following error codes:

0 No error
1 Number of elements (ElementNumber) is 0
2 Source offset is outside the source area.
3 Number of elements exceeds the source area.
4 Destination offset is outside the destination area
5 Number of elements exceeds the destination area.

Value Search mode
The Value Search mode is used to search for a specified value in an array. The value to search for is contained in the array specified with Source and SourceOffset.

The start of the array to be searched for is specified with Destination and DestinationOffset.

The length of the data block to be searched is specified with ElementNumber. The Equal output will be set if the search criterion is found within the data block defined by the destination address and ElementNumber.

Output indicates the position of this value within the array:

Output = 0.. ElementNumber–1; Equal = 1
The outputs Greater and Less are 0 in this case.

07/01 AWB2700-1306-GB

If the search criterion is not found, Output is set to the same value as ElementNumber. and the Equal, Greater and Less are set to 0.

Example: search for data

```
PROGRAM search
VAR
   search_fb : CompareArray;
   SourceAddress AT %MB0.0.0.23 : ARRAY [1..1] OF BYTE := [16#7D];
       (* MarkerByte 23 initialised with search value 7D hex*)
   DestinationAddress AT %MB0.0.0.27 : ARRAY [1..1] OF BYTE;
   Start AT %I0.0.0.0.0 : BOOL;          (* Start search operation *)
   Found : USINT;
   Result_greater : BOOL;
   Result_equal : BOOL;
   Result_less : BOOL;
END_VAR

   LD Start
   CALC search_fb (Mode := 0, (* Data search mode *)
               Source := SourceAddress,
               Destination := DestinationAddress,
               ElementNumber := 6)(* Search range of 6 bytes *)

   LD search_fb.Output
   ST Found                  (* Location of the search value *)
                             (* "Destination" plus "Output" *)
   LD search_fb.Greater
   ST Result_greater
   LD search_fb.Equal
   ST Result_equal
   LD search_fb.Less
   ST Result_less

END_PROGRAM
```

Block Comparison mode

The Block Comparison mode is used to compare two data blocks which can be parts of arrays or entire arrays.

The start of the two data blocks is specified with Source and SourceOffset and/or Destination and DestinationOffset respectively. The size of the data blocks is specified with ElementNumber.

If both data blocks are found to be identical, Output will be set to the same value as ElementNumber and the Equal output will be set to 1.

If the data blocks are not the same, Output will indicate the location in the array where there is a difference. The Equal output will be set to 0 and the Greater or Less outputs will be set according to the result of comparison at the location where the discrepancy was found.

Example: comparing data

```
PROGRAM compare
VAR
   compare_fb : CompareArray;
   SourceAddress AT %MB0.0.0.23 : ARRAY [1..7] OF BYTE := [16#7D];
   DestinationAddress AT %MB0.0.0.30 : ARRAY [1..7] OF BYTE := [16#7D];
   Start AT %I0.0.0.0.0 : BOOL;          (* Start compare operation *)
   Deviation : USINT;
   Result_greater : BOOL;
   Result_equal : BOOL;
   Result_less : BOOL;
END_VAR

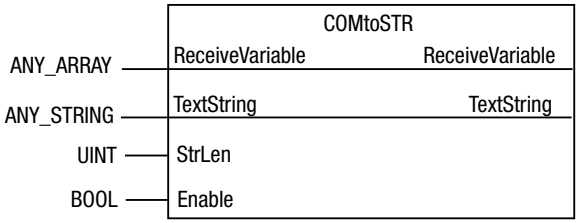
   LD Start
   CALC compare_fb(Mode := 1,            (* Block comparison *)
                   Source := SourceAddress,
                   Destination := DestinationAddress,
                   ElementNumber := 4)  (* Comparison range: 4 bytes *)

   LD compare_fb.Output
   ST Deviation     (* Location at which the blocks differ*)
                    (* : Destination plus Output *)
   LD compare_fb.Greater
   ST Result_greater
   LD compare_fb.Equal
   ST Result_equal
   LD compare_fb.Less
   ST Result_Less

END_PROGRAM
```


07/01 AWB2700-1306-GB

COMtoSTR
Copy Received Characters to String



Function block prototype

Meaning of operands

ReceiveVariable	Start address of the marker range where the data to be converted is located
TextString	String to contain the converted data from the array
StrLen	Number of converted characters
Enable	Enable the conversion

Description

The function block is used to convert text data received via an interface and stored in an array to a variable of data type ANY_STRING.

A 1 or TRUE on the Enable input starts the conversion. The input parameter StrLen defines the number of characters to be converted and thus the length of the resulting string.

If the declared length of string variables is insufficient to hold all the data from the array, only the number of characters are copied that the declared variable can take. The data is copied to the string 1:1 without ASCII conversion.

Example: converting received characters into a string

```
PROGRAM Str_Gen

VAR
   EnableReceive AT%I0.0.0.0.0 : BOOL;
   ArrayAfterString : COMtoSTR;
   ReceiveText : STRING(20):
   ReceiveArray : ARRAY[1..20] of Byte;
   ReceiveLength : UINT;
   ReceiveFromSerial : COM ;
END_VAR

       (* Receive data from serial interface *)
   LD      EnableReceive
   ANDN    ReceiveFromSerial.ReceiveActive
   ST      ReceiveFromSerial.ReceiveStrobe
   CAL     ReceiveFromSerial(Mode := 1;
                         ReceiveStrobe := ;
                         SlotNumber := 6; (* COM200 on slot 6 *)
                         ReceiveNumber := 20 ;
                         ReceiveVariable := ReceiveArray;

                         ReceiveActive := ;
                         ReceivedBytes := ;
                         ReceiveError := ;
                         InterfaceStatus := ;
                         SlotError := )

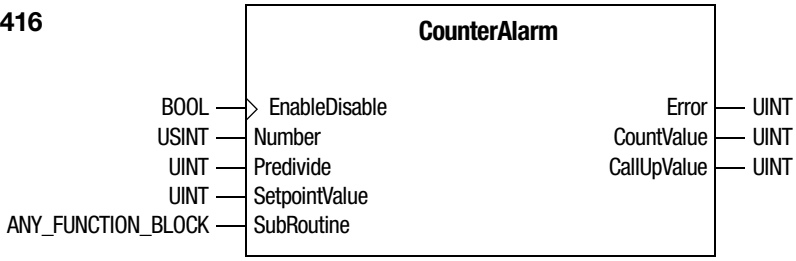
    (* Copy string to array *)
   LD      EnableReceive
   ANDN    ReceiveFromSerial.ReceiveActive
   ST      Array_to_String.Enable
   CAL     Array_to_String (Enable := ;
                    StrLen := ReceiveFromSerial.ReceivedBytes;
                    TextString := ReceiveText,
                    ReceiveVariable := ReceiveArray)

END_PROGRAM
```


07/01 AWB2700-1306-GB

PS4-200[1)]
PS4-300
PS416

CounterAlarm
Counter Alarm Function Block



Function block prototype

Meaning of operands

EnableDisable	Enable and disable alarm execution Start condition rising edge (0 = Disable, 1 = Enable)
Number	Alarm channel number (permissible value for all controllers = 0)
Predivide	Predivider for the counter
SetpointValue	Setpoint value for the counter (permissible value range: 1 to 65535)
SubRoutine	Instance name of the function block called by the alarm
Error	Error code
CountValue	Counter pulse actual value
CallUpNumber	Number of alarm-controlled function block invocations

1) Not for PS4-271

Description

The CounterAlarm function block is used to detect and count impulses at the specified input.

Counter inputs:
In the basic units of the PS4-150, PS4-200 or PS4-300 and the basic rack of the PS416 the CounterAlarm function block is assigned the hardware input %I0.0.0.0.0.

The EnableDisable input controls the start of the counter. The counter is started if there is a positive edge at this input. It is stopped and reset if the input status changes from 1 or TRUE to 0 or FALSE.

The current number of count pulses of the hardware counter are indicated at the CountValue output.

The input Predivide (value range: 1 – 65535) is used to define the number of signals required at the hardware input to increment the counter. The SetPointValue (value range: 1 – 65535) input defines the number of counted signals required to activate the alarm or call the function block.

The maximum permissible load on the system depends on the following three factors:
Predivide, SetPointValue the specified frequency.
These factors must be selected so that no more than two interrupts occur per millisecond.

Maximum counter frequency:

PLC type	Max. counter frequency/Khz
PS4-150	3
PS4-200	3
PS4-300	50
PS 416	0.5

07/01 AWB2700-1306-GB

The Predivide > 1 setting ensures the counter frequency of 3 kHz on the PS4-200 and 50 kHz on the PS4-300 irrespective of the PLC load (Suconet/ PC communication).

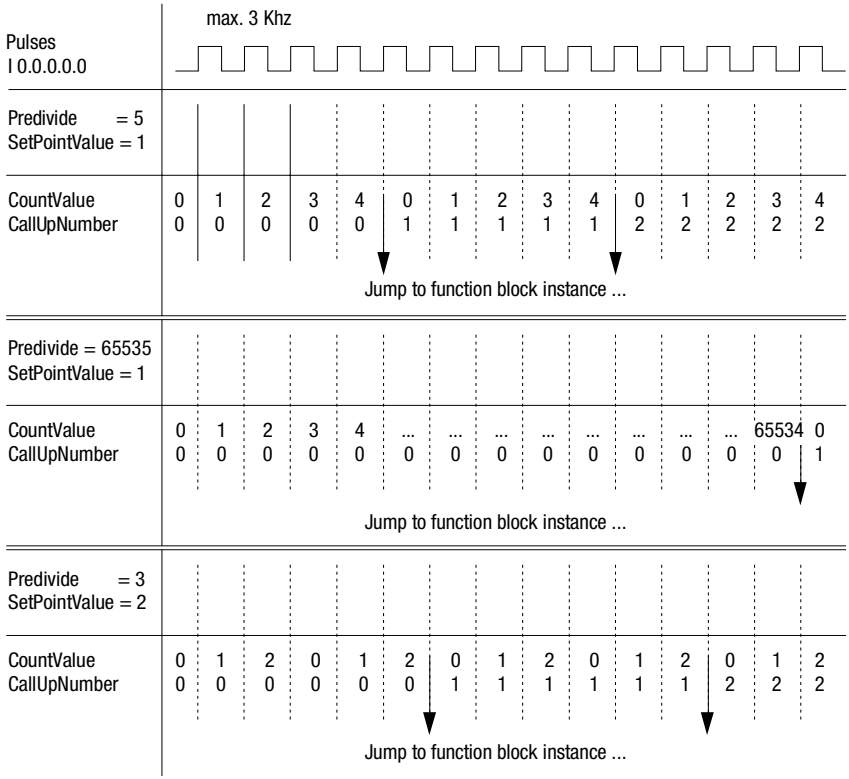
Counter outputs:
The Error output can indicate one of the following codes:

Signal	PS4-150	PS4-200	PS4-300	PS416
0	No error, activation successful			
1	Setpoint value SetPointValue = 0			
2	Setpoint value Predivide = 0			
3	–	–	Invalid alarm channel number	
4	–	–	Counter alarm already in use.	
5	–	–	Counter channel not set as up/down counter	–

The CallUpNumber output (value range: 1 – 65535) indicates how often the value defined at the SetpointValue input has been reached.

The SubRoutine input allows event-driven programming to be implemented. This is done by stating the instance name of the function block to be activated when the event is reached. If no instance is specified, the CountValue will continue to be incremented without triggering the alarm.

The function block called by the alarm function block instance must not have any VAR_IN_OUT parameters. This kind of instance must only have local variables (VAR) or global variables that have been declared externally as VAR_EXTERNAL variables.



Observe the following notes for event-driven program processing:

After the event has occurred, the user program is interrupted, the register status is saved and the subprogram specified by Subroutine is executed.

The alarm function block cannot be interrupted by other alarm function blocks (all alarm function blocks have the same priority). The maximum execution time of alarm blocks must be restricted to 5 ms (approx. 1K IL instructions) in the user program since the alarm function blocks cannot be interrupted even by the operating system in the event of a voltage

07/01 AWB2700-1306-GB

drop.(approx. 1 k IL instructions). If the execution time is exceeded, an EDC error may occur when switching off the power supply.

The execution time of the alarm function block is added to the execution time of the cyclical user program and also monitored by the cycle time monitoring function.

Since the event-driven program processing enables access to the entire image register, access to data that is used by the event-driven and cyclical user program must be disabled. Bit accesses may only occur on the same byte in the cyclical user program and in the Alarm function block if the access is synchronized with Disable_Interrupt and Enable_Interrupt.

Since, due to its fast reaction times, an Alarm function block requires a high-speed peripheral access (direct output), the QP, QPB peripheral operands available in the basic unit should be used.

An alarm function block can be used several times (multiple instantiation) although this should normally be avoided since each function block group has the same event source and only the last function block instance in the program is valid. Multiple instantiation is only useful if the parameters of the function block are set again within an interrupt routine and an edge disable is to be implemented.

By multiple instantiation is meant the reservation of several data ranges for each parameter set of a particular function block type.



The input parameters of an alarm function block should only be assigned between the parentheses when it is called. An explicit parameter assignment outside of the parentheses before or after the invocation is not possible.

Example: counter alarm function block
The pulses of a rotary encoder are to be counted. The time between the pulses is less than the cycle time of the PLC. The encoder supplies 1000 pulses per degree of rotation. The scaling ratio should be set to 100 on the function block so that the position of rotation is provided at the CNT output in 1/10 degree of rotation.

```
PROGRAM calarm

VAR
  CountAlarm : CounterAlarm;
  Error : USINT;
  CountValue : UINT;
END_VAR

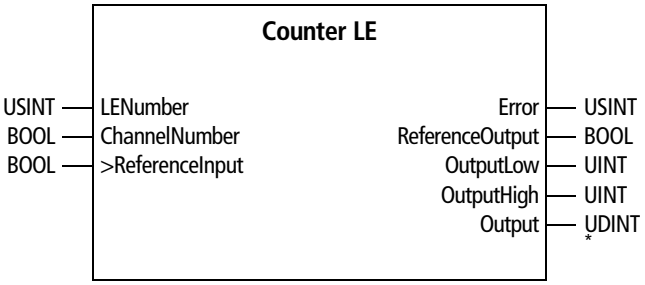
  CAL CountAlarm(EnableDisable := 1,
                 Number := 0,
                 Predivide := 100,
                 SetpointValue := 1)

  LD CountAlarm.Error
  ST Error
  LD CountAlarm.CountValue
  ST CountValue
   (*This shows the current counter value *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

CounterLE
Activation of the Local Expansion Module



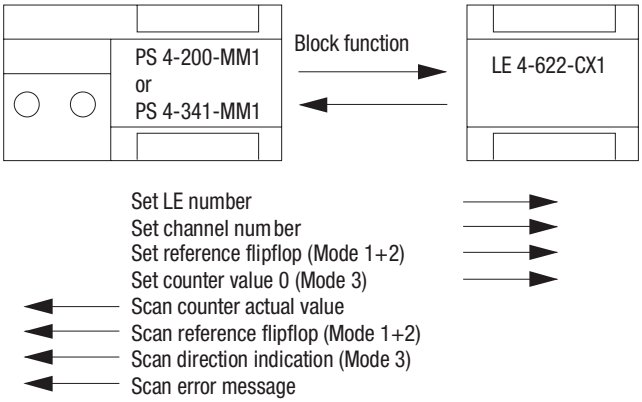
Function block prototype (PS4-300 only)*

Meaning of operands

LENumber	Module address of LE (can only be 1 or 2)
ChannelNumber	Channel number (0 = Channel 1; 1 = Channel 2)
ReferenceInput	Edge-controlled 0 -> 1: Set reference flip flop (Mode 1 + 2) or set counter actual value to 0 (Mode 3)
Error	Error code
ReferenceOutput	0 = Reference flip flop reset (Modes 1 + 2) 1 = Reference flipflop set (Modes 1 + 2) Direction indication (Mode 3); this shows the status of the LE input Y
OutputLOW	Current counter value (lower word)
OutputHigh	Current counter value (higher word)
Output	Current counter value (only for PS4-300)

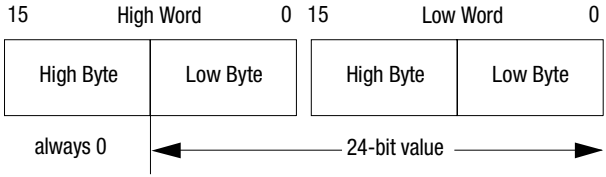
Description

The CounterLE function block controls the LE4-622-CX1. The following functions are possible:



A function block instance is assigned to every channel of the LE. The function block instances are not retentive, i.e. the counter actual values are lost in the event of a voltage drop.

The LENumber and ChannelNumber inputs are used to define the counter channel whose current counter value can be called using the OutputLOW and OutputHigh parameters below:



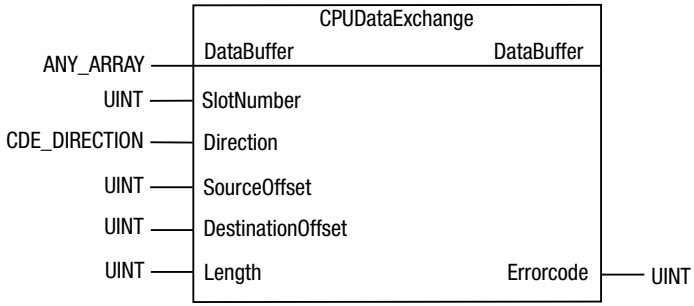
Structure of the 24-bit counter value

The counter status is refreshed every time the CounterLE function block is processed.

Further information on the operation and special features of the LE4-622-CX1 are provided in the manual AWB2700-1324-GB.

07/01 AWB2700-1306-GB

CPUDataExchange
Data Exchange between PLCs



Function block prototype

Meaning of operands

SlotNumber	Slot number of the CPU which exchanges data
Direction	Data direction CD_READ: From the master CPU to the slave CPU CD_WRITE: From the slave CPU to the master CPU
SourceOffset	Offset of memory location, first byte to read
DestinationOffset	Offset of memory location, first byte to write
Length	Number of bytes to read or write
DataBuffer	Array variable for exchanging data with the CPU
Errorcode	Error code

Description

When using several PS416-CPU-200/-300/-400 PLCs in a rack, the CPUDataExchange function block can be used to exchange data between PLCs. The same function block is used for both master and slave CPUs.

The following PS416 system configuration must be ensured:

Slot 2: MASTER CPU; PS416-CPU
Slot 4/6/8/...: SLAVE CPUs; PS416-CPU

Data exchange takes place using a special memory chip of size 2032 bytes. Read or write operations can be made to this memory area with a function block call.

Information for configuring and programming the function block is provided in the manual Sucosoft S40, User Interface, AWB2700-1305-GB.

SlotNumber specifies the slot number of the CPU which receives the data or from which the data is read. Permissible values are 2, 4, 6,...

The Direction input specifies the direction of data transfer:

CD_READ: data flow from a master CPU to a slave CPU

CD_WRITE: data flow from a slave CPU to a master CPU

SourceOffset specifies the offset of the memory location from which the first byte is read. For CD_READ, SourceOffset specifies a memory location in the memory module of the PS 416-CPU. For CD_WRITE, SourceOffset specifies a memory location in the array DataBuffer from which the first byte to be written is read. Permissible values for SourceOffset are 0 - 2031.

For CD_READ, DestinationOffset specifies a memory location in the array DataBuffer into which the first byte read from the special memory area is written. For CD_WRITE, DestinationOffset specifies a memory location in the special memory area into which the first byte from the array DataBuffer is written. Permissible values for DestinationOffset are 0 – 2031.

Length specifies the number of bytes to read or write. Permissible values are 1 – 2032.

The Array variable DataBuffer contains the data to be transferred. The content depends on the direction of transfer: for CD_READ, after the function block has been executed, DataBuffer contains the data read from the special memory area. For CD_WRITE, DataBuffer must contain the data to be transferred to the special memory area before the function block is called.

ErrorCode provides feedback on the processing state following execution of the command.

0 No error occurred during execution of the command.
1 SlotNumber specifies an invalid slot number. Permissible values are 2, 4, 6, 8 etc.
2 SourceOffset is larger than the maximum value of 2031.
3 DestinationOffset is larger than the maximum value of 2031.
4 Length is larger than 2032, or Length + SourceOffset or DestinationOffset exceeds the size of the special memory area (2032 bytes) of the size of the array DataBuffer.
5 DataBuffer is not or incorrectly specified.
6 The Direction input has been configured with an illegal value.

Example: data exchange between PLCs

```
PROGRAM cde

VAR
  DataExchange : CPUDataExchange ;
  EnableRead AT%i0.0.0.0.0 : Bool ;
  ReadData : Array[1..20] of Byte;
  ReadError : Uint ;
END_VAR

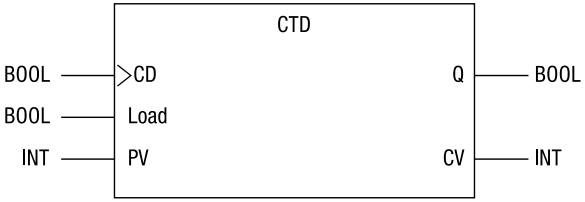
  LD   EnableRead
  CALC DataExchange(SlotNumber := 4,
                    Direction := CD_READ,
                    SourceOffset := 0,
                    DestinationOffset := 0,
                    Length := 20,
                    DataBuffer := ReadData
                    |
                    ReadError := Errorcode )

END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC

CTD
Down Counter



Function block prototype

Meaning of operands

CD	Counter pulses, rising edge
Load	Set condition
PV	Initial value
Q	Message: counter status ≤ 0
CV	Counter status

Description

The function block CTD is used for the down counting of impulses provided at the input operand CD. The counter is set to 0 when initialised.

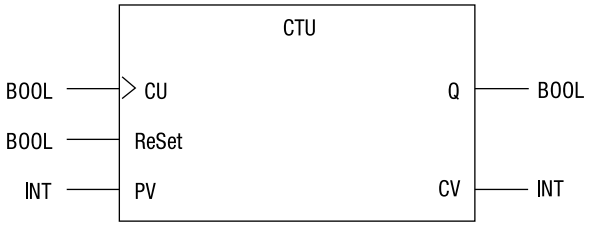
The value defined by the operand PV is loaded into the counter as the initial value if the value of the operand Load is 1.

Each rising edge on the input CD is decremented, i.e. reduced by one.

The output operand CV outputs the current counter value. If the counter value is above the value 0, the output operand Q has the Boolean value 0. The output Q is set to 1 when reaching or dropping below the counter value 0.

IEC

CTU
Up counter



Function block prototype

Meaning of operands

CU	Counter pulses, rising edge
ReSet	Reset condition
PV	Counter limit
Q	Signal: counter status PV
CV	Counter status

Description

The function block CTU is used for the up counting of impulses provided by the input operand CU. The counter is set to 0 when initialised. Each rising edge on input CU increments the counter, i.e. increases its value by one.

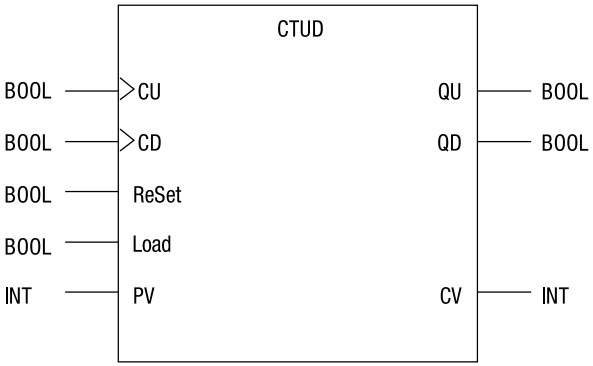
The counter value can be cleared with the value 1 at the operand ReSet.

The output operand CV outputs the current counter value. If the counter value is below the limit value PV, the output operand Q has the Boolean value 0. The output Q is set to 1 when reaching or exceeding the limit value.

07/01 AWB2700-1306-GB

IEC

CTUD
Up And Down Counter



Function block prototype

Meaning of operands

CU	Count pulse for upward counting, rising edge
CD	Count pulse for downward counting, rising edge
ReSet	Reset condition
Load	Load condition
PV	Load value
QU	Indication: counter status greater than or equal to PV
QD	Indication: counter status less than or equal to 0
CV	Counter status

Description

The function block CTUD is used for the counting up and down of impulses. The counter is set to 0 when initialised. Each rising edge on input CU increments the counter, i.e. increases its value by one. A rising edge on input CD decrements the counter, i.e. reduces its value by one.

The value defined by the operand PV is loaded into the counter if the value of the operand Load is 1.

The counter value can be cleared with the value 1 at the operand ReSet. While the static status 1 of the operands ReSet is kept, the fulfilled counter or load conditions have no effect on the counter value.

The output operand CV outputs the current counter value. If the counter value is below the load value PV, the output operand QU has the Boolean value 0. The output QU is set to 1 when reaching or exceeding the load value. If the counter value is over the value 0, the output operand QD has the Boolean value 0. The output QD is set to 1 when reaching or dropping below the counter value 0.

Example: upward and downward counting

```
PROGRAM visitors

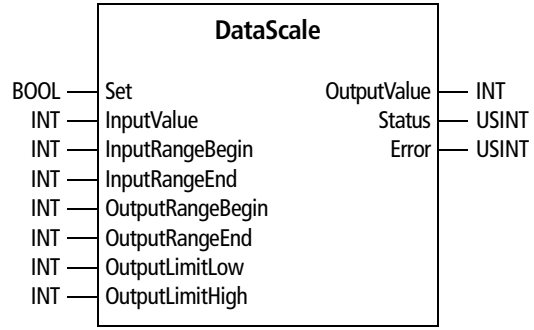
VAR
  PersonCounter : CTUD;
  Lightbarrier1 AT %IO.0.0.0.2 : BOOL;
  Lightbarrier2 AT %IO.0.0.0.3 : BOOL;
  Reset AT %IO.0.0.0.1 : BOOL;
  PersonNumber AT %QW0.0.0.0 : INT;
END_VAR

  CAL  PersonCounter(CU := Lightbarrier1,
                     CD := Lightbarrier2,
                     ReSet:= Reset)
  LD PersonCounter.CV
  ST PersonNumber

END_PROGRAM
```


07/01 AWB2700-1306-GB

DataScale
Scaling of Data



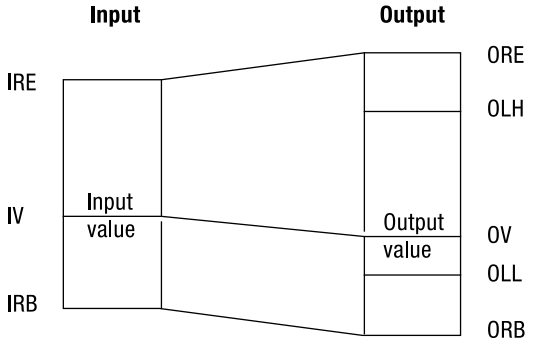
Function block prototype

Meaning of operands

Set	Activates the function block
InputValue	Input value (IV) which can vary within the input range; IRB ≤ IV ≤ IRE
InputRangeBegin	Bottom of input range (IRB) IRB < IRE
InputRangeEnd	Top of input range (IRE)
OutputRangeBegin	Bottom of output range (ORB) ORB < ORE
OutputRangeEnd	Top of output range (ORE)
OutputLimitLow	Lower limit value (OLL) OLL < OLH ≤ ORE and OLL ≥ ORB
OutputLimitHigh	Upper limit value (OLH) OLH > OLL ≥ ORB and OLH ≤ ORE
OutputValue	Output value:
Status	Indications (range overshoots)
Error	Error messages (parameters)

Description

The input range within which the value to be processed (IV) can vary is defined with the values IRE (greatest input value) and IRB (smallest input value). You can assign any output range you wish (ORE and ORB) to this input range. The input value is converted using the ratio of input range to output range. 2 limit values (OLL and OLH) must be defined within this output range.
If the converted value exceeds or undershoots a limit, a message is output.



When the Set input has status 1, the range values set by the user are checked. Incorrect values are signalled at the Error output byte. The value at the InputValue input is converted and sent to the OutputValue output.

If the limit is undershot or overshot, the OutputValue output is set to the relevant limit and the error bits set at the Status output.

When there is no error, conversion continues for as long the Set input has status 1.

Changes to the value range are accepted once the Set input has been set to 0 then back to 1 again.

07/01 AWB2700-1306-GB

When the Set input has status 0, output OutputValue = ORB, output Error = 0 and output Status = 0.

Error (Value)	**Error messages (parameters)**
0	no error
1	IRB ≥ IRE
2	ORB ≥ ORE
3	OLL > OLH, OLL < ORB, OLL > ORE, OLH < ORB, OLH > ORE

Status (Value)	**Messages (range overshoots)**
0	no error
1	OV < OLL
2	OV > OLH
3	IV < IRB
4	IV > IRE

Removal of the error causes the byte to be reset to 0.

Example Scaling

```
PROGRAM analog_processing

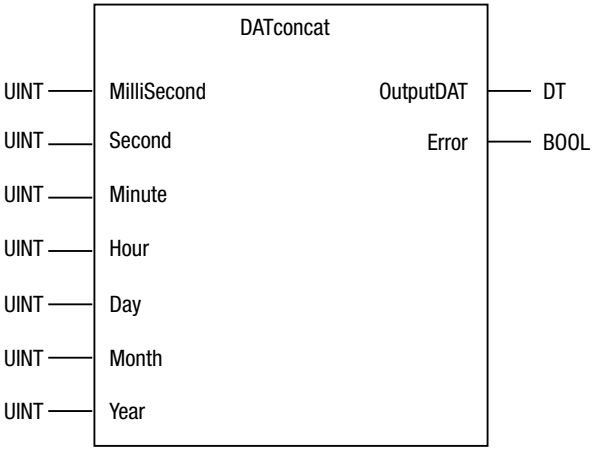
VAR
  Converter:              DataScale;
  Start                   AT %IO.0.0.0.0: BOOL;
  AnalogInput             AT %IAWO.0.0.4: INT;
  AnalogOutput:           INT;
  FurtherProcessing:      INT;
  RangeError:             USINT;
  ParameterError:         USINT;
END_VAR

  CAL Converter  (Set             := Start,
                  InputValue      := AnalogInput,
                  InputRangeBegin := 200,
                  InputRangeEnd   := 400,
                  OutputRangeBegin := 200,
                  OutputRangeEnd  := 4000,
                  OutputLimitLow  := 250,
                  OutputLimitHigh := 3800
                  |
                  AnalogOutput    := OutputValue,
                  RangeError      := Status,
                  ParameterError  := Error)
  LD AnalogOutput
  ST FurtherProcessing

END_PROGRAM
```


07/01 AWB2700-1306-GB

DATconcat
Concatenate Date and Time



Function block prototype

Meaning of operands

MilliSecond	Milliseconds part of date and time
Second	Seconds part of date and time
Minute	Minutes part of date and time
Hour	Hours part of date and time
Day	Day part of date and time
Month	Month part of date and time
Year	Year part of date and time
OutputDAT	Resulting value for date and time
Error	Error message

Description

The values specified as input operands of data type UINT are combined to a value for date and time in the output operand OutputDAT.

The function block converts the individual values of the input operands into the appropriate components for date and time. The input operands must not exceed the value ranges specified for a date and a time. The year number must be between 1993 – 2092. If the specified value ranges are exceeded, the status 1 will show at the Error output.

Example: Concatenating date and time

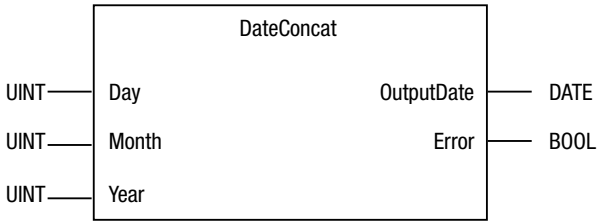
```
PROGRAM dtgen

VAR
   DateTimeConcatenation : DATconcat;
   MilliSec : UINT := 0;
   Sec : UINT := 30;
   Min : UINT := 10;
   Hr : UINT := 3;
   Dy : UINT := 12;
   Mnth : UINT := 12;
   Yr : UINT := 1997;
   Err : BOOL;
   Date_1 : DATE_AND_TIME;
END_VAR

   CAL DateTimeConcatenation(MilliSecond := MilliSec,
                             Second := Sec,
                             Minute := Min,
                             Hour := Hr,
                             Day := Dy,
                             Month := Mnth,
                             Year := Yr
                             |
                             Date_1 := OutputDAT,
                             Error := Err)

END_PROGRAM
```

DateConcat
Concatenate Date



Function block prototype

Meaning of operands

Day	Day part of a date
Month	Month part of a date
Year	Year part of a date
OutputDate	Value resulting from the defined date sections
Error	Error message

Description

The values entered as input operands of data type UINT are combined to a date value in the output operand OutputDate. The input operands must be outside of the permissible value range for dates. The year value must be between 1993 – 2092. The Error output will be set to 1 if an impermissible value is entered.

Example: concatenate date

```
PROGRAM dategen

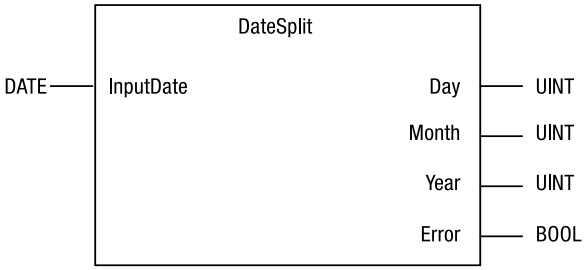
VAR
  Dateconcaten : DateConcat;
  Date1 : Date;
  Dy : UINT := 24;
  Mnth : UINT := 12;
  Yr : UINT := 1996;
  Err : BOOL;
END_VAR

  CAL Dateconcaten(Day := Dy,
                   Month := Mnth,
                   Year := Yr
                   |
                   Date1 := OutputDate,
                   Error := Err)


END_PROGRAM
```


07/01 AWB2700-1306-GB

DateSplit
Split DATE Data Type



Function block prototype

Meaning of operands

InputDate	Date
Day	Day part of date
Month	Month part of date
Year	Year part of date
Error	Error message

Description

The date value specified via the input operand InputDate is divided into elements that are each output as UINT data types.

The year value must be between 1993 – 2092. The output operand Error will be set to 1 if an impermissible year value is entered.

Example: splitting date value

```
PROGRAM datetln

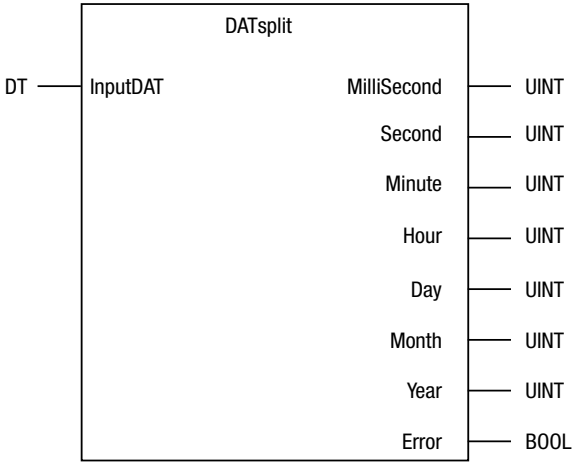
VAR
  DateElements : DateSplit;
  Date1 : Date := D#1999-12-24;
  Dy : UINT;
  Mnth : UINT;
  Yr : UINT;
  Err : BOOL;
END_VAR

  CAL  DateElements(InputDate := Date1
                        |
                       Dy := Day,
                       Mnth := Month,
                       Yr := Year,
                       Err := Error)

END_PROGRAM
```


07/01 AWB2700-1306-GB

DATsplit
Split DATE_AND_TIME Data Type



Function block prototype

Meaning of operands

InputDAT	Date and time
MilliSecond	Milliseconds part of date and time
Second	Seconds part of date and time
Minute	Minutes part of date and time
Hour	Hours part of date and time
Day	Day part of date and time
Month	Month part of date and time
Year	Year part of date and time
Error	Error message

Description

The date and time value specified via input operand InputDAT is divided into elements that are each output as UINT data types.

The year number must be between 1993 – 2092. The output operand Error will be set to 1 if an impermissible year value is entered.

Example: Splitting date and time values

```
PROGRAM dtsplit

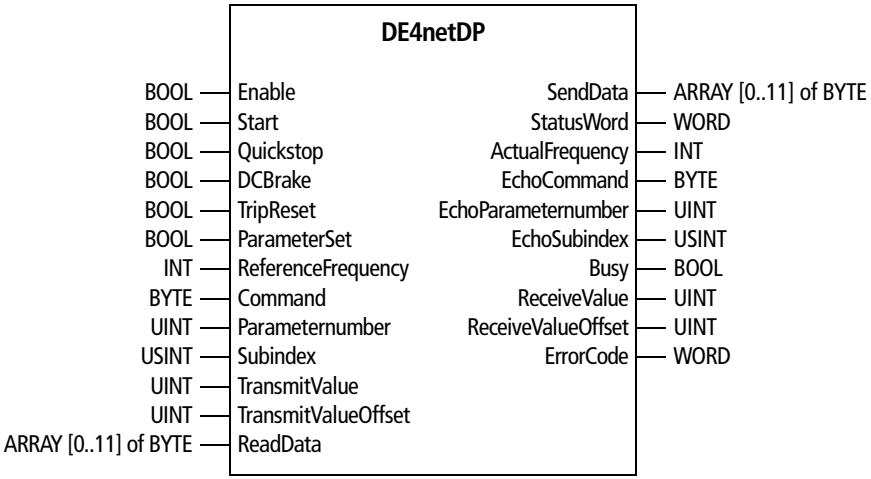
VAR
  DT_Elements : DATsplit;
  Date_1 : DT := DT#1999-11-20-22:13:12;
  Millisec : UINT;
  Sec : UINT;
  Min : UINT;
  Hr : UINT;
  Dy : UINT;
  Mnth : UINT;
  Yr : UINT;
  Err : BOOL;
END_VAR

  CAL DT_Elements(InputDAT := Date_1
                  |
                  Millisec := MilliSecond,
                  Sec := Second,
                  Min := Minute,
                  Hr := Hour,
                  Dy := Day,
                  Mnth := Month,
                  Yr := Year,
                  Err := Error)

END_PROGRAM
```


07/01 AWB2700-1306-GB

DE4netDP
PROFIBUS-DP Connection for DF4-120 and DF4-34x Frequency Inverters



Function block prototype

Meaning of operands

Enable	1 signal activates frequency inverter. 0 signal disables any further control commands. With a drive running, change of signal from 1 to 0 causes motor to coast.
Start	A 1 state enables the motor drive, the output frequency is integrated to the specified setpoint according to the defined acceleration ramp. A change from 1 to 0 stops the motor with the defined deceleration ramp.
Quickstop	At 1 state, motor brakes to standstill using the quick-stop ramp. The ramp time is adjustable with the parameter 105 (PNU 105) (only DF4-34x).
DCBrake	1 state activates DC injection braking. Brakes motor to standstill. The DC injection braking remains active as long as the signal state is 1.

TripReset	With change of signal from 0 to 1, the current error is reset.
ParameterSet	0 activates parameter set 1. 1 activates parameter set 2.
Reference Frequency	Frequency setpoint for motor speed. The actual output frequency is limited by f_{max}. Positive values mean clockwise phase sequence, negative values mean anticlockwise phase sequence. A 500 means 50.0 Hz output frequency, i.e. the required value has to be multiplied 10.
Command	Auszuführender Auftrag: 0 = No command 1 = Read parameter 2 = Write parameter
Parameternumber	Parameter number to be read/written. All available parameter numbers that can be transferred are to be found in the manual for the motor controller.
Subindex	Subindex of parameter numbers to be read/written. All available parameter numbers with subindex that can be transferred are to be found in the manual for the drive controller (always zero for DF4).
TransmitValue	Parameter value to transfer. The last two places of the value entered are always interpreted as decimal places. e.g. 223.10 = 22310 or 1 = 100, i.e. the required value has to be multiplied by 100. The valid value range (depending on parameter number) is to be found in the manual for the drive controller.
TransmitValueOffset	An offset can be entered here for parameter values greater than 655.35 (65535), which the function block adds to TransmitValue. The last two places of the value entered are similarly interpreted as decimal places.
ReadData	First byte in receive register (%IBx.y.0.0) which is written by the function block. The entire input data range must be declared in the Variables Editor as an array.

07/01 AWB2700-1306-GB

SendData	First byte in receive register (%QBx.y.0.0) which is read by the function block.The entire output data range must be declared in the Variables Editor as an array.
StatusWord	The status word contains the current state of the motor controller in bit code. Refer to the DE4-NET-DP manual (AWB823-1290-GB) for a full description of the status word.
ActualFrequency	This displays the current output frequency of the DF4 in Hz. Positive values mean clockwise phase sequence, negative values mean anticlockwise phase sequence. A 500 means an output frequency of 50.0 Hz.
EchoCommand	Command executed (0 = No command, 1: Read parameter, 2: Write parameter)
EchoParameternumber	Parameter number that was read/written.
EchoSubindex	Subindex that was read/written. (always zero for DF4)
Busy	Signal status 1 indicates that a job is progress. When this is completed, the signal status changes to 0.
ReceiveValue	Received parameter value. The last two places of the displayed value must be interpreted as decimal places. Example: 22310 = 223.10 or 100 = 1. When ReceiveValue is 65535 => don't forget ReceiveValueOffset!
ReceiveValueOffset	If the value range of ReceiveValue has been exceeded (> 65535), an offset is entered here, which the user must add to the value in ReceiveValue. The last two places of the displayed value must likewise be interpreted as decimal places.

ErrorCode	Error messages during parameter transfer:	
	0000hex	No error
	6300hex	No access right
	6510hex	Impermissible command parameter
	6511hex	Invalid subindex
	6712hex	Data length too large
	6700hex	Object does not exist
	6800hex	Data types do not agree
	8021hex	Cannot be executed due to local control
	8022hex	Cannot be executed due to operating state of device
	8030hex	Outside value range
	8040hex	Collision with other values
	8020hex	Command cannot be executed at present
	00FEhex	Time out (reaction time of a command > 5 s)

07/01 AWB2700-1306-GB

Description
The DE4netDP function block is used to transfer process and parameter data. The identifier of the user data length in the PROFIBUS configurator must be $B7_{hex}$ 71_{hex}. This means, 8 consistent bytes are transmitted via the DP parameter channel plus 2 words of process data.

The mode of the frequency inverter must first be set to 3 “Fieldbus mode”. This can be done with the keypad, operating software or even via the field bus.

As soon as communication is established between PLC and the DF4, the process data is transferred automatically and continuously with no explicit programming required. If other parameters of the DF4 are to be changed via PROFIBUS-DP, this is done via the parameter channel.

Process input data:
Enable, Start, Quickstop, DCBrake, TripReset, ParameterSet, ReferenceFrequency:

Process output data:
StatusWord, ActualFrequency



To achieve the shortest possible reaction times, you should call this function block once only in each user program cycle in order to update the inputs and outputs.

All further parameters of the frequency inverter must be read or written via the parameter data channel.

Read parameter:

- Specify parameter number in Parameternumber
- Specify subindex in the Subindex field (always equals zero for the DF4)
- Specify “Command = 1”. Busy changes from 0 to 1 automatically

- Wait until Busy has changed from 1 to 0:
  ErrorCode equals 0:
  accept ReceiveValue and possibly ReceiveValueOffset

  ErrorCode not equal 0:
  job not executed,
  check ErrorCode
- Specify “Command = 0”. Next command can be started

Write parameter:

- Specify parameter number in Parameternumber
- Specify subindex in the Subindex field (always equals zero for the DF4)
- Specify parameter value in TransmitValue and possible TransmitValueOffset
- Specify Command = 2. Busy changes from 0 to 1 automatically
- Wait until Busy has changed from 1 to 0:
  ErrorCode equals 0:
  accept ReceiveValue and possibly ReceiveValueOffset
  ErrorCode not equal “0“:
  job correctly executed,
  check ErrorCode
- Specify Command = 0. Next command can be started

07/01 AWB2700-1306-GB

Response when bus interrupted (PNU 4000)



This parameter is only available for communication via PROFIBUS and is therefore not described in the manual for the frequency inverter.
This parameter, and thus the response, is not stored retentively in the DF4 frequency inverter but in the optional DE4-NET-DP module.

PNU 4000 defines the response of the frequency inverter if the PROFIBUS-DP connection is interrupted.

PNU 4000 = 0:
no action, the motor retains the current state (factory setting)

PNU 4000 = 1:
Controller disable, the motor runs down

PNU 4000 = 2:
Quickstop, the motor brakes to a standstill using the quickstop ramp. The controller is then in the disabled state.

Example

The following example shows the declaration in the Variables Editor:

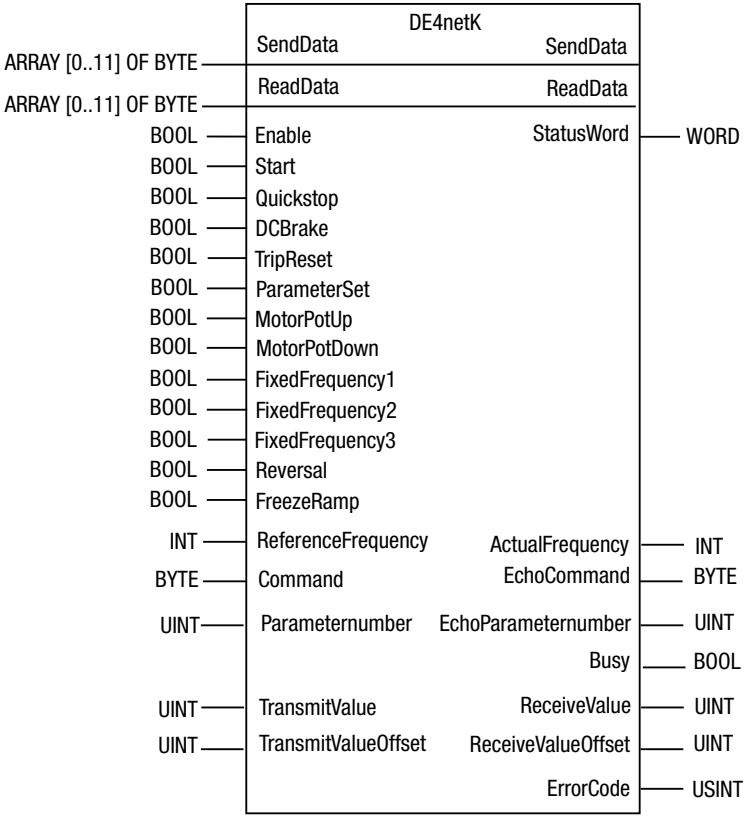
```
VAR
  Drive_1                    : DE4netDP;
  IN_PDP_S1T3at%IB1.3.0.0    : Array [0..11] of BYTE;
  OUT_PDP_S1T3at%QB1.3.0.0   : Array [0..11] of BYTE;
END_VAR
```

The following example shows the DE4netDP function block in the POU Editor:

```
CAL Drive_1(
  Enable                  := ,
  Start                   := ,
  Quickstop               := ,
  DCBrake                 := ,
  TripReset               := ,
  ParameterSet            := ,
  ReferenceFrequency      := ,
  Command                 := ,
  Parameternumber         := ,
  Subindex                := ,
  TransmitValue           := ,
  TransmitValueOffset     := ,
  ReadData                := IN_PDP_S1T3
  |
  OUT_PDP_S1T3    :=SendData,
                  :=StatusWord,
                  :=ActualFrequency,
                  :=EchoCommand,
                  :=EchoParameternumber,
                  :=EchoSubindex,
                  :=Busy,
                  :=ReceiveValue,
                  :=ReceiveValueOffset,
                  :=ErrorCode
)
```


07/01 AWB2700-1306-GB

DE4netK
DF4 Frequency Inverter



Function block prototype

Meaning of operands

SendData	Operator list for the send register
ReadData	Operator list for the receive register
Enable	1: Enable frequency inverter 0: Disable frequency inverter

Start	1: Accelerate motor to frequency setpoint, 0: Brake motor using braking ramp
Quickstop	1: Brake motor to standstill using quick-stop ramp
DCBrake	1: Activate DC injection braking
TripReset	Signal transition 0 to 1: Reset last error
ParameterSet	0: Activate parameter set 1 1: Activate parameter set 2
MotorPotUp	1: Activate function Motor potentiometer UP
MotorPotDown	1: Activate function Motor potentiometer DOWN
FixedFrequency1	1: Activate jog frequency 1 if FixedFrequency2 and FixedFrequency3 = 0
FixedFrequency2	1: Activate jog frequency 2 if FixedFrequency3 = 0
FixedFrequency3	1: Activate jog frequency 3
Reversal	Reverse motor direction
FreezeRamp	0: Enable acceleration ramp 1: Freeze acceleration ramp
ReferenceFrequency	Frequency setpoint for motor speed
Command	Execute command (0: No command, 1: Read parameter, 2: Write parameter)
Parameternumber	Parameter number to transfer
TransmitValue	Parameter value to transfer
TransmitValueOffset	Offset for parameter values above 655.35
StatusWord	Drive controller status information
ActualFrequency	Display current output frequency
EchoCommand	Command executed (0: No command, 1: Read parameter, 2: Write parameter)
EchoParameter-number	Parameter number of command executed

07/01 AWB2700-1306-GB

Busy	1: Command being processed
ReceiveValue	Received parameter value
ReceiveValueOffset	Offset for parameter values above 655.35
ErrorCode	Error code

Description
The DE4netK function block forms the interface between a PLC program and the DE4-NET-K fieldbus module.

As soon as communication is established between the PLC and the DF4, process data is exchanged automatically and continuously without requiring explicit programming. Other DF4 parameters are changed via Suconet using the parameter channel.



To minimise reaction times, call the function block just once per program cycle in order to update the inputs and outputs.

The following inputs and outputs are read and/or written by the function block each time it is called (process data):

Inputs: Enable, Start, Quickstop, DCBrake, TripReset, ParameterSet, MotorPotUp, MotorPotDown, FixedFrequency1, FixedFrequency2, FixedFrequency3, Reversal, FreezeRamp, ReferenceFrequency.

Outputs: StatusWord, ActualFrequency

All further parameters of the frequency inverter must be read or written via the parameter data channel.

The operators SendData and ReadData must be assigned to the first send data and/or receive data byte, respectively. The following 11 bytes are then automatically declared by the array (see example).

The motor drive is enabled by a 1 at the Start input. The output frequency of the drive is accelerated to the specified frequency setpoint using the configured acceleration ramp. A transition from 1 to 0 at the Start input stops the motor with the configured deceleration ramp.

ReferenceFrequency specifies the frequency setpoint, i.e. the required motor speed. Positive values mean clockwise phase sequence, negative values mean anticlockwise phase sequence. 500 signifies an output frequency of 50.0 Hz, i.e. the required value must be multiplied by a factor of 10.

Please see the user manual for the respective frequency inverter model (motor drive controller), for information on available parameter numbers, permissible ranges for parameters and the device status word.

The last two digits of a parameter value at the TransmitValue input are interpreted as decimal places, e.g.: 223.10 = 22310 or 1 = 100. This means that the value in the operating manual must be multiplied by 100.

For parameter values larger than 655.35 (i.e. 65535) you must specify an offset in TransmitValueOffset which is added to the value in TransmitValue. In TransmitValueOffset too, the last two digits are interpreted as decimal places.

The status word returns current drive status information in bit-mapped format.

The current output frequency of the DF4 is indicated by ActualFrequency in Hz. Positive values mean clockwise phase sequence, negative values mean anticlockwise phase sequence. 500 signifies an output frequency of 50.0 Hz.

07/01 AWB2700-1306-GB

The last two digits of the value displayed at the ReceiveValue output are interpreted as decimal places, e.g.: 22310 = 223,10 or 100 = 1.

For parameter values larger than 655.35 (i.e. 65535) the remainder of the total parameter value required is automatically entered in ReceiveValueOffset. The last two digits are interpreted here too as decimal places.

Read parameter:

- Specify parameter number in Parameternumber
- Specify “Command = 1”. Busy changes from 0 to 1 automatically
- Wait until Busy has changed from 1 to 0:
  ErrorCode equals 0:
  accept ReceiveValue and possibly ReceiveValueOffset
  ErrorCode not equal 0:
  job not executed,
  check ErrorCode
- Specify “Command = 0”. Next command can be started

Write parameter:

- ▶ Specify parameter number in Parameternumber
- ▶ Specify parameter value in TransmitValue (and in TransmitValueOffset if reqd.)
- ▶ Specify Command = 2. Busy changes from 0 to 1 automatically
- ▶ Wait until Busy has changed from 1 to 0:
  ErrorCode equals 0:
  accept ReceiveValue and possibly ReceiveValueOffset
  ErrorCode not equal “0“:
  job correctly executed,
  check ErrorCode
- ▶ Specify “Command = 0”. Next command can be started

Error codes for parameter data transfer at the ErrorCode output:

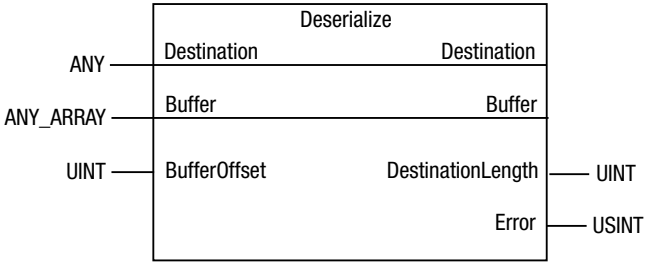
0 No error
1 Invalid service
5 Invalid parameter number
6 Invalid command parameter
10 Access not allowed, e.g. write request for read-only parameters
14 Permissible range exceeded, e.g. value too high
254 Timeout if the reaction time to a read/write command is longer than 5 seconds
255 Other error – command cannot be executed at present

Example: function block DE4netK
Declare station 1 on line 1

```
VAR
  Drive_1 :DE4netK;
  RD_S1T1 at %RDB1.1.0.0 :ARRAY[0..11] OF BYTE;
  SD_S1T1 at %SDB1.1.0.0 :ARRAY[0..11] OF BYTE;
END_VAR


CAL Drive_1   (Enable := ,
               Start := ,
               Quickstop := ,
               DCBrake := ,
               TripReset := ,
               ParameterSet := ,
               MotorPotUp := ,
               MotorPotDown := ,
               FixedFrequency1 := ,
               FixedFrequency2 := ,
               FixedFrequency3 := ,
               Reversal := ,
               FreezeRamp := ,
               ReferenceFrequency := ,
               Command := ,
               Parameternumber := ,
               TransmitValue := ,
               TransmitValueOffset := ,
               SendData := SD_S1T1,
               ReadData := RD_S1T1
               |
               :=StatusWord,
               :=ActualFrequency,
               :=EchoCommand,
               :=EchoParameternumber,
               :=Busy,
               :=ReceiveValue,
               :=ReceiveValueOffset,
               :=ErrorCode)
```

Deserialize
Extract Data from Arrays



Function block prototype

Meaning of operands

Destination	Destination variable
Buffer	Source area
BufferOffset	Offset in Source area
DestinationLength	Length of destination variable in bytes
Error	Error code

Description

The Deserialize function block extracts any required data type from a source area at a starting position specified by Buffer + Bufferoffset.

The DestinationLength output specifies the length of the destination data type in bytes.

Buffer is an array of any size. All ANY_BIT and ANY_INT types apart from BOOL are permissible as array elements.

07/01 AWB2700-1306-GB

Error codes at the Error output:

0 No error
1 Buffer overflow
2 Offset lies outside the buffer
3 Destination lies within the buffer
4 DestinationLength is 0



The destination data type BOOL is not supported and will result in an error during compilation. For this see the function Serialize on page 485.

Example: Extracting 20 bytes from a 30 byte array

```
PROGRAM sendstruct
VAR
   deserie_fb : Deserialize;
   struct_buffer AT %RDB1.1.0.0 : ARRAY [1..30] OF BYTE;
   CheckSumOffset : UINT := 0;
   CheckSum : UINT;
   DTOffset : UINT := 2;
   ReceiveDate : DT;                    (* Receive date*)
   DataOffset: UINT := 10;
   ReceiveData :Array [1..20] OF BYTE; (* User data *)
   DeserializedLength : UINT;
   Error : USINT;
END_VAR

   (* Extract receive data *)
   CAL deserie_fb( Destination := CheckSum,
                   Buffer := struct_buffer,
                   BufferOffset := CheckSumOffset,
                   |
                   DeserializedLength := DestinationLength)
   LD deserie_fb.Error
   USINT_TO_BOOLTab>                    (* Any Error ? *)
   JMPC _Fail       (* Error handling *)
```

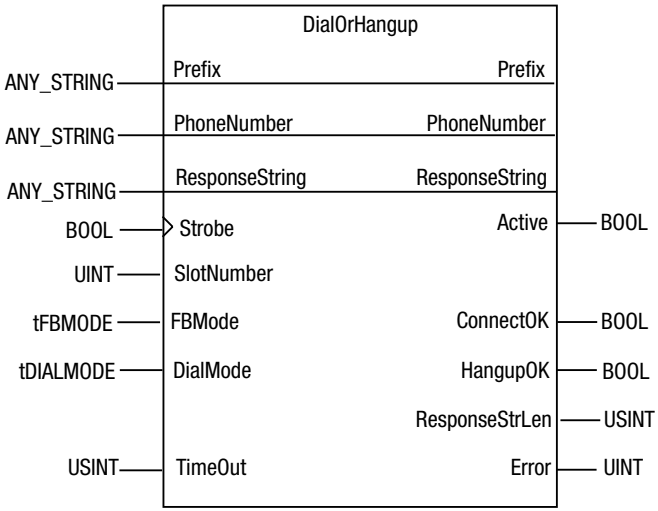
```
   CAL deserie_fb( Destination := ReceiveDate,
                   Buffer := struct_buffer,
                   BufferOffset := DTOffset,
                   |
                   DeserializedLength := DestinationLength)
   LD    deserie_fb.Error
   USINT_TO_BOOL                         (* Any Error ? *)
   JMPC  _Fail                           (* Error handling *)

   CAL deserie_fb( Destination := ReceiveData,
                   Buffer := struct_buffer,
                   BufferOffset := DataOffset,
                   |
                   DeserializedLength := DestinationLength)
   LD    deserie_fb.Error
   USINT_TO_BOOL                         (* Any Error ? *)
   JMPC   _Fail                          (* Error handling *)

   (* Received data complete *)

_Fail:              (* Error handling *)
```

DialOrHangup
Establish or Clear a Connection to a DTE via Modem



Function block prototype

Meaning of operands

Prefix	String in front of the telephone number (e.g. "0W") If no prefix is required, the associated string must at least contain an empty string (String(0):="";).
PhoneNumber	Telephone number (sum of Prefix and PhoneNumber max 34 characters)
ResponseString	Modem response string (max. 254 bytes)
Strobe	Enable function block
SlotNumber	Selection of PLC interface: 0: PS416-CPU 400/300-SBI or PS4-341-MM1 SBI (Transparent mode), 4 to 20: Slot of specified COM card
FBMode	Mode Dial or Hangup

DialMode	Dialing mode Tone or Pulse
TimeOut	Wait for response (10..90 seconds), Standard value: 60 seconds
Active	Status of function block processing 1: Function block active (parameters accepted) 1 -> 0: Function block completed
ConnectOK	CONNECT string in ResponseString 1: CONNECT received, 0: CONNECT not received
HangupOK	OK string in ResponseString 1: OK received, 0: OK not received
ResponseStrLen	Number of characters received
Error	Error code

Description

The DialOrHangup function block has two operating modes: Dial or Hangup. This is specified with the FBMode input:

```
TYPE tFBMODE:
       ( Dial,
       Hangup ):=Dial;
END_TYPE
```

In Dial mode, the specified telephone number is dialed to establish a connection to a DTE via modem.

In Hangup mode, an existing connection to a DTE is cleared.

The function block requires that both PLC and DTE are connected to modems; furthermore, the parameters of the serial interface should be set to the same values on the PLC and the PLC modem.

An active send job will be aborted on the PS416-COM-200 card if the modem function block is activated via the TransmitActive output on the COM function block. This can be avoided by starting a

07/01 AWB2700-1306-GB

timer with the TransmitActive output, that causes a delay according to the specified baud rate and the data volume to be sent. This does not affect Transparent mode with the central unit.

Error codes are output to the Error output. Errors 1 to 11 can occur during function block execution; with errors 16 to 25 the function block was not executed.

0 No error
1 ResponseString contains an unknown response
2 ResponseString incomplete or is longer than the declared length of the ResponseString variable
3 ResponseTimeOut. For example, no response received because
  – Problem with connection to modem,
  – Interface parameters for the PLC and modem do not match,
  – Modem not active.
4 Error when receiving data
5 Error when transmitting data
6 ResponseString contains: BUSY called phone number engaged
7 ResponseString contains: NO CARRIER
  No connection established or carrier lost for more than 10 seconds
8 ResponseString contains: ERROR
  Invalid modem command
9 ResponseString contains: NO DIALTONE
  No dial tone received
10 ResponseString contains for example:
  DIAL LOCKED, .. DELAYED
  Dialing locked
11 ResponseString contains: NO ANSWER
  No answer from called number
16 SlotNumber invalid (0, 4...20)
20 TimeOut not in permissible range (10...90)
21 CPU 400-SBI/COM 200: Transparent mode not or incorrectly set

22 COM 200: Not in specified slot
23 COM 200: No or incorrect IFM inserted
25 Modem not configured for Echo-OFF

Dial Mode
On detecting a rising edge at the Strobe input, all input parameters (SlotNumber, FBMode, DialMode, Prefix, PhoneNumber and TimeOut) are read.

The settings for the parameters DialMode and Prefix are dependent on the type of modem and telephone connection. DialMode specifies the two different dialing methods which are used.

```
Type tDIALMODE:
       ( Tone,
       Pulse ) :=Tone;
END_TYPE
```

The parameters Prefix, DialMode and PhoneNumber are combined by the function block into an AT command and sent to the modem.

After enabling the function block, Active is set to 1. Active is set back to 0 if a modem response is received, if a parameter error is detected or if the timeout is exceeded.

The timeout is specified with TimeOut in seconds.

Signal transitions at the Strobe input are ignored while Active is 1.

ResponseString contains the entire response string from the modem. ConnectOK is set to 1 if the modem has returned a connect message.

The HangupOK output is not used in Dial mode.

Error is 0 if no error was detected. It contains an error code if an error was detected or the modem response is not a connect message.

07/01 AWB2700-1306-GB

Hangup mode
On detecting a rising edge at the Strobe input, an existing connection to a DTE is cleared. The Prefix and PhoneNumber output are not used in Hangup mode but must still be specified.



The function block automatically configures the modem as follows
– Modem result codes on,
– Modem result codes plain text on
– Modem command echo off.
Other function blocks can also be used to access the COM or SBI interface. Conflicts can occur if several function blocks try to access the same interface. If necessary, the user program should include interlocks to make sure that this cannot happen.

Example: establishing a connection and hanging up

```
PROGRAM Dial_Com_Hangup
(* Press strobe button=Dial -> Send message with FB "COM"
-> Press strobe button=Hangup *)

VAR
   FB_DAH : DialOrHangup;
   FB_SEND : COM;
   StringToArray : STRtoCOM;
   DAH_FallingEdge : F_TRIG;
   SEND_FallingEdge : F_TRIG;
   Make_Dial : BOOL;
   DAHFlag : BOOL;
   SENDFlag : BOOL;
   PhoneString : String(30);
   FB_Strobe AT %I0.0.0.0.0 : BOOL;
   Display_DAH_Active AT %Q0.0.0.0.0 : BOOL;
   Display_ConnectOK AT %Q0.0.0.0.1 : BOOL;
   Display_HangupOK AT %Q0.0.0.0.2 : BOOL;
   Display_DAH_Error AT %QW0.0.0.2 : UINT;
   Prefix : STRING(3):='OOW';
   PhoneNumber : STRING(13):='0228,602,1414';
   Response : STRING(254);
   message : String(54):='Hello, this is the PS 416 $R$L';
   sendString : String(80);
   sendData : ARRAY [1..256] OF BYTE;
   receiveData : ARRAY [1..256] OF BYTE;
   ...
END_VAR

FbMODE:  (* FBMode_Selection: Dial or Hangup mode *)
   LD    Make_Dial
   JMPC  disconnect
   LD    Dial
   ST    FB_DAH.FBMode
   JMP   Phone

Disconnect:
   LD    Hangup
   ST    FB_DAH.FBMode
   JMP   Start

Phone:
   LD    PhoneNumber
   St    PhoneString
   JMP   Start
```


07/01 AWB2700-1306-GB

```
Start:(* Enable Dial or Hangup *)
   LD    DAHFlag
   JMPC  CheckAnswer

Call:
   CAL FB_DAH( Strobe :=FB_Strobe,       (* Push-button*)
               SlotNumber :=10,
               FBMode :=,
               DialMode :=Tone,
               Prefix :=Prefix,
               PhoneNumber :=PhoneString,
               ResponseString :=Response,
               TimeOut := 60
               |
               Display_DAH_Active :=Active,
               Display_ConnectOK :=ConnectOK,
               Display_HangupOK :=HangupOK,
               :=ResponseStrLen,
               Display_DAH_Error :=Error)

   LD    Response
   LD    DAHFlag      (* Evaluate falling edge of FB_Active*)
   JMPC  CheckAnswer
   CAL   DAH_FallingEdge(
               CLK :=FB_DAH.Active
               |
               :=Q)
   LD    DAH_FallingEdge.Q
   JMPCN End
   LD    1
   ST    DAHFlag

CheckAnswer:
   ...
   LD    FB_DAH.Error
   GT    0
   JMPC  Error
   LD    FB_DAH.ConnectOK
   JMPC  Connect
   LD    FB_DAH.HangupOK
   JMPC  Disconnected
   JMP   End

Connect: (* Send text, set strobe *)
   ...
   LD    1
   ST    FB_SEND.TransmitStrobe
   JMP   Send
```

```
Disconnected:
   ...
   LD    0
   ST    Make_Dial   (* Redial now possible *)
   LD    0
   ST    SENDFlag
   ST     DAHFlag
   LD    0           (* Send text, reset Strobe *)
   ST    FB_SEND.TransmitStrobe
   JMP   Send

Send: (* Convert string to ByteArray *)
   LD    message
   ST    sendString
   CAL   StringToArray( Enable :=1,
                TextString :=sendString,
                TransmitVariable :=sendData
                |
                :=StrLen )
   CAL   FB_SEND( ReSet :=,
                Mode :=1,
                TransmitStrobe :=,
                ReceiveStrobe :=,
                SlotNumber :=10,
                TransmitNumber :=StringToArray.StrLen,
                ReceiveNumber :=,
                TransmitVariable :=sendData,
                ReceiveVariable :=receiveData
                |
                :=TransmitActive,
                :=ReceiveActive,
                :=TransmitError,
                :=ReceiveError,
                :=ReceivedBytes,
                :=InterfaceStatus,
                :=SlotError)

   (* Detect falling edge on FB_TransmitActive *)
   LD    SENDFlag
   JMPC  CheckSendAnswer
   CAL   SEND_FallingEdge( CLK :=FB_SEND.TransmitActive
                   |
                   :=Q )

   LD    SEND_FallingEdge.Q
   JMPCN End
   LD    1
   ST    SENDFlag
```


07/01 AWB2700-1306-GB

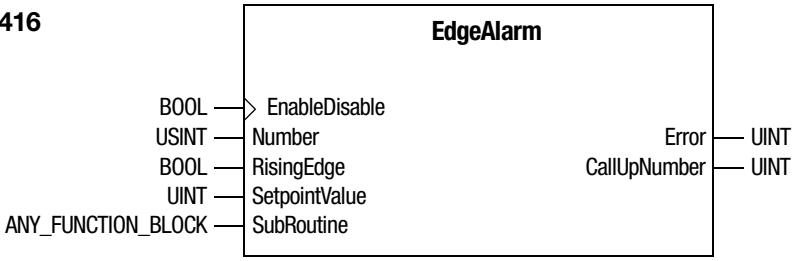
```
CheckSendAnswer:
   ...
   LD    0
   ST    DAHFlag
   LD    0
   ST    Make_Dial    (* Data sent; hang up*)
   JMP   End

Error:
   ...
   LD    1
   ST    Make_Dial    (* Dial error, hang up*)
   LD    0
   ST    SENDFlag
   ST    DAHFlag

End:
END_PROGRAM
```

PS4-200[1)]
PS4-300
PS416

EdgeAlarm
Edge-Controlled Alarm Function Block



Function block prototype

Meaning of operands

EnableDisable	Enable and disable alarm execution Start condition rising edge (0 = Disable, 1 = Enable)	
Number	PLC type	>Alarm channel number (permissible value):
	PS4-150	0
	PS4-200	0 (not suitable for PS4-271)
	PS4-300	0 and 1
	PS416	0 – 7
RisingEdge	Edge selection (0 = rising, 1 = falling)	
SetpointValue	Number of edges that have to be detected to trip the alarm (valid range: 1 to 65535)	
SubRoutine	Instance name of the function block called by the alarm	
Error	Error code	
CallUpNumber	Number of alarm-controlled function block invocations	

1) Not for PS4-271

07/01 AWB2700-1306-GB

Description

The EdgeAlarm function block is used to detect and count fast signal changes at specified inputs. There are two ways of responding to the alarm if one is detected:

– Scan of CallUpNumber output:
  The value of the CallUpNumber output is incremented with every occurrence of an alarm. The output in the user program can be scanned in order to indicate how often the alarm has occurred.
– Execution of an alarm routine:
  The alarm routine, i.e. the function block specified at the SubRoutine input is called every time an alarm occurs. In this way, you can configure event-driven reactions to the alarm. If a function block instance is not specified, just the value at the CallUpNumber output will be incremented.

The following alarm inputs are available depending on the type of PLC used:

PLC type	Input
PS4-141, PS4-151	%I0.0.0.1.0
PS4-201	%I0.0.0.0.1
PS4-271	No alarm input
PS4-300	%I0.0.0.1.0 and %I0.0.0.1.1
PS416	%I0.0.0.1.0 - %I0.0.0.1.7, in the basic unit of the PLC

One instance of the EdgeAlarm function block can be instantiated for each alarm input.

The EnableDisable input starts and stops the counter. With a rising edge (0–1 transition) the counter is started. The counter is stopped and reset if the input status changes from 1 or TRUE to 0 or FALSE.

The RisingEdge input is used to define which signal edge is to be evaluated.

The SetPointValue input defines the number of signals to be counted before the alarm and the function block are activated.

The maximum permissible load on the system depends on the following three factors:
Predivide, SetPointValue the specified frequency.
These factors must be selected so that no more than two interrupts occur per millisecond.

Function block outputs:

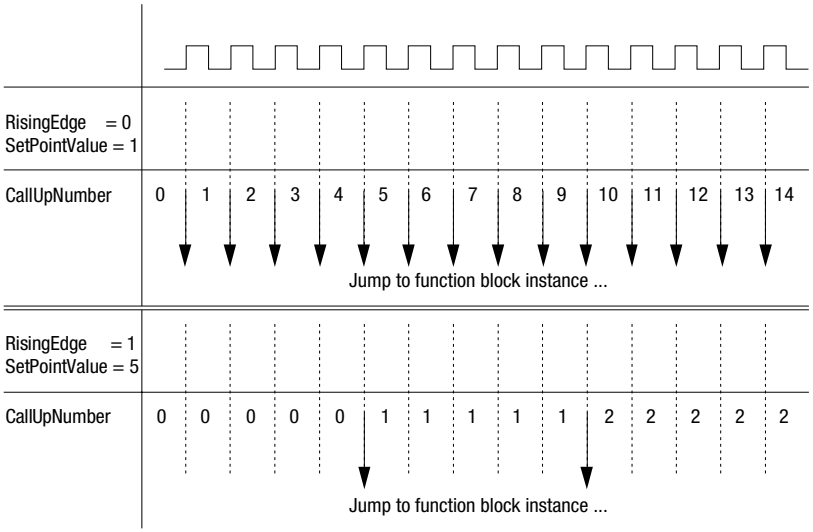
The Error output can indicate one of the following codes:

Signal	PS4-150	PS4-200	PS4-300	PS416
0	No error, activation successful			
1	Setpoint value SetPointValue = 0			
3	–	–	Invalid alarm channel number	
4	–	–	Edge alarm already used	

The CallUpNumber output (value range: 0 to 65535) indicates how often the value defined has been reached.

07/01 AWB2700-1306-GB

The SubRoutine input allows event-controlled programming to be implemented. This is done by stating the instance name of the function block to be activated when the event is reached. If no instance is specified, the CountValue will continue to be incremented without triggering the alarm.



Observe the following notes for event-driven program processing:

After the event has occurred, the user program is interrupted, the register status is saved and the function block instance defined at SubRoutine is executed. The function block instance called by the alarm function block must not have any VAR_IN_OUT parameters. This kind of instance must only have local variables (VAR) or global variables that have been declared externally as VAR_EXTERNAL variables.

The alarm function block cannot be interrupted by other alarm function blocks (all alarm function blocks

have the same priority). The maximum execution time of alarm function blocks is to be restricted to 5 ms (approx. 1K IL instructions) in the user program since the alarm function blocks cannot be interrupted even by the operating system in the event of a voltage drop. If the execution time is exceeded, an EDC error may occur when switching off the power supply.

The execution time of the alarm function block is added to the execution time of the cyclical user program and also monitored by the cycle time monitoring function.

Since the event-driven program processing enables access to the entire image register, access to data that is used by the event-driven and cyclical user program must be disabled. Bit accesses may only occur on the same byte in the cyclical user program and in the Alarm function block if the access is synchronized with Disable_Interrupt and Enable_Interrupt.

Since, due to its fast reaction times, an Alarm function block requires a high-speed peripheral access (direct output), the QP, QPB peripheral operands available in the basic unit should be used.

An alarm function block can be used several times (multiple instantiation) although this should normally be avoided since each function block group has the same event source and only the last function block instance in the program is valid. Multiple instantiation is only useful if the parameters of the function block are set again within an interrupt routine and an edge disable is to be implemented.

07/01 AWB2700-1306-GB

By multiple instantiation is meant the reservation of several data ranges for each parameter set of a particular function block type.



The input parameters of an alarm function block should only be assigned between the parentheses when it is called. An explicit parameter assignment outside of the parentheses before or after the invocation is not possible.

Example: event-driven program processing with EdgeAlarm

```
PROGRAM alarm

VAR
  Edgealarm : EdgeAlarm;
  Error : USINT;
  CountValue : UINT;
  Pump : react_fb;
  (*Declaration of function block
 that is activated in the event of an alarm *)
END_VAR

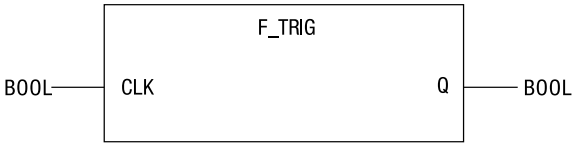
(* Call EdgeAlarm function block
  to call a function block for pumping -
  when a particular water level
  is reached in tank9.
  This is indicated via a rising edge *)

  CAL Edgealarm(EnableDisable := 1,
                  Number := 0,
                  RisingEdge := 0,
                  SetPointValue := 1,
                  Subroutine := Pump)
  LD EdgeAlarm.Error
  ST Error
  LD EdgeAlarm.CallUpNumber
  ST CountValue
   (*This indicates the number of alarms *)

END_PROGRAM
```

IEC

F_TRIG
Edge Detection of the Falling Edge



Function block prototype

Meaning of operands

CLK	Input operand whose falling edge is detected.
Q	Output operand; indicates the falling edge of CLK.

Description

The function block F_TRIG evaluates the status of the input operand CLK. It detects the status change from 1 in a processing cycle to 0 in the subsequent cycle and indicates this on the output Q with the Boolean value 1. The output is only set to 1 during the processing cycle in which the change of the status of CLK and a falling edge was detected.

07/01 AWB2700-1306-GB

Example: evaluating edges

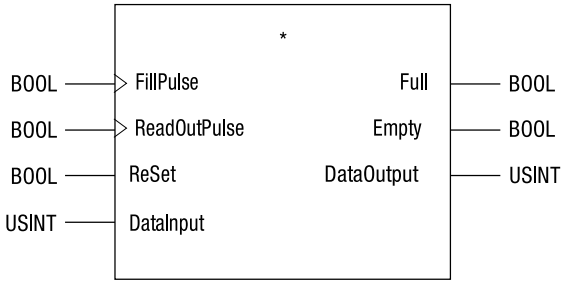
```
PROGRAM edges

VAR
  Signal1_incoming : R_TRIG;
  Signal1_outgoing : F_TRIG;
  Signal1 AT %IO.0.0.0.2 : BOOL;
  Pulse_0_1 : BOOL;
  Pulse_1_0 : BOOL;
END_VAR

  CAL  Signal1_incoming(CLK := Signal1)
  LD   Signal1_incoming.Q
  ST   Pulse_0_1
  CAL  Signal1_outgoing(CLK := Signal1)
  LD   Signal1_outgoing.Q
  ST   Pulse_1_0

END_PROGRAM
```

FifoBx
8-Bit FirstIn-FirstOut Register (Queue)



Function block prototype

Meaning of operands

FillPulse	Fill pulse
ReadOutPulse	Read pulse
ReSet	Reset
DataInput	Data input
Full	Register full
Empty	Register empty
DataOutput	Data output

Description

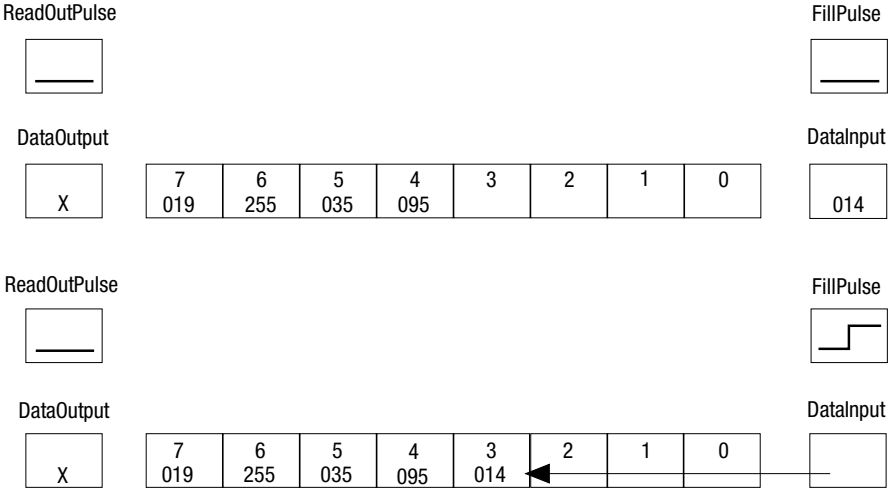
There are 128 function block prototypes. The following names can be used for *:

FifoB1
FifoB2
.
.
FifoB128

07/01 AWB2700-1306-GB

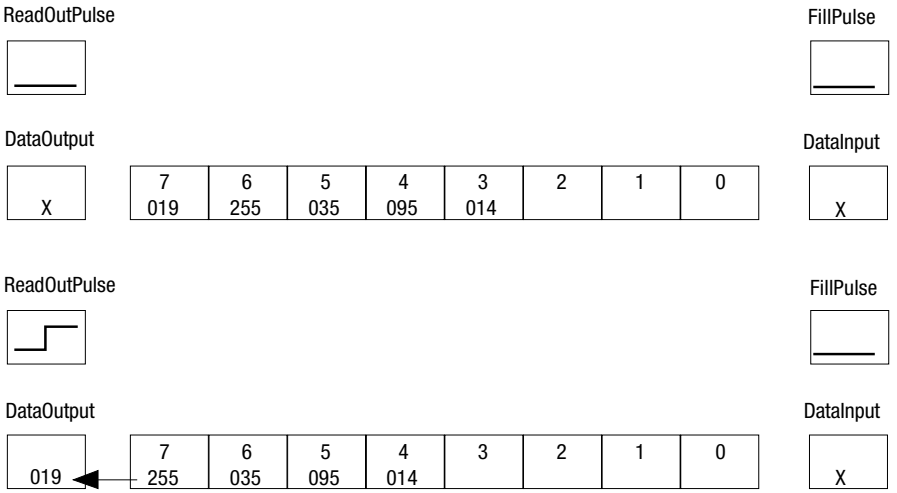
The numbers 1 ... 128 indicate the register length – the number of 8-bit register fields – of the individual prototypes.

A rising edge on the FillPulse operand transfers the value at the DataInput operand to the lowest free register field of the register. The register can be filled with every FillPulse signal until all register fields are assigned. When the Full output is 1, the register is full.



Contents of the FifoB8 register before and after a fill pulse

A rising edge on the ReadOutPulse operand reads the first register field that has been written (the lowest in the queue register) and transfers it to the DataOutput operand. Each further ReadOutPulse signal initiates the transfer of the next lowest elements of the register field to the DataOutput operand. When the Empty output is 1, this indicates that the last register field has been read and the register is empty.



Contents of the FifoB8 register before and after a ReadOut pulse

If the ReSet operand is set to 1, this clears the contents of the entire register.

07/01 AWB2700-1306-GB

Example: Queue with FifoB32

```
PROGRAM fifo8_1

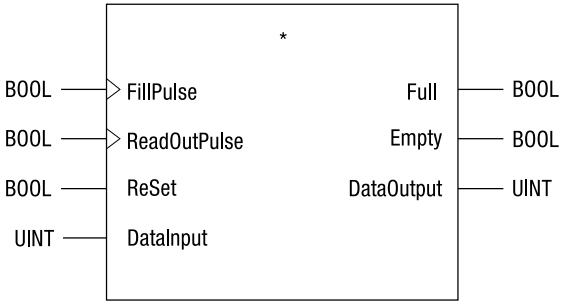
VAR
   Queue_32_byte : FifoB32;
   Pulse1 : BOOL;
   Pulse2 : BOOL;
   Reset : BOOL;
   QueueData : USINT;
   Full : BOOL;
   Empty : BOOL;
   Outputdata : USINT;
END_VAR

   .
   .
   CAL Queue_32_byte(FillPulse := Pulse1,
                             ReadOutPulse := Pulse2,
                             ReSet := Reset,
                             DataInput := QueueData)

   LD    Queue_32_byte.Full
   ST    Full
   LD    Queue_32_byte.Empty
   ST    Empty
   LD    Queue_32_byte.DataOutput
   ST    Outputdata
    .
   .

END_PROGRAM
```

FifoWx
16-Bit FirstIn-FirstOut Register (Queue)



Function block prototype

Meaning of operands

FillPulse	Fill pulse
ReadOutPulse	Read pulse
ReSet	Reset
DataInput	Data input
Full	Register full
Empty	Register empty
DataOutput	Data output

Description

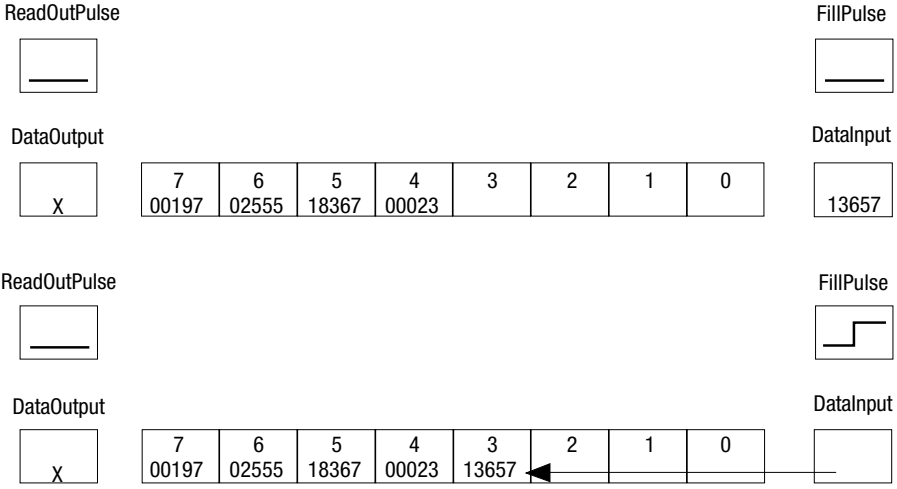
There are 128 function block prototypes. The following names can be used for *:

FifoW1
FifoW2
.
.
FifoW128

07/01 AWB2700-1306-GB

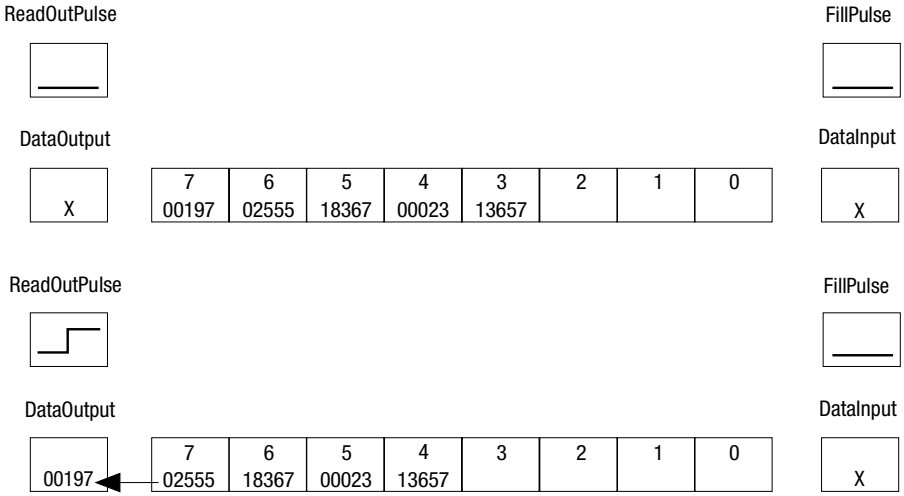
The numbers 1 ... 128 indicate the register length – the number of 16-bit register fields – of the individual prototypes.

A rising edge on the FillPulse operand transfers the value at the DataInput operand to the lowest free register field of the register. The register can be filled with every FillPulse signal until all register fields are assigned. When the Full output is 1, the register is full.



Contents of the FifoW8 register before and after a fill pulse

A rising edge on the ReadOutPulse operand reads the first register field that has been written (the lowest in the queue register) and transfers it to the DataOutput operand. Each further ReadOutPulse signal initiates the transfer of the next lowest elements of the register field to the DataOutput operand. When the Empty output is 1, this indicates that the last register field has been read and the register is empty.



Contents of the FifoW8 register before and after a ReadOut pulse

If the ReSet operand is set to 1, this clears the contents of the entire register.

07/01 AWB2700-1306-GB

Example: Queue with FifoW16

```
PROGRAM fifo16_1

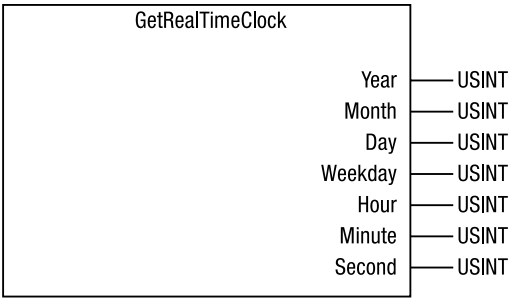
VAR
  Queue_16_word : FifoW16;
  Pulse1 : BOOL;
  Pulse2 : BOOL;
  Reset : BOOL;
  QueueData : UINT;
  Full : BOOL;
  Empty : BOOL;
  Outputdata : UINT;
END_VAR

  .
  .
  CAL Queue_16_word(FillPulse := Pulse1,
                    ReadOutPulse := Pulse2,
                    ReSet := Reset,
                    DataInput := QueueData)
  LD   Queue_16_word.Full
  ST   Full
  LD   Queue_16_word.Empty
  ST   Empty
  LD   Queue_16_word.DataOutput
  ST   Outputdata

  .

END_PROGRAM
```

GetRealTimeClock
Evaluation of the Real-Time Clock



Function block prototype

Meaning of operands

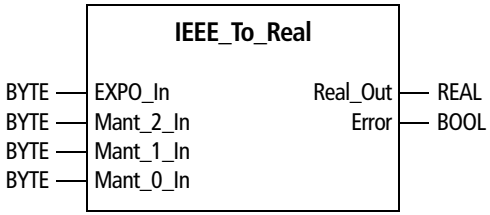
Year	Current year (two-digit)
Month	Month
Day	Day
Weekday	Weekday (0 = Sunday)
Hour	Hour
Minute	Minute
Second	Second

Description

The GetRealTimeClock function block gets the current time and date from the real-time clock and provides them at the output. An enable input is not required. The function block provides values as soon as it is called.

07/01 AWB2700-1306-GB

IEEE_To_Real
Conversion of Numbers from IEEE 754 Standard Format to Data Type REAL



Function block prototype

Meaning of operands

EXPO_In	IEEE floating point number to be converted: Exponent (incl. sign)
Mant_2_In	IEEE floating number to be converted: mantissa byte 2 (incl.1 bit exponent)
Mant_1_In	IEEE floating number to be converted: mantissa byte 1
Mant_0_In	IEEE floating number to be converted: mantissa byte 0
Real_Out	Converted real number
Error	Conversion error: IEEE number cannot be represented

Description

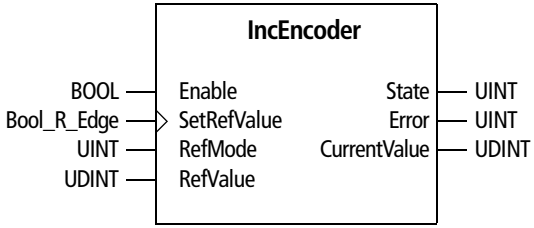
The IEEE_To_Real function block converts a floating-point number in the IEEE-754 Single Precision format to a REAL type number. To represent the floating-point number, four bytes (32 bit) are needed. This conversion only converts the format but not the value:

EXPO_In	Mant_2_In	Mant_1_In	Mant_0_In

The IEEE format is mostly used for the transfer of floating-point numbers to other systems (e.g. via PROFIBUS).

PS4-300

IncEncoder
Activation of the Local Incremental Encoder



Function block prototype

Meaning of operands

Enable	Enable for processing
SetRefValue	Initiate transfer of reference value via software
RefMode	Calibration mode: 0 = by edge on SetRefValue 1 = once via hardware 2 = permanently via hardware The RefMode setting is transferred to the FB with the first On signal at the Enable input after the PLC is cold started.
RefValue	32-bit start value of counter in Reference point/home position
State	State of calibration: 0 = not calibrated, no counter values 1 = calibrated, normal operation
Error	Error message: 0 = No errors 1 = Incremental encoder not configured 2 = Incorrect RefMode 3 = Internal error
CurrentValue	Current counter value 32-bit

07/01 AWB2700-1306-GB

Description

The IncEncoder function block counts the pulses of an incremental encoder connected locally to the PS4-300.



Refer to AWB 2700-1311 GB for information on the connection and engineering of the incremental encoder.

The following functions are possible:

Initiate reference approach

Define type of reference

Specify the counter value at zero point (home position)

Scan counter actual value

The Enable input operand starts the evaluation of the incremental encoder.

The SetRefValue operand allows the counter start value RefValue to be transferred irrespective of the encoder enable signal. This input is only evaluated with reference mode 0.

Define the type of calibration and reference procedure via the RefMode. 0 initialises the start value of the counter with RevValue with each edge on SetRefValue, irrespective of a hardware signal. 1 initialises the counter with the first pass via the reference point. The subsequent passes are no longer registered. 2, on the other hand, calibrates the counter with every pass over the reference point.

Before counting can be started, the so-called reference step (calibration) must be carried out. This procedure is for moving to the home position that is the basis for moving to other positions during operation.

For this define the required type of calibration (RefMode) as well as the counter start value for the home position (RefValue) and start the reference procedure with the Enable input set.



The counter can only be calibrated via the hardware signal if there is a rising edge at input I0.2 when at the same time there is a +24V voltage signal present on input I0.3 of the PS4-300 (see also section Connecting incremental encoders in the manual Hardware and Engineering the PS4-300, AWB2700-1311-GB).

After the calibration, the State output will show 1. The current counter value can then be scanned via the CurrentValue output if the Enable input is set.
A falling edge at the Enable input stops counting and resets the function block outputs.



The function block must only be instantiated once.

07/01 AWB2700-1306-GB

Example

```
PROGRAM IncEncTest

VAR
   IncEncChannel:            IncEncoder ;
         ActualPosition :    UDINT ;
         ErrorIncEnc:        UINT ;
END_VAR

   CAL IncEncChannel (
         Enable :=1,         (* Enable Incremental Encoder *)
         SetRefValue :=0,
         RefMode :=1,        (* Reference step once via Hardware *)
         RefValue :=10000    (* Zero point with 10,000 increments *)
         |
          :=State,
          :=Error,
          :=CurrentValue
   )

   (* Errors? *)
   ld    IncEncChannel.Error
   st    ErrorIncEnc
   eq    0
   jmpcn ERROR

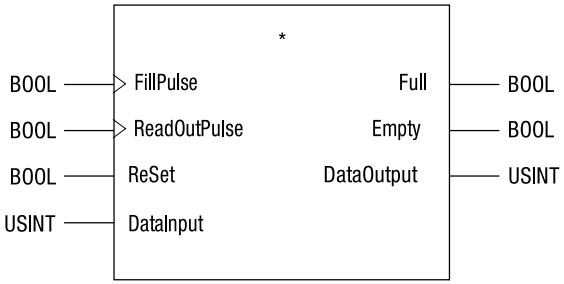
   (* Encoder calibrated? *)
   ld    IncEncChannel.State
   eq    1
   jmpcn NO_REF

   (* Encoder calibrated; counter values are permissible *)
   ld    IncEncChannel.CurrentValue
   st    ActualPosition
   jmp   EOP

NO_REF:
   (* More program code *)
ERROR:
   (* Error handling *)
EOP:
   (* End Of Program *)

END_PROGRAM
```

LifoBx
8-Bit LastIn-FirstOut Register (Stack Register)



Function block prototype

Meaning of operands

FillPulse	Fill pulse
ReadOutPulse	Read pulse
ReSet	Reset
DataInput	Data input
Full	Register full
Empty	Register empty
DataOutput	Data output

Description

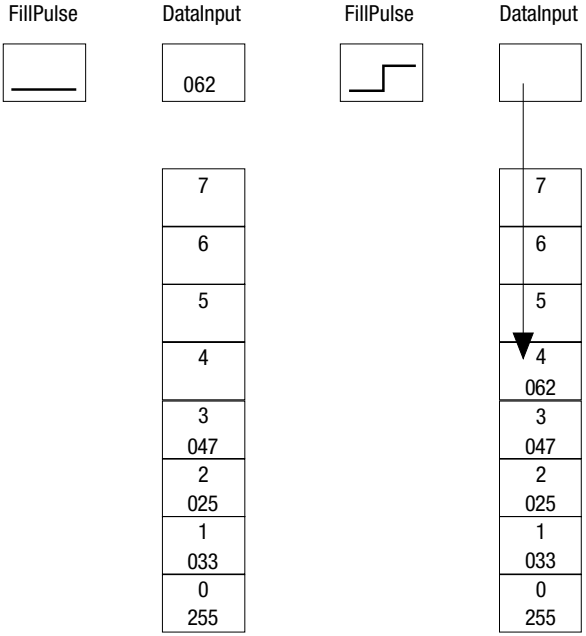
There are 128 function block prototypes. The following names can be used for *:

LifoB1
LifoB2
.
.
LifoB128

07/01 AWB2700-1306-GB

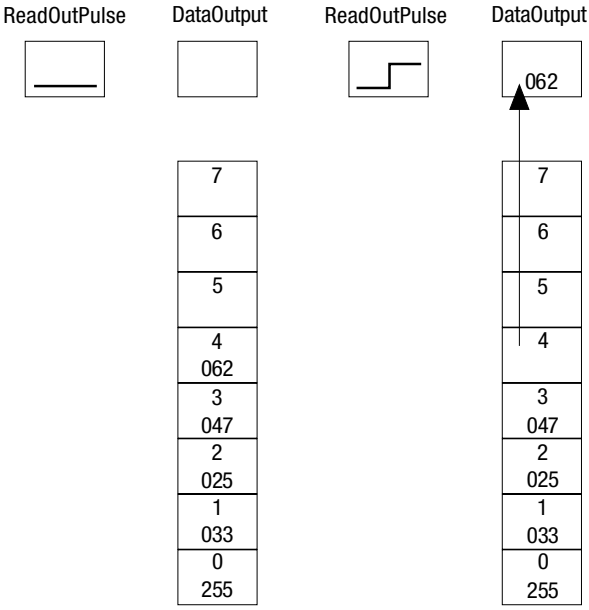
The numbers 1 ... 128 indicate the register length – the number of 8-bit register fields – of the individual prototypes.

A rising edge on the FillPulse operand transfers the value at the DataInput operand to the lowest free register field of the stack register. The stack register can be filled with every FillPulse signal until all register fields are assigned. When the Full output is 1, the stack register is full.



Contents of the stack LifoB8 register before and after a fill pulse

A rising edge on the ReadOutPulse operand reads the register field last written and transfers it to the DataOutput operand. Each further ReadOutPulse signal initiates the transfer of the next lowest elements of the register field to the DataOutput operand. When the Empty output is 1, this indicates that the last register field has been read and the register is empty.



Contents of the stack LifoB8 register before and after a readout pulse

If the ReSet operand is set to 1, this clears the contents of the entire stack register.

07/01 AWB2700-1306-GB

Example: Stack register with LifoB64

```
PROGRAM lifo8_1

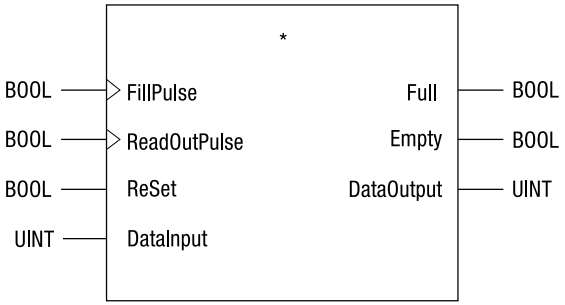
VAR
   StackRegister_64_byte : LifoB64;
   Pulse1 : BOOL;
   Pulse2 : BOOL;
   Reset : BOOL;
   StackRegisterData : USINT;
   Full : BOOL;
   Empty : BOOL;
   Outputdata : USINT;
END_VAR

   .
   .
   CAL StackRegister_64_byte(FillPulse := Pulse1,
                             ReadOutPulse := Pulse2,
                             ReSet := Reset,
                             DataInput := StackRegisterData)

   LD    StackRegister_64_byte.Full
   ST    Full
   LD    StackRegister_64_byte.Empty
   ST    Empty
   LD    StackRegister_64_byte.DataOutput
   ST    Outputdata
    .
   .

END_PROGRAM
```

LifoWx
16-Bit LastIn-FirstOut Register
(Stack Register)



Function block prototype

Meaning of operands

FillPulse	Fill pulse
ReadOutPulse	Read pulse
ReSet	Reset
DataInput	Data input
Full	Register full
Empty	Register empty
DataOutput	Data output

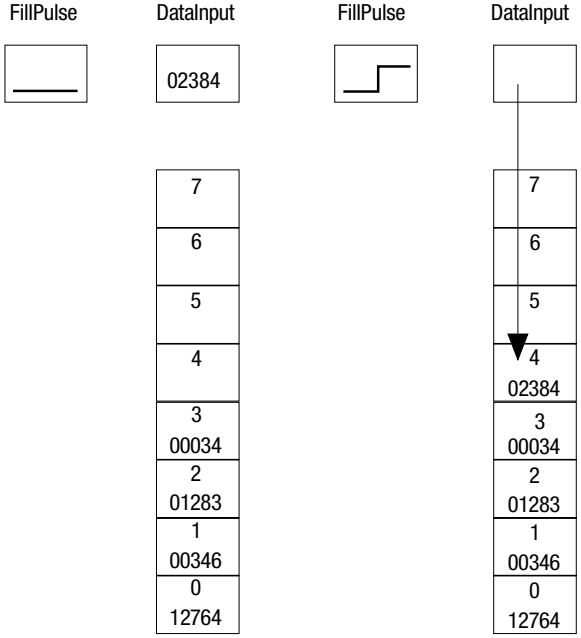
Description
There are 128 function block prototypes. The following names can be used for *:

LifoW1
LifoW2
.
.
LifoW128

07/01 AWB2700-1306-GB

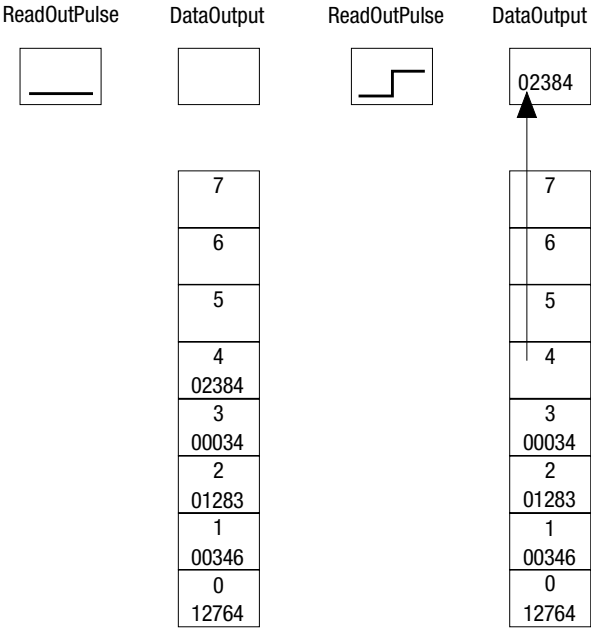
The numbers 1 ... 128 indicate the register length – the number of 16-bit register fields – of the individual prototypes.

A rising edge on the FillPulse operand transfers the value at the DataInput operand to the lowest free register field of the stack register. The stack register can be filled with every FillPulse signal until all register fields are assigned. When the Full output is 1, the stack register is full.



Contents of the LifoW8 stack register before and after a fill pulse

A rising edge on the ReadOutPulse operand reads the register field last written and transfers it to the DataOutput operand. Each further ReadOutPulse signal initiates the transfer of the next lowest elements of the register field to the DataOutput operand. When the Empty output is 1, this indicates that the last register field has been read and the register is empty.



Contents of the stack LifoW8 register before and after a readout pulse

If the ReSet operand is set to 1, this clears the contents of the entire stack register.

07/01 AWB2700-1306-GB

Example: Stack register with LifoW32

```
PROGRAM lifo16_1

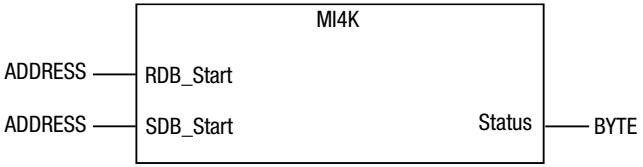
VAR
   StackRegister_32_byte : LifoW32;
   Pulse1 : BOOL;
   Pulse2 : BOOL;
   Reset : BOOL;
   StackData : UINT;
   Full : BOOL;
   Empty : BOOL;
   Outputdata : UINT;
END_VAR

   .
   .
   CAL Stackregister_32_word(FillPulse := Pulse1,
                             ReadOutPulse := Pulse2,
                             ReSet := Reset,
                             DataInput := StackData)
   LD    Stackregister_32_word.Full
   ST    Full
   LD    Stackregister_32_word.Empty
   ST    Empty
   LD    Stackregister_32_word.DataOutput
   ST    Outputdata
    .
   .

END_PROGRAM
```

PS4-200
PS4-300

MI4K
MI4 Communication Function Block



Function block prototype

Meaning of operands

RDB_Start	Start address of the Suconet data exchange interface input buffer (RDB) e.g. MI4 as slave 1 on line 1 &%RDB1.1.0.0
SDB_Start	Start address of the Suconet data exchange interface output buffer (SDB) e.g. MI4 as slave 1 on line 1 &%SDB1.1.0.0
Status	MI 4 communication status

Description

The MI4K function block is used for communication between the MI4 series visualisation systems and the PLC via Suconet K.

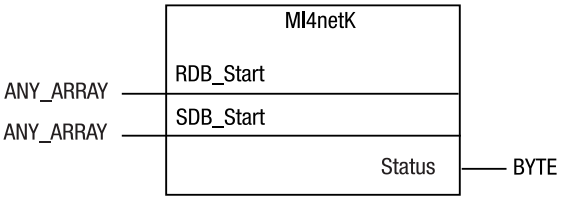
Status returns information and error codes for the communication:

01h: PLC is sending a data packet to the MI4
02h: PLC is receiving a data packet from the MI4
04h: No data request
C0h: Error in input parameters
80h: MI4 has sent an invalid operation code/ communication error
81h: Marker area exceeded/telegram length error when reading markers from the PLC
82h: Marker area exceeded/telegram length error when writing markers to the PLC

07/01 AWB2700-1306-GB

Please refer to the manual “Communication MI4-PLC” (AWB-C27-1303-GB) for a detailed description of this function block.

MI4netK
MI4 and MV4 Communication Function Block



Function block prototype

Meaning of operands

RDB_Start	Suconet K data exchange interface input buffer (RDB), e.g. MI 4 as slave 1 on line 1 with data buffer 20/20: ARRAY [1..20] OF BYTE AT %RDB1.1.0.0
SDB_Start	Suconet K data exchange interface output buffer (SDB), e.g. MI 4 as slave 1 on line 1 with data buffer 20/20: ARRAY [1..20] OF BYTE AT %SDB1.1.0.0
Status	MI 4 communication status

Description

The MI4netK function block is used for communication between the MI4 and MV4 series of visualisation systems and the PLC via Suconet K.

Status returns information and error codes for the communication:

01h: PLC is sending a data packet to the MI4
02h: PLC is receiving a data packet from the MI4
04h: No data request
C0h: Error in input parameters
80h: MI4 has sent an invalid operation code/ communication error
81h: Marker area exceeded/telegram length error when reading markers from the PLC

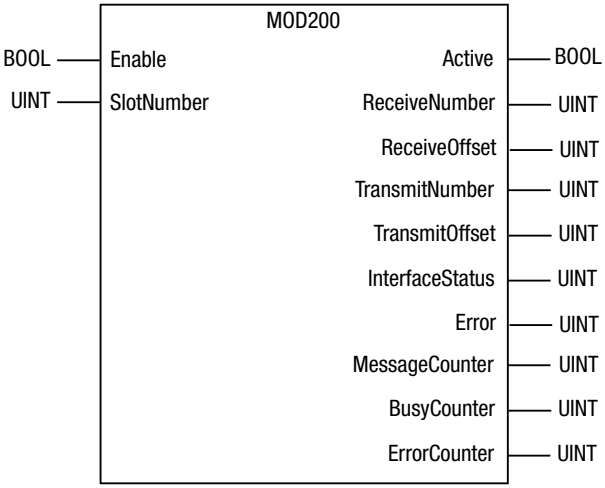
07/01 AWB2700-1306-GB

82h: Marker area exceeded/telegram length error when writing markers to the PLC

Please refer to the manuals Communication MI4 - PLC (AWB-C2700-1303 GB) and Communication MV4-PLC (AWB-C2700-349-GB) for a detailed description of this function block.

PS416

MOD200
MODBUS/JBUS Serial Communication Function Block



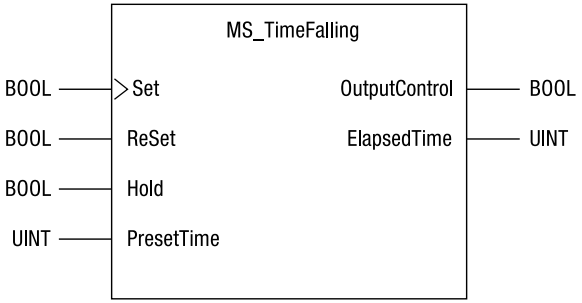
Function block prototype

Description:
The MOD200 function block is used to handle serial communication via the MOD-BUS/JBUS card PS416-MOD-200.

Please refer to the manual “Hardware and Engineering, PS-MOD416-200-MOD-BUS/JBUS Slave” (AWB-EM 27-1244-GB) for a description of this function block and some application notes.

07/01 AWB2700-1306-GB

MS_TimeFalling
Off-Delay Timer, Milliseconds

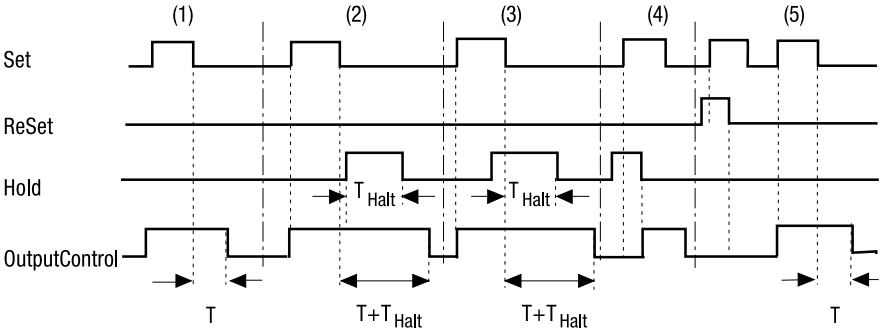


Function block prototype

Meaning of operands

Set	Start condition, rising edge
ReSet	Reset condition
Hold	Time interruption
PresetTime	Time setpoint in milliseconds
OutputControl	Control output
ElapsedTime	Time actual value in milliseconds

Description



Time diagram

A rising edge on the Set input transfers the PresetTime value as delay time T in milliseconds to the timer. The OutputControl output is set to 1. The time is started when the Set input is reset to 0. The OutputControl output is reset to 0 once the delay time has elapsed, i.e. it returns to 0 (1). The ElapsedTime output shows the current time value in milliseconds. If the status 0 at the Set input is shorter than T, the OutputControl output stays at 1.

Elapsing of the delay time can be stopped by setting the Hold input to 1. The held delay time then continues to run when the Hold input is reset to 0. The off-delay time is therefore lengthened by the duration of the Hold signal, i.e. $T+T_{HALT}$ (2).

If the Hold input is 1 when the rising edge occurs on the Set input, the start of the PresetTime value will be delayed further by the duration of the 1 signal on the Hold input. The OutputControl (4) output also has the delayed status 1.

If the Set input is reset from 1 to 0 while the Hold input is set to 1, the time is started if Hold is then

07/01 AWB2700-1306-GB

reset to 0. The off-delay time is thus lengthened by the duration of the Hold signal, i.e. $T+T_{HALT}$ (3).

If the Set input switches to 1 and then back to 0 whilst the Hold input is set to 1 the OutputControl output will remain 0.

The timer is reset when the ReSet input is 1. A rising edge on the Set input is only recognised as a start when the ReSet has changed from 1 to 0 (5).

Example: Off-delay, 25 millisecond

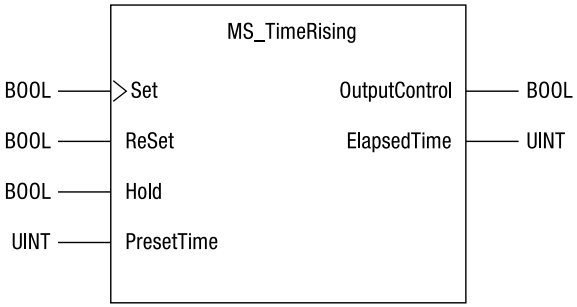
```
PROGRAM time4

VAR
  Timer4 : S_TimeFalling;
  Start AT %I0.0.0.0.0 : BOOL;
  Halt AT %I0.0.0.0.1 : BOOL;
  TimeValue4 : UINT := 25;
   (* TimeValue4 = 25 milliseconds *)
   Output4 AT %Q0.0.0.0.0 : BOOL;
END_VAR

  CAL Timer4(Set := Start,
          Hold := Halt,
          PresetTime := TimeValue4)
  LD Timer4.OutputControl
  ST Output4

END_PROGRAM
```

MS_TimeRising
On-Delay Timer, Milliseconds



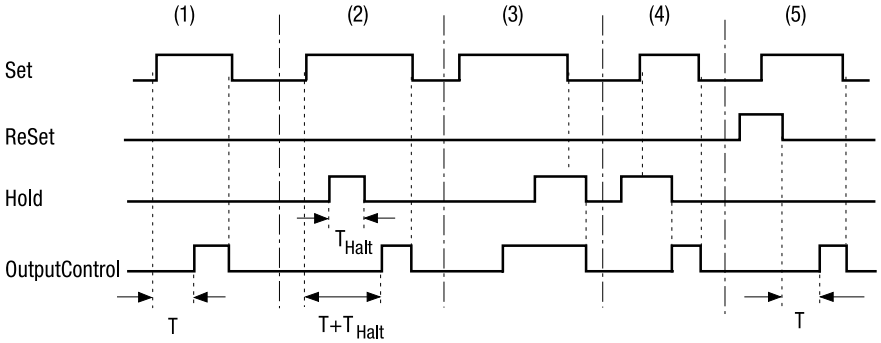
Function block prototype

Meaning of operands

Set	Start condition, rising edge
ReSet	Reset condition
Hold	Time interruption
PresetTime	Time setpoint in milliseconds
OutputControl	Control output
ElapsedTime	Time actual value in milliseconds

07/01 AWB2700-1306-GB

Description



Time diagram

A rising edge on the Set input transfers the PresetTime value as a delay time T in milliseconds to the time pulse generator and starts it. The OutputControl output switches to 1 after the delay time has elapsed and keeps this status until the Set input is reset to 0 (1). The ElapsedTime output shows the current time value in milliseconds. If the status 1 at the Set input is shorter than T, the OutputControl output stays at 0.

Elapsing of the delay time can be stopped by setting the Hold input to 1. The held delay time then continues to run when the Hold input is reset to 0. The delay time is therefore lengthened by the duration of the Hold signal, i.e. $T+T_{HOLD}$ (2).

If the Hold input is 1 when the rising edge occurs on the Set input, the start of the time delay will be delayed further by the duration of the 1 signal on the Hold input (4). If the Set input is reset from 1 to 0 after the time has elapsed, and while the Hold input is set to 1, OutputControl output returns to 0 when the Hold input is reset to 0 (3).

The timer is reset when the ReSet input is 1. The time cannot be started when the ReSet input is 1. A rising edge on the Set input is only recognised as a start when the ReSet has changed from 1 to 0 (5).

Example: On-delay, 7 milliseconds

```
PROGRAM time3

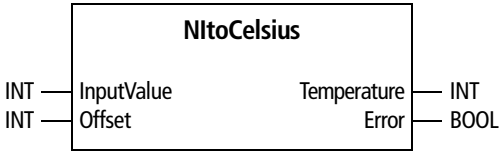
VAR
  Timer3 : MS_TimeRising;
  Start AT %I0.0.0.0.0 : BOOL;
  Halt AT %I0.0.0.0.1 : BOOL;
  Reset AT %I0.0.0.0.2 : BOOL;
  TimeValue3: UINT := 7;
           (* TimeValue3 = 7 milliseconds *)
  ActTime3 AT %QW0.0.0.2 : BOOL;
  Output3 AT %Q0.0.0.0.0 : BOOL;
END_VAR

  CAL Timer3(Set := Start,
           ReSet := Reset,
           Hold := Halt,
           PresetTime := TimeValue3)
  LD Timer3.OutputControl
  ST Output3
  LD Timer3.ElapsedTime
  ST ActTime3

END_PROGRAM
```


07/01 AWB2700-1306-GB

NItoCelsius
Linearisation of NI1000 Values



Function block prototype

Meaning of operands

InputValue	Value of resistance thermometer The input can accept the following values including offset (InputValue + Offset) depending on the resistance thermometer: 695 (= –60.0 °C) to 1500 (= 82.5 °C)
Offset	Offset in ohms which is added to the InputValue, e.g. to take account of the resistance of the cable
Temperature	Limits in °C –600 (= –60.0 °C) to 825 (= 82.5 °C)
Error	Status = 1 when InputValue rises above or falls below the indicated limits. When in the normal range, the error bit is set to 0.

Description

The function block is used in conjunction with the NI1000 input of the PS4-271-MM1.

It linearises the converted value (0 to 1500) of a NI1000 resistance into a temperature value. The process of linearisation conforms to DIN IEC 751 for NI1000 resistors. Since the value stated in the tables is for the NI100 resistor necessary to multiply the value for the NI1000 resistor by 10.

Example of linearisation

```
PROGRAM boiler_temperature
VAR
  Temperature_sensed AT %IAW0.0.0.8: INT;
  Offset:              2;
  Temperature_value:   INT;
  Error:               BOOL;
END_VAR

CAL NItoCelsius     (InputValue := Temperature_sensed,
                    Offset   := Offset
                    |
                    Error    := Error)
LD    NItoCelsius.Temperature
ST    Temperature_value

END_PROGRAM
```


07/01 AWB2700-1306-GB

NItoFahrenheit
Linearisation of NI1000 Values



Function block prototype

Meaning of operands

InputValue	Value of resistance thermometer The input can accept the following values including offset (InputValue + Offset) depending on the resistance thermometer: 695 (= –76.0 °F) to 1500 (= 180.6 °F)
Offset	Offset in ohms which is added to the InputValue, e.g. to take account of the resistance of the cable
Temperature	Limits in °F –760 (= –76.0 °F) to 1806 (= 180.6 °F)
Error	Status = 1 when InputValue rises above or falls below the indicated limits. When in the normal range, the error bit is set to 0.

Description

The function block is used in conjunction with the NI1000 input of the PS4-271-MM1.

It linearises the converted value (0 to 1500) of a NI1000 resistance into a temperature value. The process of linearisation conforms to DIN IEC 751 for NI1000 resistors. Since the value stated in the tables is for the NI100 resistor necessary to multiply the value for the NI1000 resistor by 10.

Example of linearisation

```
PROGRAM boiler_temperature
VAR
  Temperature_sensed AT %IAW0.0.0.8: INT;
  Offset:              2;
  Temperature_value:   INT;
  Error:               BOOL;
END_VAR

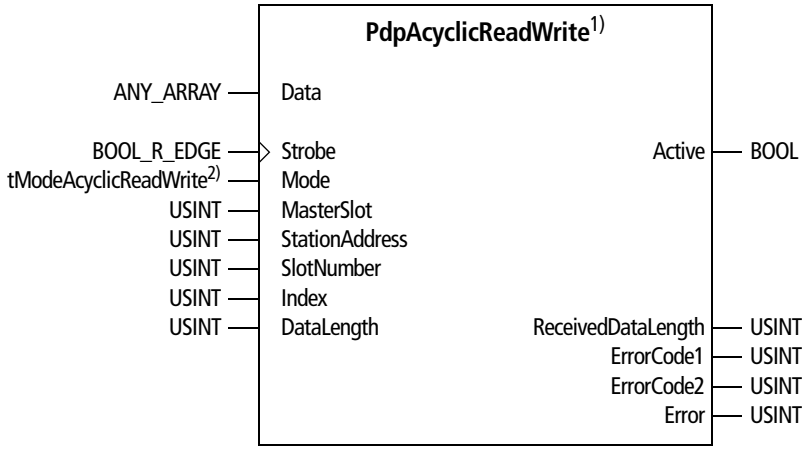
CAL NItoFahrenheit  (InputValue := Temperature_sensed,
                     Offset   := Offset
                     |
                     Error    := Error)
LD    NItoFahrenheit.Temperature
ST    Temperature_value

END_PROGRAM
```


07/01 AWB2700-1306-GB

PS4-300
PS416

PdpAcyclicReadWrite
PROFIBUS-DPV1 Acyclical Data Exchange with DPV1 Slaves; Read or Write



1) Can be used with Sucosoft S40 Version 4.2 upwards and DP master firmware version 2.0 upwards.
2) Manufacturer-defined data type; Read, Write permissible

Function block prototype

Meaning of operands

Data	Is the start address of a byte array that is – destination in Read mode and – source in Write mode. Value range 1 to 240 bytes The slave determines the data content (see Slave documentation).
Strobe	Initiate a Read or Write job
Mode	Read or Write job
MasterSlot	Slot number of the DP master PS416: value range 4 to 19 (permissible slot) PS4-300: value range 1, 2 (local expansion)

StationAddress	Network address of DP slave Value range 0 to 126
SlotNumber	Slot number in DP slave (see slave documentation) Value range 0 to 254
Index	Index in DP slave (see slave documentation) Value range 0 to 254
DataLength	Number of read/write data bytes The number of read/write data bytes depends on the DPV1 slave (see relevant documentation). Read mode: value range 0 to 240 If the number of read data bytes is unknown, specify the maximum value of 240. The actual number of bytes read is indicated in ReceivedDataLength. Write Mode: value range 0 to 240 The number of bytes to be sent must be specified exactly, otherwise this will output an error.
Active	Status of function block processing 1: Job accepted 1 -> 0: Job completed
ReceivedData Length	Read mode: actual number of bytes read Write mode: nothing indicated, always 0
Error	Error messages/Error code 0: no error >0: error = 54: ErrorCode1 and ErrorCode2 must be evaluated
ErrorCode1	DPV1 Error code1: Only evaluate, if Error = 54
ErrorCode2	Error code: Only evaluate, if Error = 54 Error code 2 is slave specific (see Slave documentation).

07/01 AWB2700-1306-GB

Description



Only one function block of all PROFIBUS-DP function blocks currently available, i.e. PdpStationDiag, PdpFreezeSync and PdpAcyclicReadWrite, should show Active = 1. If this condition is not fulfilled, an error code 215 is output by the PdpAcyclicReadWrite function block and the error code 15 by PdpStationDiag and PdpFreezeSync function blocks. The PROFIBUS-DP function blocks should only be instantiated once for each DP master in order to avoid errors.



The PS416-NET-440 and LE4-504-BS1 DP masters support acyclical data exchange according to DPV1 from firmware version 2.0 upwards.

Use the CFG-DP Configurator in Configuration mode to check the firmware version on your DP master (see also online documentation for the CFG-DP V2.0 Configurator, AWB-C2700-1336-GB).

Choose menu item Online → Firmware Reset. This will open the Firmware/Reset window. This window displays the name and firmware version currently used.

To load a new firmware version on your DP master, select menu item Online → Firmware Download. Select the new firmware from the available firmware files and load into the device via Download.

The PdpAcyclicReadWrite function block allows you to request (Read) and transfer (Write) data acyclically from and to a DPV1 slave. For this cyclical data exchange must be present on the slave concerned. The characteristics of the slave concerned determine the number of Read/Write data bytes, the relevant SlotNumber and index possible. Refer to the slave documentation for this information.

A rising edge at the Strobe input will activate the function block. The Active output will then be set to 1. After the plausibility check of the input values of the function block and the check of the operating state of the DP master, the entered mode (read/write) is transferred to the master. The master and the addressed slave accept the parameters, check them and execute the read or write job accordingly. The Active output is reset to 0, when the command has been successfully completed or an error has occurred.

Error output
The following errors (error code) are detected by the function block (FB) and indicated at the Error output:

2 No resources are available in the slave for processing the job (internal slave error).
3 Master has not activated DPV1 mode for this slave. Check DP configuration.
9 Invalid response (internal slave error).
17 No response from this StationAddress. Invalid StationAddress, slave not active or bus not active.

07/01 AWB2700-1306-GB

18 General bus error. Check bus cable, check master, check DP address or High Station address of other masters in the configuration.
25 Non-understandable response. Slave does not fulfill the DPV1 standard.
54 Invalid response. Evaluate ErrorCode1 and ErrorCode2.
129 DPV1 communication not configured and activated, or slave address missing.
130 DPV1 communication disabled, the response of a previously addressed slave is invalid.
131 A job is still active (internal FB error).
132 Parameter and data error (internal FB error).
133 Parameter error. Mode, StationAdddress, SlotNumber or Index invalid.
154 Unknown command (internal FB error).
201 The FB is faulty (internal FB error).
202 The entered command cannot be executed (internal FB error).
203 The execution of the command was incorrect (internal FB error).
204 The specified MasterSlot number is invalid.
207 No DP master defined in the topology configuration for the slot specified, or no card present.
208 The DP master in the specified MasterSlot does not contain the permissible firmware.
209 The DP master is not ready.
210 The transferred command is not valid (internal FB error).
213 The FB was unable to access the DualPortRam of the DP master (internal FB error)
215 The slot is already taken by another FB; the FB was not called in sequence.
216 The array transferred on the Data input is less than the value entered at the Length input.
217 The value range of StationAddress was exceeded.

218 The value range of SlotNumber exceeded.
219 The value range of Index exceeded.
220 The value range of Length exceeded.



Internal errors normally exclude the possibility of programming errors in the user program. If errors of this kind occur and cannot be rectified by updating the hardware and/or software version, contact your sales office.

ErrorCode1 output
The ErrorCode1 output is generated by the operating system of the slave and is output transparently by the function block. Refer to the relevant slave documentation for a detailed description of errors.

The signal at the ErrorCode1 output is structured in such a way that the higher 4 bits indicate the error class (ErrorClass) and the lower 4 bits the error code (ErrorCode).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	1 bit	Bit 0
ErrorClass				ErrorCode			

07/01 AWB2700-1306-GB

The following table shows the meaning of the error codes at the ErrorCode1 output:

ErrorClass	Meaning	ErrorCode
0 to 9 =	Reserved	
10 =	General Slave firmware error	0 = Read error 1 = Write error 2 = Internal module error 3 to 7 = reserved 8 = Version conflict 9 = Feature not supported 10 to 15 = slave specific
11 =	Slave firmware error on access	0 = Index invalid *) 1 = Write length error *) 2 = SlotNumber invalid *) 3 = Type conflict 4 = Range invalid 5 = Status conflict 6 = Access denied 7 = Value range invalid 8 = Parameter invalid 9 = Data type invalid 10 to 15 = slave specific
12 =	Slave firmware error with Resource problems	0 = Read conflict 1 = Write conflict 2 = Resource occupied 3 = Resource not available 4 to 7 = reserved 8 to 15 = slave specific
13 to 15 =	Slave specific (see slave documentation)	

The errors marked with *) are output by the function block if the input parameters Index, SlotNumber or DataLength were not accepted by the operating system of the slave.

ErrorCode2 output
The content of the ErrorCode2 output is slave specific. Refer to the slave documentation for an explanation of the coding.

Example of PdpAcyclicReadWrite (PS416 PLC):

```
PROGRAM Acycl_RW
(*
Example for acyclical reading and writing of data record 133 in the Simocode ZWK-100-TM:
(the reading of the write result can be checked after writing)

The FB is activated with a Start pulse on the Strobe input.
Acyclical read or write operations are selected using the Mode switch.
MasterSlot = 7; Master: PS416-NET-440, Slot 7,
StationAddress = 3; Slave: Simocode-DPV1, ZWK-100-TM, Basic type 1, Address = 3
SlotNumber = 4, (Slot=4 for DPV1 acyclical Read/Write functions)
Index = 133 (data record 133, Statistic data)
DataLength = 7 (data length of data record 133)
WriteData = Byte Array, source for write data
ReadData = Byte Array, destination for read data
DP line No. = 2

Requirement:
The slave was configured as DPV1.
Cyclical data exchange is active!
*)

VAR
   Start                AT %I0.0.0.0.0      : BOOL;
   Mode                 AT %I0.0.0.0.1      : BOOL;
   ModeFlag                                 : BOOL;

(* Status byte DP master, MasterSlot 7, line 2*)
   State_DPMaster       AT %ISB0.0.7.0      : Byte;
(* Status byte line 2, slave address 3*)
   State_ADR_3          AT %ISB2.3.0.0      : Byte;

(* Cyclical data *)
(* IN *)
   IBB_Array            AT %IB2.3.0.0       : Array [0..11] of Byte;
(* OUT *)
   QB0                  AT %QB2.3.0.0       : Byte;
   QB1                  AT %QB2.3.0.1       : Byte;
   QB2                  AT %QB2.3.0.2       : Byte;
   QB3                  AT %QB2.3.0.3       : Byte;

   Value                                    : Byte:=33;
```


07/01 AWB2700-1306-GB

```
(* Acylical data *)
   FBAcyclicReadWrite                    : PDPACYCLREADWRITE;
   WriteData                             : Array[1..7]of Byte;
   ReadData                              : Array[1..7]of Byte;

(* Edge detection *)
   FBActiveHighLow                       : F_TRIG;

END_VAR

(* Status evaluation *)
   LD                    State_DPMaster
   NE                    0
   JMPC                  _StateMaster

   LD                    State_ADR_3
   NE                    0
   JMPC                  _StateSlave3

_StateMaster:
(* Evaluate *)

(*...*)

(* in event of error JMP _End *)

_StateSlave3:
(* Evaluate *)

(*...*)

(* in event of error JMP _End *)

(*******************************************************)
(* Cyclical data exchange to slave 3 in line 2 *)
   LD                    Value
   ST                    QB0

                         (*...*)

   LD                    IBB_Array[0]

                         (*...*)
```

```
(*******************************************************)
(* Acyclical data exchange *)

(* Mode selection; Mode=0 Read, Mode=1 Write *)
(* Mode selection is only permissible if FB is not active.*)
   LD                   FBAcyclicReadWrite.Active
   JMPC                 _ModeFlag

   LD                   Mode
   JMPC                 _WriteMode

_ReadMode:
   LD                   0
   ST                   ModeFlag
   JMP                  _ModeFlag

_WriteMode:
   LD                   1
   ST                   ModeFlag

_ModeFlag:
   LD                   ModeFlag
   JMPC                 _AcyclicWrite

(*******************************************************)
(* Acyclical read data*)
_AcyclicRead:

(* FB call *)
   CAL FBAcyclicReadWrite(
                        Strobe :=Start,
                        Mode :=Read,
                        MasterSlot :=7,
                        StationAddress :=3,
                        SlotNumber :=4,
                        Index :=133,
                        DataLength :=7,
                        Data :=ReadData
                        |
                        :=Active,
                        :=ReceivedDataLength,
                        :=ErrorCode1,
                        :=ErrorCode2,
                        :=Error )
```

```
(* High-Low edge evaluation*)

   CAL FBActiveHighLow(
   CLK := FBAcyclicReadWrite.Active
   |
   := Q)

   LD                   FBActiveHighLow.Q
   JMPCN                _Next

(* Error detection *)

   LD                   FBAcyclicReadWrite.Error
   NE                   0
   JMPC                 _FBError

(* Evaluation of data record read *)

(* Operating hours, 2 bytes *)
   LD                   ReadData[1]
   LD                   ReadData[2]

(* Number of starts, 3 bytes *)
   LD                   ReadData[3]
   LD                   ReadData[4]
   LD                   ReadData[5]

(* Number of overload trips, 2 bytes *)
   LD                   ReadData[6]
   LD                   ReadData[7]

   (*...*)

   JMP                  _Next

(*******************************************************)
(* Acyclical write data*)
_AcyclicWrite:

(* Prepare WriteData, e.g. set initial values*)
(* Operating hours, 2 bytes *)
   LD                   0
   ST                   WriteData[1]        (* High byte *)
   LD                   1
   ST                   WriteData[2]        (* Low byte *)
```

```
(* Number of starts, 3 bytes *)
   LD                    0
   ST                    WriteData[3]        (* High byte *)
   LD                    0
   ST                    WriteData[4]
   LD                    2
   ST                    WriteData[5]        (* Low byte *)

(* Number of overload trips, 2 bytes *)
   LD                    0
   ST                    WriteData[6]        (* High byte *)
   LD                    3
   ST                    WriteData[7]        (* Low byte *)

(* FB call *)
   CAL FBAcyclicReadWrite(
                         Strobe :=Start,
                         Mode :=Write,
                         MasterSlot :=7,
                         StationAddress :=3,
                         SlotNumber :=4,
                         Index :=133,
                         DataLength :=7,
                         Data :=WriteData
                         |
                         :=Active,
                         :=ReceivedDataLength,
                         :=ErrorCode1,
                         :=ErrorCode2,
                         :=Error )

(* High-Low edge evaluation *)
   CAL FBActiveHighLow(
   CLK := FBAcyclicReadWrite.Active
   |
   := Q)

   LD                    FBActiveHighLow.Q
   JMPCN                 _Next

(* Error detection *)

   LD                    FBAcyclicReadWrite.Error
   NE                    0
   JMPC                  _FBError

   JMP                   _Next
```


07/01 AWB2700-1306-GB

```
(*******************************************************)
(* Error evaluation *)

_FBError:
(* Error handing of FB Error *)
   LD                   FBAcyclicReadWrite.Error
   EQ                   54
   JMPC                 _FBErrorCode1_2

   (*...*)

   JMP                  _Next

_FBErrorCode1_2:
(* Error handing FB ErrorCode1 *)
   LD                   FBAcyclicReadWrite.ErrorCode1

(*...*)

(* Error handling FB ErrorCode2 *)
   LD                   FBAcyclicReadWrite.ErrorCode2

(*...*)

   JMP                  _Next

(*...*)

_Next:

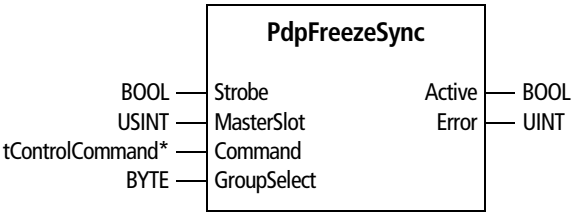
(*...*)

_End:

END_PROGRAM
```

PS4-300
PS416

PdpFreezeSync
Send PROFIBUS-DP Control Commands to DP Slaves



* Manufacturer-specific data type

Function block prototype

Meaning of operands

Strobe	Initiate a Freeze, UnFreeze, Sync or UnSync command
MasterSlot	Slot number of DP master PS416: value range 4 to 19 (permissible slot) PS4-300: value range 1, 2 (permissible LE)
Command	Permitted command are Freeze UnFreeze (factory setting) Sync UnSync
GroupSelect	Select which group or combinations of groups of slaves are to be addressed. Value range: 0 to FFHex Meaning: 0: all slaves connected to the DP line (default value) Bit(0) = 1 : Group 1 Bit(1) = 1 : Group 2 ... Bit (7) = 1 : Group 8 FFHex = all groups 1 to 8 are addressed (for this, see DP configurator)
Active	State of command execution 1: Command accepted 1->0: Command completed

07/01 AWB2700-1306-GB

Error	Error messages/error code 0 No error > 0 Error, for error code see description

Description

Task of the function block

The PdpFreezeSync function block allows you to send the PROFIBUS-DP control commands Freeze, Unfreeze, Sync and Unsync to one or several groups of DP slaves. The slaves can be organised in up to 8 groups in the DP Configurator using menu item Settings → Group Assignment.

The master sends the Freeze command to one or several groups of slaves to read the current input data, to send it on the bus and to freeze it. Any input data that changes after this point is not passed onto the bus by the slave until the next Freeze command is issued.

After every other subsequent Freeze command, the last input data is read simultaneously by the slaves, transferred to the bus and then frozen once more.

The Unfreeze command is used to deactivate Freeze for one or several groups of slaves.

The master uses the Sync command to order one or several groups of slaves to accept, freeze and output the output data on the bus. The data subsequently transferred on the bus is then not accepted by the slaves until the next Sync command is issued.

After every other Sync command the output data is accepted by the slaves simultaneously, frozen once more and output.

The Unsync command deactivates Sync command for one or several groups of slaves.

If the DP slave is to work both in Sync and in Freeze mode, both commands can be transferred in succession in any order.

Working with the function block



With all PROFIBUS-DP function blocks, i.e. PdpStationDiag, PdpFreezeSync and PdpAcyclicReadWrite, that access a specific DP master, only one function block should have the status Active = 1 at a time. The PROFIBUS-DP function blocks should only be instantiated once for each DP master in order to avoid errors. If this is not observed, the Error output will indicate error code 15.

A rising edge at the Strobe input triggers the function block which causes the master to execute the control command entered in Command. The group of slaves addressed is the one entered in GroupSelect.

07/01 AWB2700-1306-GB

When the rising edge is detected, the function block becomes active. The output Active has the value 1. After the plausibility check of the input values of the function block and the check of the operating state of the DP master, the entered control command is transferred to the master. The Active output is reset (falling edge, 1 to 0), when the command has been successfully completed or an error occurred.

The function block detects the following errors (error code):

1 The FB is faulty (internal error).
2 The entered command cannot be executed (internal error).
3 The execution of the command was incorrect (internal error).
4 The specified slot number is invalid.
7 No DP master is specified in the topology configuration for the slot number entered.
8 The DP master is not configured at the slot number specified in the topology configuration or the firmware is older than version 1.2.
9 The DP master is not ready.
10 The transferred command is not valid (internal error).
12 The user program was unable to access the DualPortRam of the DP master (internal error) (internal error)
13 The FB was unable to access the DualPortRam of the DP master (internal error)
15 The slot is already occupied by a different function block; the function blocks PdpStationDiag or PdpFreezeSync were not called in sequence, at least one of these function blocks is still active.

The bit of the group to be addressed is to be set in the input byte GroupSelect. Bit 0 is to be set for Group1, bit 1 is to be set for Group2, etc.. All bit combinations are permissible.



If no bit is set, all slaves connected to the master are addressed!

Points to remember with the execution of the Freeze and Sync commands:

The PLC user program cycle (PLC cycle) and the DP cycle work independently of each other (asynchronous). For synchronisation of the control commands Freeze or Sync to the input and output data, the following must be noted for calls of the function block and for evaluation of the Active output.

The function block may only be called once in the PLC cycle and must, if it has been activated, be called in each subsequent cycle!

Freeze command:
The function block checks that the frozen input data belongs to the specified freeze command. The function block notifies the user program that the slaves have executed the command and that the associated input data is valid by resetting (falling edge) the Active output.

The Active output is reset after no less than 5 PLC cycles. The number of cycles increases if the DP cycle is longer than the PLC cycle or there are access conflicts when data is exchanged via the DualPortRam of the DP master.

07/01 AWB2700-1306-GB

Sequence in the user program:

1. Activate FB with Freeze command, Active = 1 (1st PLC cycle)
2. Wait for falling edge at Active, (2nd to nth PLC cycle)
3. Falling edge detected at Active, Freeze was successfully executed, the input data now received by the user program belong to this Freeze cycle, next Freeze command can be started (5th to nth PLC cycle).

Sync command

The function block checks that the output data from the PLC cycle are frozen by activating the Sync command. The function block notifies the user program that the slaves have executed the Sync command by resetting (falling edge) the Active output.

The Active output is reset after no less than 5 PLC cycles. The number of cycles increases if the DP cycle is longer than the PLC cycle or there are access conflicts when data is exchanged via the DualPortRam of the DP master.

Sequence in the user program:

1. Output data processed and activate FB with Sync command, Active = 1 (1st PLC cycle)
2. Wait for falling edge of Active, (2nd to mth PLC cycle)
3. Falling edge of Active detected, Sync was successfully executed, prepare new output data, next Sync command can be triggered (5th to nth PLC cycle).

Example for Freeze and UnFreeze

```
PROGRAM docfreeze

(*
A Freeze command is sent to slaves of group 3 in cycles
of 100 ms. If the FB is active longer than 100 ms, the
Strobe is not executed for a further 100 ms etc.
The frozen values are stored for further processing after
the falling edge of Active has been detected.
Edge detection is performed by the F_TRIG function block.
Freeze mode is terminated with an UnFreeze command if 100
Freeze input bytes (FreezeByteCounter) have been logged.

Requirement:
Connected to the DP_Master are at least 3 slaves which
have the addresses 2, 4 and 125 and have at least 1 input
byte. The slaves were combined as Group 3 in the DP
configurator. In the topology configurator, the DP master
was configured at slot 7.
The DP line has the number 2.
*)

VAR
  FBPulse                   : TimeGenerator;
  FBStrobe                  : R_TRIG;
  FBFreezeSync              : PdpFreezeSync;
  FBActiveHighLowEdge       : F_TRIG;

(*DP line 2 station 2*)
  Slave2_IB0    AT %IB2.2.0.0  : BYTE;
  (*...*)
(*DP line 2 station 4*)
  Slave4_IB0    AT %IB2.4.0.0  : BYTE;
  (*...*)
(*DP line 2 station 125*)
  Slave125_IB0  AT %IB2.125.0.0: BYTE;
  (*...*)

(* FreezeValues *)
  FreezeByte1                 : BYTE;
  FreezeByte2                 : BYTE;
  FreezeByte3                 : BYTE;
  (*...*)
```


07/01 AWB2700-1306-GB

```
(* Freeze Counter *)
  FreezeByteCounter           : USINT;

(* UnFreezeFlag *)
  UnFreezeFlag              : BOOL;

END_VAR
  (*...*)

(* Pulse generator with a period duration of 100 ms *)
  CAL FBPulse(
       Set :=1,
       Period :=100
       |
       :=PulseOutput)

(* Rising edge detection of the pulse generator. *)
(* After 100 ms each time, Q is 1 for a cycle *)
  CAL FBStrobe(
       CLK :=FBPulse.PulseOutput
       |
       :=Q)

(* UnFreeze abort condition *)
  LD       UnFreezeFlag
  JMPC     _UnFreezeEnd

(* Freeze abort condition *)
  LD       FreezeByteCounter
  EQ       100
  JMPC     _UnFreeze

(* Freeze command ********************************)

(* FB call *)
  CAL FBFreezeSync(
       Strobe :=FBStrobe.Q, (* ignored if
                            Active = 1 *)
       MasterSlot :=7,      (* DP master located
                            in Slot 7*)
       Command :=Freeze,
       GroupSelect :=16#04 (* Bit 2 set =
                            Group 3 *)
       |
       :=Active,
       :=Error)
```

```
(* Falling edge evaluation*)
  CAL FBActiveHighLowEdge(
        CLK :=FBFreezeSync.Active
        |
        :=Q)
  LD    FBActiveHighLowEdge.Q
  JMPCN _NextCycle
  LD    FBFreezeSync.Error
  EQ    0
  JMPCN _FBError

(* Falling edge detected and no error occurred *)
(* Freeze InputData current after falling edge *)
(* Active and assigned to FreezeByte1 to 3 *)
(* for further processing *)

  LD   Slave2_IB0
  ST   FreezeByte1
  LD   Slave4_IB0
  ST   FreezeByte2
  LD   Slave125_IB0
  ST   FreezeByte3

  LD   FreezeByteCounter
  ADD  1
  ST   FreezeByteCounter
  (*...*)

(* Error detection *)
_FBError:
  (*...*)

(* Wait for falling edge *)
_NextCycle:
  (*...*)

  JMP  _End

_UnFreeze:
(* UnFreeze command*******************************)

(* FB call *)
  CAL FBFreezeSync(
       Strobe :=1,(* ignored if
          Active = 1        *)
       MasterSlot :=7,     (* DP master located
                           in slot 7*)
       Command :=UnFreeze,
```

```
      GroupSelect :=16#04 (* Bit 2 set =
                           Group 3 *)
      |
      :=Active,
      :=Error)

(* Falling edge detection*)
  CAL FBActiveFallingEdge(
      CLK :=FBFreezeSync.Active
      |
      :=Q)

  LD    FBActiveFallingEdge.Q
  JMPCN _End

  LD   FBFreezeSync.Error
  EQ   0
JMPCN   _FBUnFreezeError

(* UnFreeze executed *)
  LD    1
  ST    UnFreezeFlag

JMP_UnFreezeEnd

(* Error detection *)
_FBUnFreezeError:
  (*...*)

_UnFreezeEnd:

(* Freeze mode terminated. *)

  LD   Slave2_IBO
  LD   Slave4_IBO
  LD   Slave125_IBO

_End:

END_PROGRAM
```

Example for Sync and UnSync

```
PROGRAM docsync

(*
A Sync command is sent to slaves of group 3 in cycles of
100 ms. If the FB is active longer than 100 ms, the Strobe
is not executed for a further 100 ms etc.
The output values are changed after the falling edge of
Active is detected (increased by 1 in this case) and
transferred to the master.
The function block F_TRIG performs edge detection.
The Sync is terminated with an UnSync command when the
output value (SyncByteCounter) is 100.

Requirement:
Connected to the DP_Master are at least 3 slaves which
have the addresses 99, 4 and 125 and have at least 1 input
byte. The slaves were combined as Group 8 in the DP
configurator. In the topology configurator, the DP master
was configured at slot 7.
The DP line has the number 2.
*)

VAR
  FBPulse                          : TimeGenerator;
  FBStrobe                         : R_TRIG;
  FBFreezeSync                     : PdpFreezeSync;
  FBActiveHighLowEdge              : F_TRIG;

(*DP line 2 station 99 *)
  Slave99_QB0     AT %QB2.99.0.0   : BYTE;
  (*...*)
(*DP line 2 station 4, Module 2 *)
  Slave4_QB0      AT %QB2.4.2.0    : BYTE;
  (*...*)
(*DPline 2 station 125, Module 3 *)
  Slave125_QB0    AT %QB2.125.3.0  : BYTE;
  (*...*)

(* SyncValue*)
  SyncByteCounter                  : USINT;
  (*...*)
(* UnSync Flag *)
  UnSyncFlag                       : BOOL;

END_VAR
```


07/01 AWB2700-1306-GB

```
(* Example of the                                        *)
(* output of counter values to a group of              *)
(* slaves with the certainty that each counter value *)
(* is output with synchronisation by the slaves.      *)

  (*...*)

(* Pulse generator with a period duration of 100 ms *)
  CAL FBPulse(
       Set :=1,
       Period :=100
       |
       :=PulseOutput)

(* Rising edge detection of the pulse generator.      *)
(* After 100 ms each time, Q is 1 for one cycle *)

  CAL FBStrobe(
       CLK :=FBPulse.PulseOutput
       |
       :=Q)

(* UnSync abort condition *)
  LD   UnSyncFlag
  JMPC _UnSyncEnd

(* Sync abort condition *)
  LD   SyncByteCounter
  EQ   100
  JMPC _UnSync

(* Sync command ***********************************)

(* FB call *)
  CAL FBFreezeSync(
       Strobe :=FBStrobe.Q,(* ignored when
                            Active = 1 *)
       MasterSlot :=7,      (* DP master located
                            in Slot 7*)
       Command :=Sync,
       GroupSelect :=16#80  (* Bit 7 set =Group 8 *)
       |
       :=Active,
       :=Error)
```

```
(* Falling edge detection*)
  CAL FBActiveHighLowEdge(
       CLK :=FBFreezeSync.Active
       |
       :=Q)

  LD    FBActiveFallingEdge.Q
  JMPCN _NextCycle

  LD    FBFreezeSync.Error
  EQ    0
  JMPCN _FBError

(* Falling edge was detected and an error has        *)
(* occurred.*)
(* New output data can be sent on the bus           *).
(* After 100 ms, a Sync command is initiated.*)
(* The new output data is accepted by the slaves *)
(* of Group 8 and frozen.*)
(* This ensures that each value of the variable      *)
(* SyncByteCounter is sent to the slaves.            *)

  LD   SyncByteCounter
  ADD  1
  ST   SyncByteCounter
  USINT_TO_BYTE
  ST   Slave4_QB0
  ST   Slave99_QB0
  ST    Slave125_QB0

  (*...*)

(* Error evaluation *)
_FBError:
  (*...*)

(* Wait for falling edge *)
_NextCycle:
  (*...*)

  JMP  _End

_Unsync:
```


07/01 AWB2700-1306-GB

```
(* UnSync Command ********************************)

(* FB call *)
  CAL FBFreezeSync(
           Strobe :=FBStrobe.Q, (* ignored when
                                Active = 1  *)
        MasterSlot :=7,         (* DP master located
                                in Slot 7*)1
        Command :=UnSync,
        GroupSelect :=16#80 (* Bit 7 set =Group 8 *)
        |
        :=Active,
        :=Error)

(* Falling edge detection*)
  CAL FBActiveFallingEdge(
        CLK :=FBFreezeSync.Active
        |
        :=Q)

  LD    FBActiveFallingEdge.Q
  JMPCN _End

  LD    FBFreezeSync.Error
  EQ    0
  JMPCN  _FBUnSyncError

(* UnSync executed *)
  LD    1
  ST    UnSyncFlag

  JMP  _UnSyncEnd

(* Error detection *)
_FBUnSyncError:
  (*...*)

_UnSyncEnd:

(* Sync mode has been terminated. *)

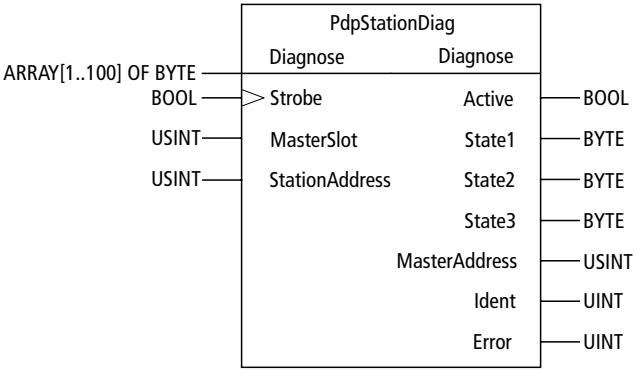
  (*...*)

_End:

END_PROGRAM
```

PS4-300
PS416

PdpStationDiag
Request Diagnostics Data from the PROFIBUS-DP Station



Function block prototype

Meaning of operands

Strobe	Initiate diagnostics job
MasterSlot	Slot number of DP master Value range for PS416: 4 to 19 decimal Value range for PS44-300: 1, 2
StationAddress	Network address of the DP slave
Active	State of job execution 1: Job accepted 1 –> 0: Job completed
State1	Standard diagnostics byte 1 of PROFIBUS-DP
State2	Standard diagnostics byte 2 of PROFIBUS-DP
State3	Standard diagnostics byte 3 of PROFIBUS-DP
MasterAddress	Address of master card, assigned to the scanned slave.
Ident	ID of the PROFIBUS-DP station
Error	Error code

07/01 AWB2700-1306-GB

Description
The PdpStationDiag enables the diagnostics data of a PROFIBUS-DP slave to be scanned.



With all PROFIBUS-DP function blocks, i.e. PdpStationDiag, PdpFreezeSync and PdpAcyclicReadWrite, that access a specific DP master, only one function block should have the status Active = 1 at a time. The PROFIBUS-DP function blocks should only be instantiated once for each DP master in order to avoid errors. If this is not observed, the Error output will indicate error code 15.

The Diagnose parameter defines an array with 100 elements of type BYTE into which the PS416-NET-440 enters the extended diagnostics data of the addressed slave.

A successful job will cause the PS416-NET-440 to return 100 bytes, irrespective of the actual diagnostics data length.



The array that you have transferred at the Diagnose input and output must be 100 byte. The relevant diagnostics bytes for the application are described in the documentation of the DP slave. Ignore the other values.

A rising edge at the Strobe input starts the diagnostics function.

Active output = 1 indicates that the job was accepted after a plausibility check on the input values. The status of the Strobe input is ignored for as long as the Active output has the value 1.

If Active changes from 1 to 0, this means that the job was completed.

The Error output indicates whether an error has occurred. Error = 0 indicates that the job was completed without errors.

0 No error
1 FB faulty (faulty function block library)
2 Diagnostics data cannot be requested
3 Error when receiving diagnostics data
4 Invalid slot number (permissible values from 4 to 19)
5 Invalid station address (permissible values from 1 to 125)
7 No card defined in the topology configuration for the slot specified.
8 No PS416-NET-440 defined in the topology configuration for the slot specified. Instead, a different card was defined.
9 The PS416-NET-440 is not ready for operation.
10 The specified station is not included in the DP configuration.
11 No diagnostics data available for the specified station.
13 The FB was unable to access the DualPortRam of the DP master (internal error)
15 The slot is already occupied by a different function block; the function blocks PdpStationDiag, PdpAyclicReadWrite or PdpFreezeSync were not called in sequence, at least one of these function blocks is still active.

When job processing is successful, the function block returns three standard PROFIBUS-DP diagnostics bytes:

State1 diagnostics byte:

Bit 0: Station not responding

Bit 1: Station not ready for data exchange

Bit 2: Incorrect station parameters

Bit 3: Station-specific diagnostics data present

Bit 4: Station has detected an unknown command

Bit 5: Station response not plausible

Bit 6: Parameter error (e.g. ID number)

Bit 7: Station parameters set by a different master

State2 diagnostics byte

Bit 0: Station parameters not set

Bit 1: Static diagnostics

Bit 2: Value permanently at 1

Bit 3: Watchdog active

Bit 4: Freeze command active

Bit 5: Sync command active

Bit 6: reserved

Bit 7: Station not included in the DP configuration.

“State3” diagnostics byte

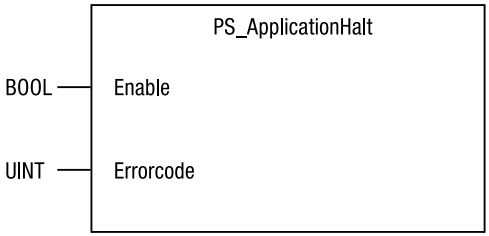
Bit 0..6: reserved

Bit 7: The extended diagnostics data of the station is longer than 100 bytes.

A detailed description of this function block is provided in the manual AWB 27-1330-GB “Hardware and Engineering, PS416-NET-440” (PROFIBUS-DP card).

PS4-300
PS416

PS_ApplicationHalt
Stop User Program



Function block prototype

Meaning of operands

Enable	Event which stops program execution
Errorcode	User-defined error number which is output following a program halt when Enable changes to 1.

Description

The PS_ApplicationHalt function block can be used to halt a user program in the PS4-300 or PS416 controller in the event of a fatal error status defined by the user. The error status is defined as an event in the program.

The status of the event is assigned to the Enable input which stops the program when its value changes to 1.

Errorcode specifies a user-defined error number which the function block outputs to Sucosoft S40 when the event occurs.

If status 1 occurs at the Enable input, the user program is stopped. The PS4-300 switches from Run to Not Ready, and the PS4-416 from Run to Ready. The outputs of both PLCs are set to zero.

After the cause of the error is rectified, the user program can only be restarted via a cold start.

07/01 AWB2700-1306-GB

You can inspect the error number in Sucosoft S40 in the Program Diagnostics dialog box of the Test and Commissioning tool.

The event which stops the user program can be for example an error message from a monitoring program or a critical process condition which is notified through a physical address.

Example: Halting a program and reporting an error

```
PROGRAM proghalt

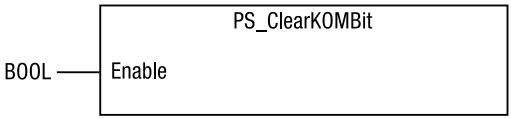
VAR
  Halt_on_Error : PS_ApplicationHalt;
  Error17 AT %I0.0.0.0.0 : BOOL;
END_VAR

  CAL Halt_on_Error(Enable := Error17,
                          Errorcode := 17)

END_PROGRAM
```

PS416

PS_ClearKOMBit
Delete the KOM Diagnostics Bit



Function block prototype

Meaning of operands

Enable	Enables function block

Description

The PS_ClearKOMBit function block is used to clear the KOM diagnostic bit. In certain situations it is necessary to temporarily suppress the setting of the KOM error bit ("Suconet K communication partner has failed"). This may be necessary, for example, for Suconet K stations which start up very slowly (PC with EPC 335 card) or when replacing a Suconet K station.

If the operating system of the PS416-CPU has not detected any other errors when the function block is called, the call will clear the error LED on the front of the PS416-CPU.

The function block should be called at the start of the user program, and in any case before checking the KOM bit.
Since the Suconet K line is checked cyclically by the operating system of the PS416-CPU at the start of each program cycle, the error LED may flash slightly if the error persists.

07/01 AWB2700-1306-GB

Example: Clearing the KOM diagnostic bit

```
PROGRAM CheckKOM

VAR
  ClearKOMBit : PS_ClearKOMBit ;
  CheckKOMBit : PS_Message;
  StartupDelay : TON ;
  KOMError    AT%Q0.0.0.0.0 : BOOL;
END_VAR

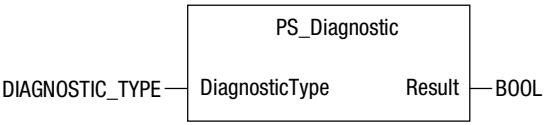
  CAL StartupDelay( IN := 1,
          PT := t#120s )

  LDN StartupDelay.Q
  CALC ClearKOMBit(Enable := 1)
  CAL CheckKOMBit(MessageType := KOM
                  |
                  KOMError := Result )

END_PROGRAM
```

PS416

PS_Diagnostic
Access the Diagnostics Flags



Function block prototype

Meaning of operands

DiagnosticType	Read diagnostic flags using the Moeller enumerated data type DIAGNOSTIC_TYPE
Result	Status 1: diagnostics bit is set Status 0: diagnostics bit is not set

Description

When calling the PS_Diagnostic function block, the diagnostics flag to be monitored must be specified in DiagnosticType.

DIAGNOSTIC_TYPE is defined as follows:

```
TYPE DIAGNOSTIC_TYPE:
                  DBM,
                  DBC,
                  DAK) := DBM;
END_TYPE
```

DBM indicates the charge status of the backup battery on the CPU. DBC indicates the charge status of the memory card battery. The DAK flag indicates whether all cards defined in the device configuration have been installed. Status 1 for the individual data elements has the following meaning:

DBM	Failure of the CPU battery
DBC	Failure of the memory card battery
DAK	Card missing

07/01 AWB2700-1306-GB

Example: checking battery state

```
PROGRAM battery

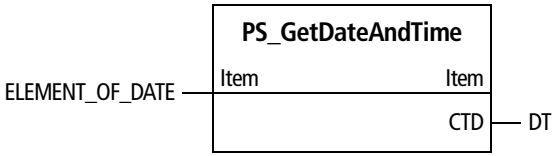
VAR
  CPU_Bat_Test : PS_Diagnostic;
  MC_Bat_Test : PS_Diagnostic;
  CPU_Bat_Empty : BOOL;
  MC_Bat_Empty : BOOL;
END_VAR

  CAL CPU_Bat_Test(DiagnosticType := DBM)
  CAL MC_Bat_Test(DiagnosticType := DBC)
  LD CPU_Bat_Test.Result
  ST CPU_Bat_Empty
  LD MC_Bat_Test.Result
  ST MC_Bat_Empty

END_PROGRAM
```

PS4-300
PS416

PS_GetDateAndTime
Scan Clock Elements



Function block prototype

Meaning of operands

Item — Scan of real-time clock via Moeller structure ELEMENT_OF_DATE

CDT — Returns the current date and time from the CPU in the format of the DT data type

Description

The real-time clock of the PLC can be read by the PLC program so that the program can react to particular time-related events and execute time-dependent routines (e.g. midday signal or similar).

Item is a VAR_IN_OUT data element. When the function block is invoked, the date elements are automatically stored in the structure. The structure is defined as follows:

```
TYPE ELEMENTE_OF_DATE:
       STRUCT
          Millisecond : UINT;
          Second : USINT;
          Minute : USINT;
          Hour : USINT;
          Day : USINT;
          Month : USINT;
          Year : USINT;
          WeekDay : USINT;
       END_STRUCT;
END_TYPE
```


07/01 AWB2700-1306-GB

Example: Reading time and date from the PLC

```
PROGRAM clock

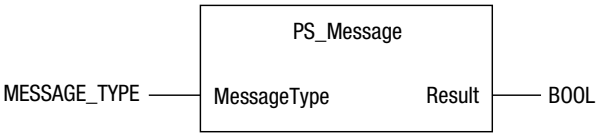
VAR
  RealtimeClock : PS_GetDateAndTime;
  Date1 : ELEMENT_OF_DATE
  Year : USINT;
  Month : USINT;
  Day : USINT;
  DateTime : DT
END_VAR

  CAL RealtimeClock(Item := Date)
  LD Date1.Year
  ST Year
  LD Date1.Month
  ST Month
  LD Date1.Day
  ST Day
  LD Date1.CDT
  ST DateTime

END_PROGRAM
```

PS416

PS_Message
Scan Cycle Status



Function block prototype

Meaning of operands

MessageType	Read the cycle flag via the Moeller enumerated data type MESSAGE_TYPE
Result	Status 1: the bit is set Status 0: the bit is not set

Description

The messages and error messages output from a running PLC program which can be displayed in Sucosoft S40 with the Diagnostics button in the Program Status window of the Test and Commissioning tool can be read and processed by the PLC program itself by using the PS_Message function block. For example, this can be used to carry out a particular initialisation routine when the program has just started (i.e. during the first cycle).

The Moeller enumerated data type MESSAGE_TYPE is defined as follows:

```
TYPE MESSAGE_TYPE:
              ICS,
              ISA,
              RTW,
              REC,
              DAKE,
              KOM) := ICS;
END_TYPE
```


07/01 AWB2700-1306-GB

Status 1 for the individual data elements has the following meaning:

ICS	1st cycle after cold start is running
ISA	1st cycle after warm start is running
RTW	Run Time Warning when a variable exceeds the admissible range. Does not stop the program. The variable which exceeded the admissible range is not indicated.
REC	Rest Cycle. Indicates that after a warm start the program is continuing from the position of interruption to the end of the cycle. The output states are not refreshed during the rest cycle.
DAKE	One of the configured cards is not fitted.
KOM	A communication partner (SUCOnet K or InterBus S) has failed.

Example: Reading program status

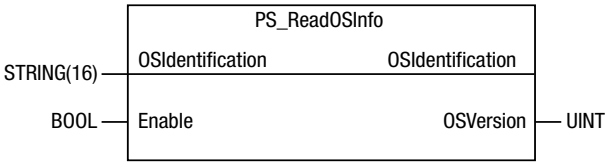
```
PROGRAM cycle

VAR
  Coldstart_Test : PS_Message;
  RestCycle_Test : PS_Message;
  Coldstart : BOOL;
  RestCycle : BOOL;
END_VAR

  CAL Coldstart_Test(MessageType := ICS)
  CAL RestCycle_Test(MessageType := REC)
  LD Coldstart_Test.Result
  ST Coldstart
  LD RestCycle_Test.Result
  ST RestCycle

END_PROGRAM
```

PS_ReadOSInfo
Read Operating System Information



Function block prototype

Meaning of operands

OSIdentification	Operating system identification text
Enable	Enable the function block
OSVersion	Operating system version number

Description

This function block is used to read the type and version number of the operating system running in the PLC.

If the Enable input is set to 1, the operating system version number is output to OSVersion and the operating system identification text is output to OSIdentification. The information is still available if the Enable input returns to 0.

The operand OSIdentification must be assigned to an IN_OUT variable of type STRING and length 16.

The content of the OSVersion must be interpreted as a 2-byte hexadecimal value.
Example: 256 Dec = 0100 hex -> Version 1.00

07/01 AWB2700-1306-GB

Example: Read operating system information

```
PROGRAM os_info

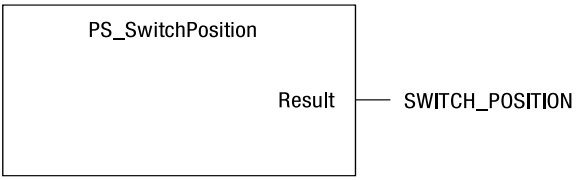
VAR
  FB_OS_Info : PS_ReadOSInfo;
  Ident : STRING(16);
  Enable AT %I0.0.0.0.0 : BOOL;
  Version: UINT;
END_VAR

  CAL FB_OS_Info(OSIdentification:= Ident,
                 Enable := Enable
                 |
                 Version := OSVersion)

END_PROGRAM
```

PS4-300
PS416

PS_SwitchPosition
Scan Switch Position



Function block prototype

Meaning of operands

Result	Check the position of the operating mode selector switch via the Moeller enumerated data type SWITCH_POSITION.

Description

The function block PS_SwitchPosition allows the PLC program to check the status of the operating mode switch, for example, to react to a change in the switch position in order to output an appropriate warning.

The data type SWITCH_POSITION is defined as follows:

```
TYPE SWITCH_POSITION:
              RUN,
              RUN_MR,
              HALT) := RUN;
END_TYPE
```


07/01 AWB2700-1306-GB

Example: Checking the position of the operating mode switch

```
PROGRAM switch
VAR
  SWITCH : PS_SwitchPosition;
  Indicate_RUN : BOOL;
  Indicate_RUN_MR : BOOL;
  Indicate_HALT : BOOL;
END_VAR

  CAL SWITCH
  LD SWITCH.Result
  EQ RUN
  JMPCN _RUN_MR
  LD 1
  ST Indicate_RUN
  LD 0
  ST Indicate_RUN_MR
  ST Indicate_HALT
  JMP _OUT

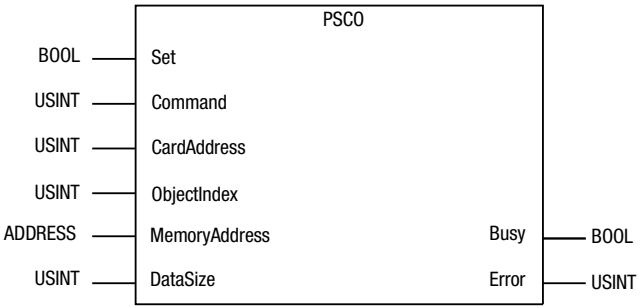
_RUN_MR :
  LD SWITCH.Result
  EQ RUN_MR
  JMPCN _HALT
  LD 1
  ST Indicate_RUN_MR
  LD 0
  ST Indicate_RUN
  ST Indicate_HALT
  JMP _OUT

_HALT :
  LD1
  ST Indicate_HALT
  LD 0
  ST Indicate_RUN
  ST Indicate_RUN_MR
_OUT:

END_PROGRAM
```

PS4-200
PS4-300

PSCO
PROFIBUS-FMS Communication Function Block



Function block prototype

Meaning of operands

Set	Initiate job
Command	Job type
CardAddress	Interface selection
ObjectIndex	Logical addressing of communication objects (variables and data arrays) of a station
MemoryAddress	Start address of source and destination range
DataSize	Size of source and destination range
Busy	Function block status
Error	Error code

Description

The PSCO function block is used to handle PROFIBUS communication via the LE4-503-BS1 local expansion module. The program on the PLC accesses the objects of the LE4 by means of a function block instance.

To find out more about possibilities and procedures in PROFIBUS communication with the LE4-503-BS1

07/01 AWB2700-1306-GB

local expansion module please refer to the manual AWB27-1264-GB for the module.



No more than one function block instance can be used per object.

A 1 at the Set input starts the job specified at the Command input. The function block stays active for as long as the Set input is 1. A 0 or FALSE at the Set input deactivates the function block. Any job being processed will thus be aborted.

The parameters are checked after the function block has been started. Incorrect parameters are indicated by a message on the Error output.

The Command parameter is used to specify the job. The following commands are available:

0 PUT OBJECT
1 GET OBJECT
2 PUT OBJECT CONTROL
3 CHECK

The individual commands are described in the corresponding sections.
The number of the LE4 module is set at the CardAddress input. The value 1 is used since only one communication LE is allowed for each PLC.

The LE4-503-BS1 supports the logical addressing of objects via the ObjectIndex parameter (see AWB27-1264-GB, Appendix: Object directory PROFIBUS features). An object cannot be accessed by several function block instances.

The MemoryAddress input defines the start address of the data block in the marker area that is defined as the data destination or source of a job.

The DataSize input defines the size of the data field, i.e. the number of data bytes. When using command 3 (CHECK) a 1 should always be entered as the data size.

The Busy output is only used in conjunction with the PUT OBJECT CONTROL command.
1 = Job being processed
0 = Job completed.

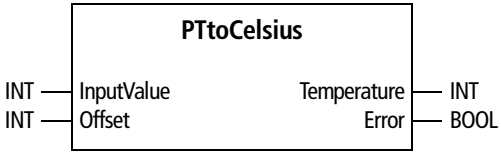
The Error output indicates the following error codes:

0 No error
1 Invalid command
2 Invalid module address (CardAddress)
3 Invalid object index
4 Start address of the data field + data field size violates range limit
5 Incorrect operand type on MemoryAddress input
6 Number data bytes (DataSize) does not match the object size
7 Incorrect access to object, e.g. PUT command to WRITE object
8 Hardware error on local expansion module
9 Access attempt of several function blocks to one object

Please refer to the documentation for the LE4-503-BS1 module (AWB27-1264-GB) for details of the commands and the PROFIBUS services.

07/01 AWB2700-1306-GB

PTtoCelsius
Linearisation of PT1000 Values



Function block prototype

Meaning of operands

InputValue	Value of resistance thermometer The input can accept the following values including offset (InputValue + Offset) depending on the resistance thermometer: 185 (= –200.0 °C) to 1500 (= 130.5 °C)
Offset	Offset in ohms which is added to the InputValue, e.g. to take account of the resistance of the cable
Temperature	Limits in °C –2000 (= –200.0 °C) to 1305 (= 130.5 °C)
Error	Status = 1 when InputValue rises above (>) or falls below (<) the indicated limits. When in the normal range, the error bit is set to 0.

Description

The function block is used in conjunction with the PT1000 input of the PS4-271-MM1.

It linearises the converted value (0 to 1500) of a PT1000 resistor into a temperature value. The method of linearisation conforms to DIN IEC 751 for PT1000 resistors. Since the value stated in the tables is for the PT100 resistor, it is necessary to multiply the value for the PT1000 resistance by 10.

Example of linearisation

```
PROGRAM boiler_temperature
VAR
  Temperature_sensed AT %IAW0.0.0.8: INT;
  Offset:                    2;
  Temperature_value:         INT;
  Error:                     BOOL;
END_VAR

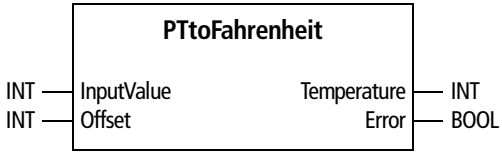
CAL PTtoCelsius    (InputValue   := Temperature_sensed,
                    Offset       := Offset
                    |
                    Error        := Error)
LD  PTtoCelsius.Temperature
ST  Temperature_value

END_PROGRAM
```


07/01 AWB2700-1306-GB

PS4-271

PTtoFahrenheit
Linearisation of PT1000 Values



Function block prototype

Meaning of operands

InputValue	Value of resistance thermometer The input can accept the following values including offset (InputValue + Offset) depending on the resistance thermometer: 185 (= –328.0 °F) to 1500 (= 266.8 °F)
Offset	Offset in ohms which is added to the InputValue, e.g. to take account of the resistance of the cable
Temperature	Limits in °F –3280 (= –328.0 °F) to 2668 (= 266.8 °F)
Error	Status = 1 when InputValue increases above (>) or falls below (<) the shown limits. When in the normal range, the error bit is set to 0.

Description

The function block is used in conjunction with the PT1000 input of the PS4-271-MM1.

It linearises the converted value (0 to 1500) of a PT1000 resistor into a temperature value. The method of linearisation conforms to DIN IEC 751 for PT1000 resistors. Since the value stated in the tables is for the PT100 resistor, it is necessary to multiply the value for the PT1000 resistance by 10.

Example of linearisation

```
PROGRAM boiler_temperature
VAR
  Temperature_sensed AT %IAW0.0.0.8: INT;
  Offset:                    2;
  Temperature_value:         INT;
  Error:                     BOOL;
END_VAR

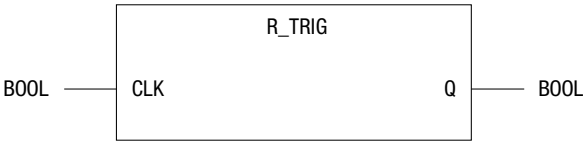
CAL PTtoFahrenheit (InputValue   := Temperature_sensed,
                    Offset       := Offset
                    |
                    Error        := Error)
LD  PTtoFahrenheit.Temperature
ST  Temperature value

END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC

R_TRIG
Edge Detection, Rising Edge



Function block prototype

Meaning of operands

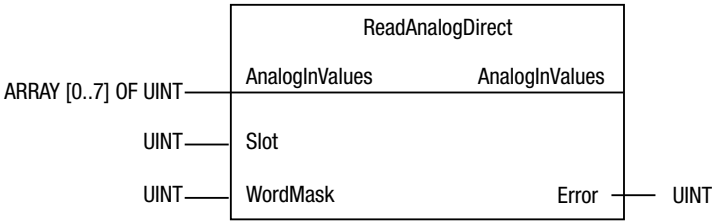
CLK	Input operand whose rising edge is detected.
Q	Output operand; indicates the rising edge of CLK.

Description

The function block R_TRIG evaluates the status of the input operand CLK. It detects the status change from 0 in a processing cycle to 1 in the subsequent cycle and indicates this on the output Q with the Boolean value 1. The output is only set to 1 during the processing cycle in which the change of the status of CLK and a rising edge was detected.

PS416

ReadAnalogDirect
Direct Reading of Analog Inputs



Function block prototype

Meaning of operands

Slot	Slot for an analog card in the basic unit (4..20)
WordMask	Bit mask specifying which channels are to be read
AnalogInValues	Bit mask specifying which channels are to be written
Error	Error code

Description:
The ReadAnalogDirect function block enables direct read operations to analog input channels in the PS416 basic unit.

The data is not read from the inputs at the start of the cycle but directly during the current cycle. This function block therefore allows a fast and direct response to events such as required in alarm routines. The maximum on/off delay of the analog cards for the controller should be taken into account when using this function block.

The Slot input specifies the slot in which the card to be read is fitted.

07/01 AWB2700-1306-GB

The WordMask input specifies which channels are to be read on the card. The bit position determines the channel:

Bit 0 – channel 0

Bit 1 – channel 1

Bit 2 – ...

This therefore enables you to specify which channels are to be read.

The AnalogInValues input is used to save the read values in the array. Only the values for the channels specified in Wordmask are updated.

The Error output indicates the following error codes:
0 No error
1 Invalid slot
2 No PS416-AIO-400 or PS416-AIN-400 in the slot
3 Incorrect channel number requested via mask

Example “Read analog data direct”

```
PROGRAM ReadAnalog
VAR
   ReadAnalog : ReadAnalogDirect;
   AnalogData : ARRAY[0..7] OF UINT;
   ReadError : UINT;
END_VAR

   CAL ReadAnalog(Slot := 11,       (* Card in Slot 11 *)
                  Wordmask := 16#5, (* Read channel no.0 and no.2 *)
                  AnalogInValues := AnalogData
                  |
                  ReadError = Error )
   LD    ReadError
   EQ    0
   JMPCN  READ_ERROR
   LD    AnalogData[0]       (* Read data Channel no.0 *)
   LD    Analogdata[2]       (* Read data Channel no.2 *)
   JMP   Read_DONE

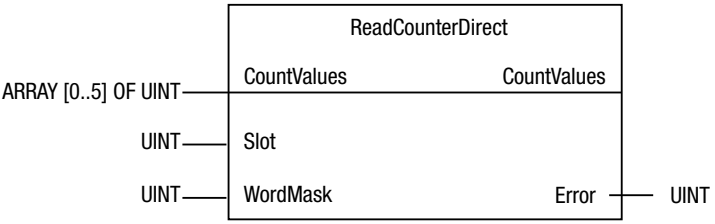
READ_ERROR:                  (* Error handling *)

READ_DONE:                   (* Further program processing *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

ReadCounterDirect
Direct Reading of Counter Inputs



Function block prototype

Meaning of operands

Slot	Slot for a counter card in the basic unit
WordMask	Bit mask specifying which channels are to be read
CountValues	Bit mask specifying which channels are to be written
Error	Error code

Description:

The ReadCounterDirect function block enables direct read operations to counter channels in the PS416 basic unit.

The data is not read from the inputs at the start of the cycle but directly during the current cycle. This function block therefore allows a fast and direct response to events such as required in alarm routines. The maximum on/off delay of the analog cards for the controller should be taken into account when using this function block.

The Slot input specifies the slot in which the PS416-CNT-200 card to be accessed is fitted.

The WordMask input specifies which channels are to be write accessed on the card. The bit position determines the channel:

Bit 0 – channel 0

Bit 1 – channel 1

Bit 2 – ...

This therefore enables you to specify which channels are to be read.

The read values are saved via the CountValue input. Only the values for the channels specified in Wordmask are updated.

The Error output indicates the following error codes:
0 No error
1 Invalid slot
2 No PS4-300-CNT-200 present on the slot
3 Incorrect channel number requested via mask

07/01 AWB2700-1306-GB

Example “Read counter data direct”

```
PROGRAM ReadCounter

VAR
   ReadCounter:   ReadCounterDirect;
   CounterData:   ARRAY[0..5] OF UINT;
   ReadError:     UINT;
END_VAR

   CAL ReadCounter(Slot := 7,               (* Card in Slot 7 *)
                   Wordmask := 16#4,        (* Read channel no. 2 *)
                   CountValues := CounterData
                   |
                   ReadError = Error )

   LD    ReadError
   EQ    0
   JMPCN READ_ERROR
   LD    CounterData[2]  (* Read data channel no.2*)
   JMP   Read_DONE

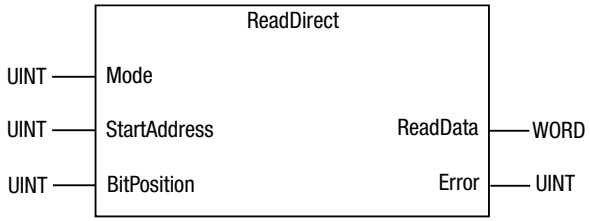
READ_ERROR:             (* Error handling *)

READ_DONE:              (* Further program processing *)

END_PROGRAM
```

PS416

ReadDirect
Direct Reading of Digital Inputs



Function block prototype

Meaning of operands

Mode	Access mode
StartAddress	Byte or word address of the digital input; Permissible range of values: Bit and byte access: 0 to 127, Word access: 0 to 126
BitPosition	Bit position with bit access Valid range: 0 to 7
ReadData	Data which is read
Error	Error message

Description:
The ReadDirect function block enables direct read operations to digital input cards in the PS 416 basic unit. The data is not read from the inputs at the start of the cycle but directly during the current cycle. This function block therefore allows a fast and direct response to events such as required in alarm routines. When using this function block, observe the maximum on/off delay of the digital I/O cards for the PS 416 system.

07/01 AWB2700-1306-GB

The Mode input defines the access type.
1 Read a single bit from an input card
2 Read a byte (8 bit) from an input card
3 Read a word (16 bit) from an input card.

The StartAddress and BitPosition inputs address an input and are assigned according to the setting of the Mode input. BitPosition is only relevant for Mode = 1. Examples:

Mode = 1 StartAddress = 4, BitPosition := 6
=> %I0.0.0.4.6,

Mode = 2 StartAddress = 2 => %IB0.0.0.2,

Mode = 3 StartAddress = 126 => %IW0.0.0.126.

The ReadData output contains the data value read from the input card. Before the output value is processed it may be necessary to carry out a type conversion with WORD_TO_xxx. However, direct access to individual bits of ReadData is also possible. Depending on the Mode the following bits are relevant:

Mode = 1 ReadData.0

Mode = 2 ReadData.0 to ReadData.7

Mode = 3 ReadData.0 to ReadData.15



Input variables declared in the main program cannot be used directly for the ReadData output since input variables can only be read whereas outputs of function blocks can only be written.

The Error output indicates the following error codes:

0 No error, ReadData valid
1 Value of Mode input invalid
2 Value of StartAddress input is too large.
3 Value of StartAddress input not an even number (only Mode = 3)
4 Value of BitPosition input invalid

Example: Directly reading a block input

```
PROGRAM ReadIO

VAR
   ReadEnable AT%I0.0.0.0.0 : BOOL;
   ReadIODirect : ReadDirect;
   ReadData : WORD;
   ReadError : UINT;
   IOPoint5_3 : BOOL;
END_VAR

   (* Direct read access to the digital
        I/O. I/O byte 5, Bit 3 is
        read. This corresponds to the variable
        declaration %I0.0.0.5.3 :BOOL;*)
   LD    ReadEnable
   CALC  ReadIODirect(Mode := 1,             (* Bit access *)
                      StartAddress := 5, (* Byte address 5 *)
                      BitPosition := 3    (* Bit position 3 *)
                      |
                      ReadData := ReadData,
                      ReadError := Error)

   LD    ReadError
   EQ    0
   JMPCN READ_ERROR
   LD    ReadData.0
   ST    IOPoint5_3
   JMP   READ_DONE

READ_ERROR:

    (* Error handling *)

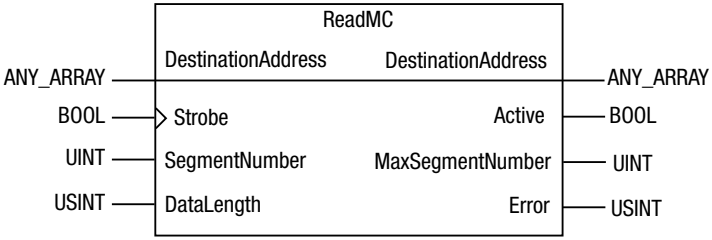
READ_DONE:

    (* Further sections of the program *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

ReadMC
Read Data from Memory Card



Function block prototype

Meaning of operands

DestinationAddress	Buffer for data to be read
Strobe	Start transfer
SegmentNumber	Segment number (0 - n)
DataLength	Number of data bytes of type USINT to be read
Active	Data read operation active
MaxSegmentNumber	Last possible segment number of the file
Error	Error message

Description

The ReadMC function block allows data to be read from the PLC's memory card (MC). Together with the WriteMC function block, important applications of this function block are data backup, process data acquisition, saving recipe data and/or saving variables before a program update.

You can store recipe data on the memory card of the PS4-300 by selecting the appropriate option when formatting the memory card.

The memory card is addressed in segments. The function block can read up to 128 bytes from the memory card.

To determine the MC file size, call the function block with DataLength set to 0. This does not read from the memory card. MaxSegmentNumber then indicates the highest segment number in the MC file.

The function block is executed asynchronously in relation to the user program cycle and can take several program cycles. The Active output is set to 1 during this time. Error-free completion of the function block is indicated by the Active output changing to 0 and by 0 at the Error output.

You can activate several function block instances simultaneously. They will be processed one after the other in the order that they were called.

Error codes at the Error output:

0 Function block execution completed without errors
1 No memory card inserted
4 SegmentNumber exceeds the value of MaxSegmentNumber
5 DataLength $>$ 128 or DestinationAddress does not have the required size
6 Start address is invalid

Error codes with PS4-200 only:

2 No access possible due to an online connection
3 No access possible since memory card is currently being accessed by WriteMC
7 No access possible since MC is currently being used as a source memory.

Error code with PS 4-300 only

8 The MC was formatted without recipe data range

Error codes with PS416 only:

10 Memory card type is not SRAM.
11 Memory card is not formatted.
12 Memory card is write-protected.
13 Not enough free memory in the memory card to generate the file RSDAT.DAT.
14 The file RSDAT.DAT does not have the required minimum size of 128 bytes.
15 Internal error during execution of the function block

The following applies to the ReadMC function block only in conjunction with the PS416:

The data is saved on the memory card in the MS-DOS file RSDAT.DAT and can be read and processed with a PCMCIA drive. The file is subdivided into 128 byte segments and its size is only limited by the capacity of the memory card used.

If a memory card is inserted and the file RSDAT.DAT is not present on the memory card when the function block is first called, it is created automatically with a default size of 512×128 byte (64 KByte), and all sectors are initialised with the value “0xFF”. Other file sizes can be created with the SetMCFileLength.

Only applicable with PS4-200: the Active output of these function blocks is always 0 in the PS4-150/-200. The current function block status can be scanned via the Error output when the function blocks are called again. If the Error output is 0, processing was carried out error-free. If the output is 3, the function block is processing.

To refresh the Error output of this function block a new edge on the Strobe input must be generated. Otherwise the error message will be kept (with 1 on Strobe) or will be set to 0 (with 0 on Strobe).

Example: Reading data from memory card

```
PROGRAM RELOAD

VAR
   StrobeReload AT %I0.0.0.0.0 :BOOL;
   ActiveReload AT %Q0.0.0.0.0 :BOOL;
   ErrorReload : USINT;
   FBReloadData : ReadMC;
   SegmentNumb : UINT;
   DataLength : USINT;
   LastSegmentNumber : UINT;
   DataArea : ARRAY[0..15] of USINT;
END_VAR

   LD 8
   ST SegmentNumber
   LD 16
   ST DataLength
   (* Read 16 bytes from segment 8
         into the array DataArea *)

   CAL FBReloadData(Strobe := StrobeReload,
                    SegmentNumber := SegmentNumb,
                    DataLength := DataLength,
                    DestinationAddress := DataArea
                    |
                    ActiveReload := Active,
                    LastSegmentNumber := MaxSegmentNumber,
                    ErrorReload := Error)

   LD ActiveReload
   JMPC CONTINUE
   LD ErrorReload
   EQ 0
   JMPCN ERRORHANDLING

   (* Continue processing read data *)
   JMP CONTINUE

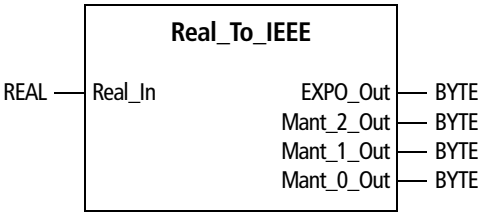
ERROR_HANDLING:
   (* Carry out error handling *)

CONTINUE:
        (* Further parts of user program *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

Real_To_IEEE
Conversion of Numbers of Data Type REAL into IEEE 754 Standard Format



Function block prototype

Meaning of operands

Real_In	Real number to be converted
EXPO_Out	IEEE floating point number: Exponent (incl. sign)
Mant_2_Out	IEEE floating number to be converted: mantissa byte 2 (incl.1 bit exponent)
Mant_1_Out	IEEE floating number: mantissa byte 1
Mant_0_Out	IEEE floating number: mantissa byte 0

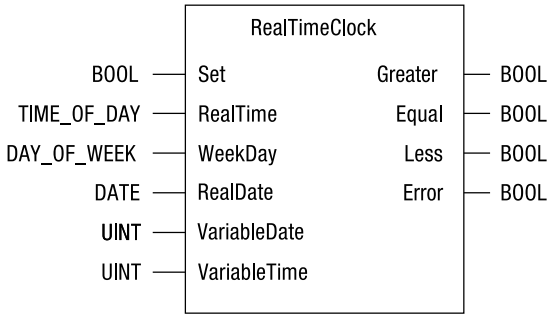
Description

The Real_to_IEEE function block converts a number of data type REAL to a floating-point number in the IEEE-754 Single Precision format. To represent the floating-point number, four bytes (32 bit) are needed. This conversion only converts the format but not the value:

EXPO_Out	Mant_2_Out	Mant_1_Out	Mant_0_Out

The IEEE format is mostly used for the transfer of floating-point numbers to other systems (e.g. via PROFIBUS).

RealTimeClock
Compare With Real-Time Clock



Function block prototype

Meaning of operands

Set	Enable signal for comparing the time of the PLC's real-time clock with the input values
RealTime	Time (hours, minutes) that is compared with the PLC's real-time clock. This is a constant that is entered during programming and is not changed while the program is running.
WeekDay	Weekday that is compared with the PLC's real-time clock. This is a constant that is entered during programming and is not changed while the program is running. DAY_OF_WEEK is Moeller data type that is defined as follows: TYPE DAY_OF_WEEK: (Sunday, Monday, Tuesday. Wednesday, Thursday, Friday, Saturday) := Monday; END_TYPE

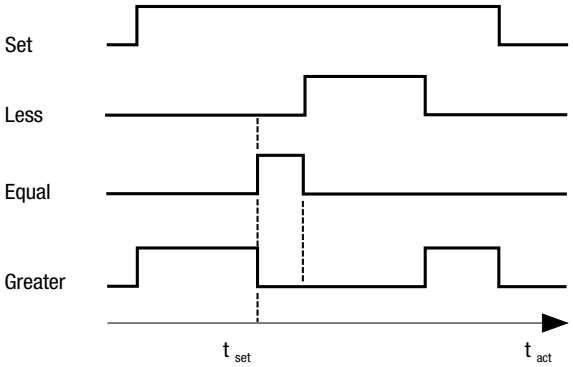
07/01 AWB2700-1306-GB

RealDate	Date (day, month and year) that is compared with the PLC's real-time clock. This is a constant that is entered during programming and is not changed while the program is running.
VariableDate	Specified day and month as variable comparison value for comparison with the status of the real-time clock. The day is entered as a binary value in the high byte, the month in the low byte.
VariableTime	Variable time for comparing with the status of the real-time clock. The hours are entered as a binary value in the high byte, the minutes in the low byte.
Greater	Status 1: the entered comparison value is greater than the status of the real-time clock.
Equal	Status 1: the entered comparison value is the same as the status of the real-time clock.
Less	Status 1: the entered comparison value is less than the status of the real-time clock.
Error	Error message 1: the values entered are impermissible

Description

The function block compares the current status of the real time clock with the specified values t_{SET}, that are entered via input operands. Two different types of comparison are possible.

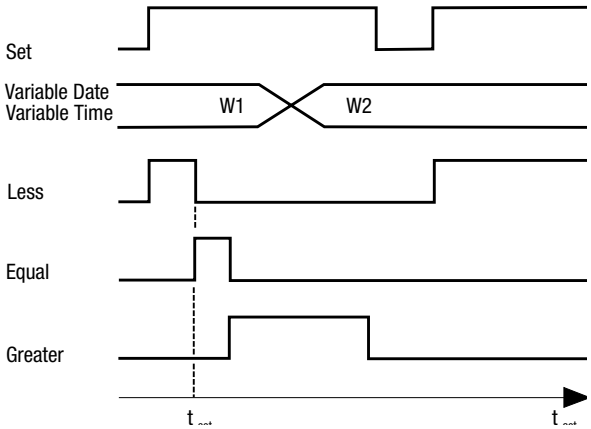
Example 1:



The function block compares the status of the running real-time clock with the values of the constants Time, WeekDay and Date, and outputs the signal Greater, Equal or Less accordingly. This comparison is only possible if the Set operand is 1 and the operands VariableDate and VariableTime were not set when the function block was invoked.

07/01 AWB2700-1306-GB

Example 2:



While the Set operand is 1, the function block compares the status of the running real-time clock with the variable values of the operand VariableDate and/or VariableTime and outputs the Greater, Equal or Less signal accordingly.

The change from Greater to Less is carried out according to the input operands entered:

with VariableTime operands this is at 24 hours,

with VariableDate operands this is on the last day of the month at 24 hours,

with VariableTime and VariableDate operands at 24 hours at the end of the year.

The function block is enabled by setting the Set input to 1. If Set is 0, the comparison is disabled and all outputs are set to 0.

Example: Time comparison

```
PROGRAM clock

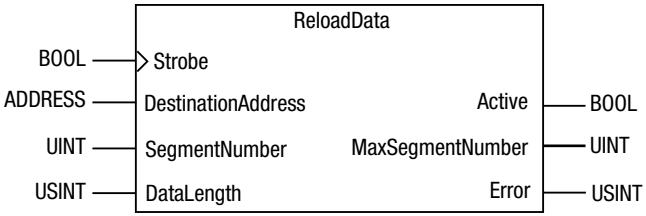
VAR
  Time1 : RealTimeClock;
  Clock_free : BOOL;
  Time2 : TIME_OF_DAY := TOD#12:00:00;
  Date1 : DATE := D#1996-12-12;
  Midday AT %Q0.0.0.0.0 : BOOL;
END_VAR

  .
  .     .
   .
  CAL Time1(Set := Clock_free,
            RealTime := Time2,
            RealDate := Date1
            WeekDay := Monday)
  LD   Time1.Equal
  ST   Midday

END_PROGRAM
```


07/01 AWB2700-1306-GB

ReloadData
Read Data from Memory Card



Function block prototype

Meaning of operands

Strobe	Start transfer
SegmentNumber	Segment number (0 - n)
DataLength	Number of array elements of type USINT to be reloaded, Value range:1 to 128
DestinationAddress	Start address of the marker area in which the ADDRESS type data should be stored
Active	Data read operation active
MaxSegmentNumber	Last possible segment number of the file
Error	Error code

Description

The ReloadData function block is used to read data from the PLC's memory card and store it in a marker area in the PLC memory.

The memory card is logically divided into 512 segments of 128 bytes each. The function block can read up to 128 bytes with one invocation.

An important application of this function block is to save the retentive marker area (the recipe range that was specified in the compiler configuration during code generation) with SaveData and to read it back again after changing the device configuration.

The Error output indicates the following error codes:

1 No memory module present
2 No access possible due to Online connection
3 No access possible due to simultaneous access via SaveData function block
4 Invalid segment number (permissible: 0 – 511)
5 Invalid data length (valid range for DataLength: 1 – 128)
6 Start address is invalid
7 Only for PS4-200: No access possible. MC is used as a source
8 Only for PS4-300: The MC was formatted without recipe data range

Only applicable with PS4-200: the Active output of these function blocks is always 0 in the PS4-150/-200. The current function block status can be scanned via the Error output when the function blocks are called again. If the Error output is 0, processing was carried out error-free. If the output is 3, the function block is processing.

To refresh the Error output of this function block a new edge on the Strobe input must be generated. Otherwise the error message will be kept (with 1 on Strobe) or will be set to 0 (with 0 on Strobe).



The SaveData function block allows data to be written to the memory card.

07/01 AWB2700-1306-GB

Example: Read data from memory card

```
PROGRAM R_CardVAR
   ReadData : ReloadData;
   Start AT %IO.0.0.0.2 : BOOL;
   Mem_start AT %MBO.0.0.100 : BYTE;
   Segment_NO : UINT;
   DataLength1 : USINT;
   ErrorMessage : USINT;
END_VAR
   CAL ReadData(Strobe := Start,
                DestinationAddress := &Mem_start,
                SegmentNumber := Segment_NR,
                DataLength := DataLength1)
   LD ReadData.Error
   ST ErrorMessage
   NE 0 (*If content not equal then error*)
   JMPC ErrorEvaluation
    .
   .
Error detection:
   .
   .
END_PROGRAM
```

IEC

RS
Flip Flop With Reset Dominant



Function block prototype

Meaning of operands

SET	Set condition
RESET1	Reset condition
Q1	Output state of the bistable element

Description

The RS function block is used to statically set a data element – the output Q1 – to the Boolean status 1 or 0. Switching between the two states is controlled by the Boolean input operands SET1 and RESET1.

The output Q1 is initialised with the value 0 when starting the process. The first processing of the function block with the value 1 of the operand SET causes the output Q1 to be set to 1. A change of the value of SET no longer then effects the output Q1. The value 1 of the input operand RESET1 sets the output Q1 to 0 – the output is reset.

If both input operands have the value 1, the fulfilled reset condition is dominant, i.e. Q1 is reset with priority.

07/01 AWB2700-1306-GB

Example: Fill and close valve again

```
PROGRAM fill

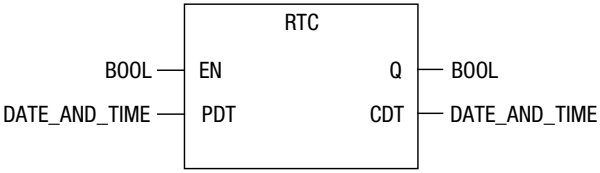
VAR
  Valve1 : RS;
  Position AT %I0.0.0.0.0 : BOOL;
  Barrel_Full AT %I0.0.0.0.1 : BOOL;
  Halt AT %I0.0.0.0.2 : BOOL;
  Valve1_Shut : BOOL;
  Valve1_Open_Shut AT %Q0.0.0.0.3 : BOOL;
END_VAR

  LD Barrel_Full
  OR Halt
  ST Valve1_Shut
  CAL Valve1(SET := Position,
          RESET1 := Valve1_Shut)
  LD Valve1.Q1
  ST Valve1_Open_Shut

END_PROGRAM
```

IEC

RTC
Set Real-Time Clock



Function block prototype

Meaning of operands

EN	Set condition
PDT	Manipulated value for the real-time clock, valid years from 1993 to 2092
Q	Indication whether valid value at CDT
CDT	Current date and time value

Description

The manipulated value Date and Time specified with the operand PDT is transferred to the real-time clock with the rising edge of the operand EN. The output operand Q indicates the validity of CDT.The current clock state is output via the operand CDT irrespective of the state of EN.

07/01 AWB2700-1306-GB

Example: Setting the real-time clock

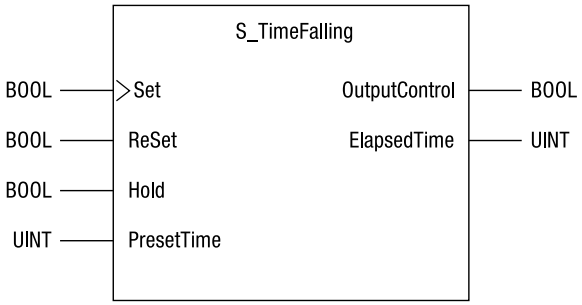
```
PROGRAM clock

VAR
  Set_Clock : RTC;
  Set_Input AT %IO.0.0.0.0 : BOOL;
  Date_and_Time: DATE_AND_TIME := DT#1995-01-
                  01-00:00:00;
  Clock_Set : BOOL;
  Clock_Status : DATE_AND_TIME;
END_VAR

  CAL Set_Clock (EN := Set_Input,
                  PDT := Date_and Time)
  LD Set_Clock .Q
  ST Clock_Set
  LD Set_Clock .CDT
  ST Clock_Status

END_PROGRAM
```

S_TimeFalling
Off-Delay Timer, Seconds



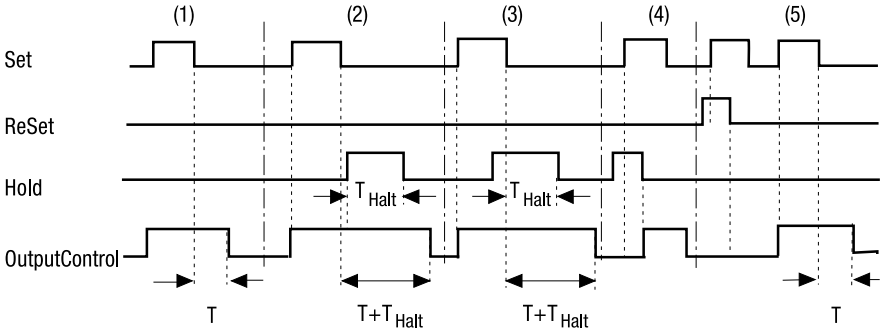
Function block prototype

Meaning of operands

Set	Start condition, rising edge
ReSet	Reset condition
Hold	Time interruption
PresetTime	Time setpoint in seconds
OutputControl	Control output
ElapsedTime	Actual time value in seconds

07/01 AWB2700-1306-GB

Description



Time diagram

A rising edge on the Set input transfers the PresetTime value as delay time T in seconds to the timer and starts the time. The OutputControl output is set to 1. The time is started when the Set input is reset to 0. The OutputControl output is reset to 0 once the delay time has elapsed, i.e. it returns to 0 (1). The ElapsedTime output shows the current time value in seconds. If the status 0 at the Set input is shorter than T, the OutputControl output stays at 1.

Elapsing of the delay time can be stopped by setting the Hold input to 1. The held delay time then continues to run when the Hold input is reset to 0. The off-delay time is therefore lengthened by the duration of the Hold signal, i.e. $T+T_{HALT}$ (2).

If the Hold input is 1 when the rising edge occurs on the Set input, the start of the time delay will be delayed further by the duration of the 1 signal on the Hold input. The OutputControl (4) output also has the delayed status 1.

If the status on the Set input changes from 1 to 0 while the Hold input is 1, the time is started when Hold is reset to 0. The off-delay time is thus increased by the duration of the Hold signal, i.e. $T+T_{HALT}$ (3).

The timer is reset when the ReSet input is 1. A rising edge on the Set input is only recognised as a start when the ReSet has changed from 1 to 0.

Example: 32 seconds off-delay

```
PROGRAM time2

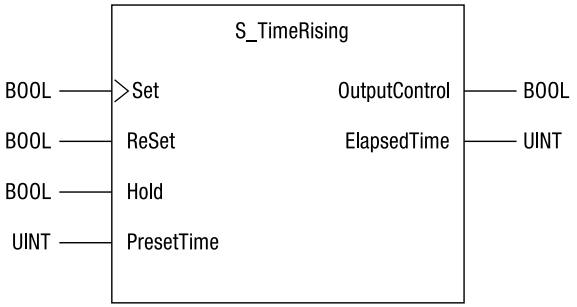
VAR
  Timer2 : S_TimeFalling;
  Start AT %I0.0.0.0.0 : BOOL;
  Halt AT %I0.0.0.0.1 : BOOL;
  TimeValue2 : UINT:= 32;
  (* TimeValue2 = 32 seconds *)
  Output2 AT %Q0.0.0.0.0 : BOOL;
  ActTime2 AT %QW0.0.0.2 : UINT;
END_VAR

  CAL Timer2(Set := Start,
          Hold := Halt,
          PresetTime := TimeValue2)
  LD Timer2.OutputControl
  ST Output2
  LD Timer2.ElapsedTime
  ST ActTime2

END_PROGRAM
```


07/01 AWB2700-1306-GB

S_TimeRising
On-Delay Timer, Seconds

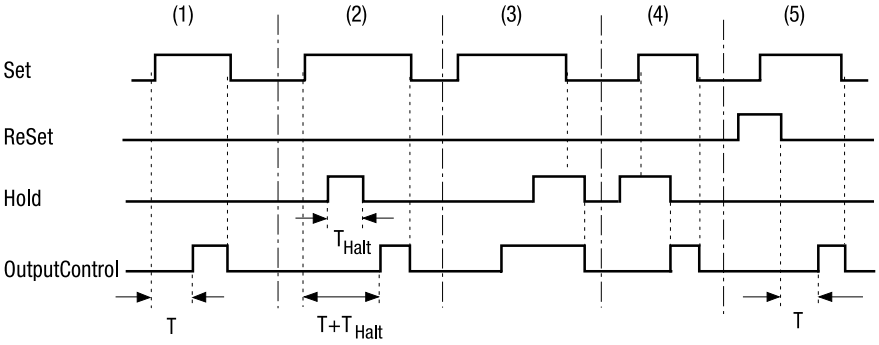


Function block prototype

Meaning of operands

Set	Start condition, rising edge
ReSet	Reset condition
Hold	Time interruption
PresetTime	Time setpoint in seconds
OutputControl	Control output
ElapsedTime	Actual time value in seconds

Description



Time diagram

A rising edge on the Set input transfers the PresetTime value as delay time T in seconds to the timer and starts the time. The OutputControl output switches to 1 after the delay time has elapsed and keeps this status until the Set input is reset to 0 (1). The ElapsedTime output shows the current time value in seconds. If the status 1 at the Set input is shorter than T, the OutputControl output stays at 0.

Elapsing of the delay time can be stopped by setting the Hold input to 1. The held delay time then continues to run when the Hold input is reset to 0. The on-delay time is therefore lengthened by the duration of the Hold signal, i.e. $T+T_{HALT}$ (2).

07/01 AWB2700-1306-GB

If the Hold input is 1 when the rising edge occurs on the Set input, the start of the time delay will be delayed further by the duration of the 1 signal on the Hold input (4). If the Set input is reset from 1 to 0 after the time has elapsed, and while the Hold input is set to 1, the OutputControl output returns to 0 when the Hold input is reset to 0 (3).

The timer is reset when the ReSet input is 1. The time cannot be started when the ReSet input is 1. A rising edge on the Set input is only recognised as a start when the ReSet has changed from 1 to 0 (5).

Example: 12 seconds on-delay

```
PROGRAM time1

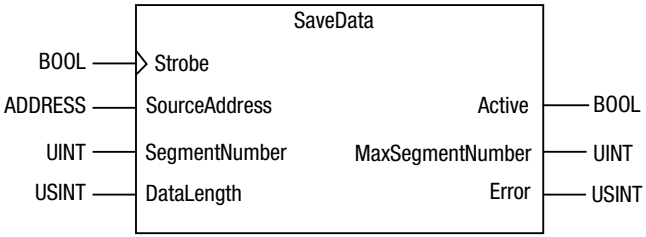
VAR
  Timer1 : S_TimeRising;
  Start AT %I0.0.0.0.0 : BOOL;
  Halt AT %I0.0.0.0.1 : BOOL;
  TimeValue1 : UINT := 12;
           (* TimeValue1 = 12 seconds *)
  Output1 AT %Q0.0.0.0.0 : BOOL;
END_VAR

  CAL Timer1(Set := Start,
           Hold := Halt,
           PresetTime := TimeValue1)
  LD Timer1.OutputControl
  ST Output1

END_PROGRAM
```

PS4-200
PS4-300

SaveData
Save Data on Memory Card



Function block prototype

Meaning of operands

Strobe	Start save operation
SegmentNumber	Segment number (0 - n)
DataLength	Number of array elements of type USINT, Value range:1 to 128
SourceAddress	Source array of the ADDRESS type data area to be reloaded
Active	Save operation running
MaxSegmentNumber	Last possible segment number of the file
Error	Error code

Description

The SaveData function block allows data to be saved to the PLC's memory card. The memory card is logically divided into 512 segments of 128 bytes each. The function block can transfer up to 128 bytes to the memory card.

An important application of this function block is to save the retentive marker area (the recipe range that was specified in the compiler configuration during code generation) and to read it back again with ReloadData after changing the device configuration.

07/01 AWB2700-1306-GB

The Error output indicates the following error codes:

1 No memory module present
2 No access possible due to Online connection
3 No access possible due to simultaneous access via SaveData and WriteMC function block
4 SegmentNumber exceeds the value of MaxSegmentNumber
5 Invalid data length (valid range for Data Length: 1 – 128)
6 Start address is invalid
7 Only for PS4-200: No access possible. MC is used as a source
8 Only for PS4-300: The MC was formatted without recipe data range

Only applicable with PS4-200: the Active output of these function blocks is always 0 in the PS4-150/-200. The current function block status can be scanned via the Error output when the function blocks are called again. If the Error output is 0, processing was carried out error-free. If the output is 3, the function block is processing.

To refresh the Error output of this function block a new edge on the Strobe input must be generated. Otherwise the error message will be kept (with 1 on Strobe) or will be set to 0 (with 0 on Strobe).



The ReloadData function block allows data to be read back from the memory card.

Example: Writing data to the memory card

```
PROGRAM S_Card

VAR
  SendData : SaveData;
  Start AT %IO.0.0.0.2 : BOOL;
  Mem_start AT %MBO.0.0.100 : BYTE;
  Segment_NR : UINT;
  DataLength : USINT;
  ErrorMessage : USINT;
END_VAR

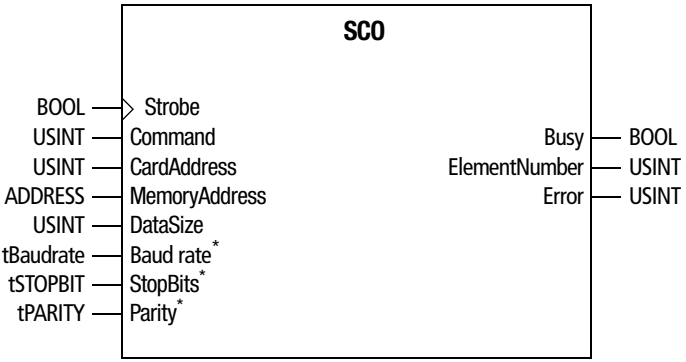
  CAL SendData(Strobe := Start,
                  SourceAddress := &Mem_start,
                  SegmentNumber := Segment_NR,
                  DataLength := DataLength)

  LD SendData.Error
  ST ErrorMessage
  NE 0 (*If content not equal then error*)
  JMPC ErrorEvaluation
   .
   .
  ErrorEvaluation:
  .
Error detection:
  .
  .
END_PROGRAM
```


07/01 AWB2700-1306-GB

PS4-200
PS4-300

SCO
Serial Communication Function Block:



Function block prototype

Meaning of operands

Strobe	Initiate job (rising edge)
Command	Job type
CardAddress	Interface selection (0 = RS 232, 1 = RS 485)
MemoryAddress	Start address of send or receive range
DataSize	Size of send or receive data
Busy	Function block status
ElementNumber	Number of data bytes sent/received
Error	Error code
Baudrate*	Transfer speed, Default: 9600 Baud
StopBit*	Number of Stop bits, Default: 1
Parity*	Transfer parity, Default: without

* Only for RS 232 interface of PS4-300

Description
The SCO function block allows transparent communication between the user program and the connected data terminals via the RS 232 and RS 485 interfaces of the PLC.

The function block can control send operations and (not simultaneously) receive operations. Since the transfer is executed in half-duplex mode, simultaneous sending and receiving is not possible. A send job has priority over a receive job.

There are no retentive settings for the function block.

No CTS/RTS handshake lines are used.

The Strobe input is edge-triggered. A change from 0 to 1 or from FALSE to TRUE starts the job specified via the Command input. All input parameters of the function block are saved with the Strobe signal.

If the Strobe input triggers a send job or readiness to receive, this is indicated at the Busy output by a 1. The Busy output shows the active status of the function block. Readiness to receive can be stopped with a falling edge on the Strobe input.

When communication is aborted, the data received so far is kept in the receive buffer and is made available with a rising edge at the Strobe input.

The mode of the SCO function block is set at the Command input. The mode set does not become active until a rising edge is sent to the Strobe input.

07/01 AWB2700-1306-GB

Mode:

1. Send job
   The RS 232 or RS 485 interface is first of all initialised and then the send job is executed.

2 Readiness to receive, data stored from start address of the receive range.

   A rising edge at the Strobe input first initialises the RS 232 or RS 485 input, depending on the setting made at the CardAddress input.

   The entry for newly received data bytes starts at the beginning of the receive range specified via the MemoryAddress input.

   The number of received data bytes is indicated at the ElementNumber output.

   If the number of data bytes specified at the DataSize input has been received, the Busy output switches to 0.

   A new rising edge at the Strobe input restarts the entry of received bytes at the beginning of the receive range.

3 Readiness to receive, data saved directly behind data already in receive range.

   In Mode 3, the receive channel stays open for as long as the Strobe input is 1 irrespective of the number of data bytes specified at the DataSize input. Other responses are described under "Special features of the PS4-200/PS4-300".

   In Mode 3, the rising edge at the Strobe input also starts the entry of new data bytes at the beginning of the receive range defined via the MemoryAddress input.

   As long as the number of data bytes received with previous jobs has not reached the physical limit of

the receive buffer, new data bytes will be added to previously received data bytes in the receive range.

This also applies if a receive job was aborted with Strobe = 0 or if the number of data bytes specified at the DataSize input was reached and Busy = 0. The data bytes received so far are shown at the ElementNumber output with the next receive job and transferred directly to the application.

The receive buffer is 63 bytes for the RS 232 interface of both PLCs. With the RS 485 interface, the size for the PS4-200 is 127 bytes and 254 bytes with the PS4-300.

The number of received data bytes is indicated at the ElementNumber output.

If the receive buffer is full, the Busy output is reset to 0.



This receive mode is only designed for the commissioning phase, when the number of data bytes from the sender is not yet known.

07/01 AWB2700-1306-GB

Special features of the PS4-200:

– When using the RS 232 interface, mode 3 can only be used if the connection was established beforehand in Mode 1 or 2. Otherwise error code 7 will be output at the Error output. Only in these two modes can SUCOM be switched to serial communication via the SCO.

– In Mode 3, the receive channel stays open, even if the Strobe input is 0.

– When using the RS 232 interface, the PS4-200 switches automatically to SUCOM operation if more than 63 data bytes were received.

Special features of the PS4-300:

- The receive buffer is 254 bytes for the RS 485 interface.
- When using the RS 232 interface, the function of Mode 3 is exactly the same as with Mode 2. The receive channel is closed after receiving the number of data bytes specified at the DataSize input and the entry of receive data bytes is started at the beginning of the receive range with the next rising edge at the Strobe input.
- When using the RS 232 interface, the PS4-300 does not switch automatically to SUCOM operation if more than 63 data bytes were received.

4 Terminating serial communication of the SCO via the RS 232 interface (SUCOM operation is then possible once more).



When using the RS 232 interface, the PS4-200 switches automatically to SUCOM operation if more than 63 data bytes were received. This does not occur with the PS4-300.

The **CardAddress** input is used to select the interface required for communication:

0: RS 232
1: RS 485

07/01 AWB2700-1306-GB

The MemoryAddress input defines the start address of the send or receive range. The receive data is entered directly in the specified range. Ranges %MB, %SDB and %RDB with a one-byte data width can be used.

The **DataSize** input defines the necessary size of the receive buffer and the number of data bytes to be sent.
The following values can be set:
Interface RS 2321 to 63
Interface RS 4851 to 127

The **Busy** output indicates the status of the SCO function block:

0 = Function block inactive
1 = Function block active.

The active status of a function block is indicated via Busy=1 if a send job or readiness to receive have been started. The Busy output switches to 0 as soon as the send job has been completed or when the receive buffer is full.

The **ElementNumber** output indicates the number of data bytes already received.

With a send job the number of sent data bytes is indicated (only with RS 232 and RS 485).

The **Error** output indicates the following error codes:

0 No error
1: Command input incorrectly set. Valid parameters from 1 to 4
2: CardAddress input is incorrectly set. Valid parameters are 0 (RS232) and 1 (RS485)
3 Conflict due to SUCOM or Suconet operation
Possible causes:
RS 232: SUCOM protocol is currently active
RS 485: Incorrect parameters in the Topology Configurator
PS4-200 older than version 04.
4: Marker address (MemoryAddress) is too big. The marker range used must be within the active marker range defined for the CPU (configured via Generate → Program Parameters in the Sucosoft S40 Program Code Generation tool).
5: DataSize parameters incorrect. The following values can be set:
Interface RS 232 1 to 63
Interface RS 485 1 to 127
6: MemoryAddress parameters incorrect. The combination of MemoryAddress and DataSize violates the active marker area specified in the configuration for the PLC.
Example: active marker range: 120
MemoryAddress (100) + DataSize (40) = 140
=> too big
7 Receive buffer overflow: more than 63 or 127 bytes received.
8: Parity error RS485: interface parameters of sender and receiver do not match.
9: A send and receive job were started simultaneously via this interface.

Interface settings
The RS 232 and RS485 interfaces are available for operating the SCO function block.

With the PS4-300, the RS 232 interface is set via the SCO inputs tBAUDRATE, tSTOPBIT and tPARITY and manufacturer-defined data types.

These data types can be used with the SCO call in the instruction section either directly or using its own identifier in the declaration:

```
VAR

  Parity : tPARITY;
  .
  .
  CommunicationBlock : SCO;
END_ VAR
```

Direct use:

```
CAL    CommunicationBlock(
                              .
                              Parity := ODD_Parity,
                              .
                              )
```

Using a declared identifier:

```
ParityOdd = ODD_Parity;

CAL    CommunicationBlock(
                              .
                              Parity := Parity1,
                              .
                              )
```

The following settings are possible for the baud rate: Baud_2400, Baud_4800, Baud_9600, Baud_19200, Baud_38400 and Baud_57600

The following settings are possible for parity: NONE_Parity, ODD_Parity and EVEN_Parity

The following settings are possible for the Stop bit: ONE_Stop_Bit, TWO_Stop_Bits

When using the PS4-200, the transfer parameters are fixed: Baudrate 9600 Baud, 8 data bit, 1 Stop bit without parity check. Up to 63 bytes can be transferred.

The RS 485 interface is assigned with parameters via the Sucosoft S40 Topology Configurator. For this choose Edit → Parameters in the Topology Configurator, select Transparent mode and click the Interface Parameters button. The permissible data volume that can be transferred is between one and 127 bytes.

Impermissible combinations are detected by the software and are not accepted.

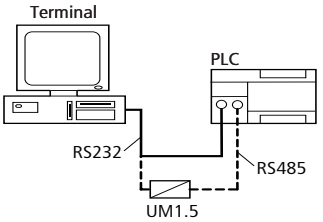
The application of the SCO function block is illustrated in the following example:

Communication PLC - Terminal

Data communication between a PLC and a terminal.

07/01 AWB2700-1306-GB

Example "Communication PLC -> Terminal", SCO
Data communication between a PLC and a PC with a terminal emulation program or a terminal.



Requirements for the following example:

Hardware:	PLC, e.g. PS4-201-MM1 from version 04 or PS4-341-MM1 ZB4-108-ES1 digital simulator ZB4-303-KB1 cable
	with communication via RS 485: UM1.5 interface converter, NG31 power supply unit for the interface converter, KPC-PS306 adapter cable
Software:	Sucosoft S 40 from version 4.0 Terminal emulation program

The interface parameters of the terminal are set accordingly in the terminal emulation program.

Function
The data exchange between the PLC and the terminal is carried out via the program SCO_BSP1.

Send
Input %I0.0.0.0.3 is used to set the required interface. Set 1 for the RS 485 interface.

Use input %I0.0.0.0.4 to select between Text 1 and Text 2:
%I0.0.0.0.4 = 1 On condition 1 fulfilled
%I0.0.0.0.4 = 0 Limit switch 2 reached.
The bytes to be sent are located in the memory from marker byte 200. The address of this marker byte is automatically sent to the MemoryAddress input.

The DataSize input is automatically set according to the text selected.

The Command input is automatically set to mode 1 (send).

A rising edge at input %I0.0.0.0.0 sends the text to the terminal.

Receive



For receive operations use the same interface settings as for send operations.

The Command input is automatically set to mode 2 (receive).

In order for the function block to copy the receive data to the marker range from address %MB0.0.0.400, the start address is automatically sent to the MemoryAddress input.

The number of bytes to be received (=size of receive buffer) is automatically entered at the DataSize input (in example 16 data bytes).

%I0.0.0.0.1 switches the SCO to receive mode, and the Busy output shows 1 to indicate that the function block is active.

The ElementNumber output shows the number of bytes already received. If the receive buffer is completely full with data (ElementNumber = DataSize), Busy switches to 0. If %I0.0.0.0.1 is triggered again, the function block is ready to receive again. The clearing of the receive buffer is indicated by 0 at the ElementNumber output.

In this example program one instance of the SCO was used for sending and for receiving. Multiple instancing would not save time since the PLC only works in half-duplex mode.



The SCO can be instantiated twice under different names, with one instance used to access the RS 232 interface and the second instance to access the RS 485 interface. Multiple instantiations of the SCO function block to a physical interface will lead to communication malfunction.

Switching to SUCOM mode
Input %I0.0.0.0.2 (in the example Switch _Strobe) enables you to switch to SUCOM mode after SCO communication via the RS 232 interface.

The Command input is automatically set to mode 4 (disconnect serial communication).

A rising edge on input %I0.0.0.0.2 triggers the changeover.



To run the example via both the RS 232 interface (using the interface converter) and the RS 485 interface, set the PLC in the Sucosoft S40 Topology Configurator to Transparent mode and change the interface via input I0.0.0.0.3 (in the example PRG_SBI).

```
PROGRAM SCO_BSP1

VAR
     (* Directly represented variables *)
Send_Strobe             : BOOL AT %IO.0.0.0.0;
Rcve_Strobe             : BOOL AT %IO.0.0.0.1;
Changeover_Strobe       : BOOL AT %IO.0.0.0.2;
PRG_SBI                 : BOOL AT %IO.0.0.0.3; (* 0 = RS232; 1 = RS485 *)
SendText_No             : BOOL AT %IO.0.0.0.4; (* 0 = Text1; 1 = Text2 *)
SCO_outputs             : BYTE AT %QB0.0.0.0;
(*Indication: Receive volume if Bit7=0, Error message if Bit7=1*)

Send_buffer             AT%MB0.0.0.200: ARRAY[1..127] of BYTE;
Receive_buffer          AT%MB0.0.0.400: ARRAY[1..127] of BYTE;

     (* Local variables/intermediate markers *)
Text                    : STRING;
Text2                   : STRING := 'Limit switch 2 reached $R$L';
(* Text variable 'Text2' is initialised here directly with Text*)
Send_active             : BOOL;
Rcve_active             : BOOL;
Changeover_active       : BOOL;
Receive_No              : USINT := 16;      (* 16 char. for example *)

     (* FB instances *)
CommunicationFB         : SCO;              (*SCO instance for send/receive*)
Send_converter          : STRtoADR;
Rcve_converter          : ADRtoSTR;
END_VAR

(* General SCO setting *)
LD    PRG_SBI
BOOL_TO_USINT
ST    CommunicationFB.CardAddress(* RS232/RS485 *)

LD    Changeover_active
JMPC  Changeover_Scan   (*Changeover job in process *)

LD    Changeover_Strobe
JMPC  Changeover_Job    (*Changeover to Sucom mode *)

LD    Rcve_active
JMPC  Recv_Scan         (*Receive job in process *)
```

```
LD    Send_active
JMPC  Send_Scan          (*Send job in process *)

LD    Rcve_Strobe
ANDN  Send_Strobe        (* Send and receive not at same time *)
JMPC  Rcve_Job           (* Start Receive job

LD    Send_Strobe
ANDN  Rcve_Strobe
JMPC  Send_ Job

LD    0
ST    CommunicationFB.Strobe
(* Strobe reset, since no job or job end *)
JMP   SCO_Call

(* *********** Processing a send job *********** *)
Send_job:
      (* Text transfer at start of send job *)
LD    'On condition 1 fulfilled $R$L'(*Text variable 1*)
ST    Text
LD    SendText_No
JMPC  TextConvert

LD    Text2                                (* Text variable 2 *)
ST    Text

TextConvert:
CAL   Send_converter(    Enable:             =1
                         TextString:         = Text,
                         TransmitVariable:   = &SendBuffer
       )

LD    Send_converter.StrLen
UINT_TO_USINT
ST    CommunicationFB.DataSize(* Parameter transfer to SCO FB*)

LD    1
ST    CommunicationFB.Command(* Send Mode *)
LD    &Send_buffer
ST    CommunicationFB.MemoryAddress
LD    1
```


07/01 AWB2700-1306-GB

```
ST    CommunicationFB.Strobe                (* Send*)

CAL   CommunicationFB                      (* SCO call *)

LD    CommunicationFB.Error
NE    0
JMPC  Communication_Error
LD    CommunicationFB.Busy
ST    Send_active

Send_Scan:
LD    CommunicationFB.Busy
JMPCN Send_Mark

LD    0
ST    CommunicationFB.Strobe
(* Send job in progress, complete job *)
JMP   SCO_Call

Send_Mark:
LD    Send_Strobe
(* New send job should not be possible until after edge change *)
JMPC  End

LD    0
ST    Send_active
JMP   End

(* ************* Processing a receive job ************ *)
Rcv_Job:
LD    0
ST    SCO_outputs

Rcv_Scan:
LD    Receive_No
ST    CommunicationFB.DataSize(*Receive job, 16 Char. *)
LD    2
ST    CommunicationFB.Command(* Receive mode *)
LD    &Receive_buffer
ST    CommunicationFB.MemoryAddress
LD    1
ST    CommunicationFB.Strobe(* Receive *)
```

```
CAL   CommunicationFB                        (* SCO call *)

LD    CommunicationFB.Error
NE    0
JMPC  Communication_Error

LD    CommunicationFB.Busy
ST    Rcve_active
LD    CommunicationFB.ElementNumber
USINT_TO_BYTE
ST    SCO_outputs

(* Example display of first 5 characters *)
ld    Receive_buffer [1]
ld    Receive_buffer [2]
ld    Receive_buffer [3]
ld    Receive_buffer [4]
ld    Receive_buffer [5]

(* Strobe change ends the receive job *)
ld    Rcve_Strobe
JMPC  End

LD    0
ST    CommunicationFB.Strobe
ST    Rcve_active
(* Receive job in progress, Complete job *)
JMP   End

Communication_Error:
LD    CommunicationFB.Error
USINT_TO_BYTE
OR    16#80                                  (* For troubleshooting, Bit7 = 1 *)
ST    SCO_outputs                            (* Bits 0-6 indicate the error *)
LD    0
ST    CommunicationFB.Strobe (* Job abort*)
JMP   SCO_Call

(* Processing a job for switching to Sucom mode *)
Changeover_Job:
LD    0
ST    SCO_outputs
LD    4
```

```
ST CommunicationFB.Command
LD    1
ST    CommunicationFB.Strobe(*Change to Sucom *)
ST    Changeover_active
JMP   SCO_Call

Changeover_Scan:
LDN   Send_Strobe
ANDN Rcve_Strobe
ANDN  Changeover_Strobe
(*New send/receive job not possible until after edge change *)
JMPCN End

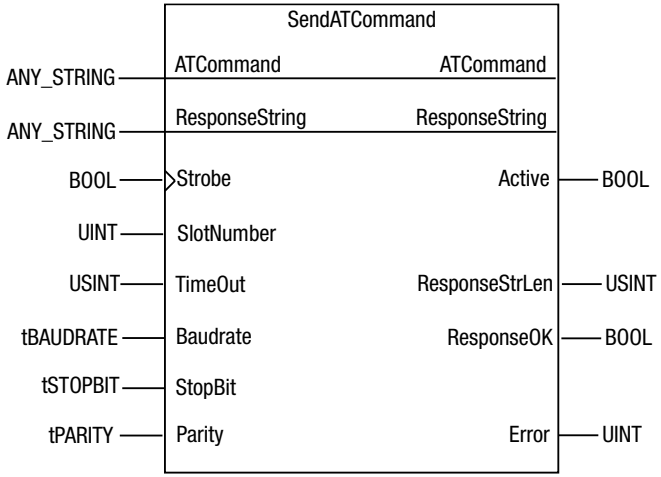
LD    0
ST    Send_active
ST    Rcve_active
ST    Changeover_active
ST    CommunicationFB.Strobe

SCO_call:
CAL   CommunicationFB                        (* SCO call *)

End:
END_PROGRAM
```

PS4-300
PS416

SendATCommand
Send AT Commands to a Modem



Function block prototype

Meaning of operands

ATCommand	AT command to send to modem (max. 254 bytes)
ResponseString	Response to AT command (max. 254 bytes)
Strobe	Enable function block
SlotNumber	Selection of PLC interface: 0 CPU 400/300-SBI or PS4-341-MM1 SBI (transparent mode) 1 CPU 400/300/200-PRG or PS4-341-MM1-PRG (transparent mode) 4..20 slot number of associated COM card
TimeOut	Wait for response to AT command, (range 10 ... 90 seconds, default:60 seconds)
Baudrate	Transmission speed, specified via tBAUDRATE
StopBit	Number of stop bits, specified via tSTOPBIT
Parity	Transmission parity, specified via tPARITY

07/01 AWB2700-1306-GB

Active	State of job processing 1: Job accepted 1->0: Job terminated
ResponseStrLen	Number of received characters
ResponseOK	Response in ResponseString: 1: OK or CONNECT received. 0: Neither OK nor CONNECT received
Error	Error code

Description

The SendATCommand function block is used to send AT commands to a modem which is attached to one of the following PLC interfaces:

CPU 400/300/200-PRG interface,

CPU 400/300-SBI interface or

PS4-341-MM1-SBI/PRG interface or

COM-200 interface.

To allow the PLC and modem to communicate with each other, the interface parameters for baud rate, stop bits, parity and data bits should be set to the same values on the PLC and the modem. Refer to your modem manual for information on setting the modem interface parameters.

The SendATCommand function block controls the TxD (transmit data) and RxD (receive data) lines. Control lines are not supported.

Before sending an AT command, you must set parameters for the CPU PRG interface. You must specify the baud rate, stop bits and parity with the following data types. The number of data bits is always 8.

```
TYPE tBAUDRATE:
       ( Baud_2400,
         Baud_4800,
         Baud_9600,
         Baud_19200,
         Baud_38400,
         Baud_57600 ):=Baud_9600;
END_TYPE

TYPE tSTOPBIT:
       ( ONE_Stop_Bit,
        TWO_Stop_Bits) := ONE_Stop_Bit;
END_TYPE

TYPE tPARITY:
       ( NONE_Parity,
        ODD_Parity,
        EVEN_Parity ):=NONE_Parity;
END_TYPE
```

You can choose between an RS 232 or RS 485 interface with the switch on the front of the CPU 400/300/200.

Parameters for the CPU 400/300-SBI interface and the COM 200 interface are set in the topology configuration as follows.

For the CPU 400/300-SBI interface, you can configure transparent mode and the values for baud rate, parity, stop bits and data format. Interface defaults to RS 485.

For the COM 200 interface, you can configure baud rate, parity, stop bits and data format and the full duplex mode, no handshake and no start and stop code. Interface module can be set to IFM 232.1 or IFM 232.2.

07/01 AWB2700-1306-GB

The SendATCommand function block can be used to send all Hayes-compatible AT commands to a modem and to receive the modem responses.



Refer to your modem manual for information on the AT commands supported by your modem.

The AT command is specified as a string which must start with AT and end with “RL”. “$R” is the code for “Carriage Return” and “$L” is the code for “Line Feed”.

Example: “ATZ RL” is used as the AT command to reset the modem.

A rising edge at the Strobe input accepts all input parameters.

If SlotNumber is 1 (CPU 400/300/200 PRG interface), the interface parameters are set before sending the AT command to the modem. For the other interfaces, only the AT command is sent (the interface parameters are already specified in the topology configuration).

After enabling the function block, Active is set to 1. It is reset to 0 as soon as a modem response is received, if a parameter error is detected or if the timeout is exceeded. The timeout is specified with TimeOut in seconds.

Signal transitions at the Strobe input are ignored while Active is 1.

An active send job will be aborted on the PS416-COM-200 card if the modem function block is activated via the TransmitActive output on the COM function block. This can be avoided by starting a timer with the TransmitActive output, that causes a delay according to the specified baud rate and the data volume to be sent. This does not affect Transparent mode with the central unit.

ResponseString contains the entire response string from the modem. The length of the response string is contained in ResponseStrLen. ResponseOK is 1 if the modem responds with OK or a connect message.

Error codes at the Error output:

0 No error
1 ResponseString contains an unknown response
2 ResponseString incomplete or is longer than the declared length of the ResponseString variable
3 ResponseTimeOut.
e.g. no response received because
Problem with connection to modem,
Interface parameters for the PLC and modem do not match,
Modem not active.
4 Error when receiving data
5 Error when transmitting data
16 SlotNumber invalid (0, 1, 4...20)
17 ATCommand does not begin with AT
18 ATCommand does not end with “RL”
19 ATCommand greater than 254 characters
20 TimeOut value invalid
21 CPU 400/300-SBI/COM-200: Transparent mode not or incorrectly set
22 COM-200: Not in specified slot
23 COM-200: No or incorrect IFM inserted
24 CPU 400/300/200-PRG: Transparent mode cannot be set

07/01 AWB2700-1306-GB

25 Modem not configured for Echo-OFF
26 CPU-400/300/200-PRG: send and receive error



The function block automatically configures the modem as follows
– Modem result codes on,
– Modem result codes plain text on
– Modem command echo off.
Other function blocks can also be used to access the COM or SBI interface. Conflicts can occur if several function blocks try to access the same interface at the same time. If necessary, the user program should include interlocks to make sure that this cannot happen.

Example: Switching off modem echo with ATE0

```
PROGRAM echo_off

VAR
   FB_ECHO_OFF : SendAtCommand;
   FallingEdge : F_TRIG;
   FB_Strobe AT %IO.0.0.0.0 : BOOL;
   ATCommand_ECHO_OFF : STRING(8) := 'ATE0$R$L';
   FB_Active : BOOL;
   ECHO_OFF_Response : STRING(254);
   ECHO_OFF_Response_Length : USINT;
   ECHO_OFF_ResponseOK AT %Q0.0.0.0.0 : BOOL;
   FB_Error : UINT;
   (*...*)
END_VAR
   (*...*)
```

```
   CAL FB_ECHO_OFF(Strobe := FB_Strobe,
                   SlotNumber := 1,
                   ATCommand := ATCommand_ECHO_OFF,
                   ResponseString := ECHO_OFF_Response,
                   TimeOut := 20,
                   Baudrate := Baud_9600,
                   StopBit := ONE_Stop_Bit,
                   Parity := NONE_Parity
                   |
                   FB_Active := Active,
                   ECHO_OFF_Response_Length := ResponseStrLen,
                   ECHO_OFF_ResponseOK = ResponseOK,
                   FB_Error := Error)

   CAL FallingEdge(CLK := FB_ActiveExample:(* Detects falling edge *)
                   |
                   := Q)

   LD      FallingEdge.Q
   JMPCN   AwaitAnswer

CheckAnswer:(* Falling edge on Active output *)
   LD      FB_Error                    (* Error handling *)
   GT      0
   JMPC    Error

   (* No error, modem echo switched off; modem response handling *)
   (* ... *)
   JMP     End

AwaitAnswer:          (* Active is 0 or 0->1, request is being processed,
                      will be checked again in next cycle *)
   ( *...* )
   JMP     End

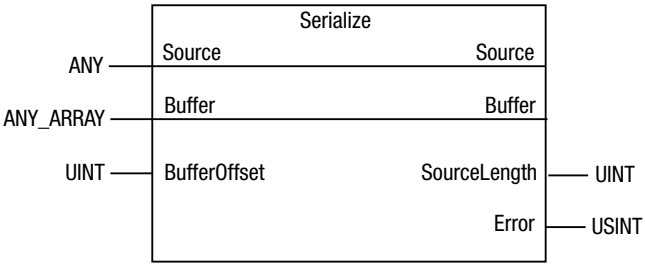
Error:(* Error handling *)

   (*...*)

End:
END_PROGRAM
```


07/01 AWB2700-1306-GB

Serialize
Copy Data Structure to Array



Function block prototype

Meaning of operands

Source	Source variable
Buffer	Destination area
BufferOffset	Offset in Destination area
SourceLength	Length of source variable in bytes
Error	Error code

Description

The Serialize function block copies any required data type to a destination area at a starting position specified by Buffer + Bufferoffset. The data can then be sent, for example, as an array to another PLC and then unpacked with the Deserialize function block.

The SourceLength output specifies the length of the source data type in bytes.

The buffer is an ARRAY OF BYTE, ARRAY OF USINT or ARRAY OF SINT of any required size.

The Error output indicates the following error codes:
0 No error
1 Buffer overflow
2 Offset lies outside the buffer
3 Source lies within the buffer
4 SourceLength is 0



The source data type BOOL is not supported and will result in an error during compilation.

Due to the different memory layout techniques available, the Serialize function will produce different results (e.g. for structures).

If you wish to exchange data between different PLC types, copy only elementary data types and arrays of type ANY_BIT or ANY_INT. Data types BOOL, STRING, ARRAY OF BOOL are not permissible for this operation. They have been implemented differently as derived types in the PLCs concerned.

The restriction does not apply if the data exchange is between PLCs of the same type.

Example: Copying 20 bytes into a 30-byte array

```
PROGRAM sendstruct
VAR
  serie_fb : Serialize;
  struct_buffer AT %SDB1.1.0.0 :
                          ARRAY [1..30] OF BYTE;
  CheckSumOffset : UINT := 0;
  CheckSum : UINT;
  DTOffset : UINT := 2;
  TransDate : DT;         (* Send date *)
  DataOffset: UINT := 10;
  TransmitData : Array [1..20] OF BYTE;
                          (* User data *)
  SerializedLength : UINT;
  Error : USINT;
END_VAR

  (* Prepare send data *)
  CAL serie_fb(Source := CheckSum,
               Buffer := struct_buffer,
```


07/01 AWB2700-1306-GB

```
                BufferOffset := CheckSumOffset,
                |
                SerializedLength := SourceLength)

  LD       serie_fb.Error
  USINT_TO_BOOL          (* Any error ? *)
  JMPC _Fail             (* Error handling *)

  CAL  serie_fb(Source := TransDate,
                Buffer := struct_buffer,
                BufferOffset := DTOffset,
                |
                SerializedLength := SourceLength)

  LD serie_fb.Error
  USINT_TO_BOOL          (* Any Error ? *)
  JMPC _Fail             (* Error handling *)

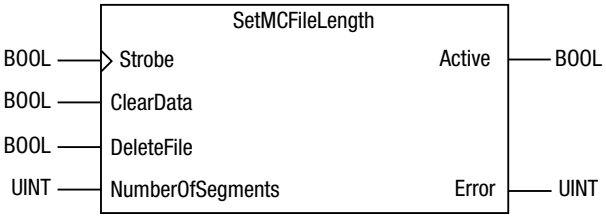

  CAL          serie_fb(Source := TransmitData,
                Buffer := struct_buffer,
                BufferOffset := DataOffset,
                |
                SerializedLength := SourceLength)
  LD serie_fb.Error
  USINT_TO_BOOL          (* Any Error ? *)
  JMPC _Fail             (* Error handling *)

  (* Structure complete, data can be sent *)

_Fail:     (* Error handling *)
```

PS416

SetMCFileLength
Create and Initialise Data File on Memory Card



Function block prototype

Meaning of operands

Strobe	Create file RSDAT.DAT
ClearData	0 Do not initialise file contents 1 Initialise file contents with 0xFF
DeleteFile	0: Do not delete existing data file on the Memory Card 1: Delete existing data file on the Memory Card
NumberOfSegments	Number of 128 byte segments to create
Active	File create operation active
Error	Error message

Description:

The SetMCFileLength function block creates the file RSDAT.DAT on an SRAM memory card (MC).

The number of 128 byte data segments is specified with NumberOfSegments and is only limited by the capacity of the memory card used. The following maximum values for NumberOfSegments apply if the memory card is formatted with a PS416:

MC capacity 512 KBytes	3944 segments
MC capacity 1 MB	7944 segments

07/01 AWB2700-1306-GB

The ClearData input specifies whether the file should be initialised with the data value “0xFF”. If the function block is called with ClearData = 0, the initial file content is not defined.

The input operand DeleteFile determines whether a file already in the memory card should be deleted.

DeleteFile = 0	Do not delete existing data file on the Memory Card
DeleteFile = 1	Delete existing data file on the Memory Card

The function block is executed asynchronously of the user program cycle and can take several program cycles. The Active output is set during this time. Error-free completion of the function block is indicated by the Active output changing to 0 and by 0 at the Error output.

Error codes at the Error output:

0 Function block execution completed without errors
1 No memory card inserted
10 Memory card type not SRAM.
11 Memory card is not formatted.
12 Memory card is write protected.
13 Insufficient free memory on memory card to create the file RSDAT.DAT.
14 The file RSDAT.DAT does not have the required minimum size of 128 bytes.
15 File RSDAT.DAT already exists.
16 Internal error during execution of the function block

Example: Create and initialise file RSDAT.DAT on the memory card

```
PROGRAM INITIALISE

VAR
   StrobeInit AT %I0.0.0.0.0 : BOOL;
   ActiveInit AT %Q0.0.0.0.0 : BOOL;
   ErrorInit :USINT;
   FBInitRSData : SetMCFileLength;
   NumberOfSegments :UINT;
END_VAR

   LD 3222
   ST NumberOfSegments
   CAL FBInitRSData(Strobe := StrobeInit,
                    ClearData := 1,
                    DeleteFile := 0,
                    NumberOfSegments := NumberOfSegments,
                    |
                    ActiveInit := Active,
                    ErrorInit := Error)

   LD ActiveInit
   JMPC CONTINUE
   LD ErrorInit
   EQ 0
   JMPC CONTINUE

ERRORHANDLING:

   (* Carry out error handling *)

CONTINUE:

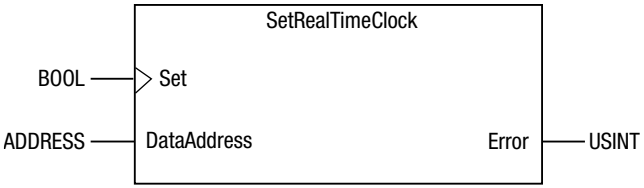
   (* Further parts of user program *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

PS4-200
PS4-300

SetRealTimeClock
Set Real-Time Clock



Function block prototype

Meaning of operands

Set	Enable for accepting the set values, rising edge
DataAddress	Start address of range for the set values
Error	Error code

Description

A rising edge at the Set input sets the real-time clock with the values stored in the DataAddress data range. The set or default values are provided in seven consecutive byte data elements. The bytes are in the order year, month, day, weekday, hours, minutes and seconds.

A rising edge at the Set input causes all values to be transferred, but not, however, selected data such as only hour or year.

The data type declared for the data elements in which the set values are stored must be in byte format, i.e. data types BYTE, SINT or USINT. Using data type USINT is recommended.

The Error output indicates the following error codes:

1 Incorrect DataAddress
2 Invalid year (valid range: 0-99)
3 Invalid month (valid range: 1-12)
4 Invalid day (valid range: 1-31)
5 Invalid weekday (valid range: 0-6, 0 = Sunday)
6 Invalid hour (valid range: 0-23)
7 Invalid minute (valid range: 0-59)
8 Invalid second (valid range: 0-59)



Use RTC in new programs instead of SetRealTimeClock.

07/01 AWB2700-1306-GB

Example: Setting the real-time clock

```
PROGRAM Clock

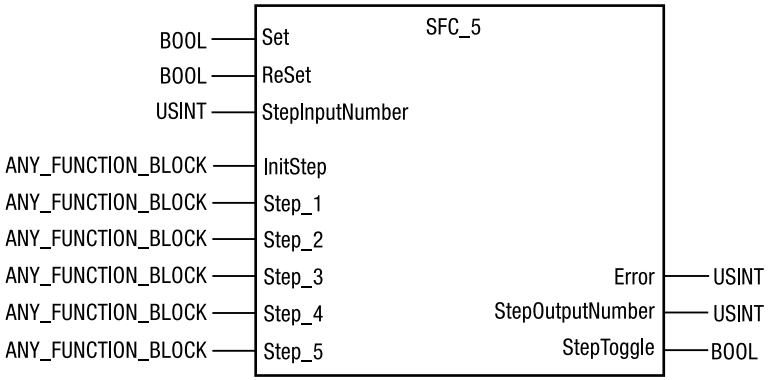
VAR
  Year1 AT %MB0.0.0.0 : USINT := 97;
  Month1 AT %MB0.0.0.1 : USINT := 1;
  Day1 AT %MB0.0.0.2 : USINT := 1;
  Weekday1 AT %MB0.0.0.3 : USINT := 2;
       (* 2 = Tuesday*)
  Hour1 AT %MB0.0.0.4 : USINT := 12;
  Minute1 AT %MB0.0.0.5 : USINT;
  Second1 AT %MB0.0.0.6 : USINT;
       (* Default value
          Tuesday, 1.1.1997 12.00h *)
  Set_time : SetRealTimeClock;
       (* Function block declaration *)
  Set : BOOL AT %I0.0.0.0.0;
       (* Declare set input*)
END_VAR

  LD 98
  ST Year1    (* Set year to 1998 *)
  LD 9
  ST Month1   (* Set month to September *)
  LD 12
  ST Day1     (* Set day to the 12th *)
  LD 6
  ST Weekday1(* Set weekday to Saturday *)
              (* Clock is completely reset
                          via function block *)
       (*CAUTION: 12.00 is used as default setting *)
  CAL Set_time (Set := Set,
                          DataAddress := &Year)
  LD Set_time.Error
  NE 0        (* Scan for no errors *)
  JMPC Error evaluation
   .
   .

END_PROGRAM
```

PS4-200
PS4-300

Sfc_x
Sequential Control Function Block
(x = 1..128)



Function block prototype

Meaning of operands

Set	Activate the sequential control function block
ReSet	Reset and call initialisation step
StepInputNumber	Step number input
InitStep	Instance name of the initialisation step
Step_x	Instance name of the steps to be called (FBs). There are as many Step parameters as steps in the sequence
Error	Error code
StepOutputNumber	Step number output
StepToggle	Step toggle

Description

The sequential control function block makes it possible to ensure that the user program is structured simply and clearly. Each sequential control function block can call up to 128 steps. Each step can itself call an entirely new sequence, thus

07/01 AWB2700-1306-GB

allowing a maximum nesting depth of 8 levels. The individual steps consist of function blocks that contain the actions to be executed. The logical structure of a sequence control can be applied to the user program directly.

The Step_x inputs define the steps to be executed by means of the corresponding function blocks. The actions making up each step are contained in the appropriate function blocks.



The step function block instances that are called from a sequential control function block must not have any input or output parameters. An instance of this kind must only contain local variables (VAR) or externally declared global variables (VAR_EXTERNAL).

Only for PS4-200:

In PS4-200 programs the InitStep and Step_x parameters can only be assigned a function block instance with the SFC_x call if it was defined in the main program. Since the called sequential control function block only works with global data, an instance defined locally in another function block cannot be assigned.

The instance of an SFC function block can only be called once in a PS4-200 program since the step sequence defined by the first instance cannot be changed when the program is running, not even via the Online modification function.

The sequential control function block is activated via the Set input. The Reset input is used to reset the function block and call the initialisation function block specified at the InitStep input/output.

The StepInitNumber input defines the next step to be executed.

The Error output indicates the error code (further information below).

The StepOutputNumber output shows the number of the current step.

The StepToggle output shows the step transition. Normally this output is 1 or TRUE; only in the first cycle after a transition is the output assigned a 0 or FALSE.

Program sequence with the SFC_x function block
The use of the sequential control function block involves the following three program stages:

1. Initialising the sequential control function block
2. Processing the sequential control function block
3. Processing status signals of the sequential control function block

1. Initialising
The StepInputNumber input specifies which step should be processed. Before calling the sequence for the first time, the number of the first step to be processed should be assigned.

The sequential control function block is initialised with the Init function block. It is called for as long as the ReSet input is 1.

To implement an automatic initialisation with the Init function block use the status bit of the PLC for marking the first cycle. This will set the Reset input to 1 or TRUE during the first cycle.

The Init function block assigns the StepInputNumber input with the number of the first step to be processed. This therefore ensures that the number of the first step is known to the sequential control function block directly after the program is started.

The StepInputNumber input is then only changed via a global variable in the individual step function blocks (in the following example via the variable StepNumber). This ensures the clear logical sequencing of the function block.

Example: Initialising the SFC_3

```
PROGRAM sequence

VAR_GLOBAL
  StepNumber : USINT ;
  BasicSetting AT %I0.0.0.0.0 : BOOL ;
  SafetyDoorClosed AT %I0.0.0.0.1 : BOOL ;
END_VAR

VAR
  Sequence : SFC_3 ;
  First_Cycle : BOOL ;
  Initial : Init_FB ;
  Fill : FB1 ;
  Heat : FB2 ;
  Drain : FB3 ;
END_VAR

  LD ISA
  PLC_Message
  ST First_Cycle
  CAL Sequence(Set := TRUE,
           ReSet := First_Cycle,
           StepInputNumber := StepNumber,
           InitStep := Initial,
           Step_1 := Fill,
           Step_2 := Heat,
           Step_3 := Drain
           |
           :=Error,
           :=StepOutputNumber,
           :=StepToggle)

END_PROGRAM

FUNCTION_BLOCK Init_FB
VAR_EXTERNAL
  StepNumber : USINT ;
  BasicSetting : BOOL ;
  SafetyDoorClosed : BOOL ;
END_VAR
```

```
  LD BasicSetting
  AND SafetyDoorClosed
  (* If safety not fulfilled then return to
   program.
   The StepInputNumber = 0 => neutral*)
  RETCN
  LD 1     (* Step 1 "fill is defined
       as first step *)
  ST StepNumber
END_FUNCTION_BLOCK
```

2. Processing

In order to activate the function block the Set input must be 1 or TRUE. This activates simultaneously the step that is defined at the StepInputNumber input. If the Set and ReSet inputs are set to 1 at the same time, only the initialisation function block is processed. In the following examples, the Set input is permanently set to TRUE. The logical sequences before the RETC or RETCN commands are transitions that can be selected as required Step enabling conditions (= transitions).

The sequential control function block can call several actions. These actions are active for as long as the transition condition at the end of the function block concerned is fulfilled. If the transition condition is fulfilled the StepInputNumber input is assigned the number of the next step to be processed. In the following cycle the new step is automatically activated and the old one deactivated.

The following example tests one transition.

Example: Processing SFC_3

```
PROGRAM T_check

VAR_GLOBAL
  StepNumber : USINT ;
  BasicSetting AT %I0.0.0.0.0 : BOOL ;
  SafetyDoor_closed AT %I0.0.0.0.1 : BOOL ;
  OFF AT %I0.0.0.0.2 : BOOL ;
  Boiler_full AT %I0.0.0.0.3 : BOOL ;
  Temperature_reached AT %I0.0.0.0.4 : BOOL ;
```


07/01 AWB2700-1306-GB

```
  Valve_open AT %QO.0.0.0.0 : BOOL ;
  Heating_on AT %QO.0.0.0.1 : BOOL ;
END_VAR

VAR
  Sequence : SFC_3 ;
  First_Cycle : BOOL ;
  Initial : Init_FB ;
  Fill : FB1 ;
  Heat : FB2 ;
  Drain : FB3 ;
END_VAR

  LD ISA
  PLC_Message
  ST First_Cycle
  CAL Sequence(Set := TRUE,
          ReSet := First_Cycle,
          StepInputNumber := StepNumber,
          InitStep := Initial,
          Step_1 := Fill,
          Step_2 := Heat,
          Step_3 := Drain
          |
          :=Error,
          :=StepOutputNumber,
          :=StepToggle)

END_PROGRAM
```

```
FUNCTION_BLOCK FB1

VAR_EXTERNAL
  StepNumber : USINT ;
  OFF : BOOL ;
  Boiler_full : BOOL ;
  Valve_open : BOOL ;
END_VAR

  LD Boiler_full
  OR OFF
  RETC
  LD TRUE
  S Valve_open
  LD 2 (*Step "2" heat is defined as next
 step*)
  ST StepNumber

END_FUNCTION_BLOCK

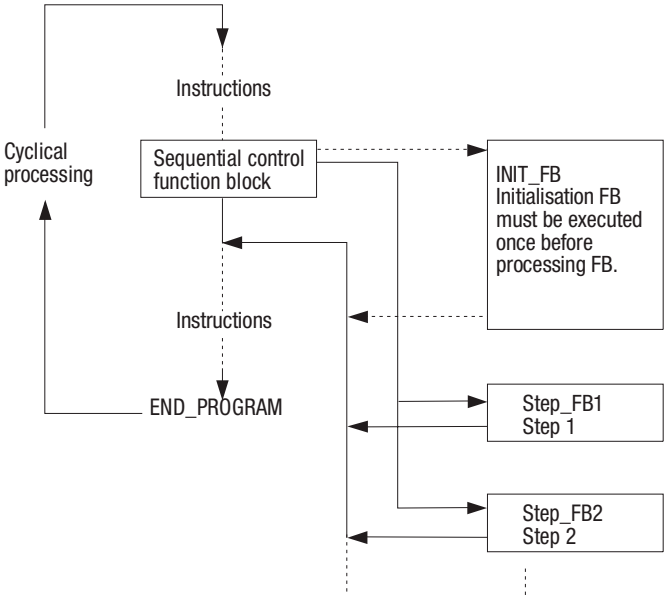
FUNCTION_BLOCK FB2

VAR_EXTERNAL
  StepNumber : USINT ;
  OFF : BOOL ;
  Heating_on : BOOL ;
  Temperature_reached : BOOL ;
  Boiler_full : BOOL ;
  Valve_open : BOOL ;
END_VAR

  LD Boiler_full
  OR OFF
  OR Temperature_reached
  RETC
  LD TRUE
  R Valve_open
  S Heating_on
  LD 3 (* Step 3 "drain" is defined as next
       step *)
  ST StepNumber

END_FUNCTION_BLOCK
```

Once a step function block has been processed, the main program is continued directly afterwards, irrespective of whether the transition was fulfilled or not. If the transition was fulfilled the step change will be executed in the next program cycle. If the transition condition was not fulfilled, the same step will be carried out as before.



3. Status indication

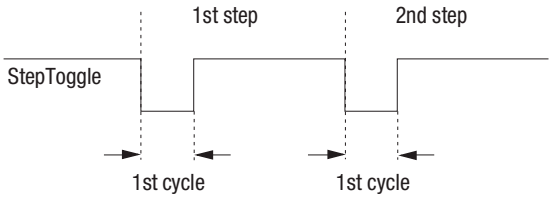
Various status outputs are provided for monitoring a step sequence. Error output indicates the following error codes during processing:

Error number	Cause of error	Error behaviour
1	0 present at the StepInput number input (when Set = 1).	Function block is not active. No step processed.
2	Step number entered exceeds maximum permissible number of steps.	Error output set. The function block stays in the current status.
4	No function block name at the parameter of the selected step	Error output set. The selected step is accepted. No processing present since no step function block entered.

The StepOutputNumber output indicates the current step number. If, for example, the output indicates the value 2, the step function block defined at Step_2 has been selected. The value 0 indicates that the next step selected is the initialisation step.

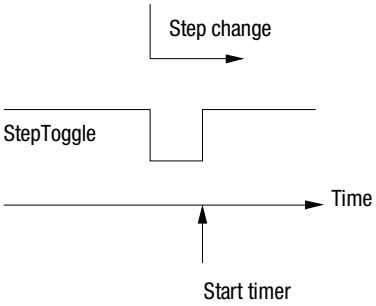
The StepToggle output indicates the transition from one step to the next. The output stays at 1 or TRUE for as long as the step is active. Only in the next cycle after the change to the new step is StepToggle set to 0 or FALSE.

07/01 AWB2700-1306-GB



The StepToggle output can be used to initialise individual steps. For example, a time monitoring function can be implemented. If a step is taking too long because a limit switch has not been reached or an electrical fault is present, an appropriate fault message can be output.

The StepToggle pulse is used to initialise a timer.



Example: SFC_3 for time monitoring

```
PROGRAM Steptime

VAR
   Step_Timer : TON ;
   Sequence : SFC_3 ;
    .
   .
   .END_VAR


   LD    ISA
   PLC.  MESSAGE
   ST    First_Cycle
   CAL   Step_Timer(IN := Sequence.StepToggle,
               PT := T#3s)    (* Step change starts *)
                      (* time set for 3s *)
   LD Step_Timer.Q
   JMPC Fault
   (* If the timer is not reset after 3s
         jump to a fault subroutine *)
   .

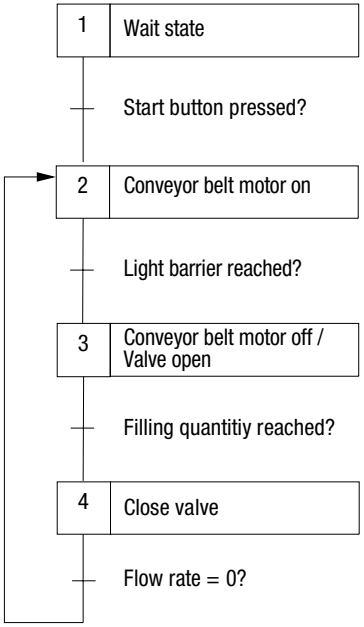
   .

END_PROGRAM
```


07/01 AWB2700-1306-GB

Other programming examples
When programming ensure that meaningful function block names, jump labels and variable names are used. This will simplify programming and the transparency of the programs. The following examples show typical applications with sequential control function blocks. The programs are designed for a paint filling plant.

Example: Linear sequence
Once the system has been switched on the control system is in wait mode. Only after the start button is pressed are the steps processed in succession. The program returns to Step 2 ConveyorMotor_On from Step 4 CloseValve.



```
PROGRAM linear

VAR_GLOBAL
  Sequence : SFC_4 ;
  Init : Init_FB ;
  FB_Start : Start_FB ;
  Motor_on : Motor_onFB ;
  Motor_off : Motor_offFB ;
  Valve_closed : ValveClFB ;
  First_Cycle : BOOL ;
  StepNumber : USINT ;
  Start AT %I0.0.0.0.0 : BOOL ;
  LightBarrier AT %I0.0.0.0.1 : BOOL ;
  Quantity AT %I0.0.0.0.2 : BOOL ;
  Flow AT %I0.0.0.0.3 : BOOL ;
  Motor AT %Q0.0.0.0.0 : BOOL ;
  FillingValve AT %Q0.0.0.0.1 : BOOL ;
END_VAR

  LD ISA
  PLC_Message
  ST First_Cycle
  CAL Sequence(Set := TRUE,
           ReSet := First_Cycle,
           StepInputNumber := StepNumber,
           InitStep := Init,
           Step_1 := FB_Start,
           Step_2 := Motor_on,
           Step_3 := Motor_off,
           Step_4 := Valve_closed)

END_PROGRAM

FUNCTION_BLOCK Init_FB

VAR_EXTERNAL
  StepNumber : USINT ;
END_VAR

  (*Initialisation of the function block and
   move to first step *)

  LD 1
  ST StepNumber

END_FUNCTION_BLOCK
```


07/01 AWB2700-1306-GB

```
FUNCTION_BLOCK Start_FB

VAR_EXTERNAL
  Start : BOOL ;
  StepNumber : USINT ;
END_VAR

  LD Start
  RETCN    (* Return to main program
           if start button not pressed *)
  LD 2     (* otherwise move to step two *)
  ST StepNumber

END_FUNCTION_BLOCK


FUNCTION_BLOCK MotonFB

VAR_EXTERNAL
  LightBarrier : BOOL ;
  Motor : BOOL ;
  StepNumber : USINT ;
END_VAR

  LD TRUE
  ST Motor (* Conveyor motor switched on *)
  LD LightBarrier

  RETC     (* Return if light barrier not
           interrupted*)
  LD 3     (* otherwise move to step three *)
  ST StepNumber

END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK MotoffFB

VAR_EXTERNAL
  Motor : BOOL ;
  Quantity : BOOL ;
  FillingValve : BOOL ;
  StepNumber : USINT ;
END_VAR

(*Step stops conveyor motor and opens filling
  valve *)
  LD FALSE
  ST Motor (*1st action motor off *)
  LD TRUE
  ST FillingValve (*2nd action: filling valve
        *)
  LD Quantity
  RETCN(*No move to step 4 if *)
       (*fill quantity not yet reached *)
  LD 4
  ST StepNumber

END_FUNCTION_BLOCK



FUNCTION_BLOCK ValveClFB

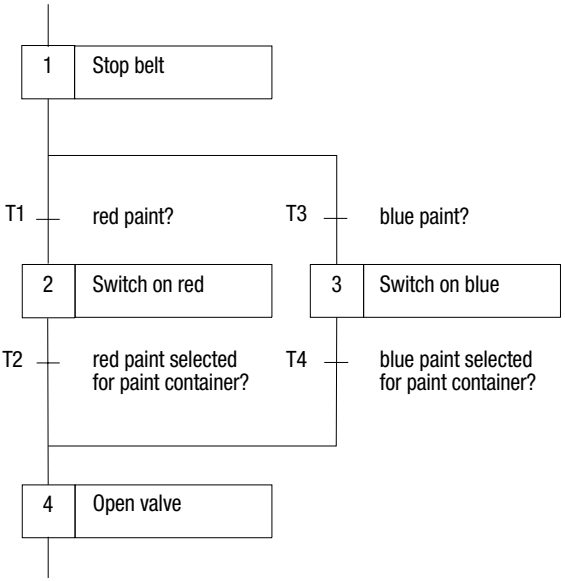
FUNCTION_BLOCK ValveClFB
VAR_EXTERNAL
  Flow : BOOL ;
  FillingValve : BOOL ;
  StepNumber : USINT ;
END_VAR

  LD FALSE
  ST FillingValve    (*Valve closed *)
  LD Flow
  RETC               (*When flow really 0 then
                     step 2 again*)
  LD 2
  ST StepNumber

END_FUNCTION_BLOCK
```


07/01 AWB2700-1306-GB

Example: OR Sequence
Red or blue paint is to be filled alternately. A selector can be used to select either the red or blue containers.



A sequential control function block must be used for the OR sequence since only one sequence can be run at a time. As soon as one of the two transitions (T1 or T3) is fulfilled, the sequence proceeds to the appropriate branch. This branch can then no longer be changed, even if the other transition was then fulfilled later.

Transitions T1 and T3 are programmed at the end of the first step (FB “Motor_off”). A jump table is created for this purpose. The step number to be processed is then defined by loading the values 2 or 3.

```
PROGRAM or1

VAR_GLOBAL
  Sequence : SFC_4 ;
  Init : Init_FB ;
  Motor_off : MotoffFB ;
  Selector_red : RED_FB ;
  Selector_blue : BLUE_FB ;
  Valve_open : Valve_FB ;
  First_cycle : BOOL ;
  StepNumber : USINT ;
  Paint_RED AT %I0.0.0.0.0 : BOOL ;
  Paint_BLUE AT %I0.0.0.0.1 : BOOL ;
  Tank_RED AT %I0.0.0.0.2 : BOOL ;
  Tank_BLUE AT %I0.0.0.0.3 : BOOL ;
  PaintSelector AT %Q0.0.0.0.0 : BOOL ;
  FillingValve AT %Q0.0.0.0.1 : BOOL ;
  Motor AT %Q0.0.0.0.2 : BOOL ;
END_VAR

  LD ISA
  PLC_Message
  ST First_cycle
  CAL Sequence(Set := TRUE,
           ReSet := First_cycle,
           StepInputNumber := StepNumber,
           InitStep := Init,
           Step_1 := Motor_off,
           Step_2 := Selector_red,
           Step_3 := Selector_blue,
           Step_4 := Valve_open)

END_PROGRAM

FUNCTION_BLOCK Init_FB

VAR_EXTERNAL
  StepNumber : USINT ;
END_VAR
  (*Initialise function block and move to
   first step *)
  LD 1
  ST StepNumber

END_FUNCTION_BLOCK
```


07/01 AWB2700-1306-GB

```
FUNCTION_BLOCK MotoffFB

VAR_EXTERNAL
  Motor : BOOL ;
  Paint_RED : BOOL ;
  Paint_BLUE : BOOL ;
  StepNumber : USINT ;
END_VAR

  (* Step stops conveyor motor and opens
       filling valve *)
  LD FALSE
  ST Motor (*Action: Motor off*)
  LD Paint_RED
  JMPCN Not_RED      (*Test colour required RED *)
  LD 2               (*Move to step 2 if RED*)
  ST StepNumber

Not_RED:
  LD Paint_BLUE
  RETCN              (*Test colour wish BLUE *)
  LD 3               (*If BLUE move to step 3*)
  ST StepNumber

END_FUNCTION_BLOCK


FUNCTION_BLOCK RED_FB

VAR_EXTERNAL
  PaintSelector : BOOL ;
  Tank_RED : BOOL ;
  StepNumber : USINT ;
END_VAR

  (*PaintSelector set to RED *)
  LD TRUE
  ST PaintSelector
  LD Tank_RED
  RETCN    (*Return if tank empty *)
  LD 4     (*Continue to valve opening *)
  ST StepNumber

END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK BLUE_FB

VAR_EXTERNAL
  PaintSelector : BOOL ;
  Tank_BLUE : BOOL ;
  StepNumber : USINT ;
END_VAR

(*PaintSelector set to BLUE *)
  LD FALSE
  ST PaintSelector
  LD Tank_BLUE
  RETCN    (*Return if tank empty *)
  LD 4     (*Continue to valve opening *)
  ST StepNumber

END_FUNCTION_BLOCK


FUNCTION_BLOCK Valve_FB

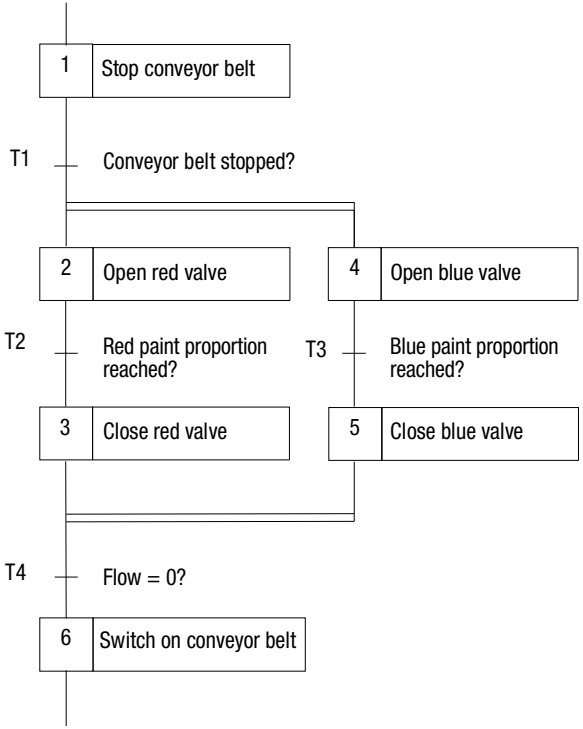
VAR_EXTERNAL
  FillingValve : BOOL ;
  StepNumber : USINT ;
END_VAR

  LD TRUE
  ST FillingValve
  (*Further actions could be followed
   here then transition condition for
   a next step *)

END_FUNCTION_BLOCK
```


07/01 AWB2700-1306-GB

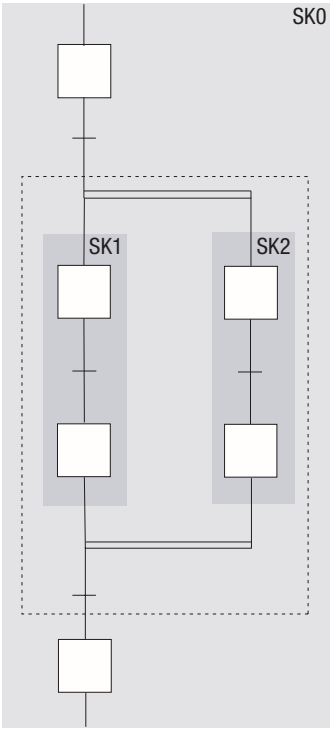
Example: AND Sequence with Synchronisation
Only one colour mix is to be produced. This example mixes the paint by filling both paint components via two different valves at the same time. The metering is variable according to the shade of colour required.



With the AND sequence the entire sequence is divided into several parallel sequences. Each individual branch requires its own sequential control function block that is called by the central sequence.

The parallel branches are combined by a synchronisation function.

Before the transition to the next step of the lower sequence, a scan is made to check whether both parallel branches are processing their last step.



SK1 and SK2
form step 2 of SK0

07/01 AWB2700-1306-GB

```
PROGRAM ANDsequence

VAR_GLOBAL
  MainSequence : SFC_3 ;
  Main_Init : Init_FB ;
  Sequence_1 : SFC_2 ;
  Init_Sequence_1 : Init_1 ;
  Sequence_2 : SFC_2 ;
  Init_Sequence_2 : Init_2 ;
  Motor_on : MotonFB ;
  Motor_off : MotoffFB ;
  Red_open : REDopenFB ;
  Blue_open : BLUEopenFB ;
  Red_closed : REDclosedFB ;
  Blue_closed : BLUEclosedFB ;
  First_Cycle : BOOL ;
  StepNumber : USINT ;
  Speed : UINT ;
  Supply AT %I0.0.0.0.0 : BOOL ;
  RedProportion AT %I0.0.0.0.1 : BOOL ;
  BlueProportion AT %I0.0.0.0.2 : BOOL ;
  FillingValve_red AT %Q0.0.0.0.0 : BOOL ;
  FillingValve_blue AT %Q0.0.0.0.1 : BOOL ;
  Conveyormotor AT %Q0.0.0.0.2 : BOOL ;
  Mix : MixFB
END_VAR

  LD ISA
  PLC_Message
  ST First_Cycle
  CAL MainSequence(Set := TRUE,
                   ReSet := First_Cycle,
                   StepInputNumber := StepNumber,
                   InitStep := Main_Init,
                   Step_1 := Motor_off,
                   Step_2 := MixFB,
                   Step_3 := Motor_on

END_PROGRAM
```

```
FUNCTION_BLOCK Init_FB

VAR_EXTERNAL
  StepNumber : USINT ;
END_VAR

  (*Initialise the function block and move to
   next step *)
  LD 1
  ST StepNumber

END_FUNCTION_BLOCK



FUNCTION_BLOCK MotoffFB

VAR_EXTERNAL
  ConveyorMotor : BOOL ;
  Speed : UINT ;
  StepNumber : USINT ;
END_VAR

  (* Step switches off the conveyor motor and
      checks when the conveyor has stopped*)
  LD FALSE
  ST Motor (*Action: Motor off*)
  LD Speed
  NE 0     (* Check whether conveyor has really stopped*)
  RETC     (* If yes, then to main *)
  LD 2     (* sequence, step 2*)
  ST StepNumber

END_FUNCTION_BLOCK



FUNCTION_BLOCK MixFB

VAR_EXTERNAL
  MainSequence : SFC_3 ;
  Main_Init : Init_FB ;
  Sequence_1 : SFC_2 ;
  Init_Sequence_1 : Init_1 ;
  Sequence_2 : SFC_2 ;
  Init_Sequence_2 : Init_2 ;
  Motor_on : MotonFB ;
  Motor_off : MotoffFB ;
  Conveyormotor : BOOL ;
```


07/01 AWB2700-1306-GB

```
  Red_open : REDopenFB ;
  Red_closed : REDclosedFB ;
  Blue_open : BLUEopenFB ;
  Blue_closed : BLUEclosedFB ;
  First_Cycle : BOOL ;
  StepNumber : USINT ;
  Speed : UINT ;
  Supply : BOOL ;
  FillingValve_red : BOOL ;
  FillingValve_blue : BOOL ;
END_VAR

VAR
  Reset_Subsequence : BOOL ;
END_VAR

  LD MainSequence.StepToggle
  STN Reset_Subsequence (*Reset if
       MainSequence does not toggle *)

CAL Sequence_1(Set := TRUE,
           ReSet := Reset_Subsequence,
           StepInputNumber := ,
           InitStep := Init_Sequence_1,
           Step_1 := Red_open,
           Step_2 := Red_closed,CAL Sequence_2(Set :=
TRUE,
           ReSet := Reset_Subsequence,
           StepInputNumber := ,
           InitStep := Init_Sequence_2,
           Step_1 := Blue_open,
           Step_2 := Blue_closed)
  (*Synchronisation: Sequence 1 already
       completed ?*)
  LD Sequence_1.StepOutputNumber
  EQ 2
  JMPCN End
```

```
  (*Synchronisation: Sequence 2 already
       completed ?*)
  LD Sequence_2.StepOutputNumber
  EQ 2
  JMPCN End
       (* Supply = 0 ?*)
  LD Supply
  JMPCN End
       (*Move to third step of MainSequence *)
  LD 3
  ST MainSequence.StepInputNumber

End:

END_FUNCTION_BLOCK

FUNCTION_BLOCK MotonFB

VAR_EXTERNAL
  Conveyormotor : BOOL ;
END_VAR

  LD 1
  ST Conveyormotor
  .
  .
  .

END_FUNCTION_BLOCK



FUNCTION_BLOCK Init_1

VAR_EXTERNAL
  Sequence_1 : SFC_2 ;
END_VAR

  LD 1
  ST Sequence_1.StepInputNumber

END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK REDopenFB

VAR_EXTERNAL
  Sequence_1 : SFC_2 ;
  FillingValve_red : BOOL ;
  RedProportion : BOOL ;
END_VAR

       (* Open valve for red*)
  LD 1
  ST FillingValve_red
  LD RedProportion
  JMPCN End              (*If RedProportion not yet
reached, continue filling*)
       (* Otherwise move to next step in sequence *)
  LD 2
  ST Sequence_1.StepInputNumber

End:

END_FUNCTION_BLOCK


FUNCTION_BLOCK REDclosedFB

VAR_EXTERNAL
  FillingValve_red : BOOL ;
END_VAR

  LD 0
  ST FillingValve_red

END_FUNCTION_BLOCK


FUNCTION_BLOCK Init_2

VAR_EXTERNAL
  Sequence_2 : SFC_2 ;
END_VAR

  (* Move to first cycle of
       send sequence;
       Initialisation*)
  LD 1
  ST Sequence_2.StepInputNumber

END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK BLUEopenFB

VAR_EXTERNAL
  Sequence_2 : SFC_2;
  FillingValve_blue : BOOL ;
  BlueProportion : BOOL ;
END_VAR

       (* Open valve for blue*)
  LD 1
  ST FillingValve_blue
  LD RedProportion
  JMPCN End       (*If BlueProportion not yet reached,
continue filling*)
       (* Otherwise move to next step in sequence *)
  LD 2
  ST Sequence_2.StepInputNumber

End:

END_FUNCTION_BLOCK


FUNCTION_BLOCK BLUEclosedFB

VAR_EXTERNAL
  FillingValve_blue : BOOL ;
END_VAR

  LD 0
  ST FillingValve_blue

END_FUNCTION_BLOCK
```


07/01 AWB2700-1306-GB

IEC

SR
Flip Flop with Set Dominant



Function block prototype

Meaning of operands

SET1	Set condition
RESET	Reset condition
Q1	Output state of the bistable element

Description

The SR function block is used to statically set a data element – the output Q1 – to the Boolean status 1 or 0.

Switching between the two states depends on the value of the Boolean input operands SET1 and RESET.

The output Q1 is initialised with the value 0 when starting the process. The first processing of the function block with the value 1 of the operand SET1 causes the output Q1 to be set to 1. A change of the value of SET1 no longer then effects the output Q1. The value 1 at the input operand RESET sets the output Q to 0 – the output is reset.

If both input operands have the value 1, the fulfilled set condition will be dominant, i.e. Q1 is set with priority.

Invocation of the function SR in the program “malfunction”

```
PROGRAM malfunction

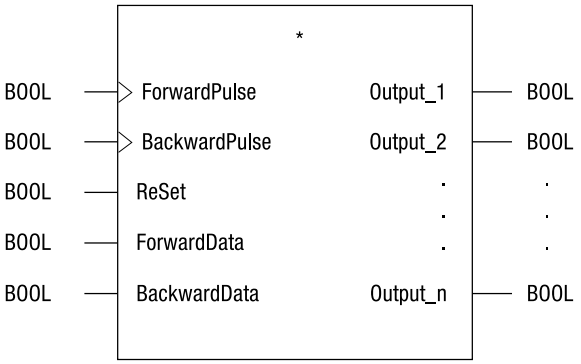
VAR
   Malfunction1_message : SR;
   Malfunction1 AT %I0.0.0.0.0 : BOOL;
   Acknowledge_malfunction AT %I0.0.0.0.1 : BOOL;
   Display_malfunction1 AT %Q0.0.0.0.6 : BOOL;
END_VAR

   CAL   Malfunction1_message (SET1 := Malfunction1,
                         RESET := Acknowledge_malfunction)
   LD    Malfunction1_message.Q1
   ST    Display_malfunction1

END_PROGRAM
```


07/01 AWB2700-1306-GB

SR_x
1-Bit Shift Register



Function block prototype

Meaning of operands

ForwardPulse	Pulse input, forward
BackwardPulse	Pulse input, backward
ReSet	Reset
ForwardData	Data input, forward
BackwardData	Data input, backward
Output_1	Output 1
Output_n	Last output

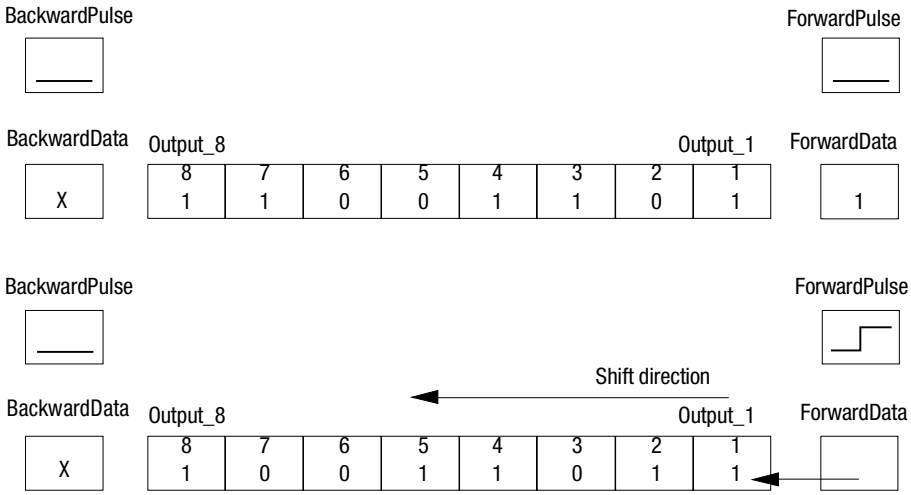
Description

There are 128 prototypes of the function block. The following names can be used for *:

SR_1
SR_2
.
.
SR_128

The numbers 1 ... 128 indicate the register length, the number of 1-bit output register fields of the individual prototypes.

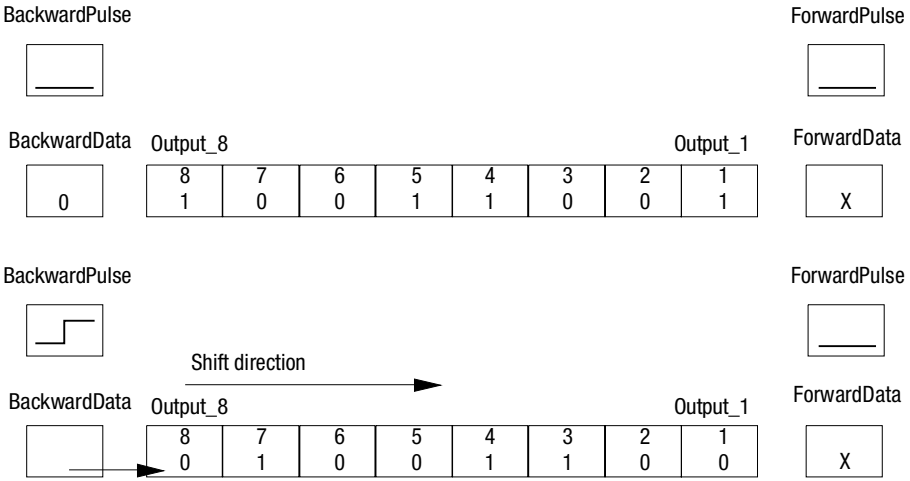
A rising edge on the ForwardPulse operand transfers the value at the ForwardData operand to the first register field Output_1. The original contents of the register fields are moved by one field in the direction of the higher field numbers.



Contents of the SR_8 shift register before and after a Forward pulse

07/01 AWB2700-1306-GB

A rising edge on the BackwardPulse operand transfers the value at the BackwardData operand to the last register field. The original contents of the register fields are moved by one field in the direction of the lower field numbers.



Contents of the SR_8 shift register before and after a Backward pulse

The status 1 of the ReSet clears the entire Output_1 ... Output_n output register.

Invocation of the function block SR_4 in the program sh_reg1

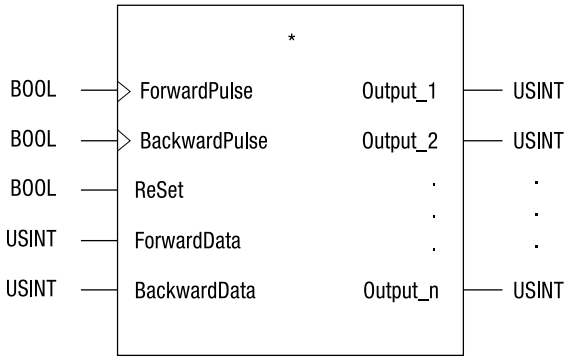
```
PROGRAM sh_reg1

VAR
   Shift_register_4_bit : SR_4;
   Pulse1 : BOOL;
   Pulse2 : BOOL;
   Reset : BOOL;
   Data_bit1 : BOOL;
   Data_bit2 : BOOL;
   Output1 : BOOL;
   Output2 : BOOL;
   Output3 : BOOL;
   Output4 : BOOL;
END_VAR

   CAL Shift_register_4_bit (ForwardPulse := Pulse1,
                             BackwardPulse := Pulse2,
                             ReSet := Reset,
                             ForwardData := Data_bit1,
                             BackwardData := Data_bit2)
   LD Shift_register_4_bit.Output_1
   ST Output1
   LD Shift_register_4_bit.Output_2
   ST Output2
   LD Shift_register_4_bit.Output_3
   ST Output3
   LD Shift_register_4_bit.Output_4
   ST Output4END_PROGRAM
```


07/01 AWB2700-1306-GB

SRB_x
8-Bit Shift Register



Function block prototype

Meaning of operands

ForwardPulse	Pulse input, forward
BackwardPulse	Pulse input, backward
ReSet	Reset
ForwardData	Data input, forward
BackwardData	Data input, backward
Output_1	Output 1
Output_n	Last output

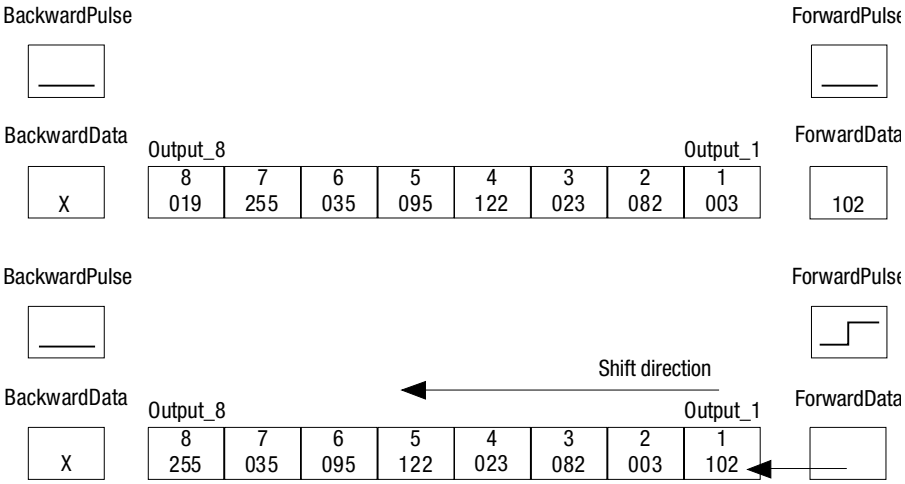
Description

There are 128 prototypes of the function block. The following names can be used for *:

SRB_1
SRB_2
.
.
SRB_128

The numbers 1 ... 128 indicate the register length, the number of 8-bit output register fields of the individual prototypes.

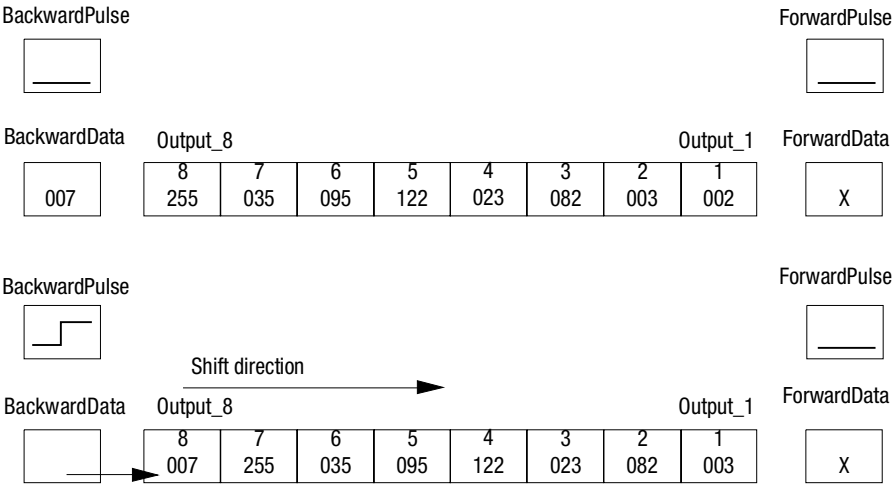
A rising edge on the ForwardPulse operand transfers the value at the ForwardData operand to the first register field Output_1. The original contents of the register fields are moved by one field in the direction of the higher field numbers.



Contents of the SRB_8 shift register before and after a Forward pulse

07/01 AWB2700-1306-GB

A rising edge on the BackwardPulse operand transfers the value at the BackwardData operand to the last register field. The original contents of the register fields are moved by one field in the direction of the lower field numbers.



Contents of the SRB_8 shift register before and after a Backward pulse

The status 1 of the ReSet clears the entire Output_1 ... Output_n output register.

Invocation of the function block SRB_2 in the program sh_reg8

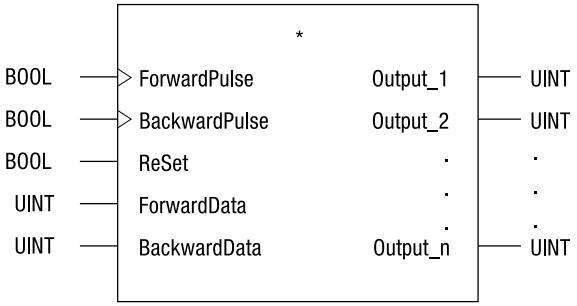
```
PROGRAM sh_reg8

VAR
   Shift_register_2_byte : SRB_2;
   Pulse1 : BOOL;
   Pulse2 : BOOL;
   Res : BOOL;
   Data1 : USINT;
   Data2 : USINT;
   Output1 : USINT;
   Output2 : USINT;
END_VAR

   CAL Shift_register_2_byte (ForwardPulse := Pulse1,
                              BackwardPulse := Pulse2,
                              ReSet := Res,
                              ForwardData := Data1,
                              BackwardData := Data2)
   LD Shift_register_2_byte.Output1
   ST Output1
   LD Shift_register_2_byte.Output2
   ST Output2END_PROGRAM
```


07/01 AWB2700-1306-GB

SRW_x
16-Bit Shift Register



Function block prototype

Meaning of operands

ForwardPulse	Pulse input, forward
BackwardPulse	Pulse input, backward
ReSet	Reset
ForwardData	Data input, forward
BackwardData	Data input, backward
Output_1	Output 1
Output_n	Last output

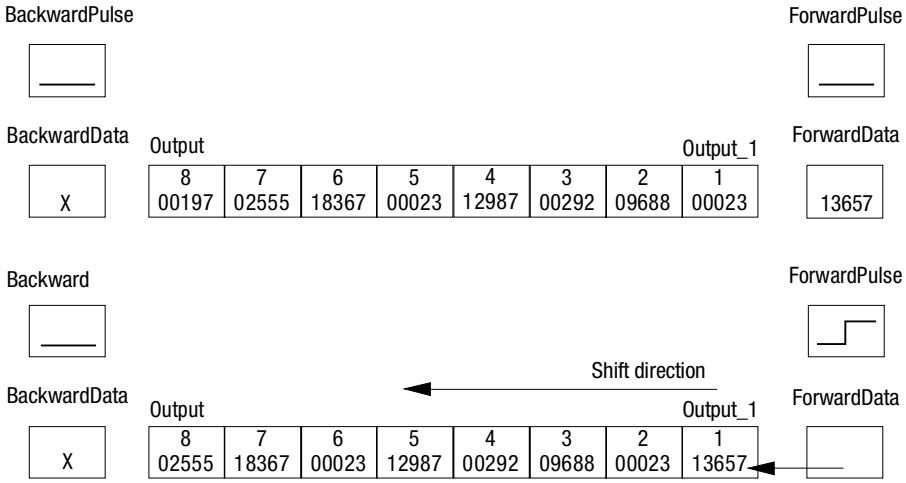
Description

There are 128 function block prototypes. The following names can be used for *:

SRW_1
SRW_2
.
.
SRW_128

The numbers 1 ... 128 indicate the register length, the number of 16-bit output register fields of the individual prototypes.

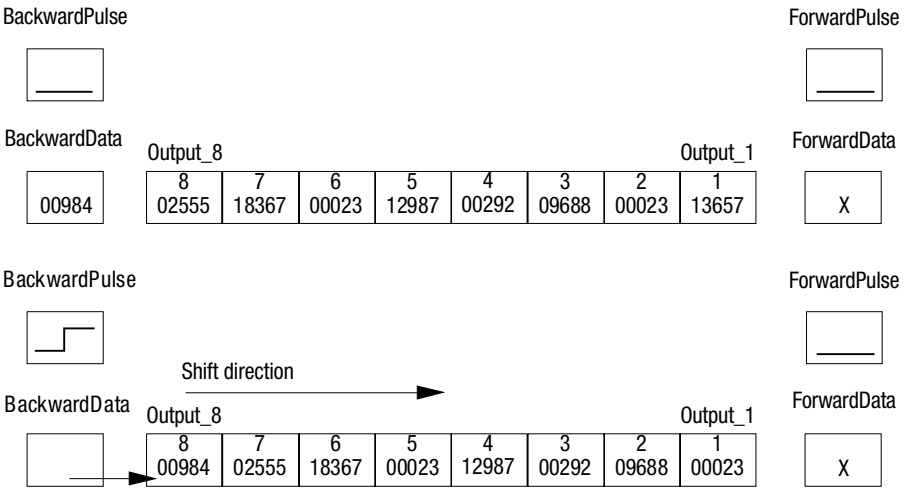
A rising edge on the ForwardPulse operand transfers the value at the ForwardData operand to the first register field Output_1. The original contents of the register fields are moved by one field in the direction of the higher field numbers.



Contents of the SRW_8 shift register before and after a Forward pulse

07/01 AWB2700-1306-GB

A rising edge on the BackwardPulse operand transfers the value at the BackwardData operand to the last register field. The original contents of the register fields are moved by one field in the direction of the lower field numbers.



Contents of the SRW_8 shift register before and after a Backward pulse

The status 1 of the ReSet clears the entire Output_1 ... Output_n output register.

Invocation of the function block SRW_2 in the program s_reg16

```
PROGRAM s_reg16

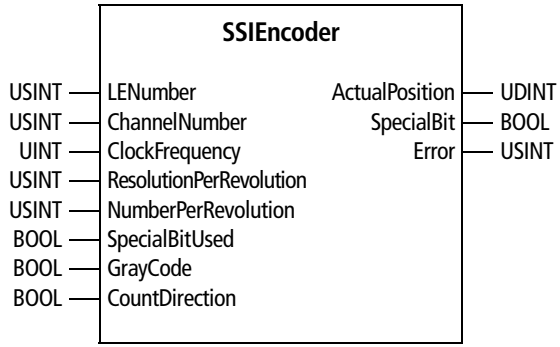
VAR
   Shift_register_2_word : SRW_2;
   Pulse1 : BOOL;
   Pulse2 : BOOL;
   Res : BOOL;
   Data1 : UINT;
   Data2 : UINT;
   Output1 : UINT;
   Output2 : UINT;
END_VAR

   CAL Shift_register_2_word (ForwardPulse := Pulse1,
                              BackwardPulse := Pulse2,
                              ReSet := Res,
                              ForwardData := Data1,
                              BackwardData := Data2)
   LD Shift_register_2_word.Output1
   ST Output1
   LD Shift_register_2_word.Output2
   ST Output2

END_PROGRAM
```


07/01 AWB2700-1306-GB

SSIEncoder
Control of LE 4-633-CX1 Local Expansion Module



Function block prototype

Meaning of operands

LENumber	Module address of LE 1 = 1st LE; 2 = 2nd LE
ChannelNumber	Channel number 1 = Channel 1; 2 = Channel 2; 3 = Channel 3
ClockFrequency	Clock output frequency in kHz125 = 125 kHz; 250 = 250 kHz
ResolutionPer-Revolution	Bit no. of increments per revolution of SSI absolute value encoder
NumberPer-Revolution	Bit no. of revolutions of SSI absolute value encoder
SpecialBitUsed	Special bit of SSI absolute value encoder 0 = No special bit;1 = Special bit present
GrayCode	Code used by SSI absolute value encoder 0 = binary code; 1 = Gray code
CountDirection	Change counting direction 0 = no counter direction change1 = Counter direction change

ActualPosition	Current position value
SpecialBit	With special bit active (SpecialBitUsed =1), the value of the special bit is output
Error	Error messages 0 = No error 1 = LE4-633-CX1 not present 2 = Incorrect parameters 4 = Wire break on SSI absolute value encoder

Description

The SSIEncoder function block controls the LE4-633-CX1. The following functions are possible:

Set LE number
Set channel number
Set absolute value encoder parameters

Scan actual position
Scan special bit
Scan error messages

Each LE channel can be assigned one function block instance. The LENumber and ChannelNumber are used to specify the SSI absolute encoder for which the current position is to be scanned. The current position value is updated every time the SSIEncoder function block is processed.

07/01 AWB2700-1306-GB

Example “Control of LE 4-633-CX1 local expansion module”

An SSI absolute encoder is connected to channel no. 1 of the 1st LE. The SSI absolute encoder has the following parameters:

Increments per revolution: 4096 (12 Bit)

No. of revolutions: 4096 (12 Bit)

Code: Gray

Special bit: no

```
PROGRAM SSIAbsoluteValue

VAR

   ssiLE1_Channel1 :       SSIEncoder ;
   ActualPosition24Bit :   UDINT ;
   Error :                 USINT ;
END_VAR

 CAL ssiLE1_Channel1(
   LENumber :=1,                  (* 1st LE                           *)
   ChannelNumber :=1,             (* Channel no. 1                    *)
   ClockFrequency :=125,          (* Clock frequency = 125 kHz        *)
   ResolutionPerRevolution :=12,  (* No. of bits per revolution =12-bit*)
   NumberPerRevolution :=12,      (* No. of bits per revolution =12-bit*)
   SpecialBitUsed :=0,            (* No special bit                   *)
   GrayCode :=1,                  (* Graycode                         *)
   CountDirection :=0             (* No direction change              *)
   )

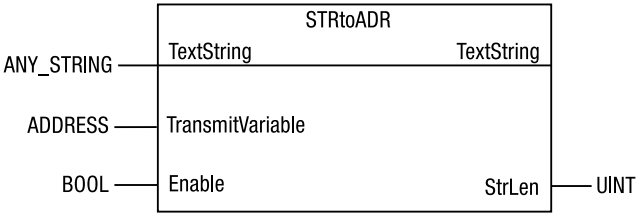
   ld    ssiLE1_Channel1.Error
   st    Error

   ld    ssiLE1_Channel1.ActualPosition
   st    ActualPosition24Bit

END_PROGRAM
```

PS4-200
PS4-300

STRtoADR
Copy String to Marker Area



Function block prototype

Meaning of operands

TextString	String to be stored in the marker range
TransmitVariable	Start address of the marker range in which the data is to be stored
Enable	Number of converted characters
StrLen	Enable of conversion process

Description

The STRtoADR function block is primarily used to copy a string to a marker area with the start address TransmitVariable. The SCO function block which requires the same start address as an input parameter then outputs this string through a serial interface.

A 1 or TRUE on the Enable input starts the conversion. The output parameter StrLen indicates the number of characters which were copied to the marker area and can pass on the amount of data to be transferred to a communication function block such as SCO.

If the length of TextString is longer than the declared marker area, the string is truncated.

07/01 AWB2700-1306-GB

Invocation of the STRtoADR function block in the Textcopy program

```
PROGRAM Textcopy

VAR
   String_to_Marker : STRtoADR;
   SendText : STRING := 'Small Example';
   StartAddress AT %MB0.0.0.100 : BYTE;
   SendLength : UINT;

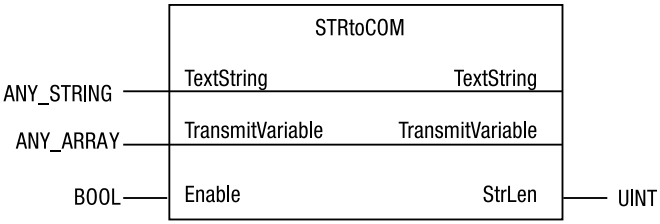
END_VAR

   CAL String_to_Marker(Enable := 1,
                   TextString := SendText,
                   TransmitVariable := &StartAddress)

         (* String is stored in the marker area from MB100 *)
   LD String_to_Marker.StrLen
   ST SendLength

END_PROGRAM
```

STRtoCOM
Copy String to Array



Function block prototype

Meaning of operands

TextString	String to be stored in the array
TransmitVariable	Start address of the array in which the data is to be stored
Enable	Number of converted characters
StrLen	Number of converted characters

Description

The STRtoCOM function block is primarily used to store a string from the start address TransmitVariable to an array. The COM or SCO function block which requires the same start address as an input parameter then outputs this string on a serial interface. An ASCII conversion of the characters in the function block is not carried out. A 1 or TRUE on the Enable input starts the conversion. The output parameter StrLen indicates the number of characters copied to the marker range and passes on the number of data units to be transferred to a communication function block such as COM or SCO. If the length of the string is longer than the declared marker area, the string is truncated when copied and StrLen shows the size of the array.

07/01 AWB2700-1306-GB

Invocation of the function block STRtoCOM in the program textcopy

```
PROGRAM Textcopy

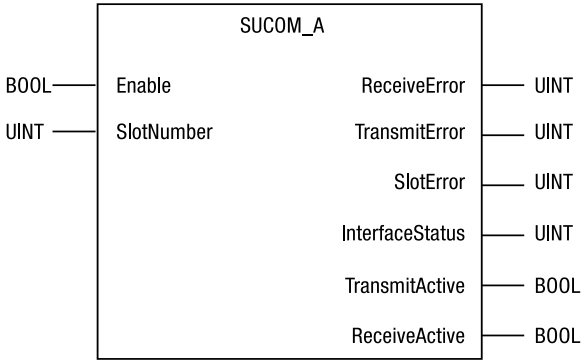
VAR
   String_to_Array : STRtoCOM;
   SendText : STRING := 'Small Example';
   SendArray : ARRAY [1..100] OF BYTE;
   SendLength : UINT;
END_VAR

   CAL String_to_ Array(Enable := 1,
                        TextString := SendText,
                        TransmitVariable := SendArray
                        |
                        SendLength :=.StrLen)

END_PROGRAM
```

PS4-300
PS416

SUCOM_A
Protocol Emulation for Direct Access



Function block prototype

Meaning of operands

Enable	Enable
SlotNumber	Slot of the assigned card 0 PS16-CPU-400/300 or PS 4-341-MM1 with SBI interface 4...20 PS416-COM-200 with RS 232 interface
ReceiveError	Error message on receive job
TransmitError	Error message on send job
SlotError	Error message when no SUCOM-A compatible card is fitted at the addressed slot.
InterfaceStatus	Indicates the operating status of the card and the fitted modules
TransmitActive	FB is processing Send job when "1"
ReceiveActive	FB is processing Receive job when "1"

07/01 AWB2700-1306-GB

Description

The following interfaces can be used to access the marker, status and diagnostic bits of the PS416 controller:

- PRG interface of a PS416-CPU-400/300/200 card
- SBI interface of PS416-CPU-400/-300 card (Transparent mode)
- SBI interface of the PS4-341-MM1 (Transparent mode)
- RS 232 interface of a PS416-COM-200 card

Access takes place through the proprietary Moeller SUCOM-A protocol using external software. The marker area can be read and written too. Status and diagnostic bits can only be read.

The SUCOM-A protocol is detected automatically on the PRG interface of the PS416-CPU-400/300/200 card by the operating system if the interface is configured for 9600 baud.

If you are using the SBI interface of the PS416-CPU or the RS 232 interface of the PS416-COM-200 card, it is first necessary to initialize the interface accordingly. Specify in the Interface Parameters window of the Topology Configurator the Baud rate, Data format, Parity and Stop bits parameters for the connected device.

If data communication in a PS416 system takes place both through the SBI interface of the CPU card and also through one or more PS416-COM-200 cards, it is necessary to program separate instances of the SUCOM_A function block for each of the interfaces and to specify the respective slot number. The slot number is specified with SlotNumber. Specify 0 for an SBI interface; specify the appropriate slot number for the PS416-COM-200 card(s).

The SUCOM_A emulation can be operated simultaneously on the SBI interface and up to 6 PS416-COM-200 cards.

In order to activate the SUCOM_A emulation for a single interface, you only need to call the respective instance of the function block once in your program with Enable = 1. The emulation remains active until the function block is called again with Enable = 0. However, the function block outputs are only updated if the function block is called cyclically with Enable = 1.

The reaction time is dependent on the number of simultaneously active interfaces and the interface being used:

PRG interface of the PS416-CPU:	approx. 30 ms
SBI interface of the PS416-CPU:	approx. 100 ms
interface of the PS416-COM-200:	approx. 200 ms

The times apply to an activated interface.

The reaction time is not affected if the SUCOM_A function block is called more than once within the same program cycle.

The external software should be programmed to wait for a response from the PLC or to have a timeout value of one second, after which the request should be repeated if necessary.

The function block outputs ReceiveError, TransmitError and SlotError can be used for diagnostics. If there are no errors, all three outputs are set to 0. If an error occurs, the associated error and status messages are output. The codes of the messages are identical to those of the PS416-COM-200 and can be found in the manual “PS416-COM-200 Point-to-Point Communication Card” (AWB 27-1237-GB) in the section “Test and Commissioning”.

The UM1.5 interface converter can be used to adapt the signal levels between the RS 232 and RS 485 (i.e. SBI interface).



If the function block is instanced more than once in your program and/or the markers in your program are also used for other purposes, e.g. for PROFIBUS communication, take care to avoid overlaps of the marker areas to avoid unintentional changes to data.

Invocation of the SUCOM_A function block for a card in slot 6 in the protocol program

```
PROGRAM protocol

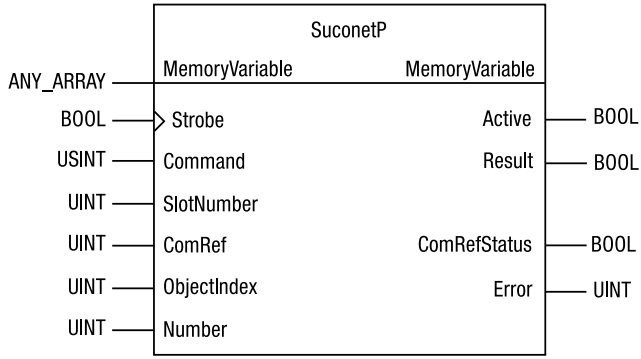
VAR
  Sucom_A_FB : SUCOM_A;
  Enable_SucomA AT %I0.0.0.0.0 : BOOL;
END_VAR

  CAL Sucom_A_FB(Enable := Enable_SucomA,
              SlotNumber := 6)

END_PROGRAM
```

PS416

SuconetP
PROFIBUS-FMS Communication Function Block



Function block prototype

Description

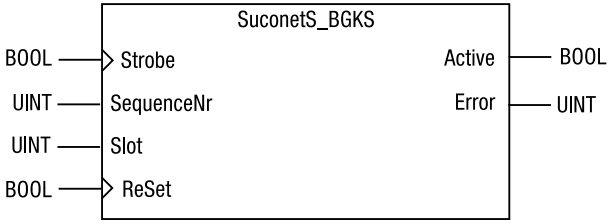
The SuconetP function block is used to handle PROFIBUS communication via the PROFIBUS-FMS card PS416-NET-230.

Please refer to the manual “Additional Information on PROFIBUS, PS416-NET-230 - Suconet-P Card” (AWB 27-1235-GB) for a description of this function block and some application notes.

If you are importing a Sucosoft S30 program to S40 that contains a Suconet P function block, inconsistencies may occur in the data array addressed via MemoryVariable due to the different operating systems of the PS316 and PS416. These inconsistencies can be prevented by not entering new data in the array with a Write job and by not reading data during a Read job when Active = 1.

07/01 AWB2700-1306-GB

SuconetS_BGKS
INTERBUS Communication



Function block prototype

Meaning of operands

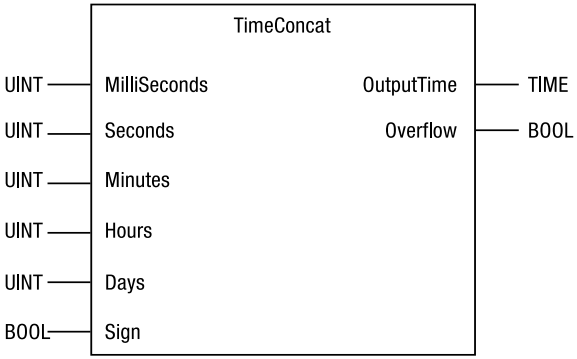
Strobe	Start bit-controlled command sequence
SequenceNr	Bit number of the command sequence (value range 0-15 decimal)
Slot	Slot for PS416-NET-220 card (value 4-19 decimal)
ReSet	Initial status
Active	Execution of command sequence active
Error	Error message

Description

The SuconetS_BGKS function block is used to access the bit-mapped command sequences of the PS416-NET-220 card.

Please refer to the manual “Hardware and Engineering, PS416-NET-220, Suconet-S Card” (AWB27-1236-GB) for a description of this function block.

TimeConcat
Generate Time Duration



Function block prototype

Meaning of operands

MilliSeconds	Milliseconds part of a time duration
Seconds	Seconds part of a time duration
Minutes	Minutes part of a time duration
Hours	Hour part of a time duration
Days	Day part of a time duration
Sign	Sign of a time duration
OutputTime	Resulting value of time duration
Overflow	Overflow signal

Description

The individual time elements that are specified as input operands are combined to form a time duration value of data type TIME in the OutputTime output operand.

The function block converts the individual values into the appropriate component of the time duration concerned and also forms a carry if necessary. The

07/01 AWB2700-1306-GB

input variables can take any value permissible for UINT data types provided that the total result does not exceed T#63d23h59m59s999ms. If this value is exceeded, the Overflow output will indicate a 1.

The sign of bit of the Sign input operand affects the result: Sign = 0 positive value, Sign = 1 negative value of the OutputTime operand.

Invocation of the function block TimeConcat in the program “time_gen”

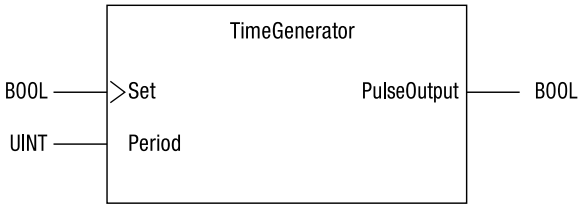
```
PROGRAM time_gen

VAR
   TimeConcaten : TimeConcat;
   Millisec : UINT;
   Sec : UINT;
   Min : UINT;
   Hrs : UINT;
   Dys : UINT;
   Sgn : BOOL := 0;
   TimeStructure : TIME;
   Overflow : BOOL;
END_VAR

   CAL TimeConcaten (MilliSeconds := Millisec,
                     Seconds := Sec,
                     Minutes := Min,
                     Hours := Hrs,
                     Days := Dys,
                     Sign := Sgn
                     |
                     TimeStructure := OutputTime,
                     Overflow := Overflow)

END_PROGRAM
```

TimeGenerator
Pulse Generator

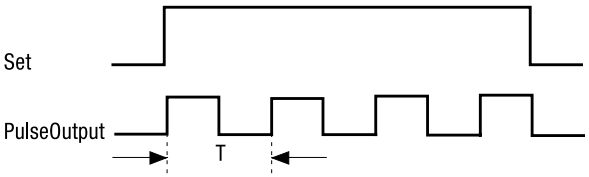


Function block prototype

Meaning of operands

Set	Start condition, rising edge
Period	Period in milliseconds
PulseOutput	Pulse output

Description



Time diagram

A rising edge on the Set input transfers the Period value as period duration T in milliseconds to the timer. As long as the Set input is 1 the PulseOutput will then supply an output pulse sequence for the duration defined on the Period input at a pulse/pause ratio of 1 : 1.

The period duration can be set at the Period input between 1 and 65535 ms.

07/01 AWB2700-1306-GB

If the Period value is changed during pulse generation, the frequency is not adapted immediately. The new value is not transferred until the next rising edge on the Set input.

The accuracy of the period duration output depends on the cycle time of the program and the calculation time of the operating system and deviates by these times.

An uneven value for the period duration is rounded off internally to the next even value. If the period duration is approximately that of the program cycle time or even less than it, the output period duration will be very inaccurate. Calling the function block several times in one program cycle helps to reduce this deviation.

Example “320 milliseconds frequency”

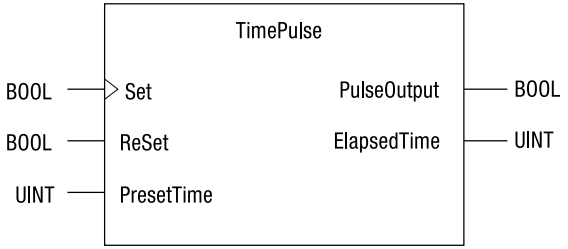
```
PROGRAM frequency

VAR
  ClockGenerator : TimeGenerator;
  Start AT %I0.0.0.0.0 : BOOL;
  PeriodTime: UINT := 320;
  Pulse : BOOL;
END_VAR


  CAL ClockGenerator(Set := Start,
              Period := PeriodTime)
  LD ClockGenerator.PulseOutput
  ST Pulse

END_PROGRAM
```

TimePulse
Time Pulse

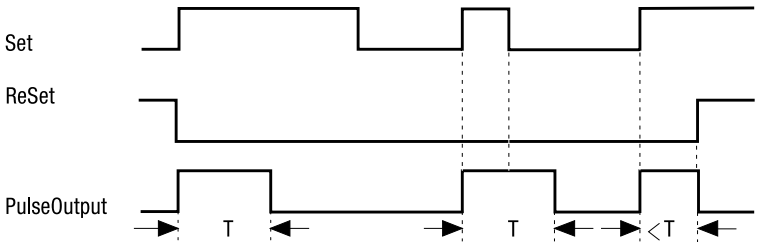


Function block prototype

Meaning of operands

Set	Start condition, rising edge
ReSet	Reset condition
PresetTime	Time setpoint in milliseconds
PulseOutput	Control output
ElapsedTime	Time actual value in milliseconds

Description



Time diagram

A rising edge on the Set input transfers the PresetTime value as a delay time T in milliseconds to the time pulse generator and starts it. The PulseOutput switches to 1 and keeps this status for the duration of the time set.

07/01 AWB2700-1306-GB

A status change on the Set input has no effect during the running of the pulse generator.

The ElapsedTime output shows the current time value in milliseconds.

The timer is reset when the ReSet input is set to 1 during operation.

Example: 125 milliseconds impulse

```
PROGRAM pulse

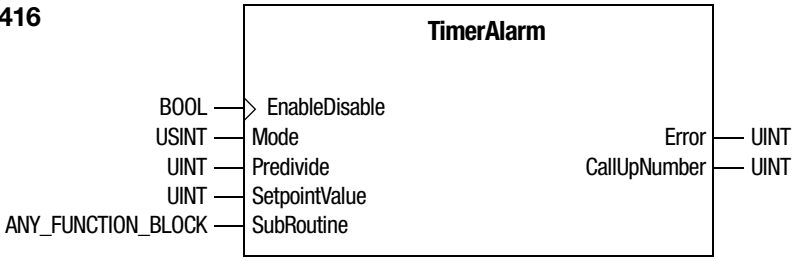
VAR
  Pause : TimePulse;
  Start AT %IO.0.0.0.0 : BOOL;
  PulseDuration: UINT := 125;
  OutputPulse : BOOL;
  ActualTime : UINT;
END_VAR

  CAL Pause(Set := Start,
            PresetTime := PulseDuration)
  LD Pause.PulseOutput
  ST OutputPulse
  LD Pause.ElapsedTime
  ST ActualTime

END_PROGRAM
```

PS4-200
PS4-300
PS416

TimerAlarm
Timer Alarm Function Block



Function block prototype

Meaning of operands

EnableDisable	Enable and disable of alarm execution (0 = Disable, 1 = Enable)
Mode	Operating mode (1 = Timer, 2 = Pulse generator)
Predivide	Predivider for the counter
SetPointValue	Setpoint entry for the timer, the number of pulses (valid value range: 1 to 65535)
SubRoutine	Instance name of the function block called by the alarm
Error	Error code
CallUpNumber	Number of alarm-controlled function block invocations

Description

The EnableDisable input controls the start of the counter. The counter is started if this input is set to 1 or TRUE and the function block instance to be called, Predivide and SetPointValue are accepted and temporarily saved. The value of CallUpNumber is reset.

The function block is stopped and reset if the input status changes from 1 or TRUE to 0 or FALSE.

With Mode = 1 the function block is a timer that calls the specified function block instance after the set

07/01 AWB2700-1306-GB

time has expired. The Predivide input defines the time base in µs. The value of 1000 represents a time base of 1 ms. The smallest selectable time base depends on the PLC used. With the PS4-200/300 it is 512 µs and with the PS416 it is 100 µs.

For example, when using a PS416 with Predivide = 100 a time base of 100 µs is the smallest value that can be set.

SetPointValue defines the setpoint of the timer, i.e. how often the time unit specified at the Predivide input (pulse length) should occur. After the setpoint number of pulses has been reached the counter CallUpNumber is increased and, if entered, a function block instance is called via SubRoutine.

In the operating mode Mode = 2 the function block is a pulse generator which is connected with the hardware output %Q0.0.0.0.0 of the basic unit of the PLC. In this operating mode, the value at the Predivide input defines half the period length in ms, i.e. the pulse length with a fixed pulse/pause ratio of 1:1.

The value at the SetPointValue (= sum of rising/falling edges) input causes half the number of pulses on output %Q0.0.0.0.0.

The pulse sequence is output continuously on the output until the setpoint value defined at the SetPointValue is reached for the first time.

The value at the CallUpNumber output indicates how often the defined setpoint value was reached.



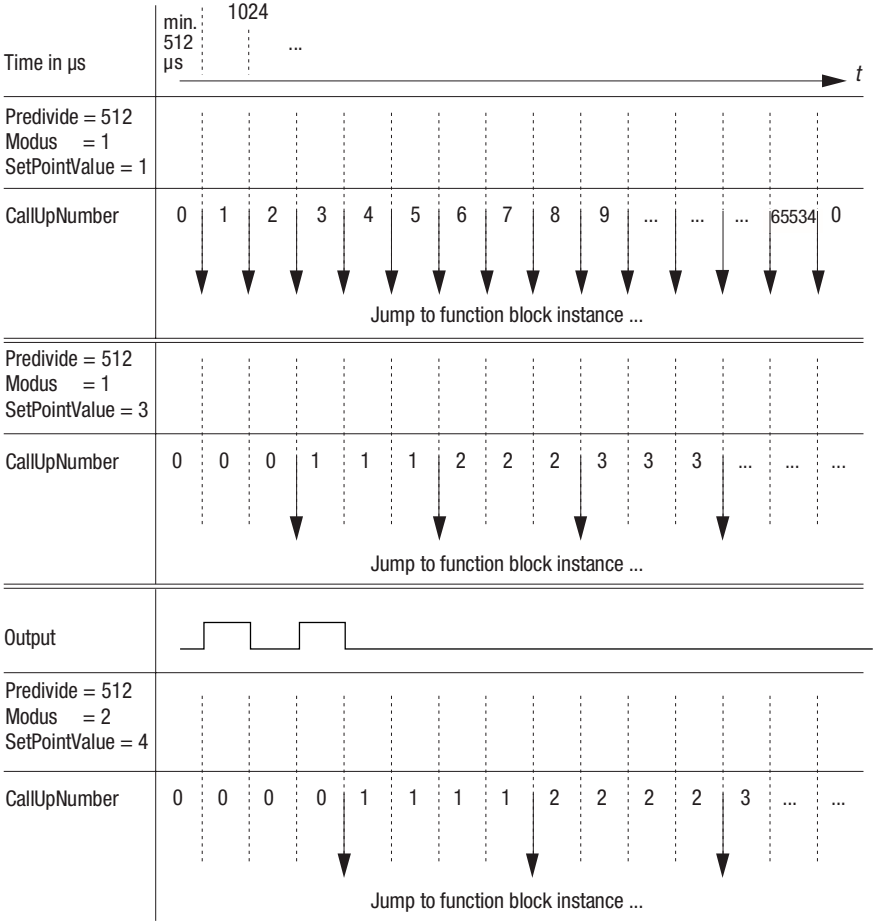
The maximum permissible load on the system depends on three factors: Predivide, SetPointValue and the specified frequency.These factors must be selected so that no more than two interrupts occur per millisecond.

The Error output can indicate one of the following codes:

Signal	PS4-150	PS4-200	PS4-300	PS416
0	No error, activation successful			
1	Setpoint value SetPointValue = 0			
2	Setpoint value Predivide is ...			
	< 512	< 512	< 512	< 100
3	–	–	Mode has invalid value	
4	–	–	Timer alarm already in use.	
5	–	–	–	Combination of SetPointValue Predivide overloads the system

The SubRoutine input allows event-controlled programming to be implemented. This is done by stating the instance name of the function block to be activated when the event is reached. If no instance is specified, the CountValue will continue to be incremented without triggering the alarm. The function block instance called by the alarm function block must not have any VAR_IN_OUT parameters. This kind of instance must only have local variables (VAR) or global variables that have been declared externally as VAR_EXTERNAL variables.

07/01 AWB2700-1306-GB



TimerAlarm function description using the example of a PS4-200/-300

Observe the following points for event-driven program processing:

After the event has occurred, the user program is interrupted, the register status is saved and the subprogram specified by Subroutine is executed.

The alarm function block cannot be interrupted by other alarm function blocks (all alarm function blocks have the same priority). The maximum execution time of alarm blocks is to be restricted to 5 ms (approx. 1K IL instructions) in the user program since the alarm function blocks cannot be interrupted even by the operating system in the event of a voltage drop. If the execution time is exceeded, an EDC error may occur when switching off the power supply.

The execution time of the alarm function block is added to the execution time of the cyclical user program and also monitored by the cycle time monitoring function.

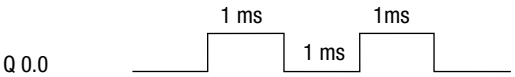
Since the event-driven program processing enables access to the entire image register, access to data that is used by the event-driven and cyclical user program must be disabled. Bit accesses may only occur on the same byte in the cyclical user program and in the Alarm function block if the access is synchronized with Disable_Interrupt and Enable_Interrupt.

Since, due to its fast reaction times, an Alarm function block requires a high-speed peripheral access (direct output), the QP, QPB peripheral operands available in the basic unit should be used.

An alarm function block can be used several times (multiple instantiation) although this should normally be avoided since each function block group has the same event source and only the last function block instance in the program is valid. Multiple instantiation is only useful if the parameters of the function block need to be reset again within an interrupt routine and an edge disable is required.

By multiple instantiation is meant the reservation of several data ranges for each parameter set of a particular function block type.

The following example shows the invocation of the Timer alarm function block operating as a pulse generator (Mode = 2). The time base, i.e. pulse length is 1 ms (Predivide = 1000). The output at %Q0.0.0.0.0 is as follows (a total of 50 pulses are output):





The input parameters of an alarm function block should only be assigned between the parentheses when it is called. An explicit parameter assignment outside of the parentheses before or after the invocation is not permissible.

Example: TimerAlarm as pulse generator

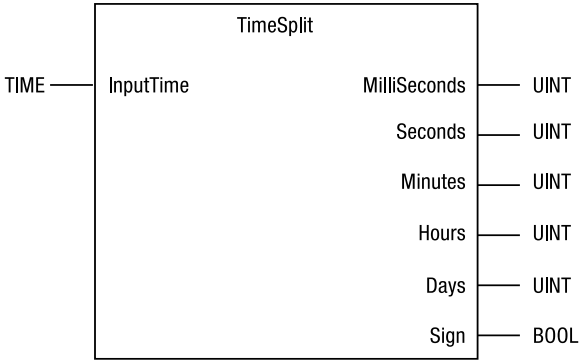
```
PROGRAM talarm

VAR
  T_Alarm : TimerAlarm;
  Error : USINT;
  CountValue : UINT;
END_VAR

  CAL T_Alarm (EnableDisable := 1,
               Mode := 2,
               Predivide := 1000,
               SetPointValue := 100)
  LD T_Alarm.Error
  ST Error
  LD T_Alarm.CallUpNumber
  ST CountValue (*This gives the number
               of complete pulse series
               that have been output*)
END_PROGRAM
```


07/01 AWB2700-1306-GB

TimeSplit
Split TIME Variable Type



Function block prototype

Meaning of operands

InputTime	Time duration to be divided in time elements
MilliSeconds	Milliseconds part of time duration
Seconds	Seconds part of time duration
Minutes	Minutes part of time duration
Hours	Hour part of time duration
Days	Day part of time duration
Sign	Sign of time duration

Description

The time duration value specified via the InputTime input operand is divided into time elements that are each output as UINT data types.

The sign of the time duration specified is indicated by the Boolean output operand Sign. 0 indicates a positive value, 1 indicates a negative value.

Example: Split time duration into time elements

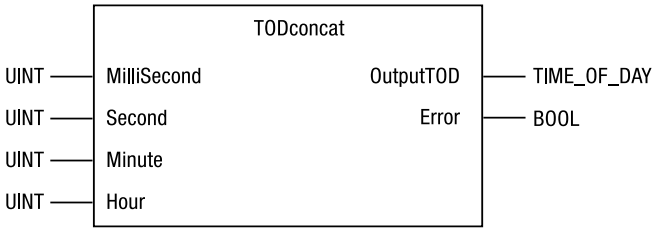
```
PROGRAM ti_split
```

```
VAR
  TimeElements : TimeSplit;
  Duration : TIME := T#20d22h13m12s800ms;
  Millisec : UINT;
  Sec : UINT;
  Min : UINT;
  Hrs : UINT;
  Dys : UINT;
  Sgn : BOOL;
END_VAR

  CAL TimeElements(InputTime := Duration
                |
              Milliseconds := MilliSec,
              Seconds := Sec,
              Minutes := Min,
              Hours := Hrs,
              Days := Dys,
              Sign := Sgn)END_PROGRAM
```


07/01 AWB2700-1306-GB

TODconcat
Generate Time



Function block prototype

Meaning of operands

MilliSecond	Milliseconds part of time
Second	Seconds part of time
Minute	Minutes part of time
Hour	Hours part of time
OutputTOD	Resulting value for time
Error	Error message

Description

The individual time elements specified as input operands are combined to a time value in the output operand OutputTOD. The input operands must not exceed the permissible value ranges for a time.

The output operand Error will be set to 1 if the permissible range of a time value is exceeded.

Example: Generate time of day

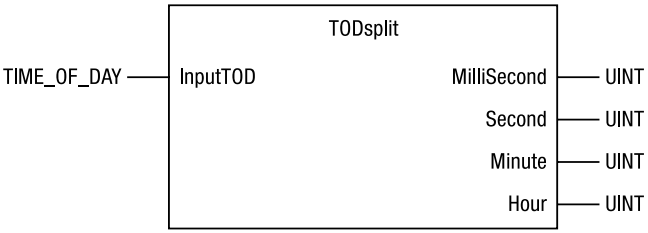
```
PROGRAM time2

VAR
  TimeOfDay : TODconcat;
  Millisecond : UINT;
  Sec : UINT;
  Min : UINT;
  Hr : UINT;
  TODStructure : TIME_OF_DAY;
  Error : BOOL;
END_VAR

  CAL TimeOfDay(MilliSecond := Millisecond,
              Second := Sec,
              Minute := Min,
              Hour := Hr
              |
              TODStructure:= OutputTOD,
              Error:= Error)END_PROGRAM
```


07/01 AWB2700-1306-GB

TODsplit
Split TIME_OF_DAY Data Type



Function block prototype

Meaning of operands

InputTOD	Time of day
MilliSecond	Milliseconds part of time
Second	Seconds part of time
Minute	Minutes part of time
Hour	Hours part of time

Description

The time value specified via the InputTOD input operand is divided into time elements that are output as UINT data types.

Example: Split time of day into time elements

```
PROGRAM time1

VAR
   Split_time : TODsplit;
   TimeOfDay : TOD := TOD#12:13:12;
   Millisec : UINT;
   Sec : UINT;
   Min : UINT;
   Hrs : UINT;
END_VAR

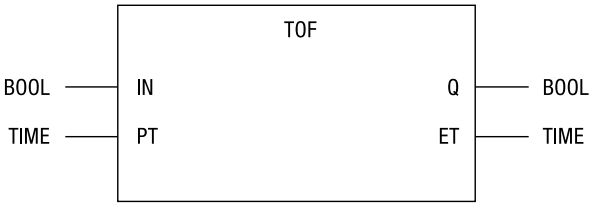
   CAL Split_time(InputTOD := TimeOfDay
                              |
                             Milliseconds := Millisec,
                             Seconds := Sec,
                             Minutes := Min,
                             Hours := Hrs)

END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC

TOF
Off delay

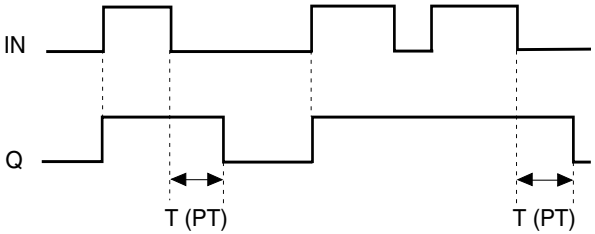


Function block prototype

Meaning of operands

IN	Start condition
PT	Predefined time value
Q	Binary status of timer
ET	Current time value

Description



Time diagram

The status 1 of the input operand IN is transferred to the output operand Q without delay. The falling edge of IN starts the time function of the timer TON for the duration of the time value defined by the operand PT.

The change of the status on the input IN to 0 during the running time does not effect the operation. After the time has elapsed, the status of the operand Q changes to 0. If the PT value changes after the start,

it only has effect with the next rising edge of the operand IN.

The output operand ET indicates the current time value. After the time has elapsed, the operand ET keeps its value as long as the operand IN has the status 1. If the status of the operand IN changes to 0, the value of ET changes to 0.

Deactivating the input IN causes the output Q to be switched off, delayed by the predefined time.

Example: Switch off delayed by 125 milliseconds

```
PROGRAM time2

VAR
  Timer3 : TOF;
  Start AT %I0.0.0.0.0 : BOOL;
  Duration : TIME := T#125ms;
  Output AT %Q0.0.0.0.0 : BOOL;
  Actual_time : TIME;
END_VAR

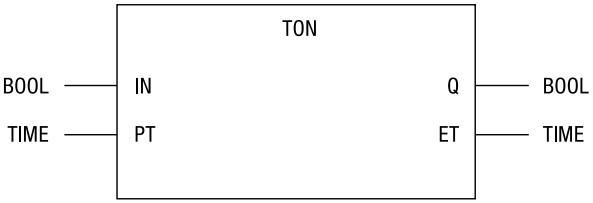
  CAL Timer3(IN := Start,
             PT := Duration)
  LD Timer3.Q
  ST Output
  LD Timer3.ET
  ST Actual_time

END_PROGRAM
```


07/01 AWB2700-1306-GB

IEC

TON
On delay

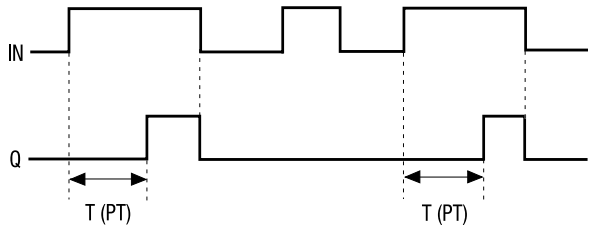


Function block prototype

Meaning of operands

IN	Start condition
PT	Predefined time value
Q	Binary status of timer
ET	Current time value

Description



Time diagram

The rising edge of the input operand IN starts the time function of the timer TON for the duration of the time value defined by the operand PT.

During this time the output operand Q has the status 0. After the time has elapsed, the status changes to 1 and is kept until the operand IN changes to 0.

If the PT value changes after the start, it only has effect with the next rising edge of the operand IN.

The output operand ET indicates the current time value. After the time has elapsed, the operand ET keeps its value as long as the operand IN has the status 1. If the status of the operand IN changes to 0, the value of ET changes to 0.

If the status of the operand IN changes to 0 during the running time, the operation is stopped and the operand ET is again 0.

The activation of the input IN causes the output Q to be switched on, delayed by the predefined time.

Example: Switch on delayed by 12 milliseconds

```
PROGRAM time1

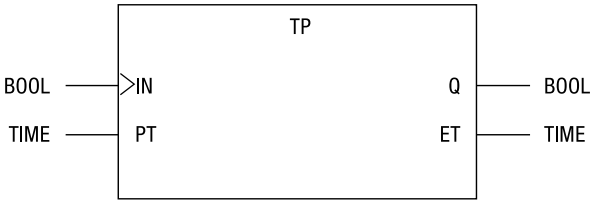
VAR
  Timer2 : TON;
  Start AT %I0.0.0.0.0 : BOOL;
  Duration : TIME := T#12ms;
  Output AT %Q0.0.0.0.0 : BOOL;
  Actual_time : TIME;
END_VAR

  CAL Timer2(IN := Start,
             PT := Duration)
  LD Timer2.Q
  ST Output
  LD Timer2.ET
  ST Actual_time

END_PROGRAM
```

IEC

TP
Impulse

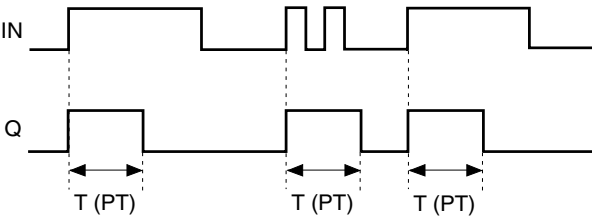


Function block prototype

Meaning of operands

IN	Start condition
PT	Predefined time value
Q	Binary status of timer
ET	Current time value

Description



Time diagram

The rising edge of the input operand IN starts the time function of the timer TP for the duration of the time value defined by the operand PT.

During this time the output operand Q has the status 1. The change of the status on the input IN does not effect the operation.

If the PT value changes after the start, it only has effect with the next rising edge of the operand IN.

The output operand ET indicates the current time value. If the operand IN is 1 after the time has elapsed, the operand ET keeps its value. The value at the ET output is retained until the function block is restarted.

Each edge occurring when the timer is not active initiates a pulse for the set duration.

Example: 125 milliseconds impulse

```
PROGRAM pulse

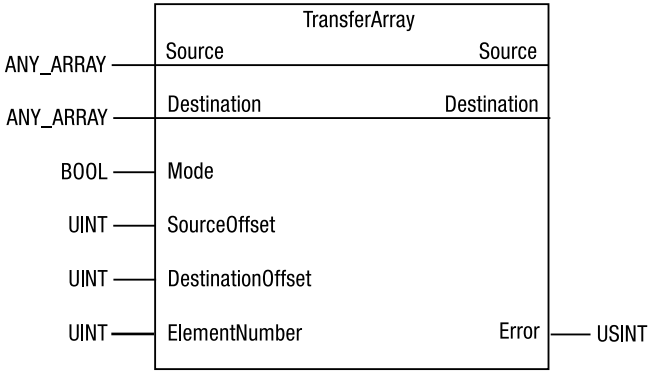
VAR
  Timer1 : TP;
  Start AT %I0.0.0.0.0 : BOOL;
  Pulse_duration: TIME := T#125ms;
  Output_pulse : BOOL;
  Actual_time : TIME;
END_VAR

  CAL Timer1(IN := Start,
             PT := Pulse_duration)
  LD Timer1.Q
  ST Output_pulse
  LD Timer1.ET
  ST Actual_time

END_PROGRAM
```


07/01 AWB2700-1306-GB

TransferArray
Copy or Initialise Array



Function block prototype

Meaning of operands

Source	Source area
Destination	Destination area
Mode	Operating mode (0 = Initialise, 1 = Copy)
SourceOffset	Offset in Source area
DestinationOffset	Offset in Destination area
ElementNumber	Number of bytes to be copied
Error	Error code

Description

The TransferArray function block has two operating modes: Initialise and Copy.

The mode is set with the Mode input.

The source and the destination can be arrays of any size. All ANY_BIT and ANY_INT arrays apart from BOOL are permissible as array elements.

The Error output indicates the following error codes:
0 No error
1 Number of elements (ElementNumber) is 0
2 Source offset is outside the source area.
3 Number of elements exceeds the source area.
4 Destination offset is outside the destination area
5 Number of elements exceeds the destination area
6 Source and destination addresses are identical

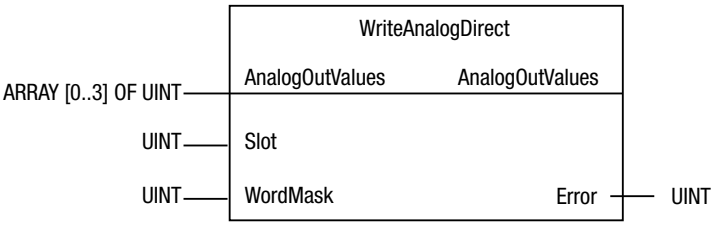
Initialise mode
In Initialise mode the function block copies the contents of the array element specified by “Source + SourceOffset” to a data block specified by “Destination + DestinationOffset”. Source and destination can lie within the same array.

Copy mode
In Copy mode the function block copies part of an array or an entire array from “Source + SourceOffset” to “Destination + DestinationOffset”. The number of elements to copy is specified with ElementNumber. Source and destination can lie within the same array to allow block moves.

Please see the description of the BlockTransfer function block for examples of using TransferArray, since these function blocks are very similar. You only need to change the declarations, i.e. replace source and destination address by array declarations.

07/01 AWB2700-1306-GB

WriteAnalogDirect
Direct Writing to Analog Outputs



Function block prototype

Meaning of operands

Slot	Slot of a PS416-AIO card in the basic unit (4..20)
WordMask	Bit mask specifying which channels are to be written
AnalogOutValues	Bit mask specifying which channels are to be written
Error	Error code

Description

The WriteAnalogDirect function block allows direct write access to analog output channels in the basic unit of the PS416.

The data is not written to the outputs at the end of a cycle but directly during the current cycle. This function block therefore allows fast and direct response to events such as required in alarm routines. The maximum on/off delay of the analog cards for the controller should be taken into account when using this function block.

The Slot input specifies the slot in which the card to be accessed is fitted.

The WordMask input specifies which channels are to be write accessed on the card. The bit position determines the channel:

Bit 0 – channel 0

Bit 1 – channel 1

Bit 2 – ...

This therefore enables you to specify channels to be write accessed as required.

The AnalogOutValues input is used to transfer the write values. Only the values for the channels specified in Wordmask are output.

The Error output indicates the following error codes:
0 No error
1 Invalid slot
2 No PS416-AIO-400 present on the slot
3 Incorrect channel number requested via mask

07/01 AWB2700-1306-GB

Example “Write analog data direct”

```
PROGRAM WriteAnalogVAR
   WriteAnalog : WriteAnalogDirect;
   AnalogData : ARRAY[0..3] OF UINT;
   WriteError : UINT;
END_VAR

   LD 1234
   ST AnalogData[0]                    (* Data channel no.0 *)
   LD 567
   ST Analogdata[1]                    (* Data channel no.1 *)
   CAL WriteAnalog(Slot := 5,          (* Card fitted in slot 5 *)
                   Wordmask := 16#3, (* Write channel no. 0 and no.1 *)
                   AnalogOutValues := AnalogData
                   |
                   WriteError = Error)
   LD WriteError
   EQ 0
   JMPC WRITE_DONE

    (* Error handling *)

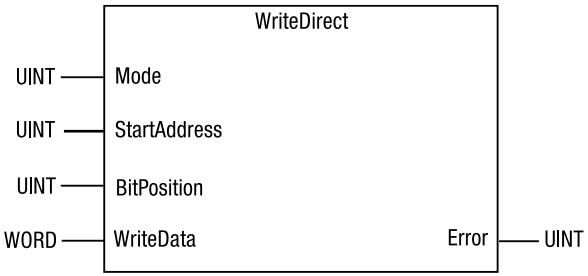
WRITE_DONE:

    (* Further sections of the program *)

END_PROGRAM
```

PS416

WriteDirect
Direct Writing to Digital Outputs



Function block prototype

Meaning of operands

Mode	Access mode
StartAddress	Byte or word address of the digital input; Permissible range of values: Bit and byte access: 0 to 127, Word access: 0 to 126
BitPosition	Bit position with bit access Vaild range: 0 to 7
WriteData	Data value to be written
Error	Error message

Description

The WriteDirect function block enables direct write operations to digital output cards in the PS416 basic unit. The data is not written to the outputs at the start of a cycle but directly during the current cycle. This function block can thus be used for fast and direct reactions to events, e.g. in alarm routines. When using this function block, observe the maximum on/off delay of the digital I/O cards for the PS 416 system.

07/01 AWB2700-1306-GB

The Mode input defines the access type.
1 Write a single bit to an output card.
2 Write a byte (8 bit) to an output card.
3 Write a word (16 bit) to an output card.

The StartAddress and BitPosition inputs address an output and are dependent on Mode. BitPosition is only relevant for Mode = 1. Examples:

Mode = 1: StartAddress := 4, Position := 6 ==> %Q0.0.0.4.6

Mode = 2 StartAddress := 2 ==> %QB0.0.0.2

Mode = 3 StartAddress := 126 ==> %QW0.0.0.126.

The WriteData input must contain the data value to write to the output card. The WORD data type allows individual bits of WriteData to be accessed. Depending on the Mode the following bits are relevant:

Mode = 1 WriteData.0

Mode = 2 WriteData.0 - WriteData.7

Mode = 3 WriteData.0 - WriteData.15.



Outputs written by the WriteDirect function block must be declared in the declaration section of the user program to allow them to be reset by the PLC operating system when the user program is stopped.



Warning
Undeclared outputs written by the WriteDirect function block can result in hazardous states of the machine, since they are not automatically reset in the READY state of the PLC.
Ensure that the outputs that are written with “WriteDirect” are declared in the user program and that the declaration and the values output with “WriteDirect” are identical.

The Error output indicates the following error codes:

0 No error, WriteData valid
1 Value of Mode input invalid
2 Value of StartAddress input is too large.
3 Value of StartAddress input not an even number (only Mode = 3)
4 Value of BitPosition input invalid

07/01 AWB2700-1306-GB

```
PROGRAM WriteIO

VAR
   WriteEnable AT%IO.0.0.0.1:BOOL;
   WriteIODirect                          :WriteDirect;
   WriteData                              :WORD;
   WriteError                             :UINT;

   (* The following declaration ensures that output byte 32 is reset
    to a defined state by the operating system of the PS416 if the user program
    is stopped. *)
   IOByte32 AT %QB0.0.0.32 :USINT;
END_VAR

   (*Further parts of the program *)
   LD IOByte32
   ADD 1
   ST IOByte32
   ...

   LD IOByte32
   USINT_TO_WORD
   ST WriteData

   (* Direct read access to the digital I/O.
   Data is written to I/O byte 32. This corresponds to the variable declaration
%IB0.0.0.32 : USINT;*)
   LD WriteEnable
   CALC WriteIODirect(Mode := 2,(* Byte access *)
                      Address := 32,       (* Byte address 32*)
                      BitPosition := ,     (* Unused *)
                      WriteData := WriteData
                      |
                      WriteError := Error )

   LD WriteError
   EQ 0
   JMPCN WRITE_DONE

WRITE_ERROR:

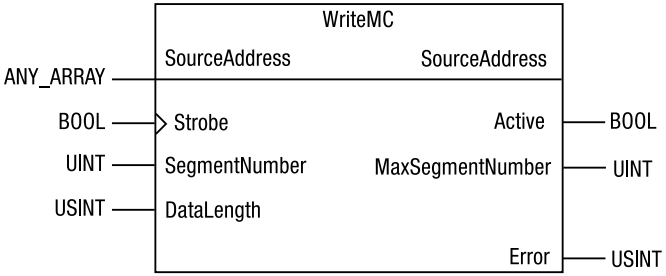
    (* Error handling *)

WRITE_DONE:

    (* Further sections of the program *)

END_PROGRAM
```

WriteMC
Save Data on Memory Card



Function block prototype

Meaning of operands

SourceAddress	Data buffer with the data to be saved
Strobe	Start save operation
SegmentNumber	Segment number (0 - n)
DataLength	Number of data bytes to be saved
Active	Save operation running
MaxSegmentNumber	Last possible segment number of the file
Error	Error message

Description

The WriteMC function block allows data to be saved to the PLCs memory card (MC). Together with the ReadMCFB_ReadMC function block, important applications of this function block are data backup, process data acquisition, saving recipe data and/or saving variables before a program update or the handing of recipe data.

You can store recipe data on the memory card of the PS4-300 by selecting the appropriate option when formatting the memory card.

07/01 AWB2700-1306-GB

The memory card is addressed by segment number. The function block can write up to 128 bytes to the memory card.

To determine the MC file size, call the function block with DataLength set to 0. This does not write to the memory card. MaxSegmentNumber then indicates the highest segment number in the MC file.

The function block is executed asynchronously of the user program cycle and can take several program cycles. The Active output is set during this time. Error-free completion of the function block is indicated by the Active output changing to 0 and by 0 at the Error output.

You can activate several function block instances simultaneously. They will be processed one after the other in the order that they were called.

Error codes at the Error output:

0 Function block execution completed without errors
1 No memory card inserted
4 SegmentNumber exceeds the value of MaxSegmentNumber
5 DataLength is larger than 128 bytes or SourceAddress is the wrong size

Some messages only apply to particular PLCs. Error codes with PS4-200 only:

2 No access possible due to Online connection
3 No access possible due to simultaneous access via “ReadMC“
7 No access possible, MC being used as source directory.

Error code with PS 4-300 only

8 The MC was formatted without recipe data range

Error codes with PS416 only:

10 Memory card type is not SRAM.
11 Memory card is not formatted.
12 Memory card is write-protected.
13 Insufficient free memory on memory card to create the file RSDAT.DAT.
14 File RSDAT.DAT does not have the required minimum size of 128 bytes.
15 Internal error during execution of the function block.

The following information on the WriteMC function block only applies to the PS416 controller:

The data is saved on the memory card in the MS-DOS file RSDAT.DAT and can be read and processed with a PCMCIA drive. The file is subdivided into 128 byte segments and its size is only limited by the capacity of the memory card used.

07/01 AWB2700-1306-GB

If a memory card is inserted and the file RSDAT.DAT is not present on the memory card when the function block is first called, it is created automatically with a default size of 512 x 128 byte (64 KByte) and all sectors are initialised with the value “0xFF”. Other file sizes can be created with the SetMCFileLength.



The ACTIVE output of these function blocks is always 0 when used in PS 4-200 and PS4-150 controllers. The current function block status can be scanned via the Error output when the function blocks are called again. If the Error output is 0, processing was carried out error-free. If the output is 3, the function block is processing. To refresh the Error output of this function block a new edge on the Strobe input must be generated. Otherwise the error message will be kept (with 1 on Strobe) or will be set to 0 (with 0 on Strobe).

Invocation of the function block WriteMC

```
PROGRAM SAVE

VAR
   StrobeSave AT %I0.0.0.0.0 : BOOL;
   ActiveSave AT %Q0.0.0.0.0 : BOOL;
   ErrorSave : USINT;
   FBSaveData : WriteMC;
   SegmentNumber : UINT;
   DataLength : USINT;
   LastSegmentNumber : UINT;
   DataArea : ARRAY[0..31] of USINT;
END_VAR

   LD ActiveSave
   JMPCN CONTINUE

    (* Prepare data to save *)

CONTINUE:


   LD 47
   ST SegmentNumber
   LD 32
   ST DataLength

    (* Write from array DataArea a total of 32 bytes to memory card
    segment 47 *)
   CAL FBSaveData(Strobe := StrobeSave,
                  SegmentNumber := SegmentNumber,
                  DataLength := DataLength,
                  SourceAddress := DataArea
                  |
                  ActiveSave := Active,
                  LastSegmentNumber := MaxSegmentNumber,
                  ErrorSave := Error)

   LD ErrorSave
   EQ 0
   JMPC NO_ERROR

    (* Error handling *)

NO_ERROR:

   (* Further parts of user program *)

END_PROGRAM
```


07/01 AWB2700-1306-GB

7 Language Extension

Mixing IL and ST language elements

Self-contained IL sequences or ST instructions can be placed together in succession in the instruction section of a POU.

An ST instruction cannot be used inside an IL sequence and an IL sequence cannot be placed inside an ST instruction.

Single-bit access

In the program you can access an individual bit located in a data element declared as BYTE, WORD or DWORD by specifying the corresponding bit number, separated by a dot. The bit number, beginning with “0” for the least significant bit, is specified as follows:

0 ... 7 in the byte

0 ... 15 in the word

0 ... 31 in the double word

Example

```
PROGRAM only 1 bit

VAR
  BitPattern1 : BYTE := 2#10101010;
  BitPattern2 AT %IW0.0.0.0 : WORD;
END_VAR

    (* Copying of bit 15*)
  LD BitPattern2.15(* from BitPattern2 *)
  ST BitPattern1.0(* to Bit 0 of BitPattern1 *)
  .
  .

END_PROGRAM
```

FB invocation with direct parameter passing

Two ways of invoking a function block are described in Chapter 3 from Page 71. The two variants are defined in accordance with the IEC standard.

In addition Moeller offers a further variant whereby upon invocation of the function block the parameters are passed directly to all operands used in the function block concerned. The inputs are configured together after the CAL command and then, separated by the | character, the outputs.

Example

```
CAL SR_Instance_1(Set1 := ON,
  ReSet := OFF
  |
  Result := Q1)
```

Direct peripheral I/O access for PS4-200 and PS4-300

If the value once updated at the start or end of the cycle is not to be used for a directly declared input or output but instead the value at the time of processing in the user program is used, the following should be taken into account when declaring these variables (see also example on Page 589):

1. Direct access is only possible for input byte %IB0.0.0.0 or output byte %QB0.0.0.0 on a bit and/or byte basis.
   This restriction only applies to PS4-200!
   As described in point 3, words and double words can also be addressed with the PS4-300.
   The entire %I, %Q, %IA, %QA, %IC image of the PLC can be explicitly addressed.

07/01 AWB2700-1306-GB

2. The memory location information for such a variable includes the additional letter “P”, i.e. for an input byte %IPB0.0.0.0 and for an output byte %QPB0.0.0.0.
3. The following data types are permissible:
   BOOL_EXP, BYTE_EXP, SINT_EXP and USINT_EXP
   The following data types are only possible for the PS4-300:
   WORD_EXP, DWORD_EXP, INT_EXP, UINT_EXP, DINT_EXP and UDINT_EXP.

The image register of the PS4-200 is refreshed at the end of a cycle. The image register of the PS4-300 is refreshed immediately.

Example

```
VAR
  Input AT%IPO.0.0.0.0: BOOL_EXP;
  Output AT%QPBO.0.0.0: USINT_EXP;
END_VAR
```

This access method is useful in alarm routines where as a rule a rapid reaction is required (see also “CounterAlarm”, Page 271; “EdgeAlarm”, Page 326; “TimerAlarm”, Page 554).

With the PS416 it is possible to program direct I/O access with the function blocks “ReadDirect” and “WriteDirect”.

07/01 AWB2700-1306-GB

8 Compliance with the IEC Standard

The following tables have the same numbering as those in the
IEC 61131-3, 2nd edition. Tables showing features not yet supported by version 5.x of Sucosoft S40 are not listed in this manual.

The “PLC” column indicates the PLCs that support the features concerned:

x	All PLCs
PS416	PS416-CPU-200/-300/-400 modular PLCs
PS4-200	PS4-201-MM1, PS4-141-MM1, PS4-151-MM1, PS4-271-MM1 compact PLCs
PS4-300	PS4-341-MM1 compact PLC

Common Elements

Sucosoft S40 V5.x fulfills the requirements of the IEC/EN 61131-3 in the following language features:

Table 1: Character set features

No.	Description	PLC	Comment
1	Required character set	x	according to ISO 646 IRV, Table 1 (ASCII)
2	Lower case	x	
3a	Number sign (#)	x	
3b	Pound sign (£)	–	
4a	Dollar sign ($)	x	
4b	Currency sign	–	
5a	Vertical bar (|)	–	Not applicable, since connecting lines are not shown in text form
5b	Exclamation mark (!)	–	

Table 2: Identifier features

No.	Description	PLC
1	Upper case and numbers	x
2	Upper and lower case, numbers, embedded underlines	x
3	Upper and lower case, numbers, leading or embedded underlines	x

07/01 AWB2700-1306-GB

Table 3: Comment features

No.	Description	PLC
1	Block comment (* *)	x

Table 4: Numeric literals

No.	Description	PLC
1	Integer literals	x
2	Real literals	x
3	Real literals with exponents	x
4	Base 2 literals	x
5	Base 8 literals	x
6	Base 16 literals	x
7	0 and 1 instead of FALSE and TRUE	x
8	FALSE and TRUE	x
9	Typed literals	–

Table 5: Character string literal features

No.	Description		PLC
1	ASCII strings		x
	''	Empty string (length zero)	x
	'A'	String of length 1 containing single character A	x
	' '	String of length 1 containing “space” character	x
	'$''	String of length 1 containing single “quote” character	x
	'""'	String of length one, with double quote characters	x
	'RL'	String of length 2 containing CR and LF	x
	'$0A'	String of length 1 containing LF	x
	'$$1.00'	String of length 5 which is printed as “$1.00”	x
	'$$1.00'	String of length 2 which is printed as “$1.00”	x
	'ÄË' '$C4$CB'	String of length 2 containing ANSI special characters	x
2		WSTRING string	–
3		Typed ASCII string	–
4		Typed WSTRING string	–

07/01 AWB2700-1306-GB

Table 6: Two character combinations in character string

No.	Description	PLC
2	Dollar sign ($$)	x
3	Single quote character ($') in ASCII strings	x
4	Line feed ($L or $l)	x
5	New line ($N or $n)	x
6	New page ($P or $p)	x
7	Carriage return ($R or $r)	x
8	Tab ($T or $t)	x
9	Double quote character ($") in WSTRING character strings	–

Table 7: Duration and literal features

No.	Description	PLC
	Literals without underlines:	
1a	– short prefix (T#)	x
1b	– long prefix (TIME#)	x
	Literals with underlines:	
2a	– short prefix (T#)	x
2b	– long prefix (TIME#)	x

Table 8: Date and time of day literals

No.	Description	PLC
1	Date literals (long prefix: DATE#)	x
2	Date literals (short prefix: D#)	x
3	Time of day literals (long prefix: TIME_OF_DAY#)	x
4	Time of day literals (short prefix: TOD#)	x
5	Literals for Date and time (long prefix: DATE_AND_TIME#)	x
6	Literals for date and time (short prefix: DT#)	x

Table 10: Elementary data types

No.	PLC	Description	PLC
1	BOOL	Boolean	x
2	SINT	Signed integer (8 bit)	x
3	INT	Signed integer (16 bit)	x
4	DINT	Signed integer (32 bit)	PS416, PS4-300
5	LINT	Signed integer (64 bit)	–
6	USINT	Unsigned integer (8 bit)	x
7	UINT	Unsigned integer (16 bit)	x
8	UDINT	Unsigned integer (32 bit)	PS416, PS4-300
9	ULINT	Unsigned integer (64 bit)	–
10	REAL	Real number (32 bit)	PS416, PS4-300
11	LREAL	Real number (64 bit)	–
12	TIME	Duration	x
13	DATE	Date	x
14	TOD or TIME_OF_DAY	Time of day	x
15	DT or DATE_ AND_TIME	Date and time	x
16	STRING	Variable length ASCII character string	x
17	BYTE	Bit string of length 8	x
18	WORD	Bit string of length 16	x
19	DWORD	Bit string of length 32	PS416, PS4-300
20	LWORD	Bit string of length 64	–
21	WSTRING	Variable length Unicode character string	–

07/01 AWB2700-1306-GB

Table 12: Data type declaration features

No.	Description	PLC
1	Alias MY_REAL:REAL;	x
2	Enumerated data types SIGNAL_TYPE:(SINGLE,DIFFERENT);	x
3	Subrange data types ANALOG_DATA:INT(–4095..4096);	x
4	Array data types MY_ARRAY:ARRAY[1..16] OF INT;	x
5	Structures CHANNEL:STRUCT RANGE:SIGNAL_TYPE; MIN_SCALE:ANALOG_DATA; MAX_SCALE:ANALOG_DATA; END_STRUCT;	x

Table 13: Initial values

Description	Initial value
SINT, INT, DINT	0
USINT, UINT, UDINT	0
BOOL, BYTE, WORD, DWORD	0
REAL	0.0
TIME	T#0s
DATE	D#1900-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#1900-01-01-00:00:00
STRING	'' (empty string)

Table 14: Data type initial value declaration features

No.	Description	PLC
1	Initialization of Alias types	x
2	Initialization of enumerated data types	x
3	Initialization of subrange data types	x
4	Initialization of array data types	x
5	Initialization of structured data types	x
6	Initialization of derived structured data types	x

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

No.	Description	PLC
1	I: Input location	x
2	Q: Output location	x
3	M: Marker location	x
4	X: Single bit size	x
5	None: Single bit size	x
6	B: Byte size (8 bits)	x
7	W: Word size (16 bits)	x
8	D: Double word size (32 bits)	PS416, PS4-300
9	L: Quadruple word size (64 bits)	–
10	*: Placeholder for incompletely specified memory location	–

07/01 AWB2700-1306-GB

Table 16a: Keywords for variable declaration

Keyword	PLC
VAR	x
VAR_INPUT	x
VAR_OUTPUT	x
VAR_IN_OUT	x
VAR_EXTERNAL	x
VAR_GLOBAL	x
VAR_ACCESS	–
VAR_TEMP	–
VAR_CONFIG	–
RETAIN	x
NON_RETAIN	–
CONSTANT	x
AT	x

Table 17: Supported variable declarations

No.	Description	PLC
1	Declaration of directly represented, non-retentive variables	x
2	Declaration of directly represented, retentive variables	x
3	Declaration of the memory location for symbolic variables	x
4	Declaration of the memory location for symbolic array variables	x
5	Automatic memory allocation for symbolic variables	x
6	Array declaration	x
7	Declaration of retentive arrays	x
8	Declaration of structured variables	x

Table 18: Explicit initialization of variables

No.	Description	PLC
1	Initialization of directly represented, non-retentive variables	x
2	Initialization of directly represented, retentive variables	x
3	Initialization of symbolic variables and declaration of the memory location	x
4	Initialization of symbolic array variables and declaration of the memory location	x
5	Initialization of symbolic variables	x
6	Initialization of array data types	x
7	Retentive array declaration and initialization	x
8	Initialization of structured variables	x
9	Initialization of constants	x
10	Initialization of function block parameters on instantiation	–

Table 19: Graphical negation of Boolean signals

No.	Description	PLC
1	Negated input	x
2	Negated output	x

Table 19a: Call of functions in IL and ST

No.	Description	PLC
1	Specification of parameter names, any order of parameters, any number of parameters listed	–
2	No specification of parameter names, any order of parameters as in function prototype, fixed number of parameters listed	x

07/01 AWB2700-1306-GB

Table 20: The use of the “EN” input and “ENO” output

No.	Description	PLC
1	Use of “EN” and “ENO” with LD	–
2	FBD without “EN” and “ENO”	x
3	Use of “EN” without “ENO” with FBD	–
4	Use of “EN” without “ENO” with FBD	–

Table 21: Typed and overloaded functions

No.	Description	PLC
1	Overloaded, non type-dependent functions	x
2	Typed functions	–

Table 22: Type conversion function features

No.	Description	PLC
1	*_TO_**	x
2	TRUNC	PS416, PS4-300
3a	BCD_TO_**	x
3b	*_BCD_TO_**	–
4a	**_TO_BCD	x
4b	**_TO_BCD_*	-

Table 23: Standard functions with one numeric input

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

Table 24: Standard arithmetic functions

No.	Name	Symbol	PLC
12	ADD	+	x
13	MUL	*	x
14	SUB	–	x
15	DIV	/	x
16	MOD	MOD	PS416, PS4-300
17	EXPT	**	-
18	MOVE	:=	-

07/01 AWB2700-1306-GB

Table 25: Standard bit-shift functions

No.	Name	PLC
1	SHL	x
2	SHR	x
3	ROR	x
4	ROL	x

Table 26: Logical standard functions

No.	Name	Symbol	PLC
5	AND	&	x
6	OR		x
7	XOR		x
8	NOT		x

Table 27: Standard selection functions

No.	Name	PLC
1	SEL	PS416, PS4-300
2a	MAX	PS416, PS4-300
2b	MIN	PS416, PS4-300
3	LIMIT	PS416, PS4-300
4	MUX	PS416, PS4-300

Table 28: Standard comparison functions

No.	Name	Symbol	PLC
5	GT	>	x
6	GE	>=	x
7	EQ	=	x
8	LE	<=	x
9	LT	<	x
10	NE	>	x

Table 29: Standard character string functions

No.	Name	PLC
1	LEN	x
2	LEFT	PS416, PS4-300
3	RIGHT	PS416, PS4-300
4	MID	PS416, PS4-300
5	CONCAT	PS416, PS4-300
6	INSERT	PS416, PS4-300
7	DELETE	PS416, PS4-300
8	REPLACE	PS416, PS4-300
9	FIND	PS416, PS4-300

Table 30: Functions for time and date

No.	Name	Operation	PLC
1	ADD	TIME + TIME → TIME	x
2	ADD	TOD + TIME → TOD	x
3	ADD	DAT + TIME → DAT	PS416, PS4-300
4	SUB	TIME – TIME → TIME	x
5	SUB	DATE – DATE → TIME	PS416, PS4-300
6	SUB	TOD – TIME → TOD	x
7	SUB	TOD – TOD → TIME	x
8	SUB	DAT – TIME → DAT	PS416, PS4-300
9	SUB	DAT – DAT → TIME	PS416, PS4-300
10	MUL	TIME*ANY_NUM → TIME	PS416, PS4-300
11	DIV	TIME/ANY_NUM → TIME	PS416, PS4-300
12	CONCAT	CONCAT(DATE,TOD) → DAT	PS416, PS4-300
13	DT_TO_TOD		x
14	DT_TO_DATE		x

07/01 AWB2700-1306-GB

Table 31: Functions of enumerated data types

No.	Name	Symbol	PLC
1	SEL		–
2	MUX		–
3	EQ	=	x
4	NE	<>	x

Table 32: Access to function block parameters

No.	Access	Inside the FB	Outside the FB
1	Read input	x	x
2	Write input	x	x
3	Read output	x	x
4	Write output	x	Prohibited

Table 33: Function block declaration features

No.	Description	PLC
1a	RETAIN for internal variables	PS416, PS4-300
1b	NON_RETAIN for internal variables	–
2a	RETAIN for output parameters	PS416, PS4-300
2b	RETAIN for input parameters	–
2c	NON_RETAIN for output parameters	–
2d	NON_RETAIN for input parameters	–
3a	RETAIN for internal function blocks	PS416, PS4-300
3b	NON_RETAIN for internal function blocks	–
4a	VAR_IN_OUT declaration	x
4b	VAR_IN_OUT assignment in the graphical languages	x
4c	Linking of VAR_IN_OUT parameters with different variables (graphical)	–
5a	Assign function block instance at input (textual)	PS416, PS4-300
5b	Assign function block instance at input (graphical)	PS416, PS4-300
6a	Assign function block instance at VAR_IN_OUT input (textual)	PS416, PS4-300

No.	Description	PLC
6b	Assign function block instance at VAR_IN_OUT input (graphical)	PS416, PS4-300
7a	External function block instances (textual)	x
7b	External function block instances (graphical)	x
8a 8b	Input declaration with – rising edge – falling edge inputs	 PS4-200, PS4-300 PS4-200, PS4-300
9a 9b	Graphical input declaration with – rising edge – falling edge inputs	 x x
10a	Declaration of external variables	x
10b	Declaration of external write-protected variables (CONSTANT)	x
11	Declaration of temporary variables (VAR_TEMP)	–

Table 34: Standard bistable function blocks[1)]

No.	Name	PLC
1	SR	x
2	RS	x

Table 35: Standard edge detection function blocks

No.	Name	PLC
1	R_TRIG	x
2	F_TRIG	x

07/01 AWB2700-1306-GB

Table 36: Standard counter function blocks[1)]

No.	Name	PLC
1a	CTU	x
1b	CTU_DINT	–
1c	CTU_LINT	–
1d	CTD_UDINT	–
1e	CTD_ULINT	–
2a	CTD	x
2b	CTD_DINT	–
2c	CTD_LINT	–
2d	CTD_UDINT	–
2e	CTD_ULINT	–
3a	CTUD	x
3b	CTUD_DINT	–
3c	CTUD_LINT	–
3d	CTUD_ULINT	–

1) Input parameters such as R (Reset), S (Set), LD (Load) have already been defined as keywords. They cannot therefore be used as operators. The standard itself does not permit the use of keywords as identifiers. The original definitions of the standard counter modules contradict this requirement. The parameters have therefore been changed in Sucosoft:

   – S parameter in SET
   – R parameter in RESET
   – LD parameter in LOAD

   The modifications refer to a proposal of the PLCOpen committee.

Table 37: Standard timer function blocks

No.	Name	PLC
1	TP (Pulse)	x
	On-delay	
2a	– TON	x
2b	– T---0	x
	Off-delay	
3a	– TOF	x
3b	– 0---T	x
4	RTC (real-time clock)	x

Table 39: Program declaration features

No.	Description	PLC
1a	RETAIN for internal variables	x
1b	NON_RETAIN for internal variables	–
2a	RETAIN for output parameters	–
2b	RETAIN for input parameters	–
2c	NON_RETAIN for output parameters	–
2d	NON_RETAIN for input parameters	–
3a	RETAIN for internal function blocks	x
3b	NON_RETAIN for internal function blocks	–
4a	VAR_IN_OUT declaration	–
4b	VAR_IN_OUT assignment in the graphical languages	–
4c	Linking of VAR_IN_OUT parameters with different variables (graphical)	–
5a	Assign function block instance at input (textual)	–
5b	Assign function block instance at input (graphical)	–
6a	Assign function block instance at VAR_IN_OUT input (textual)	–
6b	Assign function block instance at VAR_IN_OUT input (graphical)	–
7a	External function block instances (textual)	–
7b	External function block instances (graphical)	–
	Input declaration with	
8a	– rising edge	–
8b	– falling edge inputs	–

07/01 AWB2700-1306-GB

No.	Description	PLC
	Graphical Input declaration with	
9a	– rising edge	–
9b	– falling edge inputs	–
10	Formal input and output parameters	–
11	Declaration of directly represented, non-retentive variables	x
12	Declaration of directly represented, retentive variables	x
13	Declaration of the memory location for symbolic variables	x
14	Declaration of the memory location for arrays	x
15	Initialization of directly represented, non-retentive variables	x
16	Initialization of directly represented, retentive variables	x
17	Declaration of the memory location and initial value for symbolic variables	x
18	Declaration of the memory location and initial value for arrays	x
19	Use of directly represented variables	x
20	VAR_GLOBAL	x
21	VAR_ACCESS	–
22a	VAR_EXTERNAL	–
22b	VAR_EXTERNAL CONSTANT	–
23	VAR_GLOBAL CONSTANT	x
24	VAR_TEMP	–

Language-specific elements

Table 51b: Support of parenthesis expressions in IL

No.	Description	PLC
1	Parenthesis express starts with explicit operator (long form)	–
2	Parenthesis expression starts with operand (short form)	x

Table 52: IL operators

No.	Operator	Modifier	PLC
1	LD	N	x
2	ST	N	x
3	S R		x x
4	AND	N,(	x
5	&	N,(	x
6	OR	N,(	x
7	XOR	N,(	x
7a	NOT		x
8	ADD	(	x
9	SUB	(	x
10	MUL	(	x
11	DIV	(	x
11a	MOD	(	PS416, PS4-300
12	GT	(	x
13	GE	(	x
14	EQ	(	x
15	NE	(	x
16	LE	(	x
17	LT	(	x
18	JMP	C, N	x
19	CAL	C, N	x
20	RET	C, N	x
21	)		x

07/01 AWB2700-1306-GB

Table 53: Supported call types for functions and function blocks in IL

No.	Description	PLC
1a	Call of function blocks without parameter names specified	-
1b	Call of function blocks with parameter names specified	x[1)]
2	Call of function blocks with explicit loading or saving of parameters	x
3	Use of inputs as operators	-
4	Call of functions with parameter names specified	-
5	Call of functions without parameter names specified	x

1) Outputs are programmed using a Sucosoft syntax

Table 55: ST operators

No.	Operation	Symbol	PLC
1	Parentheses	( )	x
2	Function call	name(arguments)	x
4	Two's complement	–	x
5	One's complement	NOT	x
3	Exponentiation	**	–
6	Multiplication	*	x
7	Division	/	x
8	Remainder formation	MOD	PS416, PS4-300
9	Addition	+	x
10	Subtraction	–	x
11	Greater/Less (Equal)	<, >, <=, >=	x
12	Equal	=	x
13	Not equal	<>	x
14, 15	Logical AND	&, AND	x
16	Logical EXCLUSIVE OR	XOR	x
17	Logical OR	OR	x

Table 56: ST language elements

No.	Description	PLC
1	Assignment	x
2	Call of function blocks and use of outputs	x
3	RETURN	x
4	IF	x
5	CASE	x
6	FOR	x
7	WHILE	x
8	REPEAT	x
9	EXIT	x
10	Empty instruction	x

Table 57: Representation of lines and blocks (graphical)

No.	Feature	PLC
2	Horizontal lines	x
4	Vertical lines	x
6	Horizontal/vertical connection	x
8	Line crossings without connection	–
10	Connected and not connected	x
12	Blocks with connecting lines	x
14	Continuation marks (connectors)	–[1)]

1) Not used since the right current rail is unlimited in Sucosoft.

Table 58: Graphical elements for execution control

No.	Description	PLC
1	Unconditional jump in FBD	x
2	Unconditional jump in LD	x
3	Conditional jump in FBD	x
4	Conditional jump in LD	x
5	Conditional return in LD	x
6	Conditional return in FBD	x

07/01 AWB2700-1306-GB

No.	Description	PLC
7	Unconditional return in FBD	x
8	Unconditional return in LD	x

Table 59: Power rails

No.	Description	PLC
1	Left power rail (with connected horizontal connection)	x
2	Right power rail (with connected horizontal connection)	x

Table 60: Link elements

No.	Description	PLC
1	Horizontal connection	x
2	Vertical connection	x

Table 61: Static contacts

No.	Symbol	Description	PLC
1	--| |--	Normally open contact	x
3	--|/|--	Normally closed contact	x
5	--|P|--	Positive transition sensing contact	–
7	--|N|--	Negative transition sensing contact	–

Table 62: Coils

No.	Symbol	Description	PLC
1	--( )--	Coil	x
2	--(/)--	Negation coil	x
3	--(S)--	SET coil	x
4	--(R)--	RESET coil	x
8	--(P)--	Positive transition sensing coil	–
9	--(N)--	Negative transition sensing coil6	–

D.1: Implementation-dependent parameters

Table D.1: Implementation-dependent parameters

Clause	Parameter	PS4-200	PS416	PS4-300
2.1.2	Maximum length of identifiers	64 characters for variable names 32 characters for POU names		
2.1.5	Maximum comment length	512		
2.1.6	Syntax and semantics of Pragma instructions	Pragma instructions are generally not supported		
2.2.2	Syntax and semantics of the double quote character in strings	The data type WSTRING is generally not supported. In STRING character strings the double quote characters have no meaning		
2.3.1	Supported value range for DATE	1900-1-1 to 2154-12-31		
	Supported value range for TIME	+63/–64 days, 23 hrs., 59 min., 59 sec., 999 ms	+127/–128 days, 23 hrs., 59 min., 59 sec., 999 ms.	+127/–128 days, 23 hrs., 59 min., 59 sec., 999 ms.
	Accuracy of representation of TOD and DT values	1 millisecond		
	Accuracy of representation of TIME values	1 millisecond		
2.3.3.1	Maximum number of array dimensions	3	4	4
	Maximum array size	8 KByte	16 KByte	16 KByte
	Maximum index range	0 to 255	–2147483648 to +2147483647	–2147483648 to +2147483647
	Maximum number of structure elements	100	100	100
	Maximum structure size	254 Bytes	16 KByte	16 KByte
	Maximum nesting depth for structures	50	50	50
	Maximum number of enumerated values	256	65000	65000
2.3.3.2	Default maximum length of STRING variable	32 Bytes		
	Maximum permissible length of STRING variable	253	1024	1024

07/01 AWB2700-1306-GB

Clause	Parameter	PS4-200	PS416	PS4-300
2.4.1.1	Maximum number of hierarchical levels	5		
	Logical or physical mapping	Both are supported		
2.4.2	Initialization of system inputs	The value of the system inputs corresponds to value of the physical inputs at the start of the program.		
2.4.3	Maximum number of variables per declaration block	The value is only restricted by the data memory available in the PLC (see Sucosoft limit values)		
	Use of AT% in function blocks	Syntax error		
	Warm start behaviour if a variable was not declared with RETAIN or NON_RETAIN	Program variables are not retentive in this case. A local FB variable is only retentive if it is part of a retentive function block instance. The special characteristics of markers are described in the chapter “Sucosoft S40 Navigator” in the "User Interface” manual (AWB2700-1305GB)		
2.5	Information to determine execution times of POUs	No support		
2.5.1.2	Value of outputs if ENO is FALSE	ENO not supported		
2.5.1.3	Maximum number of functions	No support of user functions	255	255
2.5.1.5	Maximum number of inputs of extensible functions	Comparison functions: max. 2 inputs MUX: max. 7 selection operands Otherwise any number		
2.5.1.5.1	Effects of type conversions on accuracy	none	none	none
	Error situations with type conversion	Non-representable values are restricted to the destination type without notification		
2.5.1.5.2	Accuracy of numeric functions	The numeric functions concerned are not yet supported		
2.5.1.5.6	Effects of a type conversion between ANY_DATE types and other types not contained in table 30	Syntax error: Other type conversions are not supported		
2.5.2	Maximum number of function blocks and instances	Restricted by the maximum program size, however, no more than 4000 instances per program.		
2.5.2.1a	Assignment at function block input if ENO is FALSE	ENO not supported		
2.5.2.3.3	PV_{min}, PV_{max} of counters	–32768 to 32767		

Clause	Parameter	PS4-200	PS416	PS4-300
2.5.2.3.4	Effect of a status change at a PT input during a TIMER operation	The value present at the PT input is only transferred when the TIMER function block is called with a rising edge at the IN input. If the function block is not called again and if there is no rising edge, the status change will have no effect.		
2.5.3	approx. 56 KByte	approx. 56 KByte	CPU 400: 1 MByte CPU 300: 512 KByte CPU 200: 256 KByte	512 KByte
3.3.1	Maximum length and nesting depth of an ST expression	Only restricted by the maximum stack load of 127 bytes	No restriction	No restriction
3.3.2	Maximum length and nesting depth of ST instructions	Length as required; Loops: max. 8 nesting levels CASE: max. 4 nesting levels IF: max. 8 nesting levels		
3.3.2.3	Maximum number of branches within a CASE instruction	Only restricted by the type of CASE expression		
3.3.2.4	Status loop variable after exiting a FOR loop	a) Start status if loop not completed. b) End status plus step width if the loop was completed at least once, no exit using EXIT instruction and no direct manipulation on the loop variable.		
4.1.1	Network topology restrictions	Max. 32767 networks		
4.1.3	Execution order of “Feedback” loops	Explicit loops are not supported. With implicit loops the evaluation is run from left to right and from top to bottom, i.e. the “Feedback” variable is normally accessed first. A status change therefore does not have an effect until the next cycle.		

07/01 AWB2700-1306-GB

Reaction to errors

Table E.1: Reaction to errors

Clause	Error situation	Sucosoft behaviour
2.1.5	Nested comments	Nested comments are explicitly supported
2.3.3.1	Non-typed enumerator value cannot be uniquely assigned to a type	The declaration of an enumerator type must not cause a name conflict with other definitions. The error situation can therefore not occur.
	Status of a variable outside of the restricted value range	Runtime correction: During a write access there is an automatic correction to the lower and upper range limit
2.4.1.1	The complete specification in CONFIGURATION is missing in an incomplete address	Syntax error: Incomplete addresses are not supported
2.4.2	Length of the initialization list does not match the number of array entries	Syntax error
2.4.3	Write access to a CONSTANT variable	Syntax error if the access is carried out in the same POU. Otherwise, error message during program generation.
	Declaration of a VAR_GLOBAL CONSTANT variable declared as VAR_EXTERNAL in the function block without a CONSTANT	Permitted if there is no write access in the function block
2.5.1	Incorrect use of directly addressed or global variables within functions	Syntax error: Use is never supported
2.5.1.1	Incorrectly programmed VAR_IN_OUT parameter	Syntax error
2.5.1.5.1	Type conversion errors	No error message; non-representable values are restricted to the destination type
2.5.1.5.2	Result of a numeric operation outside of the range of the result type	No error message; overflow can be scanned during runtime
	Division by zero	No error message; overflow can be scanned during runtime
2.5.1.5.3	Call of a bit shift function with a negative value on input “N”	Function executed with the two's complement of the value
2.5.1.5.4	Call of a selection function with different types of selection operands	Syntax error: Type conflict
	Call of the MUX function with an impermissible value on input “K”	The function returns the last selection operand

Clause	Error situation	Sucosoft behaviour
2.5.1.5.5	Invalid character position	Runtime correction: String returned unchanged
	Result exceeds maximum length for strings	Runtime correction: Result truncated to maximum length
	Call of a STRING function with a negative value on input ANY_INT	Function executed with the two's complement of the value
2.5.1.5.6	Result of a time operation outside of the range of the result type	No error message; overflow can be scanned during runtime
2.5.2.2	Function block type VAR_INPUT parameter not programmed	No error message; the last assigned instance is used. If the parameter was never connected, the behaviour of the called function block is undefined.
	VAR_IN_OUT parameter not programmed	Syntax error: VAR_IN_OUT must be programmed with every call.
3.2.2	Result of a numeric operation outside of the range of the result type	No error message; overflow can be scanned during runtime
	CR type and data type of operand not compatible with each other	Syntax error: Type conflict
3.3.1	Division by zero	No error message; overflow can be scanned during runtime
	Result of a numeric operation outside of the range of the result type	No error message; overflow can be scanned during runtime
	Operand type not permissible for the operation	Syntax error: Type conflict
3.3.2.1	Return from a function without function result set	Warning with the syntax check: the function returns the default initial value of the function type.
3.3.2.4	Repeat instruction not terminated	Runtime error: Cycle time violation
4.1.1	Name of a connector already used for other purposes	Connectors are not generally supported
4.1.3	Non-initialised “Feedback” variable	All variables are pre-initialised according to their type. VAR_TEMP not supported by S40.

07/01 AWB2700-1306-GB

Appendix

Sucosoft conventions

Keywords

Reserved keywords in Sucosoft
Reserved keywords are character combinations of the IEC 1131-3 Standard or Sucosoft which have a defined meaning. These keywords must not be used as designators or identifiers, i.e. as variable names, jump labels or POU names.

Individual special characters like the semicolon or dollar sign must not be used in identifiers either.

List of reserved keywords

*_TO_*
**_TO_BCD

ABS
ACOS
ACTION
ADD
ADDRESS
AND
ANDN
ANY
ANY_ARRAY
ANY_BIT
ANY_DATE
ANY_DERIVED
ANY_ELEMENTARY
ANY_FUNCTION_BLOCK
ANY_INT
ANY_MAGNITUDE
ANY_NUM
ANY_REAL
ANY_STRING
ARRAY
ASIN
AT
ATAN

BCD_TO_**
BOOL
BOOL_EXP
BY
BYTE
BYTE_EXP

CAL
CALC
CALCN
CASE
CONCAT
CONFIGURATION
CONSTANT
COS

DATE
DATE_AND_TIME
***_TO_TIME_OF_DAY
***_TO_DATE
DELETE
DINT
DINT_EXP
DIV
DO
DT
DT_TO_DATE
DT_TO_TOD
DWORD
DWORD_EXP

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
EQ
EXIT
EXP
EXPT

F_EDGE
FALSE
FIND
FOR
FROM
FUNCTION
FUNCTION_BLOCK

GE
GETCARRY
REGISTTER
GT

IF
INITIAL_STEP
INSERT
INT
INT_EXP
INTERVAL
ISCARRY
ISZERO
ISOVERFLOW
ISPOSITIVE
ISBITTRUE

JMP
JMPC
JMPCN

LD
LDN
LE
LEFT
LEN
LIMIT
LINT
LINT_EXP
LN
LREAL
LOG
LT
LWORD

MAX
MID
MIN
MOD
MOVE
MUL
MUX
NE
NEG
NON_RETAIN
NOT

OF
ON
OR
ORN

PRIORITY
PROGRAM

R
R_EDGE
READ_ONLY
READ_WRITE
REAL
REPEAT
REPLACE
RESOURCE
RET
RETAIN
RETC
RETCN
RETURN
RIGHT
ROL
ROR

S
SEL
SHIFTLEFTCARRY
SHIFTRIGHTCARRY
SHL
SHR
SIN
SINGLE
SINT
SINT_EXP
SQRT
ST
STEP
STN
STRING
STRUCT
SUB

TAN
TASK
THEN
TIME
TIME_TO_USINT
TIME_TO_UINT
TIME_OF_DAY
TO
TOD
TRANSITION
TRUE
TRUNC
TYPE

UDINT
UDINT_EXP
UINT
UINT_EXP
ULINT
ULINT_EXP
UNTIL
USERTYPE
USINT
USINT_EXP

VAR
VAR_ACCESS
VAR_CONFIG
VAR_EXTERNAL
VAR_GLOBAL
VAR_IN_OUT
VAR_INPUT
VAR_OUTPUT
VAR_TEMP

WHILE
WITH
WORD
WORD_EXP
WSTRING
XOR
XORN

* all combinations of ANY_NUM, ANY_BIT, ANY_STRING; ** all combinations of ANY_INT; *** DATE_AND_TIME

07/01 AWB2700-1306-GB

Sucosoft limit values

	PS4-200	PS4-300	PS416
Project management restrictions			
Max. number of POUs/Program	256	256	256
Max. length of project names. Project names must be different within the first eight characters.	32 characters	32 characters	32 characters
Editor restrictions			
Max. number of lines in a POU	64 KByte	64 KByte	64 KByte
Max. length of a variable name	64 characters	64 characters	64 characters
Max. length of a POU name	32 characters	32 characters	32 characters
Status display restrictions			
Max. number of simultaneously displayable variables	approx. 50[1)]	–	–
Restrictions of the Topology Configurator			
Max. number of components per configuration [2)]	175	630	753
Max. number of simultaneously selectable components in the I/O monitor	10	10	10
Restrictions due to the CPU and the compiler			
Max. program size including all FBs (Code + Data)	63 KByte	512 KByte	CPU 200: 256 KB CPU 300: 512 KB CPU 400: 1 MB
Max. code size of a POU in the controller	56 KByte	64 KByte	64 KByte
Max. size of the total of all variables in the main program and all function blocks	32000 global 32000/POU local	16000 global 16 kB/POU local	16000 global 16 kB/POU local
Max. depth of the instance tree (max. number of nested instances)	8	16	16
Max. depth of the function block invocation tree	16	16	16
Max. number of instances per program[2)]	4000	4000	4000
Max. number of AT declarations [2)]	16 KByte	16 KByte	5450
Max. size of an array	8 KByte	16 KByte	16 KByte
Max. number of array dimensions	3	4	4
Range of values of an array index	0 to 255	–2147483648 to +2147483647	
Max. number of enumerations per type	256	65000	65000
Max. length of a string	253 characters	1024 characters	1024 characters

	PS4-200	PS4-300	PS416
Max. number of global type definitions	1100	1100	1100
Permissible range of values of the data type TIME	+63/–64 days 23 hrs., 59 min. 59 secs., 999 ms	+127/–128 days, 23 hrs., 59 min. 59 secs., 999 ms	
Permissible range of values of the data type DATE	1900-1-1 to 2154-12-31		
Max. size of an S30 program which can be converted and loaded into the PLC	54 KByte	no limit	
Max. definable cycle time	255 ms	1000 ms	1000 ms
Recommended program code size of a POU for efficient on-line processing	< 2 KByte	< 8 KByte	< 8 KByte

1) Depending on data type. If the limit value is exceeded, "?" will be inserted instead of values.
2) Restricted also by the PLC main memory available.

Program examples LD/FBD

The programming examples presented here are part of complex control tasks. The used manufacturer functions and function blocks have been combined into useful processes. The variable declarations only contain the variables used in the programming examples.

Programming examples of functions

Task 1
The rotation to right of a robot arm is to be halted when the end position is reached or the Halt button is pressed.

The following variables are defined:

Value of the variable Actual_Position. The value is the same as the counter status that is supplied via a pulse generator.

Value of the variable Reference_Point.

07/01 AWB2700-1306-GB

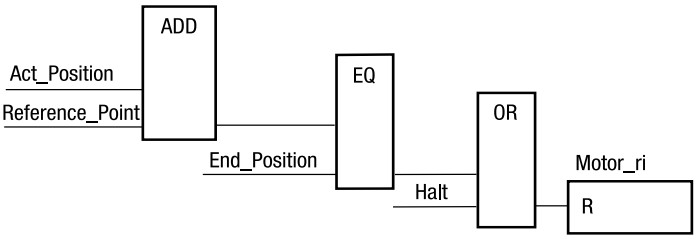
Value of the variable End_Position.

```
VAR

  Act_Position : INT;
  Reference_Point : INT;
  End_Position : INT;
  Halt AT %I0.0.0.0.0 : BOOL;
  Motor_re AT %Q0.0.0.0.0 : BOOL;

END_VAR
```

0003



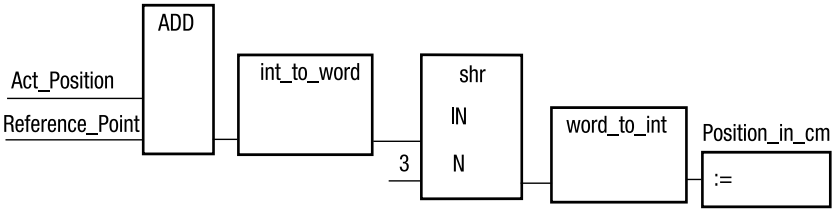
Task 2

The variable Act_Position in task 1 shows the distance of the robot arm from the reference point as a pulse number. Eight pulses are equivalent to a 1 cm movement. In network 0007 the absolute position is to be output in centimetres. The SHR shift function is used for converting the pulse number into centimetres.

Additional declaration:

```
VAR
  Position_in_cm : INT;
END_VAR
```

0007



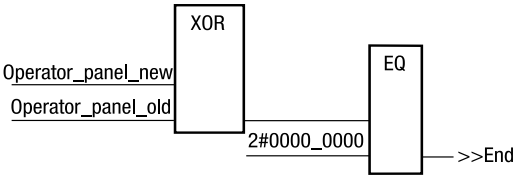
Task 3

A part of the operating elements of an operator panel is connected with the IB0.0.0.0 input byte. The program section where the operating elements are evaluated should only be executed if there is a different status compared with the previous program cycle in the operator panel.

```
VAR
  Operator_panel_new AT %IB0.0.0.0 : BYTE;
  Operator_panel_old : BYTE;
END_VAR
```

.
.
.

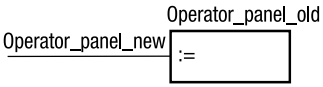
0004 Manual operation



07/01 AWB2700-1306-GB

0005

Updating the bit pattern



.
.
.

0020 End

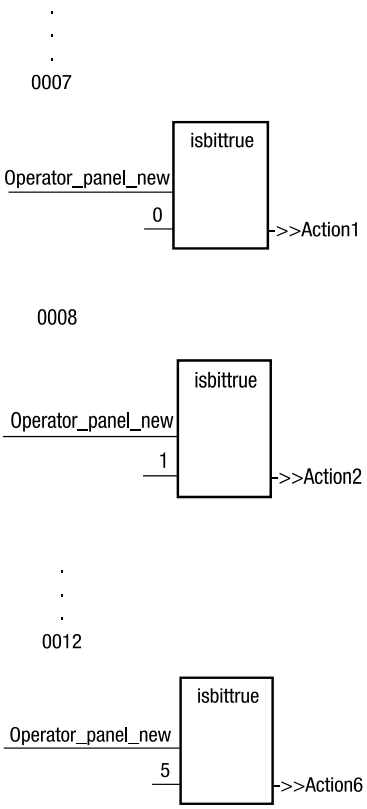
Task 4

The robot arm is moved into the Hand operating mode via bits 0 to 5 of the operating elements listed in task 3. In this operating mode only one single process activity may be executed by a short key stroke. Variable declaration will be completed. The program is continued from network 0007 task 3.

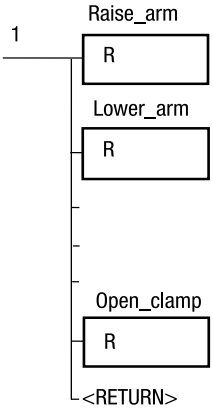
Additional declaration:

```
VAR
  Automatic AT %I0.0.0.1.6 : BOOL;
  Manual AT %I0.0.0.1.7 : BOOL;
  Raise_Arm AT %Q0.0.0.1.0 : BOOL;
  Lower_Arm AT %Q0.0.0.1.1 : BOOL;
  Move_Arm_ri AT %Q0.0.0.1.2 : BOOL;
  Move_Arm_le AT %Q0.0.0.1.3 : BOOL;
  Close_Clamp AT %Q0.0.0.1.4 : BOOL;
  Open_Clamp AT %Q0.0.0.1.5 : BOOL;

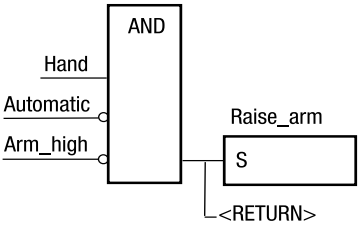
  Arm_top AT %I0.0.0.1.0 : BOOL;
  Arm_bottom AT %I0.0.0.1.1 : BOOL;
  Arm_right AT %I0.0.0.1.2 : BOOL;
  Arm_left AT %I0.0.0.1.3 : BOOL;
  Close_Clamp AT %I0.0.0.1.4 : BOOL;
  Open_Clamp AT %I0.0.0.1.5 : BOOL;
END_VAR
```


0007
isbittrue
Operator_panel_new
0
->>Action1
0008
isbittrue
Operator_panel_new
1
->>Action2
0012
isbittrue
Operator_panel_new
5
->>Action6

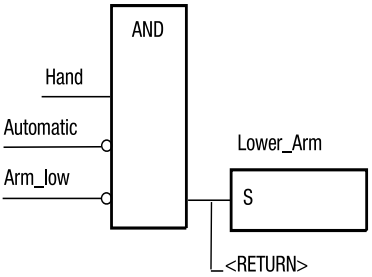
0013



0014 Action1

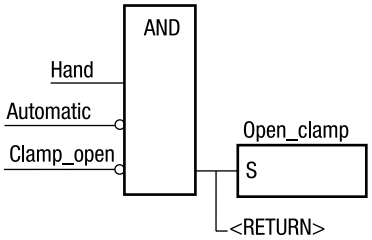


0015 Action2



.
.
.

0019 Action6



Function blocks

A single function block can be programmed in a network in the LD/FBD editor. There are two ways of transferring the parameters to the function block:

- Entering the variable names or values at the individual connection lines directly when calling the function block where only the parameters for the relevant operands must be set.

07/01 AWB2700-1306-GB

Assigning the variable names or values to the input operands in the networks before calling the function block and sequencing the output operands in networks after calling the function block.

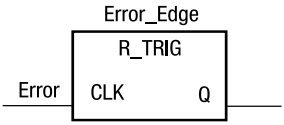
Programming examples of function blocks

Task 5

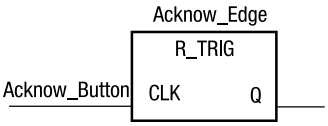
An error recognised by the monitoring program is indicated by an acoustic signal and a 1-Hz flashing light. An acknowledge button switches off the acoustic signal immediately but not the flashing light unless the error message is no longer active.

```
VAR
  Error_Edge : R_TRIG ;
  Ackn_Edge : R_TRIG ;
  Klaxon_ON_OFF : SR ;
  ClockEnable : SR ;
  Clock : TimeGenerator ;
  Error : BOOL ;
  AcknowledgeButton AT %I0.0.0.0.1 : BOOL ;
  Klaxon AT %Q0.0.0.0.0 : BOOL ;
  FlashingLight AT %Q0.0.0.0.1 :BOOL ;
  Edge : BOOL ;
  Clock_OFF : BOOL ;
  Flash : BOOL ;
END_VAR
```

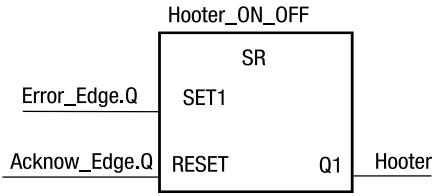
0001



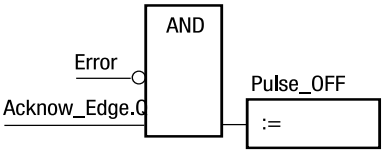
0002



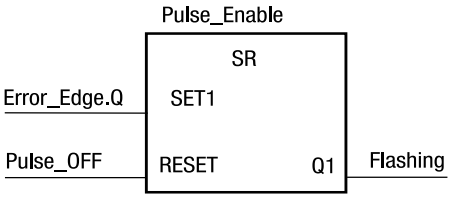
0003



0004

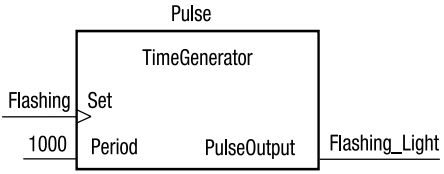


0005



07/01 AWB2700-1306-GB

0006



Task 6

A conveyor belt is controlled by a motor at the drive shaft. There are proximity switches at the two drive shafts which provide at least 5 pulses per second when the belt is running at the required speed.

A monitoring program outputs an error message if

the number of revolutions is too low after a startup time of 8 seconds

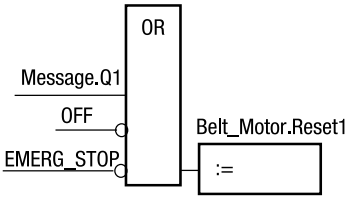
or

the belt is broken.

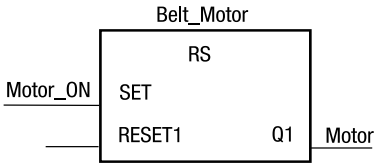
With an error message the motor is switched off and a fault indication is activated.

```
VAR
  Beltmotor : RS ;
  Signal : SR ;
  Speed : TOF ;
  BeltBreak : CTUD ;
  StartupTime : TP ;
  Ackn_Edge : R_TRIG ;
  Motor_ON AT %I0.0.0.0.0 : BOOL ;
  OFF AT %I0.0.0.0.1 : BOOL ;
  EMERG_OFF AT %I0.0.0.0.2 : BOOL ;
  Lamp_Ackn AT %I0.0.0.0.3 : BOOL ;
  NS_Drive AT %I0.0.0.0.4 : BOOL ;
  NS_2 AT %I0.0.0.0.5 : BOOL ;
  Lamp AT %Q0.0.0.0.0 : BOOL ;
  Motor AT %Q0.0.0.0.1 : BOOL ;
END_VAR
```

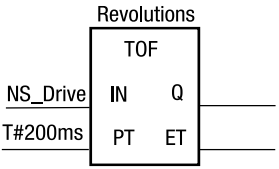
0001



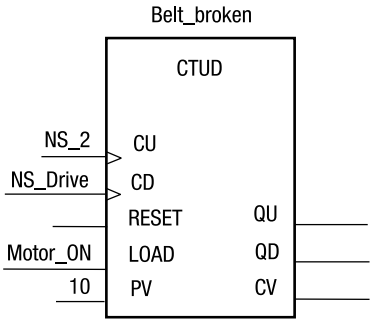
0002



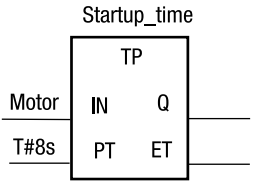
0003



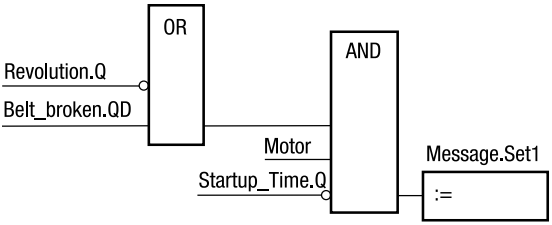
07/01 AWB2700-1306-GB



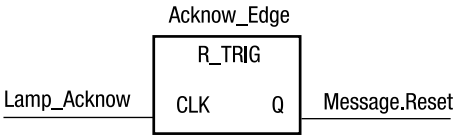
0005



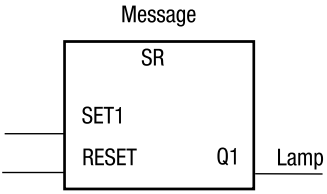
0006



0007



0008

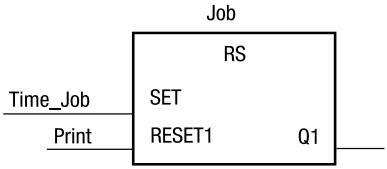


Task 7

An installation report can be created and printed out. A manually initiated protocol job is executed immediately or within 24 hours. With a time-dependent job the time is set via a 4-digit decade switch which is used to set hours and minutes in decimal form.

```
VAR
  Clock : PS_GetDateAndTime ;
  Job : RS ;
  Edge_1 : R_TRIG ;
  Edge_2 : R_TRIG ;
  TimeValue : ELEMENT_OF_DATE ;
  Hours AT %IB0.0.0.0 : BYTE ;
  Minutes AT %IB0.0.0.1 : BYTE ;
  Time_Job AT %I0.0.0.2.0 : BOOL ;
  Immediate_Job AT %I0.0.0.2.1 : BOOL ;
  Printing : BOOL ;
END_VAR
```

0001

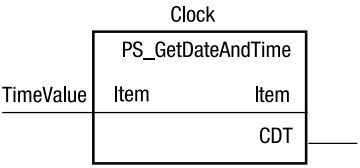


07/01 AWB2700-1306-GB

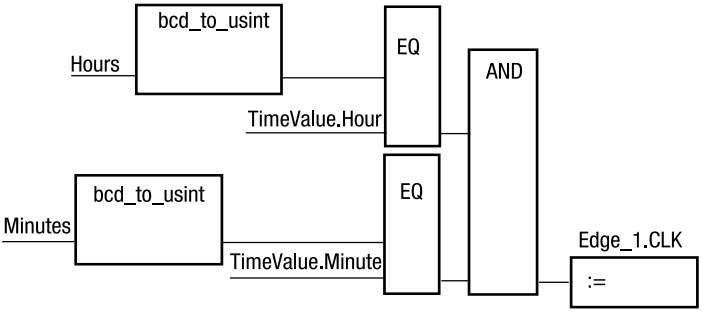
0002



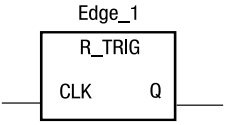
0003



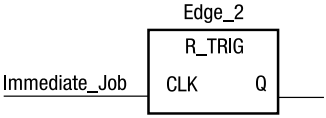
0004



0005

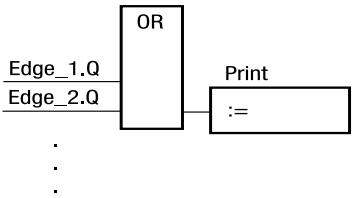


0006 GoOn



0007

The print job in form of a pulse causes the protocol to be output



Task 8

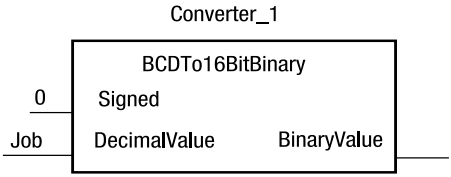
In a paint preparation plant paints are mixed according to different recipes from various basic elements and the prepared paint is poured into cans of different sizes. Selection of an ordered recipe and a type of can is carried out by setting the corresponding 4-digit order number on a decade switch. After a button is pushed as confirmation the job goes into a queue which can manage up to 10 jobs.

07/01 AWB2700-1306-GB

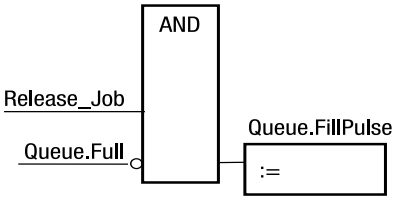
If the plant is in status Ready and the jobs are stored in the queue, the first job is read automatically and the process is started.

```
VAR
  Converter_1:BCDTo16BitBinary;
  Converter_2:_16BitBinaryToBCD;
  Queue:FifoW10;
  Job AT %IW0.0.0.0:UINT;
  Release_Job AT %I0.0.1.0.0:BOOL;
  Delete_Jobs AT %I0.0.1.0.1:BOOL;
  Display_Current_Job AT%QW0.0.3.0:UINT;
  Display_Input_Disable AT %Q0.0.2.0.0:BOOL;
  System_Ready:BOOL;
END_VAR
```

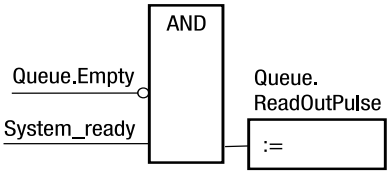
0001



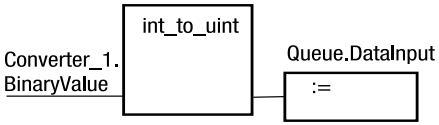
0002



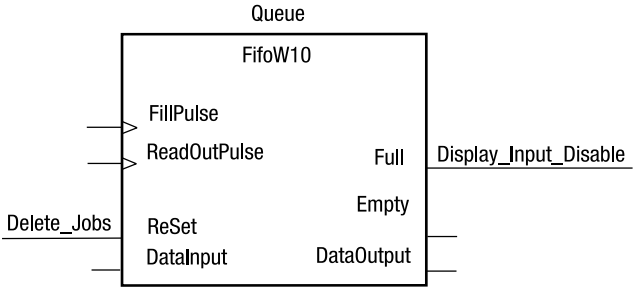
0003



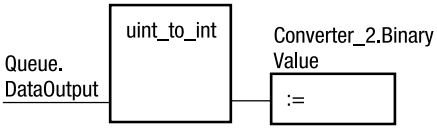
0004



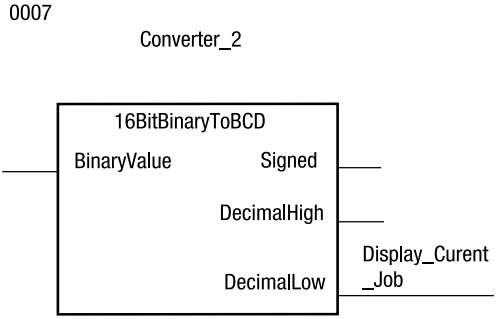
0005



0006



07/01 AWB2700-1306-GB


0007
Converter_2
16BitBinaryToBCD
BinaryValue
Signed
DecimalHigh
DecimalLow
Display_Curent
_Job

07/01 AWB2700-1306-GB

Index

Symbols

& 49
&N 51
*_TO_** 132
_16Bit_Compare 228
_16BitBinaryToBCD 230
_16BitCounter 232
_32Bit_Compare 234
_32BitBinaryToBCD 236
_32BitCounter 238
“#” character 221

A

ABS 154
ADD
  Addresses 159
  Numeric 56, 155
  Time values 157
Addition 56, 117
ADDRESS_TO_UINT 153
ADRtoSTR 240, 242
Alarm
  Disabling 165
  Enabling 171
Alarm Function Block
  Counter 271
Alarm function block
  Edge-controlled 326
  TimerAlarm 554
Alarm function blocks, Overview 223
Alternative statement 78
AND 49
AND sequence
  in FBD 110
  in IL 49
  in LD 97
ANDN 51, 111
Arithmetic functions, Overview 128
Arithmetic operators 56

Array operations 224
Assignments 45
AT 28, 30

B

BCD_TO_DINT 151
BCD_TO_INT 151
BCD_TO_SINT 151
BCD_TO_UDINT 151
BCD_TO_UINT 151
BCD_TO_USINT 151
BCDTo16BitBinary 245
BCDTo32BitBinary 247
Bistable data elements 46
Bistable elements 521
Bit string functions
  Rotation functions 208, 209
  Shift functions 212, 213, 214, 215
BlockCompare 249
BlockTransfer 254
BOOL_TO_BYTE 142
BOOL_TO_DINT 139
BOOL_TO_DWORD 142
BOOL_TO_INT 139
BOOL_TO_SINT 139
BOOL_TO_UDINT 139
BOOL_TO_UINT 139
BOOL_TO_USINT 139
BOOL_TO_WORD 142
BYTE_TO_BOOL 142
BYTE_TO_DINT 139
BYTE_TO_DWORD 142
BYTE_TO_INT 139
BYTE_TO_SINT 139
BYTE_TO_UDINT 139
BYTE_TO_UINT 139
BYTE_TO_USINT 139
BYTE_TO_WORD 142

C

CALC 66
CALCN 66
CASE statement 80

07/01 AWB2700-1306-GB

Character string functions
Functions (Overview) 129
Character string processing
Function blocks (Overview) 224
Check battery via flags 406
ClearDiagnosticBits 258
Code converter 223
Cold Start Retention 260
ColdstartRetention 260
COM 264
Commands
Arithmetic operators 56
Assignments 45
Bistable data elements 46
Comparison operators 59
Conditional and unconditional jumps 61
Conditional and unconditional returns 63
FBD representation 108
Invocation of function blocks 65
Invocation of functions 66
LD representation 95
Load instructions 43
Logic operators 49
Comments
Notation 6
Communication function blocks 224
Compare
16-bit values 228
32-bit values 234
Compare operations 119
CompareArray 265
Comparison function blocks 223
Comparison operators 59
Complement
Bitwise 201
Two's 200
COMtoSTR 269
CONCAT 160, 162
Conditional statement, ST 78
CONSTANT 28, 30

Constants 28
Character string 34
Overview 33
Pre-defined 34
Conversion functions 128, 225
Convert binary to decimal 230, 236
Counter
Decrementing 283
Decrementing and incrementing 285
Incrementing 284
Counter function blocks 223
CounterAlarm 271
CounterLE 277
CPUDataExchange 279
Create RSDAT.DAT 488
CTD 283
CTU 284
CTUD 285
Current result
Assign 96, 115
Negate 100

D

Data types 11
Derived 14
Elementary 11
DataScale 287
DATconcat 291
Date and time, function block 222
DATE_AND_TIME_TO_ TIME_OF_DAY 146
DATE_AND_TIME_TO_DATE 146
DateConcat 293
DateSplit 295
DATsplit 297
DE4netDP 299
DE4netK 307
DELETE 163
Derived data types
Array 17
Declaration 14
Enumeration 16
Keywords 14

07/01 AWB2700-1306-GB

Structure 20
Value range restriction 15
Deserialize 314
Diagnostic bit "KOM" 404
DialOrHangup 317
DINT_TO_BCD 150
DINT_TO_BOOL 135
DINT_TO_DWORD 134
DINT_TO_INT 138
DINT_TO_REAL 143
DINT_TO_SINT 138
DINT_TO_UDINT 137
DINT_TO_UINT 138
DINT_TO_USINT 138
Direct reading of inputs
Analog inputs 424
Counter input 427
Digital inputs 430
Directly represented variables 21
Disable_Interrupt 165
DIV 168
Addresses 170
Duration 169
Division 58, 118
Remainder 176
DWORD_TO_BOOL 142
DWORD_TO_BYTE 142
DWORD_TO_DINT 139
DWORD_TO_INT 140
DWORD_TO_SINT 140
DWORD_TO_UDINT 139
DWORD_TO_UINT 140
DWORD_TO_USINT 140
DWORD_TO_WORD 142

E

Edge detection 128
Falling edge 332
Function block 222
Rising edge 423
EdgeAlarm 326
Elementary data types 11
ELSE branch 79

ELSIF branches 79
Empty statement 77
Enable_Interrupt 171
End value, with FOR loop 82
END_TYPE 26
END_VAR 26
EQ 59, 119
Equivalence function 100
Exclusive OR sequence
  in FBD 113
  in IL 54
  in LD 100
Execution condition (ST) 81

F
F_TRIG 332
FB
  Function block 5
FBD 91
FBD, programming in 108
FifoBx 334
FifoWx 338
FIND 174
FOR loop 82
Frequency inverter 307
FU 5
Function 5
  Application range 8
  Data type of operands 132
  Example 69
  Invocation in LD and FBD 105
  Overloaded 131
  Parameter transfer 69
  Prototype representation 130
Function block diagram 91
Function block, POU type 9
Function blocks
  Call 31
  Call in LD and FBD 106
  Call in ST 84
  Conditional call 66
  Direct parameter passing 588
  Example 71

07/01 AWB2700-1306-GB

Instance 10
Invocation in IL 65
Invocation in LD and FBD 124
Overview 127, 221
Parameter transfer 72, 106, 125, 628
Programming examples 629
Prototype representation 226
Retentive 30
Function, POU type 8
Functions
Call in ST 86
Invocation in IL 66
Invocation in LD and FBD 123
Overview 127, 221
Programming examples 622
Type conversion 132

G

GE 59, 119
GetCarryRegister 176
GetRealTimeClock 342
Global variable 26
Graphical programming languages 91
GT 59, 119

I

IEC/EN 61 131-3 3
IEEE_To_Real 343
IF construct 79
IL 37
IL instruction set 41
IL sequences, overview 39
IN_OUT variable 26
IncEncoder 344
Initial value, with FOR loop 82
Input variable 26
INSERT 177
Instance 10, 31
Instruction list 37
Instructions
Sequences 39
Structure 37
INT_TO_BCD 150

INT_TO_BOOL 135
INT_TO_BYTE 134
INT_TO_DINT 137
INT_TO_DWORD 134
INT_TO_REAL 143
INT_TO_SINT 138
INT_TO_UDINT 137
INT_TO_UINT 137
INT_TO_USINT 138
INT_TO_WORD 134
Intermediate result
  Negate 114
Interrupt
  Disabling 165
  Enabling 171
Interrupt functions 129
IsBitTrue 179
IsCarry 180
IsOverflow 181
IsPositive 182
IsZero 183

J
JMP 61, 103, 121
JMPC 62, 104, 122
JMPCN 62, 104, 122
Jump
  Conditional 62, 104, 122
  Return 63, 104
  Unconditional 61, 103, 121
Jump commands
  in FBD 120
  in IL 61
  in LD 102
Jump destination 102, 120
Jump label 40, 102

K
Keywords 619
  Of variable types 26, 28
  Reserved 619

L
Label 39, 40
Ladder diagram 91
LD 43, 91
  Programming in 95
LDN 44
LE 59, 119
LEFT 184
LEN 185
LifoBx 348
LifoWx 352
LIMIT 186
Load instructions 43
Local variable 26
Logic operators 49
Logic sequences 97, 110, 129
Loop
  For 82
  Repeat 81
  While 81
Loop exit 83
Loop variable 82
LT 59, 119

M
Marker 23
MAX 188
Memory card
  Initialising 488
  Readout 443
  Save data 456, 582
MID 190
MIN 192
Mixed sequences 98
MOD 200 360
Modem
  Establish data connection 317
  Sending data to 478
MS_TimeFalling 361
MS_TimeRising 364
MUL
  Addresses 197
  Multiplication of numeric data types 194

Numeric 57
Time duration 196
Multiple instantiation 331
Function block 10
Multiple selection 80
Multiplexer 198
Multiplication 57, 118
Carry 176
MUX 198

N

NE 59, 119
NEG 200
Negation symbol
in FBD 110
in LD 100
Networks (program sequence in LD/FBD) 92
NItoCelsius 367
NItoFahrenheit 369
NOT 114, 201
Numeric 58
Numerical functions 128

O

Operand 37
Operators
Arithmetic operators 56
Assignments 45
Bistabile data elements 46
Comparison operators 59
Conditional and unconditional jumps 61
Conditional and unconditional returns 63
Invocation of function blocks 65
Invocation of functions 66
Load instructions 43
Logic operators 49
Overview 41
OR 52
OR sequence
in FBD 112
in IL 52
in LD 98
ORN 53

07/01 AWB2700-1306-GB

OS system functions 129, 225
Output symbols 96, 115
Output variable 26, 96, 116
Outputs, writing direct to 575, 578
Overloaded function 131

P

Parameter assignments 84
Parameter list 86
Parameter passing, direct 588
Parenthesization 89
Passing parameter 26
PdpFreezeSync 384
PdpStationDiag 398
Peripheral access, function blocks 225
PLC_Message 202
PLC_ReadClock 204
POU Editor 91
POU type
  Function 8
  Function block 9
  Program 7
Precedence rules 89
Pre-defined constants 34
PROFIBUS diagnostic data 398
Program 7
Program organisation units
  Comments 6
  Structure 5
  Variables declaration 5
Program sequences 92
Programming examples LD/FBD 622
Programming language
  Convert 91
  FBD 108
  IL 37
  LD 95
  ST 75
Programming rules 39
PS_ApplicationHalt 402
PS_ClearKOMBit 404
PS_Diagnostic 406
PS_GetDateAndTime 408

PS_Message 410
PS_ReadOSInfo 412
PS_SwitchPosition
  Operating mode selector switch
    Read switch position 414
PSCO 416
PTtoCelsius 419
PTtoFahrenheit 421

Q
Queue 334, 338

R
R 48
R_TRIG 423
Read diagnostic flags 406
Read operating system information 412
Read out date 204
Read out time 204
ReadAnalogDirect 424
ReadCounterDirect 427
ReadDirect 430
Reading and writing the memory card,
  function blocks 225
Reading diagnostic flags 23
Reading inputs directly
  Analog inputs 424
  Counter input 427
  Digital inputs 430
ReadMC
  Memory card
    Readout 433
REAL _TO_ SINT 144
REAL_TO_DINT 144
REAL_TO_IEEE 437
REAL_TO_INT 144
REAL_TO_UDINT 144
REAL_TO_UINT 144
REAL_TO_USINT 144
Real-time clock
  Compare 438
  Evaluate 342

07/01 AWB2700-1306-GB

Read 408
Set 491
RealTimeClock 438
Register function blocks 223
ReloadData 443
REPEAT loop 81
REPLACE 205
Reserved keywords 619
Reset 48, 101, 117
RET 63, 104, 122
RETAIN 28
RETC 64, 104, 123
RETCN 64, 105, 123
Retentive variables 28
Return 63
Conditional 104, 123
Unconditional 104, 122
RIGHT 207
ROL 208
ROR 209
Rotation functions 208, 209
RS 446
RTC 448

S

S 47
S_TimeFalling 450
S_TimeRising 453
SaveData 456
SCO 459
SEL 210
Selection
Input operand 198, 210
Limiting values 186
Maximum 188
Minimum 192
Selection functions 130
SendATCommand 478
Sending AT commands 478
Sequence control function block 494
Sequences
in FBD 113
in LD 97

Sequential control function block 224
Serial communication 459
Serialize 485
Set 47, 101, 116
SetMCFileLength 488
SetRealTimeClock 491
Sfc_x 494
Shift functions 212, 213, 214, 215
ShiftLeftCarry 212, 213
SHL 214
SHR 215
Single-bit access 587
SINT_TO_BCD 150
SINT_TO_BOOL 135
SINT_TO_DINT 137
SINT_TO_DWORD 134
SINT_TO_INT 137
SINT_TO_REAL 143
SINT_TO_UDINT 137
SINT_TO_UINT 137
SINT_TO_USINT 137
SINT_TO_WORD 134
SR 521
SR_x 523
SRB_x 527
SRW_x 531
ST 45
ST value assignment 77
ST, programming in 75
Stack register 348, 352
Standard function blocks 127, 221
Standard functions 127, 221
Statement, conditional (ST) 78
Status information 23
Step width, with FOR loop 82
STN 46
String
  Concatenating 160
  Copy to array 540
  Copy to marker range 538
  Deleting characters from 163
  Determining the length 185
  Finding 174

07/01 AWB2700-1306-GB

Inserting a further 177
Removing a sequence of characters from 190
Replacing characters in a 205
Selecting characters from left 184
Selection of characters from right of 207
STRtoADR 538
STRtoCOM 540
STRUCT 18
Structure 18
Structured text, programming in 75
SUB 216
Addresses 219
Numeric 57
Time values 217
Subtraction 57, 118
Summation 154
System functions, OS 129

T

Termination condition (ST) 81
Textual programming languages 37
TIME_TO_UINT 146
TIME_TO_USINT 146
TimeConcat 548
TimeGenerator 550
TimePulse 552
Timer
Off-delay 361
On-delay 364
Timer function blocks 222
TimerAlarm 554
TimeSplit 561
TODconcat 563
TODsplit 565
TOF 567
TON 569
TP 571
TransferArray 573
Transition 498
TRUNC 220
TYPE 26
Type conversion functions 132

U

UDINT_TO_BCD 150
UDINT_TO_BOOL 135
UDINT_TO_BYTE 134
UDINT_TO_DWORD 134, 137
UDINT_TO_INT 138
UDINT_TO_REAL 143
UDINT_TO_SINT 138
UDINT_TO_USINT 138
UDINT_TO_WORD 134
UINT_TO_BCD 150
UINT_TO_BOOL 135
UINT_TO_BYTE 134
UINT_TO_DINT 137
UINT_TO_DWORD 134
UINT_TO_INT 137
UINT_TO_REAL 143
UINT_TO_SINT 138
UINT_TO_UDINT 137
UINT_TO_USINT 138
UINT_TO_WORD 134
USINT_TO_BCD 150
USINT_TO_BYTE 134, 135
USINT_TO_DINT 137
USINT_TO_DWORD 134
USINT_TO_INT 137
USINT_TO_REAL 143
USINT_TO_SINT 137
USINT_TO_UDINT 137
USINT_TO_UINT 137, 138
USINT_TO_WORD 134

V

Value assignment, ST 77
Value list 80
VAR 26
VAR_EXTERNAL 26
VAR_GLOBAL 26
VAR_IN_OUT 26
VAR_INPUT 26
VAR_OUTPUT 26
Variable types
  Overview, use in POUs 31

07/01 AWB2700-1306-GB

Variables 20, 97, 109
Constant 30
Declaration 25
Directly represented 21
Retentive 28
Types, Overview 26, 28
Version number 412

W
WHILE loop 81
WORD_TO_BOOL 142
WORD_TO_BYTE 142
WORD_TO_DINT 139
WORD_TO_DWORD 142
WORD_TO_INT 139
WORD_TO_SINT 140
WORD_TO_UDINT 139
WORD_TO_UINT 139
WORD_TO_USINT 140
WriteAnalogDirect 575
WriteDirect 578
WriteMC 582
Writing direct
to analog outputs 575
to digital outputs 578

X
XOR 54, 113
XORN 55

07/01 AWB2700-1306-GB